HCL OneDB 2.0.1

OneDB Performance Guide

Contents

Chapter 1. Performance Guide

The *HCL OneDB™ Performance Guide* describes how to configure and operate your HCL OneDB™ database server to improve overall system throughput and to improve the performance of SQL queries.

This information contains performance tuning issues and methods that are relevant to daily database server administration and query execution. Performance measurement and tuning encompass a broad area of research and practice and can involve information beyond the scope of this publication.

This information is intended for the following users:

- Database administrators
- Database server administrators
- Database-application programmers
- Performance engineers

This information assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with database server administration, operating-system administration, or network administration

Information in these topics can help you perform the following tasks:

- Monitor the system resources that are critical to performance
- Identify database activities that affect these critical resources
- Identify and monitor queries that are critical to performance
- Use the database server utilities (especially **onperf** and **onstat**) for performance monitoring and tuning
- Use the database server utilities (especially **onstat**) for performance monitoring and tuning
- Eliminate performance bottlenecks by:
    - Balancing the load on system resources
    - Adjusting the configuration parameters or environment variables of your database server
    - Adjusting the arrangement of your data
    - Allocating resources for decision-support queries
    - Creating indexes to speed up retrieval of your data

Performance measurement and tuning encompass a broad area of research and practice and can involve information beyond the scope of these topics.

Performance basics

Performance measurement and tuning issues and methods are relevant to daily database server administration and query execution.

These topics:

- Describe a basic approach for performance measurement and tuning
- Provide guidelines for a quick start to obtain acceptable initial performance on a small database
- Describe roles in maintaining good performance

Developing a basic approach to performance measurement and tuning

To maintain optimum performance for your database applications, develop a plan for measuring system performance, making adjustments to maintain good performance and taking corrective measures when performance degrades. Regular, specific measurements can help you to anticipate and correct performance problems.

About this task

By recognizing problems early, you can prevent them from affecting users significantly. Early indications of a performance problem are often vague; users might report that the system seems sluggish. Users might complain that they cannot get all their work done, that transactions take too long to complete, that queries take too long to process, or that the application slows down at certain times during the day.

To determine the nature of the problem, you must measure the actual use of system resources and evaluate the results.

Users typically report performance problems in the following situations:

- Response times for transactions or queries take longer than expected.
- Transaction throughput is insufficient to complete the required workload.
- Transaction throughput decreases.

An iterative approach to optimizing database server performance is recommended. If repeating the steps found in the following list does not produce the desired improvement, insufficient hardware resources or inefficient code in one or more client applications might be causing the problem.

To optimize performance:

1. Establish performance objectives.
2. Take regular measurements of resource utilization and database activity.
3. Identify symptoms of performance problems: disproportionate utilization of CPU, memory, or disks.
4. Tune the operating-system configuration.
5. Tune the database server configuration.
6. Optimize the chunk and dbspace configuration, including placement of logs, sort space, and space for temporary tables and sort files.
7. Optimize the table placement, extent sizing, and fragmentation.
8. Improve the indexes.
9. Optimize background I/O activities, including logging, checkpoints, and page cleaning.
10. Schedule backup and batch operations for off-peak hours.
11. Optimize the implementation of the database application.
12. Repeat steps 2 through 11.

Quick start for acceptable performance on a small database

If you have a small database with each table residing on only one disk and using only one CPU virtual processor, you can take specific measurements to help you anticipate and correct performance problems.

To achieve acceptable initial performance on a small database:

1. Generate statistics of your tables and indexes to provide information to the query optimizer to enable it to choose query plans with the lowest estimated cost.

   These statistics are a minimum starting point to obtain good performance for individual queries. For guidelines, see Update statistics when they are not generated automatically on page 378. To see the query plan that the optimizer chooses for each query, see Display the query plan on page 366.

2. If you want a query to run in parallel with other queries, you must turn on the Parallel Database Query (PDQ) feature.

   Without table fragmentation across multiple disks, parallel scans do not occur. With only one CPU virtual processor, parallel joins or parallel sorts do not occur. However, PDQ priority can obtain more memory to perform the sort. For more information, see Parallel database query (PDQ) on page 345.

3. If you want to mix online transaction processing (OLTP) and decision-support system (DSS) query applications, you can control the amount of resources a long-running query can obtain so that your OLTP transactions are not affected.

   For information about how to control PDQ resources, see The allocation of resources for parallel database queries on page 352.

4. Monitor sessions and drill down into various details to improve the performance of individual queries.

   For information about the various tools and session details to monitor, see Monitoring memory usage for each session on page 424 and Monitor sessions and threads on page 429.

Performance goals

When you plan for measuring and tuning performance, you should consider performance goals and determine which goals are the most important.

Many considerations go into establishing performance goals for the database server and the applications that it supports. Be clear and consistent about articulating performance goals and priorities, so that you can provide realistic and consistent expectations about the performance objectives for your application. Consider the following questions when you establish performance goals:

- Is your top priority to maximize transaction throughput, minimize response time for specific queries, or achieve the best overall mix?
- What sort of mix between simple transactions, extended decision-support queries, and other types of requests does the database server typically handle?
- At what point are you willing to trade transaction-processing speed for availability or the risk of loss for a particular transaction?

- Is this database server instance used in a client/server configuration? If so, what are the networking characteristics that affect its performance?
- What is the maximum number of users that you expect?
- Is your configuration limited by memory, disk space, or CPU resources?

The answers to these questions can help you set realistic performance goals for your resources and your mix of applications.

Measurements of performance

You can use throughput, response time, cost per transaction, and resource utilization measures to evaluate performance.

Throughput, response time, and cost per transaction are described in the topics that follow.

Resource utilization can have one of two meanings, depending on the context. The term can refer to the amount of a resource that a particular operation requires or uses, or it can refer to the current load on a particular system component. The term is used in the former sense to compare approaches for accomplishing a given task. For instance, if a given sort operation requires 10 megabytes of disk space, its resource utilization is greater than another sort operation that requires only 5 megabytes of disk space. The term is used in the latter sense to refer, for instance, to the number of CPU cycles that are devoted to a particular query during a specific time interval.

For a discussion about the performance impact of different load levels on various system components, see Resource utilization and performance on page 11.

Throughput

Throughput measures the overall performance of the system. For transaction processing systems, throughput is typically measured in *transactions per second* (TPS) or *transactions per minute* (TPM).

Throughput depends on the following factors:

- The specifications of the host computer
- The processing overhead in the software
- The layout of data on disk
- The degree of parallelism that both hardware and software support
- The types of transactions being processed

Ways to measure throughput

The best way to measure throughput for an application is to include code in the application that logs the time stamps of transactions as they commit.

If your application does not provide support for measuring throughput directly, you can obtain an estimate by tracking the number of COMMIT WORK statements that the database server logs during a given time interval. You can use the **onlog** utility to obtain a listing of logical-log records that are written to log files. You can use information from this command to

track insert, delete, and update operations as well as committed transactions. However, you cannot obtain information stored in the logical-log buffer until that information is written to a log file.

If you need more immediate feedback, you can use **onstat -p** to gather an estimate. You can use the SET LOG statement to set the logging mode to unbuffered for the databases that contain tables of interest. You can also use the trusted auditing facility in the database server to record successful COMMIT WORK events or other events of interest in an audit log file. Using the auditing facility can increase the overhead involved in processing any audited event, which can reduce overall throughput.

Related information

Auditing data security

Standard throughput benchmarks

The Transaction Processing Performance Council (TPC) provides standard benchmarks that allow reasonable throughput comparisons across hardware configurations and database servers. HCL is an active member in good standing of the TPC.

The TPC provides the following standardized benchmarks for measuring throughput:

- TPC-A

  This benchmark is used for simple online transaction-processing (OLTP) comparisons. It characterizes the performance of a simple transaction-processing system, emphasizing update-intensive services. TPC-A simulates a workload that consists of multiple user sessions connected over a network with significant disk I/O activity.

- TPC-B

  This benchmark is used for stress-testing peak database throughput. It uses the same transaction load as TPC-A but removes any networking and interactive operations to provide a best-case throughput measurement.

- TPC-C

  This benchmark is used for complex OLTP applications. It is derived from TPC-A and uses a mix of updates, read-only transactions, batch operations, transaction rollback requests, resource contentions, and other types of operations on a complex database to provide a better representation of typical workloads.

- TPC-D

  This benchmark measures query-processing power in terms of completion times for very large queries. TPC-D is a decision-support benchmark built around a set of typical business questions phrased as SQL queries against large databases (in the gigabyte or terabyte range).

Because every database application has its own particular workload, you cannot use TPC benchmarks to predict the throughput for your application. The actual throughput that you achieve depends largely on your application.

Response time

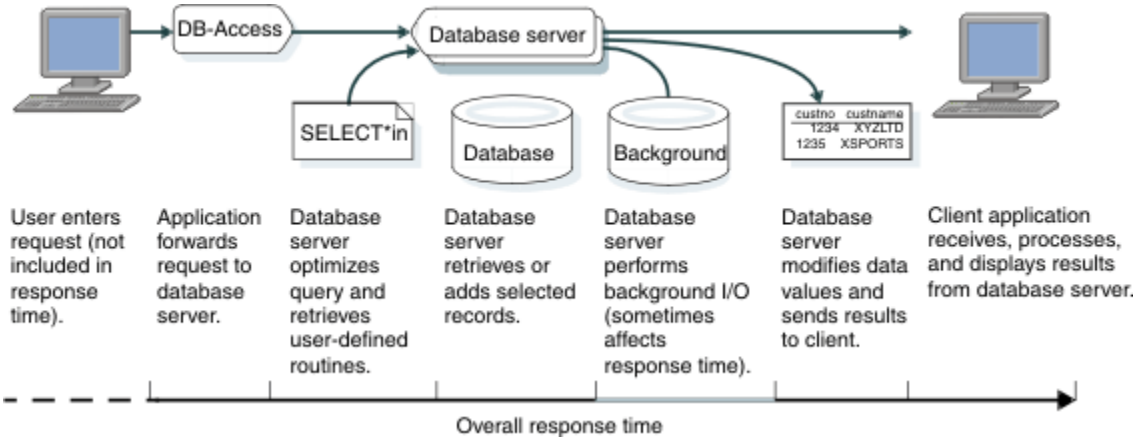
Response time measures the performance of an individual transaction or query. Response time is typically treated as the elapsed time from the moment that a user enters a command or activates a function until the time that the application indicates that the command or function has completed.

The response time for a typical HCL OneDB™ application includes the following sequence of actions. Each action requires a certain amount of time. The response time does not include the time that it takes for the user to think of and enter a query or request:

1. The application forwards a query to the database server.
2. The database server performs query optimization and retrieves any user-defined routines (UDRs). UDRs include both SPL routines and external routines.
3. The database server retrieves, adds, or updates the appropriate records and performs disk I/O operations directly related to the query.
4. The database server performs any background I/O operations, such as logging and page cleaning, that occur during the period in which the query or transaction is still pending.
5. The database server returns a result to the application.
6. The application displays the information or issues a confirmation and then issues a new prompt to the user.

Figure 1: Components of the response time for a single transaction on page 9 contains a diagram that shows how the actions just described in steps 1 through 6 contribute to the overall response time.

Figure 1. Components of the response time for a single transaction



Response time and throughput

Response time and throughput are related. The response time for an average transaction tends to decrease as you increase overall throughput.

However, you can decrease the response time for a specific query, at the expense of overall throughput, by allocating a disproportionate amount of resources to that query. Conversely, you can maintain overall throughput by restricting the resources that the database allocates to a large query.

The trade-off between throughput and response time becomes evident when you try to balance the ongoing need for high transaction throughput with an immediate need to perform a large decision-support query. The more resources that you apply to the query, the fewer you have available to process transactions, and the larger the impact your query can have on transaction throughput. Conversely, the fewer resources you allow the query, the longer the query takes.

Response-time measurement

To measure the response time for a query or application, you can use the timing commands and performance monitoring and timing functions that your operating system provides.

Operating-system timing commands

Your operating system typically has a utility that you can use to time a command. You can often use this timing utility to measure the response times to SQL statements that a DB-Access command file issues.

UNIX™ Only

If you have a command file that performs a standard set of SQL statements, you can use the **time** command on many systems to obtain an accurate timing for those commands.

The following example shows the output of the UNIX™ **time** command:

```
time commands.dba
...
4.3 real          1.5 user              1.3 sys
```

The **time** output lists the amount of elapsed time (real), the user CPU time, and the system CPU time. If you use the C shell, the first three columns of output from the C shell **time** command show the user, system, and elapsed times, respectively. In general, an application often performs poorly when the proportion of system CPU time exceeds one-third of the total elapsed time.

The **time** command gathers timing information about your application. You can use this command to invoke an instance of your application, perform a database operation, and then exit to obtain timing figures, as the following example illustrates:

```
time sqlapp
   (enter SQL command through sqlapp, then exit)
10.1 real          6.4 user           3.7 sys
```

You can use a script to run the same test repeatedly, which allows you to obtain comparable results under different conditions. You can also obtain estimates of your average response time by dividing the elapsed time for the script by the number of database operations that the script performs.

Operating-system tools for monitoring performance

Operating systems usually have a performance monitor that you can use to measure response time for a query or process.

Windows™ Only

You can often use the Performance Logs and Alerts that the Windows™ operating system supplies to measure the following times:

- User time
- Processor time
- Elapsed time

Timing functions within your application

Most programming languages have a library function for the time of day. If you have access to the source code, you can insert pairs of calls to this function to measure the elapsed time between specific actions.

ESQL/C Only

For example, if the application is written in , you can use the **dtcurrent()** function to obtain the current time. To measure response time, you can call **dtcurrent()** to report the time at the start of a transaction and again to report the time when the transaction commits.

Elapsed time, in a multiprogramming system or network environment where resources are shared among multiple processes, does not always correspond to execution time. Most operating systems and C libraries contain functions that return the CPU time of a program.

Cost per transaction

The cost per transaction is a financial measure that is typically used to compare overall operating costs among applications, database servers, or hardware platforms. You can measure the cost per transaction.

To measure the cost per transaction:

1. Calculate all the costs associated with operating an application. These costs can include the installed price of the hardware and software; operating costs, including salaries; and other expenses.
   These costs can include the installed price of the hardware and software; operating costs, including salaries; and other expenses.
2. Project the total number of transactions and queries for the effective life of an application.
3. Divide the total cost over the total number of transactions.

Results

Although this measure is useful for planning and evaluation, it is seldom relevant to the daily issues of achieving optimum performance.

Resource utilization and performance

A typical transaction-processing application undergoes different demands throughout its various operating cycles. Peak loads during the day, week, month, and year, as well as the loads imposed by decision-support (DSS) queries or backup operations, can significantly impact any system that is running near capacity. You can use direct historical data derived from your particular system to pinpoint this impact.

You must take regular measurements of the workload and performance of your system to predict peak loads and compare performance measurements at different points in your usage cycle. Regular measurements help you to develop an overall

performance profile for your database server applications. This profile is critical in determining how to improve performance reliably.

For the measurement tools that the database server provides, see Database server tools on page 22. For the tools that your operating system provides for measuring performance impacts on system and hardware resources, see Operating-system tools on page 21.

Utilization is the percentage of time that a component is actually occupied, as compared with the total time that the component is available for use. For instance, if a CPU processes transactions for a total of 40 seconds during a single minute, its utilization during that interval is 67 percent.

Measure and record utilization of the following system resources regularly:

- CPU
- Memory
- Disk

A resource is said to be *critical* to performance when it becomes overused or when its utilization is disproportionate to that of other components. For instance, you might consider a disk to be critical or overused when it has a utilization of 70 percent and all other disks on the system have 30 percent. Although 70 percent does not indicate that the disk is severely overused, you can improve performance by rearranging data to balance I/O requests across the entire set of disks.

How you measure resource utilization depends on the tools that your operating system provides for reporting system activity and resource utilization. After you identify a resource that seems overused, you can use the performance-monitoring utilities that the database server provides to gather data and make inferences about the database activities that might account for the load on that component. You can adjust your database server configuration or your operating system to reduce those database activities or spread them among other components. In some cases, you might need to provide additional hardware resources to resolve a performance bottleneck.

Resource utilization

Whenever a system resource, such as a CPU or a particular disk, is occupied by a transaction or query, the resource is unavailable for processing other requests. Pending requests must wait for the resources to become available before they can complete.

When a component is too busy to keep up with all its requests, the overused component becomes a bottleneck in the flow of activity. The higher the percentage of time that the resource is occupied, the longer each operation must wait for its turn.

You can use the following formula to estimate the service time for a request based on the overall utilization of the component that services the request. The expected service time includes the time that is spent both waiting for and using the resource in question. Think of service time as that portion of the response time accounted for by a single component within your computer, as the following formula shows:

$$S = P/(1-U)$$

S

is the expected service time.

P

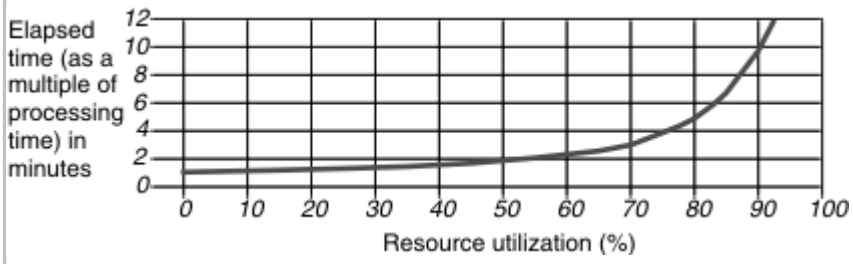
is the processing time that the operation requires after it obtains the resource.

U

is the utilization for the resource (expressed as a decimal).

As Figure 2: Service Time for a Single Component as a Function of Resource Utilization on page 13 shows, the service time for a single component increases dramatically as the utilization increases beyond 70 percent. For instance, if a transaction requires 1 second of processing by a given component, you can expect it to take 2 seconds on a component at 50 percent utilization and 5 seconds on a component at 80 percent utilization. When utilization for the resource reaches 90 percent, you can expect the transaction to take 10 seconds to make its way through that component.

Figure 2. Service Time for a Single Component as a Function of Resource Utilization



If the average response time for a typical transaction soars from 2 or 3 seconds to 10 seconds or more, users are certain to notice and complain.

Important: Monitor any system resource that shows a utilization of over 70 percent or any resource that exhibits symptoms of overuse as described in the following sections.

When you consider resource utilization, also consider whether increasing the page size of a standard or temporary dbspace is beneficial in your environment. If you want a longer key length than is available for the default page size of a standard or temporary dbspace, you can increase the page size.

CPU utilization

Estimates of CPU utilization and response time can help you determine if you need to eliminate or reschedule some activities.

You can use the resource-utilization formula in the previous topic (Resource utilization on page 12) to estimate the response time for a heavily loaded CPU. However, high utilization for the CPU does not always indicate a performance problem. The CPU performs all calculations that are needed to process transactions. The more transaction-related calculations that it performs within a given period, the higher the throughput will be for that period. As long as transaction throughput is high and seems to remain proportional to CPU utilization, a high CPU utilization indicates that the computer is being used to the fullest advantage.

On the other hand, when CPU utilization is high but transaction throughput does not keep pace, the CPU is either processing transactions inefficiently or it is engaged in activity not directly related to transaction processing. CPU cycles are being diverted to internal housekeeping tasks such as memory management.

You can easily eliminate the following activities:

- Large queries that might be better scheduled at an off-peak time
- Unrelated application programs that might be better performed on another computer

If the response time for transactions increases to such an extent that delays become unacceptable, the processor might be swamped; the transaction load might be too high for the computer to manage. Slow response time can also indicate that the CPU is processing transactions inefficiently or that CPU cycles are being diverted.

When CPU utilization is high, a detailed analysis of the activities that the database server performs can reveal any sources of inefficiency that might be present due to improper configuration. For information about analyzing database server activity, see Database server tools on page 22.

Memory utilization

Memory is not managed as a single component, such as a CPU or disk, but as a collection of small components called *pages*.

The size of a typical page in memory can range from 1 to 8 kilobytes, depending on your operating system. A computer with 64 megabytes of memory and a page size of 2 kilobytes contains approximately 32,000 pages.

When the operating system needs to allocate memory for use by a process, it scavenges any unused pages within memory that it can find. If no free pages exist, the memory-management system has to choose pages that other processes are still using and that seem least likely to be needed in the short run. CPU cycles are required to select those pages. The process of locating such pages is called a *page scan*. CPU utilization increases when a page scan is required.

Memory-management systems typically use a *least recently used* algorithm to select pages that can be copied out to disk and then freed for use by other processes. When the CPU has identified pages that it can appropriate, it *pages out* the old page images by copying the old data from those pages to a dedicated disk. The disk or disk partition that stores the page images is called the *swap disk, swap space,* or *swap area*. This paging activity requires CPU cycles as well as I/O operations.

Eventually, page images that have been copied to the swap disk must be brought back in for use by the processes that require them. If there are still too few free pages, more must be paged out to make room. As memory comes under increasing demand and paging activity increases, this activity can reach a point at which the CPU is almost fully occupied with paging activity. A system in this condition is said to be *thrashing*. When a computer is thrashing, all useful work comes to a halt.

To prevent thrashing, some operating systems use a coarser memory-management algorithm after paging activity crosses a certain threshold. This algorithm is called *swapping*. When the memory-management system resorts to swapping, it appropriates all pages that constitute an entire process image at once, rather than a page at a time.

Swapping frees up more memory with each operation. However, as swapping continues, every process that is swapped out must be read in again, dramatically increasing disk I/O to the swap device and the time required to switch between processes. Performance is then limited to the speed at which data can be transferred from the swap disk back into memory. Swapping is a symptom of a system that is severely overloaded, and throughput is impaired.

Many systems provide information about paging activity that includes the number of page scans performed, the number of pages sent out of memory (*paged out*), and the number of pages brought in from memory (*paged in)*:

- Paging out is the critical factor because the operating system pages out only when it cannot find pages that are free already.
- A high rate of page scans provides an early indicator that memory utilization is becoming a bottleneck.
- Pages for terminated processes are freed in place and simply reused, so paging-in activity does not provide an accurate reflection of the load on memory. A high rate of paging in can result from a high rate of process turnover with no significant performance impact.

Although the principle for estimating the service time for memory is the same as that described in Resource utilization and performance on page 11, you use a different formula to estimate the performance impact of memory utilization than you do for other system components.

You can use the following formula to calculate the expected paging delay for a given CPU utilization level and paging rate:

```
PD= (C/(1-U)) * R * T
```

PD

is the paging delay.

C

is the CPU service time for a transaction.

U

is the CPU utilization (expressed as a decimal).

R

is the paging-out rate.

T

is the service time for the swap device.

As paging increases, CPU utilization also increases, and these increases are compounded. If a paging rate of 10 per second accounts for 5 percent of CPU utilization, increasing the paging rate to 20 per second might increase CPU utilization by an additional 5 percent. Further increases in paging lead to even sharper increases in CPU utilization, until the expected service time for CPU requests becomes unacceptable.

Disk utilization

Because transfer rates vary among disks, most operating systems do not report disk utilization directly. Instead, they report the number of data transfers per second (in operating-system memory-page-size units.)

Because each disk acts as a single resource, you can use the following basic formula to estimate the service time, which is described in detail in Resource utilization on page 12:

```
S= P/(1-U)
```

To compare the load on disks with similar access times, simply compare the average number of transfers per second.

If you know the access time for a given disk, you can use the number of transfers per second that the operating system reports to calculate utilization for the disk. To do so, multiply the average number of transfers per second by the access time for the disk as listed by the disk manufacturer. Depending on how your data is laid out on the disk, your access times can vary from the rating of the manufacturer. To account for this variability, you should add 20 percent to the access-time specification of the manufacturer.

The following example shows how to calculate the utilization for a disk with a 30-millisecond access time and an average of 10 transfer requests per second:

```
U =     (A * 1.2) * X
  =     (.03 * 1.2) * 10
  =     .36
```

U

is the resource utilization (this time of a disk).

A

is the access time (in seconds) that the manufacturer lists.

X

is the number of transfers per second that your operating system reports.

You can use the utilization to estimate the processing time at the disk for a transaction that requires a given number of disk transfers. To calculate the processing time at the disk, multiply the number of disk transfers by the average access time. Include an extra 20 percent to account for access-time variability:

```
P = D (A * 1.2)
```

P

is the processing time at the disk.

D

is the number of disk transfers.

A

is the access time (in seconds) that the manufacturer lists.

For example, you can calculate the processing time for a transaction that requires 20 disk transfers from a 30-millisecond disk as follows:

```
P =    20 (.03 * 1.2)
  =    20 * .036
  =    .72
```

Use the processing time and utilization values that you calculated to estimate the expected service time for I/O at the particular disk, as the following example shows:

```
S =    P/(1-U)
  =    .72 / (1 - .36)
  =    .72 / .64
  =    1.13
```

Factors that affect resource utilization

The performance of your database server application depends many factors, including hardware and software configuration, your network configuration, and the design of your database.

You must consider these factors when you attempt to identify performance problems or make adjustments to your system:

- Hardware resources

  As discussed earlier in this chapter, hardware resources include the CPU, physical memory, and disk I/O subsystems.

- Operating-system configuration

  The database server depends on the operating system to provide low-level access to devices, process scheduling, interprocess communication, and other vital services.

  The configuration of your operating system has a direct impact on how well the database server performs. The operating-system kernel takes up a significant amount of physical memory that the database server or other applications cannot use. However, you must reserve adequate kernel resources for the database server to use.

- Network configuration and traffic

  Applications that depend on a network for communication with the database server, and systems that rely on data replication to maintain high availability, are subject to the performance constraints of that network. Data transfers over a network are typically slower than data transfers from a disk. Network delays can have a significant impact on the performance of the database server and other application programs that run on the host computer.

- Database server configuration

  Characteristics of your database server instance, such as the number of CPU virtual processors (VPs), the size of your resident and virtual shared-memory portions, and the number of users, play an important role in determining the capacity and performance of your applications.

- Dbspace, blobspace, and chunk configuration

  The following factors can affect the time that it takes the database server to perform disk I/O and process transactions:

  - The placement of the root dbspace, physical logs, logical logs, and temporary-table dbspaces
  - The presence or absence of mirroring
  - The use of devices that are buffered or unbuffered by the operation system

- Database and table placement

The placement of tables and fragments within dbspaces, the isolation of high-use fragments in separate dbspaces, and the spreading of fragments across multiple dbspaces can affect the speed at which the database server can locate data pages and transfer them to memory.

- Tblspace organization and extent sizing

  Fragmentation strategy and the size and placement of extents can affect the ability of the database server to scan a table rapidly for data. Avoid interleaved extents and allocate extents that are sufficient to accommodate growth of a table to prevent performance problems.

- Query efficiency

  Proper query construction and cursor use can decrease the load that any one application or user imposes. Remind users and application developers that others require access to the database and that each person's activities affect the resources that are available to others.

- Scheduling background I/O activities

  Logging, checkpoints, page cleaning, and other operations, such as making backups or running large decision-support queries, can impose constant overhead and large temporary loads on the system. Schedule backup and batch operations for off-peak times whenever possible.

- Remote client/server operations and distributed join operations

  These operations have an important impact on performance, especially on a host system that coordinates distributed joins.

- Application-code efficiency

  Application programs introduce their own load on the operating system, the network, and the database server. These programs can introduce performance problems if they make poor use of system resources, generate undue network traffic, or create unnecessary contention in the database server. Application developers must make proper use of cursors and locking levels to ensure good database server performance.

Maintenance of good performance

Performance is affected in some way by all system users: the database server administrator, the database administrator, the application designers, and the client application users.

The database server administrator usually coordinates the activities of all users to ensure that system performance meets overall expectations. For example, the operating-system administrator might need to reconfigure the operating system to increase the amount of shared memory. Bringing down the operating system to install the new configuration requires bringing the database server down. The database server administrator must schedule this downtime and notify all affected users when the system will be unavailable.

The database server administrator should:

- Be aware of all performance-related activities that occur.
- Educate users about the importance of performance, how performance-related activities affect them, and how they can assist in achieving and maintaining optimal performance.

The database administrator should pay attention to:

- How tables and queries affect the overall performance of the database server
- The placement of tables and fragments
- How the distribution of data across disks affects performance

Application developers should:

- Carefully design applications to use the concurrency and sorting facilities that the database server provides, rather than attempt to implement similar facilities in the application.
- Keep the scope and duration of locks to the minimum to avoid contention for database resources.
- Include routines within applications that, when temporarily enabled at runtime, allow the database server administrator to monitor response times and transaction throughput.

Database users should:

- Pay attention to performance and report problems to the database server administrator promptly.
- Be courteous when they schedule large, decision-support queries and request as few resources as possible to get the work done.

Performance monitoring and the tools you use

You can use performance monitoring tools to create a performance history, to monitor database resources at scheduled times, or to monitor ongoing transaction or query performance.

This chapter also contains cross-references to topics that about how to interpret the results of performance monitoring

The kinds of data that you need to collect depend on the kinds of applications that you run on your system. The causes of performance problems on OLTP (online transaction processing) systems are different from the causes of problems on systems that are used primarily for DSS query applications. Systems with mixed use provide a greater performance-tuning challenge and require a sophisticated analysis of performance-problem causes.

Evaluate the current configuration

Before you begin to adjust the configuration of your database server, evaluate the performance of your current configuration. You can view the contents of your configuration file with onstat commands.

To alter certain database server characteristics, you must bring down the database server, which can affect your production system. Some configuration adjustments can unintentionally decrease performance or cause other negative side effects.

If your database applications satisfy user expectations, avoid frequent adjustments, even if those adjustments might theoretically improve performance. If your users are reasonably satisfied, take a measured approach to reconfiguring the

database server. When possible, use a test instance of the database server to evaluate configuration changes before you reconfigure your production system.

When performance problems relate to backup operations, you might also examine the number or transfer rates for tape drives. You might need to alter the layout or fragmentation of your tables to reduce the impact of backup operations. For information about disk layout and table fragmentation, see Table performance considerations on page 156 and Indexes and index performance considerations on page 206.

For client/server configurations, consider network performance and availability. Evaluating network performance is beyond the scope of this publication. For information about monitoring network activity and improving network availability, see your network administrator or see the documentation for your networking package.

Determine whether you want to set the configuration parameters that help maintain server performance by automatically adjusting properties of the database server while it is running, for example:

- AUTO_AIOVPS: Adds AIO virtual processors when I/O workload increases.
- AUTO_CKPTS: Increases the frequency of checkpoints to avoid transaction blocking.
- AUTO_LRU_TUNING: Manages cached data flushing as the server load changes.
- AUTO_READAHEAD: Changes the automatic read-ahead mode or disables automatic read-ahead operations for a query.
- AUTO_REPREPARE: Reoptimizes SPL routines and reprepares prepared objects after a schema change.
- AUTO_STAT_MODE: Enables or disables the mode for selectively updating only stale or missing data distributions in UPDATE STATISTICS operations.
- AUTO_TUNE: Enables or disables all automatic tuning configuration parameters that have values that are not present in your configuration file.
- DYNAMIC_LOGS: Allocates additional log files when necessary.
- LOCKS: Allocates additional locks when necessary.
- RTO_SERVER_RESTART: Provides the best performance possible while meeting the recovery time objective after a problem.

Related information

onstat -c command: Print ONCONFIG file contents on page

onstat -g cfg command: Print the current values of configuration parameters on page

Create a performance history

As soon as you set up your database server and begin to run applications on it, you should begin scheduled monitoring of resource use. As you accumulate data, you can analyze performance information.

To accumulate data for performance analysis, use the command-line utilities described in Database server tools on page 22 and Operating-system tools on page 21 in operating scripts or batch files.

The importance of a performance history

If you have a history of the performance of your system, you can begin to track the cause of problems as soon as users report slow response or inadequate throughput.

If a history is not available, you must start tracking performance after a problem arises, and you might not be able to tell when and how the problem began. Trying to identify problems after the fact significantly delays resolution of a performance problem.

To build a performance history and profile of your system, take regular snapshots of resource-utilization information.

For example, if you chart the CPU utilization, paging-out rate, and the I/O transfer rates for the various disks on your system, you can begin to identify peak-use levels, peak-use intervals, and heavily loaded resources.

If you monitor fragment use, you can determine whether your fragmentation scheme is correctly configured. Monitor other resource use as appropriate for your database server configuration and the applications that run on it.

Choose tools from those described in the following sections, and create jobs that build up a history of disk, memory, I/O, and other database server resource use. To help you decide which tools to use to create a performance history, this chapter briefly describes the output of each tool.

Tools that create a performance history

When you monitor database server performance, you use tools from the host operating system and command-line utilities that you can run at regular intervals from scripts or batch files.

You also use performance monitoring tools with a graphical interface to monitor critical aspects of performance as queries and transactions are performed.

Operating-system tools

The database server relies on the operating system of the host computer to provide access to system resources such as the CPU, memory, and various unbuffered disk I/O interfaces and files. Each operating system has its own set of utilities for reporting how system resources are used.

Different implementations of some operating systems have monitoring utilities with the same name but different options and informational displays.

UNIX™ Only

The following table lists some UNIX™ utilities that monitor system resources.

UNIX™ Utility	Description
vmstat utility	Displays virtual-memory statistics
iostat utility	Displays I/O utilization statistics

UNIX™ Utility	Description
sar utility	Displays a variety of resource statistics
ps utility	Displays active process information

For details on how to monitor your operating-system resources, consult the reference manual or your system administration guide.

To capture the status of system resources at regular intervals, use scheduling tools that are available with your host operating system (for example, **cron**) as part of your performance monitoring system.

Windows™ Only

You can often use the Performance Logs and Alerts that the Windows™ operating system supplies to monitor resources such as processor, memory, cache, threads, and processes. The Performance Logs and Alerts also provide charts, alerts, reports, and the ability to save information to log files for later analysis.

For more information about how to use the Performance Logs and Alerts, consult your operating-system manuals.

Database server tools

The database server provides tools and utilities that capture snapshot information about your configuration and performance.

You can use these utilities regularly to build a historical profile of database activity, which you can compare with current operating-system resource-utilization data. These comparisons can help you discover which database server activities have the greatest impact on system-resource utilization. You can use this information to identify and manage your high-impact activities or adjust your database server or operating-system configuration.

The database server tools and utilities that you can use for performance monitoring include:

- The **onstat** utility
- The **onlog** utility
- The **oncheck** utility
- The ON-Monitor utility (on UNIX™ only)
- The **onperf** utility (on UNIX™ only)
- DB-Access and the system-monitoring interface (SMI), which you can use to monitor performance from within your application
- SQL administration API commands

You can use **onstat**, **onlog**, or **oncheck** commands invoked by the **cron** scheduling facility to capture performance-related information at regular intervals and build a historical performance profile of your database server application. The following sections describe these utilities.

You can use SQL SELECT statements to query the system-monitoring interface (SMI) from within your application.

The SMI tables are a collection of tables and pseudo-tables in the **sysmaster** database that contain dynamically updated information about the operation of the database server. The database server constructs these tables in memory but does not record them on disk. The **onstat** utility options obtain information from these SMI tables.

You can use **cron** and SQL scripts with DB-Access or **onstat** utility options to query SMI tables at regular intervals.

Tip: The SMI tables are different from the system catalog tables. System catalog tables contain permanently stored and updated information about each database and its tables (sometimes referred to as *metadata* or a *data dictionary*).

You can use ON-Monitor to check the current database server configuration.

You can use **onperf** to display database server activity with the Motif window manager.

Related information

The onstat utility on page

The onlog utility on page

The oncheck Utility on page

DB-Access User's Guide on page

The System-Monitoring Interface Tables on page

System catalog tables on page

The onperf utility on UNIX on page 443

SQL administration API portal: Arguments by functional category on page

SQL administration API portal: Arguments by privilege groups on page

Performance information that HCL OneDB™ Server Administrator provides

HCL® OneDB® Server Administrator (ISA) is a browser-based tool that provides Web-based system administration for the entire range of HCL OneDB™ database servers.

ISA is the first in a new generation of browser-based, cross-platform administrative tools. It provides access to every HCL OneDB™ database server command-line function and presents the output in an easy-to-read format.

The database server CD-ROM distributed with your product includes ISA. For information on how to install ISA, see the following file on the CD-ROM.

Table 1. Operating system file

Operating System	File
UNIX™	**/SVR_ADM/README**
Windows™	**\SVR_ADM\readme.txt**

With ISA, you can use a browser to perform these common database server administrative tasks:

- Change configuration parameters temporarily or permanently
- Change the database server mode between online and offline and its intermediate states
- Modify connectivity information in the **sqlhosts** file
- Check dbspaces, sbspaces, logs, and other objects
- Manage logical and physical logs
- Examine memory use and adding and freeing memory segments
- Read the message log
- Back up and restore dbspaces and sbspaces
- Run various **onstat** commands to monitor performance
- Enter simple SQL statements and examine database schemas
- Add and remove chunks, dbspaces, and sbspaces
- Examine and manage user sessions
- Examine and manage virtual processors (VPs)
- Use the High-Performance Loader (HPL), **dbimport**, and **dbexport**
- Manage Enterprise Replication
- Manage an HCL® OneDB® MaxConnect server
- Use the following utilities: **dbaccess**, **dbschema**, **onbar**, **oncheck**, **ondblog**, **oninit**, **onlog**, **onmode**, **onparams**, **onspaces**, and **onstat**

You also can enter any HCL OneDB™ utility, UNIX™ shell command, or Windows™ command (for example, **oncheck -cd**; **ls -l**).

Performance information that the onstat utility displays

The **onstat** utility displays a wide variety of performance-related and status information contained within the SMI tables. You can use the **onstat** utility to check the current status of the database server and monitor the activities of the database server.

For a complete list of all **onstat** options, use the onstat - - command. For a complete display of all the information that **onstat** gathers, use the onstat -a command.

Tip: Profile information displayed by **onstat** commands, such as onstat -p, accumulates from the time the database server was started. To clear performance profile statistics so that you can create a new profile, run the onstat -z. If

you use onstat -z to reset statistics for a performance history or appraisal, ensure that other users do not also enter the command at different intervals.

The following table lists some of the **onstat** commands that display general performance-related information.

Table 2. onstat commands that display performance information

onstat command	Description
onstat -p	Displays a performance profile that includes the number of reads and writes, the number of times that a resource was requested but was not available, and other miscellaneous information
onstat -b	Displays information about buffers currently in use
onstat -l	Displays information about the physical and logical logs
onstat -x	Displays information about transactions, including the thread identifier of the user who owns the transaction
onstat -u	Displays a user activity profile that provides information about user threads including the thread owner's session ID and login name
onstat -R	Displays information about buffer pools, including information about buffer pool page size.
onstat -F	Displays page-cleaning statistics that include the number of writes of each type that flushes pages to disk
onstat -g	Requires an additional argument that specifies the information to be displayed For example, onstat -g mem displays memory statistics.

For more information about options that provide performance-related information, see Monitoring fragmentation with the onstat -g ppf command on page 289 and Monitor database server resources on page 25.

Related information

onstat -g monitoring options on page

Monitor database server resources

Monitor specific database server resources to identify performance bottlenecks and potential trouble spots and to improve resource use and response time.

One of the most useful commands for monitoring system resources is onstat -g and its many options.

Monitor resources that impact CPU utilization

Threads, network communications, and virtual processors impact CPU utilization. You can use onstat -g arguments to monitor threads, network communications, and virtual processors.

Use the following onstat -g command options to monitor threads.

onstat -g Option	Description
act	Displays active threads.
ath	Displays all threads. The **sqlexec** threads represent portions of client sessions; the **rstcb** value corresponds to the user field of the onstat -u command.
cpu	Displays the last time the thread ran, how much CPU time the thread used, the number of times the thread ran, and other statistics about all the threads running in the server.
rea	Displays ready threads.
sle	Displays all sleeping threads.
sts	Displays maximum and current stack use per thread.
tpf *tid*	Displays a thread profile for *tid*. If *tid* is `0`, this argument displays profiles for all threads.
wai	Displays waiting threads, including all threads waiting on mutex or condition, or yielding.

Use the following onstat -g command options to monitor the network.

onstat -g Command Option	Description
ntd	Displays network statistics by service.
ntt	Displays network user times.
ntu	Displays network user statistics.
qst	Displays queue statistics.

Use the following onstat -g command options to monitor virtual processors.

onstat -g Command Option	Description
glo	Displays global multithreading information, including CPU-use information about virtual processors, the total number of sessions, and other multithreading global counters.
sch	Displays the number of semaphore operations, spins, and busy waits for each VP.
spi	Displays spin locks that are acquired by virtual processors after they have spun more than 10,000 times.

onstat -g Command Option	Description
	To reduce contention, reduce the number of virtual processors, reduce the load on the computer, or, on some platforms, use the no-age or processor affinity options of virtual processors. If **sh_lock** mutexes have highly contended spin locks, create private memory caches for CPU virtual processors by setting the VP_MEMORY_CACHE_KB configuration parameter.
wst	Displays wait statistics.

Monitor memory utilization

You can use some specific onstat -g command options to monitor memory utilization.

Use the following onstat -g options to monitor memory utilization. For overall memory information, omit *table name*, *pool name*, or *session id* from the commands that permit those optional parameters.

Table 3. onstat -g Options for monitoring memory utilization

Argument	Description
ffr *pool name* | *session id*	Displays free fragments for a pool of shared memory or by session
dic *table name*	Displays one line of information for each table cached in the shared-memory dictionary If you provide a specific table name as a parameter, this argument displays internal SQL information about that table.
dsc	Displays one line of information for each column of distribution statistics cached in the data distribution cache.
mem *pool name* | *session id*	Displays memory statistics for the pools that are associated with a session If you omit *pool_name* | *session id*, this argument displays pool information for all sessions.
mgm	Displays Memory Grant Manager resource information, including: • The values of the PDQ configuration parameters • Memory and scan information • Load information, such as the number of queries that are waiting for memory, the number of queries that are waiting for scans, the number of queries that are waiting for queries with higher PDQ priority to run, and the number of queries that are waiting for a query slot • Active queries and the number of queries at each gate • Statistics on free resources

Table 3. onstat -g Options for monitoring memory utilization (continued)

Argument	Description
	• Statistics on queries • The resource/lock cycle prevention count, which shows the number of times the system immediately activated a query to avoid a potential deadlock
nsc *client id*	Displays shared-memory status by client ID If you omit *client id*, this argument displays all client status areas.
nsd	Displays network shared-memory data for poll threads
nss *session id*	Displays network shared-memory status by session id If you omit session id, this argument displays all session status areas.
osi	Displays information about your operating system resources and parameters, including shared memory and semaphore parameters, the amount of memory currently configured on the computer, and the amount of memory that is unused Use this option when the server is not online.
prc	Displays one line of information for each user-defined routine (SPL routine or external routine written in C or Java™ programming language) cached in the UDR cache
seg	Displays shared-memory-segment statistics This argument shows the number and size of all attached segments.
ses *session id*	Displays memory usage for session id If you omit session id, this argument displays memory usage for all sessions.
ssc	Displays one line of information for each query cached in the SQL statement cache
stm *session id*	Displays memory usage of each SQL statement for session id If you omit session id, this argument displays memory usage for all sessions.
ufr *pool name* | *session id*	Displays allocated pool fragments by user or session

Related information

onstat -g monitoring options on page

Monitor disk I/O utilization

You can use some specific **onstat -g** arguments and the **oncheck** utility to determine if your disk I/O operations are efficient for your applications.

Using onstat -g to monitor I/O utilization

You can use some specific onstat -g command arguments to monitor disk IO.

Use the following onstat -g command arguments to monitor disk I/O utilization.

onstat -g Argument	Description
iof	Displays asynchronous I/O statistics by chunk or file This argument is similar to the **onstat -d**, except that information about nonchunk files also appears. This argument displays information about temporary dbspaces and sort files.
iog	Displays asynchronous I/O global information
ioq	Displays asynchronous I/O queuing statistics
iov	Displays asynchronous I/O statistics by virtual processor

For a detailed case study that uses various **onstat** outputs, see Case studies and examples on page 462.

Using the oncheck utility to monitor I/O utilization

Disk I/O operations are usually the longest component of the response time for a query. You can use the **oncheck** Utility to monitor disk I/O operations.

Contiguously allocated disk space improves sequential disk I/O operations, because the database server can read in larger blocks of data and use the read-ahead feature to reduce the number of I/O operations.

The **oncheck** utility displays information about storage structures on a disk, including chunks, dbspaces, blobspaces, extents, data rows, system catalog tables, and other options. You can also use **oncheck** to determine the number of extents that exist within a table and whether or not a table occupies contiguous space.

The **oncheck** utility provides the following options and information that apply to contiguous space and extents.

Option	Information
-pB	Blobspace simple large object (TEXT or BYTE data) For information about how to use this option to determine the efficiency of blobpage size, see Determine blobpage fullness with oncheck -pB output on page 124.
-pe	Chunks and extents

Option	Information
	For information about how to use this option to monitor extents, see Checking for extent interleaving on page 181 and Eliminating interleaved extents on page 182.
-pk	Index key values. For information about how to improve the performance of this option, see Improving performance for index checks on page 228.
-pK	Index keys and row IDs For information about how to improve the performance of this option, see Improving performance for index checks on page 228.
-pl	Index-leaf key values For information about how to improve the performance of this option, see Improving performance for index checks on page 228.
-pL	Index-leaf key values and row IDs For information about how to improve the performance of this option, see Improving performance for index checks on page 228.
-pp	Pages by table or fragment For information about how to use this option to monitor space, see Considering the upper limit on extents on page 181.
-pP	Pages by chunk For information about how to use this option to monitor extents, see Considering the upper limit on extents on page 181.
-pr	Root reserved pages For information about how to use this option, see Estimating tables with fixed-length rows on page 160.
-ps	Space used by smart large objects and metadata in sbspace.
-pS	Space used by smart large objects and metadata in sbspace and storage characteristics For information about how to use this option to monitor space, see Monitoring sbspaces on page 169.
-pt	Space used by table or fragment

Option	Information
	For information about how to use this option to monitor space, see Estimating table size on page 160.
-pT	Space used by table, including indexes For information about how to use this option to monitor space, see Performance of in-place alters for DDL operations on page 198.

For more information about using **oncheck** to monitor space, see Estimating table size on page 160. For more information about concurrency during **oncheck** execution, see Improving performance for index checks on page 228.

Related information

The oncheck Utility on page

Monitor transactions

You can use the **onlog** and **onstat** utilities to monitor transactions.

Using the onlog utility to monitor transactions

The **onlog** utility displays all or selected portions of the logical log. This utility can help you identify a problematic transaction or gauge transaction activity that corresponds to a period of high utilization, as indicated by your periodic snapshots of database activity and system-resource consumption.

This **onlog** utility can take input from selected log files, the entire logical log, or a backup tape of previous log files.

Use **onlog** with caution when you read logical-log files still on disk, because attempting to read unreleased log files stops other database activity. For greatest safety, back up the logical-log files first and then read the contents of the backup files. With proper care, you can use the **onlog -n** option to restrict **onlog** only to logical-log files that have been released.

To check on the status of logical-log files, use **onstat -l**.

Related information

The onlog utility on page

Using the onstat utility to monitor transactions

If the throughput of transactions is not very high, you can use some **onstat** utility commands to identify a transaction that might be a bottleneck.

Use the following **onstat** utility commands to monitor transactions.

onstat command	Description
onstat -x	Displays transaction information such as number of locks held and isolation level.
onstat -u	Displays information about each user thread
onstat -k	Displays locks held by each session
onstat -g sql	Displays last SQL statement this session executed

Related information

The onstat utility on page

Monitor sessions and queries

Monitoring sessions and threads is important for sessions that perform queries as well as sessions that perform inserts, updates, and deletes. Some of the information that you can monitor for sessions and threads allows you to determine if an application is using a disproportionate amount of the resources.

To monitor database server activity, you can view the number of active sessions and the amount of resources that they are using.

Monitoring memory usage for each session

You can use some specific onstat -g command arguments to get memory information for each session.

Use the following command arguments to get memory information for each session.

onstat -g command and argument	Description
ses	Displays one-line summaries of all active sessions
ses *session id*	Displays session information by *session id*
sql *session id*	Displays SQL information by session If you omit *session id*, this argument displays summaries of all sessions.
stm *session id*	Displays amount of memory used by each prepared SQL statement in a session If you omit *session id*, this argument displays information for all prepared statements.

For examples and discussions of session-monitoring command-line utilities, see Monitoring memory usage for each session on page 424 and Monitor sessions and threads on page 429.

Using the SET EXPLAIN statement

You can use the SET EXPLAIN statement or the EXPLAIN directive to display the query plan that the optimizer creates for an individual query.

About this task

For more information, see Display the query plan on page 366.

Effect of configuration on CPU utilization

The combination of operating-system and HCL OneDB™ configuration parameters can affect CPU utilization. You can change the settings of the HCL OneDB™ configuration parameters that directly affect CPU utilization, and you can adjust the settings for different types of workloads.

Multiple database server instances that run on the same host computer perform poorly when compared with a single database server instance that manages multiple databases. Multiple database server instances cannot balance their loads as effectively as a single database server. Avoid multiple residency for production environments in which performance is critical.

UNIX™ configuration parameters that affect CPU utilization

Your database server distribution includes a machine notes file that contains recommended values for UNIX™ configuration parameters. Because the UNIX™ parameters affect CPU utilization, you should compare the values in the machine notes file with your current operating-system configuration.

The following UNIX™ parameters affect CPU utilization:

- Semaphore parameters
- Parameters that set the maximum number of open file descriptors
- Memory configuration parameters

UNIX™ semaphore parameters

Semaphores are kernel resources with a typical size of 1 byte each. Semaphores for the database server are in addition to any that you allocate for other software packages. You can set some UNIX™ semaphore parameters.

Each instance of the database server requires the following semaphore sets:

- One set for each group of up to 100 virtual processors (VPs) that are started with the database server
- One set for each additional VP that you might add dynamically while the database server is running
- One set for each group of 100 or fewer user sessions connected through the shared-memory communication interface

Tip: For best performance, allocate enough semaphores for double the number of **ipcshm** connections that you expect. Use the NETTYPE configuration parameter to configure database server poll threads for this doubled number of connections.

Because utilities such as **onmode** use shared-memory connections, you must configure a minimum of two semaphore sets for each instance of the database server: one for the initial set of VPs and one for the shared-memory connections that database server utilities use. The SEMMNI operating-system configuration parameter typically specifies the number of semaphore sets to allocate. For information about how to set semaphore-related parameters, see the configuration instructions for your operating system.

The SEMMSL operating-system configuration parameter typically specifies the maximum number of semaphores per set. Set this parameter to at least `100`.

Some operating systems require that you configure a maximum total number of semaphores across all sets, which the SEMMNS operating-system configuration parameter typically specifies. Use the following formula to calculate the total number of semaphores that each instance of the database server requires:

```
SEMMNS = init_vps + added_vps + (2 * shmem_users) + concurrent_utils
```

init_vps

is the number of virtual processors (VPs) that are started with the database server. This number includes CPU, PIO, LIO, AIO, SHM, TLI, SOC, and ADM VPs. The minimum value is `15`.

added_vps

is the number of VPs that you intend to add dynamically.

shmem_users

is the number of shared-memory connections that you allow for this instance of the database server.

concurrent_utils

is the number of concurrent database server utilities that can connect to this instance. It is suggested that you allow for a minimum of six utility connections: two for ON-Bar and four for other utilities such as **onstat**, and **oncheck**.

If you use software packages that require semaphores, the SEMMNI configuration parameter must include the total number of semaphore sets that the database server and your other software packages require. You must set the SEMMSL configuration parameter to the largest number of semaphores per set that any of your software packages require. For systems that require the SEMMNS configuration parameter, multiply SEMMNI by the value of SEMMSL to calculate an acceptable value.

Related information

Configuring poll threads on page 46

UNIX™ file-descriptor parameters

Some operating systems require you to specify a limit on the number of file descriptors that a process can have open at any one time. To specify this limit, use an operating-system configuration parameter, typically NOFILE, NOFILES, NFILE, or NFILES.

The number of open file descriptors that each instance of the database server needs depends on the number of chunks in your database, the number of VPs that you run, and the number of network connections that your database server instance must support.

Use the following formula to calculate the number of file descriptors that your instance of the database server requires:

```
NFILES = (chunks * NUMBER_OF_AIO_VPS) + NUMBER_of_CPU_VPS + net_connections
```

chunks

is the number of chunks to be configured.

net_connections

is the number of network connections that you specify in either of the following places:

- **sqlhosts** file
- NETTYPE configuration entries

Network connections include all but those specified as the **ipcshm** connection type.

Each open file descriptor is about the same length as an integer within the kernel. Allocating extra file descriptors is an inexpensive way to allow for growth in the number of chunks or connections on your system.

UNIX™ memory configuration parameters

The configuration of memory in the operating system can affect other resources, including CPU and I/O.

Insufficient physical memory for the overall system load can lead to thrashing, as Memory utilization on page 14 describes. Insufficient memory for the database server can result in excessive buffer-management activity. For more information about configuring memory, see Configuring UNIX shared memory on page 67.

Windows™ configuration parameters that affect CPU utilization

The HCL OneDB™ distribution includes a machine notes file that contains recommended values for HCL OneDB™ configuration parameters on Windows™. Compare the values in this file with your current ONCONFIG configuration file settings.

About this task

HCL OneDB™ runs in the background. For best performance, give the same priority to foreground and background applications.

On Windows™, to change the priorities of foreground and background applications, go to **Start > Settings > Control Panel,** open the **System** icon, and click the **Advanced Tab.** Select the **Performance Options** button and select either the **Applications** or **Background Services** radio button.

The configuration of memory in the operating system can impact other resources, including CPU and I/O. Insufficient physical memory for the overall system load can lead to thrashing, as Memory utilization on page 14 describes. Insufficient memory for HCL OneDB™ can result in excessive buffer-management activity. When you set the **Virtual Memory** values in the **System** icon on the **Control Panel**, ensure that you have enough paging space for the total amount of physical memory.

Configuration parameters and environment variables that affect CPU utilization

Some configuration parameters and environment variables affect CPU utilization. You might need to adjust the settings of these parameters and variables when you consider methods of improving performance.

The following configuration parameters in the database server configuration file have a significant impact on CPU utilization:

- DS_MAX_QUERIES
- DS_MAX_SCANS
- FASTPOLL
- MAX_PDQPRIORITY
- MULTIPROCESSOR
- NETTYPE
- OPTCOMPIND
- SINGLE_CPU_VP
- VPCLASS
- VP_MEMORY_CACHE_KB

The following environment variables affect CPU utilization:

- **OPTCOMPIND**
- **PDQPRIORITY**
- **PSORT_NPROCS**

The **OPTCOMPIND** environment variable, when set in the environment of a client application, indicates the preferred way to perform join operations. This variable overrides the value that the OPTCOMPIND configuration parameter sets. For details on how to select a preferred join method, see Optimizing access methods on page 43.

The **PDQPRIORITY** environment variable, when set in the environment of a client application, places a limit on the percentage of CPU VP utilization, shared memory, and other resources that can be allocated to any query that the client starts.

A client can also use the SET PDQPRIORITY statement in SQL to set a value for PDQ priority. The actual percentage allocated to any query is subject to the factor that the MAX_PDQPRIORITY configuration parameter sets. For more information about how to limit resources that can be allocated to a query, see Limiting PDQ resources in queries on page 44.

PSORT_NPROCS, when set in the environment of a client application, indicates the number of parallel sort threads that the application can use. The database server imposes an upper limit of 10 sort threads per query for any application. For more information about parallel sorts and **PSORT_NPROCS**, see Configure dbspaces for temporary tables and sort files on page 114.

Related information

Database configuration parameters on page

Environment variables on page

Specifying virtual processor class information

Use the VPCLASS configuration parameter to specify a class of virtual processors, the number of virtual processors that the database server should start for a specific class, and the maximum number allowed.

To execute user-defined routines (UDRs), you can define a new class of virtual processors to isolate UDR execution from other transactions that execute on the CPU virtual processors. Typically you write user-defined routines to support user-defined data types.

If you do not want a user-defined routine to affect the normal processing of user queries in the CPU class, you can use the CREATE FUNCTION statement to assign the routine to a user-defined class of virtual processors. The class name that you specify in the VPCLASS configuration parameter must match the name specified in the CLASS modifier of the CREATE FUNCTION statement.

For guidelines, on using the **cpu** and **num** options of the VPCLASS configuration parameter, see Setting the number of CPU VPs on page 37.

Related information

VPCLASS configuration parameter on page

CREATE FUNCTION statement on page

Setting the number of CPU VPs

You can configure the number of CPU virtual processors (VPs) that the database server uses. Do not allocate more CPU VPs than there are CPU processors available to service them.

When the database server starts, the number of CPU VPs is automatically increased to half the number of CPU processors on the database server computer, unless the SINGLE_CPU_VP configuration parameter is enabled. However, you might want to change the number of CPU VPs based on your performance needs.

You can enable the database server to add CPU VPs as needed, up to the number of CPU processors on the computer. Include the autotune=1 option in the VPCLASS setting:

```
VPCLASS cpu,autotune=1
```

If you do not set the VPCLASS configuration parameter to autotune=1, use the following guidelines to set the number of CPU VPs.

Use the following guidelines to set the number of CPU VPs.

Uniprocessor computers

For uniprocessor computers, specify one CPU VP:

```
VPCLASS cpu,num=1
```

Dual-processor computers

For dual-processor systems, you might improve performance by running with two CPU VPs. To test if performance improves, set the num field of the VPCLASS configuration parameter to 1 in the `onconfig` file and then add a CPU VP dynamically at run time by running the onmode -p command.

Multiprocessor computers that are primarily database servers

For multiprocessor systems with four or more CPUs that are primarily used as database servers, set the num option of the VPCLASS configuration parameter in the `onconfig` file to one less than the total number of processors. For example, if you have four CPUs, use the following specification:

```
VPCLASS cpu,num=3
```

When you use this setting, one processor is available to run the database server utilities or the client application.

Multiprocessor computers that are not primarily database servers

For multiprocessor systems that you do not use primarily to support database servers, you can start with somewhat fewer CPU VPs to allow for other activities on the system and then gradually add more if necessary.

Multi-core or hardware multithreading computers with logical CPUs

For multiprocessor systems that use multi-core processors or hardware multithreading to support more logical CPUs than physical processors, you can assign the number of CPU VPs according to the number of logical CPU VPs available for that purpose. The amount of processing that an additional logical CPU can provide might be only a fraction of what a dedicated physical processor can support.

On systems, where multi-core processors are installed, the optimal configuration in most cases is the same as for systems with a number of individual processors equal to the total number of cores. Setting the number of CPU VPs to N-1, where N is number of cores is close to optimal for CPU-intensive workloads.

On computers where the CPU uses multiple threads per core, operating systems show more logical processors than actual processing cores. To take advantage of more CPU threads, the database server must be configured with the number of CPU VPs in the range between N and M, where N is number of cores and M is total number of logical CPUs reported by system. The number of CPU VPs where optimal performance is achieved depends on the workload.

When increasing the number of CPU VPs to use more threads per core, the expected gain in performance is only a fraction of what dedicated physical processor or core can provide.

If you are migrating OneDB from multi-CPU/multicore systems to systems with multiple threads per core, take special care in regard to processor affinity. When binding OneDB CPU VPs to the logical processors of the operating system, you must be aware of the architecture for the CPU. If you are not sure, do not use the CPU affinity so that the operating system schedules CPU VPs to logical processors with available resources. Using affinity without understanding the relationship between the logical CPUs and processing cores can result in severe performance degradation.

For example, to bind each of 8 configured CPU VPs to a separate core on an 8-core system with two threads per core (16 logical CPUs), use the following setting:

```
VPCLASS cpu,num=8,aff=(0-14/2)
```

Related information

Automatic addition of CPU virtual processors on page 54

VPCLASS configuration parameter on page

Disabling process priority aging for CPU VPs

Use the **noage** option of the VPCLASS configuration parameter to disable process priority aging for database server CPU VPs on operating systems that support this feature. Priority aging occurs when the operating system lowers the priority of long-running processes as they accumulate processing time. You might want to disable priority aging because it can cause the performance of the database server processes to decline over time.

Your database server distribution includes a machine notes file that contains information about whether your version of the database server supports this feature.

Specify the **noage** option of VPCLASS if your operating system supports this feature.

Related information

VPCLASS configuration parameter on page

Specifying processor affinity

Use the **aff** option of the VPCLASS parameter to specify the processors to which you want to bind CPU VPs or AIO VPs. When you assign a CPU VP to a specific CPU, the VP runs only on that CPU. However, other processes can also run on that CPU.

The database server supports automatic binding of CPU VPs to processors on multiprocessor host computers that support processor affinity. Your database server distribution includes a machine notes file that contains information about whether your version of the database server supports this feature.

You can use processor affinity for the purposes that the following sections describe.

Related information

VPCLASS configuration parameter on page

Distributing computation impact

You can use processor affinity to distribute the computation impact of CPU virtual processors (VPs) and other processes. On computers that are dedicated to the database server, assigning CPU VPs to all but one of the CPUs achieves maximum CPU utilization.

On computers that support both database server and client applications, you can bind applications to certain CPUs through the operating system. By doing so, you effectively reserve the remaining CPUs for use by database server CPU VPs, which you bind to the remaining CPUs with the VPCLASS configuration parameter. Set the **aff** option of the VPCLASS configuration parameter to the numbers of the CPUs on which to bind CPU VPs. For example, the following VPCLASS setting assigns CPU VPs to processors 4 to 7:

```
VPCLASS cpu,num=4,aff=(4-7)
```

When specifying a range of processors, you can also specify an incremental value with the range that indicates which CPUs in the range should be assigned to the virtual processors. For example, you can specify that the virtual processors are assigned to every other CPU in the range 0-6, starting with CPU 0.

```
VPCLASS CPU,num=4,aff=(0-6/2)
```

The virtual processors are assigned to CPUs 0, 2, 4, 6.

If you specify `VPCLASS CPU,num=4,aff=(1-10/3)`, the virtual processors are assigned to every third CPU in the range 1-10, starting with CPU 1. The virtual processors are assigned to CPUs 1, 4, 7, 10.

When you specify more than one value or range, the values and ranges do not have to be incremental or in any particular order. For example you can specify `aff=(8,12,7-9,0-6/2)`.

The database server assigns CPU virtual processors to CPUs in a circular pattern, starting with the first processor number that you specify in the *aff* option. If you specify a larger number of CPU virtual processors than physical CPUs, the database server continues to assign CPU virtual processors starting with the first CPU. For example, suppose you specify the following VPCLASS settings:

```
VPCLASS cpu,num=8,aff=(4-7)
```

The database server makes the following assignments:

- CPU virtual processor number `0` to CPU `4`
- CPU virtual processor number `1` to CPU `5`
- CPU virtual processor number `2` to CPU `6`
- CPU virtual processor number `3` to CPU `7`
- CPU virtual processor number `4` to CPU `4`
- CPU virtual processor number `5` to CPU `5`

- CPU virtual processor number `6` to CPU `6`
- CPU virtual processor number `7` to CPU `7`

Related information

VPCLASS configuration parameter on page

Isolating AIO VPs from CPU VPs

On a system that runs database server and client (or other) applications, you can bind asynchronous I/O (AIO) VPs to the same CPUs to which you bind other application processes through the operating system. In this way, you isolate client applications and database I/O operations from the CPU VPs.

This isolation can be especially helpful when client processes are used for data entry or other operations that require waiting for user input. Because AIO VP activity usually comes in quick bursts followed by idle periods waiting for the disk, you can often interweave client and I/O operations without their unduly impacting each other.

Binding a CPU VP to a processor does not prevent other processes from running on that processor. Application (or other) processes that you do not bind to a CPU are free to run on any available processor. On a computer that is dedicated to the database server, you can leave AIO VPs free to run on any processor, which reduces delays on database operations that are waiting for I/O. Increasing the priority of AIO VPs can further improve performance by ensuring that data is processed quickly once it arrives from disk.

Avoiding a certain CPU

The database server assigns CPU VPs to CPUs serially, starting with the CPU number you specify in this parameter. You might want to avoid assigning CPU VPs to a certain CPU that has a specialized hardware or operating-system function (such as interrupt handling).

Setting the number of AIO VPs

Use the aio and num options of the VPCLASS configuration parameter to indicate the number of AIO virtual processors that the database server starts initially.

If your operating system does not support kernel asynchronous I/O (KAIO), the database server uses AIO virtual processors (VPs) to manage all database I/O requests.

If the VPCLASS configuration parameter does not specify the number of AIO VPs to start in the `onconfig` file, the number of AIO VPs initially started is equal to the number of chunks that use AIO, up to a maximum of 128.

You can enable the database server to increase the number of AIO VPs as needed to improve performance. Include the autotune=1 option in the VPCLASS configuration parameter setting:

```
VPCLASS aio,autotune=1
```

If the VPCLASS configuration parameter does not specify the number of AIO VPs to start in the `onconfig` file, then the setting of the AUTO_AIOVPS configuration parameter controls the number of AIO VPs:

- If AUTO_AIOVPS is set to `1` (on), the number of AIO VPs initially started is equal to the number of chunks that use AIO, up to a maximum of 128.
- If AUTO_AIOVPS is set to `0` (off), the number of AIO VPs started is equal to the greater of 6 or twice the number of chunks that use AIO, up to a maximum of 128.

The recommended number of AIO virtual processors depends on how many disks your configuration supports. If KAIO is not implemented on your platform, you should allocate one AIO virtual processor for each disk that contains database tables. You can add an additional AIO virtual processor for each chunk that the database server accesses frequently.

You can use the AUTO_AIOVPS configuration parameter to enable the database server to automatically increase the number of AIO virtual processors and page-cleaner threads when the server detects that AIO virtual processors are not keeping up with the I/O workload.

The machine notes file for your version of the database server indicates whether the operating system supports KAIO. If KAIO is supported, the machine notes describe how to enable KAIO on your specific operating system.

If your operating system supports KAIO, the CPU VPs make asynchronous I/O requests to the operating system instead of AIO virtual processors. In this case, configure only one AIO virtual processor, plus two additional AIO virtual processor for every file chunk that does not use KAIO.

If you use cooked files and if you enable direct I/O using the DIRECT_IO configuration parameter, you can reduce the number of AIO virtual processors. If the database server implements KAIO and if direct I/O is enabled, the database server will attempt to use KAIO, so you probably do not need more than one AIO virtual processor. Temporary dbspaces do not use direct I/O. If you have temporary dbspaces, you will probably need more than one AIO virtual processors.

Even when direct I/O is enabled with the DIRECT_IO configuration parameter, if the file system does not support either direct I/O or KAIO, you still must allocate two additional AIO virtual processors for every active dbspace chunk that is not using KAIO.

The goal in allocating AIO virtual processors is to allocate enough of them so that the lengths of the I/O request queues are kept short (that is, the queues have as few I/O requests in them as possible). When the I/O request queues remain consistently short, I/O requests are processed as fast as they occur. Use the onstat -g ioq command to monitor the length of the I/O queues for the AIO virtual processors.

Allocate enough AIO VPs to accommodate the peak number of I/O requests. Generally, allocating a few extra AIO VPs is not detrimental. To start additional AIO VPs while the database server is in online mode, use the onmode -p command. You cannot drop AIO VPs in online mode.

Related information

AUTO_AIOVPS configuration parameter on page

VPCLASS configuration parameter on page

Setting the MULTIPROCESSOR configuration parameter when using multiple CPU VPs

If you are running multiple CPU VPs, set the MULTIPROCESSOR configuration parameter to `1`. When you set MULTIPROCESSOR to `1`, the database server performs locking in a manner that is appropriate for a multiprocessor. Otherwise, set this parameter to `0`.

The number of CPU VPs is used as a factor in determining the number of scan threads for a query. Queries perform best when the number of scan threads is a multiple (or factor) of the number of CPU VPs. Adding or removing a CPU VP can improve performance for a large query because it produces an equal distribution of scan threads among CPU VPs. For instance, if you have 6 CPU VPs and scan 10 table fragments, you might see a faster response time if you reduce the number of CPU VPs to 5, which divides evenly into 10. You can use **onstat -g ath** to monitor the number of scan threads per CPU VP or use **onstat -g ses** to focus on a particular session.

Related information

MULTIPROCESSOR configuration parameter on page

Setting the SINGLE_CPU_VP configuration parameter when using one CPU VP

If you are running only one CPU VP, set the SINGLE_CPU_VP configuration parameter to `1`. Otherwise, set this parameter to `0`.

Important: If you set the SINGLE_CPU_VP parameter to *1*, the value of the `num` option of the VPCLASS configuration parameter must also be *1*.

Note: The database server treats user-defined virtual-processor classes (that is, VPs defined with VPCLASS) as if they were CPU VPs. Thus, if you set SINGLE_CPU_VP to nonzero, you cannot create any user-defined classes.

When you set the SINGLE_CPU_VP parameter to `1`, you cannot add CPU VPs while the database server is in online mode.

Related information

SINGLE_CPU_VP configuration parameter on page

VPCLASS configuration parameter on page

Optimizing access methods

The OPTCOMPIND configuration parameter helps the query optimizer choose an appropriate access method for your application. When the optimizer examines join plans, OPTCOMPIND indicates the preferred method for performing the join operation for an ordered pair of tables.

About this task

If OPTCOMPIND is equal to `0`, the optimizer gives preference to an existing index (nested-loop join) even when a table scan might be faster. If OPTCOMPIND is set to `1` and the isolation level for a given query is set to Repeatable Read, the optimizer uses nested-loop joins.

When OPTCOMPIND is equal to `2`, the optimizer selects a join method based on cost alone even though table scans can temporarily lock an entire table. For more information about OPTCOMPIND and the different join methods, see Effect of OPTCOMPIND on the query plan on page 314.

To set the value for OPTCOMPIND for specific applications or user sessions, set the **OPTCOMPIND** environment variable for those sessions. Values for this environment variable have the same range and semantics as for the configuration parameter.

Related information

OPTCOMPIND configuration parameter on page

Setting the value of OPTCOMPIND within a session

You can set or change the value of OPTCOMPIND within a session for different kinds of queries. To do this, use the SET ENVIRONMENT OPTCOMPIND statement, not the OPTCOMPIND configuration parameter or the **OPTCOMPIND** environment variable.

For a DSS query, you should set the value of OPTCOMPIND to `2` or `1`, and you should be sure that the isolation level is not set to Repeatable Read. For an OLTP query, you could set the value to `0` or `1` with the isolation level not set to Repeatable Read.

The value that you enter using the SET ENVIRONMENT OPTCOMPIND command takes precedence over the default setting specified by the **OPTCOMPIND** environment variable or by the OPTCOMPIND configuration parameter in the **ONCONFIG** file. The default OPTCOMPIND setting is restored when the routine that issued the SET ENVIRONMENT OPTCOMPIND statement exits, or until the same routine resets the value of OPTCOMPIND to the system default by issuing the following statement:

```
SET ENVIRONMENT OPTCOMPIND DEFAULT;
```

No other user sessions or routines are affected by SET ENVIRONMENT OPTCOMPIND statements that you execute, because their scope is local to the routine in which they are issued, rather than the entire session.

Related information

OPTCOMPIND session environment option on page

Limiting PDQ resources in queries

The MAX_PDQPRIORITY configuration parameter limits the percentage of parallel database query (PDQ) resources that a query can use. Use MAX_PDQPRIORITY to limit the impact of large CPU-intensive queries on transaction throughput.

About this task

To limit the impact of large CPU-intensive queries on transaction throughput

Set the value of the MAX_PDQPRIORITY configuration parameter to an integer that represents a percentage of the following PDQ resources that a query can request:

- Memory
- CPU VPs
- Disk I/O
- Scan threads

Example

When a query requests a percentage of PDQ resources, the database server allocates the MAX_PDQPRIORITY percentage of the amount requested, as the following formula shows:

```
Resources allocated = PDQPRIORITY/100 * MAX_PDQPRIORITY/100
```

For example, if a client uses the SET PDQPRIORITY `80` statement to request 80 percent of PDQ resources, but MAX_PDQPRIORITY is set to `50`, the database server allocates only 40 percent of the resources (50 percent of the request) to the client.

For decision support and online transaction processing (OLTP), setting MAX_PDQPRIORITY allows the database server administrator to control the impact that individual decision-support queries have on concurrent OLTP performance. Reduce the value of MAX_PDQPRIORITY when you want to allocate more resources to OLTP processing. Increase the value of MAX_PDQPRIORITY when you want to allocate more resources to decision-support processing.

What to do next

For more information about how to control the use of PDQ resources, see The allocation of resources for parallel database queries on page 352.

Related information

MAX_PDQPRIORITY configuration parameter on page

Limiting the performance impact of CPU-intensive queries

The DS_MAX_QUERIES configuration parameter specifies a maximum number of decision-support queries that can run at any one time. Queries with a low PDQ priority use proportionally fewer resources, so a larger number of those queries can run simultaneously. You can use the DS_MAX_QUERIES configuration parameter to limit the performance impact of CPU-intensive queries.

The DS_MAX_QUERIES configuration parameter controls only queries with a PDQ priority that is nonzero.

The database server uses the value of DS_MAX_QUERIES with DS_TOTAL_MEMORY to calculate quantum units of memory to allocate to a query. For more information about how the database server allocates memory to queries, see The DS_TOTAL_MEMORY configuration parameter and memory utilization on page 74.

Related information

DS_MAX_QUERIES configuration parameter on page

The DS_TOTAL_MEMORY configuration parameter and memory utilization on page 74

Limiting the number of PDQ scan threads that can run concurrently

The DS_MAX_SCANS configuration parameter limits the number of PDQ scan threads that can run concurrently. This configuration parameter prevents the database server from being flooded with scan threads from multiple decision-support queries.

To calculate the number of scan threads allocated to a query, use the following formula:

```
scan_threads = min (nfrags, (DS_MAX_SCANS * pdqpriority / 100
   * MAX_PDQPRIORITY / 100) )
```

nfrags

is the number of fragments in the table with the largest number of fragments.

pdqpriority

is the PDQ priority value set by either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

Reducing the number of scan threads can reduce the time that a large query waits in the ready queue, particularly when many large queries are submitted concurrently. However, if the number of scan threads is less than *nfrags*, the query takes longer once it is underway.

For example, if a query needs to scan 20 fragments in a table, but the *scan_threads* formula lets the query begin when only 10 scan threads are available, each scan thread scans two fragments serially. Query execution takes approximately twice as long as if 20 scan threads were used.

Related information

DS_MAX_SCANS configuration parameter on page

Configuring poll threads

The NETTYPE configuration parameter configures poll threads for each connection type that your instance of the database server supports. If your database server instance supports connections over more than one interface or protocol, you must specify a separate NETTYPE configuration parameter for each connection type.

You typically include a separate NETTYPE parameter for each connection type that is associated with a dbservername. You list dbservernames in the DBSERVERNAME and DBSERVERALIASES configuration parameters. You associate connection types with dbservernames in the sqlhosts information. For details about connection types and the sqlhosts information, see Connectivity configuration on page in your.*HCL OneDB™ Administrator's Guide*.

Related reference

UNIX semaphore parameters on page 33

Related information

NETTYPE configuration parameter on page

Specifying the connection protocol

The first NETTYPE entry, which specifies the protocol for a given connection type, applies to all dbservernames associated with that type. Subsequent NETTYPE entries for that connection type are ignored.

NETTYPE entries are required for connection types that are used for outgoing communication only even if those connection types are not listed in the `sqlhosts` information.

UNIX™ Only

The following protocols apply to UNIX™ platforms:

- IPCSHM
- TLITCP
- IPCSTR
- SOCTCP
- TLIIMC
- SOCIMC
- SQLMUX
- SOCSSL

Windows™ Only

The following protocols apply to Windows™ platforms:

- SOCTCP
- IPCNMP
- SQLMUX
- SOCSSL

Related information

NETTYPE configuration parameter on page

Specifying virtual-processor classes for poll threads

Each poll thread that is configured or added dynamically by a NETTYPE entry runs in a separate VP. A poll thread can run in one of two types of VP classes: NET (network) and CPU. Network VP classes include SOC, STR, SHM, and TLI. For best

performance, use a NETTYPE entry to assign only one poll thread to the CPU VP class. Assign all additional poll threads to network VP classes by specifying NET in the NETTYPE configuration parameter values.

Related information

NETTYPE configuration parameter on page

Specifying the number of connections and poll threads

The optimum number of connections per poll thread is approximately 300 for uniprocessor computers and up to 350 for multiprocessor computers, although this can vary depending on the platform and database server workload.

A poll thread can support 1024 or more connections. If the FASTPOLL configuration parameter is enabled, you might be able to configure fewer poll threads, but test the performance to determine the optimal configuration for your environment.

Each NETTYPE entry configures the number of poll threads for a specific connection type, the number of connections per poll thread, and the type of virtual-processor class in which those poll threads run. If the number of connections per thread exceeds 350 and the number of poll threads for the current connection type is less than the number of CPU VPs, you can improve performance by specifying the CPU VP class, adding poll threads (do not exceed the number of CPU VPs), and resetting the number of connections per thread. The default number of connections per thread is 50.

Important: Each ipcshm connection requires a semaphore. Some operating systems require that you configure a maximum number of semaphores that can be requested by all software packages that run on the computer. For best performance, double the number of actual ipcshm connections when you allocate semaphores for shared-memory communications. See UNIX semaphore parameters on page 33.

If your computer is a uniprocessor and your database server instance is configured for only one connection type, you can omit the NETTYPE parameter. The database server uses the information that is provided in the sqlhosts information to establish client/server connections.

If your computer is a uniprocessor and your database server instance is configured for more than one connection type, include a separate NETTYPE entry for each connection type. If the number of connections of any one type significantly exceeds 300, assign two or more poll threads, up to a maximum of the number of CPU VPs, and specify NET for a network VP class, as the following example shows:

```
NETTYPE ipcshm,1,50,CPU
NETTYPE tlitcp,2,200,NET # supports 400 connections
```

For **ipcshm**, the number of poll threads correspond to the number of memory segments. For example, if NETTYPE is set to `3,100` and you want one poll thread, set the poll thread to `1,300`.

If your computer is a multiprocessor, your database server instance is configured for only one connection type, and the number of connections does not exceed 350, you can use NETTYPE to specify a single poll thread on either the CPU or a network VP class. If the number of connections exceeds 350, set the VP class type to NET, increase the number of poll threads, and recalculate *conn_per_thread*.

Important: Carefully distinguish between poll threads for network connections and poll threads for shared memory connections, which run one per CPU virtual processor. Configure TCP connections to run in network virtual processors, and configure the minimum that is needed to maintain responsiveness. Configure shared memory connections to run in every CPU virtual processor.

Related information

NETTYPE configuration parameter on page

VPCLASS configuration parameter on page

Improve connection performance and scalability on page 49

Improve connection performance and scalability

You can improve connection performance and scalability by specifying information in the NUMFDSERVERS and NS_CACHE configuration parameters and by using multiple listen threads.

HCL OneDB™ SQL sessions can migrate across CPU VPs. You can improve the performance and scalability of network connections on UNIX™ by using the NUMFDSERVERS configuration parameter to specify a number for the poll threads to use when distributing a TCP/IP connection across VPs. Specifying NUMFDSERVERS information is useful if the database server has a high rate of new connect and disconnect requests or if you find a high amount of contention between network shared file (NSF) locks.

You should also review and, if necessary, change the information in the NETTYPE configuration parameter, which defines the number of poll threads for a specific connection type, the number of connections per poll thread, and the virtual-processor class in which those poll threads run. You specify NETTYPE configuration parameter information as follows:

```
NETTYPE connection_type,poll_threads,conn_per_thread,vp_class
```

On UNIX™, if `vp_class` is `NET`, `poll_threads` can be a value that is greater than or equal to 1. If `vp_class` is `CPU`, the number of `poll_threads` can be 1 through the number of CPU VPs. On Windows™, `poll_threads` can be value that is greater than or equal to 1.

For example, suppose you specify 8 poll threads in the NETTYPE configuration parameter, as follows:

```
NETTYPE soctcp,8,300,NET
```

You can also specify 8 in the NUMFDSERVERS configuration parameter to enable the server to use all 8 poll thread to handle network connections migrating between VPs.

You can use the NS_CACHE configuration parameter to define the maximum retention time for an individual entry in the host name/IP address cache, the service cache, the user cache, and the group cache. The server can get information from the cache faster than it does when querying the operating system.

You can improve service for connection requests by using multiple listen threads. When you specify DBSERVERNAME and DBSERVERALIASES configuration parameter information for **onimcsoc** or **onsoctcp** protocols, you can specify the number of multiple listen threads for the database server aliases in your sqlhosts information. The default value of number is `1`.

The DBSERVERNAME and DBSERVERALIASES configuration parameters define database server names (dbservernames) that have corresponding entries in the sqlhosts information. Each dbservername parameter in the sqlhosts information has a **nettype** entry that specifies an interface/protocol combination. The database server runs one or more poll threads for each unique **nettype** entry.

You can use the onstat -g ath command to display information about all threads.

Related information

NETTYPE configuration parameter on page

NUMFDSERVERS configuration parameter on page

NS_CACHE configuration parameter on page

DBSERVERNAME configuration parameter on page

DBSERVERALIASES configuration parameter on page

Multiple listen threads on page

Name service maximum retention time set in the NS_CACHE configuration parameter on page

Specifying the number of connections and poll threads on page 48

Monitor threads with onstat -g ath output on page 432

Enabling fast polling

You can use the FASTPOLL configuration parameter to enable or disable fast polling of your network, if your operating-system platform supports fast polling. Fast polling is beneficial if you have a large number of connections.

For example, if you have more than 300 concurrent connections with the database server, you can enable the FASTPOLL configuration parameter for better performance.

Related information

FASTPOLL configuration parameter on page

Network buffer pools

The sizes of buffers for TCP/IP connections affect memory and CPU utilization. Sizing these buffers to accommodate a typical request can improve CPU utilization by eliminating the need to break up requests into multiple messages.

However, you must use this capability with care; the database server dynamically allocates buffers of the indicated sizes for active connections. Unless you carefully size buffers, they can use large amounts of memory. For details on how to size network buffers, see Network buffer size on page 53.

The database server dynamically allocates network buffers from the global memory pool for request messages from clients. After the database server processes client requests, it returns buffers to a common network buffer pool that is shared among sessions that use SOCTCP, IPCSTR, or TLITCP network connections.

This common network buffer pool provides the following advantages:

- Prevents frequent allocations and deallocations from the global memory pool
- Uses fewer CPU resources to allocate and deallocate network buffers to and from the common network buffer pool for each network transfer
- Reduces contention for allocation and deallocation of shared memory

The free network buffer pool can grow during peak activity periods. To prevent large amounts of unused memory from remaining in these network buffer pools when network activity is no longer high, the database server returns free buffers when the number of free buffers reaches specific thresholds.

The database server provides the following features to further reduce the allocation and deallocation of and contention for the free network buffers:

- A private free network buffer pool for each session to prevent frequent allocations and deallocations of network buffers from the common network buffer pool or from the global memory pool in shared memory
- Capability to specify a larger than 4-kilobyte buffer size to receive network packets or messages from clients

As the system administrator, you can control the free buffer thresholds and the size of each buffer with the following methods:

- NETTYPE configuration parameter
- **IFX_NETBUF_PVTPOOL_SIZE** environment variable
- **IFX_NETBUF_SIZE** environment variable and b (client buffer size) option in the sqlhosts information

Network buffers

The database server implements a threshold of free network buffers to prevent frequent allocations and deallocations of shared memory for the network buffer pool. This threshold enables the database server to correlate the number of free network buffers with the number of connections that you specify in the NETTYPE configuration parameter.

The database server dynamically allocates network buffers for request messages from clients. After the database server processes client requests, it returns buffers to the network free-buffer pool.

If the number of free buffers is greater than the threshold, the database server returns the memory allocated to buffers over the threshold to the global pool.

The database server uses the following formula to calculate the threshold for the free buffers in the network buffer pool:

```
free network buffers threshold =
   100 + (0.7 * number_connections)
```

The value for *number_connections* is the total number of connections that you specified in the third field of the NETTYPE entry for the different type of network connections (SOCTCP, IPCSTR, or TLITCP). This formula does not use the NETTYPE entry for shared memory (IPCSHM).

If you do not specify a value in the third field of the NETTYPE parameter, the database server uses the default value of 50 connections for each NETTYPE entry corresponding to the SOCTCP, TLITCP, and IPCSTR protocols.

Support for private network buffers

The database server provides support for private network buffers for each session that uses SOCTCP, IPCSTR, or TLITCP network connections.

For situations in which many connections and sessions are constantly active, these private network buffers have the following advantages:

- Less contention for the common network buffer pool
- Fewer CPU resources to allocate and deallocate network buffers to and from the common network buffer pool for each network transfer

The **IFX_NETBUF_PVTPOOL_SIZE** environment variable specifies the size of the private network buffer pool for each session. The default size is one buffer.

Use the **onstat** utility commands in the following table to monitor the network buffer usage.

Command	Output Field	Description
onstat -g ntu	q-pvt	The current number and highest number of buffers that are free in the private pool for this session
onstat -g ntm	q-exceeds	The number of times that the free buffer threshold was exceeded

The onstat -g ntu command displays the following format for the **q-pvt** output field:

```
current number / highest number
```

If the number of free buffers (value in **q-pvt** field) is consistently 0, you can perform one of the following actions:

- Increase the number of buffers with the environment variable **IFX_NETBUF_PVTPOOL_SIZE**.
- Increase the size of each buffer with the environment variable **IFX_NETBUF_SIZE**.

The **q-exceeds** field indicates the number of times that the threshold for the shared network free-buffer pool was exceeded. When this threshold is exceeded, the database server returns the unused network buffers (over this threshold) to the global memory pool in shared memory. Optimally, this value should be 0 or a low number so that the server is not allocating or deallocating network buffers from the global memory pool.

Related information

IFX_NETBUF_PVTPOOL_SIZE environment variable (UNIX) on page

IFX_NETBUF_SIZE environment variable on page

Network buffer size

The **IFX_NETBUF_SIZE** environment variable specifies the size of each network buffer in the common network buffer pool and the private network buffer pool.

The default buffer size is 4 kilobytes.

The **IFX_NETBUF_SIZE** environment variable allows the database server to receive messages longer than 4 kilobytes in one system call. The larger buffer size reduces the amount of overhead required to receive each packet.

Increase the value of **IFX_NETBUF_SIZE** if you know that clients send greater than 4-kilobyte packets. Clients send large packets during any of the following situations:

- Loading a table
- Inserting rows greater than 4 kilobytes
- Sending simple large objects

The **b** option for **sqlhosts** allows the client to send and receive greater than 4 kilobytes. The value for the **sqlhosts** option should typically match the value for IFX_NETBUF_SIZE.

You can use the following **onstat** command to see the network buffer size:

```
onstat -g afr global | grep net
```

The **size** field in the output shows the network buffer size in bytes.

Related information

Connectivity configuration on page

IFX_NETBUF_SIZE environment variable on page

Virtual processors and CPU utilization

While the database server is online, you can start and stop virtual processors (VPs) that belong to certain classes.

You can use **onmode -p** or ON-Monitor to start additional VPs for the following classes while the database server is online: CPU, AIO, PIO, LIO, SHM, TLI, and SOC. You can drop VPs of the CPU class only while the database server is online.

You should carefully distinguish between poll threads for network connections and poll threads for shared memory connections, which should run one per CPU virtual processor. TCP connections should only be in network virtual processors,

and you should only have the minimum needed to maintain responsiveness. Shared memory connections should only be in CPU virtual processors and should run in every CPU virtual processor

Adding virtual processors

Whenever you add a network VP (SOC or TLI), you also add a poll thread. Every poll thread runs in a separate VP, which can be either a CPU VP or a network VP of the appropriate network type.

Adding more VPs can increase the load on CPU resources, so if the NETTYPE value indicates that an available CPU VP can handle the poll thread, the database server assigns the poll thread to that CPU VP. If all the CPU VPs have poll threads assigned to them, the database server adds a second network VP to handle the poll thread.

Automatic addition of CPU virtual processors

When the database server starts, the number of CPU virtual processors (VPs) is automatically increased to half the number of CPU processors on the computer. This ratio of CPU processors to CPU VPs is a recommended minimum to ensure that the database server performs optimally in most situations.

During start up, the database server calculates a target number of CPU VPs that represents an even number equal to or greater than half the number of CPU processors and compares the target number with the currently allocated number of CPU VPs. The database server adds the necessary number of CPU VPs to equal the target number.

If fewer than eight CPU VPs are configured, the server can dynamically add CPU VPs to a total (configured plus added) of eight.

The SINGLE_CPU_VP configuration parameter must be set to 0 for CPU VPs to be automatically added. When CPU VPs are automatically added, the value of the VPCLASS configuration parameter is not updated in the `onconfig` file; therefore, the value of the VPCLASS configuration parameter for CPU VPs might not be the same as the actual number of configured CPU VPs.

Use the **auto_tune_cpu_vps** task in the Scheduler to control the automatic addition of CPU VPs. To prevent the automatic addition of CPU VPs, disable the **auto_tune_cpu_vps** task in the **ph_task** table in the **sysadmin** database:

```
UPDATE ph_task
  SET tk_enable = 'F'
WHERE tk_name = 'auto_tune_cpu_vps';
```

Example

The following table shows possible configurations and how many CPU VPs would be added automatically in each situation.

Table 4. Example of how CPU VPs are automatically added

CPU processors	Target CPU VPs	Allocated CPU VPs	Automatically added CPU VPs
8	4	3	1
3	2	2	0

Table 4. Example of how CPU VPs are automatically added (continued)

CPU processors	Target CPU VPs	Allocated CPU VPs	Automatically added CPU VPs
24	8	6	2

Related information

Setting the number of CPU VPs on page 37

Monitoring virtual processors

Monitor the virtual processors to determine if the number of virtual processors configured for the database server is optimal for the current level of activity.

To monitor virtual processors:

- Use command-line utilities, such as **onstat-g ioq** to view information. See Using some onstat-g commands to monitor virtual processors on page 55
- Use the AUTO_AIOVPS configuration parameter to enable the database server to automatically increase the number of AIO virtual processors and page-cleaner threads when the server detects that AIO virtual processors are not keeping up with the I/O workload.
- Query SMI tables. See Using SMI tables to monitor virtual processors on page 57.

Using some onstat-g commands to monitor virtual processors

You can use the onstat-g glo, onstat-g rea, and onstat-g ioq commands to monitor virtual processors.

Monitor virtual processors with the onstat-g glo command

Use the onstat-g glo command to display information about each virtual processor that is running and to display cumulative statistics for each virtual-processor class.

The onstat -g glo command provides the following types of information:

- How many session threads that are running
- How often threads switch, yield, or need to spin many times to obtain a latch or resource
- The virtual processor classes that are running and how much time each class spent running
- The number of virtual processors that are running for each virtual processor class
- The virtual processors that are running and how much time each virtual processor spent running
- The efficiency of each virtual processor

Use the onstat -g rea command to determine whether you need to increase the number of virtual processors.

Related information

onstat -g glo command: Print global multithreading information on page

Monitor virtual processors with the onstat-g rea command on page 56

Monitor virtual processors with the onstat-g rea command

Use the onstat-g rea command to monitor the number of threads in the ready queue.

onstat-g rea displays this information:

- The **status** field in the output shows the value `ready` when the thread is in the ready queue.
- The **vp-class** output field shows the virtual processor class on which the thread executes.

If the number of threads in the ready queue is growing for a class of virtual processors (for example, the CPU class), you might have to add more of those virtual processors to your configuration.

Figure 3. onstat-g rea output

```
Ready threads:
tid     tcb      rstcb    prty    status                vp-class   name

6       536a38   406464   4       ready                   3cpu    main_loop()
28      60cfe8   40a124   4       ready                   1cpu    onmode_mon
33      672a20   409dc4   2       ready                   3cpu    sqlexec
```

Related information

Monitor virtual processors with the onstat-g glo command on page 55

onstat -g rea command: Print ready threads on page

Monitor virtual processors with the onstat-g ioq command

Use the onstat-g ioq command to determine whether you need to allocate additional AIO virtual processors.

The onstat-g ioq command displays the length of the I/O queues under the column **len**, as the figure below shows. You can also see the maximum queue length (since the database server started) in the **maxlen** column. If the length of the I/O queue is growing, I/O requests are accumulating faster than the AIO virtual processors can process them. If the length of the I/O queue continues to show that I/O requests are accumulating, consider adding AIO virtual processors.

Figure 4. onstat-g ioq and onstat -d output

```
onstat -g ioq

AIO I/O queues:
q name/id    len maxlen totalops  dskread dskwrite  dskcopy
  adt   0      0      0        0        0        0        0
  msc   0      0      1       12        0        0        0
  aio   0      0      4       89       68        0        0
  pio   0      0      1        1        0        1        0
  lio   0      0      1       17        0       17        0
  kio   0      0      0        0        0        0        0
  gfd   3      0      3      254      242       12        0
  gfd   4      0     17      614      261      353        0


onstat -d
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
a1de1d8  1        1        1        1        N        informix rootdbs
a1df550  2        1        2        1        N        informix space1
 2 active, 32,678 maximum
Chunks
address  chk/dbs offset   size     free     bpages   flags pathname
a1de320  1   1   0        75000    66447              PO-   /ix/root_chunk
a1df698  2   2   0        500      447                PO-   /ix//chunk1
 2 active, 32,678 maximum
```

Each chunk serviced by the AIO virtual processors has one line in the onstat-g ioq output, identified by the value `gfd` in the **q name** column. You can correlate the line in **onstat -g ioq** with the actual chunk because the chunks are in the same order as in the onstat -d output. For example, in the onstat-g ioq output, there are two `gfd` queues. The first `gfd` queue holds requests for **root_chunk** because it corresponds to the first chunk shown in the onstat -d output. Likewise, the second `gfd` queue holds requests for **chunk1** because it corresponds to the second chunk in the onstat -d output.

If the database server has a mixture of raw devices and cooked files, the `gfd` queues correspond only to the cooked files in onstat -d output.

Related information

onstat -g ioq command: Print I/O queue information on page

Using SMI tables to monitor virtual processors

You can get information from system-monitoring interface (SMI) tables to use to monitor virtual processors.

You must connect to the **sysmaster** database to query the SMI tables. Query the **sysvpprof** SMI table to obtain information about the virtual processors that are currently running. This table contains the following columns.

Column	Description
vpid	ID number of the virtual processor

Column	Description
class	Class of the virtual processor
user cpu	Seconds of user CPU consumed
syscpu	Seconds of system CPU consumed

Private memory caches

Each CPU virtual processor (VP) or tenant VP can have a private memory cache to speed access time to memory blocks.

All memory allocations that are requested by threads in the database server are fulfilled by memory pools. When a memory pool has insufficient memory blocks to satisfy a memory allocation request, blocks are allocated from the global memory pool. Because all threads use the same global memory pool, contention can occur. Private memory caches allow each virtual processor to retain its own set of memory blocks that can be used to bypass the global memory pool. The initial allocation for private memory caches is from the global memory pool. When the blocks are freed, they are freed to the private memory cache on a specific virtual process. When a memory allocation is requested, the thread first checks whether the allocation can be satisfied by blocks in the private memory cache. Otherwise, the thread requests memory from the global memory pool.

To determine whether private memory caches might improve performance for your database server, run the onstat -g spi command and look for the **sh_lock** mutex. If onstat -g spi command output shows contention for the **sh_lock** mutex, try creating private memory caches.

You set the VP_MEMORY_CACHE_KB configuration parameter to enable private memory caches by specifying the initial combined size of all private memory caches. By default, the total size of private memory caches is limited to the size value of the VP_MEMORY_CACHE_KB configuration parameter. You can set the mode to DYNAMIC to allow the size of each private memory cache to increase or decrease automatically based on the workload of the associated VP. In dynamic mode, the total size of private memory caches can exceed the value of the VP_MEMORY_CACHE_KB configuration parameter, but cannot exceed the value of the SHMTOTAL configuration parameter.

You can view statistics about VP private memory caches by running the onstat -g vpcache command. You can view statistics about memory pools by running the onstat -g mem command.

Attention: If you have multiple VPs, private memory caches can increase the amount of memory that the database server uses.

Related information

VP_MEMORY_CACHE_KB configuration parameter on page

onstat -g vpcache command: Print CPU virtual processor and tenant virtual processor private memory cache statistics on page

onstat -g mem command: Print pool memory statistics on page

onstat -g spi command: Print spin locks with long spins on page

Connections and CPU utilization

Some applications have a large number of client/server connections. Opening and closing connections can consume a large amount of system CPU time.

The following topics describe ways that you might be able to reduce the system CPU time required to open and close connections.

Multiplexed connections and CPU utilization

Many traditional nonthreaded SQL client applications use multiple database connections to perform work for a single user. Each database connection establishes a separate network connection to the database server. The multiplexed connection facility provides the ability for one network connection in the database server to handle multiple database connections from a client application.

Multiplexed connections enable the database server to create multiple database connections without consuming the additional computer resources that are required for additional network connections.

When a nonthreaded client uses a multiplexed connection, the database server still creates the same number of user sessions and user threads as with a nonmultiplexed connection. However, the number of network connections decreases when you use multiplexed connections. Instead, the database server uses a multiplex listener thread to allow the multiple database connections to share the same network connection.

To improve response time for nonthreaded clients, you can use multiplexed connections to execute SQL queries. The amount of performance improvement depends on the following factors:

- The decrease in total number of network connections and the resulting decrease in system CPU time

  The usual cause for a large amount of system CPU time is the processing of system calls for the network connection. Therefore, the maximum decrease in system CPU time is proportional to the decrease in the total number of network connections.
- The ratio of this decrease in system CPU time to the user CPU time

  If the queries are simple and use little user CPU time, you might experience a sizable reduction in response time when you use a multiplexed connection. But if the queries are complex and use a large amount of user CPU time, you might not experience a performance improvement.

  To get an idea of the amounts of system CPU time and user CPU times per virtual processor, use the **onstat -g glo** option.

To use multiplexed connections for a nonthreaded client application, you must take the following steps before you bring up the database server:

1. Define an alias using the DBSERVERALIASES configuration parameter. For example, specify:

   ```
   DBSERVERALIASES ids_mux
   ```

2. Add an SQLHOSTS entry for the alias using `sqlmux` as the **nettype** entry, which is the second column in the SQLHOSTS file. For example, specify:

   ```
   ids_mux  onsqlmux  ......
   ```

   The other fields in this entry, the **hostname** and **servicename**, must be present, but they are ignored.

3. Enable multiplexing for the selected connection types by specifying `m=1` in the **sqlhosts** file or registry that the client uses for the database server connection.
4. On Windows™ platforms, you must also set the **IFX_SESSION_MUX** environment variable.

Warning: On Windows™, a multithreaded application must not use the multiplexed connection feature. If a multithreaded application enables the multiplexing option in the **sqlhosts** registry entry and also defines the IFX_SESSION_MUX environment variable, it can produce disastrous results, including crashing and data corruption.

Related information

Multiplexed connections on page

Supporting multiplexed connections on page

Effect of configuration on memory utilization

The combination of operating-system and HCL OneDB™ configuration parameters can affect memory utilization.

You can change the settings of the HCL OneDB™ configuration parameters that directly affect memory utilization, and you can adjust the settings for different types of workloads.

Consider the amount of physical memory that is available on your host when you allocate shared memory for the database server by setting operating-system configuration parameters. In general, if you increase space for database server shared memory, you can enhance the performance of your database server. You must balance the amount of shared memory that is dedicated to the database server against the memory requirements for VPs and other processes.

Related information

The Memory Grant Manager on page 351

Shared memory

You must configure adequate shared-memory resources for the database server in your operating system. Insufficient shared memory can adversely affect performance.

The database server threads and processes require shared memory to share data by sharing access to segments of memory.

The shared memory that HCL OneDB™ uses can be divided into the following parts, each of which has one or more shared memory segments:

- Resident portion
- Virtual portion
- Message portion
- Buffer pool portion

The resident and message portions are static; you must allocate sufficient memory for them before you bring the database server into online mode. (Typically, you must reboot the operating system to reconfigure shared memory.) The virtual portion of shared memory for the database server grows dynamically, but you must still include an adequate initial amount for this portion in your allocation of operating-system shared memory.

The amount of space that is required is the total that all portions of database server shared memory need. You specify the total amount of shared memory with the SHMTOTAL configuration parameter.

The LOCKS configuration parameter specifies the initial size of the lock table. If the number of locks that sessions allocate exceeds the value of LOCKS, the database server dynamically increases the size of the lock table. If you expect the lock table to grow dynamically, set SHMTOTAL to 0. When SHMTOTAL is 0, there is no limit on total memory (including shared memory) allocation.

Related information

LOCKS configuration parameter on page

SHMTOTAL configuration parameter on page

Resident portion of shared memory

The resident portion of shared memory includes areas of shared memory that record the state of the database server, including locks, log files, and the locations of dbspaces, chunks, and tblspaces. The resident portion of shared memory includes areas of shared memory that record the state of the database server, including buffers, locks, log files, and the locations of dbspaces, chunks, and tblspaces.

The settings that you use for the LOCKS, LOGBUFF, and PHYSBUFF configuration parameters help determine the size of the resident portion.

The BUFFERPOOL configuration parameter determines the number of buffers allocated to the resident segment when the database server is started. Subsequent buffer pools that are added while the database server is running are moved into virtual memory until the database server is restarted.

In addition to these configuration parameters, which affect the size of the resident portion, the RESIDENT configuration parameter can affect memory use. When a computer supports forced residency and the RESIDENT configuration parameter is set to a value that locks the resident or resident and virtual portions, the resident portion is never paged out.

The machine notes file for your database server indicates whether your operating system supports forced residency.

On AIX®, Solaris, and Linux™ systems that support large pages, the IFX_LARGE_PAGES environment variable can enable the use of large pages for non-message shared memory segments that are locked in physical memory. If large pages are configured by operating system commands and the RESIDENT configuration parameter specifies that some or all of the resident and virtual portions of shared memory are locked in physical memory, OneDB uses large pages for the corresponding shared memory segments, provided sufficient large pages are available. The use of large pages can offer significant performance benefits in large memory configurations.

Related reference

Configuration parameters that affect memory utilization on page 69

Related information

IFX_LARGE_PAGES environment variable on page

Virtual portion of shared memory

OneDB uses the virtual portion of shared memory to allocate memory to each database server subsystem, as needed.

The virtual portion of shared memory for the database server includes the following components:

- Large buffers, which are used for large read and write I/O operations
- Sort-space pools
- Active thread-control blocks, stacks, and heaps
- User-session data
- Caches for SQL statements, data-dictionary information, and user-defined routines
- A global pool for network-interface message buffers and other information

The SHMVIRTSIZE configuration parameter in the `onconfig` file provides the initial size of the virtual portion. As the need for additional space in the virtual portion arises, the database server adds shared memory in increments that the SHMADD configuration parameter specifies. The EXTSHMADD configuration parameter configures the size of the virtual-extension shared memory segments that are added for user-defined routines and DataBlade® routines. The limit on the total shared memory allocated to the database server is specified by the SHMTOTAL parameter.

The size of the virtual portion depends primarily on the types of applications and queries that you are running. Depending on your application, an initial estimate for the virtual portion might be as low as 100 KB per user or as high as 500 KB per user, plus an additional 4 megabytes if you intend to use data distributions.

When a computer supports forced residency and the RESIDENT configuration parameter is set to a value that locks virtual segments, the virtual segments that are locked are never paged out.

On AIX®, Solaris, and Linux™ systems that support large pages, the IFX_LARGE_PAGES environment variable can enable the use of large pages for non-message shared memory segments that are locked in physical memory. If large pages are configured by operating system commands and the RESIDENT configuration parameter specifies that some or all of the resident and virtual portions of shared memory are locked in physical memory, OneDB uses large pages for the corresponding shared memory segments, provided sufficient large pages are available. The use of large pages can offer significant performance benefits in large memory configurations.

Related reference

Configuration parameters that affect memory utilization on page 69

Related information

IFX_LARGE_PAGES environment variable on page

Creating data distributions on page 380

EXTSHMADD configuration parameter on page

SHMADD configuration parameter on page

SHMTOTAL configuration parameter on page

SHMVIRTSIZE configuration parameter on page

Message portion of shared memory

The message portion of shared memory contains the message buffers that the shared-memory communication interface uses. The amount of space required for these buffers depends on the number of user connections that you allow using a given networking interface.

If a particular interface is not used, you do not need to include space for it when you allocate shared memory in the operating system.

Buffer pool portion of shared memory

The buffer pool portion of shared memory contains one or more buffer pools. Each page size that is used by a dbspace has a buffer pool.

The BUFFERPOOL configuration parameter specifies the size of the buffer pool when the database server is started. If the buffer pool is extendable, the database server increases the size of the buffer pool in the buffer pool portion of shared memory.

You can determine the current size of the buffer pool portion of shared memory by running the onstat -g buf command and adding the values in the **Total Mem** field for each buffer pool. For example, the following output shows that the memory for one buffer pool is 32 MB:

```
Fg Writes     LRU Writes    Avg. LRU Time Chunk Writes  Total Mem
0             0             nan           10883         32Mb
```

The maximum size of each buffer pool depends on the amount of available shared memory and the values of the BUFFERPOOL configuration parameters.

Related information

Shared-memory buffer pool on page

Buffer pool portion of shared memory on page

BUFFERPOOL configuration parameter on page

onstat -g buf command: Print buffer pool profile information on page

Estimating the size of the resident portion of shared memory

You can use formulas to estimate the size of the resident portion (in KB) of shared memory when you allocate operating-system shared memory.

About this task

The result of your calculations is an estimate that normally, slightly exceeds the actual memory that is used for the resident portion of shared memory.

The following estimate was calculated to determine the resident portion of shared memory on a 64-bit server. The sizes that are shown are subject to change, and the calculation is approximate.

To estimate the size of the resident portion of shared memory

1. Estimate the size of the data buffer, using the following formula:

   ```
   buffer_value = (BUFFERS * pagesize) + (BUFFERS * 254) + 250000
   ```

   pagesize

   is the shared-memory page size, as onstat -b shows it on the last line in the **buffer size** field.

   If you have multiple buffer pools, add the buffer sizes for each buffer pool together.

2. Calculate the values in the following formulas:

   ```
   locks_value = LOCKS * 136
   logbuff_value = LOGBUFF * 1024 * 3
   physbuff_value = PHYSBUFF * 1024 * 2
   ```

   ```
   locks_value = LOCKS * 128
   logbuff_value = LOGBUFF * 1024 * 3
   physbuff_value = PHYSBUFF * 1024 * 2
   ```

3. Calculate the estimated size of the resident portion in KB, using the following formula:

   ```
   rsegsize = 1.02 * (locks_value + logbuff_value
              + physbuff_value + 1,200,000) / 1024
   ```

   ```
   rsegsize = 1.02 * (buffer_value + locks_value
    + logbuff_value  + physbuff_value + 1,200,000) / 1024
   ```

Estimating the size of the virtual portion of shared memory

You can use a formula to estimate the initial size of the virtual portion of shared memory. You specify the initial size in the SHMVIRTSIZE configuration parameter.

About this task

The formula for estimating an initial size of the virtual portion of shared memory is as follows:

```
shmvirtsize = fixed overhead + shared structures +
              (mncs * private structures) +
              other buffers
```

To estimate an SHMVIRTSIZE value with the preceding formula:

1. Estimate the value for the `fixed overhead` portion of the formula as follows:

   ```
   fixed overhead = global pool +
                    thread pool after booting
   ```

   a. Run the onstat -g mem command to obtain the pool sizes allocated to sessions.
   b. Subtract the value in the **freesize** field from the value in the **totalsize** to obtain the number of bytes allocated per session.
   c. Estimate a value for the *thread pool after booting* variable. This variable is partially dependent on the number of virtual processors.
2. Estimate the value of `shared structures` with the following formula:

   ```
   shared structures = AIO vectors + sort memory +
               dbspace backup buffers +
               data-dictionary cache size +
               size of user-defined routine cache +
               histogram pool +
               STMT_CACHE_SIZE (SQL statement cache) +
               other pools (See onstat display.)
   ```
3. Estimate the next part of the formula, as follows:

   a. Estimate the value of `mncs` (which is the maximum number of concurrent sessions) with the following formula:

      ```
      mncs = number of poll threads *
             number connections per poll thread
      ```

      The value for number of poll threads is the value that you specify in the second field of the NETTYPE configuration parameter.

      The value for *`number of connections per poll thread`* is the value that you specify in the third field of the NETTYPE configuration parameter.

      You can also obtain an estimate of the maximum number of concurrent sessions when you run the onstat -u command during peak processing. The last line of the onstat -u output contains the maximum number of concurrent user threads.

   b. Estimate the value of *`private structures`*, as follows:

```
private structures = stack + heap +
                     session control-block structures
```

stack

Generally 32 KB but dependent on recursion in user-defined routines. You can obtain the stack size for each thread with the onstat -g sts option.

heap

About 15 KB. You can obtain the heap size for an SQL statement when you use the onstat -g stm option.

session control-block structures

The amount of memory used per session. The onstat -g ses option displays the amount of memory, in bytes, in the **total memory** column listed for each session id.

c. Multiply the results of steps 3a and 3b to obtain the following part of the formula:

```
mncs * private structures
```

4. Estimate the value of *other buffers* to account for private buffers allocated for features such as lightweight I/O operations for smart large objects (about 180 KB per user).
5. Add the results of steps 1 through 4 to obtain an estimate for the SHMVIRTSIZE configuration parameter.

Results

Tip: When the database server is running with a stable workload, you can use onstat -g seg to obtain a precise value for the actual size of the virtual portion of shared memory. You can then use the value for shared memory that this command reports to reconfigure SHMVIRTSIZE.

To specify the size of segments that are added later to the virtual shared memory, set the SHMADD configuration parameter. Use the EXTSHMADD configuration parameter to specify the size of virtual-extension segments that are added for user-defined routines and DataBlade® routines.

What to do next

The following table contains a list of additional topics for estimating the size of shared structures in memory.

Table 5. Information for shared-memory structures

Shared-Memory Structure	More Information
Sort memory	Estimating memory needed for sorting on page 226
Data-dictionary cache	Data-dictionary configuration on page 86
Data-distribution cache (histogram pool)	Data-distribution configuration on page 87
User-defined routine (UDR) cache	SPL routine executable format stored in UDR cache on page 325

Table 5. Information for shared-memory structures (continued)

Shared-Memory Structure	More Information
SQL statement cache	Enabling the SQL statement cache on page 422 Monitor and tune the SQL statement cache on page 89
Other pools	To see how much memory is allocated to the different pools, use the onstat -g mem command.

Related information

SHMVIRTSIZE configuration parameter on page

NETTYPE configuration parameter on page

Session memory on page 102

onstat -g mem command: Print pool memory statistics on page

Estimating the size of the message portion of shared memory

You can estimate the size of the message portion of shared memory in kilobytes.

About this task

Estimate the size of the message portion of shared memory, using the following formula:

```
msegsize = (10,531 * ipcshm_conn + 50,000)/1024
```

ipcshm_conn

is the number of connections that can be made using the shared-memory interface, as determined by the NETTYPE parameter for the **ipcshm** protocol.

Related information

NETTYPE configuration parameter on page

Configuring UNIX™ shared memory

On UNIX™, you can configure shared-memory segments for the database server.

About this task

On UNIX™, perform the following steps to configure the shared-memory segments that your database server configuration needs. For information about how to set parameters related to shared memory, see the configuration instructions for your operating system.

To configure shared-memory segments for the database server:

1. If your operating system does not have a size limit for shared-memory segments, take the following actions:
    a. Set the operating-system configuration parameter for maximum segment size, typically SHMMAX or SHMSIZE, to the total size that your database server configuration requires. This size includes the amount of memory that is required to start your database server instance and the amount of shared memory that you allocate for dynamic growth of the virtual portion.
    b. Set the operating-system configuration parameter for the maximum number of segments, typically SHMMNI, to at least `1` per instance of the database server.
2. If your operating system has a segment-size limit, take the following actions:

    a. Set the operating-system configuration parameter for the maximum segment size, typically SHMMAX or SHMSIZE, to the largest value that your system allows.

    b. Use the following formula to calculate the number of segments for your instance of the database server. If there is a remainder, round up to the nearest integer.

    ```
    SHMMNI = total_shmem_size / SHMMAX
    ```

    total_shmem_size

    is the total amount of shared memory that you allocate for the database server use.

3. Set the operating-system configuration parameter for the maximum number of segments, typically SHMMNI, to a value that yields the total amount of shared memory for the database server when multiplied by SHMMAX or SHMSIZE. If your computer is dedicated to a single instance of the database server, that total can be up to 90 percent of the size of virtual memory (physical memory plus swap space).
4. If your operating system uses the SHMSEG configuration parameter to indicate the maximum number of shared-memory segments that a process can attach, set this parameter to a value that is equal to or greater than the largest number of segments that you allocate for any instance of the database server.

Results

For additional tips on configuring shared memory in the operating system, see the machine notes file for UNIX™ or the release notes file for Windows™.

Related information

The SHMADD and EXTSHMADD configuration parameters and memory utilization on page 80

Freeing shared memory with onmode -F

You can run the onmode -F command to free shared-memory segments that are unavailable or no longer needed for a process.

Restriction: Do not run the onmode -F command if HCL OneDB™ 11.70.xC7 is running on Solaris 11 systems. Upgrade to HCL OneDB™ 11.70.xC8 or a later version, and then run the command.

The database server does not automatically free the shared-memory segments that it adds during its operations. After memory has been allocated to the database server virtual portion, the memory remains unavailable for use by other processes running on the host computer. When the database server runs a large decision-support query, it might acquire a large amount of shared memory. After the query completes, the database server no longer requires that shared memory. However, the shared memory that the database server allocated to service the query remains assigned to the virtual portion even though it is no longer needed.

The onmode -F command locates and returns unused 8-kilobyte blocks of shared memory that the database server still holds. Although this command runs only briefly (one or two seconds), onmode -F dramatically inhibits user activity while it runs. Systems with multiple CPUs and CPU VPs typically experience less degradation while this utility runs.

You should run onmode -F during slack periods with an operating-system scheduling facility (such as cron on UNIX™). In addition, consider running this utility after you perform any task that substantially increases the size of database server shared memory, such as large decision-support queries, index builds, sorts, or backup operations.

Related information

onmode -F: Free unused memory segments on page

Configuration parameters that affect memory utilization

A large number of configuration parameters in the ONCONFIG file affect memory utilization and performance.

The following configuration parameters significantly affect memory utilization:

- BUFFERPOOL
- DS_NONPDQ_QUERY_MEM
- DS_TOTAL_MEMORY
- EXTSHMADD
- LOCKS
- LOGBUFF
- LOW_MEMORY_MGR
- LOW_MEMORY_RESERVE
- PHYSBUFF
- RESIDENT
- SHMADD
- SHMBASE
- SHMTOTAL
- SHMVIRTSIZE
- SHMVIRT_ALLOCSEG

- STACKSIZE
- Memory cache parameters (see Configure and monitor memory caches on page 83)
- Network buffer size (see Network buffer pools on page 50)

The SHMBASE parameter indicates the starting address for database server shared memory. When set according to the instructions in the machine notes file or release notes file, this parameter has no appreciable effect on performance. For the path name of each file, see the Introduction to this guide.

The DS_NONPDQ_QUERY_MEM parameter increases the amount of memory that is available for non-PDQ queries. You can only use this parameter if PDQ priority is set to zero. For more information, see Configuring memory for queries with hash joins, aggregates, and other memory-intensive elements on page 414.

The following sections describe the performance effects and considerations associated with some of the configuration parameters that are listed at the beginning of this section.

Related information

Resident portion of shared memory on page 61

Virtual portion of shared memory on page 62

LOW_MEMORY_MGR configuration parameter on page

LOW_MEMORY_RESERVE configuration parameter on page

Setting the size of the buffer pool, logical-log buffer, and physical-log buffer

The values that you specify for the BUFFERPOOL, DS_TOTAL_MEMORY, LOGBUFF, and PHYSBUFF configuration parameters depend on the type of applications that you are using (OLTP or DSS) and the page size.

Table 6: Guidelines for OLTP and DSS applications on page 70 lists suggested settings for these parameters or guidelines for setting the parameters.

For information about estimating the size of the resident portion of shared memory, see Estimating the size of the resident portion of shared memory on page 64. This calculation includes figuring the size of the buffer pool, logical-log buffer, physical-log buffer, and lock table.

Table 6. Guidelines for OLTP and DSS applications

Configuration Parameter	OLTP Applications	DSS Applications
BUFFERPOOL	The percentage of physical memory that you need for buffer space depends on the amount of memory that is available on your system and the amount of memory that is used for other applications.	Set to a small buffer value and increase the DS_TOTAL_MEMORY value for light scans, queries, and sorts. For operations such as index builds that read data through the buffer pool, configure a larger number of buffers.

Table 6. Guidelines for OLTP and DSS applications (continued)

Configuration Parameter	OLTP Applications	DSS Applications
DS_TOTAL_MEMORY	Set to a value from 20 to 50 percent of the value of SHMTOTAL, in kilobytes.	Set to a value from 50 to 90 percent of SHMTOTAL.
LOGBUFF	The default value for the logical log buffer size is 64 KB. If you decide to use a smaller value, the database server generates a message a message that indicates that optimal performance might not be obtained. Using a logical log buffer smaller than 64 KB, impacts performance, not transaction integrity. If the database or application is defined to use buffered logging, increasing the LOGBUFF size beyond 64 KB improves performance.	Because database or table logging is usually turned off for DSS applications, you can set LOGBUFF to 32 KB.
PHYSBUFF	The default value for the physical log buffer size is 128 KB. If the RTO_SERVER_RESTART configuration parameter is enabled, use the 512 kilobyte default value for PHYSBUFF. If you decide to use a value that is smaller than the default value, the database server generates a message that indicates that optimal performance might not be obtained. Using a physical log buffer that is smaller than the default size impacts performance, not transaction integrity.	Because most DSS applications do not physically log, you can set PHYSBUFF to 32 KB.

Related information

BUFFERPOOL configuration parameter on page

DS_TOTAL_MEMORY configuration parameter on page

LOGBUFF configuration parameter on page

PHYSBUFF configuration parameter on page

RTO_SERVER_RESTART configuration parameter on page

The BUFFERPOOL configuration parameter and memory utilization

The BUFFERPOOL configuration parameter specifies the properties of buffer pools. The information that you define in the BUFFERPOOL configuration parameter fields affects memory use.

You can have multiple buffer pools if you have dbspaces that use different page sizes. The `onconfig` configuration file contains a BUFFERPOOL line for each page size. For example, on a computer with a 2 KB page size, the `onconfig` file can contain up to nine lines, including the default specification. When you create a dbspace with a different page size, a buffer pool for that page size is created automatically, if it does not exist. A BUFFERPOOL entry for the page size is added to the `onconfig` file. The values of the BUFFERPOOL configuration parameter fields are the same as the default specification.

The BUFFERPOOL configuration parameter controls the number of data buffers available to the database server. These buffers are in the buffer pool portion of shared memory and are used to cache database data pages in memory.These buffers are in the resident portion of shared memory (buffer pool) and are used to cache database data pages in memory.

Increasing the number of buffers increases the likelihood that a needed data page might already be in memory as the result of a previous request. However, allocating too many buffers can affect the memory-management system and lead to excess operating system paging activity. To take advantage of the large memory available on 64-bit addressing machines, you can increase the size of the buffer pool.

The size of the buffer pool has a significant effect on database I/O and transaction throughput.

The size of the buffer pool has a significant effect on database I/O and transaction throughput. You can ensure that the buffer pool has enough buffers by making the buffer pool extendable. When the buffer pool is extendable, the database server expands the buffer pool as needed to improve performance.

The size of the buffer pool is equal to the number of buffers multiplied by the page size. The percentage of physical memory that you need for buffer space depends on the amount of memory that you have available on your system and the amount that is used for other applications. For systems with a large amount of available physical memory (4 GB or more), buffer space might be as much as 90 percent of physical memory. For systems with smaller amounts of available physical memory, buffer space might range from 20 to 25 percent of physical memory.

For example, suppose that your system has a page size of 2 KB and 100 MB of physical memory. You can set the value in the buffers field to 10,000 - 12,500, which allocates 20 - 25 MB of memory.

Calculate all other shared-memory parameters after you specify the size of the buffer pool.

Note: If you use non-default page sizes, you might need to increase the size of your physical log. If you frequently update non-default pages, you might need a 150 - 200 percent increase of the physical log size. Some

experimentation might be needed to tune the physical log. You can adjust the size of the physical log as necessary according to how frequently the filling of the physical log triggers checkpoints.

You can use onstat -g buf to monitor buffer pool statistics, including the read-cache rate of the buffer pool. This rate represents the percentage of database pages that are already present in a shared-memory buffer when a query requests a page. (If a page is not already present, the database server must copy it into memory from disk.) If the database server finds the page in the buffer pool, it spends less time on disk I/O. Therefore, you want a high read-cache rate for good performance. For OLTP applications where many users read small sets of data, the goal is to achieve a read cache rate of 95 percent or better. If the buffer pool is extendable, you can specify the read cache hit ratio below which the database server extends the buffer pool.

If the read-cache rate is low, you can repeatedly increase buffers and restart the database server. As you increase the BUFFERPOOL value of buffers, you reach a point at which increasing the value no longer produces significant gains in the read-cache rate, or you reach the upper limit of your operating-system shared-memory allocation.

Use the memory-management monitor utility in your operating system (such as vmstat or sar on UNIX™) to note the level of page scans and paging-out activity. If these levels rise suddenly or rise to unacceptable levels during peak database activity, reduce the size of the buffer pool.

Smart large objects and buffers

Depending upon your situation, you can take one of the following actions to achieve better performance for applications that use smart large objects:

- If your applications frequently access smart large objects that are 2 KB or 4 KB in size, use the buffer pool to keep them in memory longer. Use the following formula to increase the value of the buffers field:

  ```
  Additional_buffers = numcur_open_lo *
                       (lo_userdata / pagesize)
  ```

  In this formula:
  - *numcur_open_lo* is the number of concurrently opened smart large objects that you can obtain from the onstat -g smb fdd command.
  - *lo_userdata* is the number of bytes of smart-large-object data that you want to buffer.
  - *pagesize* is the default page size in bytes for the computer.

  As a rule, try to have enough buffers to hold two smart-large-object pages for each concurrently open smart large object. The additional page is available for read-ahead purposes.

- Use lightweight I/O buffers in the virtual portion of shared memory.

  Use lightweight I/O buffers only when you read or write smart large objects in operations greater than 8000 bytes and seldom access them. That is, if the read or write function calls read large amounts of data in a single-function invocation, use lightweight I/O buffers.

  When you use lightweight I/O buffers, you can prevent the flood of smart large objects into the buffer pool and leave more buffers available for other data pages that multiple users frequently access.

Related information

BUFFERPOOL configuration parameter on page

Monitor buffers on page

Lightweight I/O for smart large objects on page 130

BUFFERPOOL and its effect on page cleaning on page 149

The DS_TOTAL_MEMORY configuration parameter and memory utilization

The DS_TOTAL_MEMORY configuration parameter places a ceiling on the amount of shared memory that a query can obtain. You can use this parameter to limit the performance impact of large, memory-intensive queries. The higher you set this parameter, the more memory a large query can use, and the less memory is available for processing other queries and transactions.

For OLTP applications, set DS_TOTAL_MEMORY to 20 - 50 percent of the value of SHMTOTAL, in KB. For applications that involve large decision-support (DSS) queries, increase the value of DS_TOTAL_MEMORY to 50 - 80 percent of SHMTOTAL. If you use your database server instance exclusively for DSS queries, set this parameter to 90 percent of SHMTOTAL.

A *quantum unit* is the minimum increment of memory that is allocated to a query. The Memory Grant Manager (MGM) allocates memory to queries in quantum units. The database server uses the value of DS_MAX_QUERIES with the value of DS_TOTAL_MEMORY to calculate a quantum of memory, according to the following formula:

```
quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

The database server can adjust the size of the quantum dynamically when it grants memory. To allow for more simultaneous queries with smaller quanta each, increase the value of the DS_MAX_QUERIES configuration parameter.

Related information

The Memory Grant Manager on page 351

DS_TOTAL_MEMORY configuration parameter on page

Limiting the performance impact of CPU-intensive queries on page 45

Algorithm for determining DS_TOTAL_MEMORY

The database server derives a value for DS_TOTAL_MEMORY if you do not set the DS_TOTAL_MEMORY configuration parameter or if you set this configuration parameter to an inappropriate value.

Whenever the database server changes the value that you assigned to DS_TOTAL_MEMORY, it sends the following message to your console:

```
DS_TOTAL_MEMORY recalculated and changed from old_value Kb
            to new_value Kb
```

The variable *old_value* represents the value that you assigned to DS_TOTAL_MEMORY in your configuration file. The variable *new_value* represents the value that the database server derived.

When you receive the preceding message, you can use the algorithm to investigate what values the database server considers inappropriate. You can then take corrective action based on your investigation.

The following sections document the algorithm that the database server uses to derive the new value for DS_TOTAL_MEMORY.

Deriving a minimum for decision-support memory

In the first part of the algorithm that the database server uses to derive the new value for the DS_TOTAL_MEMORY configuration parameter, the database server establishes a minimum amount for decision-support memory.

When you assign a value to the DS_MAX_QUERIES configuration parameter, the database server sets the minimum amount of decision-support memory according to the following formula:

```
min_ds_total_memory = DS_MAX_QUERIES * 128 kilobytes
```

When you do not assign a value to the DS_MAX_QUERIES configuration parameter, the database server uses the following formula instead, which is based on the value of information in the VPCLASS configuration parameter:

```
min_ds_total_memory = NUMBER_CPUVPS * 2 * 128 kilobytes
```

Deriving a working value for decision-support memory

In the second part of the algorithm that the database server uses to derive the new value for the DS_TOTAL_MEMORY configuration parameter, the database server establishes a working value for the amount of decision-support memory.

The database server verifies this amount in the third and final part of the algorithm.

When the DS_TOTAL_MEMORY configuration parameter is set

When the DS_TOTAL_MEMORY configuration parameter is set, the database server checks whether the SHMTOTAL configuration parameter is set and then determines which formula to use to calculate the amount of decision-support memory.

When SHMTOTAL is set, the database server uses the following formula to calculate the amount of decision-support memory:

```
IF DS_TOTAL_MEMORY <= SHMTOTAL - nondecision_support_memory THEN
   decision_support_memory = DS_TOTAL_MEMORY
ELSE
    decision_support_memory = SHMTOTAL -
                  nondecision_support_memory
```

This algorithm effectively prevents you from setting DS_TOTAL_MEMORY to values that the database server cannot possibly allocate to decision-support memory.

When SHMTOTAL is not set, the database server sets decision-support memory equal to the value that you specified in DS_TOTAL_MEMORY.

Related information

DS_TOTAL_MEMORY configuration parameter on page

When the DS_TOTAL_MEMORY configuration parameter is not set

When the DS_TOTAL_MEMORY configuration parameter is not set, the database server uses other sources to calculate a value for the amount of decision-support memory.

When SHMTOTAL is set, the database server uses the following formula to calculate the amount of decision-support memory:

```
decision_support_memory = SHMTOTAL –
                    nondecision_support_memory
```

When the database server finds that you did not set SHMTOTAL, it sets decision-support memory as in the following example:

```
decision_support_memory = min_ds_total_memory
```

For a description of the variable min_ds_total_memory, see Deriving a minimum for decision-support memory on page 75.

Checking the derived value for decision-support memory

In the final part of the algorithm that the database server uses to derive the new value for the DS_TOTAL_MEMORY configuration parameter, the database server verifies that the amount of shared memory is greater than *min_ds_total_memory* and less than the maximum possible memory space for your computer.

When the database server finds that the derived value for decision-support memory is less than the value of the *min_ds_total_memory* variable, it sets decision-support memory equal to the value of *min_ds_total_memory*.

When the database server finds that the derived value for decision-support memory is greater than the maximum possible memory space for your computer, it sets decision-support memory equal to the maximum possible memory space.

The LOGBUFF configuration parameter and memory utilization

The LOGBUFF configuration parameter determines the amount of shared memory that is reserved for each of the three buffers that hold the logical-log records until they are flushed to the logical-log file on disk. The size of a buffer determines how often it fills and therefore how often it must be flushed to the logical-log file on disk.

If you log smart large objects, increase the size of the logical-log buffers to prevent frequent flushing to the logical-log file on disk.

Related reference

Configuration parameters that affect critical data on page 113

Related information

LOGBUFF configuration parameter on page

The LOW_MEMORY_RESERVE configuration parameter and memory utilization

The LOW_MEMORY_RESERVE configuration parameter reserves a specific amount of memory, in kilobytes, for the database server to use when critical activities are needed and the server has limited free memory.

If you enable the new LOW_MEMORY_RESERVE configuration parameter by setting it to a specified value in kilobytes, critical activities, such as rollback activities, can complete even when you receive out-of-memory errors.

Related information

LOW_MEMORY_RESERVE configuration parameter on page

onstat -g seg command: Print shared memory segment statistics on page

The PHYSBUFF configuration parameter and memory utilization

The PHYSBUFF configuration parameter determines the amount of shared memory that is reserved for each of the two buffers that serve as temporary storage space for data pages that are about to be modified. The size of a buffer determines how often it fills and therefore how often it must be flushed to the physical log on disk.

Choose a value for PHYSBUFF that is an even increment of the system page size.

Related information

PHYSBUFF configuration parameter on page

The LOCKS configuration parameter and memory utilization

The LOCKS configuration parameter specifies the initial size of the lock table. The lock table holds an entry for each lock that a session uses. Each lock uses 120 bytes within a lock table. You must provide for this amount of memory when you configure shared memory.

If the number of locks needed by sessions exceeds the value set in the LOCKS configuration parameter, the database server attempts to increase the lock table by doubling its size. Each time that the lock table overflows (when the number of locks needed is greater than the current size of the lock table), the database server increases the size of the lock table, up to 99 times. Each time that the database server increases the size of the lock table, the server attempts to double its size. However, the server will limit each actual increase to no more than the maximum number of added locks shown in Table 7: Maximum number of locks on 32-bit and 64-bit platforms on page 78. After the 99th time that the database server increases the lock table, the server no longer increases the size of the lock table, and an application needing a lock receives an error.

The following table shows the maximum number of locks allowed on 32-bit and 64-bit platforms

Table 7. Maximum number of locks on 32-bit and 64-bit platforms

Platform	Maximum Number of Initial Locks	Maximum Number of Dynamic Lock Table Extensions	Maximum Number of Locks Added Per Lock Table Extension	Maximum Number of Locks Allowed
32-bit	8,000,000	99	100,000	8,000,000 + (99 x 100,000)
64-bit	500,000,000	99	1,000,000	500,000,000 + (99 x 1,000,000)

The default value for the LOCKS configuration parameter is 20,000.

To estimate a different value for the LOCKS configuration parameter, estimate the maximum number of locks that a query needs and multiply this estimate by the number of concurrent users. You can use the guidelines in the following table to estimate the number of locks that a query needs.

Locks per Statement	Isolation Level	Table	Row	Key	TEXT or BYTE Data	CLOB or BLOB Data
SELECT	Dirty Read	0	0	0	0	0
SELECT	Committed Read	1	0	0	0	0
SELECT	Cursor Stability	1	1	0	0	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range
SELECT	Indexed Repeatable Read	1	Number of rows that satisfy conditions	Number of rows that satisfy conditions	0	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range
SELECT	Sequential Repeatable Read	1	0	0	0	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range
INSERT	Not applicable	1	1	Number of indexes	Number of pages in TEXT or BYTE data	1 lock for the CLOB or BLOB value
DELETE	Not applicable	1	1	Number of indexes	Number of pages in TEXT or BYTE data	1 lock for the CLOB or BLOB value

Locks per Statement	Isolation Level	Table	Row	Key	TEXT or BYTE Data	CLOB or BLOB Data
UPDATE	Not applicable	1	1	2 per changed key value	Number of pages in old plus new TEXT or BYTE data	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range

Important: During the execution of the SQL statement DROP DATABASE, the database server acquires and holds a lock on each table in the database until the entire DROP operation completes. Make sure that the value for LOCKS is large enough to accommodate the largest number of tables in a database.

Related information

Configuring and managing lock usage on page 251

LOCKS configuration parameter on page

The RESIDENT configuration parameter and memory utilization

The RESIDENT configuration parameter specifies whether shared-memory residency is enforced for the resident portion of database server shared memory. This configuration parameter works only on computers that support forced residency.

The resident portion in the database server contains the buffer pools that are used for database read and write activity. Performance improves when these buffers remain in physical memory.

You should set the RESIDENT parameter to `1`. If forced residency is not an option on your computer, the database server issues an error message and ignores this configuration parameter.

On machines that support 64-bit addressing, you can have a very large buffer pool and the virtual portion of database server shared memory can also be very large. The virtual portion contains various memory caches that improve performance of multiple queries that access the same tables (see Configure and monitor memory caches on page 83). To make the virtual portion resident in physical memory in addition to the resident portion, set the RESIDENT parameter to `-1`.

If your buffer pool is very large, but your physical memory is not very large, you can set RESIDENT to a value greater than 1 to indicate the number of memory segments to stay in physical memory. This specification makes only a subset of the buffer pool resident.

You can turn residency on or off for the resident portion of shared memory in the following ways:

- Use the **onmode** utility to reverse temporarily the state of shared-memory residency while the database server is online.
- Change the RESIDENT parameter to turn shared-memory residency on or off the next time that you start database server shared memory.

Related information

RESIDENT configuration parameter on page

The SHMADD and EXTSHMADD configuration parameters and memory utilization

The SHMADD configuration parameter specifies the size of each increment of shared memory that the database server dynamically adds to the virtual portion. The EXTSHMADD configuration parameter specifies the size of a virtual-extension segment that is added when user-defined routines or DataBlade® routines run in user-defined virtual processors. Trade-offs are involved in determining the size of an increment.

Adding shared memory uses CPU cycles. The larger each increment, the fewer increments are required, but less memory is available for other processes. Adding large increments is generally preferred; but when memory is heavily loaded (the scan rate or paging-out rate is high), smaller increments allow better sharing of memory resources among competing programs.

The range of values for SHMADD is `1024` through `4294967296` KB for a 64-bit operating system and `1024` through `524288` KB for a 32-bit operating system. The following table contains recommendations for setting SHMADD according to the size of physical memory.

Memory Size	SHMADD Value
256 MB or less	8192 KB (the default)
257 - 512 MB	16,384 KB
Larger than 512 MB	32,768 KB

The range of values for EXTSHMADD is the same as the range of values of SHMADD.

Note: A shared memory segment can be as large as 4 terabytes, depending on platform limits and the value of the SHMMAX kernel parameter. Use the onstat -g seg command to display the number of shared-memory segments that the database server is currently using.

Related information

SHMADD configuration parameter on page

EXTSHMADD configuration parameter on page

Configuring UNIX shared memory on page 67

The SHMTOTAL configuration parameter and memory utilization

The SHMTOTAL configuration parameter places an absolute upper limit on the amount of shared memory that an instance of the database server can use.

If the SHMTOTAL configuration parameter is set to `0` or left unassigned, the database server continues to attach additional shared memory as needed until no virtual memory is available on the system.

You can usually set the SHMTOTAL configuration parameter to `0`, except in the following cases:

- You must limit the amount of virtual memory that the database server uses for other applications or other reasons.
- Your operating system runs out of swap space and performs abnormally. In this case, you can set SHMTOTAL to a value that is a few megabytes less than the total swap space that is available on your computer.
- You are using automatic low memory management.

Related information

SHMTOTAL configuration parameter on page

The SHMVIRTSIZE configuration parameter and memory utilization

The SHMVIRTSIZE parameter specifies the size of the virtual portion of shared memory to allocate when you start the database server. The virtual portion of shared memory holds session- and request-specific data as well as other information.

Although the database server adds increments of shared memory to the virtual portion as needed to process large queries or peak loads, allocation of shared memory increases time for transaction processing. Therefore, you should set SHMVIRTSIZE to provide a virtual portion large enough to cover your normal daily operating requirements. The size of SHMVIRTSIZE can be as large the SHMMAX configuration parameter allows.

The maximum value of SHMVIRTSIZE, which must be a positive integer, is:

- 4 terabytes on a 64-bit database server
- 2 gigabytes on a 32-bit database server

For an initial setting, it is suggested that you use the larger of the following values:

- `8000`
- *connections* `* 350`

The *connections* variable is the number of connections for all network types that are specified in the sqlhosts information by one or more NETTYPE configuration parameters. (The database server uses *connections* `* 200` by default.)

Once system utilization reaches a stable workload, you can reconfigure a new value for SHMVIRTSIZE. As noted in Freeing shared memory with onmode -F on page 68, you can instruct the database server to release shared-memory segments that are no longer in use after a peak workload or large query.

Related information

SHMVIRTSIZE configuration parameter on page

The SHMVIRT_ALLOCSEG configuration parameter and memory utilization

The SHMVIRT_ALLOCSEG configuration parameter specifies a threshold at which the database server should allocate memory. This configuration parameter also defines an alarm event security-code that is activated if the server cannot allocate the new memory segment, thus ensuring that the database server never runs out of memory.

When you set the SHMVIRT_ALLOCSEG configuration parameter, you must:

- Specify the percentage of memory used or the whole number of kilobytes remaining on the server. You cannot use negative values and values between `0` and `.39`.
- Specify the alarm event-security code, which is a value ranging from `1` (not noteworthy) to `5` (fatal). If you do not specify an event-security code, the server sets the value to `3`, which is the default value.

Example 1:

```
SHMVIRT_ALLOCSEG 3000, 4
```

This specifies that if the database serve has 3000 kilobytes remaining in virtual memory and additional kilobytes of memory cannot be allocated, the server raises an alarm level of 4.

Example 2:

```
SHMVIRT_ALLOCSEG .8, 4
```

This specifies that if the database server has twenty percent remaining in virtual memory and additional kilobytes of memory cannot be allocated, the server raises an alarm level of 4.

Related information

Event Alarm Parameters on page

SHMVIRT_ALLOCSEG configuration parameter on page

The STACKSIZE configuration parameter and memory utilization

The STACKSIZE configuration parameter indicates the initial stack size for each thread. The database server assigns the amount of space that this parameter indicates to each active thread. This space comes from the virtual portion of database server shared memory. You can reduce the amount of shared memory that the database server adds dynamically.

To reduce the amount of shared memory that the database server adds dynamically, estimate the amount of the stack space required for the average number of threads that your system runs and include that amount in the value that you set for the SHMVIRTSIZE configuration parameter.

To estimate the amount of stack space that you require, use the following formula:

```
stacktotal = STACKSIZE * avg_no_of_threads
```

avg_no_of_threads

is the average number of threads. You can monitor the number of active threads at regular intervals to determine this amount. Use **onstat -g sts** to check the stack use of threads. A general estimate is between 60

and 70 percent of the total number of connections (specified in the NETTYPE parameters in your ONCONFIG file), depending on your workload.

The database server also executes user-defined routines (UDRs) with user threads that use this stack. Programmers who write user-defined routines should take the following measures to avoid stack overflow:

- Do not use large automatic arrays.
- Avoid excessively deep calling sequences.
- *For DB-Access only:* Use **mi_call** to manage recursive calls.

If you cannot avoid stack overflow with these measures, use the STACK modifier of the CREATE FUNCTION statement to increase the stack for a particular routine.

Related information

STACKSIZE configuration parameter on page

Configure and monitor memory caches

The database server uses caches to store information in memory instead of performing a disk read or another operation to obtain the information. These memory caches improve performance for multiple queries that access the same tables. You can set some configuration parameters to increase the effectiveness of each cache. You can view information about memory caches by running onstat commands.

The following table lists the main memory caches that have the greatest effect on performance and how to configure and monitor those caches.

Table 8. Main memory caches

Cache Name	Cache Description	Configuration Parameters	onstat command
Data Dictionary	Stores information about the table definition (such as column names and data types).	DD_HASHSIZE: The maximum number of buckets in the cache. DD_HASHMAX: The number of tables in each bucket	onstat -g dic
Data Distribution	Stores distribution statistics for a column.	DS_POOLSIZE: The maximum number of entries in the cache. DS_HASHSIZE: The number of buckets in the cache.	onstat -g dsc
SQL Statement	Stores parsed and optimized SQL statements.	STMT_CACHE: Enable the SQL statement cache.	onstat -g ssc

Table 8. Main memory caches (continued)

Cache Name	Cache Description	Configuration Parameters	onstat command
		STMT_CACHE_HITS: The number of times anSQL statement is run before it is cached. STMT_CACHE_NOLIMIT: Prohibit entries into the SQL statement cache when allocated memory exceeds the value of the STMT_CACHE_SIZE configuration parameter. STMT_CACHE_NUMPOOL: The number of memory pools for the SQL statement cache. STMT_CACHE_SIZE: The size of the SQL statement cache, in KB.	
UDR	Stores frequently used user-defined routines and SPL routines.	PC_POOLSIZE: The maximum number of user-defined routines and SPL routines in the cache. PC_HASHSIZE: The number of buckets in the UDR cache.	onstat -g prc

The following table lists more memory caches and how to configure and monitor those caches.

Table 9. Additional memory caches

Cache Name	Cache Description	Configuration Parameters	onstat command
Access method	Stores user-defined access methods.	None.	onstat -g cac am
Aggregate	Stores user-defined aggregates.	DS_POOLSIZE DS_HASHSIZE	onstat -g cac agg
Cast	Stores user-defined casts.	DS_POOLSIZE DS_HASHSIZE	onstat -g cac cast
External directives	Stores external directives.	None.	onstat -g cac ed

Table 9. Additional memory caches (continued)

Cache Name	Cache Description	Configuration Parameters	onstat command
LBAC security policy information	Stores LBAC security policies.	PLCY_POOLSIZE PLCY_HASHSIZE	onstat -g cac lbacplcy
LBAC credential memory	Stores LBAC credentials.	USRC_POOLSIZE USRC_HASHSIZE	onstat -g cac lbacusrc
Operator class instance	Stores user-defined operator classes.	DS_POOLSIZE DS_HASHSIZE	onstat -g cac opci
Procedure name	Stores user-defined routine and SPL routine names.	PC_POOLSIZE PC_HASHSIZE	onstat -g cac prn
Routine resolution	Stores user-defined routine resolution information.	DS_POOLSIZE DS_HASHSIZE	onstat -g cac rr
Secondary transient	Stores transient unnamed complex data types on secondary servers in a high-availability cluster.	DS_POOLSIZE DS_HASHSIZE	onstat -g cac ttype
Extended type ID	Stores the IDs of user-defined types.	DS_POOLSIZE DS_HASHSIZE	onstat -g cac typei
Extended type name	Stores the name of user-defined types.	DS_POOLSIZE DS_HASHSIZE	onstat -g cac typen

Related information

SPL routine executable format stored in UDR cache on page 325

onstat -g cac command: Print information about caches on page

onstat -g dsc command: Print distribution cache information on page

onstat -g prc command: Print sessions using UDR or SPL routines on page

onstat -g ssc command: Print SQL statement occurrences on page

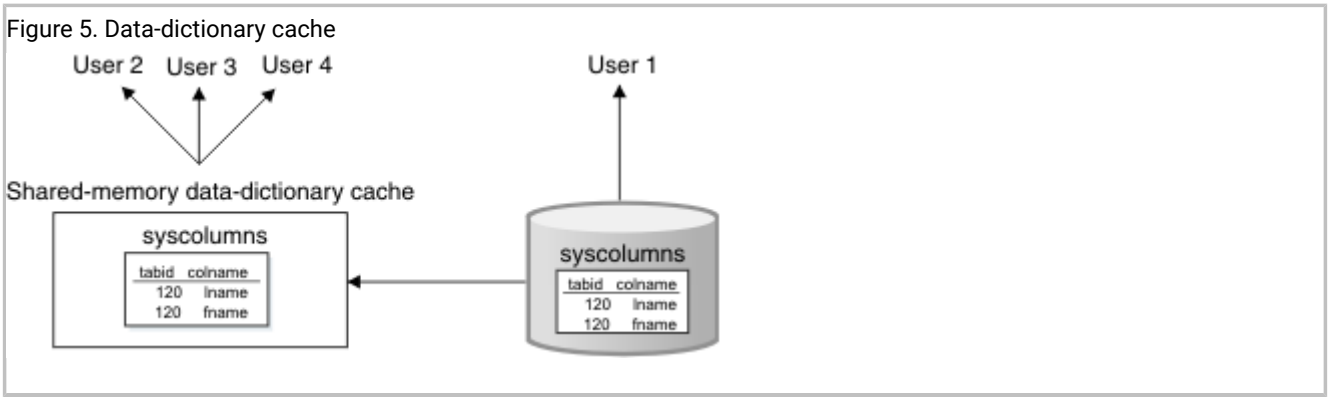
Database configuration parameters on page

Data-dictionary cache

The first time that the database server accesses a table, it retrieves the information that it needs about the table (such as the column names and data types) from the system catalog tables on disk. After the database server has accessed the table, it places that information in the data-dictionary cache in shared memory.

Figure 5: Data-dictionary cache on page 86 shows how the database server uses this cache for multiple users. User 1 accesses the column information for **tabid 120** for the first time. The database server puts the column information in the data-dictionary cache. When user 2, user 3 and user 4 access the same table, the database server does not have to read from disk to access the data-dictionary information for the table. Instead, it reads the dictionary information from the data-dictionary cache in memory.

Figure 5. Data-dictionary cache



The database server still places pages for system catalog tables in the buffer pool, as it does all other data and index pages. However, the data-dictionary cache offers an additional performance advantage, because the data-dictionary information is organized in a more efficient format and organized to allow fast retrieval.

Data-dictionary configuration

The database server uses a hashing algorithm to store and locate information within the data-dictionary cache. The DD_HASHSIZE and DD_HASHMAX configuration parameters control the size of the data-dictionary cache.

To modify the number of buckets in the data-dictionary cache, use DD_HASHSIZE (must be a prime number). To modify the number of tables that can be stored in one bucket, use DD_HASHMAX.

For medium to large systems, you can start with the following values for these configuration parameters:

- DD_HASHSIZE: `503`
- DD_HASHMAX: `4`

With these values, you can potentially store information about 2012 tables in the data-dictionary cache, and each hash bucket can have a maximum of 4 tables.

If the bucket reaches the maximum size, the database server uses a least recently used mechanism to clear entries from the data dictionary.

Related information

DD_HASHSIZE configuration parameter on page

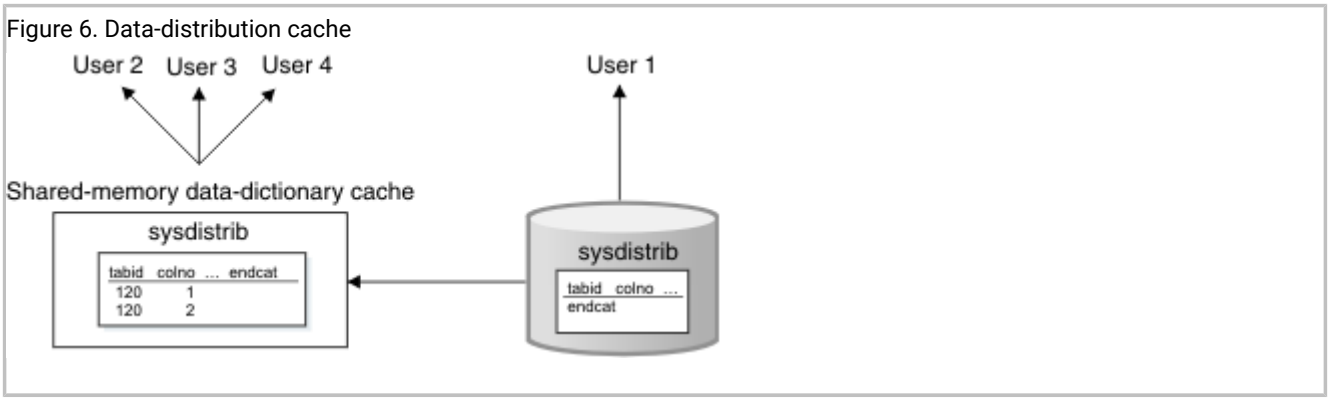
DD_HASHMAX configuration parameter on page

Data-distribution cache

The query optimizer uses distribution statistics generated by the UPDATE STATISTICS statement in the MEDIUM or HIGH mode to determine the query plan with the lowest cost. The first time that the optimizer accesses the distribution statistics for a column, the database server retrieves the statistics from the **sysdistrib** system catalog table on disk and places that information in the data-distribution cache in memory.

Figure 6: Data-distribution cache on page 87 shows how the database server accesses the data-distribution cache for multiple users. When the optimizer accesses the column distribution statistics for User 1 for the first time, the database server puts the distribution statistics in the data-distribution cache. When the optimizer determines the query plan for user 2, user 3 and user 4 who access the same column, the database server does not have to read from disk to access the data-distribution information for the table. Instead, it reads the distribution statistics from the data-distribution cache in shared memory.

Figure 6. Data-distribution cache



The database server initially places pages for the **sysdistrib** system catalog table in the buffer pool as it does all other data and index pages. However, the data-distribution cache offers additional performance advantages. It:

- Is organized in a more efficient format
- Is organized to allow fast retrieval
- Bypasses the overhead of the buffer pool management
- Frees more pages in the buffer pool for actual data pages rather than system catalog pages
- Reduces I/O operations to the system catalog table

Data-distribution configuration

The database server uses a hashing algorithm to store and locate information within the data-distribution cache. The DS_POOLSIZE configuration parameter controls the size of the data-distribution cache and controls the total number of column distributions that can be stored in the data-distribution cache. The value of the DS_POOLSIZE configuration parameter represents half of the maximum number of distributions in the data distribution cache.

To modify the number of buckets in the data-distribution cache, use the DS_HASHSIZE configuration parameter.

The following formula determines the number of column distributions that can be stored in one bucket.

```
Distributions_per_bucket = DS_POOLSIZE / DS_HASHSIZE
```

To modify the number of distributions per bucket, change either the DS_POOLSIZE or DS_HASHSIZE configuration parameter.

For example, with the default values of `127` for DS_POOLSIZE and `31` for DS_HASHSIZE, you can potentially store distributions for about 254 columns in the data-distribution cache. When the cache is full, the database server automatically increases the size of the cache by 10%.

For example, with the default values of `127` for DS_POOLSIZE and `31` for DS_HASHSIZE, you can potentially store distributions for about 127 columns in the data-distribution cache. The cache has 31 hash buckets, and each hash bucket can have an average of 4 entries.

The values that you set for DS_HASHSIZE and DS_POOLSIZE, depend on the following factors:

- The number of columns for which you run the UPDATE STATISTICS statement in HIGH or MEDIUM mode and you expect to be used most often in frequently run queries.

  If you do not specify columns when you run UPDATE STATISTICS for a table, the database server generates distributions for all columns in the table.

  You can use the values of DD_HASHSIZE and DD_HASHMAX as guidelines for DS_HASHSIZE and DS_POOLSIZE. The DD_HASHSIZE and DD_HASHMAX specify the size for the data-dictionary cache, which stores information and statistics about tables that queries access.

  For medium to large systems, you can start with the following values:

  - DD_HASHSIZE 503
  - DD_HASHMAX 4
  - DS_HASHSIZE 503
  - DS_POOLSIZE 1000
  - DS_POOLSIZE 2000

  Monitor these caches by running the onstat -g dsc command to see the actual usage, and you can adjust these parameters accordingly.

- The amount of memory available

  The amount of memory that is required to store distributions for a column depends on the level at which you run UPDATE STATISTICS. Distributions for a single column might require between 1 KB and 2 MB, depending on whether you specify medium or high mode or enter a finer resolution percentage when you run UPDATE STATISTICS.

If the size of the data-distribution cache is too small, the following performance problems can occur:

- The database server uses the DS_POOLSIZE value to determine when to remove entries from the data-distribution cache. However, if the optimizer needs the dropped distributions for another query, the database server must reaccess them from the **sysdistrib** system catalog table on disk. The additional I/O and buffer pool operations to access **sysdistrib** on disk adds to the total response time of the query.

  The database server tries to maintain the number of entries in data-distribution cache at the DS_POOLSIZE value. If the total number of entries reaches within an internal threshold of DS_POOLSIZE, the database server uses a least recently used mechanism to remove entries from the data-distribution cache. The number of entries in a hash bucket can go past this DS_POOLSIZE value, but the database server eventually reduces the number of entries when memory requirements drop.
- If DS_HASHSIZE is small and DS_POOLSIZE is large, overflow lists can be long and require more search time in the cache.

  Overflow occurs when a hash bucket already contains an entry. When multiple distributions hash to the same bucket, the database server maintains an overflow list to store and retrieve the distributions after the first one.

  If DS_HASHSIZE and DS_POOLSIZE are approximately the same size, the overflow lists might be smaller or even nonexistent, which might waste memory. However, the amount of unused memory is insignificant overall.

You might want to change the values of the DS_HASHSIZE and DS_POOLSIZE configuration parameters if you see the following situations:

- If the data-distribution cache is full most of the time and commonly used columns are not listed in the **distribution name** field, try increasing the values of the DS_HASHSIZE and DS_POOLSIZE configuration parameters.
- If the total number of entries is much lower than the value of the DS_POOLSIZE configuration parameter, you can reduce the values of the DS_HASHSIZE and DS_POOLSIZE configuration parameters.
- If the number of hits are not evenly distributed among hash lists, increase the number of hash lists by increasing the value of the DS_HASHSIZE configuration parameter. Adjust the number of hash lists to have the least number of high hit entries per hash list.

Related information

DD_HASHSIZE configuration parameter on page

DD_HASHMAX configuration parameter on page

DS_POOLSIZE configuration parameter on page

onstat -g dsc command: Print distribution cache information on page

Monitor and tune the SQL statement cache

The SQL statement cache stores optimized SQL statements so that multiple users who run the same SQL statement can achieve some performance improvements.

These performance improvements are:

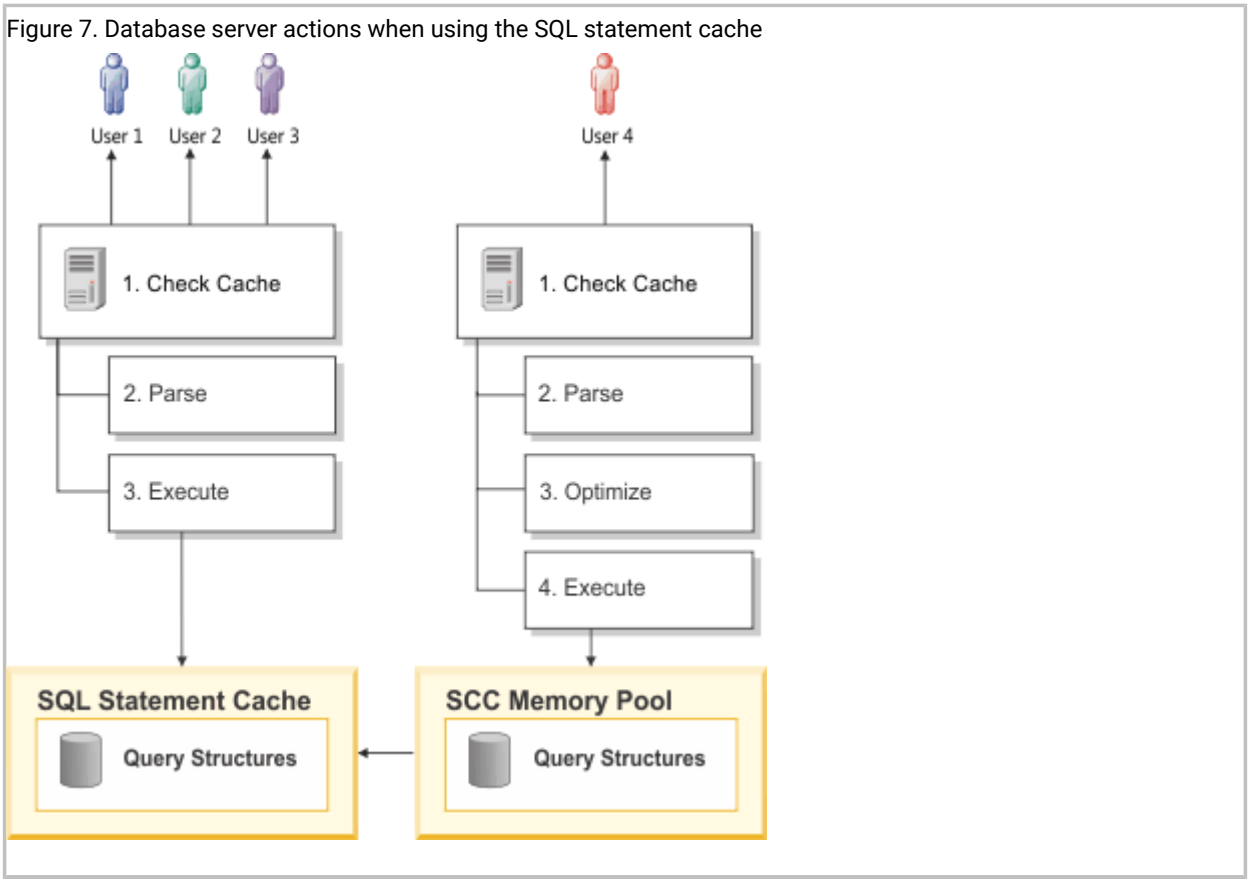
- Reduced response times because they bypass the optimization step, as Figure 7: Database server actions when using the SQL statement cache on page 90 shows
- Reduced memory usage because the database server shares query data structures among users

For more information about the effect of the SQL statement cache on the performance of individual queries, see Optimize queries with the SQL statement cache on page 420.

Figure 7: Database server actions when using the SQL statement cache on page 90 shows how the database server accesses the SQL statement cache for multiple users.

- When the database server runs an SQL statement for User 1 for the first time, the database server checks whether the same exact SQL statement is in the SQL statement cache. If it is not in the cache, the database server parses the statement, determines the optimal query plan, and runs the statement.
- When User 2 runs the same SQL statement, the database server finds the statement in the SQL statement cache and does not optimize the statement.
- Similarly, if User 3 and User 4 run the same SQL statement, the database server does not optimize the statement. Instead, it uses the query plan in the SQL statement cache in memory.

Figure 7. Database server actions when using the SQL statement cache



Prepared statements and the statement cache

Prepared statements are inherently cached for a single session. This means that if a prepared statement is executed many times or if a single cursor is opened many times, the session uses the same prepared query plan.

If a session prepares a statement and then executes it many times, the SQL statement cache does not affect performance, because the statement is optimized just once during the PREPARE statement.

However, if other sessions also prepare that same statement, or if the first session prepares the statement several times, the statement cache usually provides a direct performance benefit, because the database server only calculates the query plan once. Of course, the original session might gain a small benefit from the statement cache, even if it only prepares the statement once, because other sessions use less memory, and the database server does less work for the other sessions

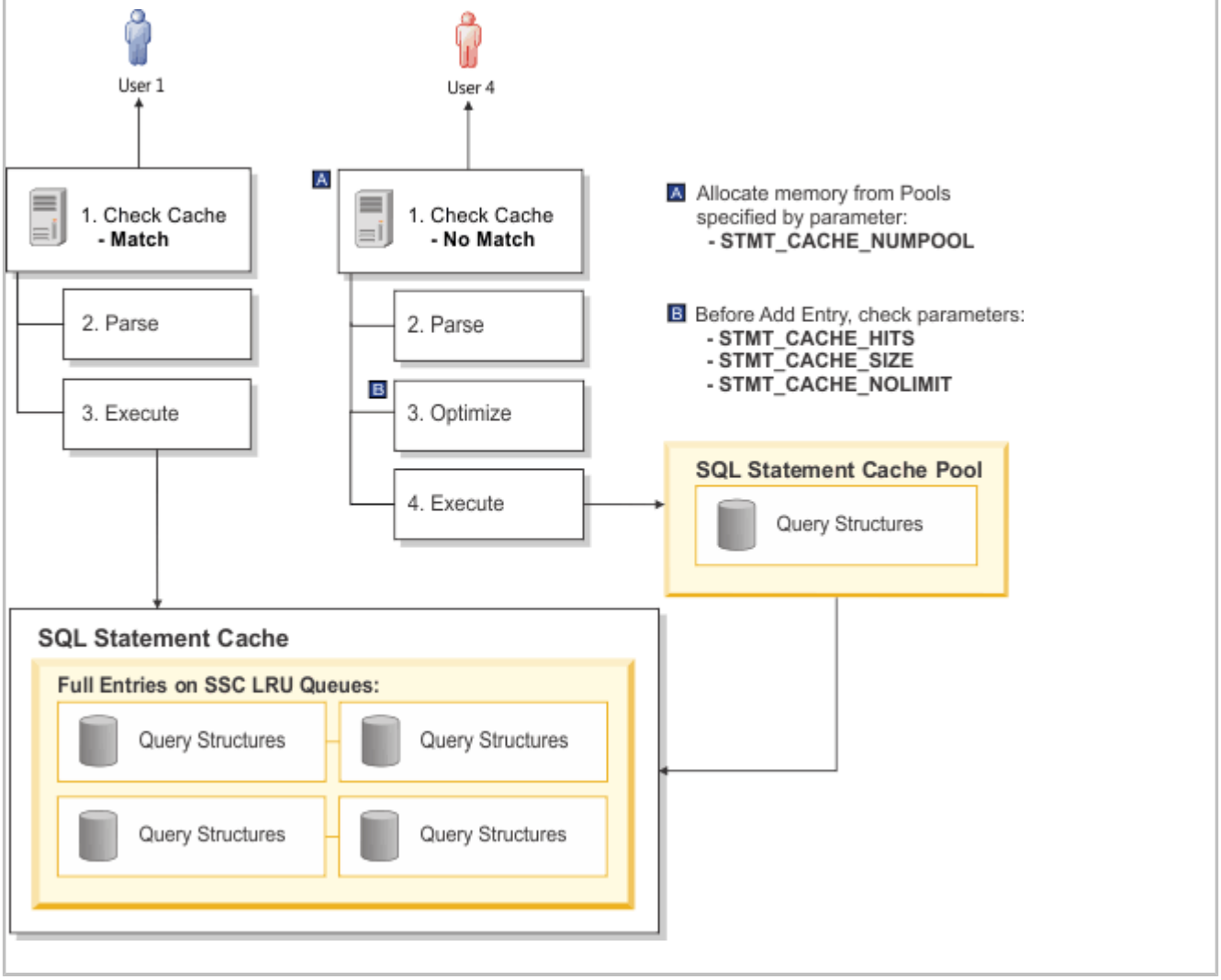
SQL statement cache configuration

The value of the STMT_CACHE configuration parameter enables or disables the SQL statement cache.

For more information about how the value of the STMT_CACHE configuration parameter enables the SQL statement cache, see Enabling the SQL statement cache on page 422 describes.

Figure 8: Configuration parameters that affect the SQL statement cache on page 92 shows how the database server uses the values of the pertinent configuration parameters for the SQL statement cache. Further explanation follows the figure.

Figure 8. Configuration parameters that affect the SQL statement cache



When the database server uses the SQL statement cache for a user, it means the database server takes the following actions:

- Checks the SQL statement cache first for a match of the SQL statement that the user is executing
- If the SQL statement matches an entry, executes the statement using the query memory structures in the SQL statement cache (User 2 in Figure 8: Configuration parameters that affect the SQL statement cache on page 92)
- If the SQL statement does not match an entry, the database server checks if it qualifies for the cache.

  For information about what qualifies an SQL statement for the cache, see SQL statement cache qualifying criteria on page .
- If the SQL statement qualifies, inserts an entry into the cache for subsequent executions of the statement.

The following parameters affect whether or not the database server inserts the SQL statement into the cache (User 1 in Figure 8: Configuration parameters that affect the SQL statement cache on page 92):

- STMT_CACHE_HITS specifies the number of times the statement executes with an entry in the cache (referred to as *hit count*). The database server inserts one of the following entries, depending on the hit count:
  - If the value of STMT_CACHE_HITS is 0, inserts a fully cached entry, which contains the text of the SQL statement plus the query memory structures
  - If the value of STMT_CACHE_HITS is not 0 and the statement does not exist in the cache, inserts a key-only entry that contains the text of the SQL statement. Subsequent executions of the SQL statement increment the hit count.
  - If the value of STMT_CACHE_HITS is equal to the number of hits for a key-only entry, adds the query memory structures to make a fully cached entry.
- STMT_CACHE_SIZE specifies the size of the SQL statement cache, and STMT_CACHE_NOLIMIT specifies whether or not to limit the memory of the cache to the value of STMT_CACHE_SIZE. If you do not specify the STMT_CACHE_SIZE parameter, it defaults to 524288 (512 * 1024) bytes.

  The default value for STMT_CACHE_NOLIMIT is `1`, which means the database server will insert entries into the SQL statement cache even though the total amount of memory might exceed the value of STMT_CACHE_SIZE.

  When STMT_CACHE_NOLIMIT is set to `0`, the database server inserts the SQL statement into the cache if the current size of the cache will not exceed the memory limit.

The following sections on STMT_CACHE_HITS, STMT_CACHE_SIZE, STMT_CACHE_NOLIMIT, STMT_CACHE_NUMPOOL and provide more details on how the following configuration parameters affect the SQL statement cache and reasons why you might want to change their default values.

Number of SQL statement executions

When the SQL statement cache is enabled, the database server inserts a qualified SQL statement and its memory structures immediately in the SQL statement cache by default.

If your workload has a disproportionate number of ad hoc queries, use the STMT_CACHE_HITS configuration parameter to specify the number of times an SQL statement is executed before the database server places a fully cached entry in the statement cache.

When the STMT_CACHE_HITS configuration parameter is greater than `0` and the number of times the SQL statement has been executed is less than STMT_CACHE_HITS, the database server inserts key-only entries in the cache. This specification minimizes unshared memory structures from occupying the statement cache, which leaves more memory for SQL statements that applications use often.

Monitor the number of hits on the SQL statement cache to determine if your workload is using this cache effectively. The following sections describe ways to monitor the SQL statement cache hits.

Related information

STMT_CACHE_HITS configuration parameter on page

Too many single-use queries in the SQL statement cache on page 98

Monitoring the number of hits on the SQL statement cache

To monitor the number of hits in the SQL statement cache, run the onstat -g ssc command.

About this task

The onstat -g ssc command displays fully cached entries in the SQL statement cache. Figure 9: onstat -g ssc output on page 94 shows sample output for onstat -g ssc.

Figure 9. onstat -g ssc output

```
onstat -g ssc

Statement Cache Summary:
#lrus   currsize  maxsize   Poolsize  #hits   nolimit
4       49456     524288    57344     0       1

Statement Cache Entries:

lru hash ref_cnt hits flag heap_ptr      database           user
------------------------------------------------------------------------------
  0  153       0    0   -F  a7e4690      vjp_stores         virginia
  SELECT * FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
    AND order_date > "01/01/07"

  1  259       0    0    -F  aa58c20     vjp_stores         virginia
  SELECT * FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
    AND order_date > "01/01/2007"

  2  232       0    1    DF  aa3d020     vjp_stores         virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num

  3  232       1    1   -F  aa8b020     vjp_stores         virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num



    Total number of entries: 4.
```

To monitor the number of times that the database server reads the SQL statement within the cache, look at the following output columns:

- In the `Statement Cache Summary` portion of the onstat -g ssc output, the **#hits** column is the value of the SQL_STMT_HITS configuration parameter.

  In Figure 9: onstat -g ssc output on page 94, the **#hits** column in the Statement Cache Summary portion of the output has a value of `0`, which is the default value of the STMT_CACHE_HITS configuration parameter.

  Important: The database server uses entries in the SQL statement cache only if the statements are exactly the same. The first two entries in Figure 9: onstat -g ssc output on page 94 are not the same because each contains a different literal value in the **order_date** filter.

- In the `Statement Cache Entries` portion of the onstat -g ssc output, the hits column shows the number of times that the database server ran each individual SQL statement from the cache. In other words, the column shows the number of times that the database server uses the memory structures in the cache instead of optimizing the statements to generate them again.

  The first time that it inserts the statement in the cache, the **hits** value is `0`.

  - The first two SQL statements in Figure 9: onstat -g ssc output on page 94 have a **hits** column value of `0`, which indicates that each statement is inserted into the cache but not yet run from the cache.
  - The last two SQL statements in Figure 9: onstat -g ssc output on page 94 have a **hits** column value of `1`, which indicates that these statements ran once from the cache.

  The **hits** value for individual entries indicates how much sharing of memory structures is done. Higher values in the **hits** column indicate that the SQL statement cache is useful in improving performance and memory usage.

For a complete description of the output fields that onstat -g ssc displays, see SQL statement cache information in onstat -g ssc output on page 100.

Determining the number of nonshared entries in the SQL statement cache

To determine how many nonshared entries exist in the SQL statement cache, run **onstat -g ssc all**.

About this task

The **onstat -g ssc all** option displays the key-only entries in addition to the fully cached entries in the SQL statement cache.

To determine how many nonshared entries exist in the cache:

1. Compare the **onstat -g ssc all** output with the **onstat -g ssc** output.
2. If the difference between these two outputs shows that many nonshared entries exist in the SQL statement cache, increase the value of the STMT_CACHE_HITS configuration parameter to allow more shared statements to reside in the cache and reduce the management overhead of the SQL statement cache.

Results

You can use one of the following methods to change the STMT_CACHE_HITS parameter value:

- Update the ONCONFIG file to specify the STMT_CACHE_HITS configuration parameter. You must restart the database server for the new value to take effect.

  You can use a text editor to edit the ONCONFIG file. Then bring down the database server with the **onmode -ky** command and restart with the **oninit** command.

- Increase the STMT_CACHE_HITS configuration parameter dynamically while the database server is running:

  You can use any of the following methods to reset the STMT_CACHE_HITS value at run time:

  - Issue the **onmode -W** command. The following example specifies that three (3) instances are required before a new query is added to the statement cache:

    ```
    onmode –W STMT_CACHE_HITS 2
    ```

  - Call the **ADMIN** or **TASK** function of the SQL administration API. The following example is equivalent to the **onmode** command in the previous example:

    ```
    EXECUTE FUNCTION TASK("ONMODE", "W", "STMT_CACHE_HITS", "2");
    ```

  If you increase STMT_CACHE_HITS dynamically without updating the configuration file, and the database server is subsequently restarted, the STMT_CACHE_HITS setting reverts the value in the ONCONFIG file. Therefore, if you want the setting to persist after subsequent restarts, modify the ONCONFIG file.

Monitoring and tuning the size of the SQL statement cache

If the size of the SQL statement cache is too small, performance problems can occur. You can monitor the effectiveness of the size of the SQL statement cache.

The following performance problems can occur:

- Frequently executed SQL statements are not in the cache

  The statements used most often should remain in the SQL statement cache. If the SQL statement cache is not large enough, the database server might not have enough room to keep these statements when other statements come into the cache. For subsequent executions, the database server must reparse, reoptimize, and reinsert the SQL statement into the cache. Try increasing STMT_CACHE_SIZE.

- The database server spends a lot of time cleaning the SQL statement cache

  The database server tries to prevent the SQL statement cache from allocating large amounts of memory by using a threshold (70 percent of the STMT_CACHE_SIZE parameter) to determine when to remove entries from the SQL statement cache. If the new entry causes the size of the SQL statement cache to exceed the threshold, the database server removes least recently used entries (that are not currently in use) before inserting the new entry.

  However, if a subsequent query needs the removed memory structures, the database server must reparse and reoptimize the SQL statement. The additional processing time to regenerate these memory structures adds to the total response time of the query.

You can set the size of the SQL statement cache in memory with the STMT_CACHE_SIZE configuration parameter. The value of the parameter is the size in kilobytes. If STMT_CACHE_SIZE is not set, the default value is 512 kilobytes.

The **onstat -g ssc** output shows the value of STMT_CACHE_SIZE in the **maxsize** column. In Figure 9: onstat -g ssc output on page 94, this **maxsize** column has a value of `524288`, which is the default value (512 * 1024 = 524288).

Use the **onstat -g ssc** and **onstat -g ssc all** options to monitor the effectiveness of size of the SQL statement cache. If you do not see cache entries for the SQL statements that applications use most, the SQL statement cache might be too small or too many unshared SQL statement occupy the cache. The following sections describe how to determine these situations.

Related information

STMT_CACHE_SIZE configuration parameter on page

Changing the size of the SQL statement cache

You can analyze **onstat -g ssc all** output to determine if the SQL statement cache is too small. If the size of the cache is too small, you can change it.

To determine if the size of the SQL statement cache is too small:

1. Run **onstat -g ssc all** to determine if the cache is too small.
2. Look at the values in the following output columns in the Statement Cache Entries portion of the **onstat -g ssc all** output:
   - The **flags** column shows the current status of an SQL statement in the cache.

     A value of `F` in the second position indicates that the statement is currently fully cached.

     A value of `-` in the second position indicates that only the statement text (key-only entry) is in the cache. Entries with this `-` value in the second position appear in the **onstat -g ssc all** output, but not in the **onstat -g ssc** output.
   - The **hits** column shows the number of times the SQL statement has been executed, excluding the first time it is inserted into the cache.

   If you do not see fully cached entries for statements that applications use most and the value in the **hits** column is large for the entries that do occupy the cache, then the SQL statement cache is too small.

To change the size of the SQL statement cache:

1. Update the value of the STMT_CACHE_SIZE configuration parameter.
2. Restart the database server for the new value to take effect.

Related information

STMT_CACHE_SIZE configuration parameter on page

Too many single-use queries in the SQL statement cache

When the database server places many queries that are only used once in the cache, they might replace statements that other applications use often. You can view **onstat -g ssc all** output to determine if too many unshared SQL statements occupy the cache. If so, you can prevent unshared SQL statements from being fully cached.

Look at the values in the following output columns in the `Statement Cache Entries` portion of the **onstat -g ssc all** output. If you see a lot of entries that have both of the following values, too many unshared SQL statements occupy the cache:

- **flags** column value of `F` in the second position

  A value of `F` in the second position indicates that the statement is currently fully cached.

- **hits** column value of `0` or `1`

  The **hits** column shows the number of times the SQL statement has been executed, excluding the first time it is inserted into the cache.

Increase the value of the STMT_CACHE_HITS configuration parameter to prevent unshared SQL statements from being fully cached.

Related information

STMT_CACHE_HITS configuration parameter on page

Number of SQL statement executions on page 93

Memory limit and size

Although the database server tries to clean the SQL statement cache, sometimes entries cannot be removed because they are currently in use. In this case, the size of the SQL statement cache can exceed the value of the STMT_CACHE_SIZE configuration parameter.

The default value of the STMT_CACHE_NOLIMIT configuration parameter is `1`, which means the database server inserts the statement even though the current size of the cache might be greater than the value of the STMT_CACHE_SIZE parameter.

If the value of the STMT_CACHE_NOLIMIT configuration parameter is `0`, the database server does not insert either a fully-qualified or key-only entry into the SQL statement cache if the size will exceed the value of STMT_CACHE_SIZE.

Use the **onstat -g ssc** option to monitor the current size of the SQL statement cache. Look at the values in the following output columns of the **onstat -g ssc** output:

- The **currsize** column shows the number of bytes currently allocated in the SQL statement cache.

  In Figure 9: onstat -g ssc output on page 94, the **currsize** column has a value of `11264`.

- The **maxsize** column shows the value of STMT_CACHE_SIZE.

  In Figure 9: onstat -g ssc output on page 94, the **maxsize** column has a value of `524288`, which is the default value (512 * 1024 = 524288).

When the SQL statement cache is full and users are currently executing all statements within it, any new SQL statements that a user executes can cause the SQL statement cache to grow beyond the size that STMT_CACHE_SIZE specifies. When the database server is no longer using an SQL statement within the SQL statement cache, it frees memory in the SQL statement cache until the size reaches a threshold of STMT_CACHE_SIZE. However, if thousands of concurrent users are executing several ad hoc queries, the SQL statement cache can grow very large before any statements are removed. In such cases, take one of the following actions:

- Set the STMT_CACHE_NOLIMIT configuration parameter to 0 to prevent insertions when the cache size exceeds the value of the STMT_CACHE_SIZE parameter.
- Set the STMT_CACHE_HITS parameter to a value greater than 0 to prevent caching unshared SQL statements.

You can use one of the following methods to change the STMT_CACHE_NOLIMIT configuration parameter value:

- Update the ONCONFIG file to specify the STMT_CACHE_NOLIMIT configuration parameter. You must restart the database server for the new value to take effect.
- Use the **onmode -W** command to override the STMT_CACHE_NOLIMIT configuration parameter dynamically while the database server is running.

  ```
  onmode -W STMT_CACHE_NOLIMIT 0
  ```

  If you restart the database server, the value reverts the value in the ONCONFIG file. Therefore, if you want the setting to remain for subsequent restarts, modify the ONCONFIG file.

Related information

STMT_CACHE_HITS configuration parameter on page

Multiple SQL statement cache pools

Under some circumstances when the SQL statement cache is enabled, the database server allocates memory from one pool for query structures.

These circumstances are:

- When the database server does not find a matching entry in the cache
- When the database server finds a matching key-only entry in the cache and the hit count reaches the value of the STMT_CACHE_HITS configuration parameter

This one pool can become a bottleneck as the number of users increases. The STMT_CACHE_NUMPOOL configuration parameter allows you to configure multiple **sscpools**.

You can monitor the pools in the SQL statement cache to determine the following situations:

- The number of SQL statement cache pools is sufficient for your workload.
- The size or limit of the SQL statement cache is not causing excessive memory management.

Related information

STMT_CACHE_NUMPOOL configuration parameter on page

Number of SQL statement cache pools

When the SQL statement cache (SSC) is enabled, the database server allocates memory from the SSC pool for unlinked SQL statements. The default value for the STMT_CACHE_NUMPOOL configuration parameter is 1. As the number of users increases, this one SSC pool might become a bottleneck.

The number of longspins on the SSC pool indicates whether or not the SSC pool is a bottleneck.

Use the **onstat -g spi** option to monitor the number of longspins on an SSC pool. The **onstat -g spi** command displays a list of the resources in the system for which a wait was required before a latch on the resource could be obtained. During the wait, the thread spins (or loops), trying to acquire the resource. The **onstat -g spi** output displays the number of times a wait (**Num Waits** column) was required for the resource and the number of total loops (**Num Loops** column). The **onstat -g spi** output displays only resources that have at least one wait.

Figure 10: onstat -g spi output on page 100 shows an excerpt of sample output for **onstat -g spi**. Figure 10: onstat -g spi output on page 100 indicates that no waits occurred for any SSC pool (the **Name** column does not list any SSC pools).

Figure 10. onstat -g spi output

```
Spin locks with waits:
Num Waits   Num Loops   Avg Loop/Wait    Name
34477       387761      11.25            mtcb sleeping_lock
312         10205       32.71            mtcb vproc_list_lock
```

If you see an excessive number of longspins (**Num Loops** column) on an SSC pool, increase the number of SSC pools in the STMT_CACHE_NUMPOOL configuration parameter to improve performance.

Related information

STMT_CACHE_NUMPOOL configuration parameter on page

SQL statement cache information in onstat -g ssc output

The onstat -g ssc command displays summary information for the SQL statement cache.

The **onstat -g ssc** command displays the following information for the SQL statement cache.

Table 10. SQL statement cache information in onstat -g ssc output

Column	Description
#lrus	The number of LRU queues. Multiple LRU queues facilitate concurrent lookup and insertion of cache entries.
currsize	The number of bytes currently allocated to entries in the SQL statement cache
maxsize	The number of bytes specified in the STMT_CACHE_SIZE configuration parameter
poolsize	The cumulative number of bytes for all pools in the SQL statement cache. Use the **onstat -g ssc pool** option to monitor individual pool usage.
#hits	Setting of the STMT_CACHE_HITS configuration parameter, which specifies the number of times that a query is executed before it is inserted into the cache
nolimit	Setting of STMT_CACHE_NOLIMIT configuration parameter

The onstat -g ssc command lists the following information for each fully cached entry in the cache. The onstat -g ssc all option lists the following information for both the fully cached entries and key-only entries.

Column	Description
lru	The LRU identifier
hash	The hash-bucket identifier
ref_cnt	The number of sessions currently using this statement
hits	The number of times that users read the query from the cache, excluding the first time the statement entered the cache
flags	Shows flag codes. The flag codes for position 1 are: **D** Indicates that the statement has been dropped A statement in the cache can be dropped (not used any more) when one of its dependencies has changed. For example, when you run UPDATE STATISTICS for the table, the optimizer statistics might change, making the query plan for the SQL statement in the cache obsolete. In this case, the database server marks the statement as dropped the next time that it tries to use it.

Column	Description
	- Indicates that the statement has not been dropped The flag codes for position 2 are: **F** Indicates that the cache entry is fully cached and contains the memory structures for the query **I** Indicates that the statement is in the process of being moved to a fully cached state **-** Indicates that the statement is not fully cached A statement is not fully cached when the number of times the statement has been executed is less than the value of the STMT_CACHE_HITS configuration parameter. Entries with this - value in the second position appear in the **onstat -g ssc all** but not in the **onstat -g ssc** output.
heap_ptr	Pointer to the associated heap for the statement
database	Database against which the SQL statement is executed
user	User executing the SQL statement
statement	Statement text as it would be used to test for a match

Session memory

The database server uses the virtual portion of shared memory mainly for user sessions. Most of the memory that each user session allocates is for SQL statements. You can determine which session and which statements are using large amounts of memory. If necessary, you can set the SESSION_LIMIT_MEMORY configuration parameter to limit the amount of memory available to a session.

Use the following utility options to determine which session and prepared SQL statements are using large amounts of memory:

- onstat -g mem
- onstat -g stm

The onstat -g mem option displays memory usage of all sessions. You can find the session that is using the most memory by looking at the **totalsize** and **freesize** output columns. The following figure shows sample output for onstat -g mem. This sample output shows the memory use for three user sessions with the values `14`, `16`, `17` in the **names** output column.

Figure 11. onstat -g mem output

```
onstat -g mem

Pool Summary:
name         class addr     totalsize freesize #allocfrag #freefrag
...
14           V     a974020  45056     11960    99         10
16           V     a9ea020  90112     10608    159        5
17           V     a973020  45056     11304    97         13
...
Blkpool Summary:
name         class addr     size      #blks
mt           V     a235688  798720    19
global       V     a232800  0         0
```

To display the memory that is allocated by each prepared statement, use the onstat -g stm option. The following figure shows sample output for onstat -g stm.

Figure 12. onstat -g stm output

```
onstat -g stm


session   25 ------------------------------------------------
 sdblock  heapsz  statement (*' = Open cursor)
 d36b018    9216  select sum(i) from t where i between -1 and ?
 d378018    6240  *select tabname from systables where tabid=7
 d36b114    8400  <SPL statement>
```

The **heapsz** column in the output in Figure 12: onstat -g stm output on page 103 shows the amount of memory that is used by the statement. An asterisk (*) precedes the statement text if a cursor is open on the statement. The output does not show the individual SQL statements in an SPL routine.

To display the memory for only one session, specify the session ID in the onstat -g stm option. For an example, see Monitor session memory with onstat -g mem and onstat -g stm output on page 436.

Set the SESSION_LIMIT_MEMORY configuration parameter to limit how much memory a session can allocate, and can prevent individual sessions from monopolizing system resources. This limit does not apply to a user who holds administrative privileges, such as user **informix** or a DBSA user.

For example, to limit each session to 10 MB of memory, set `SESSION_LIMIT_MEMORY 102400` in the ONCONFIG file.

Related information

Estimating the size of the virtual portion of shared memory on page 65

SESSION_LIMIT_MEMORY configuration parameter on page

Data-replication buffers and memory utilization

Data replication requires two instances of the database server, a primary one and a secondary one, running on two computers. If you implement data replication for your database server, the database server holds logical-log records in the data-replication buffer before it sends them to the secondary database server.

The data-replication buffer is always the same size as the logical-log buffer.

Memory latches

The database server uses latches to control access to shared memory structures such as the buffer pool or the memory pools for the SQL statement cache. You can obtain statistics on latch use and information about specific latches. These statistics provide a measure of the system activity.

The statistics include the number of times that threads waited to obtain a latch. A large number of latch waits typically results from a high volume of processing activity in which the database server is logging most of the transactions.

Information about specific latches includes a listing of all the latches that are held by a thread and any threads that are waiting for latches. This information allows you to locate any specific resource contentions that exist.

You, as the database administrator, cannot configure or tune the number of latches. However, you can increase the number of memory structures on which the database server places latches to reduce the number of latch waits. For example, you can tune the number of SQL statement cache memory pools or the number of SQL statement cache LRU queues. For more information, see Multiple SQL statement cache pools on page 99.

Warning: Never stop a database server process that is holding a latch. If you do, the database server immediately initiates an abort.

Monitoring latches with command-line utilities

You can obtain information about latches by running **onstat -p** or **onstat -s**.

Monitoring latches with onstat -p

Run **onstat -p** to obtain the values in the **lchwaits** field. This field stores the number of times that a thread was required to wait for a shared-memory latch.

Figure 13: Partial onstat -p output showing the lchwaits field on page 104 shows an excerpt of sample **onstat -p** output that shows the **lchwaits** field.

Figure 13. Partial onstat -p output showing the lchwaits field

```
...
ixda-RA  idx-RA   da-RA   logrec-RA    RA-pgsused lchwaits
5        0        204      0           148        12
```

Related information

onstat -p command: Print profile counts on page

Monitoring latches with onstat -s

Run **onstat -s** to obtain general latch information. The output includes the **userthread** column, which lists the address of any user thread that is waiting for a latch.

You can compare this address with the user addresses in the **onstat -u** output to obtain the user-process identification number.

Figure 14: onstat -s output on page 105 shows sample **onstat -s** output.

Figure 14. onstat -s output

```
...
Latches with lock or userthread set
name     address  lock wait userthread
LRU1     402e90   0    0        6b29d8
bf[34]   4467c0   0    0        6b29d8
...
```

Monitoring latches with SMI tables

You can query the **sysprofile** SMI table to obtain the number of times a thread waited for a latch.

About this task

The **latchwts** column of the **sysprofile** table contains the number of times that a thread waited for a latch.

Encrypted values

An encrypted value uses more storage space than the corresponding plain text value because all of the information needed to decrypt the value except the encryption key is stored with the value.

Omitting the hint used with the password can reduce encryption overhead by up to 50 bytes. If you are using encrypted values, you must make sure that you have sufficient space available for the values.

Note: Embedding zero bytes in the encrypted result is not recommended.

Related information

Column-level encryption on page

Calculating storage requirements for encrypted data on page

Effect of configuration on I/O activity

The configuration of your database server affects I/O activity.

The following factors affect I/O activity:

- The assignment of chunks and dbspaces can create I/O *hot spots*, or disk partitions with a disproportionate amount of I/O activity.
- Your allocation of critical data, sort areas, and areas for temporary files and index builds can place intermittent loads on various disks.
- How you configure read-ahead can increase the effectiveness of individual I/O operations.
- How you configure the background I/O tasks, such as logging and page cleaning, can affect I/O throughput.

Chunk and dbspace configuration

The number of disks that you use and the configuration of your chunks, dbspaces, and blobspaces affect the performance of your database server. You can improve performance by planning disk use and the configuration of chunks, dbspaces, and blobspaces.

All the data that resides in a database is stored on disk. The speed at which the database server can copy the appropriate data pages to and from disk determines how well your application performs.

All the data that resides in a database is stored on disk. The Optical Subsystem also uses a magnetic disk to access TEXT or BYTE data that is retrieved from optical media. The speed at which the database server can copy the appropriate data pages to and from disk determines how well your application performs.

Disks are typically the slowest component in the I/O path for a transaction or query that runs entirely on one host computer. Network communication can also introduce delays in client/server applications, but these delays are typically outside the control of the database server administrator. For information about actions that the database server administrator can take to improve network communications, see Network buffer pools on page 50 and Connections and CPU utilization on page 59.

Disks can become overused or saturated when users request pages too often. Saturation can occur in the following situations:

- You use a disk for multiple purposes, such as for both logging and active database tables.
- Disparate data resides on the same disk.
- Table extents become interleaved.

The various functions that your application requires, as well as the consistency-control functions that the database server performs, determine the optimal disk, chunk, and dbspace layout for your application. The more disks that you make available to the database server, the easier it is to balance I/O across them. For more information about these factors, see Table performance considerations on page 156.

This section outlines important issues for the initial configuration of your chunks, dbspaces, and blobspaces. Consider the following issues when you decide how to lay out chunks and dbspaces on disks:

- Placement and mirroring of critical data
- Load balancing
- Reduction of contention
- Ease of backup and restore

Together with round-robin fragmentation, you can balance chunks over disks and controllers, saving time and handling errors. Placing multiple chunks on a single disk can improve throughput.

Associate disk partitions with chunks

You should assign chunks to entire disk partitions. When a chunk coincides with a disk partition (or device), it is easy to track disk-space use, and you avoid errors caused by miscalculated offsets.

The maximum size for a chunk is 4 terabytes.

Associate dbspaces with chunks

You should associate a single chunk with a dbspace, especially when that dbspace is to be used for a table fragment.

For more information about table placement and layout, see Table performance considerations on page 156.

Placing system catalog tables with database tables

When a disk that contains the system catalog for a particular database fails, the entire database remains inaccessible until the system catalog is restored. Because of this potential inaccessibility, do not cluster the system catalog tables for all databases in a single dbspace. Instead place the system catalog tables with the database tables that they describe.

About this task

To create a system catalog table in the table dbspace:

1. Create a database in the dbspace in which the table is to reside.
2. Use the SQL statements DATABASE or CONNECT to make that database the current database.
3. Enter the CREATE TABLE statement to create the table.

I/O for cooked files for dbspace chunks

On UNIX™, you can control the use of direct I/O for cooked files used for dbspace chunks.

On UNIX™, you can allocate disk space in two ways:

- Use files that are buffered through the operating system. These files are often called *cooked files*.
- Use unbuffered disk access, also called *raw* disk space.

When dbspaces reside on raw disk devices (also called *character-special devices*), the database server uses unbuffered disk access. A raw disk directly transfers data between the database server memory and disk without also copying data.

While you should generally use raw disk devices on UNIX™ systems to achieve better performance, you might prefer to use cooked files, which are easier to allocate and manage than raw devices. If you use cooked files, you might be able to get better performance by enabling the HCL OneDB™ direct I/O option.

In addition, HCL OneDB™ supports a separate concurrent I/O option on AIX® operating systems. If you enable concurrent I/O on AIX®, you get both unbuffered I/O and concurrent I/O. With concurrent I/O, writing to two parts of a file can occur concurrently. (On some other operating systems and file systems, enabling direct I/O also enables concurrent I/O as part of the same file system direct I/O feature.)

To determine the best performance, perform benchmark testing for the dbspace and table layout on your system.

Direct I/O (UNIX™)

On UNIX™, you can use direct I/O to improve the performance of cooked files. Direct I/O can be beneficial because it avoids file system buffering. Because direct I/O uses unbuffered I/O, it is more efficient for reads and writes that go to disk (as opposed to those reads and writes that merely access the file system buffers).

Direct I/O generally requires data to be aligned on disk sector boundaries.

Direct I/O also allows the use of kernel asynchronous I/O (KAIO), which can further improve performance. By using direct I/O and KAIO where available, the performance of cooked files used for dbspace chunks can approach the performance of raw devices.

If your file system supports direct I/O for the page size used for the dbspace chunk, the database server operates as follows:

- Does not use direct I/O by default.
- Uses direct I/O if the DIRECT_IO configuration parameter is set to `1`.
- Uses KAIO (if the file system supports it) with direct I/O by default.
- Does not use KAIO with direct I/O if the environment variable KAIOOFF is set.

If HCL OneDB™ uses direct I/O for a chunk, and another program tries to open the chunk file without using direct I/O, the open will normally succeed, but there can be a performance penalty. The penalty can occur because the file system attempts to ensure that each open sees the same file data, either by switching to buffered I/O and not using direct I/O for the duration of the conflicting open, or by flushing the file system cache before each direct I/O operation and invalidating the file system cache after each direct write.

HCL OneDB™ does not use direct I/O for temporary dbspaces.

Related information

DIRECT_IO configuration parameter (UNIX) on page

Direct I/O (Windows™)

Direct I/O is used for dbspace chunks on Windows™ platforms regardless of the value of the DIRECT_IO configuration parameter.

Concurrent I/O (AIX® only)

On AIX® operating systems, you can use concurrent I/O in addition to direct I/O for chunks that use cooked files. Concurrent I/O can improve performance, because it allows multiple reads and writes to a file to occur concurrently, without the usual serialization of noncompeting read and write operations.

Concurrent I/O can be especially beneficial when you have data in a single chunk file striped across multiple disks.

Concurrent I/O, which you enable by setting the DIRECT_IO configuration parameter to `2`, includes the benefit of avoiding file system buffering and is subject to the same limitations and use of KAIO as occurs if you use direct I/O without concurrent I/O. Thus, when concurrent I/O is enabled, you get both unbuffered I/O and concurrent I/O.

If HCL OneDB™ uses concurrent I/O for a chunk, and another program (such as an external backup program) tries to open the same chunk file without using concurrent I/O, the open operation will fail.

HCL OneDB™ does not use direct or concurrent I/O for cooked files used for temporary dbspace chunks.

Related information

DIRECT_IO configuration parameter (UNIX) on page

Enabling the direct I/O or concurrent I/O option (UNIX™)

Use the DIRECT_IO configuration parameter to enable the direct I/O option on UNIX™ or the concurrent I/O option on AIX®.

Before you begin

Prerequisites:

- You must log on as user **root** or **informix**.
- Direct I/O or concurrent I/O must be available and the file system must support direct I/O for the page size used for the dbspace chunk.

About this task

To enable direct I/O, set the DIRECT_IO configuration parameter to `1`.

To enable concurrent I/O with direct I/O on AIX® operating systems, set the DIRECT_IO configuration parameter to `2`.

If you do not want to enable direct I/O or concurrent I/O, set the DIRECT_IO configuration parameter to `0`.

Related information

DIRECT_IO configuration parameter (UNIX) on page

Confirming the use of the direct or concurrent I/O option (UNIX™)

You can confirm and monitor the use of direct I/O or concurrent I/O (on AIX®) for cooked file chunks.

About this task

You can confirm the use of direct I/O or concurrent I/O by:

- Displaying onstat -d information.

  The onstat -d command displays information that includes a flag that identifies whether direct I/O, concurrent I/O (on AIX®), or neither is used for cooked file chunks.
- Verifying that the DIRECT_IO configuration parameter is set to `1` (for direct I/O) or `2` (for concurrent I/O).

Related information

DIRECT_IO configuration parameter (UNIX) on page

onstat -d command: Print chunk information on page

Placement of critical data

The disk or disks that contain the system reserved pages, the physical log, and the dbspaces that contain the logical-log files are critical to the operation of the database server. The database server cannot operate if any of these elements becomes unavailable. By default, the database server places all three critical elements in the root dbspace.

To arrive at an appropriate placement strategy for critical data, you must make a trade-off between the availability of data and maximum logging performance.

The database server also places temporary table and sort files in the root dbspace by default. You should use the DBSPACETEMP configuration parameter and the **DBSPACETEMP** environment variable to assign these tables and files to other dbspaces. For details, see Configure dbspaces for temporary tables and sort files on page 114.

Consider separate disks for critical data components

If you place the root dbspace, logical log, and physical log in separate dbspaces on separate disks, you can obtain some distinct performance advantages. The disks that you use for each critical data component should be on separate controllers.

This approach has the following advantages:

- Isolates logging activity from database I/O and allows physical-log I/O requests to be serviced in parallel with logical-log I/O requests
- Reduces the time that you need to recover from a failure

  However, unless the disks are mirrored, there is an increased risk that a disk that contains critical data might be affected in the event of a failure, which will bring the database server to a halt and require the complete restoration of all data from a level-0 backup.

- Allows for a relatively small root dbspace that contains only reserved pages, the database partition, and the **sysmaster** database

  In many cases, 10,000 kilobytes is sufficient.

The database server uses different methods to configure various portions of critical data. To assign an appropriate dbspace for the root dbspace and physical log, set the appropriate database server configuration parameters. To assign the logical-log files to an appropriate dbspace, use the **onparams** utility.

For more information about the configuration parameters that affect each portion of critical data, see Configuration parameters that affect critical data on page 113.

Consider mirroring for critical data components

Consider mirroring for the dbspaces that contain critical data. Mirroring these dbspaces ensures that the database server can continue to operate even when a single disk fails.

However, depending on the mix of I/O requests for a given dbspace, a trade-off exists between the fault tolerance of mirroring and I/O performance. You obtain a marked performance advantage when you mirror dbspaces that have a read-intensive usage pattern and a slight performance disadvantage when you mirror write-intensive dbspaces.

Most modern storage devices have excellent mirroring capabilities, and you can use those devices instead of the mirroring capabilities of the database server.

When mirroring is in effect, two disks are available to handle read requests, and the database server can process a higher volume of those requests. However, each write request requires two physical write operations and does not complete until both physical operations are performed. The write operations are performed in parallel, but the request does not complete until the slower of the two disks performs the update. Thus, you experience a slight performance penalty when you mirror write-intensive dbspaces.

Consider mirroring the root dbspace

You can achieve a certain degree of fault tolerance with a minimum performance penalty if you mirror the root dbspace and restrict its contents to read-only or seldom-accessed tables.

When you place update-intensive tables in other, nonmirrored dbspaces, you can use the database server backup-and-restore facilities to perform warm restores of those tables in the event of a disk failure. When the root dbspace is mirrored, the database server remains online to service other transactions while the failed disk is being repaired.

When you mirror the root dbspace, always place the first chunk on a different device than that of the mirror. The MIRRORPATH configuration parameter should have a different value than ROOTPATH.

Related information

MIRRORPATH configuration parameter on page

ROOTPATH configuration parameter on page

Consider mirroring smart-large-object chunks

You can achieve higher availability and faster access if you mirror chunks that contain metadata pages.

An sbspace is a logical storage unit composed of one or more chunks that store *smart large objects*, which consist of CLOB (character large object) or BLOB (binary large object) data.

The first chunk of an sbspace contains a special set of pages, called *metadata*, which is used to locate smart large objects in the sbspace. Additional chunks that are added to the sbspace can also have metadata pages if you specify them on the **onspaces** command when you create the chunk.

Consider mirroring chunks that contain metadata pages for the following reasons:

- Higher availability

  Without access to the metadata pages, users cannot access any smart large objects in the sbspace. If the first chunk of the sbspace contains all of the metadata pages and the disk that contains that chunk becomes unavailable, you cannot access a smart large object in the sbspace, even if it resides on a chunk on another disk. For high availability, mirror at least the first chunk of the sbspace and any other chunk that contains metadata pages.

- Faster access

  By mirroring the chunk that contains the metadata pages, you can spread read activity across the disks that contain the primary chunk and mirror chunk.

Related information

Sbspaces on page

Mirroring and its effect on the logical log

The logical log is write intensive. If the dbspace that contains the logical-log files is mirrored, you encounter a slight double-write performance penalty. However, you can adjust the rate at which logging generates I/O requests to a certain extent by choosing an appropriate log buffer size and logging mode.

For details on the slight double-write performance penalty, see Consider mirroring for critical data components on page 111.

With unbuffered and ANSI-compliant logging, the database server requests a flush of the log buffer to disk for every committed transaction (two when the dbspace is mirrored). Buffered logging generates far fewer I/O requests than unbuffered or ANSI-compliant logging.

With buffered logging, the log buffer is written to disk only when it fills and all the transactions that it contains are completed. You can reduce the frequency of logical-log I/O even more if you increase the size of your logical-log buffers. However, buffered logging leaves transactions in any partially filled buffers vulnerable to loss in the event of a system failure.

Although database consistency is guaranteed under buffered logging, specific transactions are not guaranteed against a failure. The larger the logical-log buffers, the more transactions you might need to reenter when service is restored after a failure.

Unlike the physical log, you cannot specify an alternative dbspace for logical-log files in your initial database server configuration. Instead, use the **onparams** utility first to add logical-log files to an alternative dbspace and then drop logical-log files from the root dbspace.

Related information

The onparams Utility on page

Mirroring and its effect on the physical log

The physical log is write intensive, with activity occurring at checkpoints and when buffered data pages are flushed. I/O to the physical log also occurs when a page-cleaner thread is activated. If the dbspace that contains the physical log is mirrored, you encounter a slight double-write performance penalty.

For details on the slight double-write performance penalty, see Consider mirroring for critical data components on page 111.

To keep I/O to the physical log at a minimum, you can adjust the checkpoint interval and the LRU minimum and maximum thresholds. (See CKPTINTVL and its effect on checkpoints on page 138 and BUFFERPOOL and its effect on page cleaning on page 149.)

Configuration parameters that affect critical data

The configuration parameters that configure the root dbspace and the logical and physical logs affect critical data.

You can use the following configuration parameters to configure the root dbspace:

- ROOTNAME
- ROOTOFFSET
- ROOTPATH
- ROOTSIZE
- MIRROR
- MIRRORPATH
- MIRROROFFSET

These parameters determine the location and size of the initial chunk of the root dbspace and configure mirroring, if any, for that chunk. (If the initial chunk is mirrored, all other chunks in the root dbspace must also be mirrored). Otherwise, these parameters have no major impact on performance.

The following configuration parameters affect the logical logs:

- LOGSIZE
- LOGBUFF

The LOGSIZE configuration parameter determines the size of each logical-log files. The LOGBUFF configuration parameter determines the size of the three logical-log buffers that are in shared memory.

The PHYSFILE configuration parameter determines the initial size of the physical log in rootdbs. This configuration parameter is used only when the instance is created.

Related information

The LOGBUFF configuration parameter and memory utilization on page 76

Checkpoints and the physical log on page 140

Configure dbspaces for temporary tables and sort files

Applications that use temporary tables or large sort operations require a large amount of temporary space. To improve performance of these applications, use the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable to designate one or more dbspaces for temporary tables and sort files.

Depending on how the temporary space is created, the database server uses the following default locations for temporary table and sort files when you do not set DBSPACETEMP:

- The dbspace of the current database, when you create an explicit temporary table with the TEMP TABLE clause of the CREATE TABLE statement and do not specify a dbspace for the table either in the IN dbspace clause or in the FRAGMENT BY clause

  This action can severely affect I/O to that dbspace. If the root dbspace is mirrored, you encounter a slight double-write performance penalty for I/O to the temporary tables and sort files.
- The root dbspace when you create an explicit temporary table with the INTO TEMP option of the SELECT statement

  This action can severely affect I/O to the root dbspace. If the root dbspace is mirrored, you encounter a slight double-write performance penalty for I/O to the temporary tables and sort files.
- The operating-system directory or file that you specify in one of the following variables:
  - In UNIX™, the operating-system directory or directories that the **PSORT_DBTEMP** environment variable specifies, if it is set

    If **PSORT_DBTEMP** is not set, the database server writes sort files to the operating-system file space in the **/tmp** directory.
  - In Windows™, the directory specified in **TEMP** or **TMP** in the User Environment Variables window on **Control Panel > System.**

  The database server uses the operating-system directory or files to direct any overflow that results from the following database operations:

- SELECT statement with GROUP BY clause
- SELECT statement with ORDER BY clause
- Hash-join operation
- Nested-loop join operation
- Index builds

Warning: If you do not specify a value for the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable, the database server uses this operating-system file for implicit temporary tables. If this file system has insufficient space to hold a sort file, the query that performs the sort returns an error. Meanwhile, the operating system might be severely impacted until you remove the sort file.

You can improve performance with the use of temporary dbspaces that you create exclusively to store temporary tables and sort files. Use the DBSPACETEMP configuration parameter and the **DBSPACETEMP** environment variable to assign these tables and files to temporary dbspaces.

When you specify dbspaces in either the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable, you gain the following performance advantages:

- Reduced I/O impact on the root dbspace, production dbspaces, or operating-system files
- Use of parallel sorts into the temporary files (to process query clauses such as ORDER BY or GROUP BY, or to sort index keys when you execute CREATE INDEX) when you specify more than one dbspace for temporary tables and PDQ priority is set to greater than `0`.
- Improved speed with which the database server creates temporary tables when you assign two or more temporary dbspaces on separate disks
- Automatic fragmentation of the temporary tables across dbspaces when SELECT....INTO TEMP statements are run

The following table shows statements that create temporary tables and information about where the temporary tables are created.

Statement That Creates Temporary Table	Database Logged	WITH NO LOG clause	FRAGMENT BY clause	Where Temp Table Created
CREATE TEMP TABLE	Yes	No	No	Root dbspace
CREATE TEMP TABLE	Yes	Yes	No	One of dbspaces that are specified in DBSPACETEMP
CREATE TEMP TABLE	Yes	No	Yes	Cannot create temp table. Error 229/196
SELECT ..INTO TEMP	Yes	Yes	No	Fragmented by round-robin only in the

Statement That Creates Temporary Table	Database Logged	WITH NO LOG clause	FRAGMENT BY clause	Where Temp Table Created
				non-logged dbspaces that are specified in DBSPACETEMP

Important: Use the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable for better performance of sort operations and to prevent the database server from unexpectedly filling file systems. The dbspaces that you list must be composed of chunks that are allocated as unbuffered devices.

Related information

DBSPACETEMP configuration parameter on page

Specify temporary tables in the DBSPACETEMP configuration parameter on page 117

CREATE TEMP TABLE statement on page

INTO TEMP clause on page

Creating temporary dbspaces

You can create a dbspace for the exclusive use of temporary tables and sort files. The database server does not perform logical or physical logging of temporary dbspaces, and temporary dbspaces are never backed up as part of a full-system backup.

About this task

To create a dbspace for the exclusive use of temporary tables and sort files, use **onspaces -t**. For best performance, use the following guidelines:

- If you create more than one temporary dbspace, create each dbspace on a separate disk to balance the I/O impact.
- Place no more than one temporary dbspace on a single disk.

You cannot mirror a temporary dbspace that you create with **onspaces -t**.

Important: In the case of a database with logging, you must include the WITH NO LOG clause in the SELECT... INTO TEMP statement to place the explicit temporary tables in the dbspaces listed in the DBSPACETEMP configuration

parameter and the **DBSPACETEMP** environment variable. Otherwise, the database server stores the explicit temporary tables in the root dbspace.

Related information

DBSPACETEMP configuration parameter on page

create tempdbspace argument: Create a temporary dbspace (SQL administration API) on page

onspaces -c -d: Create a dbspace on page

Specify temporary tables in the DBSPACETEMP configuration parameter

The DBSPACETEMP configuration parameter specifies a list of dbspaces in which the database server places temporary tables and sort files by default. Some or all of the dbspaces that you list in this configuration parameter can be temporary dbspaces, which are reserved exclusively to store temporary tables and sort files.

If the database server inserts data into a temporary table through a SELECT INTO TEMP operation that creates the TEMP table, that temporary table uses round-robin distributed storage. Its fragments are created in the temporary dbspaces that are listed in the DBSPACETEMP configuration parameter or in the DBSPACETEMP environment variable. For example, the following query uses round-robin distributed storage:

```
SELECT col1 FROM tab1
   INTO TEMP temptab1 WITH NO LOG;
```

The DBSPACETEMP configuration parameter lets the database administrator restrict which dbspaces the database server uses for temporary storage.

Important: The DBSPACETEMP configuration parameter is not set in the **onconfig.std** file. For best performance with temporary tables and sort files, use DBSPACETEMP to specify two or more dbspaces on separate disks.

Tips:

- If you work on a small system with a limited number of disks and cannot place temporary dbspaces on different disk drives, you might consider using 1 (or possibly 2) temporary dbspaces. This can reduce the logging that is associated with the temporary dbspaces.
- If you have many disk drives, you can parallelize many operations (such as sorts, joins, and temporary tables) without having multiple temporary dbspaces. The number of temporary dbspaces that you have relates to how much you want to spread the I/O out. A good starting place is 4 temporary dbspaces. If you create too many small temporary dbspaces, you will not have enough space for nonparallel creation of large objects.

Related information

Configure dbspaces for temporary tables and sort files on page 114

DBSPACETEMP configuration parameter on page

Distribution schemes on page 265

CREATE TEMP TABLE statement on page

Override the DBSPACETEMP configuration parameter for a session

To override the DBSPACETEMP configuration parameter, you can use the **DBSPACETEMP** environment variable for both temporary tables and sort files. This environment variable specifies a comma- or colon-separated list of dbspaces in which to place temporary tables for the current session.

Important: Use the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable for better performance of sort operations and to prevent the database server from unexpectedly filling file systems.

You should use DBSPACETEMP rather than the **PSORT_DBTEMP** environment variable to specify sort files for the following reasons:

- **DBSPACETEMP** typically yields better performance.

  When dbspaces reside on character-special devices (also known as raw disk devices), the database server uses unbuffered disk access. I/O is faster to unbuffered devices than to regular (buffered) operating-system files because the database server manages the I/O operation directly.
- **PSORT_DBTEMP** specifies one or more operating-system directories in which to place sort files.

  These operating-system files can unexpectedly fill on your computer because the database server does not manage them.

Estimating temporary space for dbspaces and hash joins

You can estimate and increase the amount of temporary space for dbspaces and for hash joins. If you do this, you can prevent the possible overflow of memory to temporary space on disk.

You can use the following guidelines to estimate the amount of temporary space to allocate:

- For OLTP applications, allocate temporary dbspaces that equal at least 10 percent of the table.
- For DSS applications, allocate temporary dbspaces that equal at least 50 percent of the table.

A hash join, which works by building a table (the hash table) from the rows in one of the tables in a join, and then probing it with rows from the other table, can use a significant amount of memory and can potentially overflow to temporary space on disk. The hash table size is governed by the size of the table used to build the hash table (which is often the smaller of the

two tables in the join), after applying any filters, which can reduce the number of rows and possibly reduce the number of columns.

Hash-join partitions are organized into pages. Each page has a header. The header and tuples are larger in databases on 64-bit platforms than in builds on 32-bit platforms. The size of each page is the base page size (2K or 4K depending on system) unless a single row needs more space. If you need more space, you can add bytes to the length of your rows.

You can use the following formula to estimate the amount of memory that is required for the hash table in a hash join:

```
hash_table_size = (32 bytes + row_size_smalltab) * num_rows_smalltab
```

where `row_size_smalltab` and `num_rows_smalltab` refer to the row size and the number of rows, respectively, in the smaller of the two tables participating in the hash join.

For example, suppose you have a page head that is 80 bytes in length and a row header that is 48 bytes in length. Because each row must be aligned to 8 bytes, you might need to add up to 7 bytes to the row length, as shown in these formulas:

```
per_row_size = 48 bytes + rowsize + mod(rowsize, 8)
page_size = base_page_size (2K or 4K)
rows_per_page = round_down_to_integer((page_size - 80 bytes) / per_row_size)
```

If the value of `rows_per_page` is less than one, increase the `page_size` value to the smallest multiple of the `base_page_size`, as shown in this formula:

```
size = (numrows_smalltab / rows_per_page) * page_size
```

You can use the DS_NONPDQ_QUERY_MEM configuration parameter to configure sort memory for all queries except PDQ queries. Its setting has no effect, however, if the PDQ priority setting is greater than zero.

For more information, see Hash join on page 292 and Configuring memory for queries with hash joins, aggregates, and other memory-intensive elements on page 414.

Related information

DS_NONPDQ_QUERY_MEM configuration parameter on page

PSORT_NPROCS environment variable

The **PSORT_NPROCS** environment variable specifies the maximum number of threads that the database server can use to sort a query. When a query involves a large sort operation, multiple sort threads can execute in parallel to improve the performance of the query.

When the value of PDQ priority is `0` and **PSORT_NPROCS** is greater than 1, the database server uses parallel sorts. The management of PDQ does not limit these sorts. In other words, although the sort is executed in parallel, the database server does not regard sorting as a PDQ activity. When PDQ priority is `0`, the database server does not control sorting by any of the PDQ configuration parameters.

When PDQ priority is greater than 0 and **PSORT_NPROCS** is greater than 1, the query benefits both from parallel sorts and from PDQ features such as parallel scans and additional memory. Users can use the **PDQPRIORITY** environment variable

to request a specific proportion of PDQ resources for a query. You can use the MAX_PDQPRIORITY parameter to limit the number of such user requests. For more information about MAX_PDQPRIORITY, see Limiting PDQ resources in queries on page 44.

The database server allocates a relatively small amount of memory for sorting, and that memory is divided among the **PSORT_NPROCS** sort threads. Sort processes use temporary space on disk when not enough memory is allocated. For more information about memory allocated for sorting, see Estimating memory needed for sorting on page 226.

Important: For better performance for a sort operation, set PSORT_NPROCS initially to 2 if your computer has multiple CPUs. If the subsequent CPU activity is lower than I/O activity, you can increase the value of PSORT_NPROCS.

For more information about sorts during index builds, see Improving performance for index builds on page 225.

Configure sbspaces for temporary smart large objects

Applications can use temporary smart large objects for text, image, or other user-defined data types that are only required during the life of the user session. These applications do not require logging of the temporary smart large objects. Logging adds I/O activity to the logical log and increases memory utilization.

You can store temporary smart large objects in a permanent sbspace or a temporary sbspace.

- Permanent sbspaces

  If you store the temporary smart large objects in a regular sbspace and keep the default no logging attribute, changes to the objects are not logged, but the metadata is always logged.

- Temporary sbspaces

  Applications that update temporary smart large objects stored in temporary sbspaces are significantly faster because the database server does not log the metadata or the user data in a temporary sbspace.

To improve performance of applications that update temporary smart large objects, specify the **LOTEMP** flag in the **mi_lo_specset_flags** or **ifx_lo_specset_flags** API function and specify a temporary sbspace for the temporary smart large objects. The database server uses the following order of precedence for locations to place temporary smart large objects:

- The sbspace you specify in the **mi_lo_specset_sbspace** or **ifx_lo_specset_sbspace** API function when you create the smart large object

  Specify a temporary sbspace in the API function so that changes to the objects and the metadata are not logged. The sbspace you specify in the API function overrides any default sbspaces that the SBSPACETEMP or SBSPACENAME configuration parameters might specify.

- The sbspace you specify in the IN Sbspace clause when you create an explicit temporary table with the TEMP TABLE clause of the CREATE TABLE statement

Specify a temporary sbspace in the IN Sbspace clause so that changes to the objects and the metadata are not logged.

- The permanent sbspace you specify in the SBSPACENAME configuration parameter, if you do not specify an sbspace in the SBSPACETEMP configuration parameter

If no temporary sbspace is specified in any of the above methods, then the database server issues the following error message when you try to create a temporary smart large object:

```
-12053  Smart Large Objects: No sbspace number specified.
```

Creating temporary sbspaces

To create an sbspace for the exclusive use of temporary smart large objects, use **onspaces -c -S** with the **-t** option.

For best performance, use the following guidelines:

- If you create more than one temporary sbspace, create each sbspace on a separate disk to balance the I/O impact.
- Place no more than one temporary sbspace on a single disk.

The database server does not perform logical or physical logging of temporary sbspaces, and temporary sbspaces are never backed up as part of a full-system backup. You cannot mirror a temporary sbspace that you create with **onspaces -t**.

Important: In the case of a database with logging, you must include the WITH NO LOG clause in the SELECT... INTO TEMP statement to place the temporary smart large objects in the sbspaces listed in the SBSPACETEMP configuration parameter. Otherwise, the database server stores the temporary smart large objects in the sbspace listed in the SBSPACENAME configuration parameter.

Related information

onspaces -c -S: Create an sbspace on page

Creating a temporary sbspace on page

Specify which sbspaces to use for temporary storage

The SBSPACETEMP configuration parameter specifies a list of sbspaces in which the database server places temporary smart large objects by default. Some or all of the sbspaces that you list in this configuration parameter can be temporary sbspaces, which are reserved exclusively to store temporary smart large objects.

Important: The SBSPACETEMP configuration parameter is not set in the **onconfig.std** file. For best performance with temporary smart large objects, use SBSPACETEMP to specify two or more sbspaces on separate disks.

Related information

SBSPACETEMP configuration parameter on page

Placement of simple large objects

You can store simple large objects in either the same dbspace in which the table resides or in a blobspace.

A blobspace is a logical storage unit composed of one or more chunks that store only simple large objects (TEXT or BYTE data). For information about sbspaces, which store smart large objects (such as BLOB, CLOB, or multirepresentational data), see Factors that affect I/O for smart large objects on page 127.

If you use a blobspace, you can store simple large objects on a separate disk from the table with which the data is associated. You can store simple large objects associated with different tables in the same blobspace.

You can create a blobspace with the **onspaces** utility or with an SQL administration API command that uses the **create blobspace** argument with the admin() or task() function.

You assign simple large objects to a blobspace when you create the tables with which simple large objects are associated, using the CREATE TABLE statement.

Simple large objects are not logged and do not pass through the buffer pool. However, frequency of checkpoints can affect applications that access TEXT or BYTE data. For more information, see LOGSIZE and LOGFILES and their effect on checkpoints on page 139.

Related information

CREATE TABLE statement on page

create blobspace argument: Create a blobspace (SQL administration API) on page

onspaces -c -b: Create a blobspace on page

Advantage of blobspaces over dbspaces

If you store simple large objects in a blobspace on a separate disk from the table with which it is associated, instead of storing the objects in a dbspace, you can obtain some performance advantages.

The performance advantages of storing simple large objects in a blobspace are:

- You have parallel access to the table and simple large objects.
- Unlike simple large objects stored in a dbspace, blobspace data is written directly to disk. Simple large objects do not pass through resident shared memory, which leaves memory pages free for other uses.
- Simple large objects are not logged, which reduces logging I/O activity for logged databases.

For more information, see Storing simple large objects in the tblspace or a separate blobspace on page 164.

Blobpage size considerations

Blobspaces are divided into units called *blobpages*. The database server retrieves simple large objects from a blobspace in blobpage-sized units. You specify the size of a blobpage in multiples of a disk page when you create the blobspace.

The optimal blobpage size for your configuration depends on the following factors:

- The size distribution of the simple large objects
- The trade-off between retrieval speed for your largest simple large object and the amount of disk space that is wasted by storing simple large objects in large blobpages

To retrieve simple large objects as quickly as possible, use the size of your largest simple large object rounded up to the nearest disk-page-sized increment. This scheme guarantees that the database server can retrieve even the largest simple large object in a single I/O request. Although this scheme guarantees the fastest retrieval, it has the potential to waste disk space. Because simple large objects are stored in their own blobpage (or set of blobpages), the database server reserves the same amount of disk space for every blobpage even if the simple large object takes up a fraction of that page. Using a smaller blobpage allows you to make better use of your disk, especially when large differences exist in the sizes of your simple large objects.

To achieve the greatest theoretical utilization of space on your disk, you can make your blobpage the same size as a standard disk page. Then many, if not most, simple large objects would require several blobpages. Because the database server acquires a lock and issues a separate I/O request for each blobpage, this scheme performs poorly.

In practice, a balanced scheme for sizing uses the most frequently occurring simple-large-object size as the size of a blobpage. For example, suppose that you have 160 simple-large-object values in a table with the following size distribution:

- Of these values, 120 are 12 kilobytes each.
- The other 40 values are 16 kilobytes each.

You can choose one of the following blobpage sizes:

- The 12-kilobyte blobpage size provides greater storage efficiency than a 16-kilobyte blobpage size, as the following two calculations show:
    - 12 kilobytes

      This configuration allows the majority of simple-large-object values to require a single blobpage and the other 40 values to require two blobpages. In this configuration, 8 kilobytes is wasted in the second blobpage for each of the larger values. The total wasted space is as follows:

```
wasted-space = 8 kilobtyes * 40
             = 329 kilobytes
```

- 16 kilobytes

  In this configuration, 4 kilobytes is wasted in the extents of 120 simple large objects. The total wasted space is as follows:

  ```
  wasted-space = 4 kilobtyes * 120
               = 640 kilobytes
  ```

- If your applications access the 16-kilobyte simple-large-object values more frequently, the database server must perform a separate I/O operation for each blobpage. In this case, the 16-kilobyte blobpage size provides better retrieval speed than a 12-kilobyte blobpage size.

The maximum number of pages that a blobspace can contain is 2147483647. Therefore, the size of the blobspace is limited to the blobpage size x 2147483647. This includes blobpages in all chunks that make up the blobspace.

Tip: If a table has more than one *simple-large-object column and the data values are not close in size, store t*he data in different blobspaces, each with an appropriately sized blobpage.

Optimize blobspace blobpage size

When you are evaluating blobspace storage strategy, you can measure efficiency by two criteria: blobpage fullness and the blobpages required per simple large object.

Blobpage fullness refers to the amount of data within each blobpage. TEXT and BYTE data stored in a blobspace cannot share blobpages. Therefore, if a single simple large object requires only 20 percent of a blobpage, the remaining 80 percent of the page is unavailable for use.

However, avoid making the blobpages too small. When several blobpages are needed to store each simple large object, you increase the overhead cost of storage. For example, more locks are required for updates, because a lock must be acquired for each blobpage.

Obtain blobspace storage statistics

To help you determine the optimal blobpage size for each blobspace, use the oncheck -pB command.

The command lists the following statistics for each table (or database):

- The number of blobpages used by the table (or database) in each blobspace
- The average fullness of the blobpages used by each simple large object stored as part of the table (or database)

Determine blobpage fullness with oncheck -pB output

The oncheck -pB command displays statistics that describe the average fullness of blobpages. These statistics provide a measure of storage efficiency for individual simple large objects in a database or table.

If you find that the statistics for a significant number of simple large objects show a low percentage of fullness, the database server might benefit from changing the size of the blobpage in the blobspace.

Both the oncheck -pB and onstat -d update commands display the same information about the number of free blobpages. The onstat -d update command displays the same information as onstat -d and an accurate number of free blobpages for each blobspace chunk.

Execute oncheck -pB with either a database name or a table name as a parameter. The following example retrieves storage information for all simple large objects stored in the table **sriram.catalog** in the **stores_demo** database:

```
oncheck -pB stores_demo:sriram.catalog
```

oncheck -pB Output

Figure 15: Output of oncheck -pB on page 125 shows the output of this command.

Figure 15. Output of oncheck -pB

```
              BLOBSpace Report for stores_demo:sriram.catalog

Total pages used by table            7

BLOBSpace usage:
Space   Page             Percent Full
Name    Number    Pages  0-25%  26-50%  51-75  76-100%
-------------------------------------------------------------
blobPIC 0x300080  1      x
   blobPIC 0x300082  2      x
         ------
Page Size is 6144      3

bspc1   0x2000b2  2             x
bspc1   0x2000b6  2                     x
              ------
Page Size is 2048      4
```

Space Name is the name of the blobspace that contains one or more simple large objects stored as part of the table (or database).

Page Number is the starting address in the blobspace of a specific simple large object.

Pages is the number of the database server pages required to store this simple large object.

Percent Full is a measure of the average blobpage fullness, by blobspace, for each blobspace in this table or database.

Page Size is the size in bytes of the blobpage for this blobspace. Blobpage size is always a multiple of the database server page size.

The example output indicates that four simple large objects are stored as part of the table **sriram.catalog**. Two objects are stored in the blobspace **blobPIC** in 6144-byte blobpages. Two more objects are stored in the blobspace **bspc1** in 2048-byte blobpages.

The summary information that appears at the top of the display, **Total pages used by table** is a simple total of the blobpages needed to store simple large objects. The total says nothing about the size of the blobpages used, the number of simple large objects stored, or the total number of bytes stored.

The efficiency information displayed under the **Percent Full** heading is imprecise, but it can alert an administrator to trends in the storage of TEXT and BYTE data.

Interpreting blobpage average fullness

You can analyze the output of the oncheck -pB command to calculate average fullness.

The first simple large object listed in Determine blobpage fullness with oncheck -pB output on page 124 is stored in the blobspace **blobPIC** and requires one 6144-byte blobpage. The blobpage is 51 to 75 percent full, meaning that the size is between 0.51 * 6144 = 3133 bytes and 0.75 * 6144 = 4608. The maximum size of this simple large object must be less than or equal to 75 percent of 6144 bytes, or 4608 bytes.

The second object listed under blobspace **blobPIC** requires two 6144-byte blobpages for storage, or a total of 12,288 bytes. The average fullness of all allocated blobpages is 51 to 75 percent. Therefore, the minimum size of the object must be greater than 50 percent of 12,288 bytes, or 6144 bytes. The maximum size of the simple large object must be less than or equal to 75 percent of 12,288 bytes, or 9216 bytes. The average fullness does not mean that each page is 51 to 75 percent full. A calculation would yield 51 to 75 percent average fullness for two blobpages where the first blobpage is 100 percent full and the second blobpage is 2 to 50 percent full.

Now consider the two simple large objects in blobspace **bspc1**. These two objects appear to be nearly the same size. Both objects require two 2048-byte blobpages, and the average fullness for each is 76 to 100 percent. The minimum size for these simple large objects must be greater than 75 percent of the allocated blobpages, or 3072 bytes. The maximum size for each object is slightly less than 4096 bytes (allowing for overhead).

Analyzing efficiency criteria with oncheck -pB output

You can analyze the output of the oncheck -pB command to determine if there is a more efficient storage strategy.

Looking at the efficiency information for that is shown for blobspace **bspc1** in Figure 15: Output of oncheck -pB on page 125, a database server administrator might decide that a better storage strategy for TEXT and BYTE data would be to double the blobpage size from 2048 bytes to 4096 bytes. (Blobpage size is always a multiple of the database server page size.) If the database server administrator made this change, the measure of page fullness would remain the same, but the number of locks needed during an update of a simple large object would be reduced by half.

The efficiency information for blobspace **blobPIC** reveals no obvious suggestion for improvement. The two simple large objects in **blobPIC** differ considerably in size, and there is no optimal storage strategy. In general, simple large objects of similar size can be stored more efficiently than simple large objects of different sizes.

Factors that affect I/O for smart large objects

An sbspace is a logical storage unit, composed of one or more chunks, in which you can store smart large objects (such as BLOB, CLOB, or multi representational data). Disk layout for sbspaces, the settings of certain configuration parameters, and some **onspaces** utility options affect I/O for smart large objects.

The DataBlade® API and the application programming interface also provide functions that affect I/O operations for smart large objects.

Important: For most applications, you should use the values that the database server calculates for the disk-storage information.

Related information

Sbspaces on page

What is Informix ESQL/C? on page

DataBlade API overview on page

Disk layout for sbspaces

You create sbspaces on separate disks from the table with which the data is associated. You can store smart large objects associated with different tables within the same sbspace. When you store smart large objects in an sbspace on a separate disk from the table with which it is associated, the database server provides some performance advantages.

These performance advantages are:

- You have parallel access to the table and smart large objects.
- When you choose not to log the data in an sbspace, you reduce logging I/O activity for logged databases.

To create an sbspace, use the **onspaces** utility. You assign smart large objects to an sbspace when you use the CREATE TABLE statement to create the tables with which the smart large objects are associated.

Related information

onspaces -c -S: Create an sbspace on page

CREATE TABLE statement on page

Configuration parameters that affect sbspace I/O

The SBSPACENAME, BUFFERPOOL, and LOGBUFF configuration parameters affect the I/O performance of sbspaces.

The SBSPACENAME configuration parameter indicates the default sbspace name if you do not specify the sbspace name when you define a column of data type CLOB or BLOB. To reduce disk contention and provide better load balancing, place the default sbspace on a separate disk from the table data.

The BUFFERPOOL configuration parameter specifies the default values for buffers and LRU queues in a buffer pool for both the default page size buffer pool and for any non-default pages size buffer pools. The size of your memory buffer pool affects I/O operations for smart large objects because the buffer pool is the default area of shared memory for these objects. If your applications frequently access smart large objects, it is advantageous to have these objects in the buffer pool. Smart large objects only use the default page size buffer pool. For information about estimating the amount to increase your buffer pool for smart large objects, see The BUFFERPOOL configuration parameter and memory utilization on page 72.

By default, the database server reads smart large objects into the buffers in the resident portion of shared memory. For more information on using lightweight I/O buffers, see Lightweight I/O for smart large objects on page 130.

The LOGBUFF configuration parameter affects logging I/O activity because it specifies the size of the logical-log buffers that are in shared memory. The size of these buffers determines how quickly they fill and therefore how often they need to be flushed to disk.

If you log smart-large-object user data, increase the size of your logical-log buffer to prevent frequent flushing to these log files on disk.

Related information

SBSPACENAME configuration parameter on page

BUFFERPOOL configuration parameter on page

LOGBUFF configuration parameter on page

onspaces options that affect sbspace I/O

When you create an sbspace with the **onspaces** utility, you specify information that affects I/O performance. This information includes the size of extents, the buffering mode (and whether you want the server to use lightweight I/O), and logging.

Sbspace extents

As you add smart large objects to a table, the database server allocates disk space to the sbspace in units called *extents*. Each extent is a block of physically contiguous pages from the sbspace.

Even when the sbspace includes more than one chunk, each extent is allocated entirely within a single chunk so that it remains contiguous. Contiguity is important to I/O performance.

When the pages of data are contiguous, disk-arm motion is minimized when the database server reads the rows sequentially. The mechanism of extents is a compromise between the following competing requirements:

- The size of some smart large objects is not known in advance.
- The number of smart large objects in different tables can grow at different times and different rates.
- All the pages of a single smart large object should ideally be adjacent for best performance when you retrieve the entire object.

Because you might not be able to predict the number and size of smart large objects, you cannot specify the extent length of smart large objects. Therefore, the database server adds extents only as they are needed, but all the pages in any one extent are contiguous for better performance. In addition, when the database server creates a new extent that is adjacent to the previous extent, it treats both extents as a single extent.

The number of pages in an sbspace extent is determined by one of the following methods:

- The database server calculates the extent size for a smart large object from a set of heuristics, such as the number of bytes in a write operation. For example, if an operation asks to write 30 kilobytes, the database server tries to allocate an extent the size of 30 kilobytes.
- The final size of the smart large object as indicated by one of the following functions when you open the sbspace in an application program:
  - *For DB-Access:* the DataBlade® API **mi_lo_specset_estbytes** function. For more information about the DataBlade® API functions to open a smart large object and set the estimated number of bytes, see the *HCL OneDB™ DataBlade® API Programmer's Guide*.
  - *For ESQL/C:* the **ifx_lo_specset_estbytes** function. For more information about the functions to open a smart large object and set the estimated number of bytes, see the *HCL OneDB™ ESQL/C Programmer's Manual*.

These functions are the best way to set the extent size because they reduce the number of extents in a smart large object. The database server tries to allocate the entire smart large object as one extent (if an extent of that size is available in the chunk).

- The EXTENT_SIZE flag in the **-Df** option of the **onspaces** command when you create or alter the sbspace

  Most administrators do not use the **onspaces** EXTENT_SIZE flag because the database server calculates the extent size from heuristics. However, you might consider using the **onspaces** EXTENT_SIZE flag in the following situations:

  - Many one-page extents are scattered throughout the sbspace.
  - Almost all smart large objects are the same length.
- The EXTENT SIZE keyword of the CREATE TABLE statement when you define the CLOB or BLOB column

  Most administrators do not use the EXTENT SIZE keyword when they create or alter a table because the database server calculates the extent size from heuristics. However, you might consider using this EXTENT SIZE keyword if almost all smart large objects are the same length.

Important: For most applications, you should use the values that the database server calculates for the extent size. Do not use the DataBlade® API **mi_lo_specset_extsz** function or the **ifx_lo_specset_extsz** function to set the extent size of the smart large object.

If you know the size of the smart large object, it is recommended that you specify the size in the **DataBlade® API mi_lo_specset_estbytes()** function or **ifx_lo_specset_estbytes()** function instead of in the **onspaces** utility or the CREATE TABLE or the ALTER TABLE statement. These functions are the best way to set the extent size because the database server allocates the entire smart large object as one extent (if it has contiguous storage in the chunk).

Extent sizes over one megabyte do not provide much I/O benefit because the database server performs read and write operations in multiples of 60 kilobytes at the most. However, the database server registers each extent for a smart large object in the metadata area; therefore, especially large smart large objects might have many extent entries. Performance of the database server might degrade when it accesses these extent entries. In this case, you can reduce the number of extent entries in the metadata area if you specify the eventual size of the smart large object in the **mi_lo_specset_estbytes()** function or **ifx_lo_specset_estbytes()** function.

For more information, see Improving metadata I/O for smart large objects on page 168.

Lightweight I/O for smart large objects

Instead of using the buffer pool, the administrator and programmer have the option to use *lightweight I/O*. Lightweight I/O operations use private buffers in the session pool of the virtual portion of shared memory.

By default, smart large objects pass through the buffer pool in the resident portion of shared memory. Although smart large objects have lower priority than other data, the buffer pool can become full when an application accesses many smart large objects. A single application can fill the buffer pool with smart large objects and leave little room for data that other applications might need. In addition, when the database server performs scans of many pages into the buffer pool, the overhead and contention associated with checking individual pages in and out might become a bottleneck.

Important: Use private buffers only when you read or write smart large objects in read or write operations greater than 8080 bytes and you seldom access them. That is, if you have infrequent read or write function calls that read large amounts of data in a single function invocation, lightweight I/O can improve I/O performance.

Related information

The BUFFERPOOL configuration parameter and memory utilization on page 72

Advantages of lightweight I/O for smart large objects

Lightweight I/O provides some performance advantages, because the database server is not using the buffer pool.

Lightweight I/O provides the following advantages:

- Transfers larger blocks of data in one I/O operation

  These I/O blocks can be as large as 60 kilobytes. But the bytes must be adjacent for the database server to transfer them in a single I/O operation.
- Bypasses the overhead of the buffer pool when many pages are read
- Prevents frequently accessed pages from being forced out of the buffer pool when many sequential pages are read for smart large objects

When you use lightweight I/O buffers for smart large objects, the database server might read several pages with one I/O operation. A single I/O operation reads in several smart-large-object pages, up to the size of an extent. For information about when to specify extent size, see Sbspace extents on page 128.

Logging

If you decide to log all write operations on data stored in sbspaces, logical-log I/O activity and memory utilization increases.

For more information, see Configuration parameters that affect sbspace I/O on page 127.

How the Optical Subsystem affects performance

The Optical Subsystem extends the storage capabilities of the database server for simple large objects (TEXT or BYTE data) to write-once-read-many (WORM) optical subsystems. The database server uses a cache in memory to buffer initial TEXT or BYTE data pages requested from the Optical Subsystem.

The memory cache is a common storage area. The database server adds simple large objects requested by any application to the memory cache if the cache has space. To free space in the memory cache, the application must release the TEXT or BYTE data that it is using.

A significant performance advantage occurs when you retrieve TEXT or BYTE data directly into memory instead of buffering that data on disk. Therefore, proper cache sizing is important when you use the Optical Subsystem. You specify the total amount of space available in the memory cache with the OPCACHEMAX configuration parameter. Applications indicate that they require access to a portion of the memory cache when they set the **INFORMIXOPCACHE** environment variable. For details, see INFORMIXOPCACHE, an Optical Subsystem environment variable on page 132.

Simple large objects that cannot fit entirely into the space that remains in the cache are stored in the blobspace that the STAGEBLOB configuration parameter names. This staging area acts as a secondary cache on disk for blobpages that are retrieved from the Optical Subsystem. Simple large objects that are retrieved from the Optical Subsystem are held in the staging area until the transactions that requested them are complete.

The database server administrator creates the staging-area blobspace with the onspaces utility or with ON-Monitor (UNIX™ only.

You can use onstat -O to monitor utilization of the memory cache and STAGEBLOB blobspace. If contention develops for the memory cache, increase the value listed in the configuration file for OPCACHEMAX. (The new value takes effect the next time that the database server starts shared memory.) For a complete description of the Optical Subsystem, see the .

Environment variables and configuration parameters for the Optical Subsystem

The STAGEBLOB and OPCACHEMAX configuration parameters and the **INFORMIXOPCACHE** environment variable affect the performance of the Optical Subsystem.

STAGEBLOB, an Optical Subsystem configuration parameter

The STAGEBLOB configuration parameter identifies the blobspace that is to be used as a staging area for TEXT or BYTE data that is retrieved from the Optical Subsystem, and it activates the Optical Subsystem.

If the configuration file does not list the STAGEBLOB parameter, the Optical Subsystem does not recognize the optical-storage subsystem.

The structure of the staging-area blobspace is the same as all other database server blobspaces. When the database server administrator creates the staging area, it consists of only one chunk, but you can add more chunks as desired. You cannot mirror the staging-area blobspace. The optimal size for the staging-area blobspace depends on the following factors:

- The frequency of simple-large-object storage
- The frequency of simple-large-object retrieval
- The average size of the simple large object to be stored

To calculate the size of the staging-area blobspace, you must estimate the number of simple large objects that you expect to reside there simultaneously and multiply that number by the average simple-large-object size.

Related information

STAGEBLOB configuration parameter on page

OPCACHEMAX, an Optical Subsystem configuration parameter

The OPCACHEMAX configuration parameter specifies the total amount of space that is available for simple-large-object retrieval in the memory cache that the Optical Subsystem uses.

Until the memory cache fills, it stores simple large objects that are requested by any application. Simple large objects that cannot fit in the cache are stored on disk in the blobspace that the STAGEBLOB configuration parameter indicates. You can increase the size of the cache to reduce contention among simple-large-object requests and to improve performance for requests that involve the Optical Subsystem.

Related information

OPCACHEMAX configuration parameter (UNIX) on page

INFORMIXOPCACHE, an Optical Subsystem environment variable

The **INFORMIXOPCACHE** environment variable sets the size of the memory cache that a given application uses for simple-large-object retrieval.

If the value of this variable exceeds the maximum that the OPCACHEMAX configuration parameter specifies, OPCACHEMAX is used instead. If **INFORMIXOPCACHE** is not set in the environment, the cache size is set to OPCACHEMAX by default.

Related information

INFORMIXOPCACHE environment variable on page

Table I/O

One of the most frequent functions that the database server performs is to bring data and index pages from disk into memory. Pages can be read individually for brief transactions and sequentially for some queries. You can configure the number of pages that the database server brings into memory, and you can configure the timing of I/O requests for sequential scans.

You can also indicate how the database server is to respond when a query requests data from a dbspace that is temporarily unavailable.

The following sections describe these methods of reading pages.

For information about I/O for smart large objects, see Factors that affect I/O for smart large objects on page 127.

Sequential scans

When the database server performs a sequential scan of data or index pages, most of the I/O wait time is caused by seeking the appropriate starting page. To dramatically improve performance for sequential scans, you can bring in a number of contiguous pages with each I/O operation.

The action of bringing additional pages along with the first page in a sequential scan is called *read ahead*.

The timing of I/O operations that are needed for a sequential scan is also important. If the scan thread must wait for the next set of pages to be brought in after working its way through each batch, a delay occurs. Timing second and subsequent read requests to bring in pages before they are needed provides the greatest efficiency for sequential scans. The number of pages to bring in and the frequency of read-ahead I/O requests depends on the availability of space in the memory buffers. Read-ahead operations can increase page cleaning to unacceptable levels if too many pages are brought in with each batch or if batches are brought in too often.

Related information

Read-ahead operations on page

Light scans

Some sequential scans of tables can use *light scans* to read the data. A light scan bypasses the buffer pool by utilizing session memory to read directly from disk.

Light scans can provide performance advantages over use of the buffer pool for sequential scans and skip scans of large tables. These advantages include:

- Bypassing the overhead of the buffer pool when many data pages are read
- Preventing frequently accessed pages from being forced out of the buffer pool when many sequential pages are read for a single query.

Light scans occur under these conditions:

- The optimizer chooses a sequential scan or a skip-scan of the table.
- The amount of data in the table exceeds one MB.
- The query meets one of the following locking conditions:
  - The isolation level is Dirty Read (or the database has no transaction logging).
  - The table has at least a shared lock on the entire table and the isolation level is *not* Cursor Stability.

Note: A sequential scan in Repeatable Read isolation automatically acquires a share lock on the table.

Tables that cannot be accessed by light scans

Light scans are only performed on user tables whose data rows are stored in tblspaces. Light scans are not used to access indexes, or to access data stored in blobspaces, smart blob spaces, or partition blobs. Similarly, light scans are not used to access data in the system catalog tables, nor in the tables and pseudotables of system databases like **sysadmin**, **sysmaster**, **sysuser**, and **sysutils**.

Configuration settings that affect light scans

If the BATCHEDREAD_TABLE configuration parameter or the IFX_BATCHEDREAD_TABLE session environment option to the SET ENVIRONMENT statement is set to `0`, light scans are not used to access tables that have variable length rows, or tables where the row length is greater than the pagesize of the dbspace in which the table is contained. A *variable length* row includes tables that have a variable length column, such as VARCHAR, LVARCHAR or NVARCHAR, as well as tables that are compressed.

You can use the IFX_BATCHEDREAD_TABLE session environment option of the SET ENVIRONMENT statement, or the onmode -wm command, to override the setting of the BATCHEDREAD_TABLE configuration parameter for the current session. You can use the onmode -wf command to change the value of BATCHEDREAD_TABLE in the ONCONFIG file.

Example of onstat output during a light scan

If you have a long-running scan, you can view output from the onstat -g scn command to check the progress of the scan, to determine how long the scan will take before it completes, and to see whether the scan is a light scan or a bufferpool scan.

The following example shows some of the output from onstat -g scn for a light scan. The word `Light` in the `Scan Type` field identifies the scan as a light scan.

```
SesID  Thread Partnum  Rowid  Rows Scan'd  Scan Type  Lock Mode  Notes
17     48     300002   207    15           Light                 Forward row lookup
```

Related information

BATCHEDREAD_TABLE configuration parameter on page

onstat -g scn command: Print scan information on page

Unavailable data

Another aspect of table I/O pertains to situations in which a query requests access to a table or fragment in a dbspace that is temporarily unavailable. When the database server determines that a dbspace is unavailable as the result of a disk failure, queries directed to that dbspace fail by default. The database server allows you to specify dbspaces that, when unavailable, can be skipped by queries,

For information about specifying dbspaces that, when unavailable, can be skipped by queries, see How DATASKIP affects table I/O on page 135.

Warning: If a dbspace containing data that a query requests is listed in the DATASKIP configuration parameter and is currently unavailable because of a disk failure, the data that the database server returns to the query can be inconsistent with the actual contents of the database.

Configuration parameters that affect table I/O

The AUTO_READAHEAD configuration parameter changes the automatic read-ahead mode or disables automatic read-ahead for a query. In addition, the DATASKIP configuration parameter enables or disables data skipping.

Automatic read-ahead processing helps improve query performance by issuing asynchronous page requests when HCL OneDB™ detects that the query is encountering I/O. Asynchronous page requests can improve query performance by overlapping query processing with the processing necessary to retrieve data from disk and put it in the buffer pool. You can also use the AUTO_READAHEAD environment option of the SET ENVIRONMENT statement of SQL to enable or disable the value of the AUTO_READAHEAD configuration parameter for a session.

Related information

AUTO_READAHEAD configuration parameter on page

How DATASKIP affects table I/O

The DATASKIP configuration parameter allows you to specify which dbspaces, if any, queries can skip when those dbspaces are unavailable as the result of a disk failure. You can list specific dbspaces and turn data skipping on or off for all dbspaces.

When data skipping is enabled, the database server sets the sixth character in the SQLWARN array to `W`..

Warning: The database server cannot determine whether the results of a query are consistent when a dbspace is skipped. If the dbspace contains a table fragment, the user who executes the query must ensure that the rows within that fragment are not needed for an accurate query result. Turning DATASKIP on allows queries with incomplete data

to return results that can be inconsistent with the actual state of the database. Without proper care, that data can yield incorrect or misleading query results.

Related information

DATASKIP Configuration Parameter on page

SQLWARN array on page

Background I/O activities

Background I/O activities do not service SQL requests directly. Many of these activities are essential to maintain database consistency and other aspects of database server operation. However, they create overhead in the CPU and take up I/O bandwidth.

These overhead activities take time away from queries and transactions. If you do not configure background I/O activities properly, too much overhead for these activities can limit the transaction throughput of your application.

The following list shows some background I/O activities:

- Checkpoints
- Logging
- Page cleaning
- Backup and restore
- Rollback and recovery
- Data replication
- Auditing

Checkpoints occur regardless of whether much database activity occurs; however, they can occur with greater frequency as activity increases. Other background activities, such as logging and page cleaning, occur more frequently as database use increases. Activities such as backups, restores, or fast recoveries occur only as scheduled or under exceptional circumstances.

For the most part, tuning your background I/O activities involves striking a balance between appropriate checkpoint intervals, logging modes and log sizes, and page-cleaning thresholds. The thresholds and intervals that trigger background I/O activity often interact; adjustments to one threshold might shift the performance bottleneck to another.

The following sections describe the performance effects and considerations that are associated with the configuration parameters that affect these background I/O activities.

Configuration parameters that affect checkpoints

The RTO_SERVER_RESTART, CKPTINTVL, LOGSIZE, LOGFILES, PHYSFILE, and ONDBSPACEDOWN configuration parameters affect checkpoints.

RTO_SERVER_RESTART and its effect on checkpoints

The RTO_SERVER_RESTART configuration parameter specifies the amount of time, in seconds, that OneDB has to recover from an unplanned outage.

The performance advantage of enabling this configuration parameter is:

- Enabling fast recovery to meet the RTO_SERVER_RESTART policy by seeding the buffer pool with the data pages required by log replay.

The performance disadvantages of enabling this configuration parameter are:

- Increased physical log activity which might slightly impact transaction performance
- Increased checkpoint frequency, because the physical log space is depleted more quickly (You can increase the size of the physical log to avoid the increase in checkpoint frequency.)

When RTO_SERVER_RESTART is enabled, the database server:

- Attempts to make sure nonblocking checkpoints do not run out of critical resources during checkpoint processing by triggering more frequent checkpoints if transactions might run out of physical or logical log resources, which would cause transaction blocking.
- Ignores the CKPTINTVL configuration parameter.
- Automatically controls checkpoint frequency to meet the RTO policy and to prevent the server from running out of log resources.
- Automatically adjusts the number of AIO virtual processors and cleaner threads and automatically tunes LRU flushing.

The database server prints warning messages in the message log if the server cannot meet the RTO_SERVER_RESTART policy.

Related information

RTO_SERVER_RESTART configuration parameter on page

Automatic checkpoints, LRU tuning, and AIO virtual processor tuning

The database server automatically adjusts checkpoint frequency to avoid transaction blocking. The server monitors physical and logical log consumption along with information about past checkpoint performance. Then, if necessary, the server triggers checkpoints more frequently to avoid transaction blocking.

You can turn off automatic checkpoint tuning by setting onmode -wf AUTO_CKPTS to 0, or setting the AUTO_CKPTS configuration parameter to 0.

Because the database server does not block transactions during checkpoint processing, LRU flushing is relaxed. If the server is not able to complete checkpoint processing before the physical log is full (which causes transaction blocking), and if you cannot increase the size of the physical log, you can configure the server for more aggressive LRU flushing. The increase in

LRU flushing impacts transaction performance, but reduces transaction blocking. If you do not configure the server for more aggressive flushing, the server automatically adjusts LRU flushing to be more aggressive only when the server is unable to find a low priority buffer for page replacement.

When the AUTO_AIOVPS configuration parameter is enabled, the database server automatically increases the number of AIO virtual processors and page-cleaner threads when the server detects that AIO virtual processors are not keeping up with the I/O workload.

If the VPCLASS configuration parameter setting for AIO virtual processors is set to autotune=1, the database server automatically increases the number of AIO virtual processors and page-cleaner threads when the server detects that AIO virtual processors are not keeping up with the I/O workload.

Automatic LRU tuning affects all buffer pools and adjusts lru_min_dirty and lru_max_dirty values in the BUFFERPOOL configuration parameter.

Related information

AUTO_CKPTS configuration parameter on page

AUTO_AIOVPS configuration parameter on page

BUFFERPOOL configuration parameter on page

VPCLASS configuration parameter on page

LRU tuning on page 155

CKPTINTVL and its effect on checkpoints

If the RTO_SERVER_RESTART configuration parameter is not on, the CKPTINTVL configuration parameter specifies the frequency, in seconds, at which the database server checks to determine whether a checkpoint is needed.

When the RTO_SERVER_RESTART configuration parameter is on, the database server ignores the CKPTINTVL configuration parameter. Instead, the server automatically triggers checkpoints in order to maintain the RTO_SERVER_RESTART policy.

The database server can skip a checkpoint if all data is physically consistent when the checkpoint interval expires.

Checkpoints also occur in either of these circumstances:

- Whenever the physical log becomes 75 percent full
- If a high number of dirty partitions exist, even if the physical log is not 75 percent full.

  This occurs because when the database server checks if the physical log is 75 percent full, the server also checks if the following condition is true:

```
(Physical Log Pages Used + Number of Dirty Partitions) >=
(Physical Log Size * 9) /10)
```

A partition, which represents one page going into the physical log during checkpoint processing and has a page that maintains information (such as the number of rows and number of data pages) about the partition, becomes dirty when the partition is updated.

If you set CKPTINTVL to a long interval, you can use physical-log capacity to trigger checkpoints based on actual database activity instead of an arbitrary time unit. However, a long checkpoint interval can increase the time needed for recovery in the event of a failure. Depending on your throughput and data-availability requirements, you can choose an initial checkpoint interval of 5, 10, or 15 minutes, with the understanding that checkpoints might occur more often, depending on physical-logging activity.

The database server writes a message to the message log to note the time that it completes a checkpoint. To read these messages, use **onstat -m**.

Related information

CKPTINTVL configuration parameter on page

LOGSIZE and LOGFILES and their effect on checkpoints

The LOGSIZE and LOGFILES configuration parameters indirectly affect checkpoints because they specify the size and number of logical-log files. A checkpoint can occur when the database server detects that the next logical-log file to become current contains the most-recent checkpoint record.

If you need to free the logical-log file that contains the last checkpoint, the database server must write a new checkpoint record to the current logical-log file. If the frequency with which logical-log files are backed up and freed increases, the frequency at which checkpoints occur increases. Although checkpoints block user processing, they no longer last as long. Because other factors (such as the physical-log size) also determine the checkpoint frequency, this effect might not be significant.

When the dynamic log allocation feature is enabled, the size of the logical log does not affect the thresholds for long transactions as much as it did in previous versions of the database server. For details, see LTXHWM and LTXEHWM and their effect on logging on page 147.

The LOGSIZE, LOGFILES, and LOGBUFF configuration parameters also affect logging I/O activity and logical backups. For more information, see Configuration parameters that affect logging on page 141.

Related information

LOGFILES configuration parameter on page

LOGSIZE configuration parameter on page

Estimate the number of logical-log files on page

Checkpoints and the physical log

The PHYSFILE configuration parameter specifies the size of the initial physical log. A checkpoint occurs when either the physical log becomes 75 percent full or a high number of dirty partitions exist.

The rate at which transactions generate physical log activity can affect checkpoint performance. To avoid transaction blocking during checkpoint processing, consider the size of the physical log and how quickly it fills.

You can enable the database server to expand the size of the physical log as needed to improve performance by creating an extendable plogspace for the physical log.

For example, operations that do not perform updates do not generate before-images. If the size of the database is growing, but applications rarely update the data, little physical logging occurs. In this situation, you might not need a large physical log.

Similarly, you can define a smaller physical log if your application updates the same pages. The database server writes the before-image of only the first update that is made to a page for the following operations:

- Inserts, updates, and deletes for rows that contain user-defined data types (UDTs), smart large objects, and simple large objects
- ALTER statements
- Operations that create or modify indexes (B-tree, R-tree, or user-defined indexes)

Because the physical log is recycled after each checkpoint, the physical log must be large enough to hold before-images from changes between checkpoints. If the database server frequently triggers checkpoints because it runs out of physical log space, consider increasing the size of the physical log.

If you increase the checkpoint interval or if you anticipate increased update activity, you might want to increase the size of the physical log.

The physical log is an important part of maintaining RTO_SERVER_RESTART policy. To ensure that you have an abundance of space, set the size of the physical log to at least 110 percent of the size of all buffer pools.

You can use the onparams utility to change the physical log location and size. You can change the physical log while transactions are active and without restarting the database server.

Related reference

Configuration parameters that affect critical data on page 113

Related information

PHYSFILE configuration parameter on page

Strategy for estimating the size of the physical log on page

Change the physical-log location and size on page

Plogspace on page

ONDBSPACEDOWN and its effect on checkpoints

The ONDBSPACEDOWN configuration parameter specifies the response that the database server makes when an I/O error indicates that a dbspace is down. By default, the database server identifies any dbspace that contains no critical data as `down` and continues processing. Critical data includes the root dbspace, the logical log, or the physical log.

To restore access to that database, you must back up all logical logs and then perform a warm restore on the down dbspace.

The database server halts operation whenever a disabling I/O error occurs on a nonmirrored dbspace that contains critical data, regardless of the setting for ONDBSPACEDOWN. In such an event, you must perform a cold restore of the database server to resume normal database operations.

The value of ONDBSPACEDOWN has no affect on temporary dbspaces. For temporary dbspaces, the database server continues processing regardless of the ONDBSPACEDOWN setting. If a temporary dbspace requires fixing, you can drop and recreate it.

When ONDBSPACEDOWN is set to `2`, the database server continues processing to the next checkpoint and then suspends processing of all update requests. The database server repeatedly retries the I/O request that produced the error until the dbspace is repaired and the request completes or the database server administrator intervenes. The administrator can use **onmode -O** to mark the dbspace `down` and continue processing while the dbspace remains unavailable or use **onmode -k** to halt the database server.

Important: This *2* setting for ONDBSPACEDOWN can affect the performance for update requests severely because they are suspended due to a down dbspace. When you use this setting for ONDBSPACEDOWN, be sure to monitor the status of the dbspaces.

When you set ONDBSPACEDOWN to `1`, the database server treats all dbspaces as though they were critical. Any nonmirrored dbspace that becomes disabled halts normal processing and requires a cold restore. The performance impact of halting and performing a cold restore when any dbspace goes down can be severe.

Important: If you decide to set ONDBSPACEDOWN to *1*, consider mirroring all your dbspaces.

Related information

ONDBSPACEDOWN configuration parameter on page

Configuration parameters that affect logging

The LOGBUFF, PHYSBUFF, LOGFILES, LOGSIZE, DYNAMIC_LOGS, AUTO_LLOG, LTXHWM, LTXEHWM, SESSION_LIMIT_LOGSPACE, SESSION_LIMIT_TXN_TIME, and TEMPTAB_NOLOG configuration parameters affect logging. The LOGBUFF, PHYSBUFF, LOGFILES, LOGSIZE, DYNAMIC_LOGS, LTXHWM, LTXEHWM, and TEMPTAB_NOLOG configuration parameters affect logging.

Logging, checkpoints, and page cleaning are necessary to maintain database consistency. A direct trade-off exists between the frequency of checkpoints or the size of the logical logs and the time that it takes to recover the database in the event of a failure. Therefore, a major consideration when you attempt to reduce the overhead for these activities is the delay that you can accept during recovery.

LOGBUFF and PHYSBUFF and their effect on logging

The LOGBUFF and PHYSBUFF configuration parameters affect logging I/O activity because they specify the respective sizes of the logical-log and physical-log buffers that are in shared memory. The size of these buffers determines how quickly the buffers fill and therefore how often they need to be flushed to disk.

Related information

LOGBUFF configuration parameter on page

PHYSBUFF configuration parameter on page

LOGFILES and its effect on logging

The LOGFILES configuration parameter, which specifies the number of logical-log files, affects logging.

When you initialize or restart the database server, it creates the number of logical-log files that you specify in the LOGFILES configuration parameter.

You might add logical-log files for the following reasons:

- To increase the disk space allocated to the logical log
- To change the size of your logical-log files
- To enable an open transaction to roll back
- As part of moving logical-log files to a different dbspace

Related information

LOGFILES configuration parameter on page

Estimate the number of logical-log files on page

Calculating the space allocated to logical log files

If all of your logical log files are the same size, you can calculate the total space allocated to the logical log files.

To calculate the space allocated to these files, use the following formula:

```
total logical log space = LOGFILES * LOGSIZE
```

If you add logical-log files that are not the size specified by the LOGSIZE configuration parameter, you cannot use the `LOGFILES * LOGSIZE` expression to calculate the size of the logical log. Instead, you need to add the sizes for each individual log file on disk.

Use the **onstat -l** utility to monitor logical-log files.

LOGSIZE and its effect on logging

The LOGSIZE configuration parameter specifies the size of each logical log file. It is difficult to predict how much logical-log space your database server system requires until the system is fully in use.

The size of the logical log space (LOGFILES * LOGSIZE) is determined by these policies:

Recovery time objective (RTO)

This is the length of time you can afford to be without your systems. If your only objective is failure recovery, the total log space only needs to be large enough to contain all the transactions for two checkpoint cycles. When the RTO_SERVER_RESTART configuration parameter is enabled and the server has a combined buffer pool size of less that four gigabytes, you can configure the total log space to 110% of the combined buffer pool sizes. Too much log space does not impact performance; however, too little log space can cause more frequent checkpoints and transaction blocking.

Recovery point objective (RPO)

This describes the age of the data you want to restore in the event of a disaster. If the objective is to make sure transactional work is protected, the optimum LOGSIZE should be a multiple of how much work gets done per RPO unit. Because the database server supports partial log backup, an optimal log size is not critical and a non-optimal log size simply means more frequent log file changes. RPO is measured in units of time. If the business rule is that the system cannot lose more than ten minutes of transactional data if a complete site disaster occurs, then a log backup should occur every ten minutes.

You can use the Scheduler, which manages and executes scheduled administrative tasks, to set up automatic log backup.

Long Transactions

If you have long transactions that require a large amount of log space, you should allocate that space for the logs. Inadequate log space impacts transaction performance.

Choose a log size based on how much logging activity occurs and the amount of risk in case of catastrophic failure. If you cannot afford to lose more than an hour's worth of data, create many small log files that each hold an hour's worth of transactions. Turn on continuous-log backup. Small logical-log files fill sooner, which means more frequent logical-log backups.

If your system is stable with high logging activity, choose larger logs to improve performance. Continuous-log backups occur less frequently with large log files. Also consider the maximum transaction rates and speed of the backup devices. Do not let the whole logical log fill. Turn on continuous-log backup and leave enough room in the logical logs to handle the longest transactions.

The backup process can hinder transaction processing that involves data located on the same disk as the logical-log files. If enough logical-log disk space is available, however, you can wait for periods of low user activity before you back up the logical-log files.

Related information

LOGSIZE configuration parameter on page

The Scheduler on page

Estimating logical-log size when logging dbspaces

To estimate the size of logical logs, use a formula or **onstat -u** information.

Use the following formula to obtain an initial estimate for LOGSIZE in kilobytes:

```
LOGSIZE = (connections * maxrows * rowsize) / 1024) / LOGFILES
```

In this formula:

- *connections* is the maximum number of connections for all network types specified in the sqlhosts information by one or more NETTYPE parameters. If you configured more than one connection by setting multiple NETTYPE configuration parameters in your configuration file, sum the **users** fields for each NETTYPE parameter, and substitute this total for *connections* in the preceding formula.
- *maxrows* is the largest number of rows to be updated in a single transaction.
- *rowsize* is the average size of a row in bytes. You can calculate *rowsize* by adding up the length (from the **syscolumns** system catalog table) of the columns in a row.
- *1024* is a necessary divisor because you specify LOGSIZE in kilobytes.

To obtain a better estimate during peak activity periods, execute the onstat -u command. The last line of the onstat -u output contains the maximum number of concurrent connections.

You need to adjust the size of the logical log when your transactions include simple large objects or smart large objects, as the following sections describe.

You also can increase the amount of space devoted to the logical log by adding another logical-log file.

Related information

Adding logical-log files manually on page

Estimating the logical-log size when logging simple large objects

To obtain better overall performance for applications that perform frequent updates of TEXT or BYTE data in blobspaces, reduce the size of the logical log.

Blobpages cannot be reused until the logical log to which they are allocated is backed up. When TEXT or BYTE data activity is high, the performance impact of more frequent checkpoints is balanced by the higher availability of free blobpages.

When you use volatile blobpages in blobspaces, smaller logs can improve access to simple large objects that must be reused. Simple large objects cannot be reused until the log in which they are allocated is flushed to disk. In this case, you can justify the cost in performance because those smaller log files are backed up more frequently.

Estimating the logical-log size when logging smart large objects

If you plan to log smart-large-object user data, you must ensure that the log size is considerably larger than the amount of data being written. Smart-large-object metadata is always logged even if the smart large objects are not logged.

Use the following guidelines when you log smart large objects:

- If you are appending data to a smart large object, the increased logging activity is roughly equal to the amount of data written to the smart large object.
- If you are updating a smart large object (overwriting data), the increased logging activity is roughly twice the amount of data written to the smart large object. The database server logs both the before-image and after-image of a smart large object for update transactions. When updating the smart large objects, the database server logs only the updated parts of the before and after image.
- Metadata updates affect logging less. Even though metadata is always logged, the number of bytes logged is usually much smaller than the smart large objects.

DYNAMIC_LOGS and its effect on logging

The dynamic log file allocation feature prevents hanging problems that are caused by rollbacks of a long transaction because the database server does not run out of log space. The DYNAMIC_LOGS configuration parameter specifies whether the dynamic log file allocation feature is off, on, or causes the server to pause to allow the manual addition of a logical log file.

Dynamic log allocation allows you to do the following actions:

- Add a logical log file while the system is active, even during fast recover.
- Insert a logical log file immediately after the current log file, instead of appending it to the end.
- Immediately access the logical log file even if the root dbspace is not backed up.

The default value for the DYNAMIC_LOGS configuration parameter is `2`, which means that the database server automatically allocates a new logical log file after the current log file when it detects that the next log file contains an open transaction. The database server automatically checks if the log after the current log still contains an open transaction at the following times:

- Immediately after it switches to a new log file while writing log records (not while reading and applying log records)
- At the beginning of the transaction cleanup phase which occurs as the last phase of logical recovery

  Logical recovery happens at the end of fast recovery and at the end of a cold restore or roll forward.
- During transaction cleanup (rollback of open transactions), a switch to a new log file log might occur

  The database server also checks after this switch because it is writing log records for the rollback.

When you use the default value of `2` for DYNAMIC_LOGS, the database server determines the location and size of the new logical log for you:

- The database server uses the following criteria to determine on which disk to allocate the new log file:
  - Favor mirrored dbspaces
  - Avoid root dbspace until no other critical dbspace is available
  - Least favored space is unmirrored and noncritical dbspaces
- The database server uses the average size of the largest log file and the smallest log file for the size of the new logical log file. If not enough contiguous disk space is available for this average size, the database server searches for space for the next smallest average size. The database server allocates a minimum of 200 kilobytes for the new log file.

If you want to control the location and size of the additional log file, set DYNAMIC_LOGS to `1`. When the database server switches log files, it still checks if the next active log contains an open transaction. If it does find an open transaction in the next log to be active, it does the following actions:

- Issues alarm event 27 (log required)
- Writes a warning message to the online log
- Pauses to wait for the administrator to manually add a log with the **onparams -a -i** command-line option

You can write a script that will execute when alarm event 27 occurs to execute **onparams -a -i** with the location you want to use for the new log. Your script can also execute the **onstat -d** command to check for adequate space and execute the **onparams -a -i** command with the location that has enough space. You must use the **-i** option to add the new log right after the current log file.

If you set DYNAMIC_LOGS to 0, the database server still checks whether the next active log contains an open transaction when it switches log files. If it does find an open transaction in the next log to be active, it issues the following warning:

```
WARNING: The oldest logical log file (%d) contains records
from an open transaction (0x%p), but the Dynamic Log
Files feature is turned off.
```

Related information

DYNAMIC_LOGS configuration parameter on page

Fast recovery on page

AUTO_LLOG and its effect on logging

Insufficient logical logs can affect performance by triggering frequent checkpoints, blocking checkpoints, or long checkpoints. The AUTO_LLOG configuration parameter controls whether the database server automatically adds logical logs to improve performance.

If you created a server during installation, the AUTO_LLOG configuration parameter is enabled automatically. Otherwise, you can edit the value of the AUTO_LLOG configuration parameter.

If the AUTO_LLOG configuration parameter is enabled, the database server automatically adds logical log files under the following circumstances:

- When a substantial portion of the last 20 checkpoints were caused by logical logs filling up
- When inadequate logical log space causes a blocking checkpoint
- When inadequate logical log space causes a long checkpoint

The AUTO_LLOG configuration parameter also specifies the dbspace for new logical log files and the maximum size of all logical log files before the server stops adding logical logs for performance. The following guidelines show estimates of the maximum amount of space for logical logs that you might need, depending on the number of concurrent users who access your database server:

- 1 - 100 users: 200 MB
- 101 - 500 users: 5 MB
- 501 - 1000 users: 1 GB
- More than 1000 users: 2 GB

The settings of the AUTO_LLOG configuration parameter and the DYNAMIC_LOGS configuration parameters do not interact.

Related reference

AUTO_LLOG configuration parameter

LTXHWM and LTXEHWM and their effect on logging

The LTXHWM and LTXEHWM configuration parameters define long transaction watermarks.

After the release of the dynamic log file feature, long transaction high watermarks are no longer as critical, because the server does not run out of log space unless you use up the physical disk space available on the system. The LTXHWM parameter still indicates how full the logical log is when the database server starts to check for a possible long transaction and to roll it back. LTXEHWM still indicates the point at which the database server suspends new transaction activity to locate and roll back a long transaction. These events are usually rare, but if they occur, they can indicate a serious problem within an application.

Under normal operations, use the default values for LTXHWM and LTXEHWM. However, you might want to change these default values for one of the following reasons:

- To allow other transactions to continue update activity (which requires access to the log) during the rollback of a long transaction

  In this case, you increase the value of LTXEHWM to raise the point at which the long transaction rollback has exclusive access to the log.

- To run scheduled transactions of unknown length, such as large loads that are logged

  In this case, you increase the value of LTXHWM so that the transaction has a chance to complete before it reaches the high watermark.

Related information

LTXEHWM configuration parameter on page

LTXHWM configuration parameter on page

TEMPTAB_NOLOG and its effect on logging

The TEMPTAB_NOLOG configuration parameter allows you to disable logging on temporary tables. You can do this to improve performance and to prevent OneDB from transferring temporary tables when using High-Availability Data Replication (HDR).

To disable logging on temporary tables, set the TEMPTAB_NOLOG configuration parameter to `1`.

To enable logging on temporary tables for primary server and to disable logging on temporary tables for secondary servers(HDR, RSS and SDS), set the TEMPTAB_NOLOG configuration parameter to `2`.

Related information

TEMPTAB_NOLOG configuration parameter on page

SESSION_LIMIT_LOGSPACE and its effect on logging

The SESSION_LIMIT_LOGSPACE configuration parameter specifies the maximum amount of log space that a session can use for individual transactions, and can prevent individual sessions from monopolizing the logical log.

SESSION_LIMIT_LOGSPACE does not apply to a user who holds administrative privileges, such as user **informix** or a DBSA user.

Related information

SESSION_LIMIT_LOGSPACE configuration parameter on page

SESSION_LIMIT_TXN_TIME configuration parameter on page

SESSION_LIMIT_TXN_TIME and its effect on logging

The SESSION_LIMIT_TXN_TIME configuration parameter limits how much time a transaction can run in a session, and can prevent individual session transactions from monopolizing the logical log.

The database server terminates a transaction that exceeds the SESSION_LIMIT_TXN_TIME limit, and produces an error in the database server message log.

SESSION_LIMIT_TXN_TIME does not apply to a user who holds administrative privileges, such as user **informix** or a DBSA user.

Configuration parameters that affect page cleaning

Several configuration parameters, including the CLEANERS and RTO_SERVER_RESTART configuration parameters, affect page cleaning. If pages are not cleaned often enough, an **sqlexec** thread that performs a query might be unable to find the available pages that it needs.

If the **sqlexec** thread cannot find the available pages that it needs, the thread initiates a *foreground write* and waits for pages to be freed. Foreground writes impair performance, so you should avoid them. To reduce the frequency of foreground writes, increase the number of page cleaners or decrease the threshold for triggering a page cleaning.

Use **onstat -F** to monitor the frequency of foreground writes.

The following configuration parameters affect page cleaning:

- BUFFERPOOL, which contains **lrus**, **lru_max_dirty**, and **lru_min_dirty** values

  Information that was specified with the BUFFERS, LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY configuration parameters before Version 10.0 is now specified using the BUFFERPOOL configuration parameter.
- CLEANERS
- RTO_SERVER_RESTART

CLEANERS and its effect on page cleaning

The CLEANERS configuration parameter indicates the number of page-cleaner threads to run. For installations that support fewer than 20 disks, one page-cleaner thread is recommended for each disk that contains database server data. For installations that support between 20 and 100 disks, one page-cleaner thread is recommended for every two disks. For larger installations, one page-cleaner thread is recommended for every four disks.

If you increase the number of LRU queues, you must increase the number of page-cleaner threads proportionally.

Related information

CLEANERS configuration parameter on page

BUFFERPOOL and its effect on page cleaning

The BUFFERPOOL configuration parameter specifies the number of least recently used (LRU) queues to set up within the shared-memory buffer pool. The buffer pool is distributed among LRU queues. Configuring more LRU queues allows more page cleaners to operate and reduces the size of each LRU queue.

For a single-processor system, set the lrus field of the BUFFERPOOL configuration parameter to a minimum of 8. For multiprocessor systems, set the lrus field to a minimum of 8 or to the number of CPU VPs, whichever is greater.

For a single-processor system, set the lrus field of the BUFFERPOOL configuration parameter to a minimum of 4. For multiprocessor systems, set the lrus field to a minimum of 4 or to the number of CPU VPs, whichever is greater.

The lrus, lru_max_dirty, and lru_min_dirty values control how often pages are flushed to disk between checkpoints. Automatic LRU tuning, as set by the AUTO_LRU configuration parameter, affects all buffer pools and adjusts the lru_min_dirty and lru_max_dirty values in the BUFFERPOOL configuration parameter.

If you increase the lru_max_dirty and lru_min_dirty values to improve transaction throughput, do not change the gap between the lru_max_dirty and lru_min_dirty.

When the buffer pool is very large and transaction blocking is occurring during checkpoint processing, look in the message log to determine which resource is triggering transaction blocking. If the physical or logical log is critically low and triggers transaction blocking, increase the size of the resource that is causing the transaction blocking. If you cannot increase the size of the resource, consider making LRU flushing more aggressive by decreasing the lru_min_dirty and lru_max_dirty settings so that the server has fewer pages to flush to disk during checkpoint processing.

To monitor the percentage of dirty pages in LRU queues, use the onstat -R command. When the number of dirty pages consistently exceeds the lru_max_dirty limit, you have too few LRU queues or too few page cleaners. First, use the BUFFERPOOL configuration parameter to increase the number of LRU queues. If the percentage of dirty pages still exceeds the lru_max_dirty limit, update the CLEANERS configuration parameter to increase the number of page cleaners.

Related information

The BUFFERPOOL configuration parameter and memory utilization on page 72

BUFFERPOOL configuration parameter on page

Number of LRU queues to configure on page

RTO_SERVER_RESTART and its effect on page cleaning

The RTO_SERVER_RESTART configuration parameter allows you to use recovery time objective (RTO) standards to set the amount of time, in seconds, that OneDB has to recover from a problem after you restart OneDB and bring it into online or quiescent mode.

When this configuration parameter is enabled, the database server automatically adjusts the number of AIO virtual processors and cleaner threads and automatically tunes LRU flushing.

Use the AUTO_LRU_TUNING configuration parameter to specify whether automatic LRU tuning is enabled or disabled when the server starts.

Related information

RTO_SERVER_RESTART configuration parameter on page

AUTO_LRU_TUNING configuration parameter on page

Configuration parameters that affect backup and restore

Four configuration parameters that affect backup and restore on all operating systems also affect background I/O. Additional configuration parameters affect backup and restore on UNIX™.

The following configuration parameters affect backup and restore on all operating systems:

- BAR_MAX_BACKUP
- BAR_NB_XPORT_COUNT
- BAR_PROGRESS_FREQ
- BAR_XFER_BUF_SIZE

In addition, the following configuration parameters affect backup and restore on UNIX™:

- LOG_BACKUP_MODE

ON-Bar configuration parameters

BAR_MAX_BACKUP, BAR_NB_XPORT_COUNT, BAR_PROGRESS_FREQ, and BAR_XFER_BUF_SIZE are some ON-Bar configuration parameters that affect background I/O.

The BAR_MAX_BACKUP configuration parameter specifies the maximum number of backup processes per ON-Bar command. This configuration parameter also defines the degree of parallelism, determining how many processes start to run concurrently, including processes for backing up and restoring a whole system. When the number of running processes is reached, further processes start only when a running process completes its operation.

BAR_NB_XPORT_COUNT specifies the number of shared-memory data buffers for each backup or restore process.

BAR_PROGRESS_FREQ specifies, in minutes, how frequently the backup or restore progress messages display in the activity log.

BAR_XFER_BUF_SIZE specifies the size, in pages, of the buffers.

Related information

BAR_MAX_BACKUP configuration parameter on page

BAR_NB_XPORT_COUNT configuration parameter on page

BAR_PROGRESS_FREQ configuration parameter on page

BAR_XFER_BUF_SIZE configuration parameter on page

Configuration parameters that affect rollback and recovery

The OFF_RECVRY_THREADS, ON_RECVRY_THREADS, PLOG_OVERFLOW_PATH, and RTO_SERVER_RESTART configuration parameters affect recovery. The LOW_MEMORY_RESERVE configuration parameter reserves a specific amount of memory, in kilobytes, for the database server to use when critical activities, such as rollback activities, are needed.

OFF_RECVRY_THREADS and ON_RECVRY_THREADS and their effect on fast recovery

The OFF_RECVRY_THREADS configuration parameter specifies the number of recovery threads that operate when the database server performs a cold restore or fast recovery. The setting of ON_RECVRY_THREADS specifies the number of recovery threads that operate when the database server performs a warm restore.

To improve the performance of fast recovery, increase the number of recovery threads with the OFF_RECVRY_THREADS configuration parameter. When fast recovery begins, the database server creates an LGR memory pool and allocates approximately 100 KB from this pool for each recovery thread. The LGR pool and its memory are freed when fast recovery completes. Because secondary servers in a high-availability cluster are almost always in fast recovery mode, the LGR memory pool is almost always present on secondary servers.

Follow these guidelines when you set the OFF_RECVRY_THREADS configuration parameter:

- If you have enough shared memory, set the number of threads to the number of tables or fragments that are frequently updated. Balance the number of threads with the amount of shared memory.
- On a single-CPU computer, set the number of threads to 10 - 30 or 40. The cost of too many threads can outweigh the advantages of parallel operations.

A warm restore takes place concurrently with other database operations. To reduce the impact of the warm restore on other users, you can allocate fewer threads to it than you might allocate to a cold restore. However, to replay logical-log transactions in parallel during a warm restore, specify more threads with the ON_RECVRY_THREADS configuration parameter.

Related information

OFF_RECVRY_THREADS configuration parameter on page

ON_RECVRY_THREADS configuration parameter on page

PLOG_OVERFLOW_PATH and its effect on fast recovery

The PLOG_OVERFLOW_PATH configuration parameter specifies the location of a disk file (named **plog_extend.servernum**) that the database server uses if the physical log file overflows during fast recovery.

The database server removes the **plog_extend.servernum** file when the first checkpoint is performed during a fast recovery.

Related information

PLOG_OVERFLOW_PATH configuration parameter on page

RTO_SERVER_RESTART and its effect on fast recovery

The RTO_SERVER_RESTART configuration parameter enables you to use recovery time objective (RTO) standards to set the amount of time, in seconds, that OneDB has to recover from a problem after you restart OneDB and bring it into online or quiescent mode.

Related information

RTO_SERVER_RESTART configuration parameter on page

The LOW_MEMORY_RESERVE configuration parameter and memory utilization

The LOW_MEMORY_RESERVE configuration parameter reserves a specific amount of memory, in kilobytes, for the database server to use when critical activities are needed and the server has limited free memory.

If you enable the new LOW_MEMORY_RESERVE configuration parameter by setting it to a specified value in kilobytes, critical activities, such as rollback activities, can complete even when you receive out-of-memory errors.

Related information

LOW_MEMORY_RESERVE configuration parameter on page

onstat -g seg command: Print shared memory segment statistics on page

Configuration parameters that affect data replication and auditing

Data replication and auditing are optional. If you use these features, you can set configuration parameters that affect data-replication performance and auditing performance.

To obtain immediate performance improvements, you can disable these features, provided that the operating requirements for your system allow you to do so.

Configuration parameters that affect data replication

Synchronized data replication can increase the amount of time it take longer to free the log buffer after a log flush. The DRINTERVAL, DRTIMEOUT, and HDR_TXN_SCOPE configuration parameters can adjust synchronization and system performance.

The DRINTERVAL configuration parameter indicates whether the data-replication buffer is flushed synchronously or asynchronously to the secondary database server. If this parameter is set to flush asynchronously, it specifies the interval between flushes. Each flush impacts the CPU and sends data across the network to the secondary database server.

If the DRINTERVAL configuration parameter is set to 0, the synchronization mode that is specified by the HDR_TXN_SCOPE configuration parameter is used. The HDR_TXN_SCOPE configuration parameter specifies whether HDR replication is fully synchronous, nearly synchronous, or asynchronous.

- In fully synchronous mode, transactions require acknowledgement of completion on the HDR secondary server before they can complete.
- In asynchronous mode, transactions do not require acknowledgement of being received or completed on the HDR secondary server before they can complete.
- In nearly synchronous mode, transactions require acknowledgement of being received on the HDR secondary server before they can complete.

The DRTIMEOUT configuration parameter specifies the interval for which either database server waits for a transfer acknowledgment from the other. If the primary database server does not receive the expected acknowledgment, it adds the transaction information to the file named in the DRLOSTFOUND configuration parameter. If the secondary database server receives no acknowledgment, it changes the data-replication mode as the DRAUTO configuration parameter specifies.

Related information

DRINTERVAL configuration parameter on page

DRTIMEOUT configuration parameter on page

DRLOSTFOUND configuration parameter on page

DRAUTO configuration parameter on page

HDR_TXN_SCOPE configuration parameter on page

onstat -g dri command: Print high-availability data replication information on page

Replication of primary-server data to secondary servers on page

Fully synchronous mode for HDR replication on page

Nearly synchronous mode for HDR replication on page

Asynchronous mode for HDR replication on page

Configuration parameters that affect auditing

The ADTERR and ADTMODE configuration parameters affect auditing performance.

The ADTERR configuration parameter specifies whether the database server is to halt processing for a user session for which an audit record encounters an error. When ADTERR is set to halt such a session, the response time for that session appears to degrade until one of the successive attempts to write the audit record succeeds.

The ADTMODE configuration parameter enables or disables auditing according to the audit records that you specify with the **onaudit** utility. Records are written to files in the directory that the AUDITPATH parameter specifies. The AUDITSIZE parameter specifies the size of each audit-record file.

The effect of auditing on performance is largely determined by the auditing events that you choose to record. Depending on which users and events are audited, the impact of these configuration parameters can vary widely.

Infrequent events, such as requests to connect to a database, have low performance impact. Frequent events, such as requests to read any row, can generate a large amount of auditing activity. The more users for whom such frequent events are audited, the greater the impact on performance.

Related information

ADTERR configuration parameter on page

ADTMODE configuration parameter on page

Auditing data security on page

LRU tuning

The LRU settings for flushing each buffer pool between checkpoints are not critical to checkpoint performance. The LRU settings are necessary only for maintaining enough clean pages for page replacement.

The default settings for LRU flushing are 50 percent for **lru_min_dirty** and 60 percent for **lru_max_dirty**.

If your database server has been configured for more aggressive LRU flushing because of checkpoint performance, you can decrease the LRU flushing at least to the default values.

The database server automatically tunes LRU flushing when the AUTO_LRU_TUNING configuration parameter is on and in the following cases:

- A page replacement is forced to perform a foreground write in order to find an empty page. In this case, LRU flushing is adjusted to be 5 percent more aggressive for the specific bufferpool where the foreground write took place.
- A page replacement is forced to use a buffer that is marked as high priority, meaning it is frequently accessed. In this case, LRU flushing is adjusted to be one (1) percent more aggressive for the specific bufferpool where the page replacement using high priority buffer took place.
- If the RTO_SERVER_RESTART configuration parameter is on and the time it takes to flush the bufferpool is longer than the recovery time objective, LRU flushing is adjusted to be 10 percent more aggressive for all bufferpools.

After a checkpoint has occurred, if a page replacement performed a foreground write during the previous checkpoint interval, the database server increases the LRU settings by 5 percent and continues to increase the LRU flushing at each subsequent checkpoint until the foreground write stops or until the **lru_max_dirty** for a given buffer pool falls below 10 percent. For example, if a page replacement performs a foreground write and the LRU settings for a buffer pool are `80` and `90`, the database server adjusts these to `76` and `85.5`.

In addition to foreground writes, LRU flushing is tuned more aggressively whenever a page fault replaces high priority buffers and non-high priority buffers are on the modified LRU queue. Automatic LRU adjustments only make LRU flushing more aggressive; they do not decrease LRU flushing. Automatic LRU adjustments are not permanent and are not recorded in the ONCONFIG file.

LRU flushing is reset to the values contained in the ONCONFIG file on which the database server starts.

The AUTO_LRU_TUNING configuration parameter specifies whether automatic LRU tuning is enabled or disabled when the server starts.

Related information

Automatic checkpoints, LRU tuning, and AIO virtual processor tuning on page 137

AUTO_LRU_TUNING configuration parameter on page

RTO_SERVER_RESTART configuration parameter on page

Table performance considerations

Some performance issues are associated with unfragmented tables and table fragments.

Issues include:

- Table placement on disk to increase throughput and reduce contention
- Space estimates for tables, blobpages, sbspaces, and extents
- Changes to tables that add or delete historical data
- Denormalization of the database to reduce overhead

Placing tables on disk

Tables that the database server supports reside on one or more portions of one or more disks. You control the placement of a table on disk when you create it by assigning it to a dbspace.

Tables that the database server supports reside on one or more portions of a disk or disks. You control the placement of a table on disk when you create it by assigning it to a dbspace. A dbspace consists of one or more chunks. Each chunk corresponds to all or part of a disk partition. When you assign chunks to dbspaces, you make the disk space in those chunks available for storing tables or table fragments.

When you configure chunks and allocate them to dbspaces, you must relate the size of the dbspaces to the tables or fragments that each dbspace is to contain. To estimate the size of a table, follow the instructions in Estimating table size on page 160.

The database administrator (DBA) who is responsible for creating a table assigns that table to a dbspace in one of the following ways:

- By using the IN DBSPACE clause of the CREATE TABLE statement
- By using the dbspace of the current database

  The most recent DATABASE or CONNECT statement that the DBA issues before issuing the CREATE TABLE statement sets the current database.

The DBA can fragment a table across multiple dbspaces, as described in Planning a fragmentation strategy on page 260, or use the ALTER FRAGMENT statement to move a table to another dbspace. The ALTER FRAGMENT statement provides the simplest method for altering the placement of a table. However, the table is unavailable while the database server processes the alteration. Schedule the movement of a table or fragment at a time that affects the fewest users.

Related information

ALTER FRAGMENT statement on page

LOAD statement on page

UNLOAD statement on page

The onunload and onload utilities on page

Moving data with external tables on page

CREATE EXTERNAL TABLE Statement on page

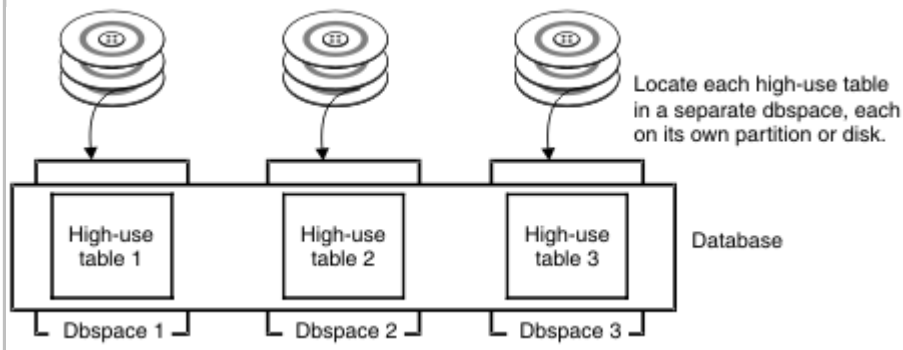
Isolating high-use tables

You can place a table with high I/O activity on a dedicated disk device. Doing this reduces contention for the data that is stored in that table.

When disk drives have different performance levels, you can put the tables with the highest use on the fastest drives. Placing two high-use tables on separate disk devices reduces competition for disk access when the two tables experience frequent, simultaneous I/O from multiple applications or when joins are formed between them.

To isolate a high-use table on its own disk device, assign the device to a chunk, assign that chunk to a dbspace, and then place the table in the dbspace that you created. Figure 16: Isolating high-use tables on page 157 shows three high-use tables, each in a separate dbspace, placed on three disks.

Figure 16. Isolating high-use tables

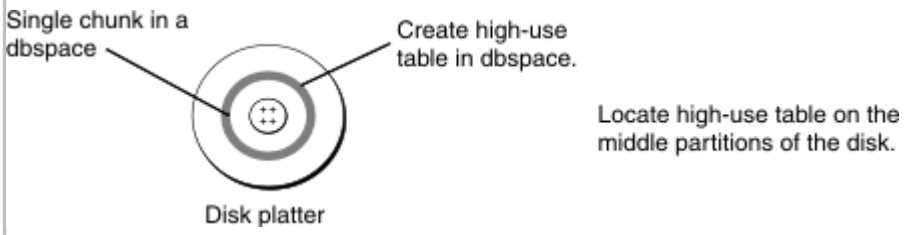


Placing high-use tables on middle partitions of disks

To minimize disk-head movement, place the most frequently accessed data on partitions close to the middle band of the disk (not near the center and not near the edge). This approach minimizes disk-head movement to reach data in the high-demand table.

The following figure shows the placement of the most frequently accessed data on partitions close to the middle band of the disk.

Figure 17. Disk platter with high-use table located on middle Partitions



To place high-use tables on the middle partition of the disk, create a raw device composed of cylinders that reside midway between the spindle and the outer edge of the disk. (For instructions on how to create a raw device, see the *HCL OneDB™ Administrator's Guide* for your operating system.) Allocate a chunk, associating it with this raw device, as your *HCL OneDB™ Administrator's Reference* describes. Then create a dbspace with this same chunk as the initial and only chunk. When you create a high-use table, place the table in this dbspace.

Using multiple disks

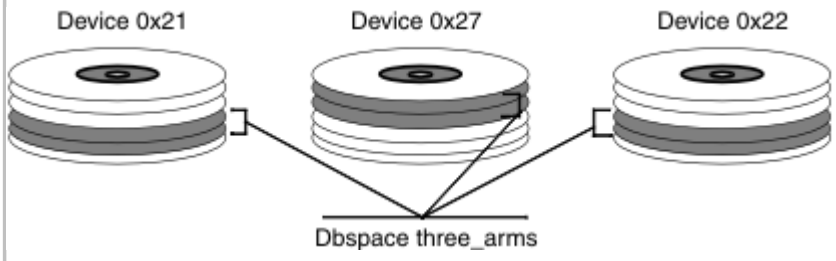
You can use multiple disks for dbspaces, logical logs, temporary tables, and sort files.

Using multiple disks for a dbspace

Using multiple disks for a dbspace helps to distribute I/O across dbspaces that contain several small tables.

A dbspace can include multiple chunks, and each chunk can represent a different disk. The maximum size for a chunk is 4 terabytes. This arrangement allows you to distribute data in a dbspace over multiple disks. Figure 18: A dbspace distributed over three disks on page 158 shows a dbspace distributed over three disks.

Figure 18. A dbspace distributed over three disks



Because you cannot use this type of distributed dbspace for parallel database queries (PDQ), you should use the table-fragmentation techniques described in Distribution schemes on page 265 to partition large, high-use tables across multiple dbspaces.

Using multiple disks for logical logs

You can distribute logical logs in different dbspaces on multiple disks in round-robin fashion to improve logical backup performance. This scheme allows the database server to back up logs on one disk, while performing logging operations on the other disks.

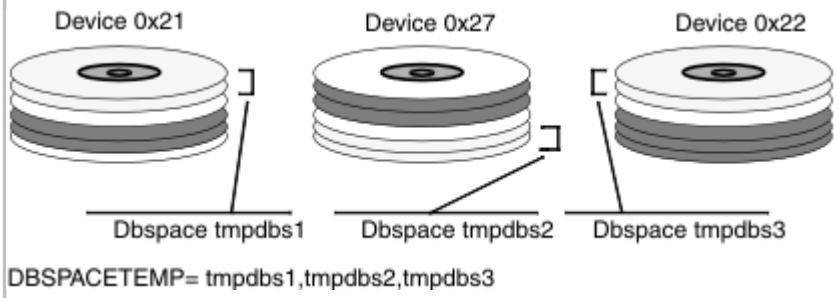
Keep your logical logs and the physical log on separate devices to improve performance by decreasing I/O contention on a single device. The logical and physical logs are created in the root dbspace when the database server is initialized. After initialization, you can move them to other dbspaces.

Spreading temporary tables and sort files across multiple disks

You can spread the I/O associated with temporary tables and sort files across multiple disks, after defining dbspaces for temporary tables and sort files. This can improve performance for applications that require a large amount of temporary space for temporary tables or large sort operations.

To define several dbspaces for temporary tables and sort files, use **onspaces -t**. When you place these dbspaces on different disks and list them in the DBSPACETEMP configuration parameter, you spread the I/O associated with temporary tables and sort files across multiple disks, as Figure 19: Dbspaces for temporary tables and sort files on page 159 illustrates. You can list dbspaces that contain regular tables in DBSPACETEMP.

Figure 19. Dbspaces for temporary tables and sort files



Users can specify their own lists of dbspaces for temporary tables and sort files with the **DBSPACETEMP** environment variable. For details, see Configure dbspaces for temporary tables and sort files on page 114.

Backup and restore considerations when placing tables on disks

When you decide where to place your tables or fragments, remember that if a device that contains a dbspace fails, all tables or table fragments in that dbspace are rendered inaccessible, even though tables and fragments in other dbspaces are accessible. The need to limit data unavailability in the event of a disk failure might influence which tables you group together in a particular dbspace.

Although you must perform a cold restore if a dbspace that contains critical data fails, you need only perform a warm restore if a noncritical dbspace fails. The desire to minimize the impact of cold restores might influence the dbspace that you use to store critical data.

Factors affecting the performance of nonfragmented tables and table fragments

Numerous factors affect the performance of an individual table or table fragment. These include the placement of the table or fragment, the size of the table or fragment, the indexing strategy that was used, the size and placement of table extents with respect to one another, and the frequency of access to the table.

Estimating table size

You can calculate the approximate sizes (in disk pages) of tables.

For a description of size calculations for indexes, see Estimating index pages on page 209.

The disk pages allocated to a table are collectively referred to as a *tblspace*. The tblspace includes data pages. A separate tblspace includes index pages. If simple large objects (TEXT or BYTE data) are associated with a table that is not stored in an alternative dbspace, pages that hold simple large objects are also included in the tblspace.

The tblspace does not correspond to any fixed region within a dbspace. The data extents and indexes that make up a table can be scattered throughout the dbspace.

The size of a table includes all the pages within the tblspace: data pages and pages that store simple large objects. Blobpages that are stored in a separate blobspace are not included in the tblspace and are not counted as part of the table size.

The size of a table includes all the pages within the tblspace: data pages and pages that store simple large objects. Blobpages that are stored in a separate blobspace or on an optical subsystem are not included in the tblspace and are not counted as part of the table size.

The following sections describe how to estimate the page count for each type of page within the tblspace.

Tip: If an appropriate sample table exists, or if you can build a sample table of realistic size with simulated data, you do not need to make estimates. You can run oncheck -pt to obtain exact numbers.

Estimating data pages

How you estimate the data pages of a table depends on whether that table contains fixed-length or variable-length rows.

Estimating tables with fixed-length rows

You can estimate the size (in pages) of a table with fixed-length rows. A table with fixed-length rows has no columns of the VARCHAR or NVARCHAR data type.

About this task

Perform the following steps to estimate the size (in pages) of a table with fixed-length rows.

To estimate the page size, row size, number of rows, and number of data pages:

1. Use **onstat -b** to obtain the size of a page.

   The **buffer size** field in the last line of this output displays the page size.
2. Subtract 28 from this amount to account for the header that appears on each data page.
   The resulting amount is referred to as *pageuse*.

3. To calculate the size of a row, add the widths of all the columns in the table definition. TEXT and BYTE columns each use 56 bytes.
   If you have already created your table, you can use the following SQL statement to obtain the size of a row:

   ```
   SELECT rowsize FROM systables WHERE tabname =
      'table-name';
   ```

4. Estimate the number of rows that the table is expected to contain.
   This number is referred to as *rows*. The procedure for calculating the number of data pages that a table requires differs depending on whether the row size is less than or greater than *pageuse*.
5. If the size of the row is less than or equal to *pageuse*, use the following formula to calculate the number of data pages.
   The **trunc()** function notation indicates that you are to round down to the nearest integer.

   ```
   data_pages = rows / trunc(pageuse/(rowsize + 4))
   ```

   The maximum number of rows per page is 255, regardless of the size of the row.

   Important: Although the maximum size of a row that the database server accepts is approximately 32 kilobytes, performance degrades when a row exceeds the size of a page. For information about breaking up wide tables for improved performance, see Denormalize the data model to improve performance on page 201.

6. If the size of the row is greater than *pageuse*, the database server divides the row between pages.
   The page that contains the initial portion of a row is called the *home page*. Pages that contains subsequent portions of a row are called *remainder pages*. If a row spans more than two pages, some of the remainder pages are completely filled with data from that row. When the trailing portion of a row uses less than a page, it can be combined with the trailing portions of other rows to fill out the partial remainder page. The number of data pages is the sum of the home pages, the full remainder pages, and the partial remainder pages.
   a. Calculate the number of home pages.

      The number of home pages is the same as the number of rows:

      ```
      homepages = rows
      ```

   b. Calculate the number of full remainder pages.

      First calculate the size of the row remainder with the following formula:

      ```
      remsize = rowsize - (pageuse + 8)
      ```

      If *remsize* is less than *pageuse* `- 4`, you have no full remainder pages.

      If *remsize* is greater than *pageuse* `- 4`, use *remsize* in the following formula to obtain the number of full remainder pages:

      ```
      fullrempages = rows * trunc(remsize/(pageuse - 8))
      ```

   c. Calculate the number of partial remainder pages.

First calculate the size of a partial row remainder left after you have accounted for the home and full remainder pages for an individual row. In the following formula, the **remainder()** function notation indicates that you are to take the remainder after division:

```
partremsize = remainder(rowsize/(pageuse - 8)) + 4
```

The database server uses certain size thresholds with respect to the page size to determine how many partial remainder pages to use. Use the following formula to calculate the ratio of the partial remainder to the page:

```
partratio = partremsize/pageuse
```

Use the appropriate formula in the following table to calculate the number of partial remainder pages.

`partratio` Value	Formula to Calculate the Number of Partial Remainder Pages
Less than .1	partrempages = *rows*/(trunc((*pageuse*/10)/*remsize*) + 1)
Less than .33	partrempages = *rows*/(trunc((*pageuse*/3)/*remsize*) + 1)
.33 or larger	partrempages = *rows*

d. Add up the total number of pages with the following formula:

```
tablesize = homepages + fullrempages + partrempages
```

Estimating tables with variable-length rows

You can estimate the size of a table with variable-length rows with columns of the VARCHAR or NVARCHAR data type.

About this task

When a table contains one or more VARCHAR or NVARCHAR columns, its rows can have varying lengths. These varying lengths introduce uncertainty into the calculations. You must form an estimate of the typical size of each VARCHAR column, based on your understanding of the data, and use that value when you make the estimates.

Important: When the database server allocates space to rows of varying size, it considers a page to be full when no room exists for an additional row of the maximum size.

To estimate the size of a table with variable-length rows, you must make the following estimates and choose a value between them, based on your understanding of the data:

- The maximum size of the table, which you calculate based on the maximum width allowed for all VARCHAR or NVARCHAR columns
- The projected size of the table, which you calculate based on a typical width for each VARCHAR or NVARCHAR column

To estimate the maximum number of data pages:

1. To calculate *rowsize*, add together the maximum values for all column widths.
2. Use this value for *rowsize* and perform the calculations described in Estimating tables with fixed-length rows on page 160. The resulting value is called *maxsize*.

Results

To estimate the projected number of data pages:

1. To calculate *rowsize*, add together typical values for each of your variable-width columns. It is suggested that you use the most frequently occurring width within a column as the typical width for that column. If you do not have access to the data or do not want to tabulate widths, you might choose to use some fractional portion of the maximum width, such as 2/3 (.67).
2. Use this value for *rowsize* and perform the calculations described in Estimating tables with fixed-length rows on page 160. The resulting value is called *projsize*.

Selecting an intermediate value for the size of the table

The actual table size should fall somewhere between the projected number of data pages (*projsize*) and the maximum number of data pages (*maxsize*).

Based on your knowledge of the data, choose a value within that range that seems most reasonable to you. The less familiar you are with the data, the more conservative (higher) your estimate should be.

Estimating pages that simple large objects occupy

You can estimate the total number of pages for all simple large objects, or you can estimate the number of pages based on the median size of the simple large objects.

About this task

The blobpages can reside in either the dbspace where the table resides or in a blobspace. For more information about when to use a blobspace, see Storing simple large objects in the tblspace or a separate blobspace on page 164.

The following methods for estimating blobpages yield a conservative (high) estimate because a single TEXT or BYTE column does not necessarily occupy the entire blobpage within a tblspace. In other words, a blobpage in a tblspace can contain multiple TEXT or BYTE columns.

To estimate the number of blobpages:

1. Obtain the page size with **onstat -b**.
2. Calculate the usable portion of the blobpage with the following formula:

```
bpuse = pagesize – 32
```

3. For each byte of blobsize *n*, calculate the number of pages that the byte occupies (*bpages_n*) with the following formula:

```
bpages1 = ceiling(bytesize1/bpuse)
bpages2 = ceiling(bytesize2/bpuse)
```

```
...
bpages_n = ceiling(bytesize_n/bpuse)
```

The **ceiling()** function indicates that you should round up to the nearest integer value.

4. Add up the total number of pages for all simple large objects, as follows:

```
blobpages = bpages1 + bpages2 + ... + bpagesn
```

Results

Alternatively, you can base your estimate on the median size of simple large objects (TEXT or BYTE data); that is, the simple-large-object data size that occurs most frequently. This method is less precise, but it is easier to calculate.

To estimate the number of blobpages based on the median size of simple large objects:

1. Calculate the number of pages required for simple large objects of median size, as follows:

```
mpages = ceiling(mblobsize/bpuse)
```

2. Multiply this amount by the total number of simple large objects, as follows:

```
blobpages = blobcount * mpages
```

Storing simple large objects in the tblspace or a separate blobspace

When you create a simple-large-object column on magnetic disk, you have the option of storing the column data in the tblspace or in a separate blobspace. You can often improve performance by storing simple-large-object data in a separate blobspace, and by storing smart large objects and user-defined data in sbspaces.

You can also store simple large objects on optical media, but this discussion does not apply to simple large objects stored in this way.

In the following example, a TEXT value is stored in the tblspace, and a BYTE value is stored in a blobspace named **rasters**:

```
CREATE TABLE examptab
   (
   pic_id SERIAL,
   pic_desc TEXT IN TABLE,
   pic_raster BYTE IN rasters
   )
```

For information about storing simple-large-object data in a separate blobspace, see Estimating pages that simple large objects occupy on page 163.

A TEXT or BYTE value is always stored apart from the rows of the table; only a 56-byte descriptor is stored with the row. However, a simple large object occupies at least one disk page. The simple large object to which the descriptor points can reside in the same set of extents on disk as the table rows (in the same tblspace) or in a separate blobspace.

When simple large objects are stored in the tblspace, the pages of their data are interspersed among the pages that contain rows, which can greatly increase the size of the table. When the database server reads only the rows and not the simple large objects, the disk arm must move farther than when the blobpages are stored apart. The database server scans only the row pages in the following situations:

- When it performs any SELECT operation that does not retrieve a simple-large-object column
- When it uses a filter expression to test rows

Another consideration is that disk I/O to and from a dbspace is buffered in shared memory of the database server. Pages are stored in case they are needed again soon, and when pages are written, the requesting program can continue before the actual disk write takes place. However, because blobspace data is expected to be voluminous, disk I/O to and from blobspaces is not buffered, and the requesting program is not allowed to proceed until all output has been written to the blobspace.

For best performance, store a simple-large-object column in a blobspace in either of the following circumstances:

- When single data items are larger than one or two pages each
- When the number of pages of TEXT or BYTE data is more than half the number of pages of row data

Estimating tblspace pages for simple large objects

In your estimate of the space required for a table, include blobpages for any simple large objects that are to be stored in that tblspace. For a table that is both relatively small and nonvolatile, you can achieve the effect of a dedicated blobspace by separating row pages and blobpages.

About this task

To separate row pages from blobpages within a dbspace:

1. Load the entire table with rows in which the simple-large-object columns are null.
2. Create all indexes.
   The row pages and the index pages are now contiguous.
3. Update all the rows to install the simple large objects.
   The blobpages now appear after the pages of row and index data within the tblspace.

Managing the size of first and next extents for the tblspace tblspace

The tblspace **tblspace** is a collection of pages that describe the location and structure of all tblspaces in a dbspace. Each dbspace has one tblspace **tblspace**. When you create a dbspace, you can use the TBLTBLFIRST and TBLTBLNEXT configuration parameters to specify the first and next extent sizes for the tblspace **tblspace** in a root dbspace.

You can use the **onspaces** utility to specify the initial and next extent sizes for the tblspace **tblspace** in non-root dbspaces.

Specify the initial and next extent sizes if you want to reduce the number of tblspace **tblspace** extents and reduce the frequency of situations when you need to place the tblspace **tblspace** extents in non-primary chunks.

The ability to specify a first extent size that is larger than the default provides flexibility for managing space. When you create an extent, you can reserve space during creation of the dbspace, thereby decreasing the risk of needing additional extents created in chunks that are not initial chunks.

You can only specify the first and next extent sizes when you create a dbspace. You cannot alter the specification of the first and next extents sizes after the creation of the dbspace. In addition, you cannot specify extent sizes for temporary dbspaces, sbspaces, blobspaces, or external spaces.

If you do not specify first and next extent sizes for the tblspace **tblspace**, OneDB uses the existing default extent sizes.

Related information

TBLTBLFIRST configuration parameter on page

TBLTBLNEXT configuration parameter on page

Specifying the first and next extent sizes for the tblspace tblspace on page

Managing sbspaces

An *sbspace* is a logical storage unit composed of one or more chunks that store smart large objects. You can estimate the amount of storage needed for smart large objects, improve metadata I/O, monitor sbspaces, and change storage characteristics.

Estimating pages that smart large objects occupy

In your estimate of the space required for a table, you should also consider the amount of sbspace storage for any smart large objects (such as CLOB, BLOB, or multi-representative data types) that are part of the table. An sbspace contains user-data areas and metadata areas.

About this task

CLOB and BLOB data is stored in sbpages that reside in the user-data area. The metadata area contains the smart-large-object attributes, such as average size and whether or not the smart large object is logged. For more information about sbspaces, see your *HCL OneDB™ Administrator's Guide*.

Estimating the size of the sbspace and metadata area

The first chunk of an sbspace must have a metadata area. When you add smart large objects, the database server adds more control information to this metadata area.

If you add a chunk to the sbspace after the initial allocation, you can take one of the following actions for metadata space:

- Allocate another metadata area on the new chunk by default.

  This action provides the following advantages:

  - It is easier because the database server automatically calculates and allocates a new metadata area on the added chunk based on the average smart large object size
  - Distributes I/O operations on the metadata area across multiple disks

- Use the existing metadata area

  If you specify the **onspaces -U** option, the database server does not allocate metadata space in the new chunk. Instead it must use a metadata area in one of the other chunks.

In addition, the database server reserves 40 percent of the user area to be used in case the metadata area runs out of space. Therefore, if the allocated metadata becomes full, the database server starts using this reserved space in the user area for additional control information.

You can let the database server calculate the size of the metadata area for you on the initial chunk and on each added chunks. However, you might want to specify the size of the metadata area explicitly, to ensure that the sbspace does not run out of metadata space and the 40 percent reserve area. You can use one of the following methods to explicitly specify the amount of metadata space to allocate:

- Specify the AVG_LO_SIZE tag on the **onspaces -Df** option.

  The database server uses this value to calculate the size of the metadata area to allocate when the **-Ms** option is not specified. If you do not specify AVG_LO_SIZE, the database server uses the default value of 8 kilobytes to calculate the size of the metadata area.
- Specify the metadata area size in the **-Ms** option of the **onspaces** utility.

  Use the procedure that Sizing the metadata area manually for a new chunk on page 167 describes to estimate a value to specify in the **onspaces -Ms** option.

Sizing the metadata area manually for a new chunk

Each chunk can contain metadata, but the sum total must accommodate enough room for all LO headers (average length 570 bytes each) and the chunk free list (which lists all the free extents in the chunk).

The following procedure assumes that you know the sbspace size and need to allocate more metadata space.

To size the metadata area manually for a new chunk:

1. Use the **onstat -d** option to obtain the size of the current metadata area from the **Metadata size** field.
2. Estimate the number of smart large objects that you expect to reside in the sbspace and their average size.
3. Use the following formula to calculate the total size of the metadata area:

   ```
   Total metadata kilobytes = (LOcount*570)/1024 +
                              (numchunks*800) + 100
   ```

   LOcount

   is the number of smart large objects that you expect to have in all sbspace chunks, including the new one.

   numchunks

   is the total number of chunks in the sbspace.

4. To obtain the additional required area for metadata, subtract the current metadata size that you obtained in step 1 from the value that you obtained in step 3.
5. When you add another chunk, specify in the **-Ms** option of the **onspaces -a** command the value that you obtained in step 4.

Example of calculating the metadata area for a new chunk

This topic contains an example showing how to estimate the metadata size required for two sbspaces chunks.

About this task

Suppose the **Metadata size** field in the **onstat -d** option shows that the current metadata area is 1000 pages. If the system page size is 2048 bytes, the size of this metadata area is 2000 kilobytes, as the following calculation shows:

```
current metadata = (metadata_size * pagesize) / 1024
                 = (1000 * 2048) / 1024
               = 2000 kilobytes
```

Suppose you expect 31,000 smart large objects in the two sbspace chunks. The following formula calculates the total size of metadata area required for both chunks, rounding up fractions:

```
Total metadata = (LOcount*570)/1024 + (numchunks*800) + 100
               = (31,000 * 570)/1024 + (2*800) + 100
               = 17256 + 1600 + 100
             = 18956 kilobytes
```

To obtain the additional area that is required for metadata:

1. Subtract the current metadata size from the total metadata value.

   ```
   Additional metadata = Total metadata - current metadata
                       = 18956 - 2000
                     = 16956 kilobytes
   ```

2. When you add the chunk to the sbspace, use the **-Ms** option of the **onspaces -a** command to specify a metadata area of 16,956 kilobytes.

   ```
   % onspaces -a sbchk2 -p /dev/raw_dev1 -o 200 -Ms 16956
   ```

Improving metadata I/O for smart large objects

The metadata pages in an sbspace contain information about the location of the smart large objects in the sbspace. Typically, these pages are read intensive. You can improve metadata I/O by redistributing it.

You can distribute I/O to these pages in one of the following ways:

- Mirror the chunks that contain metadata.

  For more information about the implications of mirroring, see Consider mirroring for critical data components on page 111.
- Position the metadata pages on the fastest portion of the disk.

Because the metadata pages are the most read-intensive part of an sbspace, place the metadata pages toward the middle of the disk to minimize disk seek time. To position metadata pages, use the **-Mo** option when you create the sbspace or add a chunk with the **onspaces** utility.

- Spread metadata pages across disks.

  To spread metadata pages across disks, create multiple chunks in an sbspace, with each chunk residing on a separate disk. When you add a chunk to the sbspace with the **onspaces** utility, specify the **-Ms** option to allocate pages for the metadata information.

  Although the database server attempts to keep the metadata information with its corresponding data in the same chunk, it cannot guarantee that they will be together.

- Decrease the number of extents each smart large object occupies.

  When a smart large object spans multiple extents, the metadata area contains a separate descriptor for each extent. To decrease the number of descriptor entries that must be read for each smart large object, specify the expected final size of the smart large object when you create the smart large object.

  The database server allocates the smart large object as a single extent (if it has contiguous storage in the chunk) when you specify the final size in either of the following functions:

  - The DataBlade® API **mi_lo_specset_estbytes** function
  - The **ifx_lo_specset_estbytes** function

  For more information about the functions to open a smart large object and to set the estimated number of bytes, see the *HCL OneDB™ ESQL/C Programmer's Manual* and *HCL OneDB™ DataBlade® API Programmer's Guide*.

  For more information about sizing extents, see Sbspace extents on page 128.

Important: For highest data availability, mirror all sbspace chunks that contain metadata.

Monitoring sbspaces

You can monitor the effectiveness of I/O operations on smart large objects. For better I/O performance, all smart large objects should be allocated in one extent to be contiguous.

For more information about sizing extents, see Sbspace extents on page 128.

Contiguity provides the following I/O performance benefits:

- Minimizes the disk-arm motion
- Requires fewer I/O operations to read the smart large object
- When doing large sequential reads, can take advantage of lightweight I/O, which reads in larger blocks of data (60 kilobytes or more, depending on your platform) in a single I/O operation

You can use the following command-line utilities to monitor the effectiveness of I/O operations on smart large objects:

- **oncheck -cS, -pe** and **-pS**
- **onstat -g smb s** option

The following sections describe how to use these utility options to monitor sbspaces.

Monitoring sbspaces with oncheck -cS

The **oncheck -cS** option checks smart-large-object extents and the sbspace partitions in the user-data area.

Figure 20: oncheck -cS output on page 170 shows an example of the output from the **-cS** option for **s9_sbspc**.

The values in the **Sbs#**, **Chk#**, and **Seq#** columns correspond to the **Space Chunk Page** value in the **-pS** output. The **Bytes** and **Pages** columns display the size of each smart large object in bytes and pages.

To calculate the average size of smart large objects, you can total the numbers in the **Size (Bytes)** column and then divide by the number of smart large objects. In Figure 20: oncheck -cS output on page 170, the average number of bytes allocated is 2690, as the following calculation shows:

```
Average size in bytes = (15736 + 98 + 97 + 62 + 87 + 56) / 6
                      = 16136 / 6
                      = 2689.3
```

For information about how to specify smart large object sizes to influence extent sizes, see Sbspace extents on page 128.

Figure 20. oncheck -cS output

```
Validating space 's9_sbspc' ...

Large Objects
  ID               Ref Size       Allocced       Creat   Last
  Sbs# Chk#  Seq#  Cnt    (Bytes)    Pages Extns Flags   Modified
  ---- ---- ----- ---- ---------- -------- ----- ----- ------------------------
     2    2     1    1      15736        8     1 N-N-H Thu Jun 21 16:59:12 2007
     2    2     2    1         98        1     1 N-K-H Thu Jun 21 16:59:12 2007
     2    2     3    1         97        1     1 N-K-H Thu Jun 21 16:59:12 2007
     2    2     4    1         62        1     1 N-K-H Thu Jun 21 16:59:12 2007
     2    2     5    1         87        1     1 N-K-H Thu Jun 21 16:59:12 2007
     2    2     6    1         56        1     1 N-K-H Thu Jun 21 16:59:12 2007
```

The **Extns** field shows the minimum extent size, in number of pages, allocated to each smart large object.

Monitoring sbspaces with oncheck -pe

The **oncheck -pe** option displays information that includes the size in pages of the chunk, the number of pages used, the number of pages that are free, and a list of all the tables in the chunk, with the initial page number and the length of the table in pages. This option also shows if smart large objects occupy contiguous space within an sbspace.

Execute **oncheck -pe** to display the following information to determine if the smart large objects occupy contiguous space within an sbspace:

- Identifies each smart large object with the term `SBLOBSpace LO`

  The three values in brackets following `SBLOBSpace LO` correspond to the **Sbs#**, **Chk#**, and **Seq#** columns in the **-cS** output.
- Offset of each smart large object
- Number of disk pages (*not* sbpages) used by each smart large object

Tip: The **oncheck -pe** option provides information about sbspace use in terms of database server pages, not sbpages.

Figure 21: oncheck -pe output that shows contiguous space use on page 171 shows sample output. In this example, the **size** field shows that the first smart large object occupies eight pages. Because the **offset** field shows that the first smart large object starts at page 53 and the second smart large object starts at page 61, the first smart large object occupies contiguous pages.

Figure 21. oncheck -pe output that shows contiguous space use

```
Chunk Pathname                                    Size      Used      Free
                                                  1000       940        60

    Description                                        Offset     Size
    -------------------------------------------------- -------- --------
    RESERVED PAGES                                          0        2
    CHUNK FREELIST PAGE                                     2        1
    s9_sbspc:'informix'.TBLSpace                            3       50
    SBLOBSpace LO [2,2,1]                                  53        8
    SBLOBSpace LO [2,2,2]                                  61        1
    SBLOBSpace LO [2,2,3]                                  62        1
    SBLOBSpace LO [2,2,4]                                  63        1
    SBLOBSpace LO [2,2,5]                                  64        1
    SBLOBSpace LO [2,2,6]                                  65        1
...
```

Monitoring sbspaces with oncheck -pS

The **oncheck -pS** option displays information about smart-large-object extents and metadata areas in sbspace partitions. If you do not specify an sbspace name on the command line, **oncheck** checks and displays the metadata for all sbspaces.

Figure 22: oncheck -pS output on page 172 shows an example of the **-pS** output for **s9_sbspc**.

To display information about smart large objects, execute the following command:

```
oncheck -pS spacename
```

The **oncheck -pS** output displays the following information for each smart large object in the sbspace:

- Space chunk page
- Size in bytes of each smart large object
- Object ID that DataBlade® API and functions use
- Storage characteristics of each smart large object

When you use **onspaces -c -S** to create an sbspace, you can use the **-Df** option to specify various storage characteristics for the smart large objects. You can use **onspaces -ch** to change attributes after the sbspace is created. The **Create Flags** field in the **oncheck -pS** output displays these storage characteristics and other attributes of each smart large object. In Figure 22: oncheck -pS output on page 172, the **Create Flags** field shows LO_LOG because the LOGGING tag was set to ON in the **-Df** option.

Figure 22. oncheck -pS output

```
Space Chunk Page = [2,2,2] Object ID = 987122917
  LO SW Version             4
  LO Object Version         1
  Created by Txid           7
  Flags               0x31 LO_LOG LO_NOKEEP_LASTACCESS_TIME LO_HIGH_INTEG
  Data Type                 0
  Extent Size               -1
  IO Size                   0
  Created                   Thu Apr 12 17:48:35 2007
  Last Time Modified        Thu Apr 12 17:48:43 2007
  Last Time Accessed        Thu Apr 12 17:48:43 2007
  Last Time Attributes Modified Thu Apr 12 17:48:43 2007
  Ref Count                 1
  Create Flags        0x31 LO_LOG LO_NOKEEP_LASTACCESS_TIME LO_HIGH_INTEG
  Status Flags              0x0 LO_FROM_SERVER
  Size (Bytes)              2048
  Size Limit                -1
  Total Estimated Size      -1
  Deleting TxId             -1
  LO Map Size               200
  LO Map Last Row           -1
  LO Map Extents            2
  LO Map User Pages         2
```

Monitoring sbspaces with onstat -g smb

The **onstat -g smb s** option displays sbspace attributes.

Use the **onstat -g smb s** option to display the following characteristics that affect the I/O performance of each sbspace:

- Logging status

  If applications are updating temporary smart large objects, logging is not required. You can turn off logging to reduce the amount of I/O activity to the logical log, CPU utilization, and memory resources.

- Average smart-large-object size

  Average size and extent size should be similar to reduce the number of I/O operations required to read in an entire smart large object. The **avg s/kb** output field shows the average smart-large-object size in kilobytes. In Figure 23: onstat -g smb s output on page 173, the **avg s/kb** output field shows the value `30` kilobytes.

  Specify the final size of the smart large object in either of the following functions to allocate the object as a single extent:

- The DataBlade® API **mi_lo_specset_estbytes** function
- The **ifx_lo_specset_estbytes** function

For more information about the functions to open a smart large object and to set the estimated number of bytes, see the *HCL OneDB™ ESQL/C Programmer's Manual* and *HCL OneDB™ DataBlade® API Programmer's Guide*.

- First extent size, next extent size, and minimum extent size

  The **1st sz/p**, **nxt sz/p**, and **min sz/p** output fields show these extent sizes if you set the extent tags in the **-Df** option of **onspaces**. In Figure 23: onstat -g smb s output on page 173, these output fields show values of `0` and `-1` because these tags are not set in **onspaces**.

Figure 23. onstat -g smb s output

```
sbnum 7    address 2afae48
  Space    : flags    nchk owner    sbname
             -------- 1    informix client
  Defaults : LO_LOG LO_KEEP_LASTACCESS_TIME

  LO     : ud b/pg flags   flags    avg s/kb max lcks
           2048    0       -------- 30       -1
  Ext/IO : 1st sz/p nxt sz/p min sz/p mx io sz
           4        0        0        -1

  HdrCache : max   free
             512   0
```

Changing storage characteristics of smart large objects

When you create an sbspace, but do not specify values in the **-Df** option of the **onspaces -c -S** command, you use the defaults for the storage characteristics and attributes (such as logging and buffering). After you monitor sbspaces, you might want to change the storage characteristics, logging status, lock mode, or other attributes for new smart large objects.

The database administrator or programmer can use the following methods to override these default values for storage characteristics and attributes:

- The database administrator can use one of the following **onspaces** options:
  - Specify values when the sbspace is first created with the **onspaces -c -S** command.
  - Change values after the sbspace is created with the **onspaces -ch** command.

  Specify these values in the tag options of the **-Df** option of **onspaces**. For more information about the **onspaces** utility, see the *HCL OneDB™ Administrator's Reference*.
- The database administrator can specify values in the PUT clause of the CREATE TABLE or ALTER TABLE statements.

  These values override the values in the **onspaces** utility and are valid only for smart large objects that are stored in the associated column of the specific table. Other smart large objects (from columns in other tables) might also reside in this same sbspace. These other columns continue to use the storage characteristics and attributes of the

sbspace that **onspaces** defined (or the default values, if **onspaces** did not define them) unless these columns also used a PUT clause to override them for a particular column.

If you do not specify the storage characteristics for a smart-large-object column in the PUT clause, they are inherited from the sbspace.

If you do not specify the PUT clause when you create a table with smart-large-object columns, the database server stores the smart large objects in the system default sbspace, which is specified by the SBSPACENAME configuration parameter in the ONCONFIG file. In this case, the storage characteristics and attributes are inherited from the SBSPACENAME sbspace.

- Programmers can use functions in the DataBlade® API and to alter storage characteristics for a smart-large-object column.

  For information about the DataBlade® API functions for smart large objects, see the *HCL OneDB™ DataBlade® API Programmer's Guide*. For information about the functions for smart large objects, see the *HCL OneDB™ ESQL/C Programmer's Manual*.

Table 11: Altering storage characteristics and other attributes of an sbspace on page 174 summarizes the ways to alter the storage characteristics for a smart large object.

Table 11. Altering storage characteristics and other attributes of an sbspace

Storage Character-istic or Attribute	System Default Value	System-Specified Storage Characteristics Specified by -Df Option in onspaces Utility	Column-Level Storage Characteristics Specified by PUT clause of CREATE TABLE or ALTER TABLE	Storage Characteris-tics Specified by a DataBlade® API Function	Storage Characteris-tics Specified by an ESQL/C Function
Last-access time	OFF	ACCESSTIME	KEEP ACCESS TIME, NO KEEP ACCESS TIME	Yes	Yes
Lock mode	BLOB	LOCK_MODE	No	Yes	Yes
Logging status	OFF	LOGGING	LOG, NO LOG	Yes	Yes
Data integrity	HIGH INTEG	No	HIGH INTEG, MODERATE INTEG	Yes	No
Size of extent	None	EXTENT_SIZE	EXTENT SIZE	Yes	Yes
Size of next extent	None	NEXT_SIZE	No	No	No
Minimum extent size	2 kilobytes on Windows™ 4 kilobytes on UNIX™	MIN_EXT_SIZE	No	No	No

Table 11. Altering storage characteristics and other attributes of an sbspace (continued)

Storage Character-istic or Attribute	System Default Value	System-Specified Storage Characteristics Specified by -Df Option in onspaces Utility	Column-Level Storage Characteristics Specified by PUT clause of CREATE TABLE or ALTER TABLE	Storage Characteris-tics Specified by a DataBlade® API Function	Storage Characteris-tics Specified by an ESQL/C Function
Size of smart large object	8 kilobytes	Average size of all smart large objects in sbspace: AVG_LO_SIZE	No	Estimated size of a particular smart large object Maximum size of a particular smart large object	Estimated size of a particular smart large object Maximum size of a particular smart large object
Buffer pool usage	ON	BUFFERING	No	LO_BUFFER and LO_ NOBUFFER flags	LO_BUFFER and LO_ NOBUFFER flags
Name of sbspace	SBSPACE-NAME	Not in **-Df** option. Name specified in **onspaces -S** *option.*	Name of an existing sbspace in which a smart large object resides: PUT ... IN clause	Yes	Yes
Fragmenta-tion across multiple sbspaces	None	No	Round-robin distribution scheme: PUT ... IN clause	Round-robin or expression-based distribution scheme	Round-robin or expression-based distribution scheme
Last-access time	OFF	ACCESSTIME	KEEP ACCESS TIME, NO KEEP ACCESS TIME	Yes	Yes

Altering smart-large-object columns

When you create or modify a table, you have several options for choosing storage characteristics and other attributes (such as logging status, buffering, data integrity, and locking granularity) for specific smart-large-object columns.

When you create or modify a table that can store BLOB or CLOB objects, you have these options:

- Use the values that were set when the sbspace was created. These values are specified in one of the following ways:
  - With the various flags of the **-Df** option of the **onspaces -c -S** command
  - With the system default value for any flag that was not specified.

  For guidelines to change the default storage characteristics of the **-Df** flags, see onspaces options that affect sbspace I/O on page 128.

- Use the PUT clause of the CREATE TABLE statement to specify non-default values for particular characteristics or attributes, including the number of sbspaces, the extent size, the logging, buffering, and data integrity status, and the locking granularity.

  Characteristics or attributes that you do not specify in the PUT clause default to the values set in the **onspaces -c -S** command, or to system default values (for example, no logging).

Later, you can use the PUT clause of the ALTER TABLE statement to change the optional storage characteristics of BLOB or CLOB columns. See Table 11: Altering storage characteristics and other attributes of an sbspace on page 174 for characteristics and attributes of sbspaces that you can change.

You can use the PUT clause of the ALTER TABLE statement to perform the following actions:

- Specify the smart-large-object characteristics and storage location when you add a new BLOB or CLOB column to a table.

  The smart large objects in the new columns can have characteristics different from those in the existing columns.
- Change the smart-large-object characteristics of an existing column.

  The new column characteristics apply only to smart large objects in new rows inserted after the ALTER TABLE PUT statement was issued. The old characteristics persist for any smart large objects that already existed in the column before the ALTER TABLE PUT statement modified the column.

For example, the BLOB data in the **catalog** table in the **superstores_demo** database is stored in **s9_sbspc** with logging turned off and has an extent size of 100 kilobytes. You can use the PUT clause of the ALTER TABLE statement to turn on logging and store new smart large objects in a different sbspace.

For information about changing sbspace extents with the CREATE TABLE statement, see Extent sizes for smart large objects in sbspaces on page 179.

Related information

Sbspace logging on page

CREATE TABLE statement on page

Managing extents

As you add rows to a table, the database server allocates disk space in units called *extents*. Each extent is a block of physically contiguous pages from the dbspace. Even when the dbspace includes more than one chunk, each extent is allocated entirely within a single chunk, so that it remains contiguous.

Contiguity is important to performance. When the pages of data are contiguous, and when the database server reads the rows sequentially during read-ahead, light scans, or lightweight I/O operations, disk-arm motion is minimized. For more information about these operations, see Sequential scans on page 133, Light scans on page 133, and Configuration parameters that affect sbspace I/O on page 127.

The mechanism of extents is a compromise between the following competing requirements:

- Most dbspaces are shared among several tables.
- The size of some tables is not known in advance.
- Tables can grow at different times and different rates.
- All the pages of a table should be adjacent for best performance.

If you have a table that needs more extents and the database server runs out of space on the partition header page, the database server automatically allocates extended secondary partition header pages to accommodate new extent entries. The database server can allocate up to 32767 extents for any partition, unless the size of a table dictates a limit to the number of extents.

Because table sizes are not known, the database server cannot preallocate table space. Therefore, the database server adds extents only as they are needed, but all the pages in any one extent are contiguous for better performance. In addition, when the database server creates an extent that is next to the previous one, it treats both as a single extent.

A frequently updated table can become fragmented over time which degrades the performance every time the table is accessed by the server. Defragmenting a table brings data rows closer together and avoids partition header page overflow problems.

Choosing table extent sizes

When you create a table, you can specify extent sizes for the data rows of a table in a dbspace and for each fragment of a fragmented table, and the smart large objects in an sbspace. The database server calculates extent sizes for smart large objects in sbspaces.

Extent sizes for tables in a dbspace

When you create a table, you can specify the size of the first extent, as well as the size of the extents to be added as the table grows. You can also modify the size of the first extent in a table in a dbspace, and you can modify the size of new subsequent extents.

The following sample SQL statement creates a table with a 512-kilobyte initial extent and 100-kilobyte added extents:

```
CREATE TABLE big_one (column specifications)
   IN big_space
   EXTENT SIZE 512
   NEXT SIZE 100
```

The default value for the extent size and the next-extent size is eight times the disk page size on your system. For example, if you have a 2-kilobyte page, the default length is 16 kilobytes.

You can use the ALTER TABLE statement with the MODIFY EXTENT SIZE clause to change the size of the first extent of a table in a dbspace. When you change the size of the first extent, OneDB records the change in the system catalog and on the partition page, but only makes the actual change when the table is rebuilt or a new partition or fragment is created.

You might want to change the size of the first extent of a table in a dbspace in either of these situations:

- If a table was created with small first extent size and you need to keep adding a lot of next extents, the table becomes fragmented across multiple extents and the data is scattered.
- If a table was created with a first extent that is much larger than the amount of data that is stored, space is wasted.

The following example changes the size of the first extent of a table in a dbspace to 50 kilobytes:

```
ALTER TABLE customer MODIFY EXTENT SIZE 50;
```

Changes to the first extent size are recorded into the system catalog table and on the partition page on the disk. However, changes to the first extent size do not take effect immediately. Instead, whenever a change that rebuilds the table occurs, the server uses the new first extent size.

For example, if a table has a first extent size of 8 kilobytes and you use the ALTER TABLE statement to change this to 16 kilobytes, the server does not drop the current first extent and recreate it with the new size. Instead, the new first extent size of 16 kilobytes takes effect only when the server rebuilds the table after actions such as creating a cluster index on the table or detaching a fragment from the table.

If a TRUNCATE TABLE statement without the REUSE option is executed before the ALTER TABLE statement with the MODIFY EXTENT SIZE clause, there is no change in the current first extent.

Use the MODIFY NEXT SIZE clause to change the size of the next extent to be added. This change does not affect next extents that already exist.

The following example changes the size of the next extent of a table to 50 kilobytes:

```
ALTER TABLE big_one MODIFY NEXT SIZE 50;
```

The next extent sizes of the following kinds of tables do not affect performance significantly:

- A small table is defined as a table that has only one extent. If such a table is heavily used, large parts of it remain buffered in memory.
- An infrequently used table is not important to performance no matter what size it is.
- A table that resides in a dedicated dbspace always receives new extents that are adjacent to its old extents. The size of these extents is not important because, being adjacent, they perform as one large extent.

Avoid creating large numbers of extents

When you assign an extent size to these kinds of tables, the only consideration is to avoid creating large numbers of extents. A large number of extents causes the database server to spend extra time finding the data. In addition, an upper limit exists on the number of extents allowed. (Considering the upper limit on extents on page 181 covers this topic.)

Tips for allocating space for table extents

No upper limit exists on extent sizes except the size of the chunk. The maximum size for a chunk is 4 terabytes. When you know the final size of a table (or can confidently predict it within 25 percent), allocate all its space in the initial extent. When tables grow steadily to unknown size, assign them next-extent sizes that let them share the dbspace with a small number of extents each.

Allocating space for table extents

To allocate space for table extents:

1. Decide how to allocate space among the tables.

   For example, you might divide the dbspace among three tables in the ratio 0.4: 0.2: 0.3 (reserving 10 percent for small tables and overhead).
2. Give each table one-fourth of its share of the dbspace as its initial extent.
3. Assign each table one-eighth of its share as its next-extent size.
4. Monitor the growth of the tables regularly with **oncheck**.

As the dbspace fills up, you might not have enough contiguous space to create an extent of the specified size. In this case, the database server allocates the largest contiguous extent that it can.

Related information

TBLTBLFIRST configuration parameter on page

TBLTBLNEXT configuration parameter on page

MODIFY EXTENT SIZE on page

Extent sizes for table fragments

When you fragment an existing table, you might want to adjust the next-extent size because each fragment requires less space than the original, unfragmented table.

If the unfragmented table was defined with a large next-extent size, the database server uses that same size for the next-extent on *each* fragment, which results in over-allocation of disk space. Each fragment requires only a proportion of the space for the entire table.

For example, if you fragment the preceding **big_one** sample table across five disks, you can alter the next-extent size to one-fifth the original size. The following example changes the next-extent size to one-fifth of the original size:

```
ALTER TABLE big_one MODIFY NEXT SIZE 2;
```

Related information

MODIFY NEXT SIZE clause on page

Extent sizes for smart large objects in sbspaces

When you create a table, you should use the extent size that the database server calculates for smart large objects in sbspaces. Alternatively, you can use the final size of the smart large object, as indicated by a particular function when you open the sbspace in an application program.

You can use the final size of the smart large object when you open one of the following application programs:

- *For DB-Access:* Use the DataBlade® API **mi_lo_specset_estbytes** function. For more information about the DataBlade® API functions to open a smart large object and set the estimated number of bytes, see the *HCL OneDB™ DataBlade® API Programmer's Guide*.
- *For ESQL/C:* Use the **ifx_lo_specset_estbytes** function. For more information about the functions to open a smart large object and set the estimated number of bytes, see the *HCL OneDB™ ESQL/C Programmer's Manual*.

For more information about sizing extents, see Sbspace extents on page 128. For more information, see Monitoring sbspaces on page 169.

Monitoring active tblspaces

Monitor tblspaces to determine which tables are active. Active tables are those that a thread has currently opened.

Output from the **onstat -t** option includes the tblspace number and the following four fields.

Field

Description

npages

Pages allocated to the tblspace

nused

Pages used from this allocated pool

nextns

Number of extents used

npdata

Number of data pages used

If a specific operation needs more pages than are available (**npages** minus **nused**), a new extent is required. If enough space is available in this chunk, the database server allocates the extent here; if not, the database server looks for space in other available chunks. If none of the chunks contains adequate contiguous space, the database server uses the largest block of contiguous space that it can find in the dbspace. Figure 24: onstat -t output on page 180 shows an example of the output from this option.

Figure 24. onstat -t output

```
Tblspaces
 n address  flgs ucnt tblnum   physaddr npages nused  npdata nrows  nextns
 0 422528   1    1    100001   10000e   150    124    0      0      3
 1 422640   1    1    200001   200004   50     36     0      0      1
54 426038   1    6    100035   1008ac   3650   3631   3158   60000  3
62 4268f8   1    6    100034   1008ab   8      6      4      60     1
63 426a10   3    6    100036   1008ad   368    365    19     612    3
64 426b28   1    6    100033   1008aa   8      3      1      6      1
193 42f840  1    6    10001b   100028   8      5      2      30     1
 7 active, 200 total, 64 hash buckets
```

Monitoring the upper limit on extents and extent interleaving

You can monitor the upper limit on the number of extents. You can also check for and eliminate extent interleaving.

The maximum number of extents for a partition is 32767.

Considering the upper limit on extents

Do not allow a table to acquire a large number of extents because an upper limit exists on the number of extents allowed. Trying to add an extent after you reach the limit causes error -136 (`No more extents`) to follow an INSERT request.

About this task

Results

To help ensure that the limit is not exceeded, the database server performs the following actions:

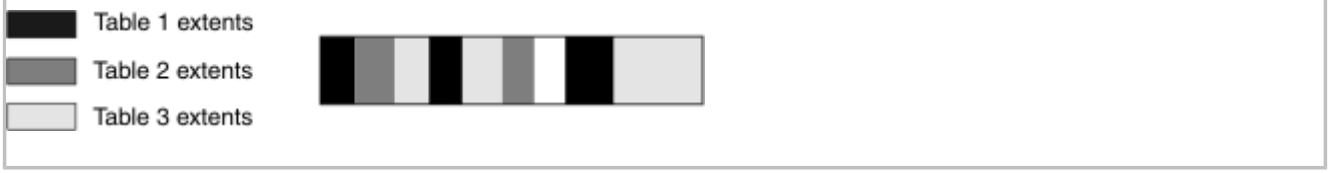
- The database server checks the number of extents each time that it creates an extent. If the number of the extent being created is a multiple of 16, the database server automatically doubles the next-extent size for the table. Therefore, at every 16th creation, the database server doubles the next-extent size.
- When the database server creates an extent next to the previous extent, it treats both extents as a single extent.

Checking for extent interleaving

When two or more growing tables share a dbspace, extents from one tblspace can be placed between extents from another tblspace. When this situation occurs, the extents are said to be *interleaved*. Performance suffers when disk seeks for a table must span more than one extent, particularly for sequential scans.

Interleaving creates gaps between the extents of a table. Figure 25: Interleaved table extents on page 181 shows gaps between table extents.

Figure 25. Interleaved table extents



Try to optimize the table-extent sizes to allocate contiguous disk space, which limits head movement. Also consider placing the tables in separate dbspaces.

Check periodically for extent interleaving by monitoring chunks. Execute **oncheck -pe** to obtain the physical layout of information in the chunk. The following information appears:

- Dbspace name and owner
- Number of chunks in the dbspace
- Sequential layout of tables and free space in each chunk
- Number of pages dedicated to each table extent or free space

This output is useful for determining the degree of extent interleaving. If the database server cannot allocate an extent in a chunk despite an adequate number of free pages, the chunk might be badly interleaved.

Eliminating interleaved extents

You can eliminate interleaved extents by reorganizing the tables with the UNLOAD and LOAD statements, creating or altering an index to cluster, or using the ALTER TABLE statement.

Creating or altering an index to cluster

Depending on the circumstances, you can eliminate extent interleaving if you create a clustered index or alter a clustered index. When you use the TO CLUSTER clause of the CREATE INDEX or ALTER INDEX statement, the database server sorts and reconstructs the table.

About this task

The TO CLUSTER clause reorders rows in the physical table to match the order in the index. For more information, see Clustering on page 217.

The TO CLUSTER clause eliminates interleaved extents under the following conditions:

- The chunk must contain contiguous space that is large enough to rebuild each table.
- The database server must use this contiguous space to rebuild the table.

  If blocks of free space exist before this larger contiguous space, the database server might allocate the smaller blocks first. The database server allocates space for the ALTER INDEX process from the beginning of the chunk, looking for blocks of free space that are greater than or equal to the size that is specified for the next extent. When the database server rebuilds the table with the smaller blocks of free space that are scattered throughout the chunk, it does not eliminate extent interleaving.

To display the location and size of the blocks of free space, execute the **oncheck -pe** command.

To use the TO CLUSTER clause of the ALTER INDEX statement:

1. For each table in the chunk, drop all fragmented or detached indexes except the one that you want to cluster.
2. Cluster the remaining index with the TO CLUSTER clause of the ALTER INDEX statement.
   This step eliminates interleaving the extents when you rebuild the table by rearranging the rows.
3. Re-create all the other indexes.

Results

You do not need to drop an index before you cluster it. However, the ALTER INDEX process is faster than CREATE INDEX because the database server reads the data rows in cluster order using the index. In addition, the resulting indexes are more compact.

To prevent the problem from recurring, consider increasing the size of the tblspace extents.

Using ALTER TABLE to eliminate extent interleaving

If you use the ALTER TABLE statement to add or drop a column or to change the data type of a column, the database server copies and reconstructs the table. When the database server reconstructs the entire table, it rewrites the table to other areas of the dbspace. However, if other tables are in the dbspace, no guarantee exists that the new extents will be adjacent to each other.

Important: For certain types of operations that you specify in the ADD, DROP, and MODIFY clauses, the database server does not copy and reconstruct the table during the ALTER TABLE operation. In these cases, the database server uses an in-place alter algorithm to modify each row when it is updated (rather than during the ALTER TABLE operation). For more information about the conditions for this in-place alter algorithm, see In-place alter on page 193.

Reclaiming unused space within an extent

After the database server allocates disk space to a tblspace as part of an extent, that space remains dedicated to the tblspace. Even if all extent pages become empty after you delete data, the disk space remains unavailable for use by other tables unless you reclaim the space.

Important: When you delete rows in a table, the database server reuses that space to insert new rows into the same table. This section describes the procedures for reclaiming unused space for use by other tables.

You might want to resize a table that does not require the entire amount of space that was originally allocated to it. You can reallocate a smaller dbspace and release the unneeded space for other tables to use.

As the database server administrator, you can reclaim the disk space in empty extents and make it available to other users by rebuilding the table. To rebuild the table, use any of the following SQL statements:

- ALTER INDEX
- UNLOAD and LOAD
- ALTER FRAGMENT

Reclaiming space in an empty extent with ALTER INDEX

If the table with the empty extents includes an index, you can run the ALTER INDEX statement with the TO CLUSTER clause. Clustering an index rebuilds the table in a different location within the dbspace.

When you run the ALTER INDEX statement with the TO CLUSTER clause, all of the extents associated with the previous version of the table are released. Also, the newly built version of the table has no empty extents.

Related information

ALTER INDEX statement on page

Clustering on page 217

Releasing space in an empty extent with ALTER FRAGMENT

You can use the ALTER FRAGMENT statement to rebuild a table. When you run this statement, it releases space within the extents that were allocated to that table.

For more information about the syntax of the ALTER FRAGMENT statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

Managing extent deallocation with the TRUNCATE keyword

TRUNCATE is an SQL keyword that quickly deletes active rows from a table and the b-tree structures of its indexes, without dropping the table or its schema, access privileges, triggers, constraints, and other attributes. With this SQL data-definition language statement, you can depopulate a local table and reuse the table without re-creating it, or you can release the storage space that formerly held its data rows and b-tree structures.

Two implementations of TRUNCATE exist:

- The first implementation, called "fast truncate," operates on most tables.
- The second implementation, called "slow truncate," operates on tables that include opaque or smart large object data types, or inherited indexes that are defined on ROW types within data type hierarchies.

The performance advantages of using the TRUNCATE TABLE statement instead of the DELETE statement are much better for the fast truncate implementation, because this implementation does not examine or run all of the rows in a table. Slow truncation implementation occurs on tables that include opaque or smart large object data types or inherited indexes that are defined on ROW types within data types, because the truncate operation examines each row containing these items.

For more information about using TRUNCATE, see the *HCL OneDB™ Guide to SQL: Syntax*.

Defragment partitions to merge extents

You can improve performance by defragmenting partitions to merge non-contiguous extents.

A frequently updated table can become fragmented over time which degrades the performance every time the table is accessed by the server. Defragmenting a table brings data rows closer together and avoids partition header page overflow problems.

Defragmenting an index brings the entries closer together which improves the speed at which the table information is accessed.

You cannot stop a defragment request after the request has been submitted. Additionally, there are specific objects that cannot be defragmented and you cannot defragment a partition if another operation is running that conflicts with the defragment request.

Tip: Before you defragment a partition:

- Review the information about important limitations and considerations in Partition defragmentation on page .
- Run the oncheck -pt and pT command to determine the number of extents for a specific table or fragment.

To defragment a table, index, or partition, run the EXECUTE FUNCTION command with the defragment argument. You can specify the table name, index name, or partition number that you want to defragment.

You can use the onstat -g defragment command to display information about the active defragment requests.

Related information

Scheduling data optimization on page

onstat -g defragment command: Print defragment partition extents on page

oncheck -pt and -pT: Display tblspaces for a Table or Fragment on page

defragment argument: Dynamically defragment partition extents (SQL administration API) on page

Storing multiple table fragments in a single dbspace

You can store multiple fragments of the same table or index in a single dbspace, thus reducing the total number of dbspaces needed for a fragmented table. You must specify a name for each fragment that you want to store in the same dbspace. Storing multiple table or index fragments in a single dbspace simplifies the management of dbspaces.

You can also use this feature to improve query performance over storing each fragment in a different dbspace when a dbspace is located on a faster device.

For more information, see information about managing partitions in the *HCL OneDB™ Administrator's Guide*.

Displaying a list of table and index partitions

Use the **onstat -g opn** option to display a list of the table and index partitions, by thread ID, that are currently open in the system.

For an example of **onstat -g opn** output and an explanation of output fields, see the *HCL OneDB™ Administrator's Reference*.

Changing tables to improve performance

You can change tables to improve performance by dropping indexes, attaching or detaching fragments, and altering table definitions. You can also create databases for decision-support applications by unloading and loading tables in OLTP databases.

You might want to change an existing table for various reasons:

- To refresh large decision-support tables with data periodically
- To add or drop historical data from a certain time period
- To add, drop, or modify columns in large decision-support tables when the need arises for different data analysis

Loading and unloading tables

You can create databases for decision-support applications by periodically loading tables that have been unloaded from active OLTP databases.

You can use one or more of the following methods to load large tables quickly:

- External tables
- Nonlogging tables

  The database server provides support to:

  - Create nonlogging or logging tables in a logging database.
  - Alter a table from nonlogging to logging and vice versa.

  The two table types are STANDARD (logging tables) and RAW (nonlogging tables). You can use any loading utility such as **dbimport** or HPL to load raw tables.

  The two table types are STANDARD (logging tables) and RAW (nonlogging tables). You can use any loading utility, such as **dbimport**, to load raw tables.
- High-Performance Loader (HPL)

  You can use HPL in express mode to load tables quickly.

The following sections describe:

- Advantages of logging and nonlogging tables
- Step-by-step procedures to load data using nonlogging tables

---

Related information

Moving data with external tables on page

CREATE EXTERNAL TABLE Statement on page

Advantages of logging tables

Logging type options specify the logging characteristics that can improve performance in various bulk operations on the table.

STANDARD, which corresponds to a table in a logged database of previous versions, is the default logging type that is used when you issue the CREATE TABLE statement without specifying the table type.

Standard tables have the following features:

- Logging to allow rollback, recovery, and restoration from archives.
- Recovery from backups
- All insert, delete, and update operations
- Constraints to maintain the integrity of your data
- Indexes to quickly retrieve a small number of rows

OLTP applications usually use standard tables. OLTP applications typically have the following characteristics:

- Real-time insert, update, and delete transactions

  Logging and recovery of these transactions is critical to preserve the data. Locking is critical to allow concurrent access and to ensure the consistency of the data selected.

- Update, insert, or delete one row or a few rows at a time

  Indexes speed access to these rows. An index requires only a few I/O operations to access the pertinent row, but scanning a table to find the pertinent row might require many I/O operations.

Advantages of nonlogging tables

Nonlogging tables, which are also called raw tables, have characteristics that enable you to load very large data warehousing tables quickly.

About this task

Raw tables have following characteristics:

- They do not use CPU and I/O resources for logging.
- They avoid problems such as running out of logical-log space.
- They are locked exclusively during an express load so that no other user can access the table during the load.
- They do not support referential constraints and unique constraints, so overhead for constraint-checking is eliminated.

Quickly loading a large standard table

You can change a large, existing standard table into a nonlogging table and then load the table.

About this task

To quickly load a large, existing standard table:

1. Drop indexes, referential constraints, and unique constraints.
2. Change the table to nonlogging.

   The following sample SQL statement changes a STANDARD table to nonlogging:

   ```
   ALTER TABLE largetab TYPE(RAW);
   ```

3. Load the table using a load utility such as **dbexport** or the High-Performance Loader (HPL).
   For more information about **dbexport** and **dbload**, see the *HCL OneDB™ Migration Guide*. For more information about HPL, see the *HCL OneDB™ High-Performance Loader User's Guide*.
4. Load the table using a load utility such as **dbexport**.
   For more information about **dbexport** and **dbload**, see the *HCL OneDB™ Migration Guide*.
5. Perform a level-0 backup of the nonlogging table.
   You must make a level-0 backup of any nonlogging table that has been modified before you convert it to STANDARD type. The level-0 backup provides a starting point from which to restore the data.
6. Change the nonlogging table to a logging table before you use it in a transaction.
   The following sample SQL statement changes a raw table to a standard table:

   ```
   ALTER TABLE largetab TYPE(STANDARD);
   ```

   Warning: Do not use nonlogging tables within a transaction where multiple users can modify the data. If you need to use a nonlogging table within a transaction, either set Repeatable Read isolation level or lock the table in exclusive mode to prevent concurrency problems.

   For more information about standard tables, see the previous section, Advantages of logging tables on page 186.
7. Re-create indexes, referential constraints, and unique constraints.

Quickly loading a new nonlogging table

You quickly create a new nonlogging table and load the table.

About this task

To quickly create and load a new, large table:

1. Create a nonlogging table in a logged database.

   The following sample SQL statements create a nonlogging table:

   ```
   CREATE DATABASE history WITH LOG;
   CONNECT TO DATABASE history;
   CREATE RAW TABLE history (...
   );
   ```
2. Load the table using a load utility such as **dbexport**. For more information about **dbexport** and **dbload**, see the *HCL OneDB™ Migration Guide*.
3. Perform a level-0 backup of the nonlogging table.

   You must make a level-0 backup of any nonlogging table that has been modified before you convert it to STANDARD type. The level-0 backup provides a starting point from which to restore the data.
4. Change the nonlogging table to a logging table before you use it in a transaction.

   The following sample SQL statement changes a raw table to a standard table:

   ```
   ALTER TABLE largetab TYPE(STANDARD);
   ```

Warning: Do not use nonlogging tables within a transaction where multiple users can modify the data. If you need to use a nonlogging table within a transaction, either set Repeatable Read isolation level or lock the table in exclusive mode to prevent concurrency problems.

For more information about standard tables, see the previous section, Advantages of logging tables on page 186.

5. Create indexes on columns most often used in query filters.
6. Create any referential constraints and unique constraints, if needed.

Dropping indexes for table-update efficiency

In some applications, you can confine most table updates to a single time period. You can set up your system so that all updates are applied overnight or on specified dates. When updates are performed as a batch, you can drop all nonunique indexes while you make updates and then create new indexes afterward.

About this task

This strategy can have two positive effects:

- The updating program runs much faster if it does not need to update indexes at the same time that it updates tables.
- Re-created indexes are more efficient.

For more information about when to drop indexes, see Nonunique indexes on page 218.

To load a table that has no indexes:

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the nonunique indexes.

Results

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create unique indexes before you load the rows. You save time if the rows are presented in the correct sequence for at least one of the indexes. If you have a choice, make it the row with the largest key. This strategy minimizes the number of leaf pages that must be read and written.

Creating and enabling referential constraints efficiently

When you create or enable foreign-key constraints on existing tables that contain data, you can sometimes achieve better performance by reducing the time that the database server spends searching for violating rows.

By maintaining the referential integrity of the database during DML operations, and by supporting efficient join-query execution paths on tables that are related by a star schema, foreign-key constraints can improve the performance of DML operations in databases where the primary key of each dimension table corresponds to a foreign key of the fact table.

When you use the ALTER TABLE ADD CONSTRAINT or ALTER TABLE MODIFY statement to define a foreign-key constraint on an existing table, you might be able to reduce the time required to validate of the new foreign-key constraint, if the referenced table already has a unique index or a primary-key constraint on the column corresponding to the key of the foreign-key constraint. When it creates a foreign-key constraint on a table that already contains data, the database server checks the table for any rows that violate the constraint. If an index exists, the database server makes a cost-based decision whether to scan every row in the table for violations, or to scan only the index valses.

For large tables, scanning only the index values can provide substantial performance improvement, unless one of the following requirements is not satisfied:

- The ALTER TABLE statement is creating only one foreign-key constraint.
- The ALTER TABLE statement is not also creating or enabling a CHECK constraint.
- The ALTER TABLE statement is not also changing the data type of any existing column in the table.
- The foreign-key columns do not include user-defined data types (UDTs) or built-in opaque data types.
- The new mode of the foreign-key constraint is not DISABLED.
- The table is not associated with an active violation table.

Except in the case of one or more violating rows, the ALTER TABLE ADD CONSTRAINT or ALTER TABLE MODIFY statement can create and validate a foreign-key constraint when some of these requirements are not satisfied, but the database server will not consider using the index-key algorithm to validate the foreign-key constraint. The additional validation costs to scan the entire table tend to be proportional to the size of the table.

Enabling a foreign-key constraint using index-scan validation

To validate the enabled foreign-key constraint, the database server performs a full-table scan to search for violating rows, unless a unique index or a primary-key constraint already exists on the foreign-key column values. In that case, the database server consider using an index scan for validation, unless one or more of the following requirements is not satisfied:

- The SET CONSTRAINTS statement is enabling only one foreign-key constraint.
- The same statement is not enabling a CHECK constraint.
- The foreign-key columns do not include user-defined data types (UDTs) or built-in opaque data types.
- The new mode of the foreign-key constraint is not DISABLED.
- The table is not associated with an active violation table.

Unless the table has one or more violating rows, the SET CONSTRAINTS statement can enable and validate a foreign-key constraint when some of these requirements are not satisfied, but the database server will not consider using the index-key algorithm to validate the foreign-key constraint. The additional validation costs for a full table scan can be substantial for very large tables.

Skipping validation of foreign-key constraints

In both the ALTER TABLE and SET CONSTRAINTS operations described above, the goal was to use a more efficient algorithm for validating the referential constraint. Greater efficiencies can be achieved, at least temporarily, by postponing or avoiding

the validation of ENABLED or FILTERING foreign-key constraints that are being created by ALTER TABLE ADD CONSTRAINT statements, or while a DISABLED foreign-key constraint is being reset to an ENABLED or FILTERING mode.

This feature can be useful when tables that enforced referential constraints need to be moved from an OLTP environment to another database or to a data warehouse. To export the tables and restore their constraints without validation might be necessary if the time available for relocation is insufficient for violations checking. The tables might seem unlikely to include violating rows, if the constraints were dropped or disabled immediately before the tables were exported.

Three alternative mechanisms are available for bypassing the validation of enabled or filtering foreign-key constraints while they are being created, or while they are being exported, or while their mode is being changed from DISABLED:

- You can include the `NOVALIDATE` keyword in the constraint mode specification
  - of the ALTER TABLE ADD CONSTRAINT statement,
  - or of the SET CONSTRAINTS ENABLED statement,
  - or of the SET CONSTRAINTS FILTERING WITH ERROR statement,
  - or of the SET CONSTRAINTS FILTERING WITHOUT ERROR statements.
- If you plan to run multiple ALTER TABLE ADD CONSTRAINT or SET CONSTRAINTS statements, run the SET ENVIRONMENT NOVALIDATE ON statement to disable the validation of foreign-key constraints during the current session.

  Setting this session environment option makes NOVIOLATE the default mode for enabled or filtering referential constraints while the DDL statement is running.
- If you are migrating data, include the -nv option in the dbimport command.

  The effect of the -nv command-line option is that the constraint modes of any ALTER TABLE ADD CONSTRAINT or SET CONSTRAINTS statements that create or enable foreign-key constraints are processed so that the ENABLED, or FILTERING WITH ERROR, or FILTERING WITHOUT ERROR constraint mode specifications are instead implemented (respectively) as the ENABLED NOVALIDATE, or FILTERING WITH ERROR NOVALIDATE, or FILTERING WITHOUT ERROR NOVALIDATE modes.

In each case, no constraint validation of existing rows occurs during the DDL statement.

The effect of the NOVALIDATE keyword or of the -nv command-line flag of dbimport does not persist outside the DDL operation that created or changed the mode of the foreign-key constraint. The same constraint enforces referential integrity during subsequent DELETE, INSERT, MERGE, and UPDATE operations. The NOVALIDATE mode of the referential constraint is not registered in the **sysobjstate** system catalog table.

If a NOVALIDATE constraint mode is used on a table that might already contains rows that violate the foreign-key constraint, it is the responsibility of the user to verify that no violating rows exist in the data.

Attaching or detaching fragments

You can use ALTER FRAGMENT ATTACH and DETACH statements to perform data warehouse-type operations. ALTER FRAGMENT DETACH provides a way to delete a segment of the table data rapidly. Similarly, ALTER FRAGMENT ATTACH

provides a way to load large amounts of data into an existing table incrementally by taking advantage of the fragmentation technology.

For more information about how to take advantage of the performance enhancements for the ATTACH and DETACH options of the ALTER FRAGMENT statement, see Improve the performance of operations that attach and detach fragments on page 278.

Altering a table definition

The database server uses one of these algorithms to process an ALTER TABLE statement in SQL: slow alter, in-place alter, or fast alter.

Slow alter

When the database server uses the slow alter algorithm to process an ALTER TABLE statement, the table can be unavailable to other users for a long period of time.

The table might be unavailable because the database server:

- Locks the table in exclusive mode for the duration of the ALTER TABLE operation
- Makes a copy of the table in order to convert the table to the new definition
- Converts the data rows during the ALTER TABLE operation
- Can treat the ALTER TABLE statement as a long transaction and abort it if the LTXHWM threshold is exceeded

Because the database server makes a copy of the table to convert the table to the new definition, a slow alter operation requires space at least twice the size of the original table plus log space.

The database server uses the slow alter algorithm when the ALTER TABLE statement makes column changes that it cannot perform in place:

- Adding or dropping a column created with the ROWIDS keyword
- Adding or dropping a column created with the REPLCHECK keyword
- Dropping a column of the TEXT or BYTE data type
- Modifying a SMALLINT column to SERIAL, SERIAL8, or BIGSERIAL
- Converting an INT column to SERIAL, SERIAL8, or BIGSERIAL
- Modifying the data type of a column so that some possible values of the old data type cannot be converted to the new data type (For example, if you modify a column of data type INTEGER to CHAR(n), the database server uses the slow alter algorithm if the value of `n` is less than `11`. An INTEGER requires 10 characters plus one for the minus sign for the lowest possible negative values.)
- Modifying the data type of a fragmentation column in a way that value conversion might cause rows to move to another fragment
- Adding, dropping or modifying any column when the table contains user-defined data types, smart large objects, or LVARCHAR, SET, MULTISET, ROW, or COLLECTION data types
- Modifying the original size or reserve specifications of VARCHAR or NVARCHAR columns
- Adding ERKEY shadow columns

In-place alter

The in-place alter algorithm provides numerous performance advantages over the slow alter algorithm

The in-place alter algorithm:

- Increases table availability

  Other users can access the table sooner when the ALTER TABLE operation uses the in-place alter algorithm, because the database server locks the table for only the time that it takes to update the table definition and rebuild indexes that contain altered columns.

  This increase in table availability can increase system throughput for application systems that require 24 by seven operations.

  When the database server uses the in-place alter algorithm, it locks the table for a shorter time than the slow alter algorithm because the database server:

  - Does not make a copy of the table to convert the table to the new definition
  - Does not convert the data rows during the ALTER TABLE operation
  - Alters the physical columns in place with the latest definition after the alter operation when you later update or insert rows. The database server converts the rows that reside on each page that you updated.
- Requires less space than the slow alter algorithm

  When the ALTER TABLE operation uses the slow alter algorithm, the database server makes a copy of the table to convert the table to the new definition. The ALTER TABLE operation requires space at least twice the size of the original table plus log space.

  When the ALTER TABLE operation uses the in-place alter algorithm, the space savings can be substantial for very large tables.
- Improves system throughput during the ALTER TABLE operation

  The database server does not log any changes to the table data during the in-place alter operation. Not logging changes has the following advantages:

  - Log space savings can be substantial for very large tables.
  - The alter operation is not a long transaction.

If the **check_for_ipa** Scheduler task is enabled, each table that has one or more outstanding in-place alter operations is listed in the **ph_alert** table in the **sysadmin** database. The alert text is: `Table database:owner.table_name has outstanding in place alters.` The alert type is informative.

Related information

The ph_alert Table on page

Conditions for in-place alter operations

The database server can use the in-place alter algorithm to process only certain ADD, DROP, or MODIFY operations of the ALTER TABLE statement, and only if the table schema or the ALTER TABLE statement does not require a slow alter algorithm.

ALTER TABLE operations that can be done in place

The database server can use the in-place alter algorithm in the following ALTER TABLE operations:

- Add columns of built-in data types, except the data types that are listed in Conditions that prevent in-place alter operations on page 197.
- Drop a column of built-in data types, except a column that contains TEXT or BYTE data types, or a column that was created with the ROWIDS keyword.
- In Enterprise Replication, add or drop a column that is created with the CRCOLS keyword.
- Modify a column for which the database server can convert all possible values of the old data type to the new data type.
- Modify a column that is part of the fragmentation expression for its table, only if value changes do not require any data row to move from one fragment to another fragment after data type conversion.

The following table shows the conditions under which the ALTER TABLE MODIFY statement uses the in-place alter algorithm to convert columns of supported data types.

Key:

All = The database server uses the in-place alter algorithm for all cases of the specific column operation.
nf = The database server uses the in-place alter algorithm when the modified column is not part of the table fragmentation expression.

Table 12. MODIFY operations and conditions that use the in-place alter algorithm

Operation on Column	Condition
Convert a SMALLINT column to an INTEGER column	All
Convert a SMALLINT column to a BIGINT column	All
Convert a SMALLINT column to an INT8 column	All
Convert a SMALLINT column to a DEC(p2,s2) column	p2-s2 >= 5
Convert a SMALLINT column to a DEC(p2) column	p2-s2 >= 5 OR nf
Convert a SMALLINT column to a SMALLFLOAT column	All
Convert a SMALLINT column to a FLOAT column	All
Convert a SMALLINT column to a CHAR(n) column	n >= 6 AND nf

Table 12. MODIFY operations and conditions that use the in-place alter algorithm (continued)

Operation on Column	Condition
Convert an INT column to an INT8 column	All
Convert an INT column to a DEC(p2,s2) column	p2-s2 >= 10
Convert an INT column to a DEC(p2) column	p2 >= 10 OR nf
Convert an INT column to a SMALLFLOAT column	nf
Convert an INT column to a FLOAT column	All
Convert an INT column to a CHAR(n) column	n >= 11 AND nf
Convert a SERIAL column to an INT8 column	All
Convert a SERIAL column to a DEC(p2,s2) column	p2-s2 >= 10
Convert a SERIAL column to a DEC(p2) column	p2 >= 10 OR nf
Convert a SERIAL column to a SMALLFLOAT column	nf
Convert a SERIAL column to a FLOAT column	All
Convert a SERIAL column to a CHAR(n) column	n >= 11 AND nf
Convert a SERIAL column to a BIGSERIAL column	All
Convert a SERIAL column to a SERIAL8 column	All
Convert a SERIAL8 column to a BIGSERIAL column	All
Convert a BIGSERIAL column to a SERIAL8 column	All
Convert a DEC(p1,s1) column to a SMALLINT column	p1-s1 < 5 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to an INTEGER column	p1-s1 < 10 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to an INT8 column	p1-s1 < 20 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a SERIAL column	p1-s1 < 10 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a BIGSERIAL column	p1-s1 < 20 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a SERIAL8 column	p1-s1 < 20 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a DEC(p2,s2) column	p2-s2 >= p1-s1 AND (s2 >= s1 OR nf)
Convert a DEC(p1,s1) column to a DEC(p2) column	p2 >= p1 OR nf
Convert a DEC(p1,s1) column to a SMALLFLOAT column	nf
Convert a DEC(p1,s1) column to a FLOAT column	nf
Convert a DEC(p1,s1) column to a CHAR(n) column	n >= 8 AND nf

Table 12. MODIFY operations and conditions that use the in-place alter algorithm (continued)

Operation on Column	Condition
Convert a DEC(p1) column to a DEC(p2) column	p2 >= p1 OR nf
Convert a DEC(p1) column to a SMALLFLOAT column	nf
Convert a DEC(p1) column to a FLOAT column	nf
Convert a DEC(p1) column to a CHAR(n) column	n >= 8 AND nf
Convert a SMALLFLOAT column to a DEC(p2) column	nf
Convert a SMALLFLOAT column to a FLOAT column	nf
Convert a SMALLFLOAT column to a CHAR(n) column	n >= 8 AND nf
Convert a FLOAT column to a DEC(p2) column	nf
Convert a FLOAT column to a SMALLFLOAT column	nf
Convert a FLOAT column to a CHAR(n) column	n >= 8 AND nf
Convert a CHAR(m) column to a CHAR(n) column	n >= m OR (nf AND not ANSI mode)
Increase the length of a character-type column	Not in ANSI mode databases
Increase the length of a DECIMAL or MONEY column	All
Convert an INT column to a SERIAL column	All
Convert an INT column to a BIGSERIAL column	All
Convert an INT column to a SERIAL8 column	All
Convert a BIGINT column to a BIGSERIAL column	All
Convert a BIGINT column to a SERIAL8 column	All
Convert a INT8 column to a BIGSERIAL column	All
Convert a INT8 column to a SERIAL8 column	All

Note: If first column of an index is altered, the operation to find the next serial value is very fast as it can make use of the index. If altered column is not first column of an index, the operation will do a sequential scan of the table to find the next serial value.

If you supply the serial value of the altered column, the operation is fast as the serial value is provided and does not require any calculation.

Conditions that prevent in-place alter operations

When the table contains an opaque data type, a user-defined data type, an LVARCHAR data type, a BOOLEAN data type, or a smart large object (BLOB or CLOB), the database server does not use the in-place alter algorithm, even when the column that is being altered is of a data type that can support in-place alter operations.

The in-place alter algorithm is not used if the ALTER TABLE DROP statement specifies BYTE or TEXT columns, or the ROWIDS keyword, or if the ALTER TABLE ADD statement includes the ROWID keyword.

If any column data types in an ALTER TABLE MODIFY statement cannot be converted by in-place alter operations, or if data movement is required for a fragmented table, the database server uses the slow alter algorithm for data type conversion instead of using the in-place alter algorithm.

For example, the database server does not use the in-place alter algorithm in the following situations:

- When more than one algorithm is needed

  For example, assume that an ALTER TABLE MODIFY statement converts a SMALLINT column to a DEC(8,2) column and converts an INTEGER column to a CHAR(8) column. The conversion of the first column is an in-place alter operation, but the conversion of the second column is a slow alter operation. The database server uses the slow alter algorithm to execute this statement.
- When the ALTER TABLE operation moves data records to another fragment

  For example, suppose you have a table with two integer columns and the following fragment expression:

  ```
  col1 < col2 IN dbspace1, REMAINDER IN dbspace2
  ```

  If you issue an ALTER TABLE MODIFY statement to convert the integer values to character values, the database server stores the row `(4, 30)` in **dbspace1** before the alter operation, but stores it in **dbspace2** after the alter operation, not as integers, `4 < 30`, but as characters, `'30' < '4'`.
- When the database server cannot convert all possible values of the old data type to the new data type.

  For example, you cannot convert a BIGSERIAL column to a SERIAL column, because the modified column cannot store BIGSERIAL values that are beyond the range of SERIAL values. (However, you can change a column from SERIAL to BIGSERIAL with an in-place alter operation, if other columns in the table do not conflict with any of the other restrictions on in-place alter operations.)

Related information

IBMInformix data types on page

DECIMAL on page

Performance considerations for DML statements

The database server performs additional actions if it detects any down-level version page during the execution of data manipulation language (DML) statements (INSERT, UPDATE, DELETE, SELECT). These actions can impact performance.

Each time you execute an ALTER TABLE statement that uses the in-place alter algorithm, the database server creates a new version of the table structure. The database server keeps track of all versions of table definitions. The database server resets the version status and all of the version structures and alter structures until the entire table is converted to the final format, or until a slow alter is performed.

If the database server detects any down-level version page during the execution of DML statements (INSERT, UPDATE, DELETE, and SELECT statements, and MERGE statements that specify Insert, Update, or Delete clauses), it performs the following actions:

- For UPDATE statements, the database server converts the entire data page or pages to the final format.
- For INSERT statements, the database server converts the inserted row to the final format and inserts it in the best-fit page. The database server converts the existing rows on the best-fit page to the final format.
- For DELETE statements, the database server does not convert the data pages to the final format.
- For SELECT statements, the database server does not convert the data pages to the final format.

  If your query accesses rows that are not yet converted to the new table definition, you might notice a slight degradation in the performance of your individual query, because the database server reformats each row before it is returned.

Performance of in-place alters for DDL operations

In-place alter operations on data definition language (DDL) statements can slow performance. Therefore, monitor outstanding in-place alter operation because many outstanding alter operations affect subsequent ALTER TABLE statements.

The oncheck -pT command displays data-page versions for outstanding in-place alter operations. An in-place alter is *outstanding* when data pages still exist with the old definition.

Figure 26: Sample oncheck -pT output for the customer table on page 199 shows a portion of the output that the following oncheck command produces after four in-place alter operations are run on the **customer** demonstration table:

Figure 26. Sample oncheck -pT output for the customer table

```
oncheck -pT stores_demo:customer

...
Home Data Page Version Summary

            Version           Count

            0 (oldest)           2
            1           0
            2           0
            3           0
            4 (current)           0
...
```

The **Count** field in Figure 26: Sample oncheck -pT output for the customer table on page 199 displays the number of pages that currently use that version of the table definition. This oncheck output shows that four versions are outstanding:

- A value of `2` in the **Count** field for the oldest version indicates that two pages use the oldest version.
- A value of `0` in the **Count** fields for the next four versions indicates that no pages were to the latest table definition.

Important: As you perform more in-place alter operation on a table, each subsequent ALTER statement or the SQL statements that run against the tables with outstanding alters take more time to run than the previous statement. To maintain efficient performance, regularly remove outstanding in-place alter operations.

You can remove in-place alter operations by running the admin( ) or task( ) SQL administration command with the table update_ipa or fragment update_ipa argument. You can include the parallel option to run the operation in parallel. For example, the following statement removes in-place alter operations in parallel from a table that is named auto:

```
EXECUTE FUNCTION task('table update_ipa parallel','auto');
```

You can remove in-place alter operations by converting data pages to the latest definition with a dummy UPDATE statement. For example, the following statement, which sets a column value to the existing value, causes the database server to convert the format of the data pages to the latest definition:

```
UPDATE tab1 SET col1 = col1;
```

If your goal is saving runtime CPU, then plan to keep as few outstanding alters operations on a table as possible (generally no more than 3 or 4). If your goal is to save on disk space and your alter operations add or grow columns, then leaving in-place alters outstanding helps reduce disk space. If you need to revert to an earlier version of the database server, however, one requirement is that no data pages can include incomplete ALTER TABLE or ALTER FRAGMENT operations.

After all outstanding in-place alter operations have been completed on a table or fragment, the oncheck -pT command displays the total number of data pages in the **Count** field for the current version of the table.

Related information

Resolve outstanding in-place alter operations on page

Altering a column that is part of an index

If the altered column is part of an index, the table is still altered in place, but in this case the database server rebuilds the index or indexes implicitly. If you do not need to rebuild the index, you should drop or disable it before you perform the alter operation. Taking these steps improves performance.

However, if the column that you modify is a primary key or foreign key and you want to keep this constraint, you must specify those keywords again in the ALTER TABLE statement, and the database server rebuilds the index.

For example, suppose you create tables and alter the parent table with the following SQL statements:

```
CREATE TABLE parent
   (si SMALLINT PRIMARY KEY CONSTRAINT pkey);
CREATE TABLE child
   (si SMALLINT REFERENCES parent ON DELETE CASCADE
   CONSTRAINT ckey);
INSERT INTO parent (si) VALUES (1);
INSERT INTO parent (si) VALUES (2);
INSERT INTO child (si) VALUES (1);
INSERT INTO child (si) VALUES (2);
ALTER TABLE parent
   MODIFY (si INT PRIMARY KEY CONSTRAINT pkey);
```

This ALTER TABLE example converts a SMALLINT column to an INT column. The database server retains the primary key because the ALTER TABLE statement specifies the PRIMARY KEY keywords and the **pkey** constraint. When you specify a PRIMARY KEY constraint in the MODIFY clause, the database server also silently creates a NOT NULL constraint on the same primary key column. However, the database server drops any referential constraints to that primary key. Therefore, you must also specify the following ALTER TABLE statement for the child table:

```
ALTER TABLE child
   MODIFY (si int references parent on delete cascade
         constraint ckey);
```

Even though the ALTER TABLE operation on a primary key or foreign key column rebuilds the index, the database server still takes advantage of the in-place alter algorithm. The in-place alter algorithm can provide performance benefits, including the following:

- It does not make a copy of the table in order to convert the table to the new definition.
- It does not convert the data rows during the alter operation.
- It does not rebuild all indexes on the table.

Warning: If you alter a table that is part of a view, you must re-create the view to obtain the latest definition of the table.

Fast alter

The database server uses the fast alter algorithm when the ALTER TABLE statement changes attributes of the table but does not affect the data.

The database server uses the fast alter algorithm when you use the ALTER TABLE statement to:

- Change the next-extent size.
- Add or drop a constraint.
- Change the lock mode of the table.
- Change the unique index attribute without modifying the column type.
- Add shadow columns for row versioning with the ADD VERCOLS keywords.

With the fast alter algorithm, the database server holds the lock on the table for just a short time. In some cases, the database server locks the system catalog tables only to change the attribute. In either case, the table is unavailable for queries for only a short time.

Denormalize the data model to improve performance

You might need to denormalize the data model to reduce overhead and optimize performance.

The entity-relationship data model, which the *HCL OneDB™ Guide to SQL: Tutorial* describes, produces tables that contain no redundant or derived data. According to the tenets of relational database theory, these tables are well structured.

Sometimes, to meet extraordinary demands for high performance, you might need to denormalize the data model by modifying it in ways that are undesirable from a theoretical standpoint. This section describes some modifications and their associated costs.

Shortening rows

Usually, tables with shorter rows yield better performance than those with longer rows because disk I/O is performed in pages, not in rows. The shorter the rows of a table, the more rows occur on a page. The more rows per page, the fewer I/O operations it takes to read the table sequentially, and the more likely it is that a nonsequential access can be performed from a buffer.

The entity-relationship data model puts all the attributes of one entity into a single table for that entity. For some entities, this strategy can produce rows of awkward lengths.

To shorten the rows, you can break columns into separate tables that are associated by duplicate key values in each table. As the rows get shorter, query performance should improve.

Expelling long strings

The most bulky attributes are often character strings. To make the rows shorter, you can remove long strings from the entity table.

You can use the following methods to expel long strings:

- Use VARCHAR columns.
- Use TEXT data.
- Move strings to a companion table.
- Build a symbol table.

Convert CHAR columns into VARCHAR columns to shorten rows (GLS)

A database might contain CHAR columns that you can convert to VARCHAR columns. You can use a VARCHAR column to shorten the average row length when the average length of the text string in the CHAR column is at least 2 bytes shorter than the width of the column.

VARCHAR data is immediately compatible with most existing programs, forms, and reports. You might need to recompile any forms produced by application development tools to recognize VARCHAR columns. Always test forms and reports on a sample database after you modify the table schema.

For information about other character data types, see the *HCL OneDB™ GLS User's Guide*.

Convert a long string to a TEXT data type column

When a string fills half a disk page or more, consider converting it to a TEXT data type column in a separate blobspace.

The column within the row page is only 56 bytes long, which allows more rows on a page than when you include a long string. However, the TEXT data type is not automatically compatible with existing programs. The application needed to fetch a TEXT value is a bit more complicated than the code for fetching a CHAR value into a program.

Move strings to a companion table

Strings that are less than half a page waste disk space if you treat them as TEXT data, but you can move them from the main table to a companion table.

If you split a table into two tables, the primary table and a companion table, repeat the primary key in each table.

Build a symbol table

If a column contains strings that are not unique in each row, you can move those strings to a table in which only unique copies are stored.

For example, the **customer.city** column contains city names. Some city names are repeated in the column, and most rows have some trailing blanks in the field. Using the VARCHAR data type eliminates the blanks but not the duplication.

You can create a table named **cities**, as the following example shows:

```
CREATE TABLE cities (
   city_num SERIAL PRIMARY KEY,
   city_name VARCHAR(40) UNIQUE
)
```

You can change the definition of the **customer** table so that its **city** column becomes a foreign key that references the **city_num** column in the **cities** table.

To insert the city of the new customer into **cities**, you must change any program that inserts a new row into **customer.** The database server return code in the **SQLCODE** field of the SQL Communications Area (SQLCA) can indicate that the insert failed because of a duplicate key. It is not a logical error; it simply means that an existing customer is located in that city. For more information about the SQLCA, see the *HCL OneDB™ Guide to SQL: Tutorial*.

Besides changing programs that insert data, you must also change all programs and stored queries that retrieve the city name. The programs and stored queries must use a join to the new **cities** table in order to obtain their data. The extra complexity in programs that insert rows and the extra complexity in some queries is the result of giving up theoretical correctness in the data model. Before you make the change, be sure that it returns a reasonable savings in disk space or execution time.

Splitting wide tables

Consider all the attributes of an entity that has rows that are too wide for good performance. Look for some theme or principle to divide them into two groups. Then split the table into two tables, a primary table and a companion table, repeating the primary key in each one.

The shorter rows allow you to query or update each table quickly.

Division by Bulk

One principle on which you can divide an entity table is bulk. Move the bulky attributes, which are usually character strings, to the companion table. Keep the numeric and other small attributes in the primary table. In the demonstration database, you can split the **ship_instruct** column from the **orders** table. You can call the companion table **orders_ship**. It has two columns, a primary key that is a copy of **orders.order_num** and the original **ship_instruct** column.

Division by Frequency of Use

Another principle for division of an entity is frequency of use. If a few attributes are rarely queried, move them to a companion table. In the demonstration database, for example, perhaps only one program queries the **ship_instruct**, **ship_weight**, and **ship_charge** columns. In that case, you can move them to a companion table.

Division by Frequency of Update

Updates take longer than queries, and updating programs lock index pages and rows of data during the update process, preventing querying programs from accessing the tables. If you can separate one table into two companion tables, one with the most-updated entities and the other with the most-queried entities, you can often improve overall response time.

Performance Costs of Splitting Tables

Splitting a table uses extra disk space and adds complexity. Two copies of the primary key occur for each row, one copy in each table. Two primary-key indexes also exist. You can use the methods described in earlier sections to estimate the number of added pages.

You must modify existing programs, reports, and forms that use SELECT * because fewer columns are returned. Programs, reports, and forms that use attributes from both tables must perform a join to bring the tables together.

In this case, when you insert or delete a row, two tables are altered instead of one. If you do not coordinate the alteration of the two tables (by making them within a single transaction, for example), you lose semantic integrity.

Redundant data

Normalized tables contain no redundant data. Every attribute appears in only one table.

Normalized tables also contain no derived data. Instead, data that can be computed from existing attributes is selected as an expression based on those attributes.

Normalizing tables minimizes the amount of disk space used and makes updating the tables as easy as possible. However, normalized tables can force you to use joins and aggregate functions often, and those processes can be time consuming.

As an alternative, you can introduce new columns that contain redundant data, provided you understand the trade-offs involved.

Adding redundant data

A correct data model avoids redundancy by keeping any attribute only in the table for the entity that it describes. If the attribute data is needed in a different context, you join tables to make the connection. But joining takes time. If a frequently used join affects performance, you can eliminate it by duplicating the joined data in another table.

In the **stores_demo** database, the **manufact** table contains the names of manufacturers and their delivery times. An actual working database might contain many other attributes of a supplier, such as address and sales representative name.

The contents of **manufact** are primarily a supplement to the **stock** table. Suppose that a time-critical application frequently refers to the delivery lead time of a particular product but to no other column of **manufact**. For each such reference, the database server must read two or three pages of data to perform the lookup.

You can add a new column, **lead_time**, to the **stock** table and fill it with copies of the **lead_time** column from the corresponding rows of **manufact**. That arrangement eliminates the lookup and therefore speeds up the application.

Like derived data, redundant data takes space and poses an integrity risk. In the example described in the previous paragraph, many extra copies of the lead time for each manufacturer can exist. (Each manufacturer can appear in **stock** many times.) The programs that insert or update a row of **manufact** must also update multiple rows of **stock**.

The integrity risk is simply that the redundant copies of the data might not be accurate. If a lead time is changed in **manufact**, the **stock** column is outdated until it is also updated. As you do with derived data, define the conditions under which redundant data might be wrong.

For more information about database design, see the *HCL OneDB™ Database Design and Implementation Guide*.

Reduce disk space in tables with variable length rows

You can enable the database server to insert more rows per page into tables with variable-length rows, if you set the MAX_FILL_DATA_PAGES configuration parameter to `1`. Allowing more variable length rows per page has advantages and disadvantages.

Potential advantages of allowing more variable length rows per page are:

- Reducing the disk space required to store data
- Enabling the server to use the buffer pool more efficiently
- Reducing table scan times

Possible disadvantages of using the MAX_FILL_DATA_PAGES allowing more variable length rows per page are:

- The server might store rows in a different physical order.
- As the page fills, updates made to the variable-length columns in a row could cause the row to expand so it no longer completely fits on the page. This causes the server to split the row onto two pages, increasing the access time for the row.

If the MAX_FILL_DATA_PAGES configuration parameter is enabled, the server will add a new row to a recently modified page with existing rows if adding the row leaves at least 10 percent of the page free for future expansion of all the rows in the page. If the MAX_FILL_DATA_PAGES configuration parameter is not enabled, the server will add the row only if there is sufficient room on the page to allow the new row to grow to its maximum length.

If you enable the MAX_FILL_DATA_PAGES configuration parameter and you want this to affect existing variable length rows, the existing tables must be reloaded.

Reduce disk space by compressing tables and fragments

You can reduce disk space by compressing data in tables and table fragments. After compressing data, you can repack the data to consolidate the free space in a table or fragment, and shrink the space for the data to return the free space to the dbspace.

Compression is advantageous for applications with a lot of I/O activity and for applications in which the reduction of disk space usage is critical. However, if your applications run with high buffer cache hit ratios and high performance is more important than space usage, you might not want to compress data, because compression might slightly decrease performance.

Compressing data, consolidating data, and returning free space have the following benefits:

- Significant savings in disk storage space
- Reduced disk usage for compressed fragments
- Significant saving of logical log usage, which saves additional space and can prevent bottlenecks for high-throughput OLTP after the compression operation is completed.

- Fewer page reads, because more rows can fit on a page
- Smaller buffer pools, because more data fits in the same size pool
- Reduced I/O activity, because:
  - More compressed rows than uncompressed rows fit on a page
  - Log records for insert, update, and delete operations of compressed rows are smaller
- Ability to compress older fragments of time-fragmented data that are not often accessed, while leaving more recent data that is frequently accessed in uncompressed form
- Ability to free space no longer needed for a table
- Faster backup and restore

Because compressed data covers fewer pages and has more rows per page than uncompressed data, the query optimizer might choose different plans after compression.

You can speed up compression and repacking by running the operations in parallel.

Related information

Compression

table or fragment arguments: Compress data and optimize storage (SQL administration API) on page

Indexes and index performance considerations

HCL OneDB™ provides several types of indexes. Some performance issues are associated with indexes.

Types of indexes

HCL OneDB™ uses B-tree indexes, R-tree indexes, functional indexes, and indexes that DataBlade® modules provide for user-defined data. The server also uses forest of trees (FOT) indexes, which are alternatives to B-tree indexes.

Related information

What is a functional index? on page 234

B-tree indexes

HCL OneDB™ uses a B-tree index for columns that contain built-in data types (referred to as a *traditional B-tree index*), columns that contain one-dimensional user-defined data types (referred to as a *generic B-tree index*), and values that a user-defined data type returns.

Built-in data types include character, datetime, integer, float, and so forth. For more information about built-in data types, see *HCL OneDB™ Guide to SQL: Reference*.

User-defined data types include opaque and distinct data types. For more information about user-defined data types, see *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

The return value of a user-defined function can be a built-in or user-defined data type, but not a simple large object (TEXT or BYTE data type) or a smart large object (BLOB or CLOB data type). For more information about how to use functional indexes, see Using a functional index on page 233.

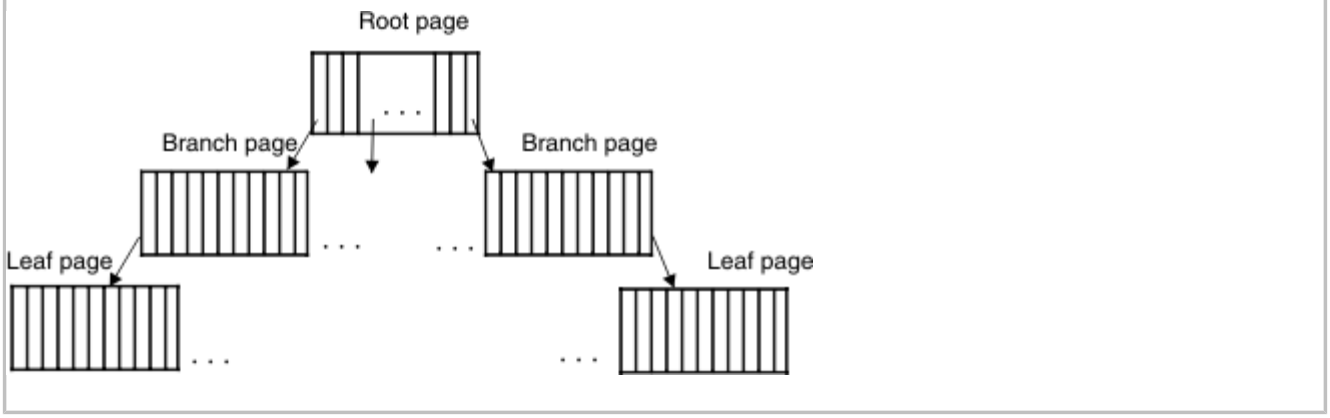
For information about how to estimate B-tree index size, see Estimating index pages on page 209.

Structure of conventional index pages

A conventional index is arranged as a hierarchy of pages (technically, a *B-tree*).

The following figure shows the B-tree structure of an index. The topmost level of the hierarchy contains a single *root page*. Intermediate levels, when needed, contain *branch pages*. Each branch page contains entries that see a subset of pages in the next level of the index. The bottom level of the index contains a set of *leaf pages*. Each leaf page contains a list of index entries that see rows in the table.

Figure 27. B-tree structure of an index



The number of levels needed to hold an index depends on the number of unique keys in the index and the number of index entries that each page can hold. The number of entries per page depends, in turn, on the size of the columns being indexed.

If the index page for a given table can hold 100 keys, a table of up to 100 rows requires a single index level: the root page. When this table grows beyond 100 rows, to a size between 101 and 10,000 rows, it requires a two-level index: a root page and between 2 and 100 leaf pages. When the table grows beyond 10,000 rows, to a size between 10,001 and 1,000,000 rows, it requires a three-level index: the root page, a set of 100 branch pages, and a set of up to 10,000 leaf pages.

Index entries contained within leaf pages are sorted in key-value order. An index entry consists of a *key* and one or more *row pointers*. The key is a copy of the indexed columns from one row of data. A row pointer provides an address used to locate a row that contains the key. A unique index contains one index entry for every row in the table.

For information about special indexes for HCL OneDB™, see Indexes on user-defined data types on page 229.

Related information

Forest of trees indexes on page 208

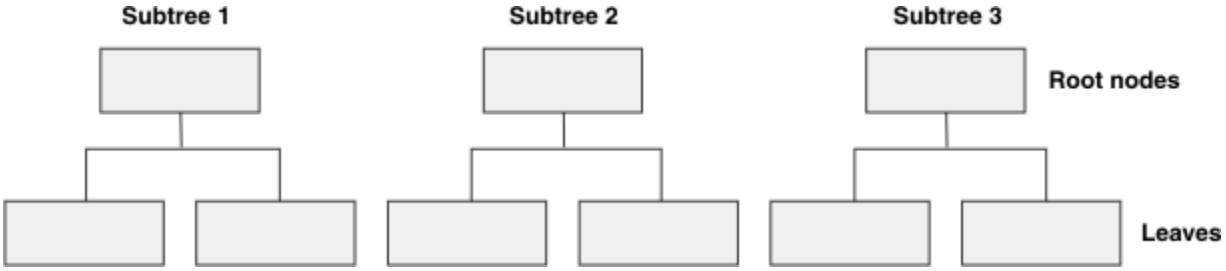
Forest of trees indexes

A forest of trees index is like a B-tree index, but it has multiple root nodes and potentially fewer levels. Multiple root nodes can alleviate root node contention, because more concurrent users can access the index. A forest of trees index can also improve the performance of a query by reducing the number of levels involved in buffer read operations.

You can create a forest of trees index as an alternative to a B-Tree index, but not as an alternative to an R-Tree index or other types of indexes.

Unlike a traditional B-tree index, which contains one root node, a forest of trees index is a large B-Tree index that is divided into smaller subtrees (which you can think of as buckets). These subtrees contain multiple root nodes and leaves. The following figure shows the structure of a forest of trees index.

Figure 28. Structure of a forest of trees index



HCL OneDB™ stores and retrieves an item from a subtree by:

1. Computing a hash value from the columns that you selected when creating the index.
2. Mapping the hash value to a subtree for storage or retrieval of the row.

Forest of trees indexes are detached indexes. The server does not support forest of trees attached indexes.

You create a forest of trees index with the CREATE INDEX statement of SQL and the HASH ON clause.

You enable or disable forest of trees indexes with the SET INDEXES statement of SQL.

You can identify a forest of trees index by the `FOT` indicator in the `Index Name` field in SET EXPLAIN output.

You can look up the number of hashed columns and subtrees in a forest of trees index by viewing information in the **sysindices** table for the database containing tables that have forest of trees indexes.

The server treats a forest of trees index the same way it treats a B-tree index. Therefore, in a logged database, you can control how the B-tree scanner threads remove deletions from both forest of trees and B-tree indexes.

Restrictions: You cannot:

- Create forest of trees indexes on columns with complex data types, UDTs, or functional columns.
- Use the FILLFACTOR option of the CREATE INDEX statement when you create forest of trees indexes, because the indexes are built from top to bottom.
- Create clustered forest of trees indexes.

- Run the ALTER INDEX statement on forest of trees indexes.
- Run the SET INDEXES statement on forest of trees indexes in a database of secondary servers within a cluster environment.
- Use forest of trees indexes in queries that use aggregates, including minimum and maximum range values.
- Perform range scans directly on the HASH ON columns of a forest of trees index.

  However, you can perform range scans on columns that are not listed in the HASH ON column list. For range scans on columns listed in HASH ON column list, you must create an additional B-tree index that contains the appropriate column list for the range scan. This additional B-tree index might have the same column list as the forest of trees index, plus or minus a column.
- Use a forest of trees index for an `OR` index path. The database server does not use forest of trees indexes for queries that have an `OR` predicate on the indexed columns.

Related information

Improve query performance with a forest of trees index on page 219

Detecting root node contention on page 220

Creating a forest of trees index on page 221

Disabling and enabling a forest of trees index on page 221

Determining if you are using a forest of trees index on page 223

Structure of conventional index pages on page 207

CREATE INDEX statement on page

HASH ON clause on page

R-tree indexes

HCL OneDB™ uses an R-tree index for spatial data (such as two-dimensional or three-dimensional data).

For information about sizing an R-tree index, see the *R-Tree Index User's Guide*.

Indexes that DataBlade® modules provide

DataBlade® modules can contain user-defined data types. A DataBlade® module can also provide a user-defined index for the new data type.

For more information about the types of data and functions that each DataBlade® module provides, see the user guide of each DataBlade® module. For information about how to determine the types of indexes available in your database, see Identifying the available access methods on page 231.

Estimating index pages

The index pages associated with a table can add significantly to the size of a dbspace.

By default, the database server creates the index in the same dbspace as the table, but in a separate tblspace from the table. To place the index in a separate dbspace, specify the IN keyword in the CREATE INDEX statement.

Although you cannot explicitly specify the extent size of an index, you can estimate the number of pages that an index might occupy to determine if your dbspace or dbspaces have enough space allocated.

Index extent sizes

The database server determines the extent size of an index based on the extent size for the corresponding table, regardless of whether the index is fragmented or not fragmented.

Formula for estimating the extent size of an attached index

For an attached index, the database server uses the ratio of the index key size to the row size to assign an appropriate extent size for the index.

The following formula shows how the database server uses the ratio of the index key size to the row size:

```
Index extent size = (index_key_size /
table_row_size) *
    table_extent_size
```

In this formula:

- *index_key_size* is the total widths of the indexed column or columns plus 5 for a key descriptor.
- *table_row_size* is the sum of all the columns in the row.
- *table_extent_size* is the value that you specify in the EXTENT SIZE keyword of the CREATE TABLE statement.

If the index is not unique, then the extent size is reduced by 20 percent.

The database server also uses this same ratio for the next-extent size for the index:

```
Index next extent size =
(index_key_size/table_row_size)*
  table_next_extent_size
```

Formula for estimating the extent size of a detached index

For a detached index, the database server uses the ratio of the index key size plus some overhead bytes to the row size to assign an appropriate extent size for the index.

The following formula shows how the database server uses the ratio of the index key size plus some overhead bytes to the row size:

```
Detached Index extent size = ( (index_key_size +
9) /              table_row_size) *
 table_extent_size
```

For example, suppose you have the following values:

```
index_key_size = 8 bytes
table_row_size = 33 bytes
table_extent_size = 150 * 2-kilobyte page
```

The above formula calculates the extent size as follows:

```
Detached Index extent size = ( (8 + 9) /
33) * 150 * 2-kilobyte page
                           = (17/33) * 300 kilobytes
                           = 154 kilobytes
```

Important: For a non-unique index, the formula calculates an extent size that is reduced by 20 percent.

Estimating conventional index pages

You can estimate the size of index pages, using a series of formulas.

About this task

To estimate the number of index pages:

1. Add up the total widths of the indexed column or columns.

   This value is referred to as colsize. Add 4 to *colsize* to obtain *keysize*, the actual size of a key in the index. For example, if *colsize* is 6, the value of *keysize* is 10.

2. Calculate the expected proportion of unique entries to the total number of rows.

   The formulas in subsequent steps see this value as *propunique*.

   If the index is unique or has few duplicate values, use 1 for *propunique*.

   If a significant proportion of entries are duplicates, divide the number of unique index entries by the number of rows in the table to obtain a fractional value for *propunique*. For example, if the number of rows in the table is 4,000,000 and the number of unique index entries is 1,000,000, the value of *propunique* is .25.

   If the resulting value for *propunique* is less than .01, use .01 in the calculations that follow.

3. Estimate the size of a typical index entry with one of the following formulas, depending on whether the table is fragmented or not:

   a. For nonfragmented tables, use the following formula:

      ```
      entrysize = (keysize * propunique) + 5 + 4
      ```

      The value 5 represents the number of bytes for the row pointer in a nonfragmented table.

      For nonunique indexes, the database server stores the row pointer for each row in the index node but stores the key value only once. The `entrysize` value represents the average length of each index entry, even though some entries consist of only the row pointer.

For example, if *propunique* is .25, the average number of rows for each unique key value is 4. If *keysize* is 10, the value of *entrysize* is 11.5, calculated as (10 * 0.25) + 5 + 4 = 2.5 + 9 = 11.5. The following calculation shows the space required for all four rows:

```
space for four rows = 4 * 11.5 = 46
```

This space requirement is the same when you calculate it for the key value and add the four row pointers, as the following formula shows:

```
space for four rows = 10 + (4 * 9) = 46
```

b. For fragmented tables, use the following formula:

```
entrysize = (keysize * propunique) + 9 + 4
```

The value 9 represents the number of bytes for the row pointer in a fragmented table.

4. Estimate the number of entries per index page with the following formula:

```
pagents = trunc(pagefree/entrysize)
```

In this formula:

- *pagefree* is the page size minus the page header (2020 for a 2-kilobyte page size).
- *entrysize* is the size of a typical index entry, which you estimated in the previous step.

The **trunc()** function notation indicates that you should round down to the nearest integer value.

5. Estimate the number of leaf pages with the following formula:

```
leaves = ceiling(rows/pagents)
```

In this formula:

- *rows* is the number of rows that you expect to be in the table.
- *pagents* is the number of entries per index page, which you estimated in the previous step.

The **ceiling()** function notation indicates that you should round up to the nearest integer value.

6. Estimate the number of branch pages at the second level of the index with the following formula:

```
branches0 = ceiling(leaves/node_ents)
```

Calculate the value for *node_ents* with the following formula:

```
node_ents = trunc( pagefree / ( keysize + 4) + 4)
```

In this formula:

- *pagefree* is the page size minus the page header (2020 for a 2-kilobyte page size).
- keysize is the *colsize* plus 4. You obtained this value in step 1.

In the formula, 4 represents the number of bytes for the leaf node pointer.

7. If the value of $branches_0$ is greater than 1, more levels remain in the index.

To calculate the number of pages contained in the next level of the index, use the following formula:

$$branches_{n+1} = ceiling(branches_n / node\_ents)$$

In this formula:

- $branches_n$ is the number of branches for the last index level that you calculated.
- $branches_{n+1}$ is the number of branches in the next level.
- *node_ents* is the value that you calculated in step 6.

8. Repeat the calculation in step 7 for each level of the index until the value of $branches_{n+1}$ equals 1.
9. Add up the total number of pages for all branch levels calculated in steps 6 through 8. This sum is called *branchtotal*.
10. Use the following formula to calculate the number of pages in the compact index:

    $$compactpages = (leaves + branchtotal)$$

11. If your database server instance uses a fill factor for indexes, the size of the index increases.

    The default fill factor value is 90 percent. You can change the fill factor value for all indexes with the FILLFACTOR configuration parameter. You can also change the fill factor for an individual index with the FILLFACTOR clause of the CREATE INDEX statement in SQL.

    To incorporate the fill factor into your estimate for index pages, use the following formula:

    $$indexpages = 100 * compactpages / FILLFACTOR$$

Results

The preceding estimate is a guideline only. As rows are deleted and new ones are inserted, the number of index entries can vary within a page. This method for estimating index pages yields a conservative (high) estimate for most indexes. For a more precise value, build a large test index with real data and check its size with the **oncheck** utility.

Tip: A forest of trees index can be larger than a B-Tree index. When you estimate the size of a forest of trees index, the estimates apply to each subtree in the index. Then, you must aggregate the buckets to calculate the total estimation.

Managing indexes

An index on the appropriate column can save thousands, tens of thousands, or in extreme cases, even millions of disk operations during a query. However, indexes entail costs.

An index is necessary on any column or combination of columns that must be unique. However, as discussed in Queries and the query optimizer on page 290, the presence of an index can also allow the query optimizer to speed up a query.

The optimizer can use an index in the following ways:

- To replace repeated sequential scans of a table with nonsequential access
- To avoid reading row data when processing expressions that name only indexed columns
- To avoid a sort (including building a temporary table) when executing the GROUP BY and ORDER BY clauses

Related information

Using a functional index on page 233

Space costs of indexes

The first cost of an index is disk space. The presence of an index can add many pages to a dbspace; it is easy to have as many index pages as row pages in an indexed table. Additionally, in an environment where multiple languages are used, indexes created for each language require additional disk space.

When you consider space costs, also consider whether increasing the page size of a standard or temporary dbspace is beneficial in your environment. If you want a longer key length than is available for the default page size, you can increase the page size. If you increase the page size, the size must be an integral multiple of the default page size, not greater than 16K bytes.

You might not want to increase the page size if your application contains small sized rows. Increasing the page size for an application that randomly accesses small rows might decrease performance. In addition, a page lock on a larger page will lock more rows, reducing concurrency in some situations.

You can save disk space by compressing detached B-tree indexes, consolidating free space in the index, and returning the free space to the dbspace.

Related information

B-tree index compression on page

Time costs of indexes

The second cost of an index is time whenever the table is modified.

The following descriptions assume that approximately two pages must be read to locate an index entry. That is the case when the index consists of a root page, one level of branch pages, and a set of leaf pages. The root page is assumed to be in a buffer already. The index for a very large table has at least two intermediate levels, so about three pages are read when the database server references such an index.

Presumably, one index is used to locate a row being altered. The pages for that index might be found in page buffers in shared memory for the database server. However, the pages for any other indexes that need altering must be read from disk.

Under these assumptions, index maintenance adds time to different kinds of modifications, as the following list shows:

- When you delete a row from a table, the database server must delete its entries from all indexes.

  The database server must look up the entry for the deleted row (two or three pages in) and rewrite the leaf page. The write operation to update the index is performed in memory, and the leaf page is flushed when the least recently used (LRU) buffer that contains the modified page is cleaned. This operation requires two or three page accesses to read the index pages if needed and one deferred page access to write the modified page.

- When you insert a row, the database server must insert its entries in all indexes.

  The database server must find a place in which to enter the inserted row within each index (two or three pages in) and rewrite (one deferred page out), for a total of three or four immediate page accesses per index.
- When you update a row, the database server must look up its entries in each index that applies to an altered column (two or three pages in).

  The database server must rewrite the leaf page to eliminate the old entry (one deferred page out) and then locate the new column value in the same index (two or three more pages in) and the row entered (one more deferred page out).

Insertions and deletions change the number of entries on a leaf page. Although virtually every *pagents* operation requires some additional work to deal with a leaf page that has either filled or been emptied, if *pagents* is greater than 100, this additional work occurs less than 1 percent of the time. You can often disregard it when you estimate the I/O impact.

In short, when a row is inserted or deleted at random, allow three to four added page I/O operations per index. When a row is updated, allow six to eight page I/O operations for each index that applies to an altered column. If a transaction is rolled back, all this work must be undone. For this reason, rolling back a transaction can take a long time.

Because the alteration of the row itself requires only two page I/O operations, index maintenance is clearly the most time-consuming part of data modification. For information about one way to reduce this cost, see Clustering on page 217.

Unclaimed index space

A background thread, the B-tree scanner, identifies an index with the most unclaimed index space. Unclaimed index space degrades performance and causes extra work for the server. When an index is chosen for scanning, the entire leaf of the index is scanned for deleted (dirty) items that were committed, but not yet removed from the index. The B-tree scanner removes these items when necessary.

The B-tree scanner allows multiple threads.

Use the BTSCANNER configuration parameter to specify the number of B-tree scanner threads to start and the priority of the B-tree scanner threads when the database server starts. For details, see the *HCL OneDB™ Administrator's Reference*.

You can invoke the B-tree scanner from the command line.

Indexes on columns

You can create an index for one or more columns in a table. Indexes are required on columns that must be unique and are not specified as primary keys.

In addition, you must add an index on columns that:

- Are used in joins that are not specified as foreign keys
- Are frequently used in filter expressions
- Are frequently used for ordering or grouping
- Do not involve duplicate keys
- Are amenable to clustered indexing

Filtered columns in large tables

If a column is often used to filter the rows of a large table, consider placing an index on it. The optimizer can use the index to select the wanted columns and avoid a sequential scan of the entire table.

Suppose you have a table that contains a large mailing list. If you find that a postal-code column is often used to filter a subset of rows, consider putting an index on that column.

This strategy yields a net savings of time only when the selectivity of the column is high; that is, when only a small fraction of rows holds any one indexed value. Nonsequential access through an index takes several more disk I/O operations than sequential access does, so if a filter expression on the column passes more than a fourth of the rows, the database server might as well read the table sequentially.

As a rule, indexing a filter column saves time in the following cases:

- The column is used in filter expressions in many queries or in slow queries.
- The column contains at least 100 unique values.
- Most column values appear in fewer than 10 percent of the rows.

Order-by and group-by columns

You can place an index on the ordering column or columns of a table. The database server then uses the index that to sort the query results in the most efficient manner.

When a large quantity of rows must be ordered or grouped, the database server must put the rows in order. One way that the database server performs this task is to select all the rows into a temporary table and sort the table. But, as explained in Queries and the query optimizer on page 290, if the ordering columns are indexed, the optimizer sometimes reads the rows in sorted order through the index, thus avoiding a final sort.

Because the keys in an index are in sorted sequence, the index really represents the result of sorting the table. By placing an index on the ordering column or columns, you can replace many sorts during queries with a single sort when the index is created.

Avoiding columns with duplicate keys

Duplicate keys in indexes can cause performance problems. You can take steps to avoid these problems.

When duplicate keys are permitted in an index, entries that match a given key value are grouped in lists. The database server uses these lists to locate rows that match a requested key value. When the selectivity of the index column is high, these lists are generally short. But when only a few unique values occur, the lists become long and can cross multiple leaf pages.

Placing an index on a column that has low selectivity (that is, a small number of distinct values relative to the number of rows) can reduce performance. In such cases, the database server must not only search the entire set of rows that match the key value, but it must also lock all the affected data and index pages. This process can impede the performance of other update requests as well.

To correct this problem, replace the index on the low-selectivity column with a composite index that has a higher selectivity. Use the low-selectivity column as the leading column and a high-selectivity column as your second column in the index. The composite index limits the number of rows that the database server must search to locate and apply an update.

You can use any second column to disperse the key values as long as its value does not change, or changes at the same time as the real key. The shorter the second column the better, because its values are copied into the index and expand its size.

Clustering

Clustering is a method for arranging the rows of a table so that their physical order on disk closely corresponds to the sequence of entries in the index.

(Do not confuse the clustered index with an *optical cluster*, which is a method for storing logically related TEXT or BYTE data together on an optical volume.)

When you know that a table is ordered by a certain index, you can avoid sorting. You can also be sure that when the table is searched on that column, it is read effectively in sequential order, instead of nonsequentially. These points are covered in Queries and the query optimizer on page 290.

Tip: For information about eliminating interleaved extents by altering an index to cluster, see Creating or altering an index to cluster on page 182.

In the **stores_demo** database, the **orders** table has an index, **zip_ix**, on the postal-code column. The following statement causes the database server to put the rows of the **customer** table in descending order by postal code:

```
ALTER INDEX zip_ix TO CLUSTER
```

To cluster a table on a nonindexed column, you must create an index. The following statement reorders the **orders** table by order date:

```
CREATE CLUSTER INDEX o_date_ix ON orders (order_date ASC)
```

To reorder a table, the database server must copy the table. In the preceding example, the database server reads all the rows in the table and constructs an index. Then it reads the index entries in sequence. For each entry, it reads the matching row of the table and copies it to a new table. The rows of the new table are in the desired sequence. This new table replaces the old table.

Clustering is not preserved when you alter a table. When you insert new rows, they are stored physically at the end of the table, regardless of their contents. When you update rows and change the value of the clustering column, the rows are written back into their original location in the table.

Clustering can be restored after the order of rows is disturbed by ongoing updates. The following statement reorders the table to restore data rows to the index sequence:

```
ALTER INDEX o_date_ix TO CLUSTER
```

Reclustering is usually quicker than the original clustering because reading out the rows of a nearly clustered table is similar in I/O impact to a sequential scan.

Clustering and reclustering take a lot of space and time. To avoid some clustering, build the table in the desired order initially.

Related information

Reclaiming space in an empty extent with ALTER INDEX on page 183

Configuration parameters that affect the degree of clustering

The **clust** field in the **sysindexes** or the **sysindices** table represents the degree of clustering of the index. The values of several configuration parameters affect the **clust** field.

The value of this field is affected by:

- The size of the buffer pool as specified by the BUFFERPOOL configuration parameter
- The value in the **buffers** field of the BUFFERPOOL configuration parameter
- The DS_MAX_QUERIES configuration parameter, which specifies the maximum number of PDQ queries that can run concurrently

Each of these configuration parameters affects the amount of buffer space available for a single user session. Additional buffers can result in better clustering (a smaller **clust** value in the **sysindexes** or **sysindices** tables).

You can create more buffers by performing one or both of the following tasks:

- Increasing the size of the buffer pool by updating the value of the BUFFERPOOL configuration parameter
- Decreasing the value of the DS_MAX_QUERIES configuration parameter

Related information

BUFFERPOOL configuration parameter on page

DS_MAX_QUERIES configuration parameter on page

Nonunique indexes

In some applications, most table updates can be confined to a single time period. You might be able to set up your system so that all updates are applied overnight or on specified dates. Additionally, when updates are performed as a batch, you can drop all nonunique indexes while you make updates and then create new indexes afterward. This strategy can improve performance.

About this task

Dropping nonunique indexes can have the following positive effects:

- The updating program can run faster with fewer indexes to update. Often, the total time to drop the indexes, update without them, and re-create them is less than the time to update with the indexes in place. (For a discussion of the time cost of updating indexes, see Time costs of indexes on page 214.)
- Newly made indexes are more efficient. Frequent updates tend to dilute the index structure so that it contains many partly full leaf pages. This dilution reduces the effectiveness of an index and wastes disk space.

As a time-saving measure, make sure that a batch-updating program calls for rows in the sequence that the primary-key index defines. That sequence causes the pages of the primary-key index to be read in order and only one time each.

The presence of indexes also slows down the population of tables when you use the LOAD statement or the **dbload** utility. Loading a table that has no indexes is a quick process (little more than a disk-to-disk sequential copy), but updating indexes adds a great deal of overhead.

To avoid this overhead, you can:

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the nonunique indexes.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create unique indexes before you load the rows. It saves time if the rows are presented in the correct sequence for at least one of the indexes. If you have a choice, make it the row with the largest key. This strategy minimizes the number of leaf pages that must be read and written.

Improve query performance with a forest of trees index

A forest of trees index is an alternate indexing method that alleviates the performance bottlenecks and root node contention that can occur when many concurrent users access a traditional B-tree index.

About this task

A forest of trees index differs from a B-tree index in that it has multiple root nodes and fewer levels. Multiple root nodes can alleviate root node contention, because more concurrent users can access the index.

If you know that a particular table has a deep tree, you can improve performance by creating a forest of trees index with fewer levels in the tree. For example, suppose you create an index where one of the columns is a 100 byte column containing character data. If you have a large number of rows in that table, the tree might contain six or seven levels. If you create a forest of trees index instead of a B-tree index, you can create more than one tree with four levels, so that every index traversal goes only four levels deep rather than seven levels deep.

Related information

Forest of trees indexes on page 208

Detecting root node contention

You can analyze the output of the onstat -g spi command to identify the performance bottlenecks that a forest of trees index can alleviate.

About this task

To detect root node contention and determine whether you need a forest of trees index:

1. Run the onstat -g spi | sort -nr command to display information about spin locks with long spins.

   The output of the onstat -g spi command shows spin locks with waits, which occur when threads are reading from or writing to an index concurrently and a particular thread did not succeed in acquiring the lock on the first try.

2. Analyze the onstat -g spioutput. Look for loop and wait information in these columns:
   - `Num Waits`: The Total number of times a thread waited for the spin lock.
   - `Num Loops`: The total number of attempts before a thread successfully acquired the spin lock.
   - `Avg Loop/Wait`: The average number of attempts to acquire the spin lock, computed as `Num Loops` / `Num Waits`.

   For example, the following output snippet shows spin locks with large numbers of waits and loops:

```
Spin locks with waits:
Num Waits Num Loops Avg Loop/Wait Name
332480    1568908   4.72  fast mutex, 3:bf[1234] 0x2d00008  0x1028a0d8000
39722     498769    12.56 mutex lock, name = log
20761     101831    4.90  fast mutex, 7:bf[62] 0x1300003  0x109da128000
14818     77680     5.24  mutex lock, name = MGM mutex
6523      34350     5.27  fast mutex, 3:bf[362] 0x20008e   0x10289a08000
```

3. Query **sysmaster:systabnames** with the hexadecimal representation of the part number shown in the onstat -g spi output. If the `tabname` represents an index name, the index is a forest of trees candidate.

   For example, run this query:

```
echo "select tabname, hex(partnum) from systabnames
 where hex(partnum) =  '0x02d00008'" | dbaccess sysmaster -

tabname       daily_market_idx
(expression)  0x02d00008

$ echo 'select tabname, hex(partnum) from systabnames'
 where hex(partnum) = 0x01300003 | dbaccess sysmaster -

tabname       trade_history_idx
(expression)  0x01300003

$ echo 'select tabname, hex(partnum) from systabnames'
 where hex(partnum) = 0x0020008E | dbaccess sysmaster -

tabname       trade_request_idx2
(expression)  0x0020008E
```

   Result

Related information

Forest of trees indexes on page 208

onstat -g spi command: Print spin locks with long spins on page

Creating a forest of trees index

You use the CREATE INDEX statement with the HASH ON clause to create a forest of trees index.

Before you begin

Prerequisite: Determine whether you need a forest of trees index to reduce performance bottlenecks and contention or to reduce the number of levels in a traditional B-Tree index.

About this task

To create a forest of trees index:

1. Choose the columns for the index and determine the number of subtrees to create.
2. Create the index by using the CREATE INDEX statement with the HASH ON clause:

   For example, the following command creates a forest of trees index with 100 subtrees (buckets) on the C1 column:

   ```
   CREATE INDEX fotidx ON tab(c1) hash on (c1) with 100 buckets
   ```

   After you create a forest of trees index, it is enabled.

What to do next

You can monitor onstat -g spi command output to verify that root node contention no longer occurs. If you identify performance bottlenecks that are caused by highly contended spin locks, you can rebuild the forest of trees index with more buckets.

Related information

Forest of trees indexes on page 208

CREATE INDEX statement on page

HASH ON clause on page

Disabling and enabling a forest of trees index

You can use the INDEXES DISABLED option of the SET Database Object Mode statement of SQL to disable a forest of trees index, if you want the server to stop updating the index and to stop using it during queries. After you are ready to put the index into production, you can use the INDEXES ENABLED option to re-enable it.

Before you begin

About this task

To disable a forest of trees index:

Run the SET INDEXES DISABLED statement of SQL.

Example

For example, for an index named `fotidx`, specify:

```
SET INDEXES fotidx DISABLED;
```

What to do next

You can re-enable a disabled forest of trees index, for example, by specifying:

```
SET INDEXES fotidx ENABLED;
```

Related information

Forest of trees indexes on page 208

Performing a range scan on a forest of trees index

While you cannot perform range scans directly on the HASH ON columns of a forest of trees index, you can perform range scans on the columns that are not listed in the HASH ON column list. To perform range scans on columns that are listed in HASH ON column list, you must create an additional B-tree index that contains the appropriate column list for the range scan.

About this task

To create indexes for range scans:

1. Create a forest of trees index with at least one column that is not hashed.

   For example, specify:

   ```
   CREATE INDEX idx1 on tab(c1,c2) HASH ON (c1) with 100 buckets;
   ```

   You can perform a range scan directly on column `c2`, but not on column `c1`, which is listed in HASH ON column list.

2. For range scans on the columns listed in HASH ON column list, create an additional B-tree index that contains the appropriate column list for the range scan. This additional B-tree index might have the same column list as the forest of trees index, plus or minus a column.

   For example, specify:

   ```
   CREATE INDEX idx2 on tab(c1, c2, c3);
   ```

Related information

CREATE INDEX statement on page

HASH ON clause on page

Determining if you are using a forest of trees index

You can determine whether an index is a forest of trees index by viewing SET EXPLAIN output. A forest of trees index has `FOT` in the `Index Name` field of the output.

Example

In the following example of partial SET EXPLAIN output, `informix.fot_idx` is the name of a forest of trees index.

```
Estimated Cost: 1
Estimated # of Rows Returned: 1

  1) informix.t: INDEX PATH

    (1) Index Name: informix.fot_idx (FOT)
        Index Keys: c1 c2   (Serial, fragments: ALL)
        Lower Index Filter: informix.t.c1 = 1
```

Related information

Forest of trees indexes on page 208

Finding the number of hashed columns and subtrees in a forest of trees index

You can look up the number of hashed columns and subtrees in a forest of trees index by viewing information in the **sysindices** table for the database containing tables that have forest of trees indexes.

About this task

To view information about a forest of trees index:

1. Query the **sysindices** table for the index.
2. Go to the row containing the forest of trees index and view information in the `nhashcols` and `nbuckets` columns.

Creating and dropping an index in an online environment

You can use the CREATE INDEX ONLINE and DROP INDEX ONLINE statements to create and drop an index in an online environment, when the database and its associated tables are continuously available.

The CREATE INDEX ONLINE statement enables you to create an index without having an exclusive lock placed over the table during the duration of the index build. You can use the CREATE INDEX ONLINE statement even when reads or updates are occurring on the table. This means index creation can begin immediately.

When you create an index online, the database server logs the operation with a flag, so data recovery and restore operations can recreate the index.

When you create an index online, you can use the ONLIDX_MAXMEM configuration parameter to limit the amount of memory that is allocated to the *preimage* log pool and to the *updator* log pool in shared memory. You might want to do this if you plan to complete other operations on a table column while executing the CREATE INDEX ONLINE statement on the column. For more information about this parameter, see Limiting memory allocation while creating indexes online on page 225.

The DROP INDEX ONLINE statement enables you to drop indexes even when Dirty Read is the transaction isolation level.

The advantages of creating indexes using the CREATE INDEX ONLINE statement are:

- If a new index is needed to improve the performance of queries on a table, you can immediately create the index without a lock placed over the table.
- The database server can create an index while a table is being updated.
- The table is available for the duration of the index build.
- The query optimizer can establish better query plans, since the optimizer can update statistics in unlocked tables.

The advantages of dropping indexes using the DROP INDEX ONLINE statement are:

- You can drop an inefficient index without disturbing ongoing queries that are using that index.
- After the index is flagged, the query optimizer will not use the index for new SELECT operations on tables.

If you initiate a DROP INDEX ONLINE statement for a table that is being updated, the operation does not occur until after the table update is completed. After you issue the DROP INDEX ONLINE statement, no one can reference the index, but concurrent operations can use the index until the operations terminate. The database server waits to drop the index until all users have finished accessing the index.

An example of creating an index in an online environment is:

```
CREATE INDEX idx_1 ON table1(col1) ONLINE
```

An example of dropping an index in an online environment is:

```
DROP INDEX idx_1 ONLINE
```

For more information about the CREATE INDEX ONLINE and DROP INDEX ONLINE statements, see the *HCL OneDB™ Guide to SQL: Syntax*.

When you cannot create or drop indexes online

You cannot use the CREATE INDEX ONLINE and the DROP INDEX ONLINE statements under certain circumstances.

You cannot use the CREATE INDEX ONLINE statement:

- To create an index at the same time that a table is being altered
- To create a clustered index

- To create a Virtual-Index Interface (VII) /R-tree index
- To create a functional index
- To create an index that is partitioned by an interval fragmentation strategy
- To create an index on a table that is partitioned by an interval fragmentation strategy

You cannot use the DROP INDEX ONLINE statement:

- To drop a Virtual-Index Interface (VII) /R-tree index
- To drop a clustered index

Creating attached indexes in an online environment

You can create attached indexes using the CREATE INDEX ONLINE statement, but the statement only operates when Dirty Read is the transaction isolation level.

The index creation takes an exclusive lock on the table and waits for all other concurrent processes scanning the table to quit using the index partitions before creating the attached index. If the table is being read or updated, the CREATE INDEX ONLINE statement waits for the exclusive lock for the duration of the lock mode setting.

Limiting memory allocation while creating indexes online

The ONLIDX_MAXMEM configuration parameter limits the amount of memory that is allocated to a single *preimage* pool and a single *updator* log pool.

The preimage and updator log pools, **pimage_<partnum>** and **ulog_<partnum>,** are shared memory pools that are created when a CREATE INDEX ONLINE statement is executed. The pools are freed when the execution of the statement is completed.

The default value of the ONLIDX_MAXMEM configuration parameter is `5120` kilobytes. The minimum value that you can specify is `16` kilobytes; the maximum value is `4294967295` kilobytes.

You can set the ONLIDX_MAXMEM configuration parameter before starting the database server, or you can change it dynamically through the **onmode -wf** and **onmode -wm** commands.

Improving performance for index builds

You can improve performance for index builds by adjusting the PDQ priority and by allocating enough memory and temporary space for the entire index.

About this task

Whenever possible, the database server uses parallel processing to improve the response time of index builds. The number of parallel processes is based on the number of fragments in the index and the value of the **PSORT_NPROCS** environment variable. The database server builds the index with parallel processing even when the value of PDQ priority is `0`.

You can often improve the performance of an index build by taking the following steps:

1. Set PDQ priority to a value greater than `0` to obtain more memory than the default 128 kilobytes.

   When you set PDQ priority to greater than `0`, the index build can take advantage of the additional memory for parallel processing.

   To set PDQ priority, use either the **PDQPRIORITY** environment variable or the SET PDQPRIORITY statement in SQL.

2. Do not set the **PSORT_NPROCS** environment variable.
   If you have a computer with multiple CPUs, the database server uses two threads per sort when it sorts index keys and **PSORT_NPROCS** is not set. The number of sorts depends on the number of fragments in the index, the number of keys, the key size, and the values of the PDQ memory configuration parameters.
3. Allocate enough memory and temporary space to build the entire index.

   a. Estimate the amount of virtual shared memory that the database server might need for sorting.

      For more information, see Estimating memory needed for sorting on page 226.

   b. Specify more memory with the DS_TOTAL_MEMORY and DS_MAX_QUERIES configuration parameters.

   c. If not enough memory is available, estimate the amount of temporary space needed for an entire index build.

      For more information, see Estimating temporary space for index builds on page 227.

   d. Use the **onspaces -t** utility to create large temporary dbspaces and specify them in the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable.

      For information about how to optimize temporary dbspaces, see Configure dbspaces for temporary tables and sort files on page 114.

Estimating memory needed for sorting

To calculate the amount of virtual shared memory that the database server might need for sorting, estimate the maximum number of sorts that might occur concurrently and multiply that number by the average number of rows and the average row size.

For example, if you estimate that 30 sorts could occur concurrently, the average row size is 200 bytes, and the average number of rows in a table is 400, you can estimate the amount of shared memory that the database server needs for sorting as follows:

```
30 sorts * 200 bytes * 400 rows = 2,400,000 bytes
```

You can use the DS_NONPDQ_QUERY_MEM configuration parameter to configure the amount sort memory available for non-PDQ queries.

Important: You can only use this parameter if the PDQ priority is set to zero. Its setting has no effect if the PDQ priority is greater than zero.

The minimum and default value of DS_NONPDQ_QUERY_MEM is 128 kilobytes. The maximum supported value is 25 percent of DS_TOTAL_MEMORY. For more information, see Configuring memory for queries with hash joins, aggregates, and other memory-intensive elements on page 414.

If the PDQ priority is greater than `0`, the maximum amount of shared memory that the database server allocates for a sort is controlled by the memory grant manager (MGM). The MGM uses the settings of PDQ priority and the following configuration parameters to determine how much memory to grant for the sort:

- DS_TOTAL_MEMORY
- DS_MAX_QUERIES
- MAX_PDQPRIORITY

For more information about allocating memory for parallel processing, see The allocation of resources for parallel database queries on page 352.

Estimating temporary space for index builds

You can estimate the number of bytes of temporary space needed for an entire index build.

About this task

To estimate the amount of temporary space needed for an index build, perform the following steps:

1. Add the total widths of the indexed columns or returned values from user-defined functions. This value is referred to as *colsize*.
2. Estimate the size of a typical item to sort with one of the following formulas, depending on whether the index is attached or not:
    a. For a nonfragmented table and a fragmented table with an index created without an explicit fragmentation strategy, use the following formula:

    ```
    sizeof_sort_item = keysize + 4
    ```

    b. For fragmented tables with the index explicitly fragmented, use the following formula:

    ```
    sizeof_sort_item =
    keysize + 8
    ```

3. Estimate the number of bytes needed to sort with the following formula:

    ```
    temp_bytes = 2 * (rows *
    sizeof_sort_item)
    ```

This formula uses the factor `2` because everything is stored twice when intermediate sort runs use temporary space. Intermediate sort runs occur when not enough memory exists to perform the entire sort in memory.

The value for *rows* is the total number of rows that you expect to be in the table.

Storing multiple index fragments in a single dbspace

You can store multiple fragments of the same index in a single dbspace, reducing the total number of dbspaces needed for a fragmented table. You must specify a name for each fragment that you want to store in the same dbspace. Storing multiple index fragments in a single dbspace simplifies the management of dbspaces.

You can also use this feature to improve query performance over storing each fragment in a different dbspace when a dbspace is located on a faster device.

For more information, see information about managing partitions in the *HCL OneDB™ Administrator's Guide*.

Improving performance for index checks

The **oncheck** utility provides better concurrency for tables that use row locking. When a table uses page locking, **oncheck** places a shared lock on the table when it performs index checks. Shared locks do not allow other users to perform updates, inserts, or deletes on the table while **oncheck** checks or prints the index information.

If the table uses page locking, the database server returns the following message if you run **oncheck** without the **-x** option:

```
WARNING: index check requires a s-lock on stable whose
lock level is page.
```

For detailed information about **oncheck** locking, see the *HCL OneDB™ Administrator's Reference*.

The following summary describes locking performed during index checks:

- By default, the database server does not place a shared lock on the table when you check an index with the **oncheck -ci, -cI, -pk, -pK, -pl,** or **-pL** options unless the table uses page locking. When **oncheck** checks indexes for a table with page locking, it places a shared lock on the table, so no other users can perform updates, inserts, or deletes until the check has completed.
- By not placing a shared lock on tables using row locks during index checks, the **oncheck** utility cannot be as accurate in the index check. For absolute assurance of a complete index check, execute **oncheck** with the **-x** option. With the **-x** option, **oncheck** places a shared lock on the table, and no other users can perform updates, inserts, or deletes until the check completes.

You can query the **systables** system catalog table to see the current lock level of the table, as the following sample SQL statement shows:

```
SELECT locklevel FROM systables
   WHERE tabname = "customer"
```

If you do not see a value of `R` (for row) in the **locklevel** column, you can modify the lock level, as the following sample SQL statement shows:

```
ALTER TABLE tab1 LOCK MODE (ROW);
```

Row locking might add other side effects, such as an overall increase in lock usage. For more information about locking levels, see Locking on page 240.

Indexes on user-defined data types

You can define your own data types and the functions that operate on these data types. You can define indexes on some kinds of user-defined data types.

DataBlade® modules also provide extended data types and functions to the database server.

You can define indexes on the following kinds of user-defined data types:

- Opaque data types

  An *opaque data type* is a fundamental data type that you can use to define columns in the same way you use built-in types. An opaque data type stores a single value and cannot be divided into components by the database server. For information about creating opaque data types, see the CREATE OPAQUE TYPE statement in the *HCL OneDB™ Guide to SQL: Syntax* and *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*. For more information about the data types and functions that each DataBlade® module provides, see the user guide of each DataBlade® module.

- Distinct data types

  A *distinct data type* has the same representation as an existing opaque or built-in data type but is different from these types. For information about distinct data types, see the *HCL OneDB™ Guide to SQL: Reference* and the CREATE DISTINCT TYPE statement in the *HCL OneDB™ Guide to SQL: Syntax*.

For more information about data types, see the *HCL OneDB™ Guide to SQL: Reference*.

Defining indexes for user-defined data types

As with built-in data types, you might improve the response time for a query when you define indexes for new data types.

The response time for a query might improve when HCL OneDB™ uses an index for:

- Columns used to join two tables
- Columns that are filters for a query
- Columns in an ORDER BY or GROUP BY clause
- Results of functions that are filters for a query

For more information about when the query performance can improve with an index on a built-in data type, see Improve performance by adding or removing indexes on page 388.

HCL OneDB™ and DataBlade® modules provide a variety of different types of indexes (also referred to as *secondary-access methods*). A secondary-access method is a set of database server functions that build, access, and manipulate an index structure. These functions encapsulate index operations, such as how to scan, insert, delete, or update nodes in an index.

To create an index on a user-defined data type, you can use any of the following secondary-access methods:

- Generic B-tree index

  A B-tree index is good for a query that retrieves a range of data values. For more information, see B-tree secondary-access method on page 230.

- R-tree index

  An R-tree index is good for searches on multidimensional data. For more information, see the *R-Tree Index User's Guide*.

- Secondary-access methods that a DataBlade® module provides for a new data type

  A DataBlade® module that supports a certain type of data can also provide a new index for that new data type. For more information, see Using an index that a DataBlade module provides on page 235.

You can create a functional index on the resulting values of a user-defined function on one or more columns. For more information, see Using a functional index on page 233.

After you choose the desired index type, you might also need to extend an operator class for the secondary-access method. For more information about how to extend operator classes, see the *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

B-tree secondary-access method

HCL OneDB™ provides the *generic B-tree index* for columns in database tables. In traditional relational database systems, the B-tree access method handles only built-in data types and therefore it can only compare two keys of built-in data types. The generic B-tree index is an extended version of a B-tree that HCL OneDB™ provides to support user-defined data types.

Tip: For more information about the structure of a B-tree index and how to estimate the size of a B-tree index, see Estimating index pages on page 209**.**

HCL OneDB™ uses the generic B-tree as the built-in secondary-access method. This built-in secondary-access method is registered in the **sysams** system catalog table with the name **btree**. When you use the CREATE INDEX statement (without the USING clause) to create an index, the database server creates a generic B-tree index. For more information, see the CREATE INDEX statement in the *HCL OneDB™ Guide to SQL: Syntax*.

Tip: *HCL OneDB™* also defines another secondary-access method, the R-tree index. For more information about how to use an R-tree index, see the *R-Tree Index User's Guide*.

Uses for a B-tree index

A B-tree index is good for a query that retrieves a range of data values. If the data to be indexed has a logical sequence to which the concepts of *less than*, *greater than*, and *equal* apply, the generic B-tree index is a useful way to index your data.

Initially, the generic B-tree index supports the relational operators (<,<=,=,>=,>) on all built-in data types and orders the data in lexicographical sequence.

The optimizer considers whether to use the B-tree index to execute a query if you define a generic B-tree index on:

- Columns used to join two tables
- Columns that are filters for a query
- Columns in an ORDER BY or GROUP BY clause
- Results of functions that are filters for a query

Extending a generic B-tree index

Initially, the generic B-tree can index data that is one of the built-in data types, and it orders the data in lexicographical sequence. However, you can extend a generic B-tree for some other data types.

You can extend a generic B-tree to support columns and functions on the following data types:

- *User-defined data types* (opaque and distinct data types) that you want the B-tree index to support

  In this case, you must extend the default operator class of the generic B-tree index.
- *Built-in data types* that you want to order in a different sequence from the lexicographical sequence that the generic B-tree index uses

  In this case, you must define a different operator class from the default generic B-tree index.

An *operator class* is the set of functions (operators) that are associated with a nontraditional B-tree index. For more details on operator classes, see Choosing operator classes for indexes on page 235.

Identifying the available access methods

To supplement the built-in B-tree secondary-access method that HCL OneDB™ provides, your enterprise might have installed DataBlade® modules that implement additional secondary-access methods. If additional access methods exist, they are defined in the **sysams** system catalog table. You can query the **sysams** system catalog to determine if additional access methods are available.

To identify the secondary-access methods that are available for your database, query the **sysams** system catalog table with the following SELECT statement:

```
SELECT am_id, am_owner, am_name, am_type FROM sysams
   WHERE am_type = 'S';
```

An 'S' value in the **am_type** column identifies the access method as a secondary-access method. This query returns the following information:

- The **am_id** and **am_name** columns identify the secondary-access method.
- The **am_owner** column identifies the owner of the access method.

In an ANSI-compliant database, the access-method name must be unique within the name space of the user. The access-method name always begins with the owner in the format **am_owner.am_name**.

By default, HCL OneDB™ provides the following definitions in the **sysams** system catalog table for two secondary-access methods, **btree** and **rtree**.

Access Method	am_id Column	am_name Column	am_owner Column
Generic B-tree	1	btree	'informix'
R-tree	2	rtree	'informix'

Important: The **sysams** system catalog table does not contain a row for the built-in primary access method. This primary access method is internal to HCL OneDB™ and does not require a definition in **sysams**. However, the built-in primary access method is always available for use.

If you find additional rows in the **sysams** system catalog table (rows with **am_id** values greater than 2), the database supports additional user-defined access methods. Check the value in the **am_type** column to determine whether a user-defined access method is a primary- or secondary-access method.

For more information about the columns of the **sysams** system catalog table, see the *HCL OneDB™ Guide to SQL: Reference*. For information about how to determine the operator classes that are available in your database, see Identifying the available operator classes on page 238.

User-defined secondary-access methods

If the concepts of *less than*, *greater than*, and *equal* do not apply to the data to be indexed, you might consider using a *user-defined secondary-access method* instead of the built-in secondary-access method, which is a B-tree index. You can use a user-defined secondary-access method to access other indexing structures, such as an R-tree index.

If your database supports a user-defined secondary-access method, you can specify that the database server uses this access method when it accesses a particular index. For information about how to determine the secondary-access methods that your database defines, see Identifying the available access methods on page 231.

To choose a user-defined secondary-access method, use the USING clause of the CREATE INDEX statement. The USING clause specifies the name of the secondary-access method to use for the index you create. This name must be listed in the **am_name** column of the **sysams** system catalog table and must be a secondary-access method (the **am_type** column of **sysams** is `'S'`).

The secondary-access method that you specify in the USING clause of CREATE INDEX must already be defined in the **sysams** system catalog. If the secondary-access method has not yet been defined, the CREATE INDEX statement fails.

When you omit the USING clause from the CREATE INDEX statement, the database server uses B-tree indexes as the secondary-access method. For more information, see the CREATE INDEX statement in the *HCL OneDB™ Guide to SQL: Syntax*.

R-tree indexes

HCL OneDB™ supports the *R-tree index* for columns that contain spatial data such as maps and diagrams. An R-tree index uses a tree structure whose nodes store pointers to lower-level nodes.

At the leaves of the R-tree are a collection of data pages that store *n*-dimensional shapes. For more information about the structure of an R-tree index and how to estimate the size of an R-tree index, see the *R-Tree Index User's Guide*.

Using a functional index

You can create a column index on the actual values in one or more columns. You can also create a functional index on the values of one or more columns that a user-defined function returns from arguments.

Important: The database server imposes the following restrictions on the user-defined routines (UDRs) on which a functional index is defined:

- The arguments cannot be column values of a collection data type.
- The function cannot return a large object (including built-in types BLOB, BYTE, CLOB, and TEXT).
- The function cannot be a VARIANT function.
- The function cannot include any DML statement of SQL.
- The function must be a UDR, rather than a built-in function. However, you can create an SPL wrapper that calls and returns the value from a built-in function of SQL.

In addition, do not create functional indexes using any routine that calls the built-in DECRYPT_BINARY( ) or DECRYPT_CHAR( ) functions, which can display encrypted data values in plain text. (Do not attempt to use data values in any encrypted column as an index key.)

To decide whether to use a column index or functional index, determine whether a column index is the right choice for the data that you want to index. An index on a column of some data types might not be useful for typical queries. For example, the following query asks how many images are dark:

```
SELECT COUNT(*) FROM photos WHERE
darkness(picture) > 0.5
```

An index on the **picture** data itself does not improve the query performance. The concepts of *less than*, *greater than*, and *equal* are not particularly meaningful when applied to an image data type. Instead, a functional index that uses the **darkness()** function can improve performance. You might also have a user-defined function that runs frequently enough that performance improves when you create an index on its values.

Related information

Managing indexes on page 213

What is a functional index?

A functional index can be a B-tree index, an R-tree index, or a user-defined index type that a DataBlade® module provides.

When you create a functional index, the database server computes the values of the user-defined function and stores them as key values in the index. When a change in the table data causes a change in one of the values of an index key, the database server automatically updates the functional index.

You can use a functional index for functions that return values of both user-defined data types (opaque and distinct) and built-in data types. However, you cannot define a functional index if the function returns a simple-large-object data type (TEXT or BYTE).

For more information about the types of indexes, see Defining indexes for user-defined data types on page 229. For information about space requirements for functional indexes, see Estimating index pages on page 209.

Related information

Types of indexes on page 206

When is a functional index used?

The optimizer considers whether to use a functional index to access the results of functions that are in a SELECT clause or are in the filters in the WHERE clause.

Creating a functional index

You can build a functional index on a user-defined function. The user-defined function can be either an external function or an SPL function.

About this task

To build a functional index on a user-defined function:

1. Write the code for the user-defined function if it is an external function.
2. Register the user-defined function in the database with the CREATE FUNCTION statement.
3. Build the functional index with the CREATE INDEX statement.

Results

For example, to create a functional index on the darkness() function:

1. Write the code for the user-defined **darkness()** function that operates on the data type and returns a decimal value.
2. Register the user-defined function in the database with the CREATE FUNCTION statement:

```
CREATE FUNCTION darkness(im image)
RETURNS decimal
EXTERNAL NAME '/lib/image.so'
LANGUAGE C NOT VARIANT
```

In this example, you can use the default operator class for the functional index because the return value of the **darkness()** function is a built-in data type, DECIMAL.

3. Build the functional index with the CREATE INDEX statement.

```
CREATE TABLE photos
(
    name char(20),
    picture image
...
);
CREATE INDEX dark_ix ON photos (darkness(picture));
```

In this example, assume that the user-defined data type of **image** has already been defined in the database.

The optimizer can now consider the functional index when you specify the **darkness()** function as a filter in the query:

```
SELECT count(*) FROM photos WHERE
darkness(picture) > 0.5
```

You can also create a composite index with user-defined functions. For more information, see Use composite indexes on page 388.

Warning: Do not create a functional index using either the DECRYPT_BINARY() or the DECRYPT_CHAR() function. These functions store plain text data in the database, defeating the purpose of encryption. For more information about encryption, see the *HCL OneDB™ Administrator's Guide*.

Using an index that a DataBlade® module provides

DataBlade® modules can provide new data types that users can access. A DataBlade® module can also provide a new index for the new data type.

For more information about the types of data and functions that each DataBlade® module provides, see the user guide for the DataBlade® module. For information about how to determine the types of indexes available in your database, see Identifying the available access methods on page 231.

Choosing operator classes for indexes

For most situations, use the default operators that are defined for a secondary-access method. However, when you want to order the data in a different sequence or provide index support for a user-defined data type, you must extend an operator class.

For more information about how to extend an operator class, see *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

Operator classes

An *operator class* is a set of function names that is associated with a secondary-access method. These functions allow the secondary-access method to store and search for values of a particular data type.

The query optimizer for the database server uses an operator class to determine if an index can process the query with the least cost. An operator class indicates two things to the query optimizer:

- Which functions that appear in an SQL statement can be evaluated with a given index

  These functions are called the *strategy functions* for the operator class.
- Which functions the index uses to evaluate the strategy functions

  These functions are called the *support functions* for the operator class.

With the information that the operator class provides, the query optimizer can determine whether a given index is applicable to the query. The query optimizer can consider whether to use the index for the given query when the following conditions are true:

- An index exists on the particular column or columns in the query.
- For the index that exists, the operation on the column or columns in the query matches one of the strategy functions in the operator class associated with the index.

The query optimizer reviews the available indexes for the table or tables and matches the index keys with the column specified in the query filter. If the column in the filter matches an index key, and the function in the filter is one of the strategy functions of the operator class, the optimizer includes the index when it determines which query plan has the lowest execution cost. In this manner, the optimizer can determine which index can process the query with the least cost.

HCL OneDB™ stores information about operator classes in the **sysopclasses** system catalog table.

Strategy and support functions of a secondary access method

HCL OneDB™ uses the *strategy functions* of a secondary-access method to help the query optimizer determine whether a specific index is applicable to a specific operation on a data type.

If an index exists and the operator in the filter matches one of the strategy functions in the operator class, the optimizer considers whether to use the index for the query.

HCL OneDB™ uses the *support functions* of a secondary-access method to build and access the index. These functions are not called directly by end users. When an operator in the query filter matches one of the strategy functions, the secondary-access method uses the support functions to traverse the index and obtain the results. Identification of the actual support functions is left to the secondary-access method.

Default operator classes

Each secondary-access method has a *default operator class* associated with it. By default, the CREATE INDEX statement associates the default operator class with an index.

For example, the following CREATE INDEX statement creates a B-tree index on the **postalcode** column and automatically associates the default B-tree operator class with this column:

```
CREATE INDEX postal_ix ON customer(postalcode)
```

For more information about how to specify a new default operator class for an index, see User-defined operator classes on page 239.

Built-in B-tree operator class

The built-in secondary-access method (the generic B-tree) has a default operator class called **btree_ops**, which is defined in the **sysopclasses** system catalog table.

By default, the CREATE INDEX statement associates the **btree_ops** operator class with it when you create a B-tree index. For example, the following CREATE INDEX statement creates a generic B-tree index on the **order_date** column of the **orders** table and associates with this index the default operator class for the B-tree secondary-access method:

```
CREATE INDEX orddate_ix ON orders (order_date)
```

HCL OneDB™ uses the **btree_ops** operator class to specify:

- The strategy functions to tell the query optimizer which filters in a query can use a B-tree index
- The support function to build and search the B-tree index

B-tree strategy functions

The **btree_ops** operator class defines the names of strategy functions for the **btree** access method.

The strategy functions that the **btree_ops** operator class defines are:

- **lessthan** (<)
- **lessthanorequal** (<=)
- **equal** (=)
- **greaterthanorequal** (>=)
- **greaterthan** (>)

These strategy functions are all *operator functions*. That is, each function is associated with an operator symbol; in this case, with a relational-operator symbol. For more information about relational-operator functions, see the *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

When the query optimizer examines a query that contains a column, it checks to see if this column has a B-tree index defined on it. If such an index exists *and* if the query contains one of the relational operators that the **btree_ops** operator class supports, the optimizer can choose a B-tree index to execute the query.

B-tree support function

The **btree_ops** operator class has one support function, a comparison function called **compare()**. The **btree_ops** operator class has one support function, a comparison function called **compare()**.

The **compare()** function is a user-defined function that returns an integer value to indicate whether its first argument is equal to, less than, or greater than its second argument, as follows:

- A value of `0` when the first argument is *equal to* the second argument
- A value less than `0` when the first argument is *less than* the second argument
- A value greater than `0` when the first argument is *greater than* the second argument

The B-tree secondary-access method uses the **compare()** function to traverse the nodes of the generic B-tree index. To search for data values in a generic B-tree index, the secondary-access method uses the **compare()** function to compare the key value in the query to the key value in an index node. The result of the comparison determines if the secondary-access method needs to search the next-lower level of the index or if the key resides in the current node.

The generic B-tree access method also uses the **compare()** function to perform the following tasks for generic B-tree indexes:

- Sort the keys before the index is built
- Determine the linear order of keys in a generic B-tree index
- Evaluate the relational operators
- Search for data values in an index

The database server uses the **compare()** function to evaluate comparisons in the SELECT statement. To provide support for these comparisons for opaque data types, you must write the **compare()** function. For more information, see the *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

The database server also uses the **compare()** function when it uses a B-tree index to process an ORDER BY clause in a SELECT statement. However, the optimizer does not use the index to perform an ORDER BY operation if the index does not use the btree-ops operator class.

Identifying the available operator classes

You can identify the operator classes that are available for your database by querying the **sysopclasses** system catalog table.

The database server provides the default operator class for the built-in secondary-access method, the generic B-tree index. In addition, your environment might have installed DataBlade® modules that implement other operator classes. All operator classes are defined in the **sysopclasses** system catalog table.

To identify the operator classes that are available for your database, query the **sysopclasses** system catalog table with the following SELECT statement:

```
SELECT opclassid, opclassname, amid, am_name
FROM sysopclasses, sysams
WHERE sysopclasses.amid = sysams.am_id
```

This query returns the following information:

- The **opclassid** and **opclassname** columns identify the operator class.
- The **am_id** and **am_name** columns identify the associated secondary-access methods.

By default, the database server provides the following definitions in the **sysopclasses** system catalog table for two operator classes, **btree_ops** and **rtree_ops**.

Access Method	opclassid Column	opclassname Column	amid Column	am_name Column
Generic B-tree	1	btree_ops	1	btree
R-tree	2	rtree_ops	2	rtree

If you find additional rows in the **sysopclasses** system catalog table (rows with **opclassid** values greater than 2), your database supports user-defined operator classes. Check the value in the **amid** column to determine the secondary-access methods to which the operator class belongs.

The **am_defopclass** column in the **sysams** system catalog table stores the operator-class identifier for the default operator class of a secondary-access method. To determine the default operator class for a given secondary-access method, you can run the following query:

```
SELECT am_id, am_name, am_defopclass, opclass_name
FROM sysams, sysopclasses
WHERE sysams.am_defopclass = sysopclasses.opclassid
```

By default, the database server provides the following default operator classes.

Access Method	am_id Column	am_name Column	am_defopclass Column	opclass_name Column
Generic B-tree	1	btree	1	btree_ops
R-tree	2	rtree	2	rtree_ops

For more information about the columns of the **sysopclasses** and **sysams** system catalog tables, see the *HCL OneDB™ Guide to SQL: Reference*. For information about how to determine the access methods that are available in your database, see Identifying the available access methods on page 231.

User-defined operator classes

The CREATE INDEX statement specifies the operator class to use for each component of an index. If you do not specify an operator class, the CREATE INDEX statement uses the default operator class for the secondary-access method that you create. You can use a user-defined operator class for components of an index.

To specify a user-defined operator class for a particular component of an index, you can:

- Use a user-defined operator class that your database already defines.
- Use an R-tree operator class, if your database defined the R-tree secondary-access method. For more information about R-trees, see the *R-Tree Index User's Guide*.

If your database supports multiple-operator classes for the secondary-access method that you want to use, you can specify which operator classes the database server is to use for a particular index. For information on how to determine the operator classes that your database defines, see Identifying the available operator classes on page 238.

Each part of a composite index can specify a different operator class. You choose the operator classes when you create the index. In the CREATE INDEX statement, you specify the name of the operator class to use after each column or function name in the index-key specification. Each name must be listed in the **opclassname** column of the **sysopclasses** system catalog table and must be associated with the secondary-access method that the index uses.

For example, if your database defines the **abs_btree_ops** secondary-access method to define a new sort order, the following CREATE INDEX statement specifies that the **table1** table associates the **abs_btree_ops** operator class with the **col1_ix** B-tree index:

```
CREATE INDEX col1_ix ON table1(col1 abs_btree_ops)
```

The operator class that you specify in the CREATE INDEX statement must already be defined in the **sysopclasses** system catalog with the CREATE OPCLASS statement. If the operator class has not yet been defined, the CREATE INDEX statement fails. For information about how to create an operator class, see *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

Locking

The database server uses locks, which can affect concurrency and performance. You can monitor and administer locks.

Locks

A *lock* is a software mechanism that you can set to prevent others from using a resource. You can place a lock on a single row or key, a page of data or index keys, a whole table, or an entire database.

Additional types of locks are available for smart large objects. For more information, see Locks for smart large objects on page 255.

The maximum number of rows or pages locked in a single transaction is controlled by the total number of locks configured. The number of tables in which those rows or pages are locked is not explicitly controlled.

Locking granularity

The level and type of information that the lock protects is called *locking granularity*. Locking granularity affects performance.

When a user cannot access a row or key, the user can wait for another user to unlock the row or key. If a user locks an entire page, a higher probability exists that more users will wait for a row in the page.

The ability of more than one user to access a set of rows is called *concurrency*. The goal of the database administrator is to increase concurrency to increase total performance without sacrificing performance for an individual user.

Row and key locks

Row and key locks generally provide the best overall performance when you are updating a relatively small number of rows, because they increase concurrency. However, the database server incurs some overhead in obtaining a lock. For an operation that changes a large number of rows, obtaining one lock per row might not be cost effective.

For an operation that changes a large number of rows, consider Page locks on page 241.

The default locking mode is page-locking. If you want row or key locks, you must create the table with row locking on or alter the table.

The following example shows how to create a table with row locking on:

```
CREATE TABLE customer(customer_num serial, lname char(20)...)
   LOCK MODE ROW;
```

The ALTER TABLE statement can also change the lock mode.

When the lock mode is ROW and you insert or update a row, the database server creates a row lock. In some cases, you place a row lock by simply reading the row with a SELECT statement.

When the lock mode is ROW and you insert, update, or delete a key (performed automatically when you insert, update, or delete a row), the database server creates a lock on the key in the index.

Key-value locks

When a user deletes a row within a transaction, the row cannot be locked because it does not exist. However, the database server must somehow record that a row existed until the end of the transaction. The database server uses *key-value locking* to lock the deleted row.

When the database server deletes a row, key values in the indexes for the table are not removed immediately. Instead, each key value is marked as deleted, and a lock is placed on the key value.

Other users might encounter key values that are marked as deleted. The database server must determine whether a lock exists. If a lock exists, the delete has not been committed, and the database server sends a lock error back to the application (or it waits for the lock to be released if the user executed SET LOCK MODE TO WAIT).

One of the most important uses for key-value locking is to assure that a unique key remains unique through the end of the transaction that deleted it. Without this protection mechanism, user A might delete a unique key within a transaction, and user B might insert a row with the same key before the transaction commits. This scenario makes rollback by user A impossible. Key-value locking prevents user B from inserting the row until the end of user A's transaction.

Page locks

Page locking is the default mode when you create a table without the LOCK MODE clause. With page locking, instead of locking only the row, the database server locks the entire page that contains the row. If you update several rows on the same page, the database server uses only one lock for the page.

When you insert or update a row, the database server creates a page lock on the data page. In some cases, the database server creates a page lock when you simply read the row with a SELECT statement.

When you insert, update, or delete a key (performed automatically when you insert, update, or delete a row), the database server creates a lock on the page that contains the key in the index.

Important: A page lock on an index page can decrease concurrency more substantially than a page lock on a data page. Index pages are dense and hold a large number of keys. By locking an index page, you make a potentially large

number of keys unavailable to other users until you release the lock. Tables that use page locks cannot support the USELASTCOMMITTED concurrency feature, which is described in the Committed Read isolation on page 245 section.

Page locks are useful for tables in which the normal user changes a large number of rows at one time. For example, an orders table that holds orders that are commonly inserted and queried individually is not a good candidate for page locking. But a table that holds old orders and is updated nightly with all of the orders placed during the day might be a good candidate. In this case, the type of isolation level that you use to access the table is important. For more information, see Isolation level on page 244.

Table locks

In a data warehouse environment, it might be more appropriate for queries to acquire locks of larger granularity. For example, if a query accesses most of the rows in a table, its efficiency increases if it acquires a smaller number of table locks instead of many page or row locks.

The database server can place two types of table locks:

- Shared lock

  No other users can write to the table.
- Exclusive lock

  No other users can read from or write to the table.

Another important distinction between these two types of table locks is the actual number of locks placed:

- In shared mode, the database server places one shared lock on the table, which informs other users that no updates can be performed. In addition, the database server adds locks for every row updated, deleted, or inserted.
- In exclusive mode, the database server places only one exclusive lock on the table, no matter how many rows it updates. If you update most of the rows in the table, place an exclusive lock on the table.

Important: A table lock on a table can decrease update concurrency radically. Only one update transaction can access that table at any given time, and that update transaction locks out all other transactions. However, multiple read-only transactions can simultaneously access the table. This behavior is useful in a data warehouse environment where the data is loaded and then queried by multiple users.

You can switch a table back and forth between table-level locking and the other levels of locking. This ability to switch locking levels is useful when you use a table in a data warehouse mode during certain time periods but not in others.

A transaction tells the database server to use table-level locking for a table with the LOCK TABLE statement. The following example places an exclusive lock on the table:

```
LOCK TABLE tab1 IN EXCLUSIVE MODE;
```

The following example places a shared lock on the table:

```
LOCK TABLE tab1 IN SHARE MODE:
```

In some cases, the database server places its own table locks. For example, if the isolation level is Repeatable Read, and the database server must read a large portion of the table, it places a table lock automatically instead of setting row or page locks. The database server places a table lock on a table when it creates or drops an index.

Database locks

You can place a lock on the entire database when you open the database with the DATABASE statement. A database lock prevents read or update access by anyone but the current user.

The following statement opens and locks the sales database:

```
DATABASE sales EXCLUSIVE
```

Configuring the lock mode

When you create a table, the default lock mode is `page`. You can change the lock mode (and thus increase or decrease concurrency) when you create or alter tables or by setting the **IFX_DEF_TABLE_LOCKMODE** environment variable or the DEF_TABLE_LOCKMODE configuration parameter.

If you know that most of your applications might benefit from a lock mode of row, you can take one of the following actions:

- Use the LOCK MODE ROW clause in each CREATE TABLE statement or ALTER TABLE statement.
- Set the **IFX_DEF_TABLE_LOCKMODE** environment variable to ROW so that all tables you subsequently create within a session use ROW without the need to specify the lock mode in the CREATE TABLE statement or ALTER TABLE statement.
- Set the DEF_TABLE_LOCKMODE configuration parameter to ROW so that all tables subsequently created within the database server use ROW without the need to specify the lock mode in the CREATE TABLE statement or ALTER TABLE statement.

If you change the lock mode with the **IFX_DEF_TABLE_LOCKMODE** environment variable or DEF_TABLE_LOCKMODE configuration parameter, the lock mode of existing tables are not affected. Existing tables continue to use the lock mode with which they were defined at the time they were created.

In addition, if you previously changed the lock mode of a table to ROW, and subsequently execute an ALTER TABLE statement to alter some other characteristic of the table (such as add a column or change the extent size), you do not need to specify the lock mode. The lock mode remains at ROW and is not set to the default PAGE mode.

You can still override the lock mode of individual tables by specifying the LOCK MODE clause in the CREATE TABLE statement or ALTER TABLE statement.

The following list shows the order of precedence for the lock mode on a table:

- The system default is page locks. The database server uses this system default if you do not set the configuration parameter, do not set the environment variable, or do not specify the LOCK MODE clause in the SQL statements.
- If you set the DEF_TABLE_LOCKMODE configuration parameter, the database server uses this value when you do not set the environment variable, or do not specify the LOCK MODE clause in the SQL statements.
- If you set the **IFX_DEF_TABLE_LOCKMODE** environment variable, this value overrides the DEF_TABLE_LOCKMODE configuration parameter and system default. The database server uses this value when you do not specify the LOCK MODE clause in the SQL statements.
- If you specify the LOCK MODE clause in the CREATE TABLE statement or ALTER TABLE statement, this value overrides the **IFX_DEF_TABLE_LOCKMODE,** the DEF_TABLE_LOCKMODE configuration parameter and system default.

Setting the lock mode to wait

When an application process encounters a lock, the default behavior of the database server is to return an error. Instead, you can run an SQL statement to set the lock mode to wait. This specifies that an application process does not proceed until the lock is removed.

About this task

To suspend the current process until the lock releases, run the following SQL statement :

```
SET LOCK MODE TO WAIT;
```

You can also specify the maximum number of seconds that a process waits for a lock to be released before issuing an error. In the following example, the database server waits for 20 seconds before issuing an error:

```
SET LOCK MODE TO WAIT 20;
```

To return to the default behavior (no waiting for locks), execute the following statement:

```
SET LOCK MODE TO NOT WAIT;
```

Locks with the SELECT statement

The type and duration of locks that the database server places depend on the isolation level set in the application, the database mode (logging, nonlogging, or ANSI,) and on whether the SELECT statement is within an update cursor. These locks can affect overall performance because they affect concurrency.

Isolation level

The number and duration of locks placed on data during a SELECT statement depend on the level of isolation that the user sets. The type of isolation can affect overall performance because it affects concurrency.

Before you execute a SELECT statement, you can set the isolation level with the SET ISOLATION statement, which is part of the HCL OneDB™ extension to the ANSI SQL-92 standard, or with the ANSI/ISO-compliant SET TRANSACTION. The main differences between the two statements are that SET ISOLATION has an additional isolation level, Cursor Stability, and SET TRANSACTION cannot be executed more than once in a transaction as SET ISOLATION can. The SET ISOLATION statement is part of the HCL OneDB™ extension to the ANSI SQL-92 standard. The SET ISOLATION statement can change the enduring isolation level for the session

Dirty Read isolation

The Dirty Read isolation (or ANSI Read Uncommitted) level does not place any locks on any rows fetched during a SELECT statement. Dirty Read isolation is appropriate for static tables that are used for queries.

Use Dirty Read isolation with care if update activity occurs at the same time. With Dirty Read, the reader can read a row that has not been committed to the database and might be eliminated or changed during a rollback. For example, consider the following scenario:

```
User 1 starts a transaction.
User 1 inserts row A.
User 2 reads row A.
User 1 rolls back row A.
```

User 2 reads row A, which user 1 rolls back seconds later. In effect, user 2 read a row that was never committed to the database. Uncommitted data that is rolled back can be a problem in applications.

Because the database server does not check or place any locks for queries, Dirty Read isolation offers the best performance of all isolation levels. However, because of potential problems with uncommitted data that is rolled back, use Dirty Read isolation with care.

Because problems with uncommitted data that is rolled back are an issue only with transactions, databases that do not have transaction (and hence do not allow transactions) use Dirty Read as a default isolation level. In fact, Dirty Read is the only isolation level allowed for databases that do not have transaction logging.

Committed Read isolation

A reader with the Committed Read isolation (or ANSI Read Committed) isolation level checks for locks before returning a row. By checking for locks, the reader cannot return any uncommitted rows.

The database server does not actually place any locks for rows read during Committed Read. It simply checks for any existing rows in the internal lock table.

Committed Read is the default isolation level for databases with logging if the log mode is not ANSI-compliant. For databases created with a logging mode that is not ANSI-compliant, Committed Read is an appropriate isolation level for most activities. For ANSI-compliant databases, Repeatable Read is the default isolation level.

Ways to reduce the risk of Committed Read isolation level conflicts

In the Committed Read isolation level, locks held by other sessions can cause SQL operations to fail if the current session cannot acquire a lock or if the database server detects a *deadlock*. (A deadlock occurs when two users hold locks, and each user wants to acquire a lock that the other user owns.) The LAST COMMITTED keyword option to the SET ISOLATION COMMITTED READ statement of SQL reduces the risk of locking conflicts.

The LAST COMMITTED keyword option to the SET ISOLATION COMMITTED READ statement of SQL instructs the server to return the most recently committed version of the rows, even if another concurrent session holds an exclusive lock. You can use the LAST COMMITTED keyword option for B-tree and functional indexes, tables that support transaction logging,

and tables that do not have page-level locking or exclusive locks. For more information, see information about the SET ISOLATION statement in the *HCL OneDB™ Guide to SQL: Syntax*.

For databases created with transaction logging, you can set the USELASTCOMMITTED configuration parameter to specify whether the database server uses the last committed version of the data, rather than wait for the lock to be released, when sessions using the Dirty Read or Committed Read isolation level (or the ANSI/ISO level of Read Uncommitted or Read Committed) attempt to read a row on which a concurrent session holds a shared lock. The last committed version of the data is the version of the data that existed before any updates occurred.

If no value or a value of `NONE` is set for the USELASTCOMMITTED configuration parameter or for the USELASTCOMMITTED session environment variable, sessions in a COMMITTED READ or READ COMMITTED isolation level wait for any exclusive locks to be released, unless the SET ISOLATION COMMITTED READ LAST COMMITTED statement of SQL instructs the database server to read the most recently committed version of the data.

Setting the USELASTCOMMITTED configuration parameter to operate with the Committed Read isolation level can affect performance only if concurrent conflicting updates occur. When concurrent conflicting updates occur, the performance of queries depends on the dynamics of the transactions. For example, a reader using the last committed version of the data, might need to undo the updates made to a row by another concurrent transaction. This situation involves reading one or more log records, thereby increasing the I/O traffic, which can affect performance.

Related information

USELASTCOMMITTED configuration parameter on page

Cursor Stability isolation

A reader with Cursor Stability isolation acquires a shared lock on the row that is currently fetched. This action assures that no other user can update the row until the user fetches a new row.

In the example for a cursor in Figure 29: Locks placed for cursor stability on page 246, at *fetch a row* the database server releases the lock on the previous row and places a lock on the row being fetched. At *close the cursor*, the server releases the lock on the last row.

Figure 29. Locks placed for cursor stability

```
set isolation to cursor stability
declare cursor for SELECT * FROM customer
open the cursor
while there are more rows
   fetch a row
   do work
end while
close the cursor
```

If you do not use a cursor to fetch data, Cursor Stability isolation behaves in the same way as Committed Read. No locks are actually placed.

Repeatable Read isolation

Repeatable Read isolation (ANSI Serializable and ANSI Repeatable Read) is the strictest isolation level. With Repeatable Read, the database server locks all rows examined (not just fetched) for the duration of the transaction.

The example in Figure 30: Locks placed for repeatable read on page 247 shows when the database server places and releases locks for a repeatable read. At *fetch a row*, the server places a lock on the row being fetched and on every row it examines in order to retrieve this row. At *close the cursor*, the server releases the lock on the last row.

Figure 30. Locks placed for repeatable read

```
set isolation to repeatable read
begin work
declare cursor for SELECT * FROM customer
open the cursor
while there are more rows
    fetch a row
    do work
end while
close the cursor
commit work
```

Repeatable Read is useful during any processing in which multiple rows are examined, but none must change during the transaction. For example, suppose an application must check the account balance of three accounts that belong to one person. The application gets the balance of the first account and then the second. But, at the same time, another application begins a transaction that debits the third account and credits the first account. By the time that the original application obtains the account balance of the third account, it has been debited. However, the original application did not record the debit of the first account.

When you use Committed Read or Cursor Stability, the previous scenario can occur. However, it cannot occur with Repeatable Read. The original application holds a read lock on each account that it examines until the end of the transaction, so the attempt by the second application to change the first account fails (or waits, depending upon SET LOCK MODE).

Because even examined rows are locked, if the database server reads the table sequentially, a large number of rows unrelated to the query result can be locked. For this reason, use Repeatable Read isolation for tables when the database server can use an index to access a table. If an index exists and the optimizer chooses a sequential scan instead, you can use directives to force use of the index. However, forcing a change in the query path might negatively affect query performance.

Locking nonlogging tables

The database server does not place page or row locks on a nonlogging table when you use the table within a transaction. However, you can lock nonlogging tables to prevent concurrency problems when other users are modifying a nonlogging table

Use one of the following methods to prevent concurrency problems when other users are modifying a nonlogging table:

- Lock the table in exclusive mode for the whole transaction.
- Use Repeatable Read isolation level for the whole transaction.

Important: Nonlogging raw tables are intended for fast loading of data. You should change the table to STANDARD before you use it in a transaction or modify the data within it.

Update cursors

An update cursor is a special kind of cursor that applications can use when the row might potentially be updated. Update cursors use *promotable locks* in which the database server places an update lock on the row when the application fetches the row. The lock is changed to an exclusive lock when the application uses an update cursor and UPDATE...WHERE CURRENT OF to update the row.

When the update lock is on the row as the application fetches it, other users can still view the row.

In some cases, the database server might place locks on rows that the database server has examined but not actually fetched. Whether this behavior occurs depends on how the database server executes the SQL statement.

The advantage of an update cursor is that you can view the row with the confidence that other users cannot change it or view it with an update cursor while you are viewing it and before you update it.

If you do not update the row, the default behavior of the database server is to release the update lock when you execute the next FETCH statement or close the cursor. However, if you execute the SET ISOLATION statement with the RETAIN® UPDATE LOCKS clause, the database server does not release any currently existing or subsequently placed update locks until the end of the transaction.

The code in Figure 31: When update locks are released on page 248 shows when the database server places and releases update locks with a cursor. At *fetch row 1*, the database server places an update lock on row 1. At *fetch row 2*, the server releases the update lock on row 1 and places an update lock on row 2. However, after the database server executes the SET ISOLATION statement with the RETAIN® UPDATE LOCKS clause, it does not release any update locks until the end of the transaction. At *fetch row 3*, it places an update lock on row 3. At *fetch row 4*, it places an update lock on row 4. At *commit work*, the server releases the update locks for rows 2, 3, and 4.

Figure 31. When update locks are released

```
declare update cursor
begin work
open the cursor
fetch row 1
fetch row 2
SET ISOLATION TO COMMITTED READ
    RETAIN UPDATE LOCKS
fetch row 3
fetch row 4
commit work
```

In an ANSI-compliant database, update cursors are usually not needed because any select cursor behaves the same as an update cursor without the RETAIN® UPDATE LOCKS clause.

The code in Figure 32: When update locks are promoted on page 249 shows the database server promoting an update lock to an exclusive lock. At *fetch the row*, the server places an update lock on the row being fetched. At *update the row*, the server promotes the lock to exclusive. At *commit work*, it releases the lock.

Figure 32. When update locks are promoted

```
declare update cursor
begin work
open the cursor
fetch the row
do work
update the row (use WHERE CURRENT OF)
commit work
```

To use an update cursor, run SELECT FOR UPDATE in your application.

Locks placed with INSERT, UPDATE, and DELETE statements

When you execute an INSERT, UPDATE, or DELETE statement, the database server uses exclusive locks. An exclusive lock means that no other users can update or delete the item until the database server removes the lock.

In addition, no other users can view the row unless they are using the Dirty Read isolation level.

When the database server removes the exclusive lock depends on whether the database supports transaction logging:

- If the database supports logging, the database server removes all exclusive locks when the transaction completes (commits or rolls back).
- If the database does not support logging, the database server removes all exclusive locks immediately after the INSERT, MERGE, UPDATE, or DELETE statement completes, except when the lock is on the row that is currently being fetched into an update cursor.

  In this situation, the lock is retained during the fetch operation on the row, but only until the server fetches the next row, or until the server updates the current row by promoting the lock to an exclusive lock.

In a nonlogging database, the promotable update lock on a row fetched for update can be released by a DDL operation on the database while the INSERT, MERGE, UPDATE, or DELETE statement that originally created the lock is still running. To reduce the risk of data corruption if a concurrent session modifies the unlocked row, restrict operations that use promotable update locks to databases that support transaction logging.

The internal lock table

The database server stores locks in an internal lock table. When the database server reads a row, it checks if the row or its associated page, table, or database is listed in the lock table. If it is in the lock table, the database server must also check the lock type.

The following table shows the types of locks that the lock table can contain.

Lock Type	Description	Statement That Usually Places the Lock
S	Shared lock	SELECT

Lock Type	Description	Statement That Usually Places the Lock
X	Exclusive lock	INSERT, UPDATE, DELETE
U	Update lock	SELECT in an update cursor
B	Byte lock	Any statement that updates VARCHAR columns

A byte lock is generated only if you shrink the size of a data value in a VARCHAR column. The byte lock exists solely for roll forward and rollback execution, so a byte lock is created only if you are working in a database that uses logging. Byte locks appear in **onstat -k** output only if you are using row-level locking; otherwise, they are merged with the page lock.

In addition, the lock table might store *intent locks*, with the same lock type as previously shown. In some cases, a user might need to register his or her possible intent to lock an item, so that other users cannot place a lock on the item.

Depending on the type of operation and the isolation level, the database server might continue to read the row and place its own lock on the row, or it might wait for the lock to be released (if the user executed SET LOCK MODE TO WAIT). The following table shows the locks that a user can place if another user holds a certain type of lock. For example, if one user holds an exclusive lock on an item, another user requesting any kind of lock (exclusive, update, or shared) receives an error.

	Hold X lock	Hold U lock	Hold S lock
Request X lock	No	No	Yes
Request U lock	No	No	Yes
Request S lock	No	Yes	Yes

Monitoring locks

You can analyze information about locks and monitor locks by viewing information in the internal lock table that contains stored locks.

View the lock table with **onstat -k**. Figure 33: onstat -k output on page 250 shows sample output for **onstat -k**.

Figure 33. onstat -k output

```
Locks
address  wtlist   owner    lklist   type     tblsnum  rowid    key#/bsiz
300b77d0 0        40074140 0        HDR+S    10002    106        0
300b7828 0        40074140 300b77d0 HDR+S    10197    123        0
300b7854 0        40074140 300b7828 HDR+IX   101e4    0          0
300b78d8 0        40074140 300b7854 HDR+X    101e4    102        0
 4 active, 5000 total, 8192 hash buckets
```

In this example, a user is inserting one row in a table. The user holds the following locks (described in the order shown):

- A shared lock on the database
- A shared lock on a row in the **systables** system catalog table
- An intent-exclusive lock on the table
- An exclusive lock on the row

To determine the table to which the lock applies, execute the following SQL statement. For *tblsnum*, substitute the value shown in the **tblsnum** field in the **onstat -k** output.

```
SELECT *
   FROM SYSTABLES
   WHERE HEX(PARTNUM) = "tblsnum";
```

Where *tblsnum* is the modified value that **onstat -k** returns. For example, if **onstat -k** returns `10027f`, *tbslnum* is `0x0010027F`.

You can also query the **syslocks** table in the **sysmaster** database to obtain information about each active lock. The **syslocks** table contains the following columns.

Column	Description
dbsname	Database on which the lock is held
tabname	Name of the table on which the lock is held
rowidlk	ID of the row on which the lock is held (0 indicates a table lock.)
keynum	The key number for the row
type	Type of lock
owner	Session ID of the lock owner
waiter	Session ID of the first waiter on the lock

Configuring and managing lock usage

The LOCKS configuration parameter specifies the initial size of the internal lock table. If the database server increases the size of the lock table, you should increase the size of the LOCKS configuration parameter.

For information about how to determine an initial value for the LOCKS configuration parameter, see The LOCKS configuration parameter and memory utilization on page 77.

If the number of locks needed by sessions exceeds the value set in the LOCKS configuration parameter, the database server attempts to increase the lock table by doubling its size. Each time that the lock table overflows (when the number of locks needed is greater than the current size of the lock table), the database server increases the size of the lock table, up to 99 times. Each time that the database server increases the size of the lock table, the server attempts to double its size. However, the server will limit each actual increase to no more than the maximum number of added locks shown in Table 13: Maximum number of locks on 32-bit and 64-bit platforms on page 252. After the 99th time that the database server increases the lock table, the server no longer increases the size of the lock table, and an application needing a lock receives an error.

Maximum number of locks allowed on 32-bit and 64-bit platforms

The following table shows the maximum number of allowed locks.

Table 13. Maximum number of locks on 32-bit and 64-bit platforms

Platform	Maximum Number of Initial Locks	Maximum Number of Dynamic Lock Table Extensions	Maximum Number of Locks Added Per Lock Table Extension	Maximum Number of Locks Allowed
32-bit	8,000,000	99	100,000	8,000,000 + (99 x 100,000)
64-bit	500,000,000	99	1,000,000	500,000,000 + (99 x 1,000,000)

View messages concerning increases to the size of the lock table

Every time the database server increases the size of the lock table, the server places a message in the message log file. You should monitor the message log file periodically and increase the size of the LOCKS configuration parameter if you see that the database server has increased the size of the lock table.

Monitor out-of-locks errors

To monitor the number of times that applications receive the out-of-locks error, view the **ovlock** field in the output of **onstat -p**. You can also see similar information from the **sysprofile** table in the **sysmaster** database. The following rows contain the relevant statistics.

Row	Description
ovlock	Number of times that sessions attempted to exceed the maximum number of locks
lockreqs	Number of times that sessions requested a lock
lockwts	Number of times that sessions waited for a lock

Examine how applications use locks

If the database server is using an unusually large number of locks, you can examine how individual applications are using locks, as follows:

1. Monitor sessions with **onstat -u** to see if a particular user is using an especially high number of locks (a high value in the **locks** column).
2. If a particular user uses a large number of locks, examine the SQL statements in the application to determine whether you should lock the table or use individual row or page locks.

A table lock is more efficient than individual row locks, but it reduces concurrency.

One way to reduce the number of locks placed on a table is to alter a table to use page locks instead of row locks. However, page locks reduce overall concurrency for the table, which can affect performance.

You can also reduce the number of locks placed on a table by locking the table in exclusive mode.

Related information

The LOCKS configuration parameter and memory utilization on page 77

Monitoring lock waits and lock errors

You can view information about sessions, lock usage, and lock waits.

About this task

If the application executes SET LOCK MODE TO WAIT, the database server waits for a lock to be released instead of returning an error. An unusually long wait for a lock can give users the impression that the application is hanging.

In Figure 34: onstat -u output that shows lock usage on page 253, the **onstat -u** output shows that session ID 84 is waiting for a lock (`L` in the first column of the **Flags** field). To find out the owner of the lock, use the **onstat -k** command.

Figure 34. onstat -u output that shows lock usage

```
onstat -u

Userthreads
address  flags   sessid user     tty    wait    tout locks nreads nwrites
40072010 ---P--D 7      informix -         0       0     0     35      75
400723c0 ---P--- 0      informix -         0       0     0      0       0
40072770 ---P--- 1      informix -         0       0     0      0       0
40072b20 ---P--- 2      informix -         0       0     0      0       0
40072ed0 ---P--F 0      informix -         0       0     0      0       0
40073280 ---P--B 8      informix -         0       0     0      0       0
40073630 ---P--- 9      informix -         0       0     0      0       0
400739e0 ---P--D 0      informix -         0       0     0      0       0
40073d90 ---P--- 0      informix -         0       0     0      0       0
40074140Y-BP---81   lsuto    4  50205788       0     4    106     221
400744f0 --BP--- 83     jsmit    -         0       0     4      0       0
400753b0 ---P--- 86     worth    -         0       0     2      0       0
40075760 L--PR--84     jones    3  300b78d8   -1     2      0       0
 13 active, 128 total, 16 maximum concurrent

onstat -k

Locks
address  wtlist   owner    lklist   type    tblsum rowid  key#/bsiz
300b77d0 0        40074140 0        HDR+S   10002  106     0
300b7828 0        40074140 300b77d0 HDR+S   10197  122     0
300b7854 0        40074140 300b7828 HDR+IX  101e4  0       0
300b78d840075760400741403 00b7854 HDR+X  101e4  100     0
300b7904 0        40075760 0             S   10002  106     0
300b7930 0        40075760 300b7904      S   10197  122     0
 6 active, 5000 total, 8192 hash buckets
```

To find out the owner of the lock for which session ID 84 is waiting:

1. Obtain the address of the lock in the **wait** field (`300b78d8`) of the **onstat -u** output.
2. Find this address (`300b78d8`) in the **Locks address** field of the **onstat -k** output.

   The **owner** field of this row in the **onstat -k** output contains the address of the user thread (`40074140`).
3. Find this address (`40074140`) in the **Userthreads** field of the **onstat -u** output.

   The **sessid** field of this row in the **onstat -u** output contains the session ID (`81`) that owns the lock.

Results

To eliminate the contention problem, you can have the user exit the application gracefully. If this solution is not possible, you can stop the application process or remove the session with **onmode -z**.

Monitoring the number of free locks

You can find the current number of free locks on a lock-free list by viewing the output of the onstat -L command .

About this task

Related information

onstat -L command: Print the number of free locks on page

Monitoring deadlocks

You can monitor deadlocks. A *deadlock* occurs when two users hold locks, and each user wants to acquire a lock that the other user owns.

For example, user **pradeep** holds a lock on row 10. User **jane** holds a lock on row 20. Suppose that **jane** wants to place a lock on row 10, and **pradeep** wants to place a lock on row 20. If both users execute SET LOCK MODE TO WAIT, they potentially might wait for each other forever.

HCL OneDB™ uses the lock table to detect deadlocks automatically and stop them before they occur. Before a lock is granted, the database server examines the lock list for each user. If a user holds a lock on the resource that the requestor wants to lock, the database server traverses the lock wait list for the user to see if the user is waiting for any locks that the requestor holds. If so, the requestor receives a deadlock error.

Deadlock errors can be unavoidable when applications update the same rows frequently. However, certain applications might always be in contention with each other. Examine applications that are producing a large number of deadlocks and try to run them at different times.

To monitor the number of deadlocks, use the **deadlks** field in the output of **onstat -p**.

In a distributed transaction, the database server does not examine lock tables from other database server systems, so deadlocks cannot be detected before they occur. Instead, you can set the DEADLOCK_TIMEOUT configuration parameter. DEADLOCK_TIMEOUT specifies the number of seconds that the database server waits for a remote database server

response before it returns an error. Although reasons other than a distributed deadlock might cause the delay, this mechanism keeps a transaction from hanging indefinitely.

To monitor the number of distributed deadlock timeouts, use the **dltouts** field in the **onstat -p** output.

Monitoring isolation levels that sessions use

The **onstat -g ses** and **onstat -g sql** output shows the isolation level that a session is currently using.

The following table summarizes the values in the **IsoLvl** column in **onstat -g ses** and **onstat -g sql** output.

Value

Description

DR

Dirty Read

CR

Committed Read

CS

Cursor Stability

CRU

Committed Read with RETAIN® UPDATE LOCKS

CSU

Cursor Stability with RETAIN® UPDATE LOCKS

DRU

Dirty Read with RETAIN® UPDATE LOCKS

LC

Committed Read, Last Committed

RR

Repeatable Read

If a great deal of lock contention occurs, check the isolation level of sessions to make sure it is appropriate for the application.

Locks for smart large objects

Smart large objects have several unique locking behaviors because their columns are typically much larger than other columns in a table.

This section discusses the following unique behaviors:

- Types of locks on smart large objects
- Byte-range locking
- Lock promotion
- Dirty Read isolation level with smart large objects

Types of locks on smart large objects

The database server uses one of the following granularity levels for locking smart large objects:

- The sbspace chunk header partition
- The smart large object
- A byte range of the smart large object

The default locking granularity is at the level of the smart large object. In other words, when you update a smart large object, by default the database server locks the smart large object that is being updated.

Locks on the sbspace chunk header partition only occur when the database server promotes locks on smart large objects. For more information, see Lock promotion on page 259.

Byte-range locking

Rather than locking the entire smart large object, you can lock only a specific byte range of a smart large object.

Byte-range locking is advantageous because it allows multiple users to update the same smart large object simultaneously, as long as they are updating different parts of it. Also, users can read a part of a smart large object while another user is updating or reading a different part of the same smart large object.

Figure 35: Example of byte-range locking on page 256 shows two locks placed on a single smart large object. The first lock is on bytes 2, 3, and 4. The second lock is on byte 6 alone.

Figure 35. Example of byte-range locking

How the database server manages byte-range locks

The database server manages byte-range locks in the lock table in a similar fashion to other locks placed on rows, pages, and tables. However, the lock table must also store the byte range.

If you place a second lock on a byte range adjacent to a byte range that is currently locked, the database server consolidates the two locks into one lock on the entire range.

If a user holds locks that the Figure 35: Example of byte-range locking on page 256 shows, and the user requests a lock on byte five, the database server consolidates the locks placed on bytes two through six into one lock.

Likewise, if a user unlocks only a portion of the bytes included within a byte-range lock, the database server might be split into multiple byte-range locks. In the Figure 35: Example of byte-range locking on page 256 the user could unlock byte three, which causes the database server to change the one lock on bytes two through four to one lock on byte two and one lock on byte four.

Using byte-range locks

By default, the database server places a lock on the smart large object. Instead, you can enable byte-range locking.

To use byte-range locks, you must perform one of the following actions:

- To set byte-range locking for the sbspace that stores the smart large object, use the **onspaces** utility. The following example sets byte-range locking for the new sbspace:

  ```
  onspaces -c -S slo -g 2 -p /ix/9.2/liz/slo -o 0 -s 1000
     -Df LOCK_MODE=RANGE
  ```

  When you set the default locking mode for the sbspace to byte-range locking, the database server locks only the necessary bytes when it updates any smart large objects stored in the sbspace.

- To set byte-range locking for the smart large object when you open it, use one of the following methods:
  - *In DB-Access:* Set the MI_LO_LOCKRANGE flag in the **mi_lo_open()** DataBlade® API function.
  - *In ESQL/C:* Set the LO_LOCKRANGE flag in the **ifx_lo_open()** function. When you set byte-range locking for the individual smart large object, the database server implicitly locks only the necessary bytes when it selects or updates the smart large object.
- To lock a byte range explicitly, use one of the following functions:
  - *For DB-Access*:**mi_lo_lock()**
  - *For ESQL/C*:**ifx_lo_lock()**

    These functions lock the range of bytes that you specify for the smart large object. If you specify an exclusive lock with either function, UPDATE statements do not place locks on the smart large object if they update the locked bytes.

    The database server releases exclusive byte-range locks placed with **mi_lo_lock()** or **ifx_lo_lock()** at the end of the transaction. The database server releases shared byte-range locks placed with **mi_lo_lock()** or **ifx_lo_lock()** based on the same rules as locks placed with SELECT statements, depending upon the isolation level. You can also release shared byte-range locks with one of the following functions:

  - *For DB-Access*:**mi_lo_unlock()**. For more information about the DataBlade® API functions, see the *HCL OneDB™ DataBlade® API Programmer's Guide*.
  - *For ESQL/C*:**ifx_lo_unlock()**. For more information about functions, see the *HCL OneDB™ ESQL/C Programmer's Manual*.

Monitoring byte-range locks

You can use **onstat -k** to list all byte-range locks. Use the **onstat -K** command to list byte-range locks and all waiters for byte-range locks.

Figure 36: Byte-range locks in onstat -k output on page 258 shows an excerpt from the output of **onstat -k**.

Figure 36. Byte-range locks in onstat -k output

```
Byte-Range Locks
rowid/LOid   tblsnum  address  status   owner    offset   size     type
104          200004   a020e90  HDR
[2, 2, 3]             a020ee4  HOLD     a1b46d0  50       10           S
202          200004   a021034  HDR
[2, 2, 5]             a021088  HOLD     a1b51e0  40       5            S
102          200004   a035608  HDR
[2, 2, 1]             a0358fc  HOLD     a1b4148  0        500          S
                      a035758  HOLD     a1b3638  300      100          S
 21 active, 2000 total, 2048 hash buckets
```

Byte-range locks produce the following information in the **onstat -k** output.

Column	Description
rowid	The rowid of the row that contains the locked smart large object
LOid	The three values: sbspace number, chunk number, and sequence number (a value that represents the position in the chunk)
tblsnum	The number of the tblspace that holds the smart large object
address	The address of the lock
status	The status of the lock HDR is a placeholder. HOLD indicates the user specified in the **owner** column owns the lock. WAIT (shown only with **onstat -K**) indicates that the user specified in the owner column is waiting for the lock.
owner	The address of the owner (or waiter) Cross reference this value with the address in **onstat -u**.
offset	The offset into the smart large object where the bytes are locked
size	The number of bytes locked, starting at the value in the offset column
type	S (shared lock) or X (exclusive)

Setting number of locks for byte-range locking

When you use byte-range locking, the database server can use more locks because of the possibility of multiple locks on one smart large object. Even though the lock table grows when it runs out of space, you might want to increase value of the LOCKS configuration parameter to match lock usage so that the database server does not need to allocate more space dynamically.

Be sure to monitor the number of locks used with **onstat -k**, so you can determine if you need to increase the value of the LOCKS configuration parameter.

Lock promotion

The database server uses lock promotion to decrease the total number of locks held on smart large objects. Too many locks can result in poorer performance because the database server frequently searches the lock table to determine if a lock exists on an object.

If the number of locks held by a transaction exceeds 33 percent of the current number of allocated locks for the database server, the database server attempts to promote any existing byte-range locks to a single lock on the smart large object.

If the number of locks that a user holds on a smart large object (not on byte ranges of a smart large object) equals or exceeds 10 percent of the current capacity of the lock table, the database server attempts to promote all of the smart-large-object locks to one lock on the smart-large-object header partition. This kind of lock promotion improves performance for applications that are updating, loading, or deleting a large number of smart large objects. For example, a transaction that deletes millions of smart large objects would consume the entire lock table if the database server did not use lock promotion. The lock promotion algorithm has deadlock avoidance built in.

You can identify a smart-large-object header partition in **onstat -k** by `0` in the **rowid** column and a tablespace number with a high-order first byte-and-a-half that corresponds to the dbspace number where the smart large object is stored. For example, if the tblspace number is listed as `0x200004` (the high-order zeros are truncated), the dbspace number `2` corresponds to the dbspace number listed in **onstat -d**.

Even if the database server attempts to promote a lock, it might not be able to do so. For example, the database server might not be able to promote byte-range locks to one smart-large-object lock because other users have byte-range locks on the same smart large object. If the database server cannot promote a byte-range lock, it does not change the lock, and processing continues as normal.

Dirty Read isolation level and smart large objects

You can use the Dirty Read isolation level for smart large objects.

For information about how Dirty Reads affects consistency, see Dirty Read isolation on page 245.

Set the Dirty Read isolation level for smart large objects in one of the following ways:

- Use the SET TRANSACTION MODE or SET ISOLATION statement.
- Use the LO_DIRTY_READ flag in one of the following functions:

- *For DB-Access*:**mi_lo_open()**
- *For ESQL/C*:**ifx_lo_open()**

If consistency for smart large objects is not important, but consistency for other columns in the row is important, you can set the isolation level to Committed Read, Cursor Stability, or Repeatable Read and open the smart large object with the LO_DIRTY_READ flag.

Fragmentation guidelines

One of the most frequent causes of poor performance in relational database systems is contention for data that resides on a single I/O device. Proper fragmentation of high-use tables can significantly reduce I/O contention. These topics discuss the performance considerations that are involved when you use table fragmentation.

The database server supports table fragmentation (also *partitioning*), which allows you to store data from a single table on multiple disk devices. Fragmentation enables you to define groups of rows or index keys within a table according to some algorithm or scheme. You can store each group or fragment (also referred to as a *partition*) in a separate dbspace associated with a specific physical disk.

For information about fragmentation and parallel execution, see Parallel database query (PDQ) on page 345.

For an introduction to fragmentation concepts and methods, see the *HCL OneDB™ Database Design and Implementation Guide*. For information about the SQL statements that manage fragments, see the *HCL OneDB™ Guide to SQL: Syntax*.

Planning a fragmentation strategy

You can decide on a fragmentation goal for your database and devise a strategy to meet that goal.

About this task

A fragmentation strategy consists of two parts:

- A distribution scheme that specifies how to group rows into fragments

  You specify the distribution scheme in the FRAGMENT BY clause of the CREATE TABLE, CREATE INDEX, or ALTER FRAGMENT statements.
- The set of dbspaces in which you locate the fragments

  You specify the set of dbspaces or in the IN clause (storage option) of these SQL statements.

To formulate a fragmentation strategy:

1. Decide on your primary fragmentation goal, which should depend, to a large extent, on the types of applications that access the table.
2. Make the following decisions based on your primary fragmentation goal:
   - Whether to fragment the table data, the table index, or both
   - What the ideal distribution of rows or index keys is for the table

3. Choose either an expression-based or round-robin distribution scheme:
   - If you choose an expression-based distribution scheme, you must then design suitable fragment expressions.
   - If you choose a round-robin distribution scheme, the database server determines which rows to put into a specific fragment.

   For more information, see Distribution schemes on page 265.
4. To complete the fragmentation strategy, you must decide on the number and location of the fragments:
   - The number of fragments depends on your primary fragmentation goal.
   - Where you locate fragments depends on the number of disks available in your configuration.

Results

When you plan a fragmentation strategy, be aware of these space and page issues:

- Although a 4-terabyte chunk can be on a 2-kilobyte page, only 32 gigabytes can be utilized in a dbspace because of a rowid format limitation.
- For a fragmented table, all fragments must use the same page size.
- For a fragmented index, all fragments must use the same page size.
- A table can be in one dbspace and the index for that table can be in another dbspace. These dbspaces do not need to have the same page size.

Fragmentation goals

You can analyze your application and workload to identify fragmentation goals and to determine the balance to strike among fragmentation goals.

Fragmentation goals can include:

- Improved performance for individual queries

  To improve the performance of individual queries, fragment tables appropriately and set resource-related parameters to specify system resource use (memory, CPU virtual processors, and so forth).
- Reduced contention between queries and between transactions

  If your database server is used primarily for online transaction processing (OLTP) and only incidentally for decision-support queries, you can often use fragmentation to reduce contention when simultaneous queries against the same table perform index scans to return a few rows.
- Increased data availability

  Careful fragmentation of dbspaces can improve data availability if devices fail. Table fragments on the failed device can be restored quickly, and other fragments are still accessible.
- Improved data-load performance

When you use the High-Performance Loader (HPL) to load a table that is fragmented across multiple disks, it allocates threads to insert the data into the fragments in parallel, using light appends. For more information about this load method, see the *HCL OneDB™ High-Performance Loader User's Guide*.

You can use the ALTER FRAGMENT ON TABLE statement with the ATTACH clause to add data quickly to a very large table. For more information, see Improve the performance of operations that attach and detach fragments on page 278.

The performance of a fragmented table is primarily governed by the following factors:

- The storage option that you use for allocating disk space to fragments (discussed in Considering physical fragmentation factors on page 265)
- The distribution scheme used to assign rows to individual fragments (discussed in Distribution schemes on page 265)

Improved query performance through fragmentation strategy

If the primary goal of fragmentation is improved performance for individual queries, try to distribute all of the rows of the table evenly over the different disks. Overall query-completion time is reduced when the database server does not need to wait for data retrieval from a table fragment that has more rows than other fragments.

If queries access data by performing sequential scans against significant portions of tables, fragment the table rows only. Do not fragment the index. If an index is fragmented and a query has to cross a fragment boundary to access the data, the performance of the query can be worse than if you do not fragment.

If queries access data by performing an index read, you can improve performance by using the same distribution scheme for the index and the table.

If you use round-robin fragmentation, do not fragment your index. Consider placing that index in a separate dbspace from other table fragments.

For more information about improving performance for queries, see Query expressions for fragment elimination on page 275 and Improving individual query performance on page 365.

Reduced contention between queries and transactions

Fragmentation can reduce contention for data in tables that multiple queries and OLTP applications use. Fragmentation often reduces contention when many simultaneous queries against a table perform index scans to return a few rows.

For tables subjected to this type of load, fragment both the index keys and data rows with a distribution scheme that allows each query to eliminate unneeded fragments from its scan. Use an expression-based distribution scheme. For more information, see Distribution schemes that eliminate fragments on page 274.

To fragment a table for reduced contention, start by investigating which queries access which parts of the table. Next, fragment your data so that some of the queries are routed to one fragment while others access a different fragment. The

database server performs this routing when it evaluates the fragmentation rule for the table. Finally, store the fragments on separate disks.

Your success in reducing contention depends on how much you know about the distribution of data in the table and the scheduling of queries against the table. For example, if the distribution of queries against the table is set up so that all rows are accessed at roughly the same rate, try to distribute rows evenly across the fragments. However, if certain values are accessed at a higher rate than others, you can compensate for this difference by distributing the rows over the fragments to balance the access rate. For more information, see Designing an expression-based distribution scheme on page 268.

Increased data availability

When you distribute table and index fragments across different disks or devices, you improve the availability of data during disk or device failures. The database server continues to allow access to fragments stored on disks or devices that remain operational.

This availability has important implications for the following types of applications:

- Applications that do not require access to unavailable fragments

  A query that does not require the database server to access data in an unavailable fragment can still successfully retrieve data from fragments that are available. For example, if the distribution expression uses a single column, the database server can determine if a row is contained in a fragment without accessing the fragment. If the query accesses only rows that are contained in available fragments, a query can succeed even when some of the data in the table is unavailable. For more information, see Designing an expression-based distribution scheme on page 268.
- Applications that accept the unavailability of data

  Some applications might be designed in such a way that they can accept the unavailability of data in a fragment and require the ability to retrieve the data that is available. To specify which fragments can be skipped, these applications can execute the SET DATASKIP statement before they execute a query. Alternatively, the database server administrator can use the onspaces -f option to specify which fragments are unavailable.

If your fragmentation goal is increased availability of data, fragment both table rows and index keys so that if a disk drive fails, some of the data is still available. If applications must always be able to access a subset of your data, keep those rows together in the same mirrored dbspace.

Increased granularity for backup and restore

You must consider backup and restore factors when you are deciding how to distribute dbspaces across disk.

Backup and restore factors to consider are:

- **Data availability.** When you decide where to place your tables or fragments, remember that if a device that contains a dbspace fails, all tables or table fragments in that dbspace are inaccessible, even though tables and fragments in other dbspaces are accessible. The need to limit data unavailability in the event of a disk failure might influence which tables you group together in a particular dbspace.
- **Cold versus warm restores.** Although you must perform a cold restore if a dbspace that contains critical data fails, you need to perform only a warm restore if a noncritical dbspace fails. The desire to minimize the impact of cold restores might influence the dbspace that you use to store critical data.

For more information about backup and restore, see your *HCL OneDB™ Backup and Restore Guide*.

Examining your data and queries

To determine a fragmentation strategy, you must gather information about the table that you might fragment. You must also know how the data in the table is used.

To gather information about your table:

1. Identify the queries that are critical to performance to determine if the queries are online transaction processing (OLTP) or decision-support system (DSS) queries.
2. Use the SET EXPLAIN statement to determine how the data is being accessed.

   For information about the output of the SET EXPLAIN statement, see Report that shows the query plan chosen by the optimizer on page 299. To determine how the data is accessed, you can sometimes simply review the SELECT statements along with the table schema.
3. Determine what portion of the data each query examines.

   For example, if certain rows in the table are read most of the time, you can isolate them in a small fragment to reduce I/O contention for other fragments.
4. Determine which statements create temporary files.

   Decision-support queries typically create and access large temporary files, and placement of temporary dbspaces can be critical to performance.
5. If particular tables are always joined together in a decision-support query, spread fragments for these tables across different disks.
6. Examine the columns in the table to determine which fragmentation scheme would keep each scan thread equally busy for the decision-support queries.

   To see how the column values are distributed, create a distribution on the column with the UPDATE STATISTICS statement and examine the distribution with **dbschema**.

   ```
   dbschema -d database -hd table
   ```

Considering physical fragmentation factors

When you fragment a table, the physical placement issues that pertain to tables apply to individual table fragments. Because each fragment resides in its own dbspace on a disk, you must address these issues separately for the fragments on each disk.

For details about placement issues that apply to tables, see Table performance considerations on page 156.

Fragmented and nonfragmented tables differ in the following ways:

- For fragmented tables, each fragment is placed in a separate, designated dbspace or multiple named fragments of the table are created within a single dbspace.

  For nonfragmented tables, the table can be placed in the default dbspace of the current database.

  Regardless of whether the table is fragmented or not, you should create a single chunk on each disk for each dbspace.
- Extent sizes for a fragmented table are usually smaller than the extent sizes for an equivalent nonfragmented table because fragments do not grow in increments as large as the entire table. For more information on how to estimate the space to allocate, see Estimating table size on page 160.
- In a fragmented table, the row pointer is not a unique unchanging pointer to the row on a disk. The database server uses the combination of fragment ID and row pointer internally, inside an index, to point to the row. These two fields are unique but can change over the life of the row. An application cannot access the fragment ID; therefore, you should use primary keys to access a specific row in a fragmented table. For more information, see the *HCL OneDB™ Database Design and Implementation Guide*.
- An attached index or an index on a nonfragmented table uses 4 bytes for the row pointer. A detached index uses 8 bytes of disk space per key value for the fragment ID and row pointer combination. For more information about how to estimate space for an index, see Estimating index pages on page 209. For more information on attached indexes and detached indexes, see Strategy for fragmenting indexes on page 270.

Decision-support queries usually create and access large temporary files; placement of temporary dbspaces is a critical factor for performance. For more information about placement of temporary files, see Spreading temporary tables and sort files across multiple disks on page 159.

Distribution schemes

After you decide whether to fragment table rows, index keys, or both, and you decide how the rows and keys should be distributed over fragments, you can decide on a scheme to implement this distribution. OneDB supports random distribution among fragments and value-based distribution among fragments.

Random distribution among fragments

Round-robin fragmentation

This type of fragmentation places rows one after another in fragments, rotating through the series of fragments to distribute the rows evenly.

For smart large objects, you can specify multiple sbspaces in the PUT clause of the CREATE TABLE or ALTER TABLE statement to distribute smart large objects in a round-robin distribution scheme so that the number of smart large objects in each space is approximately equal.

Value-based distribution among fragments

Expression-based fragmentation

This type of fragmentation puts rows that contain specified values in the same fragment. You specify a *fragmentation expression* that defines criteria for assigning a set of rows to each fragment, either as a range rule or some arbitrary rule.

You can specify a *remainder fragment* that holds all rows that do not match the criteria for any other fragment, although a remainder fragment reduces the efficiency of the expression-based distribution scheme.

List-based fragmentation

This type of fragmentation puts rows that contain specified values that match one of the specified values in a list of discrete values in the same fragment. For each fragment, you specify a list of one or more constant expressions as *fragment expressions* that correspond to one or more columns in the table. The column or set of columns from which the *fragment expressions* are calculated is called the *fragment key*.

You can optionally specify a *remainder fragment* that holds all rows that do not match the criteria for any other fragment. You can also optionally specify a NULL fragment that stores rows with missing data in the fragment key columns (because its fragment expression is NULL or IS NULL).

The most important difference between fragmentation by list and fragmentation by expression is that every value in the list for each fragment must be unique among all the lists for fragments of the same table or index.

Interval-based fragmentation

This type of fragmentation partitions data into fragments that are based on quantified values within a specific interval within the range of fragment key of a single numeric, DATE, or DATETIME column in the same fragment. You specify at least one range expression as the *fragment expression* that defines the upper limit of fragment key values for each fragment, and an *interval expression* that specifies the size of the range of system-defined fragments that the database server creates automatically.

You can optionally define a NULL fragment to store rows with missing data in the fragment key column, but no *remainder fragment* is supported or needed. The database server automatically creates a new fragment to store rows with non-NULL fragment key values outside the range of any existing fragment. The fragments that you define with range expressions are called *range fragments*, and the system-defined fragments that the database server creates at runtime are called *interval fragments*. This type of distribution scheme is sometimes called a *range interval* distribution strategy.

Related information

Specify temporary tables in the DBSPACETEMP configuration parameter on page 117

List fragmentation clause on page

Interval fragment clause on page

Fragmentation: Storage distribution strategies on page

Choosing a distribution scheme

When choosing a distribution scheme, you must consider the ease of data balancing, whether you want fragments to be eliminated, and the effect of the data skip feature.

Table 14: Distribution-Scheme Comparisons on page 267 compares round-robin and expression-based distribution schemes.

Table 14. Distribution-Scheme Comparisons

Distribution Scheme	Ease of Data Balancing	Fragment Elimination	Data Skip
Round-robin	Automatic. Data is balanced over time.	The database server cannot eliminate fragments.	You cannot determine if the integrity of the transaction is compromised when you use the data-skip feature. However, you can insert into a table fragmented by round-robin.
Expression-based	Requires knowledge of the data distribution.	If expressions on one or two columns are used, the database server can eliminate fragments for queries that have either range or equality expressions.	You can determine whether the integrity of a transaction has been compromised when you use the data-skip feature. You cannot insert rows if the appropriate fragment for those rows is down.

The distribution scheme that you choose depends on the following factors:

- The features in Table 14: Distribution-Scheme Comparisons on page 267 of which you want to take advantage
- Whether or not your queries tend to scan the entire table
- Whether or not you know the distribution of data to be added
- Whether or not your applications tend to delete many rows
- Whether or not you cycle your data through the table

Basically, the round-robin scheme provides the easiest and surest way of balancing data. However, with round-robin distribution, you have no information about the fragment in which a row is located, and the database server cannot eliminate fragments.

In general, round-robin is the correct choice only when all the following conditions apply:

- Your queries tend to scan the entire table.
- You do not know the distribution of data to be added.
- Your applications tend not to delete many rows. (If they do, load balancing can be degraded.)

An expression-based scheme might be the best choice to fragment the data if any of the following conditions apply:

- Your application calls for numerous decision-support queries that scan specific portions of the table.
- You know what the data distribution is.
- You plan to cycle data through a database.

If you plan to add and delete large amounts of data periodically, based on the value of a column such as date, you can use that column in the distribution scheme. You can then use the alter fragment attach and alter fragment detach statements to cycle the data through the table.

The ALTER FRAGMENT ATTACH and DETACH statements provide the following advantages over bulk loads and deletes:

- The rest of the table fragments are available for other users to access. Only the fragment that you attach or detach is not available to other users.
- With the performance enhancements, the execution of an ALTER FRAGMENT ATTACH or DETACH statement is much faster than a bulk load or mass delete.

For more information, see Improve the performance of operations that attach and detach fragments on page 278.

In some cases, an appropriate index scheme can circumvent the performance problems of a particular distribution scheme. For more information, see Strategy for fragmenting indexes on page 270.

Designing an expression-based distribution scheme

The first step in designing an expression-based distribution scheme is to determine the distribution of data in the table, particularly the distribution of values for the column on which you base the fragmentation expression.

To obtain this information, run the UPDATE STATISTICS statement for the table and then use the **dbschema** utility to examine the distribution.

After you know the data distribution, you can design a fragmentation rule that distributes data across fragments as required to meet your fragmentation goal. If your primary goal is to improve performance, your fragment expression should generate an even distribution of rows across fragments.

If your primary fragmentation goal is improved concurrency, analyze the queries that access the table. If certain rows are accessed at a higher rate than others, you can compensate by opting for an uneven distribution of data over the fragments that you create.

Try not to use columns that are subject to frequent updates in the distribution expression. Such updates can cause rows to move from one fragment to another (that is, be deleted from one and added to another), and this activity increases CPU and I/O overhead.

Try to create nonoverlapping regions based on a single column with no REMAINDER fragment for the best fragment-elimination characteristics. The database server eliminates fragments from query plans whenever the query optimizer can determine that the values selected by the WHERE clause do not reside on those fragments, based on the expression-based fragmentation rule by which you assign rows to fragments. For more information, see Distribution schemes that eliminate fragments on page 274.

Suggestions for improving fragmentation

You can improve fragmentation for optimal performance in decision-support and OLTP queries.

The following suggestions are guidelines for fragmenting tables and indexes:

- For optimal performance in decision-support queries, fragment the table to increase parallelism, but do not fragment the indexes. Detach the indexes, and place them in a separate dbspace.
- For best performance in OLTP queries, use fragmented indexes to reduce contention between sessions. You can often fragment an index by its key value, which means the OLTP query only has to look at one fragment to find the location of the row.

  If the key value does not reduce contention, as when every user looks at the same set of key values (for instance, a date range), consider fragmenting the index on another value used in the WHERE clause. To cut down on fragment administration, consider not fragmenting some indexes, especially if you cannot find a good fragmentation expression to reduce contention.
- Use round-robin fragmentation on data when the table is read sequentially by decision-support queries. Round-robin fragmentation is a good method for spreading data evenly across disks when no column in the table can be used for an expression-based fragmentation scheme. However, in most DSS queries, all fragments are read.
- To reduce the total number of required dbspaces and decrease the time needed for searches, you can store multiple named fragments within the same dbspace.
- If you are using expressions, create them so that I/O requests, rather than quantities of data, are balanced across disks. For example, if the majority of your queries access only a portion of data in the table, set up your fragmentation expression to spread active portions of the table across disks, even if this arrangement results in an uneven distribution of rows.
- Keep fragmentation expressions simple. Fragmentation expressions can be as complex as you want. However, complex expressions take more time to evaluate and might prevent fragments from being eliminated from queries.
- Arrange fragmentation expressions so that the most restrictive condition for each dbspace is tested within the expression first. When the database server tests a value against the criteria for a given fragment, evaluation stops when a condition for that fragment tests false. Thus, if the condition that is most likely to be false is placed first, fewer conditions need to be evaluated before the database server moves to the next fragment. For example, in the following expression, the database server tests all six of the inequality conditions when it attempts to insert a row with a value of `25`:

  ```
  x >= 1 and x <= 10 in dbspace1,
  x > 10 and x <= 20 in dbspace2,
  x > 20 and x <= 30 in dbspace3
  ```

By comparison, only four conditions in the following expression need to be tested: the first inequality for **dbspace1** `(x <= 10)`, the first for **dbspace2** `(x <= 20)`, and both conditions for **dbspace3**:

```
x <= 10 and x >= 1 in dbspace1,
x <= 20 and x > 10 in dbspace2,
x <= 30 and x > 20 in dbspace3
```

- Avoid any expression that requires a data-type conversion. Type conversions increase the time to evaluate the expression. For instance, a DATE data type is implicitly converted to INTEGER for comparison purposes.
- Do not fragment on columns that change frequently unless you are willing to incur the administration costs. For example, if you fragment on a date column and older rows are deleted, the fragment with the oldest dates tends to empty, and the fragment with the recent dates tends to fill up. Eventually you must drop the old fragment and add a new fragment for newer orders.
- Do not fragment every table. Identify the critical tables that are accessed most frequently. Put only one fragment for a table on a disk.
- Do not fragment small tables. Fragmenting a small table across many disks might not be worth the overhead of starting all the scan threads to access the fragments. Also, balance the number of fragments with the number of processors on your system.
- When you define a fragmentation strategy on an unfragmented table, check the next-extent size to ensure that you are not allocating large amounts of disk space for each fragment.

Strategy for fragmenting indexes

When you fragment a table, the indexes that are associated with that table are fragmented implicitly, according to the distribution scheme that you use, except for the round-robin fragmentation scheme when automatic location is enabled. Indexes on tables that use the round-robin distribution scheme are not fragmented when the AUTOLOCATE configuration parameter or environment option is set to a positive integer. You can use the FRAGMENT BY clause of the CREATE INDEX statement to fragment the index on any table.When you fragment a table, the indexes that are associated with that table are fragmented implicitly, according to the distribution scheme that you use. You can use the FRAGMENT BY clause of the CREATE INDEX statement to fragment the index on any table.

Each index of a fragmented table occupies its own tblspace with its own extents.

You can fragment the index with either of the following strategies:

- Same fragmentation strategy as the table
- Different fragmentation strategy from the table

Attached indexes

An *attached index* is an index that implicitly follows the table fragmentation strategy (distribution scheme and set of dbspaces in which the fragments are located). When you create an index on a fragmented table, the index is an attached index, unless you use the round-robin distribution scheme and automatic location is enabled. Indexes on tables that use the round-robin distribution scheme are not fragmented when the AUTOLOCATE configuration parameter or environment option is set to a positive integer. When you create an index on a fragmented table, the index is an attached index.

To create an attached index, do not specify a fragmentation strategy or storage option in the CREATE INDEX statement, as in the following sample SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN dbsbspace1,
      (a >=5 AND a < 10) IN dbspace2
      ...
   ;

CREATE INDEX idx1 ON tb1(a);
```

For fragmented tables that use expression-based or round-robin distribution schemes, you can also create multiple partitions of a table or index within a single dbspace. This enables you to reduce the number of required dbspaces, thereby simplifying the management of dbspaces.

To create an attached index with partitions, include the partition name in your SQL statements, as shown in this example:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
     PARTITION part1 (a >=0 AND a < 5) IN dbs1,
     PARTITION part2 (a >=5 AND a < 10) IN dbs1
          ...
 ;

    CREATE INDEX idx1 ON tb1(a);
```

You can use "PARTITION BY EXPRESSION" instead of "FRAGMENT BY EXPRESSION" in CREATE TABLE, CREATE INDEX, and ALTER FRAGMENT ON INDEX statements as shown in this example:

```
ALTER FRAGMENT ON INDEX idx1 INIT PARTITION BY EXPRESSION
           PARTITION part1   (a <= 10) IN dbs1,
           PARTITION part2   (a <= 20) IN dbs1,
            PARTITION part3  (a <= 30) IN dbs1;
```

Use ALTER FRAGMENT syntax to change fragmented indexes that do not have partitions into indexes that have partitions. The syntax below shows how you might convert a fragmented index into an index that contains partitions:

```
CREATE TABLE t1 (c1 int) FRAGMENT BY EXPRESSION
     (c1=10) IN dbs1, (c1=20) IN dbs2, (c1=30) IN dbs3
CREATE INDEX ind1 ON t1 (c1) FRAGMENT BY EXPRESSION
     (c1=10) IN dbs1, (c1=20) IN dbs2, (c1=30) IN dbs3
```

```
ALTER FRAGMENT ON INDEX ind1 INIT FRAGMENT BY EXPRESSION
     PARTITION part_1 (c1=10) IN dbs1, PARTITION part_2 (c1=20) IN dbs1,
     PARTITION part_3 (c1=30) IN dbs1,
```

Creating a table or index containing partitions improves performance by enabling the database server to search more quickly and by reducing the required number of dbspaces.

The database server fragments the attached index according to the same distribution scheme as the table by using the same rule for index keys as for table data. As a result, attached indexes have the following physical characteristics:

- The number of index fragments is the same as the number of data fragments.
- Each attached index fragment resides in the same dbspace as the corresponding table data, but in a separate tblspace.
- An attached index or an index on a nonfragmented table uses 4 bytes for the row pointer for each index entry. For more information about how to estimate space for an index, see Estimating index pages on page 209.

HCL OneDB™ does not support forest of trees attached indexes.

Detached indexes

A *detached index* is an index with a separate fragmentation strategy that you set up explicitly with the CREATE INDEX statement.

The following sample SQL statements create a detached index:

```
CREATE TABLE tb1 (a int)
      FRAGMENT BY EXPRESSION
         (a <= 10) IN tabdbspc1,
         (a <= 20) IN tabdbspc2,
         (a <= 30) IN tabdbspc3;

CREATE INDEX idx1 ON tb1 (a)
      FRAGMENT BY EXPRESSION
         (a <= 10) IN idxdbspc1,
         (a <= 20) IN idxdbspc2,
         (a <= 30) IN idxdbspc3;
```

This example illustrates a common fragmentation strategy, to fragment indexes in the same way as the tables, but specify different dbspaces for the index fragments. This fragmentation strategy of putting the index fragments in different dbspaces from the table can improve the performance of operations such as backup, recovery, and so forth.

By default, all new indexes that the CREATE INDEX statement creates are detached and stored in separate tablespaces from the data unless the deprecated IN TABLE syntax is specified.

To create a detached index with partitions, include the partition name in your SQL statements, as shown in this example:

```
CREATE TABLE tb1 (a int)
      FRAGMENT BY EXPRESSION
            PARTITION part1 (a <= 10) IN dbs1,
            PARTITION part2 (a <= 20) IN dbs2,
            PARTITION part3 (a <= 30) IN dbs3;

    CREATE INDEX idx1 ON tb1 (a)
          FRAGMENT BY EXPRESSION
            PARTITION part1   (a <= 10) IN dbs1,
            PARTITION part2   (a <= 20) IN dbs2,
            PARTITION part3   (a <= 30) IN dbs3;
```

You can use the `PARTITION BY EXPRESSION` keywords instead of the `FRAGMENT BY EXPRESSION` keywords in the CREATE TABLE, CREATE INDEX, and ALTER FRAGMENT ON INDEX statements.

If you do not want to fragment the index, you can put the entire index in a separate dbspace.

You can fragment the index for any table by expression. However, you cannot explicitly create a round-robin fragmentation scheme for an index. Whenever you fragment a table using a round-robin fragmentation scheme, convert all indexes that accompany the table to detached indexes for the best performance.

Detached indexes have the following physical characteristics:

- Each detached index fragment resides in a different tblspace from the corresponding table data. Therefore, the data and index pages cannot be interleaved within the tblspace.
- Detached index fragments have their own extents and *tblspace IDs*. The tblspace ID is also known as the *fragment* ID and *partition number*. A detached index uses 8 bytes of disk space per index entry for the fragment ID and row pointer combination. For more information on how to estimate space for an index, see Estimating index pages on page 209.

Forest of trees indexes are detached indexes. They cannot be attached indexes.

The database server stores the location of each table and index fragment, along with other related information, in the **sysfragments** system catalog table. You can view the **sysfragments** system catalog table to access information about fragmented tables and indexes, including the following :

- The value in the **partn** column is the partition number or fragment id of the table or index fragment. The partition number for a detached index is different from the partition number of the corresponding table fragment.
- The value in the **strategy** column is the distribution scheme used in the fragmentation strategy.

For a complete description of column values that the **sysfragments** system catalog table contains, see the *HCL OneDB™ Guide to SQL: Reference*. For information about how to use **sysfragments** to monitor your fragments, see Monitoring fragment use on page 289.

Restrictions on indexes for fragmented tables

If the database server scans a fragmented index, multiple index fragments must be scanned and the results merged together. (The exception is if the index is fragmented according to some index-key range rule, and the scan does not cross a fragment boundary.) Because of this requirement, performance on index scans might suffer if the index is fragmented.

Because of these performance considerations, the database server places the following restrictions on indexes:

- You cannot fragment indexes by round-robin.
- You cannot fragment unique indexes by an expression that contains columns that are not in the index key.

For example, the following statement is not valid:

```
CREATE UNIQUE INDEX ia on tab1(col1)
   FRAGMENT BY EXPRESSION
      col2<10 in dbsp1,
      col2>=10 AND col2<100 in dbsp2,
      col2>100 in dbsp3;
```

Strategy for fragmenting temporary tables

You can fragment an explicit temporary table across dbspaces that reside on different disks.

You can create a temporary, fragmented table with the TEMP TABLE clause of the CREATE TABLE statement. However, you cannot alter the fragmentation strategy of fragmented temporary tables (as you can with permanent tables). The database server deletes the fragments that are created for a temporary table at the same time that it deletes the temporary table.

You can define your own fragmentation strategy for an explicit temporary table, or you can let the database server dynamically determine the fragmentation strategy.

For more information about explicit and implicit temporary tables, see your *HCL OneDB™ Administrator's Guide*.

Distribution schemes that eliminate fragments

Fragment elimination is a database server feature that reduces the number of fragments involved in a database operation. This capability can improve performance significantly and reduce contention for the disks on which fragments reside.

Fragment elimination improves both response time for a given query and concurrency between queries. Because the database server does not need to read in unnecessary fragments, I/O for a query is reduced. Activity in the LRU queues is also reduced.

If you use an appropriate distribution scheme, the database server can eliminate fragments from the following database operations:

- The fetch portion of the SELECT, INSERT, delete or update statements in SQL

  The database server can eliminate fragments when these SQL statements are optimized, before the actual search.
- Nested-loop joins

  When the database server obtains the key value from the outer table, it can eliminate fragments to search on the inner table.

Whether the database server can eliminate fragments from a search depends on two factors:

- The distribution scheme in the fragmentation strategy of the table that is being searched
- The form of the query expression (the expression in the WHERE clause of a SELECT, INSERT, delete or update statement)

Fragmentation expressions for fragment elimination

Some operators in expressions result in automatic fragment elimination.

When the fragmentation strategy is defined with any of the following operators, fragment elimination can occur for a query on the table.

```
IN
=
<
```

```
>
<=
>=
AND
OR
NOT
IS NULL (only when not combined with other expressions using AND or OR operators)
```

If the fragmentation expression uses any of the following operators, fragment elimination does not occur for queries on the table.

```
!=
IS NOT NULL
```

For examples of fragmentation expressions that allow fragment elimination, see Effectiveness of fragment elimination on page 276.

Query expressions for fragment elimination

A query expression (the expression in the WHERE clause) can consist of simple expressions, not simple expressions, and multiple expressions.

The database server considers only simple expressions or multiple simple expressions combined with certain operators for fragment elimination.

A simple expression consists of the following parts:

```
column  operator  value
```

Simple Expression Part

Description

column

Is a single column name

The database server supports fragment elimination on all column types except columns that are defined with the NCHAR, NVARCHAR, BYTE, and TEXT data types.

operator

Must be an equality or range operator

value

Must be a literal or a host variable

The following examples show simple expressions:

```
name = "Fred"
date < "08/25/2008"
value >= :my_val
```

The following examples are not simple expressions:

```
unitcost * count > 4500
price <= avg(price)
result + 3 > :limit
```

The database server considers two types of simple expressions for fragment elimination, based on the operator:

- Range expressions
- Equality expressions

Range expressions in query

The database server can handle one or two column fragment elimination on queries with any combination of five relational operators in the WHERE clause.

Range expressions use the following relational operators:

```
<
>
<=
>=
!=
```

The database server can also eliminate fragments when these range expressions are combined with the following operators:

```
AND, OR, NOT
IS NULL, IS NOT NULL
MATCH, LIKE
```

If the range expression contains MATCH or LIKE, the database server can also eliminate fragments if the string does not begin with a wildcard character. The following examples show query expressions that can take advantage of fragment elimination:

```
columna MATCH "ab*"
columna LIKE "ab%" OR columnb LIKE "ab*"
```

Equality expressions in query

The database server can handle one or multiple column fragment elimination on queries with a combination of equality operators in the WHERE clause.

Equality expressions use the following equality operators:

```
=, IN
```

The database server can also eliminate fragments when these equality expressions are combined with the following operators:

```
AND, OR
```

Effectiveness of fragment elimination

The database server cannot eliminate fragments when you fragment a table with a round-robin distribution scheme. Furthermore, not all expression-based distribution schemes give you the same fragment-elimination behavior.

The following table summarizes the fragment-elimination behavior for different combinations of expression-based distribution schemes and query expressions. Partitions in fragmented tables do not affect the fragment-elimination behavior shown in the following table.

Table 15. Fragment elimination for different types of expression-based distribution schemes and query expressions

Type of Query (WHERE clause) Expression	Nonoverlapping Fragments on a Single Column	Overlapping or Non-contiguous Fragments on a Single Column	Nonoverlapping Fragments on Multiple Columns
Range expression	Fragments can be eliminated.	Fragments cannot be eliminated.	Fragments cannot be eliminated.
Equality expression	Fragments can be eliminated.	Fragments can be eliminated.	Fragments can be eliminated.

This table shows that the distribution schemes enable fragment elimination, but the effectiveness of fragment elimination is determined by the WHERE clause of the specified query.

For example, consider a table fragmented with the following expression:

```
FRAGMENT BY EXPRESSION
100 < column_a AND column_b < 0 IN dbsp1,
100 >= column_a AND column_b < 0 IN dbsp2,
column_b >= 0 IN dbsp3
```

The database server cannot eliminate any fragments from the search if the WHERE clause has the following expression:

```
column_a = 5 OR column_b = -50
```

However, the database server can eliminate the fragment in dbspace **dbsp3** if the WHERE clause has the following expression:

```
column_b = -50
```

Furthermore, the database server can eliminate the two fragments in dbspaces **dbsp2** and **dbsp3** if the WHERE clause has the following expression:

```
column_a = 5 AND column_b = -50
```

Partitions in fragmented tables do not affect fragment-elimination behavior.

Nonoverlapping fragments on a single column

A fragmentation rule that creates nonoverlapping fragments on a single column is the preferred fragmentation rule from a fragment-elimination standpoint.

The advantage of this type of distribution scheme is that the database server can eliminate fragments for queries with range expressions as well as queries with equality expressions. You should meet these conditions when you design your fragmentation rule. Figure 37: Example of nonoverlapping fragments on a single column on page 278 gives an example of this type of fragmentation rule.

Figure 37. Example of nonoverlapping fragments on a single column

```
...
FRAGMENT BY EXPRESSION
a<=8 OR a IN (9,10) IN dbsp1,
10<a AND a<=20 IN dbsp2,
a IN (21,22,23) IN dbsp3,
a>23 IN dbsp4;
```

You can create nonoverlapping fragments using a range rule or an arbitrary rule based on a single column. You can use relational operators, as well as AND, IN, OR, or BETWEEN. Be careful when you use the BETWEEN operator. When the database server parses the BETWEEN keyword, it includes the end points that you specify in the range of values. Avoid using a REMAINDER clause in your expression. If you use a REMAINDER clause, the database server cannot always eliminate the remainder fragment.

Overlapping fragments on a single column

The fragments on a single column can be overlapping and noncontiguous. You can use any range, MOD function, or arbitrary rule that is based on a single column.

The only restriction for this category of fragmentation rule is that you base the fragmentation rule on a single column.

Figure 38: Example of overlapping fragments on a single column on page 278 shows an example of this type of fragmentation rule.

Figure 38. Example of overlapping fragments on a single column

```
...
FRAGMENT BY EXPRESSION
a<=8 OR a IN (9,10,21,22,23) IN dbsp1,
a>10 IN dbsp2;
```

If you use this type of distribution scheme, the database server can eliminate fragments on an equality search but not a range search. This distribution scheme can still be useful because all INSERT and many UPDATE operations perform equality searches.

This alternative is acceptable if you cannot use an expression that creates nonoverlapping fragments with contiguous values. For example, in cases where a table is growing over time, you might want to use a MOD function rule to keep the fragments of similar size. Expression-based distribution schemes that use MOD function rules fall into this category because the values in each fragment are not contiguous.

Improve the performance of operations that attach and detach fragments

When you use ALTER FRAGMENT ATTACH and DETACH statements to add or remove a large amount of data in a very large table, you can take steps to improve the performance of the ATTACH and DETACH operations.

The ALTER FRAGMENT DETACH statement provides a way to delete a segment of the table data rapidly. Similarly, the ALTER FRAGMENT ATTACH statement provides a way to load large amounts of data incrementally into an existing table by taking advantage of the fragmentation technology. However, the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements can take a long time to execute when the database server rebuilds indexes on the surviving table.

The database server provides performance optimizations for the ATTACH and DETACH operations of the ALTER FRAGMENT statement that reuse the indexes on the surviving tables. By eliminating the index build during the ATTACH or DETACH operation,

- this reduces the time required for the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements to execute,
- and improves the availability of the table.

The ALTER FRAGMENT operation requires exclusive access and exclusive locks on all of the tables involved in the operation. When you use the FORCE_DDL_EXEC environment option to specify a time limit for the database server to force out any transactions in other sessions that have opened (or that hold locks on) the tables involved in an ALTER FRAGMENT ON TABLE operation, also use the SET LOCK MODE TO WAIT statement to specify that number of seconds as the limit for waiting.

If the database server is unable to get exclusive access and exclusive locks on the table because of DDL transactions in concurrent sessions, the server will start rolling back the transactions that are open or that have locks on the table, until the specified time limit is reached. You might want to enable the FORCE_DDL_EXEC option and issue the SET LOCK MODE TO WAIT statement on a busy system, perhaps one that runs 24 hours a day, if you do not want to wait for transactions in concurrent sessions to close before you can alter a fragment.

Improving ALTER FRAGMENT ATTACH performance

You can take advantage of the performance optimizations for the ALTER FRAGMENT ATTACH statement if your database meets certain requirements.

To take advantage of the performance optimization, you must meet all of the following requirements:

- Formulate appropriate distribution schemes for your table and index fragments.
- Ensure that no data movement occurs between the resultant partitions due to fragment expressions.
- Update statistics for all the participating tables.
- Make the indexes on the attached tables unique if the index on the surviving table is unique.

Important: Only logging databases can benefit from the performance improvements for the ALTER FRAGMENT ATTACH statement. Without logging, the database server works with multiple copies of the same table to ensure recoverability of the data when a failure occurs. This requirement prevents reuse of the existing index fragments.

Distribution schemes for reusing indexes

You can use one of three distribution schemes that allow the attach operation of the ALTER FRAGMENT statement to reuse existing indexes.

These distributions schemes are:

- Fragmenting the index in the same way as the table
- Fragmenting the index with the same set of fragment expressions as the table
- Attaching unfragmented tables to form a fragmented table

Fragmenting the index in the same way as the table

You fragment an index in the same way as the table when you create an index without specifying a fragmentation strategy.

A fragmentation strategy is the distribution scheme and set of dbspaces in which the fragments are located. For details, see Planning a fragmentation strategy on page 260.

Example of Fragmenting the Index in the Same Way as the Table

Suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;

CREATE INDEX idx1 ON tb1(a);
```

Suppose you then create another table that is not fragmented, and you subsequently decide to attach it to the fragmented table.

```
CREATE TABLE tb2 (a int, CHECK (a >=10 AND a<15))
   IN db3;

CREATE INDEX idx2 ON tb2(a)
   IN db3;

ALTER FRAGMENT ON TABLE tb1
      ATTACH
         tb2 AS (a >= 10 and a<15) AFTER db2;
```

This attach operation can take advantage of the existing index **idx2** if no data movement occurs between the existing and the new table fragments. If no data movement occurs:

- The database server reuses index **idx2** and converts it to a fragment of index **idx1**.
- The index **idx1** remains as an index with the same fragmentation strategy as the table **tb1**.

If the database server discovers that one or more rows in the table **tb2** belong to preexisting fragments of the table **tb1**, the database server:

- Drops and rebuilds the index **idx1** to include the rows that were originally in tables **tb1** and **tb2**
- Drops the index **idx2**

For more information about how to ensure no data movement between the existing and the new table fragments, see Ensuring no data movement when you attach a fragment on page 282.

Fragmenting the index with the same distribution scheme as the table

You fragment an index with the same distribution scheme as the table when you create an index that uses the same fragment expressions as the table.

The database server determines if the fragment expressions are identical, based on the equivalency of the expression tree instead of the algebraic equivalence. For example, consider the following two expressions:

```
(col1 >= 5)
(col1 = 5 OR col1 > 5)
```

Although these two expressions are algebraically equivalent, they are not identical expressions.

Example of Fragmenting the Index with the Same Distribution Scheme as the Table

Suppose you create two fragmented tables and indexes with the following SQL statements:

```
CREATE TABLE tb1 (a INT)
   FRAGMENT BY EXPRESSION
      (a <= 10) IN tabdbspc1,
      (a <= 20) IN tabdbspc2,
      (a <= 30) IN tabdbspc3;
CREATE INDEX idx1 ON tb1 (a)
   FRAGMENT BY EXPRESSION
      (a <= 10) IN idxdbspc1,
      (a <= 20) IN idxdbspc2,
      (a <= 30) IN idxdbspc3;

CREATE TABLE tb2 (a INT CHECK a> 30 AND a<= 40)
   IN tabdbspc4;
CREATE INDEX idx2 ON tb2(a)
   IN idxdbspc4;
```

Suppose you then attach table **tb2** to table **tb1** with the following sample SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
   ATTACH tb2 AS (a <= 40);
```

The database server can eliminate the rebuild of index **idx1** for this attach operation for the following reasons:

- The fragmentation expression for index **idx1** is identical to the fragmentation expression for table **tb1**. The database server:
  - Expands the fragmentation of the index **idx1** to the dbspace **idxdbspc4**
  - Converts index **idx2** to a fragment of index **idx1**
- No rows move from one fragment to another because the CHECK constraint is identical to the resulting fragmentation expression of the attached table.

  For more information about how to ensure no data movement between the existing and the new table fragments, see Ensuring no data movement when you attach a fragment on page 282.

Attaching unfragmented tables together

You can take advantage of the performance benefits of the ALTER FRAGMENT ATTACH operation when you combine two unfragmented tables into one fragmented table.

For example, suppose you create two unfragmented tables and indexes with the following SQL statements:

```
CREATE TABLE tb1(a int) IN db1;
   CREATE INDEX idx1 ON tb1(a) in db1;
CREATE TABLE tb2(a int) IN db2;
   CREATE INDEX idx2 ON tb2(a) in db2;
```

You might want to combine these two unfragmented tables with the following sample distribution scheme:

```
ALTER FRAGMENT ON TABLE tb1
   ATTACH
      tb1 AS (a <= 100),
      tb2 AS (a > 100);
```

If no data migrates between the fragments of **tb1** and **tb2**, the database server redefines index **idx1** with the following fragmentation strategy:

```
CREATE INDEX idx1 ON tb1(a) F
   FRAGMENT BY EXPRESSION
      a <= 100 IN db1,
      a > 100 IN db2;
```

Important: This behavior results in a different fragmentation strategy for the index prior to version 7.3 and version 9.2 of the database server. In earlier versions, the ALTER FRAGMENT ATTACH statement creates an unfragmented detached index in the dbspace **db1**.

Ensuring no data movement when you attach a fragment

You can ensure there is no data movement when you attach a fragment by establishing identical check constraint expressions and verifying that fragment expressions are not overlapping.

About this task

To ensure that no data movement occurs when you attach a fragment:

1. Establish a check constraint on the attached table that is identical to the fragment expression that it will assume after the ALTER FRAGMENT ATTACH operation.
2. Define the fragments with nonoverlapping expressions.

Example

For example, you might create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;
```

```
CREATE INDEX idx1 ON tb1(a);
```

Suppose you create another table that is not fragmented, and you subsequently decide to attach it to the fragmented table.

```
CREATE TABLE tb2 (a int, check (a >=10 and a<15))
   IN db3;

CREATE INDEX idx2 ON tb2(a)
   IN db3;

ALTER FRAGMENT ON TABLE tb1
   ATTACH
      tb2 AS (a >= 10 AND a<15) AFTER db2;
```

This ALTER FRAGMENT ATTACH operation takes advantage of the existing index **idx2** because the following steps were performed in the example to prevent data movement between the existing and the new table fragment:

- The check constraint expression in the CREATE TABLE **tb2** statement is identical to the fragment expression for table **tb2** in the ALTER FRAGMENT ATTACH statement.
- The fragment expressions specified in the CREATE TABLE **tb1** and the ALTER FRAGMENT ATTACH statements are not overlapping.

Therefore, the database server preserves index **idx2** in dbspace **db3** and converts it into a fragment of index **idx1**. The index **idx1** remains as an index with the same fragmentation strategy as the table **tb1**.

Indexes on attached tables

The database server tries to reuse the indexes on the attached tables as fragments of the resultant index. However, the corresponding index on the attached table might not exist or might not be usable due to disk-format mismatches. In these cases, it might be faster to build an index on the attached tables rather than to build the entire index on the resultant table.

OneDB estimates the cost to create the whole index on the resultant table. The server then compares this cost to the cost of building the individual index fragments for the attached tables and chooses the index build with the least cost.

Automatically Gathered Statistics for New Indexes

When the CREATE INDEX statement runs successfully, with or without the ONLINE keyword, OneDB automatically gathers the following statistics for the newly created index:

- Index-level statistics, equivalent to the statistics gathered in the UPDATE STATISTICS operation in LOW mode, for all types of indexes, including B-tree, Virtual Index Interface, and functional indexes.
- Column-distribution statistics, equivalent to the distribution generated in the UPDATE STATISTICS operation in HIGH mode, for a non-opaque leading indexed column of an ordinary B-tree index. The resolution of the HIGH mode is 1.0 for a table size that is less than 1 million rows and 0.5 for higher table sizes. Tables with more than 1 million rows have a better resolution because they have more bins for statistics.

The automatically gathered distribution statistics are available to the query optimizer when it designs query plans for the table on which the new index was created.

Run UPDATE STATISTICS Before Attaching Tables

To ensure that cost estimates are correct, you should execute the UPDATE STATISTICS statement on all of the participating tables before you attach the tables. The LOW mode of the UPDATE STATISTICS statement is sufficient to derive the appropriate statistics for the optimizer to determine cost estimates for rebuilding indexes.

For more information about using the UPDATE STATISTICS statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

Example for situation when corresponding index does not exist

When a table does not have an index on a column that can serve as the fragment of the resultant index, the database server estimates the cost of building the index fragment for the column, compares this cost to rebuilding the entire index for all fragments on the resultant table, and chooses the index build with the least cost.

Suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int, b int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;
CREATE INDEX idx1 ON tb1(a);
```

Suppose you then create two more tables that are not fragmented, and you subsequently decide to attach them to the fragmented table.

```
CREATE TABLE tb2 (a int, b int,
   CHECK (a >=10 and a<15)) IN db3;
CREATE INDEX idx2 ON tb2(a) IN db3;
CREATE TABLE tb3 (a int, b int,
   CHECK (a >= 15 and a<20)) IN db4;
CREATE INDEX idx3 ON tb3(b) IN db4;

ALTER FRAGMENT ON TABLE tb1
   ATTACH tb2 AS (a >= 10 and a<15) tb3 AS (a >= 15 and a<20);
```

The three CREATE INDEX statements automatically calculate distribution statistics for the leading column of each index in HIGH mode, as well as index statistics and table statistics in LOW mode.

The only time the `UPDATE STATISTICS LOW FOR TABLE` statement is required is after a CREATE INDEX statement in a situation in which the table has other preexisting indexes, as shown in this example:

```
CREATE TABLE tb1(col1 int, col2 int);
CREATE INDEX index idx1 on tb1(col1);
  (equivalent to update stats low on table tb1)
LOAD from tb1.unl insert into tb1; (load some data)
CREATE INDEX idx2 on tb1(col2);
```

The statement `CREATE INDEX idx2 on tb1(col2)` is NOT completely equivalent to `UPDATE STATISTICS LOW FOR TABLE tb1`, because the CREATE INDEX statement does not update index- level statistics for the preexisting index called `idx1`.

In the preceding example, table **tb3** does not have an index on column **a** that can serve as the fragment of the resultant index **idx1**. The database server estimates the cost of building the index fragment for column **a** on the consumed table **tb3** and compares this cost to rebuilding the entire index for all fragments on the resultant table. The database server chooses the index build with the least cost.

Example for situation when index on table is not usable

When the index on a table is not usable, the database server estimates the cost of building the index fragment, compares this cost to rebuilding the entire index for all fragments on the resultant table, and chooses the index build with the least cost.

Suppose you create tables and indexes as in the previous section, but the index on the third table specifies a dbspace that the first table also uses. The following SQL statements show this scenario:

```
CREATE TABLE tb1(a int, b int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;
CREATE INDEX idx1 ON tb1(a);
CREATE TABLE tb2 (a int, b int, check (a >=10 and a<15))
   IN db3;
CREATE INDEX idx2 ON tb2(a)
   IN db3;

CREATE TABLE tb3 (a int, b int, check (a >= 15 and a<20))
   IN db4;
CREATE INDEX idx3 ON tb3(a)
   IN db2 ;
```

This example creates the index **idx3** on table **tb3** in the dbspace **db2**. As a result, index **idx3** is not usable because index **idx1** already has a fragment in the dbspace **db2,** and the fragmentation strategy does not allow more than one fragment to be specified in a given dbspace.

Again, the database server estimates the cost of building the index fragment for column **a** on the consumed table **tb3** and compares this cost to rebuilding the entire index **idx1** for all fragments on the resultant table. Then the database server chooses the index build with the least cost.

Improving ALTER FRAGMENT DETACH performance

You can improve the performance of ALTER FRAGMENT DETACH statements by formulating appropriate distribution schemes for your table and index fragments and by eliminating the index build during the execution of ALTER FRAGMENT DETACH statements.

To eliminate the index build during execution of the ALTER FRAGMENT DETACH statement, use one of the following fragmentation strategies:

- Fragment the index in the same way as the table.
- Fragment the index with the same distribution scheme as the table.

Important: Only logging databases can benefit from the performance improvements for the ALTER FRAGMENT DETACH statement. Without logging, the database server works with multiple copies of the same table to ensure recoverability of the data when a failure occurs. This requirement prevents reuse of the existing index fragments.

Fragmenting the index in the same way as the table

You fragment an index in the same way that you fragment the table when you create a fragmented table and subsequently create an index without specifying a fragmentation strategy, unless the distribution scheme is round-robin and automatic location is enabled. Indexes on tables that use the round-robin distribution scheme are not fragmented when the AUTOLOCATE configuration parameter or environment option is set to a positive integer. You fragment an index in the same way that you fragment the table when you create a fragmented table and subsequently create an index without specifying a fragmentation strategy. The database server automatically creates an attached index when you first fragment a table.

For example, suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2,
      (a >=10 AND a <15) IN db3;
CREATE INDEX idx1 ON tb1(a);
```

The database server fragments the index keys into dbspaces **db1**, **db2**, and **db3** with the same column **a** value ranges as the table because the CREATE INDEX statement does not specify a fragmentation strategy.

Suppose you then decide to detach the data in the third fragment with the following SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
   DETACH db3 tb3;
```

Because the fragmentation strategy of the index is the same as the table, the ALTER FRAGMENT DETACH statement does not rebuild the index after the detach operation. The database server drops the fragment of the index in dbspace **db3**, updates the system catalog tables, and eliminates the index build.

Fragmenting the index using same distribution scheme as the table

You fragment an index with the same distribution scheme as the table when you create the index that uses the same fragment expressions as the table.

A common fragmentation strategy is to fragment indexes in the same way as the tables but to specify different dbspaces for the index fragments. This fragmentation strategy of putting the index fragments into different dbspaces from the table can improve the performance of operations such as backup and recovery.

For example, suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int, b int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2,
      (a >=10 AND a <15) IN db3;
```

```
CREATE INDEX idx1 on tb1(a)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a< 5) IN db4,
      (a >=5 AND a< 10) IN db5,
      (a >=10 AND a<15) IN db6;
```

Suppose that you then decide to detach the data in the third fragment with the following SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
   DETACH db3 tb3;
```

Because the distribution scheme of the index is the same as the table, the ALTER FRAGMENT DETACH statement does not rebuild the index after the detach operation. The database server drops the fragment of the index in dbspace **db3**, updates the system catalog tables, and eliminates the index build.

Forcing out transactions when altering table fragments

You can enable the server to force out transactions that have opened or hold locks on the target table of an ALTER FRAGMENT ON TABLE operation in a logging database. Users holding the DBA access privilege can do this by enabling the FORCE_DDL_EXEC session environment option of the SET ENVIRONMENT statement.

About this task

You might want to do this on a busy system, perhaps one that runs 24 hours a day, if you do not want to wait for sessions to close before you alter a fragment.

Be aware, however, that by forcing out concurrent transactions to avoid waiting for locks to be released, the database server closes the Update cursors and rolls back the transactions of other users.

Prerequisites:

- You must be user **informix** or hold DBA access privileges on the database.
- The table must be in a database that supports transaction logging.

To force out concurrent transactions of other sessions when altering a table fragment:

1. Set the FORCE_DDL_EXEC environment option of the SET ENVIRONMENT statement to one of the following values:
   - `ON`, `on` , `'1'`, or `"1"` to enable the server to force out transactions that are open or have a lock on the table when an ALTER FRAGMENT ON TABLE statement is issued, until the server obtains a lock and exclusive access to the table.
   - A positive integer that represents an amount of time in seconds. The numeric value enables the server to force out transactions until the server gets exclusive access and exclusive locks on the table, or until the specified time limit. If the server cannot force out transactions by the specified number of seconds, the server stops attempting to force out the transactions, and the ALTER FRAGMENT statement waits for the locks to be released when the concurrent transactions are committed or rolled back.

For example, to enable the FORCE_DDL_EXEC environment option to operate for 100 seconds when an ALTER FRAGMENT ON TABLE statement is issued, specify:

```
SET ENVIRONMENT FORCE_DDL_EXEC '100';
```

2. Set the lock mode to wait to ensure that the server will wait a specified amount of time before forcing out any transactions.

   For example, to set the lock mode to wait for 20 seconds, specify:

   ```
   SET LOCK MODE TO WAIT "20";
   ```

   For more information, see Setting the lock mode to wait on page 244.

3. Run an ALTER FRAGMENT ON TABLE statement, for example, to attach, detach, modify, add, or drop the fragment.

Example

The following SQL statements perform these actions:

- enable the FORCE_DDL_EXEC session environment option for 100 seconds,
- set the database server to wait up to 25 seconds for locks to be released,
- and change the interval size and storage location of range fragment p2 of table tabF:

```
SET ENVIRONMENT FORCE_DDL_EXEC '100';
SET LOCK MODE TO WAIT 25;
ALTER FRAGMENT ON TABLE tabF MODIFY
   PARTITION p2 TO PARTITION p2 VALUES < 500 IN dbs0;
```

Attention:

While the ALTER FRAGMENT statement above is running, other transactions that attempt to access rows in table tabF are at risk of being forced out, if their Update cursor holds locks on rows in fragment p2.

After a transaction is rolled back because the FORCE_DDL_EXEC session environment option is enabled by another session, the database server returns this error to the session whose transaction failed:

```
-458  Long transaction aborted.
```

The concurrent transaction failing with error `-458` was not necessarily "long," but it had not yet been committed after opening or holding locks on the same table that the ALTER FRAGMENT statement in this example was modifying.

What to do next

After you complete an ALTER FRAGMENT ON TABLE operation with the FORCE_DDL_EXEC session environment option enabled, you can turn the FORCE_DDL_EXEC session environment option off. For example, specify:

```
SET ENVIRONMENT FORCE_DDL_EXEC OFF;
```

Related information

FORCE_DDL_EXEC session environment option on page

Monitoring fragment use

Once you determine a fragmentation strategy, you can monitor fragmentation.

You can monitor fragmentation in the following ways:

- Run individual **onstat** utility commands to capture information about specific aspects of a running query.
- Run a SET EXPLAIN statement before you run a query to write the query plan to an output file.

Monitoring fragmentation with the onstat -g ppf command

With the **onstat -g ppf** command, you can view partition information and monitor the I/O activity to verify your strategy and determine whether the I/O is balanced across fragments.

About this task

The **onstat -g ppf** output includes the number of read-and-write requests sent to each fragment that is currently open. Because a request can trigger multiple I/O operations, these requests do not indicate how many individual disk I/O operations occur, but you can get a good idea of the I/O activity from the displayed columns.

The `brfd` column in the output displays the number of buffer reads in pages. (Each buffer can contain one page.) This information is useful if you need to monitor the time a query takes to execute. Typically query execution time has a strong dependency on the number of required buffer reads. If the size of client-server buffering is small and your database contains TEXT data, query execution time can involve significantly more buffer reads, because the server reads the prior TEXT data.

The **onstat -g ppf** output by itself does not identify the table in which a fragment is located. To determine the table for the fragment, join the `partnum` column in the output to the **partnum** column in the **sysfragments** system catalog table. The **sysfragments** table displays the associated **table id**. You can also find the table name for the fragment by joining the **table id** column in **sysfragments** to the **table id** column in **systables**.

To determine the table name in **onstat -g ppf** output:

1. Obtain the value in the **partnum** field of the **onstat -g ppf** output.
2. Join the **tabid** column in the **sysfragments** system catalog table with the **tabid** column in the **systables** system catalog table to obtain the table name from **systables**.

   Use the **partnum** field value that you obtain in step 1 in the SELECT statement.

```
SELECT a.tabname FROM systables a, sysfragments b
   WHERE a.tabid = b.tabid
      AND partn = partnum_value;
```

Monitoring fragmentation with SET EXPLAIN output

When the table is fragmented, the output of the SET EXPLAIN ON statement shows which table or index the database server scans to execute the query.

The SET EXPLAIN output identifies the fragments with a fragment number. The fragment numbers are the same as those contained in the **partn** column in the **sysfragments** system catalog table.

The following example of partial SET EXPLAIN output shows a query that takes advantage of fragment elimination and scans two fragments in table **t1**:

```
QUERY:
------
SELECT * FROM t1 WHERE c1 > 12

Estimated Cost: 3
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Serial, fragments: 1, 2)

   Filters: informix.t1.c1 > 12
```

If the optimizer must scan all fragments (that is, if it is unable to eliminate any fragment from consideration), the SET EXPLAIN output displays fragments: `ALL`. In addition, if the optimizer eliminates all the fragments from consideration (that is, none of the fragments contain the queried information), the SET EXPLAIN output displays fragments: `NONE`.

For information about how the database server eliminates a fragment from consideration, see Distribution schemes that eliminate fragments on page 274.

For more information about the SET EXPLAIN ON statement, see Report that shows the query plan chosen by the optimizer on page 299.

Queries and the query optimizer

These topics describe query plans, explain how the database server manages query optimization, and discuss factors that you can use to influence the query plan. These topics also describe performance considerations for SPL routines, the UDR cache, and triggers.

The parallel database query (PDQ) features in the database server provide the largest potential performance improvements for a query. Parallel database query (PDQ) on page 345 describes PDQ and the Memory Grant Manager (MGM) and explains how to control resource use by queries.

PDQ provides the most substantial performance gains if you fragment your tables as described in Fragmentation guidelines on page 260.

Improving individual query performance on page 365 explains how to improve the performance of specific queries.

Data warehouse queries and performance issues related to dimensional databases are described in the *HCL OneDB™ Data Warehouse Guide*.

Related information

Performance tuning dimensional databases on page

The query plan

The query optimizer evaluates the different ways in which a query might be performed and determines the best way to select the requested data. During this evaluation, the optimizer formulates a *query plan* to fetch the data rows that are required to process a query.

For example, when evaluating the different ways in which a query might be performed, the optimizer must determine whether indexes should be used. If the query includes a join, the optimizer must determine the join plan (hash or nested loop) and the order in which tables are evaluated or joined.

The following topics describe the components of a query plan and show examples of query plans.

The access plan

The way that the optimizer chooses to read a table is called an *access plan*. The simplest method to access a table is to read it sequentially, which is called a *table scan*. The optimizer chooses a table scan when most of the table must be read or the table does not have an index that is useful for the query.

The optimizer can also choose to access the table by an index. If the column in the index is the same as a column in a filter of the query, the optimizer can use the index to retrieve only the rows that the query requires. The optimizer can use a *key-only index scan* if the columns requested are within one index on the table. The database server retrieves the needed data from the index and does not access the associated table.

Important: The optimizer does not choose a key-only scan for a VARCHAR column. If you want to take advantage of key-only scans, use the ALTER TABLE with the MODIFY clause to change the column to a CHAR data type.

The optimizer compares the cost of each plan to determine the best one. The database server derives cost from estimates of the number of I/O operations required, calculations to produce the results, rows accessed, sorting, and so forth.

The join plan

When a query contains more than one table, OneDB joins the tables using filters in the query. The way that the optimizer chooses to join the tables is the *join plan*.

In the following query, the customer and orders table are joined by the `customer.customer_num = orders.customer_num` filter:

```
SELECT * from customer, orders
WHERE customer.customer_num = orders.customer_num
AND customer.lname = "Higgins";
```

The join method can be a nested-loop join or a hash join.

Because of the nature of hash joins, an application with isolation level set to Repeatable Read might temporarily lock all the records in tables that are involved in the join, including records that fail to qualify the join. This situation leads to decreased concurrency among connections. Conversely, nested-loop joins lock fewer records but provide reduced performance when a large number of rows are accessed. Thus, each join method has advantages and disadvantages.

Nested-loop join

In a nested-loop join, the database server scans the first, or *outer table*, and then joins each of the rows that pass table filters to the rows found in the second, or *inner table*.

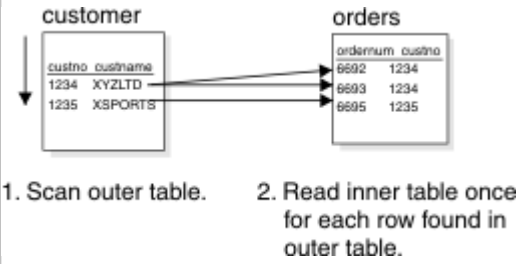
Figure 39: Nested-loop join on page 292 shows tables and rows, and the order they are read, for query:

```
SELECT * FROM customer, orders
WHERE customer.customer_num=orders.customer_num
AND order_date>"01/01/2007";
```

The database server accesses an outer table by an index or by a table scan. The database server applies any table filters first. For each row that satisfies the filters on the outer table, the database server reads the inner table to find a match.

The database server reads the inner table once for every row in the outer table that fulfills the table filters. Because of the potentially large number of times that the inner table can be read, the database server usually accesses the inner table by an index.

Figure 39. Nested-loop join



If the inner table does not have an index, the database server might construct an *autoindex* at the time of query execution. The optimizer might determine that the cost to construct an *autoindex* at the time of query execution is less than the cost to scan the inner table for each qualifying row in the outer table.

If the optimizer changes a subquery to a nested-loop join, it might use a variation of the nested-loop join, called a *semi join*. In a semi join, the database server reads the inner table only until it finds a match. In other words, for each row in the outer table, the inner table contributes at most one row. For more information on how the optimizer handles subqueries, see Query plans for subqueries on page 306.

Hash join

The optimizer usually uses a hash join when at least one of the two join tables does not have an index on the join column or when the database server must read a large number of rows from both tables. No index and no sorting is required when the database server performs a hash join.

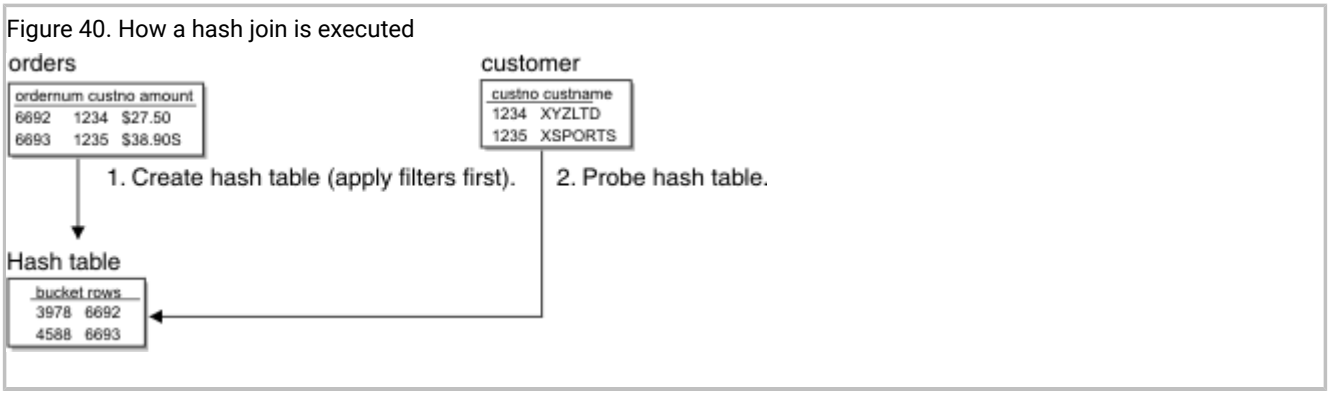
A hash join consists of two activities: first building the hash table (*build* phase) and then probing the hash table (*probe* phase). Figure 40: How a hash join is executed on page 293 shows the hash join in detail.

In the build phase, the database server reads one table and, after it applies any filters, creates a hash table. Think of a hash table conceptually as a series of *buckets*, each with an address that is derived from the key value by applying a hash function. The database server does not sort keys in a particular hash bucket.

Smaller hash tables can fit in the virtual portion of database server shared memory. The database server stores larger hash files on disk in the dbspace specified by the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable.

In the probe phase, the database server reads the other table in the join and applies any filters. For each row that satisfies the filters on the table, the database server applies the hash function on the key and probes the hash table to find a match.

Figure 40. How a hash join is executed



Join order

The order that tables are joined in a query is extremely important. A poor join order can cause query performance to decline noticeably.

The following SELECT statement calls for a three-way join:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
```

The optimizer can choose one of the following join orders:

- Join **customer** to **orders**. Join the result to **items**.
- Join **orders** to **customer**. Join the result to **items**.
- Join **customer** to **items**. Join the result to **orders**.
- Join **items** to **customer**. Join the result to **orders**.
- Join **orders** to **items**. Join the result to **customer**.
- Join **items** to **orders**. Join the result to **customer**.

For an example of how the database server executes a plan according to a specific join order, see Example of query-plan execution on page 294.

Example of query-plan execution

This topic contains an example of a query with a SELECT statement that calls for a three-way join and describes one possible query plan.

The following SELECT statement calls for a three-way join:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
```

Assume also that no indexes are on any of the three tables. Suppose that the optimizer chooses the **customer-orders-items** path and the nested-loop join for both joins (in reality, the optimizer usually chooses a hash join for two tables without indexes on the join columns). Figure 41: A query plan in pseudocode on page 294 shows the *query plan*, expressed in pseudocode. For information about interpreting query plan information, see Report that shows the query plan chosen by the optimizer on page 299.

Figure 41. A query plan in pseudocode

```
for each row in the customer table do:
   read the row into C
   for each row in the orders table do:
      read the row into O
      if O.customer_num = C.customer_num then
         for each row in the items table do:
            read the row into I
            if I.order_num = O.order_num then
               accept the row and send to user
            end if
         end for
      end if
   end for
end for
```

This procedure reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once for each row of the **customer** table
- All rows of the **items** table once for each row of the **customer-orders** pair

This example does not describe the only possible query plan. Another plan merely reverses the roles of **customer** and **orders**: for each row of **orders**, it reads all rows of **customer**, looking for a matching **customer_num**. It reads the same number of rows in a different order and produces the same set of rows in a different order. In this example, no difference exists in the amount of work that the two possible query plans need to do.

Example of a join with column filters

The presence of a *column filter* can change the query plan. A column filter is a WHERE expression that reduces the number of rows that a table contributes to a join.

The following example shows the query described in Example of query-plan execution on page 294 with a filter added:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
      AND O.paid_date IS NULL
```

The expression `O.paid_date` IS NULL filters out some rows, reducing the number of rows that are used from the **orders** table. Consider a plan that starts by reading from **orders**. Figure 42: Query plan that uses a column filter on page 295 displays this sample plan in pseudocode.

Figure 42. Query plan that uses a column filter

```
for each row in the orders table do:
   read the row into O
   if O.paid_date is null then
      for each row in the customer table do:
         read the row into C
         if O.customer_num = C.customer_num then
            for each row in the items table do:
               read the row into I
               if I.order_num = O.order_num then
                  accept row and return to user
               end if
            end for
         end if
      end for
   end if
end for
```

Let *pdnull* represent the number of rows in **orders** that pass the filter. It is the value of **COUNT(*)** that results from the following query:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

If one customer exists for every order, the plan in Figure 42: Query plan that uses a column filter on page 295 reads the following rows:

- All rows of the **orders** table once
- All rows of the **customer** table, *pdnull* times
- All rows of the **items** table, *pdnull* times

Figure 43: The alternative query plan in pseudocode on page 296 shows an alternative execution plan that reads from the **customer** table first.

Figure 43. The alternative query plan in pseudocode

```
for each row in the customer table do:
   read the row into C
   for each row in the orders table do:
      read the row into O
      if O.paid_date is null and
           O.customer_num = C.customer_num then
         for each row in the items table do:
            read the row into I
            if I.order_num = O.order_num then
               accept row and return to user
            end if
         end for
      end if
   end for
```

Because the filter is not applied in the first step that Figure 43: The alternative query plan in pseudocode on page 296 shows, this plan reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once for every row of **customer**
- All rows of the **items** table, *pdnull* times

The query plans in Figure 42: Query plan that uses a column filter on page 295 and Figure 43: The alternative query plan in pseudocode on page 296 produce the same output in a different sequence. They differ in that one reads a table *pdnull* times, and the other reads a table SELECT COUNT`(*)` FROM `customer` times. By choosing the appropriate plan, the optimizer can save thousands of disk accesses in a real application.

Example of a join with indexes

The presence of indexes and constraints in query plans provides the optimizer with options that can greatly improve query-execution time.

This topic shows the outline of a query plan that differs from query shown in Example of a join with column filters on page 294, because it is constructed using indexes.

Figure 44. Query plan with indexes

```
for each row in the customer table do:
   read the row into C
   look up C.customer_num in index on orders.customer_num
   for each matching row in the orders index do:
      read the table row for O
      if O.paid_date is null then
         look up O.order_num in index on items.order_num
         for each matching row in the items index do:
            read the row for I
            construct output row and return to user
         end for
      end if
   end for
end for
```

The keys in an index are sorted so that when the database server finds the first matching entry, it can read any other rows with identical keys without further searching, because they are located in physically adjacent positions. This query plan reads only the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once (because each order is associated with only one customer)
- Only rows in the **items** table that match *pdnull* rows from the **customer-orders** pairs

This query plan achieves a great reduction in cost compared with plans that do not use indexes. An inverse plan, reading **orders** first and looking up rows in the **customer** table by its index, is also feasible by the same reasoning.

The physical order of rows in a table also affects the cost of index use. To the degree that a table is ordered relative to an index, the overhead of accessing multiple table rows in index order is reduced. For example, if the **orders** table rows are physically ordered according to the customer number, multiple retrievals of orders for a given customer would proceed more rapidly than if the table were ordered randomly.

In some cases, using an index might incur additional costs. For more information, see Index lookup costs on page 318.

Query plans that include an index self-join path

An *index self-join* is a type of index scan that you can think of as a union of many small index scans, each one with a single unique combination of lead-key columns and filters on non-lead-key columns.

The union of small index scans results in an access path that uses only subsets of the full range of a composite index. The table is logically joined to itself, and the more selective non-leading index keys are applied as index-bound filters to each unique combination of the leading key values.

An index self-join is beneficial for situations in which:

- The lead key of an index has many duplicates, and
- Predicates on the lead key are not selective, but predicates on the non-leading index keys are selective.

The query in Figure 45: SET EXPLAIN output for a query with an index self-join path on page 298 shows the SET EXPLAIN output for a query plan that includes an index self-join path.

Figure 45. SET EXPLAIN output for a query with an index self-join path

```
QUERY:
------
SELECT a.c1,a.c2,a.c3 FROM tab1 a WHERE (a.c3 >= 100103) AND
     (a.c3 <= 100104) AND (a.c1 >= 'PICKED      ') AND
     (a.c1 <= 'RGA2        ') AND (a.c2 >= 1) AND (a.c2 <= 7)
     ORDER BY 1, 2, 3

Estimated Cost: 155
Estimated # of Rows Returned: 1
  1) informix.a: INDEX PATH
    (1) Index Keys: c1 c2 c3 c4 c5   (Key-Only)  (Serial, fragments: ALL)
        Index Self Join Keys (c1 c2 )
          Lower bound: informix.a.c1 >= 'PICKED      ' AND (informix.a.c2 >= 1 )
          Upper bound: informix.a.c1 <= 'RGA2        ' AND (informix.a.c2 <= 7 )
        Lower Index Filter: (informix.a.c1 = informix.a.c1 AND
          informix.a.c2 = informix.a.c2 ) AND informix.a.c3 >= 100103
        Upper Index Filter: informix.a.c3 <= 100104
        Index Key Filters:  (informix.a.c2 <= 7 ) AND
                            (informix.a.c2 >= 1 )
```

In Figure 45: SET EXPLAIN output for a query with an index self-join path on page 298, an index exists on columns c1, c2, c3, c4, and c5. The optimizer chooses c1 and c2 as lead keys, which implies that columns c1 and c2 have many duplicates. Column c3 has few duplicates and thus the predicates on column c3 (`c3 >= 100103` and `c3 <= 100104`) have good selectivity.

As Figure 45: SET EXPLAIN output for a query with an index self-join path on page 298 shows, an index self-join path is a self-join of two index scans using the same index. The first index scan retrieves each unique value for lead key columns, which are c1 and c2. The unique value of c1 and c2 is then used to probe the second index scan, which also uses predicates on column c3. Because predicates on column c3 have good selectivity:

- The index scan on the inner side of the nested-loop join is very efficient, retrieving only the few rows that satisfy the c3 predicates.
- The index scan does not retrieve extra rows.

Thus, for each unique value of c1 and c2, an efficient index scan on c1, c2 and c3 occurs.

The following lines in the example indicate that the optimizer has chosen an index self join path for this table, with columns c1 and c2 as the lead keys for the index self-join path:

```
Index Self Join Keys (c1 c2 )
          Lower bound: informix.a.c1 >= 'PICKED      ' AND (informix.a.c2 >= 1 )
          Upper bound: informix.a.c1 <= 'RGA2        ' AND (informix.a.c2 <= 7 )
```

The example shows the bounds for columns c1 and c2, which you can conceive of as the bounds for the index scan to retrieve the qualified leading keys of the index.

The following information in the example shows the self-join:

```
(informix.a.c1 = informix.a.c1 AND informix.a.c2 = informix.a.c2 )
```

This information represents the inner index scan. For lead key columns c1 and c2 the self- join predicate is used, indicating the value of c1 and c2 comes from the outer index scan. The predicates on column c3 serve as an index filter that makes the inner index scan efficient.

Regular index scans do not use filters on column c3 to position the index scan, because the lead key columns c1 and c2 do not have equality predicates.

The INDEX_SJ directive forces an index self-join path using the specified index, or choosing the least costly index in a list of indexes, even if data distribution statistics are not available for the leading index key columns. The AVOID_INDEX_SJ directive prevents a self-join path for the specified index or indexes. Also see Access-method directives on page 332 and the *HCL OneDB™ Guide to SQL: Syntax*.

Query plan evaluation

The optimizer considers all query plans by analyzing factors such as disk I/O and CPU costs.

The optimizer constructs all feasible plans simultaneously using a bottom-up, breadth-first search strategy. That is, the optimizer first constructs all possible join pairs. It eliminates the more expensive pair of any *redundant* pair. (Redundant pairs are join pairs that contain the same tables and produce the same set of rows as another join pair.)

For example, if neither join specifies an ordered set of rows by using the ORDER BY or GROUP BY clauses of the SELECT statement, the join pair (A x B) is redundant with respect to (B x A).

If the query uses additional tables, the optimizer joins each remaining pair to a new table to form all possible join triplets, eliminating the more expensive of redundant triplets and so on for each additional table to be joined. When a non-redundant set of possible join combinations has been generated, the optimizer selects the plan that appears to have the lowest execution cost.

Report that shows the query plan chosen by the optimizer

Any user who runs a query can use the SET EXPLAIN statement or the EXPLAIN directive to display the query plan that the optimizer chooses.

For information about how to specify the directives, see EXPLAIN directives on page 336. The user enters the SET EXPLAIN ON statement or the SET EXPLAIN ON AVOID_EXECUTE statement before the SQL statement for the query, as the following example shows.

```
SET EXPLAIN ON AVOID_EXECUTE;
SELECT * FROM customer, orders
WHERE customer.customer_num = orders.customer_num
   AND customer.lname = "Higgins";
```

If a user does not have any access to SQL code source, the Database Administrator can set dynamically the SET EXPLAIN using the onmode -Y command.

After the database server executes the SET EXPLAIN ON statement or sets dynamically the SET EXPLAIN with onmode -Y command, the server writes an explanation of each query plan to a file for subsequent queries that the user enters.

Related information

The explain output file on page 300

Query statistics section provides performance debugging information on page 301

SET EXPLAIN statement on page

Using the FILE TO option on page

Default name and location of the explain output file on UNIX on page

Default name and location of the output file on Windows on page

Report that shows the query plan chosen by the optimizer on page 299

Enabling external directives on page 344

onmode -Y: Dynamically change SET EXPLAIN on page

onmode and Y arguments: Change query plan measurements for a session (SQL administration API) on page

The explain output file

The SET EXPLAIN statement enables or disables recording measurements of queries in the current session, including the plan of the query optimizer, an estimate of the number of rows returned, and the relative cost of the query. The measurements appear in an output file.

When you run the onmode -Y command to turn on dynamic SET EXPLAIN, the output is displayed in a new explain output file. If a file from the SET EXPLAIN statement exists, the database server stops using it, and instead uses the file created by onmode -Y until the administrator turns off dynamic SET EXPLAIN for the session.

The output file specifies if external directives are in effect.

The following codes in the Query Statistics section of the explain output file provide information about external tables:

- `xlcnv` identifies an operation that is loading data from an external table and inserting the data into a base table. Here `x` = external table, `l` = loading, and `cnv` = converter
- `xucnv` identifies an operation that is unloading data from an external table and inserting the data into a base table. Here `x` = external table, `u` = unloading, and `cnv` = converter

The Query Statistics section of the explain output file is a useful resource for debugging performance problems. See Query statistics section provides performance debugging information on page 301.

Related information

SET EXPLAIN statement on page

Using the FILE TO option on page

Default name and location of the explain output file on UNIX on page

Default name and location of the output file on Windows on page

Report that shows the query plan chosen by the optimizer on page 299

Query statistics section provides performance debugging information on page 301

Enabling external directives on page 344

onmode -Y: Dynamically change SET EXPLAIN on page

onmode and Y arguments: Change query plan measurements for a session (SQL administration API) on page

Query statistics section provides performance debugging information

If the EXPLAIN_STAT configuration parameter is enabled, a Query Statistics section appears in the explain output file that the SET EXPLAIN statement of SQL and the **onmode -Y *session_id*** command displays.

The Query Statistics section of the explain output file shows the estimated number of rows that the query plan expects to return, the actual number of returned rows, and other information about the query. You can use this information, which provides an indication of the overall flow of the query plan and how many rows flow through each stage of the query, to debug performance problems.

The following example shows query statistics in SET EXPLAIN output. If the estimated and actual number of rows scanned or joined are quite different, the statistics on those tables might be old and should be updated.

Figure 46. Query statistics in SET EXPLAIN output

```
select * from tab1, tab2 where tab1.c1 = tab2.c1 and tab1.c3 between 0 and 15

Estimated Cost: 104
Estimated # of Rows Returned: 69

  1) zelaine.tab2: SEQUENTIAL SCAN

  2) zelaine.tab1: INDEX PATH

    (1) Index Keys: c1 c3   (Serial, fragments: ALL)
        Lower Index Filter: (zelaine.tab1.c1 = zelaine.tab2.c1
                             AND zelaine.tab1.c3 >= 0 )
        Upper Index Filter: zelaine.tab1.c3 <= 15
NESTED LOOP JOIN

Query statistics:
-----------------

  Table map :
  ----------------------------
  Internal name     Table name
  ----------------------------
  t1                tab2
  t2                tab1

  type     table  rows_prod  est_rows  rows_scan  time       est_cost
  ------------------------------------------------------------------
  scan     t1     50         50        50         00:00:00   4

  type     table  rows_prod  est_rows  rows_scan  time       est_cost
  ------------------------------------------------------------------
  scan     t2     67         69        4          00:00:00   2

  type     rows_prod  est_rows  time       est_cost
  ------------------------------------------------
  nljoin   67         70        00:00:00   104
```

Related information

The explain output file on page 300

SET EXPLAIN statement on page

Using the FILE TO option on page

Default name and location of the explain output file on UNIX on page

Default name and location of the output file on Windows on page

Report that shows the query plan chosen by the optimizer on page 299

Sample query plan reports on page 303

Enabling external directives on page 344

onmode -Y: Dynamically change SET EXPLAIN on page

onmode and Y arguments: Change query plan measurements for a session (SQL administration API) on page

Sample query plan reports

The topics in this section describe sample query plans that you might want to display when analyzing the performance of different kinds of queries.

Related information

Query statistics section provides performance debugging information on page 301

Single-table query

This topic shows sample SET EXPLAIN output for a simple query and a complex query on a single table.

Figure 47: Partial SET EXPLAIN output for a simple query on page 303 shows SET EXPLAIN output for a simple query.

Figure 47. Partial SET EXPLAIN output for a simple query

```
QUERY:
------
SELECT fname, lname, company FROM customer

Estimated Cost: 2
Estimated # of Rows Returned: 28

  1) virginia.customer: SEQUENTIAL SCAN
```

Figure 48: Partial SET EXPLAIN output for a complex query on page 304 shows SET EXPLAIN output for a complex query on the **customer** table.

Figure 48. Partial SET EXPLAIN output for a complex query

```
QUERY:
------
SELECT fname, lname, company FROM customer
WHERE company MATCHES 'Sport*' AND
    customer_num BETWEEN 110 AND 115
ORDER BY lname

Estimated Cost: 1
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

  1) virginia.customer: INDEX PATH

        Filters: virginia.customer.company MATCHES 'Sport*'

    (1) Index Keys: customer_num   (Serial, fragments: ALL)
        Lower Index Filter: virginia.customer.customer_num >= 110
        Upper Index Filter: virginia.customer.customer_num <= 115
```

The following output lines in Figure 48: Partial SET EXPLAIN output for a complex query on page 304 show the scope of the index scan for the second query:

- Lower Index Filter: virginia.customer.customer_num >= 110

  Start the index scan with the index key value of `110`.
- Upper Index Filter: virginia.customer.customer_num <= 115

  Stop the index scan with the index key value of `115`.

Multitable query

This topic shows sample SET EXPLAIN output for a multiple-table query.

Figure 49. Partial SET EXPLAIN output for a multi-table query

```
QUERY:
------
SELECT C.customer_num, O.order_num, SUM (I.total_price)
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
AND O.order_num = I.order_num
GROUP BY C.customer_num, O.order_num

Estimated Cost: 78
Estimated # of Rows Returned: 1
Temporary Files Required For: Group By

  1) virginia.o: SEQUENTIAL SCAN

  2) virginia.c: INDEX PATH

    (1) Index Keys: customer_num   (Key-Only)  (Serial, fragments: ALL)
        Lower Index Filter:
          virginia.c.customer_num = virginia.o.customer_num
NESTED LOOP JOIN

  3) virginia.i: INDEX PATH

    (1) Index Keys: order_num   (Serial, fragments: ALL)
        Lower Index Filter: virginia.o.order_num = virginia.i.order_num
NESTED LOOP JOIN
```

The SET EXPLAIN output lists the order in which the database server accesses the tables and the access plan to read each table. The plan in Figure 49: Partial SET EXPLAIN output for a multi-table query on page 305 indicates that the database server is to perform the following actions:

1. The database server is to read the **orders** table first.

   Because no filter exists on the **orders** table, the database server must read all rows. Reading the table in physical order is the least expensive approach.

2. For each row of **orders**, the database server is to search for matching rows in the **customer** table.

   The search uses the index on **customer_num**. The notation `Key-Only` means that only the index need be read for the **customer** table because only the **c.customer_num** column is used in the join and the output, and the column is an index key.

3. For each row of **orders** that has a matching **customer_num**, the database server is to search for a match in the **items** table using the index on **order_num**.

Key-first scan

This topic shows a sample query that uses a *key-first scan*, which is an index scan that uses keys other than those listed as lower and upper index filters.

Figure 50. Partial SET EXPLAIN output for a key-first scan

```
create index idx1 on tab1(c1, c2);

select * from tab1 where (c1 > 0) and ( (c2 = 1) or (c2 = 2))
Estimated Cost: 4
Estimated # of Rows Returned: 1

1) pubs.tab1: INDEX PATH

    (1) Index Keys: c1 c2 (Key-First) (Serial, fragments: ALL)
   Lower Index Filter: pubs.tab1.c1 > 0
   Index Key Filters:  (pubs.tab1.c2 = 1 OR pubs.tab1.c2 = 2)
```

Even though in this example the database server must eventually read the row data to return the query results, it attempts to reduce the number of possible rows by applying additional key filters first. The database server uses the index to apply the additional filter, `c2 = 1 OR c2 = 2`, before it reads the row data.

Query plans for subqueries

The optimizer can change a subquery to a join automatically if the join provides a lower cost.

For example, Figure 51: Partial SET EXPLAIN output for a flattened subquery on page 306 sample output of the SET EXPLAIN ON statement shows that the optimizer changes the table in the subquery to be the inner table in a join.

Figure 51. Partial SET EXPLAIN output for a flattened subquery

```
QUERY:
------
SELECT company, fname, lname, phone
FROM customer c
WHERE EXISTS(
   SELECT customer_num FROM cust_calls u
      WHERE c.customer_num = u.customer_num)

Estimated Cost: 6
Estimated # of Rows Returned: 7

  1) virginia.c: SEQUENTIAL SCAN

  2) virginia.u: INDEX PATH  (First Row)

    (1) Index Keys: customer_num call_dtime   (Key-Only)
                                              (Serial, fragments: ALL)
        Lower Index Filter: virginia.c.customer_num = virginia.u.customer_num
NESTED LOOP JOIN  (Semi Join)
```

For more information about the SET EXPLAIN ON statement, see Report that shows the query plan chosen by the optimizer on page 299.

When the optimizer changes a subquery to a join, it can use several variations of the access plan and the join plan:

- First-row scan

  A first-row scan is a variation of a table scan. When the database server finds one match, the table scan halts.

- Skip-duplicate-index scan

  The skip-duplicate-index scan is a variation of an index scan. The database server does not scan duplicates.

- Semi join

  The semi join is a variation of a nested-loop join. The database server halts the inner-table scan when the first match is found. For more information about a semi join, see Nested-loop join on page 292.

Query plans for collection-derived tables

A collection-derived table is a special method that the database server uses to process a query on a collection. To use a collection-derived table, a query must contain the TABLE keyword in the FROM clause of an SQL statement.

For more information about how to use collection-derived tables in an SQL statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

Although the database does not actually create a table for the collection, it processes the data as if it were a table. Collection-derived tables allow developers to use fewer cursors and host variables to access a collection, in some cases.

These SQL statements create a collection column called **children**:

```
CREATE ROW TYPE person(name CHAR(255), id INT);
CREATE TABLE parents(name CHAR(255),
id INT,
children LIST(person NOT NULL));
```

The following query creates a collection-derived table for the **children** column and treats the elements of this collection as rows in a table:

```
SELECT name, id
FROM TABLE(MUTLISET(SELECT children
FROM parents
WHERE parents.id
= 1001)) c_table(name, id);
```

Alternatively, you can specify a collection-derived table in the FROM clause, as shown in this example:

```
SELECT name, id
FROM (SELECT children
FROM parents
WHERE parents.id
= 1001) c_table(name, id);
```

Example showing how the database server completes the query

HCL OneDB™ performs several steps when completing a query for collection-derived tables.

When completing a query, the database server performs the steps shown in this example:

1. Scans the **parent** table to find the row where `parents.id = 1001`

   This operation is listed as a SEQUENTIAL SCAN in the SET EXPLAIN output that Figure 52: Query plan that uses a collection-derived table on page 308 shows.

2. Reads the value of the collection column called **children**.
3. Scans the single collection and returns the value of **name** and **id** to the application.

   This operation is listed as a COLLECTION SCAN in the SET EXPLAIN output that Figure 52: Query plan that uses a collection-derived table on page 308 shows.

Figure 52. Query plan that uses a collection-derived table

```
QUERY:
------
SELECT name, id
FROM (SELECT children
FROM parents
WHERE parents.id
= 1001) c_table(name, id);

Estimated Cost: 2
Estimated # of Rows Returned: 1

  1) lsuto.c_table: COLLECTION SCAN
    Subquery:
    ---------
    Estimated Cost: 1
    Estimated # of Rows Returned: 1

      1) lsuto.parents: SEQUENTIAL SCAN

            Filters: lsuto.parents.id = 1001
```

Derived tables folded into parent queries

You can improve the performance of collection-derived tables by using SQL to fold derived tables in simple queries into a parent query instead of into query results that are put into a temporary table.

Use SQL like that in this example:

```
select * from ((select col1, col2 from tab1)) as vtab(c1,c2)
```

However, if the query is complex because it involves aggregates, ORDER BY operations, or the UNION operation, the server creates a temporary table.

The database server folds derived tables in a manner that is similar to the way the server folds views through the IFX_FOLDVIEW configuration parameter (described in Enable view folding to improve query performance on page 412). When the IFX_FOLDVIEW configuration parameter is enabled, views are folded into a parent query. The views are not folded into query results that are put into a temporary table.

The following examples show derived tables folded into the main query.

Figure 53. Query plan that uses a derived table folded into the parent query

```
select * from ((select vcol0, tab1.col1 from
        table(multiset(select col2 from tab2 where col2 > 50 ))
 vtab2(vcol0),tab1 )) vtab1(vcol1,vcol2)
 where vcol1 = vcol2

Estimated Cost: 2
Estimated # of Rows Returned: 1

  1) informix.tab2: SEQUENTIAL SCAN

        Filters: informix.tab2.col2 > 50

  2) informix.tab1: SEQUENTIAL SCAN

        Filters:
        Table Scan Filters: informix.tab1.col1 > 50

DYNAMIC HASH JOIN
    Dynamic Hash Filters: informix.tab2.col2 = informix.tab1.col1
```

Figure 54. Second query plan that uses a derived table folded into the parent query

```
select * from (select col1 from tab1 where col1 = 100) as vtab1(c1)
left join (select col1 from tab2 where col1 = 10) as vtab2(vc1)
on  vtab1.c1 =  vtab2.vc1

Estimated Cost: 5
Estimated # of Rows Returned: 1

      1) informix.tab1: SEQUENTIAL SCAN

            Filters: informix.tab1.col1 = 100

      2) informix.tab2: AUTOINDEX PATH

        (1) Index Keys: col1   (Key-Only)
            Lower Index Filter: informix.tab1.col1 = informix.tab2.col1
            Index Key Filters:  (informix.tab2.col1 = 10 )

    ON-Filters:(informix.tab1.col1 = informix.tab2.col1
            AND informix.tab2.col1 = 10 )
    NESTED LOOP JOIN(LEFT OUTER JOIN)
```

The following example shows a complex query involving the UNION operation. Here, a temporary table has been created.

Figure 55. Complex derived-table query that creates a temporary table

```
select * from (select col1 from tab1 union select col2 from tab2 )
as vtab(vcol1)  where vcol1 < 50

Estimated Cost: 4
Estimated # of Rows Returned: 1

  1) (Temp Table For Collection Subquery): SEQUENTIAL SCAN
```

XML query plans in HCL® Data Studio

HCL® Data Studio consists of a set of tools to use for administration, data modeling, and building queries from data that comes from data servers. The EXPLAIN_SQL routine prepares a query and returns a query plan in XML. The HCL® Data Studio Administration Edition can use the EXPLAIN_SQL routine to obtain a query plan in XML format, interpret the XML, and render the plan visually.

If you plan to use HCL® Data Studio to obtain Visual Explain output, you must create and specify a default sbspace name for the SBSPACENAME configuration parameter in your `onconfig` file. The EXPLAIN_SQL routine creates BLOBs in this sbspace.

For information about using HCL® Data Studio, see HCL® Data Studio documentation.

Factors that affect the query plan

When the optimizer determines the query plan, it assigns a cost to each possible plan and then chooses the plan with the lowest cost. The optimizer analyzes several factors to determine the cost of each query plan.

Some of the factors that the optimizer uses to determine the cost of each query plan are:

- The number of I/O requests that are associated with each file system access
- The CPU work that is required to determine which rows meet the query predicate
- The resources that are required to sort or group the data
- The amount of memory available for the query (specified by the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters)

To calculate the cost of each possible query plan, the optimizer:

- Uses a set of statistics that describes the nature and physical characteristics of the table data and indexes
- Examines the query filters
- Examines the indexes that can be used in the plan
- Uses the cost of moving data to perform joins locally or remotely for distributed queries

For queries that access remote tables in cross-server operations, certain characteristics can significantly degrade performance relative to the corresponding DML operations on tables and views in the local database. Query specifications that can potentially limit performance with remote tables include the following specifications:

- ANSI LEFT OUTER JOIN syntax
- Derived tables based on remote tables
- TEMP tables as materialized views that reference remote tables.

Limitations on remote views

Reoptimization can occur with multiple executions of queries involving remote views. The optimizer does not pick up the query plans from statement cache even if the statement cache is enabled.

Statistics held for the table and index

The accuracy with which the query optimizer can assess the execution cost of a query plan depends on the information available to the optimizer. Use the UPDATE STATISTICS statement to maintain simple statistics about a table and its associated indexes. Updated statistics provide the query optimizer with information that can minimize the amount of time required to perform queries on that table.

The database server starts a statistical profile of a table when the table is created, and the profile is refreshed when you issue the UPDATE STATISTICS statement. The query optimizer does not recalculate the profile for tables automatically. In some cases, gathering the statistics might take longer than executing the query.

To ensure that the optimizer selects a query plan that best reflects the current state of your tables, run UPDATE STATISTICS at regular intervals. For guidelines, see Update statistics when they are not generated automatically on page 378.

The optimizer uses the following system catalog information as it creates a query plan:

- The number of rows in a table, as of the most recent UPDATE STATISTICS statement
- Whether a column is constrained to be unique
- The distribution of column values, when requested with the MEDIUM or HIGH keyword in the UPDATE STATISTICS statement

  For more information about data distributions, see Creating data distributions on page 380.
- The number of disk pages that contain row data

The optimizer also uses the following system catalog information about indexes:

- The indexes that exist on a table, including the columns that they index, whether they are ascending or descending, and whether they are clustered
- The depth of the index structure (a measure of the amount of work that is needed to perform an index lookup)
- The number of disk pages that index entries occupy
- The number of unique entries in an index, which can be used to estimate the number of rows that an equality filter returns
- Second-largest and second-smallest key values in an indexed column

Only the second-largest and second-smallest key values are noted, because the extreme values might have a special meaning that is not related to the rest of the data in the column. The database server assumes that key values are distributed evenly between the second largest and second smallest. Only the initial 4 bytes of these keys are stored. If you create a distribution for a column associated with an index, the optimizer uses that distribution when it estimates the number of rows that match a query.

For more information about system catalog tables, see the *HCL OneDB™ Guide to SQL: Reference*.

Filters in the query

The query optimizer bases query-cost estimates on the number of rows to be retrieved from each table. In turn, the estimated number of rows is based on the *selectivity* of each conditional expression that is used within the WHERE clause. A conditional expression that is used to select rows is termed a *filter*.

The selectivity is a value between 0 and 1 that indicates the proportion of rows within the table that the filter can pass. A selective filter, one that passes few rows, has a selectivity near 0, and a filter that passes almost all rows has a selectivity near 1. For guidelines on filters, see Improve filter selectivity on page 366.

The optimizer can use data distributions to calculate selectivity for the filters in a query. However, in the absence of data distributions, the database server calculates selectivity for filters of different types based on table indexes. The following table lists some of the selectivity values that the optimizer assigns to filters of different types. Selectivity that is calculated using data distributions is even more accurate than the selectivity that this table shows.

In the table:

- *indexed-col* is the first or only column in an index.
- *2nd-max*, *2nd-min* are the second-largest and second-smallest key values in indexed column.
- The plus sign (+) means logical union (= the Boolean OR operator) and the multiplication symbol (x) means logical intersection (= the Boolean AND operator).

Table 16. Selectivity values that the optimizer assigns to filters of different types

Filter Expression	Selectivity (F)
indexed-col = *literal-value* *indexed-col* = *host-variable* *indexed-col* IS NULL	F = 1/(number of distinct keys in index)
tab1.indexed-col = *tab2.indexed-col*	F = 1/(number of distinct keys in the larger index)
indexed-col > *literal-value*	F = (2nd-max - *literal-value*)/(2nd-max - 2nd-min)
indexed-col < *literal-value*	F = (*literal-value* - 2nd-min)/(2nd-max - 2nd-min)
any-col IS NULL *any-col* = *any-expression*	F = 1/10
any-col > *any-expression* *any-col* < *any-expression*	F = 1/3
any-col MATCHES *any-expression* *any-col* LIKE *any-expression*	F = 1/5
EXISTS *subquery*	F = 1 if *subquery* estimated to return >0 rows, else 0
NOT *expression*	F = 1 - F(*expression*)
expr1 AND *expr2*	F = F(*expr1*) x F(*expr2*)
expr1 OR *expr2*	F = F(*expr1*) + F(*expr2*) - (F(*expr1*) x F(*expr2*))
any-col IN *list*	Treated as *any-col* = $item_1$ OR . . . OR *any-col* = $item_{n.}$

Table 16. Selectivity values that the optimizer assigns to filters of different types (continued)

Filter Expression	Selectivity (F)
any-col relop ANY *subquery*	Treated as any-col relop $value_1$ OR . . . OR *any-col relop* $value_n$ for estimated size of subquery *n*. Here *relop* is any relational operator, such as <, >, >=, <=.

Indexes for evaluating a filter

The query optimizer notes whether an index can be used to evaluate a filter. For this purpose, an indexed column is any single column with an index or the first column named in a composite index.

If the values contained in the index are all that is required, the database server does not read the rows. It is faster to omit the page lookups for data pages whenever the database server can read values directly from the index.

The optimizer can choose an index for any one of the following cases:

- When the column is indexed and a value to be compared is a literal, a host variable, or an uncorrelated subquery

  The database server can locate relevant rows in the table by first finding the row in an appropriate index. If an appropriate index is not available, the database server must scan each table in its entirety.
- When the column is indexed and the value to be compared is a column in another table (a join expression)

  The database server can use the index to find matching values. The following join expression shows such an example:

  ```
  WHERE customer.customer_num = orders.customer_num
  ```

  If rows of **customer** are read first, values of **customer_num** can be applied to an index on **orders.customer_num**.
- When processing an ORDER BY clause

  If all the columns in the clause appear in the required sequence within a single index, the database server can use the index to read the rows in their ordered sequence, thus avoiding a sort.
- When processing a GROUP BY clause

  If all the columns in the clause appear in one index, the database server can read groups with equal keys from the index without requiring additional processing after the rows are retrieved from their tables.

Effect of PDQ on the query plan

When the parallel database query (PDQ) feature is turned on, the optimizer can choose to execute a query in parallel. This can improve performance dramatically when the database server processes queries that decision-support applications initiate.

For more information, see Parallel database query (PDQ) on page 345.

Effect of OPTCOMPIND on the query plan

The OPTCOMPIND setting influences the access plan that the optimizer chooses for single and multiple-table queries. You can change the value of OPTCOMPIND within a session for different kinds of queries.

To change the value of OPTCOMPIND within a session, use the SET ENVIRONMENT OPTCOMPIND command, not the OPTCOMPIND configuration parameter. For more information about using this command, see Setting the value of OPTCOMPIND within a session on page 44.

Single-table query

For single-table scans, when OPTCOMPIND is set to `0` or `1` and the current transaction isolation level is Repeatable Read, the optimizer considers two types of access plans.

If:

- An index is available, the optimizer uses it to access the table.
- No index is available, the optimizer considers scanning the table in physical order.

When OPTCOMPIND is not set in the database server configuration, its value defaults to `2`. When OPTCOMPIND is set to `2` or `1` and the current isolation level is not Repeatable Read, the optimizer chooses the least expensive plan to access the table.

Multitable query

For join plans, the OPTCOMPIND setting influences the access plan for a specific ordered pair of tables.

Set OPTCOMPIND to `0` if you want the database server to select a join method exactly as it did in previous versions of the database server. This option ensures compatibility with previous versions.

If OPTCOMPIND is set to `0` or set to `1` and the current transaction isolation level is Repeatable Read, the optimizer gives preference to the nested-loop join.

Important: When OPTCOMPIND is set to `0`, the optimizer does not choose a hash join.

If OPTCOMPIND is set to `2` or set to `1` and the transaction isolation level is not Repeatable Read, the optimizer chooses the least expensive query plan from among those previously listed and gives no preference to the nested-loop join.

Effect of available memory on the query plan

HCL OneDB™ constrains the amount of memory that a parallel query can use based on the values of the DS_TOTAL_MEMORY and DS_MAX_QUERIES configuration parameters. If the amount of memory available for the query is too low to execute a hash join, the database server uses a nested-loop join instead.

For more information about parallel queries and the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters, see Parallel database query (PDQ) on page 345.

Time costs of a query

You can adjust a few, but not all, of the response-time effects of actions that the database server performs when processing a query.

The following costs can be reduced by optimal query construction and appropriate indexes:

- Sort time
- Data mismatches
- In-place ALTER TABLE
- Index lookups

For information about how to optimize specific queries, see Improving individual query performance on page 365.

Memory-activity costs

The database server can process only data in memory. It must read rows into memory to evaluate those rows against the filters of a query. After the server finds rows that satisfy those filters, it prepares an output row in memory by assembling the selected columns.

Most of these activities are performed quickly. Depending on the computer and its workload, the database server can perform hundreds or even thousands of comparisons each second. As a result, the time spent on in-memory work is usually a small part of the execution time.

Although some in-memory activities, such as sorting, take a significant amount of time, it takes much longer to read a row from disk than to examine a row that is already in memory.

Sort-time costs

A sort requires in-memory work as well as disk work. The in-memory work depends on the number of columns that are sorted, the width of the combined sort key, and the number of row combinations that pass the query filter. You can reduce the cost of sorting.

You can use the following formula to calculate the in-memory work that a sort operation requires:

$$W_m = (c * N_{fr}) + (w * N_{fr}\log_2(N_{fr}))$$

W_m

is the in-memory work.

c

is the number of columns to order and represents the costs to extract column values from the row and concatenate them into a sort key.

w

is proportional to the width of the combined sort key in bytes and stands for the work to copy or compare one sort key. A numeric value for *w* depends strongly on the computer hardware in use.

N_{fr}

is the number of rows that pass the query filter.

Sorting can involve writing information temporarily to disk if the amount of data to sort is large. You can direct the disk writes to occur in the operating-system file space or in a dbspace that the database server manages. For details, see Configure dbspaces for temporary tables and sort files on page 114.

The disk work depends on the number of disk pages where rows appear, the number of rows that meet the conditions of the query predicate, the number of rows that can be placed on a sorted page, and the number of merge operations that must be performed. Use the following formula to calculate the disk work that a sort operation requires:

$$W_d = p + (N_{fr}/N_{rp}) * 2 * (m - 1))$$

W_d

is the disk work.

p

is the number of disk pages.

N_{fr}

is the number of rows that pass the filters.

N_{rp}

is the number of rows that can be placed on a page.

m

represents the number of *levels of merge* that the sort must use.

The factor *m* depends on the number of sort keys that can be held in memory. If there are no filters, N_{fr}/N_{rp} is equivalent to *p*.

When all the keys can be held in memory, *m*=1 and the disk work is equivalent to *p*. In other words, the rows are read and sorted in memory.

For moderate to large tables, rows are sorted in batches that fit in memory, and then the batches are merged. When *m*=2, the rows are read, sorted, and written in batches. Then the batches are read again and merged, resulting in disk work proportional to the following value:

$$W_d = p + (2 * (N_{fr}/N_{rp}))$$

The more specific the filters, the fewer the rows that are sorted. As the number of rows increases, and the amount of memory decreases, the amount of disk work increases.

To reduce the cost of sorting, use the following methods:

- Make your filters as specific (selective) as possible.
- Limit the projection list to the columns that are relevant to your problem.

Row-reading costs

When the database server needs to examine a row that is not already in memory, it must read that row from disk. The database server does not read only one row; it reads the entire page that contains the row. If the row spans more than one page, it reads all of the pages.

The actual cost of reading a page is variable and hard to predict. The actual cost is a combination of the factors shown in the following table.

Factor	Effect of Factor
Buffering	If the needed page is in a page buffer already, the cost to read is nearly zero.
Contention	If two or more applications require access to the disk hardware, I/O requests can be delayed.
Seek time	The slowest thing that a disk does is to *seek*; that is, to move the access arm to the track that holds the data. Seek time depends on the speed of the disk and the location of the disk arm when the operation starts. Seek time varies from zero to nearly a second.
Latency	The transfer cannot start until the beginning of the page rotates under the access arm. This *latency*, or rotational delay, depends on the speed of the disk and on the position of the disk when the operation starts. Latency can vary from zero to a few milliseconds.

The time cost of reading a page can vary from microseconds for a page that is already in a buffer, to a few milliseconds when contention is zero and the disk arm is already in position, to hundreds of milliseconds when the page is in contention and the disk arm is over a distant cylinder of the disk.

Sequential access costs

Disk costs are lowest when the database server reads the rows of a table in physical order.

When the first row on a page is requested, the disk page is read into a buffer page. After the page is read in, it does not need not to be read again; requests for subsequent rows on that page are filled from the buffer until all the rows on that page are processed. When one page is exhausted, the page for the next set of rows must be read in.

When you use unbuffered devices for dbspaces, and the table is organized properly, the disk pages of consecutive rows are placed in consecutive locations on the disk. This arrangement allows the access arm to move very little when it reads sequentially. In addition, latency costs are usually lower when pages are read sequentially.

Related information

Read-ahead operations on page

Nonsequential access costs

The disk-access time is much higher when a disk device reads table pages nonsequentially than when it reads that same table sequentially.

Whenever a table is read in random order, additional disk accesses are required to read the rows in the required order. Disk costs are higher when the rows of a table are read in a sequence unrelated to physical order on disk. Because the pages are not read sequentially from the disk, both seek and rotational delays occur before each page can be read.

Nonsequential access often occurs when you use an index to locate rows. Although index entries are sequential, there is no guarantee that rows with adjacent index entries must reside on the same (or adjacent) data pages. In many cases, a separate disk access must be made to fetch the page for each row located through an index. If a table is larger than the page buffers, a page that contained a row previously read might be cleaned (removed from the buffer and written back to the disk) before a subsequent request for another row on that page can be processed. That page might have to be read in again.

Depending on the relative ordering of the table with respect to the index, you can sometimes retrieve pages that contain several needed rows. The degree to which the physical ordering of rows on disk corresponds to the order of entries in the index is called *clustering*. A highly clustered table is one in which the physical ordering on disk corresponds closely to the index.

Index lookup costs

The database server incurs additional costs when it finds a row through an index. The index is stored on disk, and its pages must be read into memory with the data pages that contain the desired rows.

An index lookup works down from the root page to a leaf page. The root page, because it is used so often, is almost always found in a page buffer. The odds of finding a leaf page in a buffer depend on the size of the index, the form of the query, and the frequency of column-value duplication. If each value occurs only once in the index and the query is a join, each row to be joined requires a nonsequential lookup into the index, followed by a nonsequential access to the associated row in the table.

Reading duplicate values from an index

Reading an index with duplicate entries incurs an additional cost over reading the table sequentially.

Each entry or set of entries with the same value must be located in the index. Then, for each entry in the index, a random access must be made to the table to read the associated row. However, if there are many duplicate rows per distinct index value, and the associated table is highly clustered, the added cost of joining through the index can be slight.

Searching for NCHAR or NVARCHAR columns in an index

A query using an index on an NCHAR or NVARCHAR scans the entire index, resulting in additional time costs.

Global Language Support (GLS) Only

Indexes that are built on NCHAR or NVARCHAR columns are sorted using a locale-specific comparison value. For example, the Spanish double-l character (ll) might be treated as a single unique character instead of a pair of ls.

In some locales, the comparison value is not based on the code-set order. The index build uses the locale-specific comparison value to store the key values in the index. As a result, a query using an index on an NCHAR or NVARCHAR scans the entire index because the database server searches the index in code-set order.

In-place ALTER TABLE costs

For certain conditions, the database server uses an in-place alter algorithm to modify each row when you execute an ALTER TABLE statement. After the alter table operation, the database server inserts rows using the latest definition. If your query accesses rows that are not yet converted to the new table definition, you might notice a slight degradation in the performance of your individual query, because the database server reformats each row in memory before it is returned.

For more information about the conditions and performance advantages when an in-place alter occurs, see Altering a table definition on page 192.

View costs

A complex view could run more slowly than expected.

You can create views of tables for a number of reasons:

- To limit the data that a user can access
- To reduce the time that it takes to write a complex query
- To hide the complexity of the query that a user needs to write

However, a query against a view might execute more slowly than expected when the complexity of the view definition causes a temporary table to be created to process the query. This temporary table is referred to as a *materialized view*. For example, you can create a view with a union to combine results from several SELECT statements.

The following sample SQL statement creates a view that includes unions:

```
CREATE VIEW view1 (col1, col2, col3, col4)
   AS
      SELECT a, b, c, d
         FROM tab1 WHERE
      UNION
      SELECT a2, b2, c2, d2
         FROM tab2 WHERE
...
      UNION
      SELECT an, bn, cn, dn
         FROM tabn WHERE
;
```

When you create a view that contains complex SELECT statements, the end user does not need to handle the complexity. The end user can just write a simple query, as the following example shows:

```
SELECT a, b, c, d
   FROM view1
      WHERE a < 10;
```

However, this query against **view1** might execute more slowly than expected because the database server creates a fragmented temporary table for the view before it executes the query.

Another situation when the query might execute more slowly than expected is if you use a view in an ANSI join. The complexity of the view definition might cause a temporary table to be created.

To determine if you have a query that must build a temporary table to process the view, execute the SET EXPLAIN statement. If you see `Temp Table For View` in the SET EXPLAIN output file, your query requires a temporary table to process the view.

Small-table costs

A table is small if it occupies so few pages that it can be retained entirely in the page buffers. Operations on small tables are generally faster than operations on large tables.

As an example, in the **stores_demo** database, the **state** table that relates abbreviations to names of states has a total size of fewer than 1000 bytes; it fits in no more than two pages. This table can be included in any query at little cost. No matter how this table is used, it costs no more than two disk accesses to retrieve this table from disk the first time that it is required.

Data-mismatch costs

An SQL statement can encounter additional costs when the data type of a column that is used in a condition differs from the definition of the column in the CREATE TABLE statement.

For example, the following query contains a condition that compares a column to a data type value that differs from the table definition:

```
CREATE TABLE table1 (a integer, );
SELECT * FROM table1
   WHERE a = '123';
```

The database server rewrites this query before execution to convert `123` to an integer. The SET EXPLAIN output shows the query in its adjusted format. This data conversion has no noticeable overhead.

The additional costs of a data mismatch are most severe when the query compares a character column with a noncharacter value and the length of the number is not equal to the length of the character column. For example, the following query contains a condition in the WHERE clause that equates a character column to an integer value because of missing quotation marks:

```
CREATE TABLE table2 (char_col char(3), );
SELECT * FROM table2
   WHERE char_col = 1;
```

This query finds all of the following values that are valid for **char_col**:

```
' 1'
'001'
'1'
```

These values are not necessarily clustered together in the index keys. Therefore, the index does not provide a fast and correct way to obtain the data. The SET EXPLAIN output shows a sequential scan for this situation.

Warning: The database server does not use an index when the SQL statement compares a character column with a noncharacter value that is not equal in length to the character column.

Encrypted-value costs

An encrypted value uses more storage space than the corresponding plain-text value because all of the information needed to decrypt the value except the encryption key is stored with the value.

Most encrypted data requires approximately 33 percent more storage space than unencrypted data. Omitting the hint used with the password can reduce encryption overhead by up to 50 bytes. If you are using encrypted values, you must make sure that you have sufficient space available for the values.

GLS functionality costs

Sorting or indexing certain data sets can degrade performance.

For information about the performance degradation that occurs from indexing some data sets, see Searching for NCHAR or NVARCHAR columns in an index on page 318.

If you do not need a non-ASCII collation sequence, use the CHAR and VARCHAR data types for character columns whenever possible. Because CHAR and VARCHAR data require simple value-based comparison, sorting and indexing these columns is less expensive than for non-ASCII data types (NCHAR or NVARCHAR, for example).

For more information about other character data types, see the *HCL OneDB™ GLS User's Guide*.

Network-access costs

Moving data over a network imposes delays in addition to those you encounter with direct disk access.

Network delays can occur when the application sends a query or update request across the network to a database server on another computer. Although the database server performs the query on the remote host computer, that database server returns the output to the application over the network.

Data sent over a network consists of command messages and buffer-sized blocks of row data. Although the details can differ depending on the network and the computers, the database server network activity follows a simple model in which one computer, the *client*, sends a request to another computer, the *server*. The server replies with a block of data from a table.

Whenever data is exchanged over a network, delays are inevitable in the following situations:

- When the network is busy, the client must wait its turn to transmit. Such delays are usually less than a millisecond. However, on a heavily loaded network, these delays can increase exponentially to tenths of seconds and more.
- When the server is handling requests from more than one client, requests might be queued for a time that can range from milliseconds to seconds.
- When the server acts on the request, it incurs the time costs of disk access and in-memory operations that the preceding sections describe.

Transmission of the response is also subject to network delays.

Network access time is extremely variable. In the best case, when neither the network nor the server is busy, transmission and queuing delays are insignificant, and the server sends a row almost as quickly as a local database server might. Furthermore, when the client asks for a second row, the page is likely to be in the page buffers of the server.

Unfortunately, as network load increases, all these factors tend to worsen at the same time. Transmission delays rise in both directions, which increases the queue at the server. The delay between requests decreases the likelihood of a page remaining in the page buffer of the responder. Thus, network-access costs can change suddenly and quite dramatically.

If you use the SELECT FIRST *n* clause in a distributed query, you will still see only the requested amount of data. However, the local database server does not send the SELECT FIRST *n* clause to the remote site. Therefore, the remote site might return more data.

The optimizer that the database server uses assumes that access to a row over the network takes longer than access to a row in a local database. This estimate includes the cost of retrieving the row from disk and transmitting it across the network.

For information about actions that might improve performance across the network, see the following sections:

- Optimizer estimates of distributed queries on page 410
- #unique_533
- Multiplexed connections and CPU utilization on page 59
- Network buffer pools on page 50

Optimization when SQL is within an SPL routine

If an SPL routine contains SQL statements, the database server optimizes and executes the SQL within the SPL routine.

The topics in this section contain information about how and when the database server optimizes and executes these routines.

SQL optimization

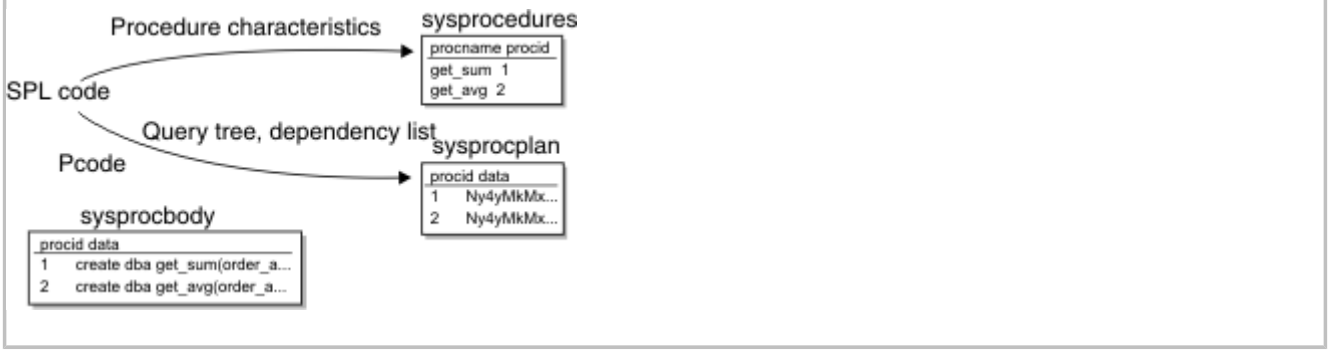
If an SPL routine contains SQL statements, at some point the query optimizer evaluates the possible query plans for SQL in the SPL routine and selects the query plan with the lowest cost. The database server puts the selected query plan for each SQL statement in an execution plan for the SPL routine.

When you create an SPL routine with the CREATE PROCEDURE statement, the database server attempts to optimize the SQL statements within the SPL routine at that time. If the tables cannot be examined at compile time (because they do not exist or are not available), the creation does not fail. In this case, the database server optimizes the SQL statements the first time that the SPL routine executes.

The database server stores the optimized execution plan in the **sysprocplan** system catalog table for use by other processes. In addition, the database server stores information about the SPL routine (such as procedure name and owner) in the **sysprocedures** system catalog table and an ASCII version of the SPL routine in the **sysprocbody** system catalog table.

Figure 56: SPL information stored in system catalog tables on page 323 summarizes the information that the database server stores in system catalog tables during the compilation process.

Figure 56. SPL information stored in system catalog tables



Displaying the execution plan

When you execute an SPL routine, it is already optimized. You can display the query plan for each SQL statement contained in the SPL routine

To display the query plan, execute the SET EXPLAIN ON statement prior to one of the following SQL statements that always tries to optimize the SPL routine:

- CREATE PROCEDURE
- UPDATE STATISTICS FOR PROCEDURE

For example, use the following statements to display the query plan for an SPL routine:

```
SET EXPLAIN ON;
UPDATE STATISTICS FOR PROCEDURE procname;
```

Automatic reoptimization

In some situations, the database server reoptimizes an SQL statement the next time an SPL routine.

If the AUTO_REPREPARE configuration parameter or the IFX_AUTO_REPREPARE session environment variable is disabled, the following error can result when prepared objects or SPL routines are executed after the schema of a table referenced by the prepared object or indirectly referenced by the SPL routine has been modified:

```
-710  Table <table-name> has been dropped, altered, or renamed.
```

The database server uses a dependency list to keep track of changes that would cause reoptimization the next time that an SPL routine executes.

The database server reoptimizes an SQL statement the next time an SPL routine executes after one of the following situations:

- Execution of any data definition language (DDL) statement (such as ALTER TABLE, DROP INDEX, and CREATE INDEX) that might alter the query plan
- Alteration of a table that is linked to another table with a referential constraint (in either direction)
- Execution of UPDATE STATISTICS FOR TABLE for any table involved in the query

  The UPDATE STATISTICS FOR TABLE statement changes the version number of the specified table in **systables.**
- Renaming a column, database, or index with the RENAME statement

Whenever the SPL routine is reoptimized, the database server updates the **sysprocplan** system catalog table with the reoptimized execution plan.

Reoptimizing SPL routines

You can run an SQL statement that reoptimizes an SPL routine to prevent automatic reoptimization.

If you do not want to incur the cost of automatic reoptimization when you first execute an SPL routine after one of the situations that Automatic reoptimization on page 323 lists, execute the UPDATE STATISTICS statement with the FOR PROCEDURE clause immediately after the situation occurs. In this way, the SPL routine is reoptimized before any users execute it.

To prevent unnecessary reoptimization of all SPL routines, ensure that you specify a specific procedure name in the FOR PROCEDURE clause.

```
UPDATE STATISTICS FOR PROCEDURE myroutine;
```

For guidelines to run UPDATE STATISTICS, see Update statistics when they are not generated automatically on page 378.

Optimization levels for SQL in SPL routines

The current optimization level set in an SPL routine affects how the SPL routine is optimized.

The algorithm that a SET OPTIMIZATION HIGH statement invokes is a sophisticated, cost-based strategy that examines all reasonable query plans and selects the best overall alternative. For large joins, this algorithm can incur more overhead than desired. In extreme cases, you can run out of memory.

The alternative algorithm that a SET OPTIMIZATION LOW statement invokes eliminates unlikely join strategies during the early stages, which reduces the time and resources spent during optimization. However, when you specify a low level of optimization, the optimal strategy might not be selected because it was eliminated from consideration during early stages of the algorithm.

For SPL routines that remain unchanged or change only slightly and that contain complex SELECT statements, you might want to set the SET OPTIMIZATION statement to `HIGH` when you create the SPL routine. This optimization level stores the best query plans for the SPL routine. Then set optimization to `LOW` before you execute the SPL routine. The SPL routine then uses the optimal query plans and runs at the more cost-effective rate if reoptimization occurs.

Execution of an SPL routine

When the database server executes an SPL routine with the EXECUTE PROCEDURE statement, with the CALL statement, or within an SQL statement, the server performs several activities.

The database server performs these activities:

- It reads the interpreter code from the system catalog tables and converts it from a compressed format to an executable format. If the SPL routine is in the UDR cache, the database server retrieves it from the cache and bypasses the conversion step.
- It executes any SPL statements that it encounters.
- When the database server encounters an SQL statement, it retrieves the query plan from the database and executes the statement. If the query plan has not been created, the database server optimizes the SQL statement before it executes.
- When the database server reaches the end of the SPL routine or when it encounters a RETURN statement, it returns any results to the client application. Unless the RETURN statement has a WITH RESUME clause, the SPL routine execution is complete.

SPL routine executable format stored in UDR cache

The first time that a user executes an SPL routine, the database server stores the executable format and any query plans in the UDR cache in the virtual portion of shared memory.

When another user executes an SPL routine, the database server first checks the UDR cache. SPL execution performance improves when the database server can execute the SPL routine from the UDR cache. The UDR cache also stores UDRs, user-defined aggregates, and extended data types definitions.

Related reference

Configure and monitor memory caches on page 83

Adjust the UDR cache

The default number of SPL routines, UDRs, and other user-defined definitions in the UDR cache is 127. You can change the number of entries with the PC_POOLSIZE configuration parameter.

The database server uses a hashing algorithm to store and locate SPL routines in the UDR cache. You can modify the number of *buckets* in the UDR cache with the PC_HASHSIZE configuration parameter. For example, if the value of the PC_POOLSIZE configuration parameter is `100` and the value of the PC_HASHSIZE configuration parameter is `10`, each bucket can have up to 10 SPL routines and UDRs.

Too many buckets cause the database server to move out cached SPL routines when the bucket fills. Too few buckets increase the number of SPL routines in a bucket, and the database server must search though the SPL routines in a bucket to determine if the SPL routine that it needs is there.

When the number of entries in a bucket reaches 75 percent, the database server removes the least recently used SPL routines from the bucket (and hence from the UDR cache) until the number of SPL routines in the bucket is 50 percent of the maximum SPL routines in the bucket.

Monitor the UDR cache by running the onstat -g prc command. If the numbers in the **hits** fields are not evenly distributed among buckets, increase the value of the PC_HASHSIZE configuration parameter. Adjust the number of buckets to have the least number of high hit entries per bucket.

Important: PC_POOLSIZE and PC_HASHSIZE also control other memory caches for the database server (excluding the buffer pool, the SQL statement cache, the data distribution cache, and the data-dictionary cache). When you modify the size and number of hash buckets for SQL routines, you also modify the size and number of hash buckets for the other caches (such as the aggregate cache, oplcass, and typename cache).

Related information

onstat -g prc command: Print sessions using UDR or SPL routines on page

PC_POOLSIZE configuration parameter on page

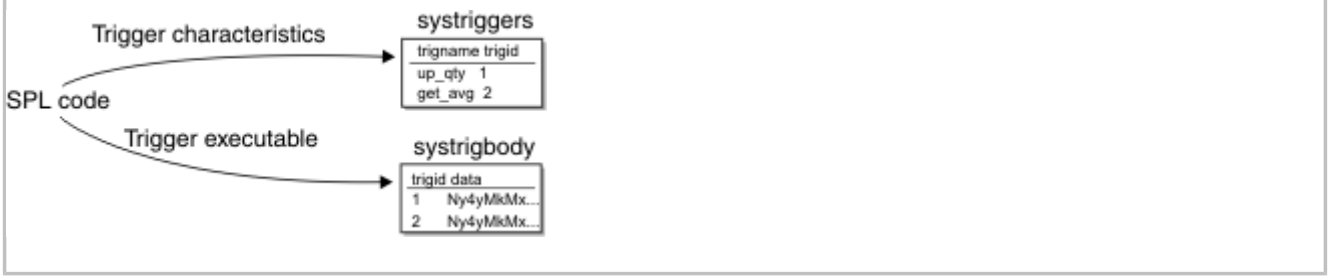
PC_HASHSIZE configuration parameter on page

Trigger execution

A *trigger* is a database object that automatically executes one or more SQL statements (the *triggered action*) when a specified data manipulation language operation (the *triggering event*) occurs. You can define one or more triggers on a table to execute after a SELECT, INSERT, UPDATE or DELETE triggering event.

You can also define INSTEAD OF triggers on a view. These triggers specify the SQL statements to be executed as triggered actions on the underlying table when a triggering INSERT, UPDATE or DELETE statement attempts to modify the view. These triggers are called INSTEAD OF triggers because only the triggered SQL action is executed; the triggering event is not executed. For more information about using triggers, see the *HCL OneDB™ Guide to SQL: Tutorial* and information about the CREATE TRIGGER statement in the *HCL OneDB™ Guide to SQL: Syntax*.

Figure 57. Trigger information stored in system catalog tables



When you use the CREATE TRIGGER statement to register a new trigger, the database server:

- Stores information about the trigger in the **systriggers** system catalog table.
- Stores the text of the statements that the trigger executes in the **systrigbody** system catalog table.

The **sysprocedures** system catalog table identifies trigger routines that can be invoked only as triggered actions.

Memory-resident tables of the **sysmaster** database indicate whether the table or view has triggers on it.

Whenever a SELECT, INSERT, UPDATE, or DELETE statement is issued, the database server checks to see if the statement is a *triggering event* that activates a trigger for the table and columns (or for the view) on which the DML statement operates. If the statement requires activating triggers, the database server retrieves the statement text of the triggered actions from the **systrigbody** table and runs the triggered DML statements or SPL routine before, during, or after the triggering events. For INSTEAD OF triggers on a view, the database server performs the triggered actions instead of the triggering events.

Performance implications for triggers

In many situations, triggers can improve performance slightly because of the reduction in the number of messages passed from the client to the database server.

For example, if the trigger fires five SQL statements, the client saves at least 10 messages passed between the client and database server (one to send the SQL statement and one for the reply after the database server executes the SQL statement). Triggers improve performance the most when they execute more SQL statements and the network speed is comparatively slow.

When the database server executes an SQL statement, it must perform the following actions:

- Determine if triggers must be fired
- Retrieve the triggers from **systriggers** and **systrigbody**

These operations cause only a slight performance impact that can be offset by the decreased number of messages passed between the client and the server.

However, triggers executed on SELECT statements have additional performance implications. The following sections explain these implications.

SELECT triggers on tables in a table hierarchy

When the database server executes a SELECT statement that includes a table that is involved in a table hierarchy, and the SELECT statement fires a SELECT trigger, performance might be slower if the SELECT statement that invokes the trigger involves a join, sort, or materialized view.

In this case, the database server does not know which columns are affected in the table hierarchy, so it can execute the query differently. The following behaviors might occur:

- Key-only index scans are disabled on the table that is involved in a table hierarchy.
- If the database server needs to sort data selected from a table involved in a table hierarchy, it copies all of the columns in the SELECT list to the temporary table, not just the sort columns.
- If the database server uses the table included in the table hierarchy to build a hash table for a hash join with another table, it bypasses the early projection, meaning it uses all of the columns from the table to build the hash table, not just the columns in the join.
- If the SELECT statement contains a materialized view (meaning a temporary table must be built for the columns in a view) that contains columns from a table involved in a table hierarchy, all columns from the table are included in the temporary table, not just the columns actually contained in the view.

SELECT triggers and row buffering

The lack of buffering for SELECT statements that fire SELECT triggers might reduce performance slightly compared to an identical SELECT statement that does not fire a SELECT trigger.

In SELECT statements whose tables do not fire SELECT triggers, the database server sends more than one row back to the client and stores the rows in a buffer even though the client application requested only one row with a FETCH statement. However, for SELECT statements that contain one or more tables that fire a SELECT trigger, the database server sends only the requested row back to the client instead of a buffer full. The database server cannot return other rows to the client until the trigger action occurs.

Optimizer directives

Optimizer directives are comments that tell the query optimizer how to execute a query. You can use optimizer directives to improve query performance.

What optimizer directives are

Optimizer directives are specifications formatted as comments that provide information to the query optimizer about how to execute a query.

You can use two kinds of optimizer directives:

- Optimizer directives in the form of instructions that are embedded in queries (For more information, see Optimizer directives that are embedded in queries on page 328.
- External optimizer directives that you create and save for use as temporary workaround solutions to problems when you do not want to change SQL statements in queries. (For more information, see External optimizer directives on page 329.)

Optimizer directives that are embedded in queries

Optimizer directives embedded in queries are comments in a SELECT statement that provide information to the query optimizer on how to execute a query. You can also place directives in UPDATE and DELETE statements, instructing the optimizer how to access the data.

Optimizer directives can either be explicit directions (for example, "use this index" or "access this table first"), or they can eliminate possible query plans (for example, "do not read this table sequentially" or "do not perform a nested-loop join").

External optimizer directives

External optimizer directives are optimizer directives that an administrator can create and store in the **sysdirectives** catalog table. The administrator can then use an ONCONFIG variable to make the directives available.

Client users also specify an environment variable and can choose to use these optimizer directives in queries in situations when they do not want to insert comments in SQL statements.

External directives are useful when it is not feasible to rewrite a query for a short-term solution to a problem, for example, when a query starts to perform poorly. Rewriting the query by changing the SQL statement is preferable for long-term solutions to problems.

External directives are for occasional use only. The number of directives stored in the **sysdirectives** catalog should not exceed 50. A typical enterprise only needs 0 to 9 directives.

Reasons to use optimizer directives

In most cases, the optimizer chooses the fastest query plan. You can use optimizer directives when the optimizer does not choose the best query plan to perform a query, because of the complexity of the query, or because the query does not have enough information about the nature of the data. A poor query plan produces poor performance.

Before you decide when to use optimizer directives, you should understand what makes a good query plan.

The optimizer creates a query plan based on costs of using different table-access paths, join orders, and join plans.

Some query plan guidelines are:

- Do not use an index when the database server must read a large portion of the table. For example, the following query might read most of the **customer** table:

  ```
  SELECT * FROM customer WHERE STATE <> "ALASKA";
  ```

  Assuming the customers are evenly spread among all 50 states, you might estimate that the database server must read 98 percent of the table. It is more efficient to read the table sequentially than to traverse an index (and subsequently the data pages) when the database server must read most of the rows.
- When you choose between indexes to access a table, use an index that can rule out the most rows. For example, consider the following query:

  ```
  SELECT * FROM customer
  WHERE state = "NEW YORK" AND order_date = "01/20/97"
  ```

  Assuming that 200,000 customers live in New York and only 1000 customers ordered on any one day, the optimizer most likely chooses an index on **order_date** rather than an index on **state** to perform the query.
- Place small tables or tables with restrictive filters early in the query plan. For example, consider the following query:

```
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
      AND
   customer.state = "NEVADA";
```

In this example, if you read the **customer** table first, you can rule out most of the rows by applying the filter that chooses all rows in which `state = "NEVADA"`.

By ruling out rows in the **customer** table, the database server does not read as many rows in the **orders** table (which might be significantly larger than the **customer** table).

- Choose a hash join when neither column in the join filter has an index.

  In the previous example, if **customer.customer_num** and **orders.customer_num** are not indexed, a hash join is probably the best join plan.

- Choose nested-loop joins if:
  - The number of rows retrieved from the outer table after the database server applies any table filters is small, and the inner table has an index that can be used to perform the join.
  - The index on the outermost table can be used to return rows in the order of the ORDER BY clause, eliminating the need for a sort.

For information about query plans, see The query plan on page 291. For more information about directives, see

- Preparation for using directives on page 330
- Guidelines for using directives on page 331
- Types of optimizer directives that are supported in SQL statements on page 331

Preparation for using directives

In most cases, the optimizer chooses the fastest query plan. However, you can take steps to assist the optimizer and to prepare for using directives.

To prepare for using directives, make sure that you perform the following tasks:

- Run UPDATE STATISTICS.

  Without accurate statistics, the optimizer cannot choose the appropriate query plan. Run UPDATE STATISTICS any time that the data in the tables changes significantly (many new rows are added, updated, or deleted). For more information, see Update the statistics for the number of rows on page 379.

- Create distributions.

  One of the first things that you should try when you attempt to improve a slow query is to create distributions on columns involved in a query. Distributions provide the most accurate information to the optimizer about the nature of the data in the table. Run UPDATE STATISTICS HIGH on columns involved in the query filters to see if performance improves. For more information, see Creating data distributions on page 380.

In some cases, the query optimizer does not choose the best query plan because of the complexity of the query or because (even with distributions) it does not have enough information about the nature of the data. In these cases, you can attempt to improve performance for a particular query by using directives.

Guidelines for using directives

Guidelines for directives include frequently analyzing the effectiveness of the query and using negative directives.

Consider the following guidelines:

- Examine the effectiveness of a particular directive frequently to make sure it continues to operate effectively. Imagine a query in a production program with several directives that force an optimal query plan. Some days later, users add, update, or delete a large number of rows, which changes the nature of the data so much that the once optimal query plan is no longer effective. This example illustrates how you must use directives with care.
- Use negative directives (such as AVOID_NL, AVOID_FULL, and so on) whenever possible. When you exclude a behavior that degrades performance, you rely on the optimizer to use the next-best choice rather than attempt to force a path that might not be optimal.

Types of optimizer directives that are supported in SQL statements

Directives that are in SQL statements are embedded in queries. These directives include access-method directives, join-order directives, join-plan directives, and optimization-goal directives.

Include the directives in the SQL statement as a comment that occurs immediately after the SELECT, UPDATE, or DELETE keyword. The first character in a directive is always a plus (+) sign. In the following query, the ORDERED directive specifies that the tables should be joined in the same order as they are listed in the FROM clause. The AVOID_FULL directive specifies that the optimizer should discard any plans that include a full table scan on the listed table (**employee**).

```
SELECT --+ORDERED, AVOID_FULL(e) * FROM employee e, department d
> 50000;
```

For a complete syntax description for directives, see the *HCL OneDB™ Guide to SQL: Syntax*.

To influence the choice of a query plan that the optimizer makes, you can alter the following aspects of a query:

- Access method
- Join order
- Join method
- Optimization goal
- Star-join directives

You can also use EXPLAIN directives instead of the SET EXPLAIN statement to show the query plan. The following sections describe these aspects in detail.

Access-method directives

The database server uses an access method to access a table. The server can either read the table sequentially via a full table scan or use any one of the indexes on the table. Access-method directives influence the access method.

The following table lists the directives that influence the access method:

Access-Method Directive	Description
INDEX	Tells the optimizer to use the index specified to access the table. If the directive lists more than one index, the optimizer chooses the index that yields the least cost.
AVOID_INDEX	Tells the optimizer not use any of the indexes listed. You can use this directive with the AVOID_FULL directive.
INDEX_SJ	Forces an index self-join path using the specified index, or choosing the least costly index in a list of indexes, even if data distribution statistics are not available for the leading index key columns of the index. For information about index self-join paths, see Query plans that include an index self-join path on page 297.
AVOID_INDEX_SJ	Tells the optimizer not to use an index self-join path for the specified index or indexes.
FULL	Tells the optimizer to perform a full table scan.
AVOID_FULL	Tells the optimizer not to perform a full table scan on the listed table. You can use this directive with the AVOID_INDEX directive.
INDEX_ALL or MULTI_INDEX	Access the table by using the specified indexes for a multi-index scan. The INDEX_ALL and MULTI_INDEX keywords are synonyms.
AVOID_MULTI_INDEX	Tells the optimizer not to consider a multi-index scan path for the specified table.

In some cases, forcing an access method can change the join method that the optimizer chooses. For example, if you exclude the use of an index with the AVOID_INDEX directive, the optimizer might choose a hash join instead of a nested-loop join.

The optimizer considers an index self-join path only if all of the following conditions are met:

- The index does not have functional keys, user-defined types, built-in opaque types, or non-B-tree indexes
- Data distribution statistics are available for the index key column under consideration
- The number of rows in the table is at least 10 times the number of unique combinations of all possible lead-key column values.

If all of these conditions are met, the optimizer estimates the cost of an index self-join path and compares it with the costs of alternative access methods. The optimizer then picks the best access method for the table. For more information about the access-method directives and some examples of their usage, see the *HCL OneDB™ Guide to SQL: Syntax*.

Join-order directives

The join-order directive ORDERED tells the optimizer to join tables in the order that the SELECT statement lists them.

Effect of join order on join plan

By specifying the join order, you might affect more than just how tables are joined.

For example, consider the following query:

```
SELECT --+ORDERED, AVOID_FULL(e)
* FROM employee e, department d
WHERE e.dept_no = d.dept_no AND e.salary > 5000
```

In this example, the optimizer chooses to join the tables with a hash join. However, if you arrange the order so that the second table is **employee** (and must be accessed by an index), the hash join is not feasible.

```
SELECT --+ORDERED, AVOID_FULL(e)
* FROM department d, employee e
WHERE e.dept_no = d.dept_no AND e.salary > 5000;
```

The optimizer chooses a nested-loop join in this case.

Join order when you use views

The ORDERED directive that is inside a view or is in a query that contains a view affect the join order.

Two cases can affect join order when you use views:

- The ORDERED directive is inside the view.

  The ORDERED directive inside a view affects the join order of only the tables inside the view. The tables in the view must be joined contiguously. Consider the following view and query:

  ```
  CREATE VIEW emp_job_view as
     SELECT {+ORDERED}
     emp.job_num, job.job_name
     FROM emp, job
     WHERE emp.job_num = job.job_num;

  SELECT * from dept, emp_job_view,  project
     WHERE dept.dept_no = project.dept_num
     AND emp_job_view.job_num = project.job_num;
  ```

  The ORDERED directive specifies that the **emp** table come before the job table. The directive does not affect the order of the **dept** and **project** table. Therefore, all possible join orders are as follows:

  - **emp**, **job**, **dept**, **project**
  - **emp**, **job**, **project**, **dept**

- **project**, **emp**, **job**, **dept**
- **dept**, **emp**, **job**, **project**
- **dept**, **project**, **emp**, **job**
- **project**, **dept**, **emp**, **job**

- The ORDERED directive is in a query that contains a view.

  If an ORDERED directive appears in a query that contains a view, the join order of the tables in the query are the same as they are listed in the SELECT statement. The tables within the view are joined as they are listed within the view.

  In the following query, the join order is **dept**, **project**, **emp**, **job**:

```
CREATE VIEW emp_job_view AS
   SELECT
   emp.job_num, job.job_name
   FROM emp, job
   WHERE emp.job_num = job.job_num;
SELECT {+ORDERED}
   * FROM dept, project, emp_job_view
   WHERE dept.dept_no = project.dept_num
   AND emp_job_view.job_num = project.job_num;
```

  An exception to this rule is when the view cannot be folded into the query, as in the following example:

```
CREATE VIEW emp_job_view2 AS
   SELECT DISTINCT
   emp.job_num, job.job_name
   FROM emp,job
   WHERE emp.job_num = job.job_num;
```

  In this example, the database server executes the query and puts the result in a temporary table. The order of tables in this query is **dept**, **project**, **temp_table**.

Join-method directives

The join-method directives influence how the database server joins two tables in a query.

The following directives influence the join method between two tables:

- USE_NL

  Use the listed tables as the inner table in a nested-loop join.

- USE_HASH

  Access the listed tables with a hash join. You can also choose whether the table is used to create the hash table or to probe the hash table.

- AVOID_NL

  Do not use the listed tables as the inner table in a nested-loop join. A table listed with this directive can still participate in a nested-loop join as an outer table.

- AVOID_HASH

  Do not access the listed tables with a hash join. Optionally, you can allow a hash join but restrict the table from being the one that is probed or the table from which the hash table is built.

You can specify the keyword /BUILD after the name of a table in a USE_HASH or AVOID_HASH optimizer directives:

- With USE_HASH directives, the /BUILD keyword tells the optimizer to use the specified table to build the hash table.
- With AVOID_HASH, the /BUILD keyword tells the optimizer to avoid using the specified table to build the hash table.

You can specify the keyword /PROBE after the name of a table in a USE_HASH or AVOID_HASH optimizer directives:

- With USE_HASH directives, the /PROBE keyword tells the optimizer to use the specified table to probe the hash table.
- With AVOID_HASH directives, the /PROBE keyword tells the optimizer to avoid using the specified table to probe the hash table.

Optimization-goal directives

In some queries, you might want to find only the first few rows in the result of a query. Or, you might know that all rows must be accessed and returned from the query. You can use the optimization-goal directives to find the first row that satisfies the query or all rows that satisfy the query.

For example, you might want to find only the first few rows in the result of a query, because the program opens a cursor for the query and performs a FETCH to find only the first row.

Use the optimization-goal directives to optimize the query for either one of these cases:

- FIRST_ROWS

  Choose a plan that optimizes the process of finding only the first row that satisfies the query.

- ALL_ROWS

  Choose a plan that optimizes the process of finding all rows (the default behavior) that satisfy the query.

If you use the FIRST_ROWS directive, the optimizer might abandon a query plan that contains activities that are time-consuming up front. For example, a hash join might take too much time to create the hash table. If only a few rows must be returned, the optimizer might choose a nested-loop join instead.

In the following example, assume that the database has an index on **employee.dept_no** but not on **department.dept_no**. Without directives, the optimizer chooses a hash join.

```
SELECT *
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

However, with the FIRST_ROWS directive, the optimizer chooses a nested-loop join because of the high initial overhead required to create the hash table.

```
SELECT {+first_rows} *
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

Star-join directives

Star-join directives can specify how the query optimizer joins two or more tables, among which one or more dimension tables have foreign-key dependencies on one or more fact tables.

The following directives can influence the join plan for tables that logically participate in a star schema or in a snowflake schema:

- FACT

  The optimizer considers a query plan in which the specified table is a fact table in a star-join execution plan.

- AVOID_FACT

  The optimizer does not consider a star-join execution plan that treats the specified table (or any of the tables in the list of tables) as a fact table.

- STAR_JOIN

  The optimizer favors a star-join execution plan, if available.

- AVOID_STAR_JOIN

  The optimizer chooses a query execution plan that is not a star-join plan.

These star-join directives have no effect unless the parallel database query feature (PDQ) is enabled.

Related information

Star-Join Directives on page

Concepts of dimensional data modeling on page

Keys to join the fact table with the dimension tables on page

Use the snowflake schema for hierarchical dimension tables on page

EXPLAIN directives

You can use the EXPLAIN directives to display the query plan that the optimizer chooses, and you can specify to display the query plan without running the query.

You can use these directives:

- EXPLAIN

  Displays the query plan that the optimizer chooses.

- EXPLAIN AVOID_EXECUTE

  Displays the query plan that the optimizer chooses, but does not run the query.

When you want to display the query plan for one SQL statement only, use these EXPLAIN directives instead of the SET EXPLAIN ON or SET EXPLAIN ON AVOID_EXECUTE statements.

When you use AVOID_EXECUTE (either the directive or in the SET EXPLAIN statement), the query does not execute but displays the following message:

```
No rows returned.
```

Figure 58: Result of EXPLAIN AVOID_EXECUTE directives on page 337 shows sample output for a query that uses the EXPLAIN AVOID_EXECUTE directive.

Figure 58. Result of EXPLAIN AVOID_EXECUTE directives

```
QUERY:
------
select --+ explain avoid_execute
  l.customer_num, l.lname, l.company,
  l.phone, r.call_dtime, r.call_descr
from customer l, cust_calls r
where l.customer_num = r.customer_num

DIRECTIVES FOLLOWED:
EXPLAIN
AVOID_EXECUTE
DIRECTIVES NOT FOLLOWED:

Estimated Cost: 7
Estimated # of Rows Returned: 7

  1) informix.r: SEQUENTIAL SCAN

  2) informix.l: INDEX PATH

    (1) Index Keys: customer_num   (Serial, fragments: ALL)
        Lower Index Filter: informix.l.customer_num = informix.r.customer_num
NESTED LOOP JOIN
```

The following table describes the pertinent output lines in Figure 58: Result of EXPLAIN AVOID_EXECUTE directives on page 337 that describe the chosen query plan.

Output Line in Figure 58: Result of EXPLAIN AVOID_EXECUTE directives on page 337	**Chosen Query Plan Description**
`DIRECTIVES FOLLOWED: EXPLAIN AVOID_EXECUTE`	Use the directives EXPLAIN and AVOID_EXECUTE to display the query plan and do not execute the query.
`Estimated # of Rows Returned: 7`	Estimate that this query returns seven rows.

Output Line in Figure 58: Result of EXPLAIN AVOID_EXECUTE directives on page 337	Chosen Query Plan Description
`Estimated Cost: 7`	This estimated cost of `7` is a value that the optimizer uses to compare different query plans and select the one with the lowest cost.
`1) informix.r: SEQUENTIAL SCAN`	Use the **cust_calls r** table as the outer table and scan it to obtain each row.
`2) informix.l: INDEX PATH`	For each row in the outer table, use an index to obtain the matching row(s) in the inner table **customer l**.
`(1) Index Keys: customer_num (Serial, fragments: ALL)`	Use the index on the **customer_num** column, scan it serially, and scan all fragments (the **customer l** table consists of only one fragment).
`Lower Index Filter: informix.l.customer_num = informix.r.customer_num`	Start the index scan at the **customer_num** value from the outer table.

Example of directives that can alter a query plan

Directives can alter the query plan. You can use particular directives to force the optimizer to choose a particular type of query plan, for example one that uses hash joins and the order of tables as they appear in the query.

The following example shows how directives can alter the query plan.

Suppose you have the following query:

```
SELECT * FROM emp,job,dept
WHERE emp.location = 10
   AND emp.jobno = job.jobno
   AND emp.deptno = dept.deptno
   AND dept.location = "DENVER";
```

Assume that the following indexes exist:

```
ix1: emp(empno,jobno,deptno,location)
ix2: job(jobno)
ix3: dept(location)
```

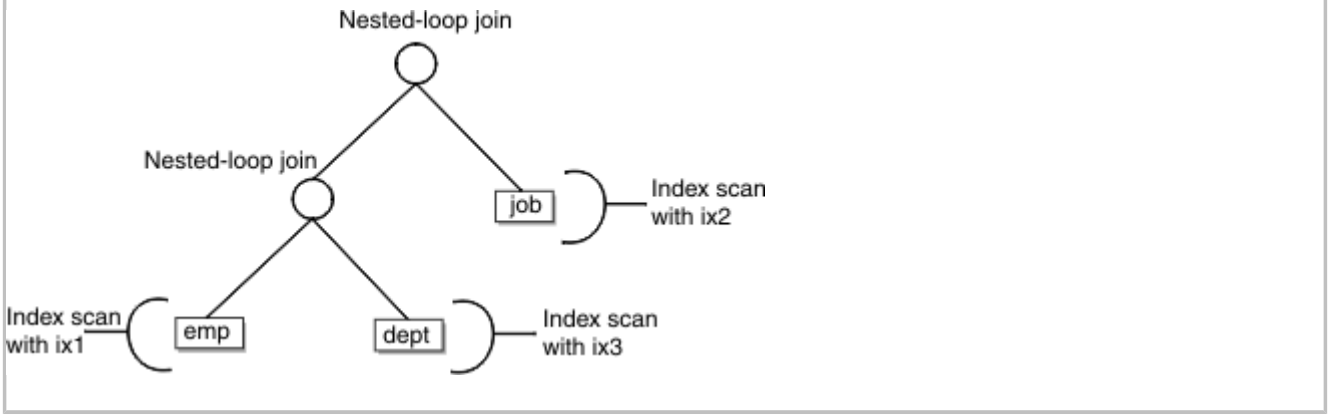
You run the query with SET EXPLAIN ON to display the query path that the optimizer uses.

```
QUERY:
------
SELECT * FROM emp,job,dept
WHERE emp.location = "DENVER"
   AND emp.jobno = job.jobno
   AND emp.deptno = dept.deptno
   AND dept.location = "DENVER"

Estimated Cost: 5
Estimated # of Rows Returned: 1
```

```
1) informix.emp: INDEX PATH

    Filters: informix.emp.location = 'DENVER'

    (1) Index Keys: empno jobno deptno location   (Key-Only)

2) informix.dept: INDEX PATH

    Filters: informix.dept.deptno = informix.emp.deptno

    (1) Index Keys: location
        Lower Index Filter: informix.dept.location = 'DENVER'
NESTED LOOP JOIN

3) informix.job: INDEX PATH

    (1) Index Keys: jobno   (Key-Only)
        Lower Index Filter: informix.job.jobno = informix.emp.jobno
NESTED LOOP JOIN
```

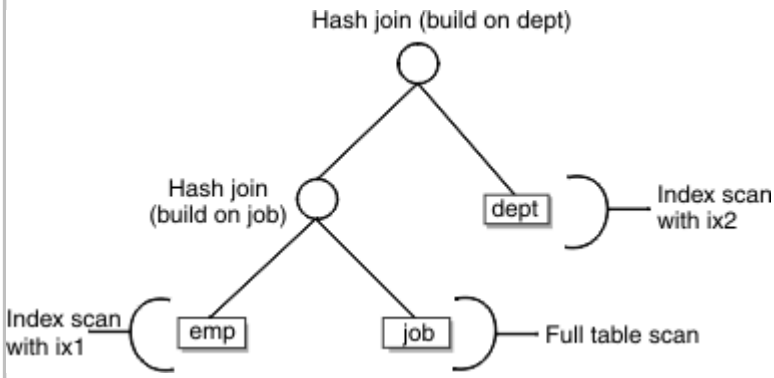
The diagram in Figure 59: Possible query plan without directives on page 339 shows a possible query plan for this query. The query plan has three levels of information: (1) a nested-loop join, (2) an index scan on one table and a nested-loop join, and (3) index scans on two other tables.

Figure 59. Possible query plan without directives



Perhaps you are concerned that using a nested-loop join might not be the fastest method to execute this query. You also think that the join order is not optimal. You can force the optimizer to choose a hash join and order the tables in the query plan according to their order in the query, so the optimizer uses the query plan that Figure 60: Possible query plan with directives on page 340 shows. This query plan that has three levels of information: (1) a hash join, (2) an index scan and a hash join, and (3) an index scan on two other tables.

Figure 60. Possible query plan with directives



To force the optimizer to choose the query plan that uses hash joins and the order of tables shown in the query, use the directives that the following partial SET EXPLAIN output shows:

```
QUERY:
------
SELECT {+ORDERED,
   INDEX(emp ix1),
   FULL(job),
   USE_HASH(job /BUILD),
   USE_HASH(dept /BUILD),
   INDEX(dept ix3)}
   * FROM emp,job,dept
   WHERE emp.location = 1
   AND emp.jobno = job.jobno
   AND emp.deptno = dept.deptno
   AND dept.location = "DENVER"

DIRECTIVES FOLLOWED:
ORDERED
INDEX ( emp ix1 )
FULL ( job )
USE_HASH ( job/BUILD )
USE_HASH ( dept/BUILD )
INDEX ( dept ix3 )
DIRECTIVES NOT FOLLOWED:

Estimated Cost: 7
Estimated # of Rows Returned: 1

1) informix.emp: INDEX PATH

    Filters: informix.emp.location = 'DENVER'

    (1) Index Keys: empno jobno deptno location   (Key-Only)

2) informix.job: SEQUENTIAL SCAN


DYNAMIC HASH JOIN
    Dynamic Hash Filters: informix.emp.jobno = informix.job.jobno
```

```
3) informix.dept: INDEX PATH

    (1) Index Keys: location
        Lower Index Filter: informix.dept.location = 'DENVER'

DYNAMIC HASH JOIN
    Dynamic Hash Filters: informix.emp.deptno = informix.dept.deptno
```

Configuration parameters and environment variables for optimizer directives

You can use the DIRECTIVES configuration parameter to turn on or off all directives that the database server encounters, and you can use the **IFX_DIRECTIVES** environment variable to override the setting of the DIRECTIVES configuration parameter.

If the DIRECTIVES configuration parameter is set to `1` (the default), the optimizer follows all directives. If the DIRECTIVES configuration parameter is set to `0`, the optimizer ignores all directives.

You can override the setting of DIRECTIVES. If the **IFX_DIRECTIVES** environment variable is set to `1` or `ON`, the optimizer follows directives for any SQL the client session executes. If **IFX_DIRECTIVES** is `0` or `OFF`, the optimizer ignores directives for any SQL in the client session.

Any directives in an SQL statement take precedence over the join plan that the OPTCOMPIND configuration parameter forces. For example, if a query includes the USE_HASH directive and OPTCOMPIND is set to `0` (nested-loop joins preferred over hash joins), the optimizer uses a hash join.

Optimizer directives and SPL routines

Directives operate differently for a query in an SPL routine because a SELECT statement in an SPL routine is not necessarily optimized immediately before the database server executes it.

The optimizer creates a query plan for a SELECT statement in an SPL routine when the database server creates the SPL routine or during the execution of the UPDATE STATISTICS statement that include the `FOR FUNCTION`, `FOR PROCEDURE`, or `FOR ROUTINE` keywords.

The optimizer reads and applies directives at the time that it creates the query plan. Because it stores the query plan in a system catalog table, the SELECT statement is not reoptimized when it is executed. Therefore, settings of **IFX_DIRECTIVES** and DIRECTIVES affect SELECT statements inside an SPL routine when they are set at any of the following times:

- Before the CREATE PROCEDURE statement
- Before the UPDATE STATISTICS FOR ROUTINE statements that cause SQL data-manipulation statements in SPL routines to be optimized
- During certain circumstances when SELECT statements have variables supplied at runtime

Avoiding index or prepared object exceptions by forced reoptimization

If the AUTO_REPREPARE configuration parameter and the IFX_AUTO_REPREPARE session environment variable are enabled, OneDB automatically recompiles prepared statements and SPL routines after the schema of a referenced table is modified by a DDL statement. If the AUTO_REPREPARE configuration parameter or the IFX_AUTO_REPREPARE session environment variable is disabled, you can take steps to prevent errors.

If the AUTO_REPREPARE configuration parameter or the IFX_AUTO_REPREPARE session environment variable is disabled, the following error can result when prepared objects or SPL routines are executed after the schema of a table referenced by the prepared object or indirectly referenced by the SPL routine has been modified.

```
-710  Table <table-name> has been dropped, altered, or renamed.
```

This error can occur with explicitly prepared statements. These statements have the following form:

```
PREPARE statement_id FROM quoted_string;
```

After a statement has been prepared in the database server and before execution of the statement, a table to which the statement refers might have been renamed or altered, possibly changing the structure of the table. Problems can occur as a result.

Adding an index to the table after preparing the statement can also invalidate the statement. A subsequent OPEN command for a cursor fails if the cursor refers to the invalid prepared statement; the failure occurs even if the OPEN command has the WITH REOPTIMIZATION clause.

If an index was added after the statement was prepared, you must prepare the statement again and declare the cursor again. You cannot simply reopen the cursor if it was based on a prepared statement that is no longer valid.

This error can also occur with SPL routines. Before the database server executes a new SPL routine the first time, it optimizes the code (statements) in the SPL routine. Optimization makes the code depend on the structure of the tables that the procedure references. If the table structure changes after the procedure is optimized, but before it is executed, this error can occur.

Each SPL routine is optimized the first time that it is run (not when it is created). This behavior means that an SPL routine might succeed the first time it is run but fail later under virtually identical circumstances. The failure of an SPL routine can also be intermittent, because failure during one execution forces an internal warning to reoptimize the procedure before the next execution.

The database server keeps a list of tables that the SPL routine references explicitly. Whenever any one of these explicitly referenced tables is modified, the database server reoptimizes the procedure the next time the procedure is executed.

However, if the SPL routine depends on a table that is referenced only indirectly, the database server cannot detect the need to reoptimize the procedure after that table is changed. For example, a table can be referenced indirectly if the SPL routine invokes a trigger. If a table that is referenced by the trigger (but not directly by the SPL routine) is changed, the database server does not know that it should reoptimize the SPL routine before running it. When the procedure is run after the table has been changed, this error can occur.

Use one of two methods to recover from this error:

- Issue the UPDATE STATISTICS FOR PROCEDURE statement to force reoptimization of the procedure.
- Rerun the procedure.

To prevent this error, you can force reoptimization of the SPL routine. For example, to force reoptimization of an SPL routine called **procedure_name**, execute the following statement:

```
UPDATE STATISTICS FOR PROCEDURE procedure_name;
```

Note that the following UPDATE STATISTICS statement has the same effect:

```
UPDATE STATISTICS FOR ROUTINE procedure_name;
```

Important:

Keep in mind that in databases that use transaction logging, you must run the UPDATE STATISTICS statement in a transaction that does not contain any other SQL statements.

You can add this statement to your program in either of the following ways:

- Place the UPDATE STATISTICS statement after each DDL statement that changes the mode of an object.
- Place the UPDATE STATISTICS statement before each execution of the SPL routine.

For efficiency, you can put the UPDATE STATISTICS statement with the action that occurs less frequently in the program (change of object mode or execution of the procedure). In most cases, the action that occurs less frequently in the program is the change of object mode.

When you follow this method of recovering from this error, you must execute the UPDATE STATISTICS FOR PROCEDURE statement for each procedure that references the changed tables indirectly, unless the procedure also references the tables explicitly.

You can also recover from this error by simply rerunning the SPL routine. The first time that the stored procedure fails, the database server marks the procedure as needing reoptimization. The next time that you run the procedure, the database server reoptimizes the procedure before running it. However, running the SPL routine twice might not be practical or safe. A safer choice is to use the UPDATE STATISTICS FOR PROCEDURE statement to force reoptimization of the procedure.

External optimizer directives

If you are user **informix**, you can create, save, and delete external directives.

About this task

Creating and saving external directives

You can define external directives by creating *association records* that include query optimizer directives, and saving those records in the **sysdirectives** system catalog table. Association records associate a list of one or more optimizer directives with a specific query text. The database server can apply those optimizer directives to subsequent instances of the same query text.

Use the SAVE EXTERNAL DIRECTIVES statement to create the association record to use for the list of one or more query directives These directives are applied automatically to subsequent instances of the same query.

The following example shows a SAVE EXTERNAL DIRECTIVES statement that registers an association-record in the system catalog as a new row in the **sysdirectives** table that can be used as a query optimizer directive.

```
SAVE EXTERNAL DIRECTIVES {+INDEX(t1,i11)} ACTIVE FOR
   SELECT {+INDEX(t1, i2) } c1 FROM t1 WHERE c1=1;
```

The following data is stored in the association record that the SQL statement above defined:

```
id              16
query           select {+INDEX(t1, i2) } c1 from t1 where c1=1
directive       INDEX(t1,i11)
directivecode   BYTE value
```

```
active          1
hashcode        -589336273
```

Here `{+INDEX(t1,i11)}`, the external directive that followed the DIRECTIVES keyword, will be applied to future instances of the specified query, but the inline `{+INDEX(t1,i2)}` directive will be ignored.

The information in the external directives that immediately follow the DIRECTIVES keyword must be within comment indicators, just as the same directives would appear in SELECT, UPDATE, MERGE, and DELETE statements, except that blank characters, rather than comma (`,`) symbols, are the required separators if the list of external directives includes more than one directive.

Enabling external directives

After you create and save external directives, you must set the configuration parameter and environmental variable that enable the directives. The database server searches for a directive for a query only if the external directives are set on both the database server and the client.

Enable the directive by using a combination of the EXT_DIRECTIVES configuration parameter, which is in the ONCONFIG file, and the **IFX_EXTDIRECTIVES** client-side environment variable.

The EXT_DIRECTIVE values that you can use are:

Value	Explanation
`0` (default)	Off. The directive cannot be enabled, even if IFX_EXTDIRECTIVES is enabled.
`1`	On. The directive can be enabled for a session if IFX_EXTDIRECTIVES is enabled.
`2`	On. The directive can be used even if IFX_EXTDIRECTIVES is not enabled.

You can also use the EXTDIRECTIVES option of the SET ENVIRONMENT statement to enable or disable external directives during a session. What you specify with the EXTDIRECTIVES option overwrites the external directive setting that is specified in the EXT_DIRECTIVES configuration parameter in the ONCONFIG file.

To overwrite the value for enabling or disabling the external directive in the ONCONFIG file:

- To enable the external directives during a session, specify `1`, `on`, or `ON` as the value for SET ENVIRONMENT EXTDIRECTIVES.
- To disable the external directives during a session, specify `0`, `off`, or `OFF` as the value for SET ENVIRONMENT EXTDIRECTIVES.

To enable the default values specified in the EXT_DIRECTIVES configuration parameter and in the client-side IFX_EXTDIRECTIVES environment variable during a session, specify DEFAULT as the value for the EXTDIRECTIVES option of the SET ENVIRONMENT statement.

The explain output file specifies whether external directives are in effect.

Related information

The explain output file on page 300

Query statistics section provides performance debugging information on page 301

Report that shows the query plan chosen by the optimizer on page 299

SET EXPLAIN statement on page

Using the FILE TO option on page

Default name and location of the explain output file on UNIX on page

Default name and location of the output file on Windows on page

onmode -Y: Dynamically change SET EXPLAIN on page

onmode and Y arguments: Change query plan measurements for a session (SQL administration API) on page

Deleting external directives

When you no longer need an external directive, the DBA or user **informix** can use the DELETE statement of SQL to remove it from the **sysdirectives** system catalog table.

When external directives are enabled and the **sysdirectives** system catalog table is not empty,

- the database server compares every query with the query text of every ACTIVE external directive,
- and for queries executed by the DBA (or by user **informix**) with every TEST ONLY external directive.

The purpose of external directives is to improve the performance of queries that match the query string, but the use of such directives can potentially slow other queries, if the query execution optimizer must compare the query strings of a large number of active external directives with the text of every SELECT statement. For this reason, HCL recommends that the DBA not allow the **sysdirectives** table to accumulate more than a few ACTIVE rows. (An alternative way to avoid unintended performance impact on other queries is to disable support for external directives by setting the EXT_DIRECTIVES configuration parameter to `0`. Setting the **IFX_EXTDIRECTIVES** client environment variable to `0` has the same effect.)

Parallel database query (PDQ)

You can manage how the database server performs PDQ and you can monitor the resources that the database server uses for PDQ.

What PDQ is

Parallel database query (PDQ) is a database server feature that can improve performance dramatically when the server processes queries that decision-support applications initiate. PDQ enables OneDB to distribute the work for one aspect of a query among several processors. For example, if a query requires an aggregation, OneDB can distribute the work for the aggregation among several processors.

PDQ also includes tools for resource management.

Another database server feature, *table fragmentation*, allows you to store the parts of a table on different disks. PDQ delivers maximum performance benefits when the data that you query is in fragmented tables. For information about how to use fragmentation for maximum performance, see Planning a fragmentation strategy on page 260.

Related information

Database server operations that use PDQ on page 347

The allocation of resources for parallel database queries on page 352

Managing PDQ queries on page 358

Monitoring resources used for PDQ and DSS queries on page 361

Structure of a PDQ query

Each decision-support query has a primary thread. The database server can start additional threads to perform tasks for the query (for example, scans and sorts). Depending on the number of tables or fragments that a query must search and the resources that are available for a decision-support query, the database server assigns different components of a query to different threads.

The database server initiates these PDQ threads, which are listed as *secondary threads* in the SET EXPLAIN output.

Secondary threads are further classified as either *producers* or *consumers*, depending on their function. A producer thread supplies data to another thread. For example, a scan thread might read data from shared memory that corresponds to a given table and pass it along to a join thread. In this case, the scan thread is considered a producer, and the join thread is considered a consumer. The join thread, in turn, might pass data along to a sort thread. When doing so, the join thread is considered a producer, and the sort thread is considered a consumer.

Several producers can supply data to a single consumer. When this situation occurs, the database server sets up an internal mechanism, called an *exchange*, that synchronizes the transfer of data from those producers to the consumer. For instance, if a fragmented table is to be sorted, the optimizer typically calls for a separate scan thread for each fragment. Because of different I/O characteristics, the scan threads can be expected to complete at different times. An exchange is used to funnel the data produced by the various scan threads into one or more sort threads with a minimum of buffering. Depending on the complexity of the query, the optimizer might call for a multilayered hierarchy of producers, exchanges, and consumers. Generally speaking, consumer threads work in parallel with producer threads so that the amount of intermediate buffering that the exchanges perform remains negligible.

The database server creates these threads and exchanges automatically and transparently. They terminate automatically as they complete processing for a given query. The database server creates new threads and exchanges as needed for subsequent queries.

Database server operations that use PDQ

HCL OneDB™ processes some types of SQL operations that the database server processes in parallel. However some situations limit the degree of parallelism that HCL OneDB™ can use.

In the topics on database server operations that use PDQ in this section, a *query* is any SELECT statement.

Related information

What PDQ is on page 346

Parallel update and delete operations

OneDB performs some types of update and delete operations in parallel.

The database server takes the following two steps to process UPDATE and DELETE statements:

1. Fetches the qualifying rows.
2. Applies the action of updating or deleting.

The database server performs the first step of an UPDATE or DELETE statement in parallel, with the following exceptions:

- The targeted table in a DELETE statement has a referential constraint that can cascade to a child table.
- The UPDATE or DELETE statement contains an OR clause and the optimizer chooses an OR index to process the OR filter.
- The UPDATE statement contains a subquery that the optimizer converts into a join.

Parallel insert operations

OneDB performs some types of insert operations in parallel.

The types of insert operations that the server performs in parallel are:

- SELECT...INTO TEMP inserts using explicit temporary tables.
- INSERT INTO...SELECT inserts using implicit temporary tables.

Explicit inserts with SELECT...INTO TEMP statements

The database server can insert rows in parallel into explicit temporary tables that you specify in SQL statements of the form SELECT....INTO TEMP.

About this task

For example, the database server can perform the inserts in parallel into the temporary table, **temp_table**, as the following example shows:

```
SELECT * FROM table1 INTO TEMP temp_table
```

To perform parallel inserts into a temporary table:

1. Set PDQ priority > `0`.

   You must meet this requirement for any query that you want the database server to perform in parallel.

2. Set DBSPACETEMP to a list of two or more dbspaces.

   This step is required because of the way that the database server performs the insert. To perform the insert in parallel, the database server first creates a fragmented temporary table. So that the database server knows where to store the fragments of the temporary table, you must specify a list of two or more dbspaces in the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable. In addition, you must set DBSPACETEMP to indicate storage space for the fragments before you execute the SELECT...INTO statement.

Results

The database server performs the parallel insert by writing in parallel to each of the fragments in a round-robin fashion. Performance improves as you increase the number of fragments.

Implicit inserts with INSERT INTO...SELECT statements

The database server can also insert rows in parallel into implicit tables that it creates when it processes SQL statements of the form INSERT INTO...SELECT.

For example, the database server processes the following INSERT statement in parallel:

```
INSERT INTO target_table SELECT * FROM source_table
```

The target table can be either a permanent table or a temporary table.

The database server processes this type of INSERT statement in parallel only when the target tables meet the following criteria:

- The value of PDQ priority is greater than `0.`
- The target table is fragmented into two or more dbspaces.
- The target table has no enabled referential constraints or triggers.
- The target table is not a remote table.
- In a database with logging, the target table does not contain filtering constraints.
- The target table does not contain columns of TEXT or BYTE data type.

The database server does not process parallel inserts that reference an SPL routine. For example, the database server never processes the following statement in parallel:

```
INSERT INTO table1 EXECUTE PROCEDURE ins_proc
```

Parallel index builds

Index builds can take advantage of PDQ and can be parallelized. The database server performs both scans and sorts in parallel for index builds.

The following operations initiate index builds:

- Create an index.
- Add a unique, primary key.
- Add a referential constraint.
- Enable a referential constraint.

When PDQ is in effect, the scans for index builds are controlled by the PDQ configuration parameters described in The allocation of resources for parallel database queries on page 352.

If you have a computer with multiple CPUs, the database server uses two sort threads to sort the index keys. The database server uses two sort threads during index builds without the user setting the **PSORT_NPROCS** environment variable.

Parallel user-defined routines

If a query contains a user-defined routine (UDR) in an expression, the database server can execute a query in parallel when you turn on PDQ.

The database server can perform the following parallel operations if the UDR is written and registered appropriately:

- Parallel scans
- Parallel comparisons with the UDR

For more information about how to enable parallel execution of UDRs, see #unique_579.

Hold cursors that use PDQ

When hold cursors that are created by declaring the WITH HOLD qualifier have no locks, PDQ is enabled.

PDQ will be set for hold cursors in the following cases:

- Queries with Dirty Read or Committed Read isolation level, ANSI, and read-only cursor
- Queries with Dirty Read or Committed Read isolation level, NON-ANSI, non-updateable cursor

SQL operations that do not use PDQ

The database server does not process some types of queries in parallel.

For example, the server does not process the following types of queries in parallel:

- Queries started with an isolation level of Cursor Stability

  Subsequent changes to the isolation level do not affect the parallelism of queries already prepared. This situation results from the inherent nature of parallel scans, which scan several rows simultaneously.

- Queries that use a cursor declared as FOR UPDATE
- Queries in the `FOR EACH ROW` section of the Action clause of a *Select* trigger
- A DELETE or MERGE statement in the `FOR EACH ROW` section of the Action clause of a *Delete* trigger
- An INSERT or MERGE statement in the `FOR EACH ROW` section of the Action clause of an *Insert* trigger
- An UPDATE or MERGE statement in the `FOR EACH ROW` section of the Action clause of an *Update* trigger
- Data definition language (DDL) statements.

  For a complete list of the DDL statements of SQL that HCL OneDB™ supports, see the *HCL OneDB™ Guide to SQL: Syntax*.

In addition, the database server does not process sequence objects in PDQ operations. If your SQL statement includes sequencing operations, such as expressions that include the **NEXTVAL** or **CURRVAL** operators, PDQ processing is unavailable for that statement.

Update statistics operations affected by PDQ

An SQL UPDATE STATISTICS statement that is not processed in parallel, is affected by PDQ because it must allocate the memory used for sorting. Thus the behavior of the UPDATE STATISTICS statement is affected by the memory management associated with PDQ.

The database server must allocate the memory that the UPDATE STATISTICS statement uses for sorting.

If you have an extremely large database and indexes are fragmented, UPDATE STATISTICS LOW can automatically run statements in parallel. For more information, see Update statistics in parallel on very large databases on page 384.

SPL routines and triggers and PDQ

Statements that involve SPL routines are not executed in parallel. However, statements within procedures are executed in parallel.

When the database server executes an SPL routine, it does not use PDQ to process non-related SQL statements contained in the procedure. Each SQL statement can be executed independently in parallel, however, using intraquery parallelism when possible. As a consequence, you should limit the use of procedure calls from within data manipulation language (DML) statements if you want to use the parallel-processing abilities of the database server. For a complete list of DML statements, see the *HCL OneDB™ Guide to SQL: Syntax*.

The database server uses intraquery parallelism to process the statements in the body of an SQL trigger in the same way that it processes statements in SPL routines. For restrictions on using PDQ for queries in some triggered actions of Select, Insert, and Update triggers, see SQL operations that do not use PDQ on page 349.

Correlated and uncorrelated subqueries

The database server does not use PDQ to process correlated subqueries. Only one thread at a time can execute a correlated subquery. While one thread executes a correlated subquery, other threads that request to execute the subquery are blocked until the first one completes.

For uncorrelated subqueries, only the first thread that makes the request actually executes the subquery. Other threads simply use the results of the subquery and can do so in parallel.

As a consequence, it is strongly recommended that, whenever possible, you use joins rather than subqueries to build queries so that the queries can take advantage of PDQ.

OUTER index joins and PDQ

The database server reduces the PDQ priority of queries that contain OUTER index joins to LOW (if the priority is set to a higher value) for the duration of the query. If a subquery or a view contains OUTER index joins, the database server lowers the PDQ priority of only that subquery or view, not of the parent query or any other subquery.

Remote tables used with PDQ

Although the database server can process the data stored in a remote table in parallel, the data is communicated serially because the database server allocates a single thread to submit and receive the data from the remote table. The database server lowers the PDQ priority of queries that require access to a remote database to LOW.

In this case, all local scans are parallel, but all local joins and remote access are nonparallel.

The Memory Grant Manager

The Memory Grant Manager (MGM) is a database server component that coordinates the use of memory, CPU virtual processors (VPs), disk I/O, and scan threads among decision-support queries. The MGM uses the DS_MAX_QUERIES, DS_TOTAL_MEMORY, DS_MAX_SCANS, and MAX_PDQPRIORITY configuration parameters to determine the quantity of these PDQ resources that can be granted to a decision-support query.

The MGM dynamically allocates the following resources for decision-support queries:

- The number of scan threads that are started for each decision-support query
- The number of threads that can be started for each query
- The amount of memory in the virtual portion of database server shared memory that the query can reserve

When your database server system has heavy OLTP use, and you find performance is degrading, you can use the MGM facilities to limit the resources that are committed to decision-support queries. During off-peak hours, you can designate a larger proportion of the resources to parallel processing, which achieves higher throughput for decision-support queries.

The MGM grants memory to a query for such activities as sorts, hash joins, and processing of GROUP BY clauses. The amount of memory that decision-support queries use cannot exceed DS_TOTAL_MEMORY.

The MGM grants memory to queries in *quantum* increments. To calculate the approximate size of the quantum, use the following formula:

```
memory quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

For example, if DS_TOTAL_MEMORY is 12 MB and DS_MAX_QUERIES is 4, the quantum is 3 MB (12/4). Thus, with these values in effect, a quantum of memory equals 3 MB. The database server can adjust the size of the quantum dynamically when it grants memory. In general, memory is allocated more efficiently when quanta are smaller. You can often improve performance of concurrent queries by increasing DS_MAX_QUERIES to reduce the size of a quantum of memory.

To monitor resources that the MGM allocates, run the onstat -g mgm command. This command shows only the amount of memory that is used; it does not show the amount of memory that is granted.

The MGM also grants a maximum number of scan threads per query that is based on the values of the DS_MAX_SCANS and the DS_MAX_QUERIES parameters.

The following formula yields the maximum number of scan threads per query:

```
scan_threads = min (nfrags, DS_MAX_SCANS * (pdqpriority / 100)
   * (MAX_PDQPRIORITY / 100))
```

nfrags

Is the number of fragments in the table with the largest number of fragments.

pdqpriority

Is the value for PDQ priority that is set by either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

The **PDQPRIORITY** environment variable and the SQL statement SET PDQPRIORITY request a percentage of PDQ resources for a query. You can use the MAX_PDQPRIORITY configuration parameter to limit the percentage of the requested resources that a query can obtain and to limit the impact of decision-support queries on OLTP processing.

Related reference

onstat -g mgm command: Print MGM resource information

Related information

Effect of configuration on memory utilization on page 60

Limiting the priority of decision-support queries on page 353

The DS_TOTAL_MEMORY configuration parameter and memory utilization on page 74

The allocation of resources for parallel database queries

When you configure the database server, consider how the use of PDQ affects users of OLTP, decision-support (DSS) applications, and other applications. Then you can plan how to allocate resources for PDQ.

When the database server uses PDQ to perform a query in parallel, it puts a heavy load on the operating system. In particular, PDQ exploits the following resources:

- Memory
- CPU VPs
- Disk I/O (to fragmented tables and temporary table space)
- Scan threads

You can control how the database server uses resources in the following ways:

- Limit the priority of parallel database queries.
- Adjust the amount of memory.
- Limit the number of scan threads.
- Limit the number of concurrent queries.

Related information

What PDQ is on page 346

Limiting the priority of decision-support queries

You can limit the parallel processing resources that decision-support (DSS) queries consume by adjusting the values of **PDQPRIORITY** environment variable, the MAX_PDQPRIORITY configuration parameter, and other configuration parameters.

The default value for the PDQ priority of individual applications is `0`, which means that PDQ processing is not used. The database server uses this value unless one of the following actions overrides it:

- You set the **PDQPRIORITY** environment variable.
- The application uses the SET PDQPRIORITY statement.

The **PDQPRIORITY** environment variable and the MAX_PDQPRIORITY configuration parameter work together to control the amount of resources to allocate for parallel processing. Setting these correctly is critical for the effective operation of PDQ.

The MAX_PDQPRIORITY configuration parameter allows you to limit the parallel processing resources that DSS queries consume. Thus, the **PDQPRIORITY** environment variable sets a *reasonable* or *recommended* priority value, and MAX_PDQPRIORITY limits the resources that an application can claim.

The MAX_PDQPRIORITY configuration parameter specifies the maximum percentage of the requested resources that a query can obtain. For instance, if **PDQPRIORITY** is `80` and MAX_PDQPRIORITY is `50`, each active query receives an amount of memory equal to 40 percent of DS_TOTAL_MEMORY, rounded down to the nearest quantum. In this example, MAX_PDQPRIORITY effectively limits the number of concurrent decision-support queries to two. Subsequent queries must wait for earlier queries to finish before they acquire the resources that they need to run.

An application or user can use the DEFAULT tag of the SET PDQPRIORITY statement to use the value for PDQ priority if the value has been set by the **PDQPRIORITY** environment variable. DEFAULT is the symbolic equivalent of a `-1` value for PDQ priority.

You can use the **onmode** command-line utility to change the values of the following configuration parameters temporarily:

- Use **onmode -M** to change the value of DS_TOTAL_MEMORY.
- Use **onmode -Q** to change the value of DS_MAX_QUERIES.
- Use **onmode -D** to change the value of MAX_PDQPRIORITY.
- Use **onmode -S** to change the value of DS_MAX_SCANS.

These changes remain in effect only as long as the database server remains up and running. When the database server starts, it uses the values listed in the ONCONFIG file.

For more information about the preceding parameters, see Effect of configuration on memory utilization on page 60. For more information about **onmode**, see your *HCL OneDB™ Administrator's Reference*.

If you must change the values of the decision-support parameters on a regular basis (for example, to set MAX_PDQPRIORITY to `100` each night for processing reports), you can use a scheduled operating-system job to set the values. For information about creating scheduled jobs, see your operating-system manuals.

To obtain the best performance from the database server, choose values for the **PDQPRIORITY** environment variable and MAX_PDQPRIORITY parameter, observe the resulting behavior, and then adjust the values for these parameters. No well-defined rules exist for choosing these environment variable and parameter values. The following sections discuss strategies for setting **PDQPRIORITY** and MAX_PDQPRIORITY for specific needs.

Related information

The Memory Grant Manager on page 351

Limiting the value of the PDQ priority

You can adjust the value of the MAX_PDQPRIORITY configuration parameter to adjust the PDQ priority and allocate more resources to either OLTP or decision-support processing.

The MAX_PDQPRIORITY configuration parameter limits the PDQ priority that the database server grants when users either set the **PDQPRIORITY** environment variable or issue the SET PDQPRIORITY statement before they issue a query. When an application or an end user attempts to set a PDQ priority, the priority that is granted is multiplied by the value that MAX_PDQPRIORITY specifies.

Set the value of MAX_PDQPRIORITY lower when you want to allocate more resources to OLTP processing.

Set the value of MAX_PDQPRIORITY higher when you want to allocate more resources to decision-support processing.

The possible range of values is 0 to 100. This range represents the percent of resources that you can allocate to decision-support processing.

Maximizing OLTP throughput for queries

At times, you might want to allocate resources to maximize the throughput for individual OLTP queries rather than for decision-support queries.

In this case, set MAX_ PDQPRIORITY to `0`, which limits the value of PDQ priority to `OFF`. A PDQ priority value of `OFF` does not prevent decision-support queries from running. Instead, it causes the queries to run without parallelization. In this configuration, response times for decision-support queries might be slow.

Conserving resources when using PDQ

If applications make little use of queries that require parallel sorts and parallel joins, consider using the `LOW` setting for PDQ priority.

If the database server is operating in a multiuser environment, you might set MAX_PDQPRIORITY to `1` to increase interquery performance at the cost of some intraquery parallelism. A trade-off exists between these two different types of parallelism because they compete for the same resources. As a compromise, you might set MAX_PDQPRIORITY to some intermediate value (perhaps `20` or `30`) and set **PDQPRIORITY** to `LOW`. The environment variable sets the default behavior to `LOW`, but the MAX_PDQPRIORITY configuration parameter allows individual applications to request more resources with the SET PDQPRIORITY statement.

Allowing maximum use of parallel processing

Set **PDQPRIORITY** and MAX_PDQPRIORITY to `100` if you want the database server to assign as many resources as possible to parallel processing.

This setting is appropriate for times when parallel processing does not interfere with OLTP processing.

Determining the level of parallel processing

You can use different numeric settings for **PDQPRIORITY** to experiment with the effects of parallelism on a single application.

For information about how to monitor parallel execution, see Monitoring resources used for PDQ and DSS queries on page 361.

Limits on parallel operations associated with PDQ priority

The database server reduces the PDQ priority of queries that contain outer joins to `LOW` (if set to a higher value) for the duration of the query. If a subquery or a view contains outer joins, the database server lowers the PDQ priority only of that subquery or view, not of the parent query or of any other subquery.

The database server lowers the PDQ priority of queries that require access to a remote database (same or different database server instance) to `LOW` if you set it to a higher value. In that case, all local scans are parallel, but all local joins and remote accesses are nonparallel.

Using SPL routines with PDQ queries

The database server freezes the PDQ priority that is used to optimize SQL statements within SPL routines at the time of procedure creation or the last manual recompilation with the UPDATE STATISTICS statement. You can change the client value of **PDQPRIORITY**.

To change the client value of **PDQPRIORITY**, embed the SET PDQPRIORITY statement within the body of your SPL routine.

The PDQ priority value that the database server uses to optimize or reoptimize an SQL statement is the value that was set by a SET PDQPRIORITY statement, which must have been executed within the same procedure. If no such statement has been executed, the value that was in effect when the procedure was last compiled or created is used.

The PDQ priority value currently in effect outside a procedure is ignored within a procedure when it is executing.

It is suggested that you turn PDQ priority off when you enter a procedure and then turn it on again for specific statements. You can avoid tying up large amounts of memory for the procedure, and you can make sure that the crucial parts of the procedure use the appropriate PDQ priority, as the following example illustrates:

```
CREATE PROCEDURE my_proc (a INT, b INT, c INT)
   Returning INT, INT, INT;
SET PDQPRIORITY 0;
...
SET PDQPRIORITY 85;
SELECT ... (big complicated SELECT statement)
SET PDQPRIORITY 0;
...
;
```

Adjusting the amount of memory for DSS and PDQ queries

You can estimate the amount of shared memory to allocate to decision-support (DSS) queries. Then, if necessary, you can adjust the value of the DS_TOTAL_MEMORY configuration parameter, which specifies the amount of memory available for PDQ queries.

Use the following formula as a starting point for estimating the amount of shared memory to allocate to DSS queries:

```
DS_TOTAL_MEMORY = p_mem – os_mem – rsdnt_mem –
                  (128 kilobytes * users) – other_mem
```

p_mem

represents the total physical memory that is available on the host computer.

os_mem

represents the size of the operating system, including the buffer cache.

resdnt_mem

represents the size of HCL OneDB™ resident shared memory.

users

is the number of expected users (connections) specified in the NETTYPE configuration parameter.

other_mem

is the size of memory used for other (non-HCL® OneDB®) applications.

The value for DS_TOTAL_MEMORY that is derived from this formula serves only as a starting point. To arrive at a value that makes sense for your configuration, you must monitor paging and swapping. (Use the tools provided with your operating

system to monitor paging and swapping.) When paging increases, decrease the value of DS_TOTAL_MEMORY so that processing the OLTP workload can proceed.

The amount of memory that is granted to a single parallel database query depends on many system factors, but in general, the amount of memory granted to a single parallel database query is proportional to the following formula:

```
memory_grant_basis = (DS_TOTAL_MEMORY/DS_MAX_QUERIES) *
               (PDQPRIORITY / 100) *
               (MAX_PDQPRIORITY / 100)
```

However, if the currently executing queries on all databases of the server instance require more memory than this estimate of the average allocation, another query might overflow to disk or might wait until concurrent queries completed execution and released sufficient memory resources for running the query. The following alternative formula estimates the PDQ memory available for a single query directly:

```
memory_for_single_query = DS_TOTAL_MEMORY  *
               (PDQPRIOIRTY / 100) *
               (MAX_PDQPRIORITY / 100)
```

Limiting the number of concurrent scans

The database server apportions some number of scans to a query according to its PDQ priority (among other factors). You can adjust the value of the DS_MAX_SCANS configuration parameter to limit the number of concurrent scans.

The DS_MAX_SCANS and MAX_PDQPRIORITY configuration parameters allow you to limit the resources that users can assign to a query, according to the following formula:

```
scan_threads = min (nfrags, (DS_MAX_SCANS * (pdqpriority / 100)
   * (MAX_PDQPRIORITY / 100) )
```

nfrags

is the number of fragments in the table with the largest number of fragments.

pdqpriority

is the PDQ priority value set by either the **PDQPRIORITY** environment variable or the SET PDQPRIORITY statement.

For example, suppose a large table contains 100 fragments. With no limit on the number of concurrent scans allowed, the database server would concurrently execute 100 scan threads to read this table. In addition, many users could initiate this query.

As the database server administrator, you set the DS_MAX_SCANS configuration parameter to a value lower than the number of fragments in this table to prevent the database server from being flooded with scan threads by multiple decision-support queries. You can set DS_MAX_SCANS to `20` to ensure that the database server concurrently executes a maximum of 20 scan threads for parallel scans. Furthermore, if multiple users initiate parallel database queries, each query receives only a percentage of the 20 scan threads, according to the PDQ priority assigned to the query and the value for the MAX_PDQPRIORITY configuration parameter that the database server administrator sets.

Limiting the maximum number of PDQ queries

You can adjust the maximum number of PDQ queries that can run concurrently by changing the value of the DS_MAX_QUERIES configuration parameter.

The DS_MAX_QUERIES configuration parameter limits the number of concurrent decision-support queries that can run.

To estimate the number of decision-support (DSS) queries that the database server can run concurrently, count each query that runs with PDQ priority set to 1 or greater as one full query.

The database server allocates less memory to queries that run with a lower priority, so you can assign lower-priority queries a PDQ priority value that is between 1 and 30, depending on the resource impact of the query. The total number of queries with PDQ priority values greater than 0 cannot exceed the value of DS_MAX_QUERIES.

Managing PDQ queries

The database server administrator, the writer of an application, and the users all have a certain amount of control over the amount of resources that OneDB allocates to processing a query. The database server administrator exerts control through the use of configuration parameters. The application developer or the user can exert control through an environment variable or SQL statement.

Related information

What PDQ is on page 346

Analyzing query plans with SET EXPLAIN output

You can use SET EXPLAIN output to study the query plans of an application. The output of the SET EXPLAIN statement shows decisions that the query optimizer makes. It shows whether parallel scans are used, the maximum number of threads required to answer the query, and the type of join used for the query.

You can restructure a query or use OPTCOMPIND to change how the optimizer treats the query.

Influencing the choice of a query plan

The **OPTCOMPIND** environment variable and the OPTCOMPIND configuration parameter indicate the preferred join plan, thus assisting the optimizer in selecting the appropriate join method for parallel database queries. To influence the optimizer in its choice of a join plan, you can set the OPTCOMPIND configuration parameter.

The value that you assign to the OPTCOMPIND configuration parameter is referenced only when applications do not set the **OPTCOMPIND** environment variable.

Set OPTCOMPIND to `0` if you want the database server to select a join plan exactly as it did in versions of the database server prior to version 6.0. This option ensures compatibility with previous versions of the database server.

An application with an isolation mode of Repeatable Read can lock all records in a table when it performs a hash join. For this reason, you should set OPTCOMPIND to `1`.

If you want the optimizer to make the determination for you based on cost, regardless of the isolation level of applications, set OPTCOMPIND to `2`.

You can use the SET ENVIRONMENT OPTCOMPIND command to change the value of OPTCOMPIND within a session. For more information about using this command, see Setting the value of OPTCOMPIND within a session on page 44.

For more information about OPTCOMPIND and the different join plans, see The query plan on page 291.

Setting the PDQ priority dynamically

You can use the SET PDQPRIORITY statement to set PDQ priority dynamically within an application. The PDQ priority value can be any integer from -1 through 100.

The PDQ priority set with the SET PDQPRIORITY statement supersedes the **PDQPRIORITY** environment variable.

The DEFAULT tag for the SET PDQPRIORITY statement allows an application to revert to the value for PDQ priority as set by the environment variable, if any. For more information about the SET PDQPRIORITY statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

Enabling the database server to allocate PDQ memory

You can set the IMPLICIT_PDQ session environment option of the SET ENVIRONMENT statement to enable the database server to calculate the amount of PDQ memory to allocate to queries during the current session. This potentially overrides the current **PDQPRIORITY** setting.

The maximum amount of memory that the database server can allocate, however, is limited by the physical memory available to your system, and by the settings of these parameters:

- The **PDQPRIORITY** environment variable
- The most recent SET PDQPRIORITY statement of SQL
- The MAX_PDQPRIORITY configuration parameter
- The DS_TOTAL_MEMORY configuration parameter
- The BOUND_IMPL_PDQ session environment variable.

The IMPLICIT_PDQ session environment option is available only on systems that support **PDQPRIORITY**.

By default, the IMPLICIT_PDQ session environment variable is set to `OFF`. When IMPLICIT_PDQ is set to OFF, the server does not override the current **PDQPRIORITY** setting.

To enable the database server to calculate memory allocations for queries and to distribute memory among query operators according to their needs, enter the following statement before you issue the query:

```
SET ENVIRONMENT IMPLICIT_PDQ ON;
```

If you instead set the IMPLICIT_PDQ value to an integer in the range from 1 to 100, the database server scales its estimate by the specified value. For example, the following example restricts memory allocation in subsequent queries of the session to half of the current **PDQPRIORITY** setting:

```
SET ENVIRONMENT IMPLICIT_PDQ '50';
```

If you set a low IMPLICIT_PDQ value, the amount of memory assigned to the query is proportionally reduced, which might increase the amount of query-operator overflow.

The IMPLICIT_PDQ functionality for a query requires at least LOW distribution statistics on all tables that the query accesses. If distribution statistics are missing for one or more tables that the query references, the IMPLICIT_PDQ setting has no effect on the resources available for query execution. This restriction also applies to star-join queries, which are not supported in the case of missing statistics.

Limiting PDQ resource allocation by setting BOUND_IMPL_PDQ

If IMPLICIT_PDQ is set to ON or to a numeric value, you can also use the BOUND_IMPL_PDQ session environment option of the SET ENVIRONMENT statement of SQL to specify that the allocated PDQ memory should be no greater than the current explicit PDQPRIORITY value or range. If the IMPLICIT_PDQ session environment setting is OFF, whether explicitly off by default, then the BOUND_IMPL_PDQ setting has no effect.

For example, the following statement forces the database server to use explicit PDQPRIORITY values as guidelines in allocating memory, if the IMPLICIT_PDQ session environment option has already been set:

```
SET ENVIRONMENT BOUND_IMPL_PDQ ON;
```

If the IMPLICIT_PDQ setting is an integer in the range from 1 to 100, the explicit PDQPRIORITY value is scaled by that setting as a percentage during the current session.

When the BOUND_IMPL_PDQ session environment option is set to ON (or to one), you require the database server to use the explicit PDQPRIORITY setting as the upper bound for memory that can be allocated to a query. If you set both IMPLICIT_PDQ and BOUND_IMPL_PDQ, then the explicit PDQPRIORITY value determines the upper limit of memory that can be allocated to a query.

If you include an integer value in the SET ENVIRONMENT statement, you must put quote marks around the value. However, do not put quote marks around the ON and OFF keywords.

Example

The following examples are statements with integer values:

```
SET ENVIRONMENT IMPLICIT_PDQ "50";
SET ENVIRONMENT BOUND_IMPL_PDQ "1";
```

User control of PDQ resources

To indicate the PDQ priority of a query, you can set the **PDQPRIORITY** environment variable or execute the SET PDQPRIORITY statement prior to issuing a query. These PDQ priority options allow you to request a certain amount of parallel-processing resources for the query.

The resources that you request and the amount of resources that the database server allocates for the query can differ. This difference occurs when the database server administrator uses the MAX_PDQPRIORITY configuration parameter to put a ceiling on user-requested resources, as the following topic explains.

DBA control of resources for PDQ and DSS queries

To manage the total amount of resources that the database server allocates to parallel database and decision-support (DSS) queries, the database server administrator can set the environment variable and configuration parameters.

Controlling resources allocated to PDQ

To control resources allocated to PDQ, you can set the **PDQPRIORITY** environment variable. The queries that do not set the **PDQPRIORITY** environment variable before they issue a query do not use PDQ. In addition, to place a ceiling on user-specified PDQ priority levels, you can set the MAX_PDQPRIORITY configuration parameter.

When you set the **PDQPRIORITY** environment variable and MAX_PDQPRIORITY parameter, you exert control over the resources that the database server allocates between OLTP and DSS applications. For example, if OLTP processing is particularly heavy during a certain period of the day, you might want to set MAX_PDQPRIORITY to `0`. This configuration parameter puts a ceiling on the resources requested by users who use the **PDQPRIORITY** environment variable, so PDQ is turned off until you reset MAX_PDQPRIORITY to a nonzero value.

DBA control of resources allocated to decision-support queries

A DBA can control the resources that the database server allocates to decision-support queries by setting the DS_TOTAL_MEMORY, DS_MAX_SCANS, and DS_MAX_QUERIES configuration parameters.

In addition to setting limits for decision-support memory and the number of decision-support queries that can run concurrently, the database server uses these parameters to determine the amount of memory to allocate to individual decision-support queries as users submit them. To do so, the database server first calculates a unit of memory called a *quantum* by dividing DS_TOTAL_MEMORY by DS_MAX_QUERIES. When a user issues a query, the database server allocates a percent of the available quanta equal to the PDQ priority of the query.

You can also limit the number of concurrent decision-support scans that the database server allows by setting the DS_MAX_SCANS configuration parameter.

Previous versions of the database server allowed you to set a PDQ priority configuration parameter in the ONCONFIG file. If your applications depend on a global setting for PDQ priority, you can use one of the following methods:

- *For UNIX™:* Define **PDQPRIORITY** as a shared environment variable in the **informix.rc** file. For more information about the **informix.rc** file, see the *HCL OneDB™ Guide to SQL: Reference*.
- *For Windows™:* Set the **PDQPRIORITY** environment variable for a particular group through a logon profile. For more information about the logon profile, see your operating-system manual.

Monitoring resources used for PDQ and DSS queries

You can monitor the resources (shared memory and threads) that the Memory Grant Manager (MGM) has allocated for PDQ queries and the resources that PDQ and decision-support (DSS) queries currently use.

You monitor PDQ resource use in the following ways:

- Run individual **onstat** utility commands to capture information about specific aspects of a running query.
- Execute a SET EXPLAIN statement before you run a query to write the query plan to an output file.

Related information

What PDQ is on page 346

Monitoring PDQ resources by using the onstat Utility

You can use various onstat utility commands to determine how many threads are active and the shared-memory resources that those threads use.

You can use the onstat -g mgm command to monitor how the Memory Grant Manager (MGM) coordinates memory use and to scan threads.

Related reference

onstat -g mgm command: Print MGM resource information

Monitoring PDQ threads with onstat utility commands

You can obtain information about all of the threads that are running for a decision-support query by running the **onstat -u** and **onstat -g ath** commands.

The **onstat -u** option lists all the threads for a session. If a session is running a decision-support query, the output lists the primary thread and any additional threads. For example, session `10` in Figure 61: onstat -u output on page 362 has a total of five threads running.

Figure 61. onstat -u output

```
Userthreads
address  flags   sessid   user     tty      wait     tout locks nreads   nwrites
80eb8c   ---P--D 0        informix -        0        0    0     33       19
80ef18   ---P--F 0        informix -        0        0    0     0        0
80f2a4   ---P--B 3        informix -        0        0    0     0        0
80f630   ---P--D 0        informix -        0        0    0     0        0
80fd48   ---P--- 45       chrisw   ttyp3    0        0    1     573      237
810460   ------- 10       chrisw   ttyp2    0        0    1     1        0
810b78   ---PR-- 42       chrisw   ttyp3    0        0    1     595      243
810f04   Y------ 10       chrisw   ttyp2    beacf8   0    1     1        0
811290   ---P--- 47       chrisw   ttyp3    0        0    2     585      235
81161c   ---PR-- 46       chrisw   ttyp3    0        0    1     571      239
8119a8   Y------ 10       chrisw   ttyp2    a8a944   0    1     1        0
81244c   ---P--- 43       chrisw   ttyp3    0        0    2     588      230
8127d8   ----R-- 10       chrisw   ttyp2    0        0    1     1        0
812b64   ---P--- 10       chrisw   ttyp2    0        0    1     20       0
812ef0   ---PR-- 44       chrisw   ttyp3    0        0    1     587      227
 15 active, 20 total, 17 maximum concurrent
```

The **onstat -g ath** output also lists these threads and includes a **name** column that indicates the role of the thread. Threads that a primary decision-support thread started have a name that indicates their role in the decision-support query. For example, Figure 62: onstat -g ath Output on page 363 lists four *scan* threads, started by a primary thread (**sqlexec**).

Figure 62. onstat -g ath Output

```
Threads:
tid     tcb     rstcb   prty    status                  vp-class   name
...
11      994060   0        4     sleeping(Forever)         1cpu    kaio
12      994394   80f2a4   2     sleeping(secs: 51)        1cpu    btclean
26      99b11c   80f630   4     ready                     1cpu    onmode_mon
32      a9a294   812b64   2     ready                     1cpu    sqlexec
113     b72a7c   810b78   2     ready                     1cpu    sqlexec
114     b86c8c   81244c   2     cond wait(netnorm)        1cpu    sqlexec
115     b98a7c   812ef0   2     cond wait(netnorm)        1cpu    sqlexec
116     bb4a24   80fd48   2     cond wait(netnorm)        1cpu    sqlexec
117     bc6a24   81161c   2     cond wait(netnorm)        1cpu    sqlexec
118     bd8a24   811290   2     ready                     1cpu    sqlexec
119     beae88   810f04   2     cond wait(await_MC1)      1cpu    scan_1.0
120     a8ab48   8127d8   2     ready                     1cpu    scan_2.0
121     a96850   810460   2     ready                     1cpu    scan_2.1
122     ab6f30   8119a8   2     running                   1cpu    scan_2.2
```

Monitoring resources allocated for a session running a DSS query

You can monitor the resources allocated for, and used by, a session that is running a decision-support (DSS) query by running the **onstat -g ses** command.

The **onstat -g ses** option displays the following information:

- The shared memory allocated for a session that is running a decision-support query
- The shared memory used by a session that is running a decision-support query
- The number of threads that the database server started for a session

For example, in Figure 63: onstat -g ses output on page 363, session number `49` is running five threads for a decision-support query.

Figure 63. onstat -g ses output

```
session                                    #RSAM    total      used
id       user     tty      pid   hostname threads  memory     memory
57       informix -        0     -        0        8192       5908
56       user_3   ttyp3    2318  host_10  1        65536      62404
55       user_3   ttyp3    2316  host_10  1        65536      62416
54       user_3   ttyp3    2320  host_10  1        65536      62416
53       user_3   ttyp3    2317  host_10  1        65536      62416
52       user_3   ttyp3    2319  host_10  1        65536      62416
51       user_3   ttyp3    2321  host_10  1        65536      62416
49       user_1   ttyp2    2308  host_10  5        188416     178936
2        informix -        0     -        0        8192       6780
1        informix -        0     -        0        8192       4796
```

Identifying parallel scans in SET EXPLAIN output

When PDQ is turned on, the SET EXPLAIN output shows whether the optimizer chose parallel scans. If the optimizer chose parallel scans, the output lists `Parallel`. (If PDQ is turned off, the output lists `Serial`.)

If PDQ is turned on, the optimizer also indicates the maximum number of threads that are required to answer the query. The `# of Secondary Threads` field in the SET EXPLAIN output indicates the number of threads that are required in addition to your user session thread. The total number of threads necessary is the number of secondary threads plus 1.

The following example shows SET EXPLAIN output for a table with fragmentation and PDQ priority set to `LOW`:

```
SELECT * FROM t1 WHERE c1 > 20

Estimated Cost: 2
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Parallel, fragments: 2)

 Filters: informix.t1.c1 > 20

# of Secondary Threads = 1
```

The following example of partial SET EXPLAIN output shows a query with a hash join between two fragmented tables and PDQ priority set to `ON`. The query is marked with `DYNAMIC HASH JOIN`, and the table on which the hash is built is marked with `Build Outer`.

```
QUERY:
------
SELECT h1.c1, h2.c1 FROM h1, h2 WHERE h1.c1 = h2.c1

Estimated Cost: 2
Estimated # of Rows Returned: 5

1) informix.h1: SEQUENTIAL SCAN (Parallel, fragments: ALL)

2) informix.h2: SEQUENTIAL SCAN (Parallel, fragments: ALL)


DYNAMIC HASH JOIN (Build Outer)
 Dynamic Hash Filters: informix.h1.c1 = informix.h2.c1

# of Secondary Threads = 6
```

The following example of partial SET EXPLAIN output shows a table with fragmentation, PDQ priority set to `LOW`, and an index that was selected as the access plan:

```
SELECT * FROM t1 WHERE c1 < 13

Estimated Cost: 2
Estimated # of Rows Returned: 1

1) informix.t1: INDEX PATH

 (1) Index Keys: c1 (Parallel, fragments: ALL)
```

```
 Upper Index Filter: informix.t1.c1 < 13


# of Secondary Threads = 3
```

Improving individual query performance

You can test, monitor, and improve queries.

Related information

Tune the new version for performance and adjust queries on page

Test queries using a dedicated test system

You can test a query on a system that does not interfere with production database servers. However, you must be careful, because testing queries on a separate system might distort your tuning decisions.

Even if you use your database server as a data warehouse, you might sometimes test queries on a separate system until you understand the tuning issues that are relevant to the query.

If you are trying to improve performance of a large query, one that might take several minutes or hours to complete, you can prepare a scaled-down database in which your tests can complete more quickly. However, be aware of these potential problems:

- The optimizer can make different choices in a small database than in a large one, even when the relative sizes of tables are the same. Verify that the query plan is the same in the real and the model databases.
- Execution time is rarely a linear function of table size. For example, sorting time increases faster than table size, as does the cost of indexed access when an index goes from two to three levels. What appears to be a big improvement in the scaled-down environment can be insignificant when applied to the full database.

Therefore, any conclusion that you reach as a result of tests in the model database must be tentative until you verify them in the production database.

You can often improve performance by adjusting your query or data model with the following goals in mind:

- If you are using a multiuser system or a network, where system load varies widely from hour to hour, try to perform your experiments at the same time each day to obtain repeatable results. Start tests when the system load is consistently light so that you are truly measuring the impact of your query only.
- If the query is embedded in a complicated program, you can extract the SELECT statement and embed it in a DB-Access script.

Related information

Tune the new version for performance and adjust queries on page

Display the query plan

Before you change a query, display its query plan to determine the kind and amount of resources that the query requires. The query plan shows what parallel scans are used, the maximum number of threads required, and the indexes used.

After you study the query plan, examine your data model to see if the changes this chapter suggests will improve the query.

You can display the query plan with one of the following methods:

- Execute one of the following SET EXPLAIN statements just before the query:
  - SET EXPLAIN ON

    This SQL statement displays the chosen query plan. For a description of the SET EXPLAIN ON output, see Report that shows the query plan chosen by the optimizer on page 299.
  - SET EXPLAIN ON AVOID_EXECUTE

    This SQL statement displays the chosen query plan and does not execute the query.
- Use one of the following EXPLAIN directives in the query:
  - EXPLAIN
  - EXPLAIN AVOID_EXECUTE

For more information about these EXPLAIN directives, see EXPLAIN directives on page 336.

Improve filter selectivity

You can control the amount of information that a query evaluates. The greater the precision with which you specify the desired rows, the greater the likelihood that your queries will complete quickly.

To control the amount of information that the query evaluates, use the WHERE clause of the SELECT statement. The conditional expression in the WHERE clause is commonly called a *filter*.

For information about how filter selectivity affects the query plan that the optimizer chooses, see Filters in the query on page 312. The following sections provide some guidelines to improve filter selectivity.

Filters with user-defined routines

You can improve the selectivity of query filters that include user-defined routines (UDRs).

You can improve the selectivity if the UDRs have the following features:

- Functional indexes

  You can create a *functional index* on the resulting values of a user-defined routine or a built-in function that operates on one or more columns. When you create a functional index, the database server computes the return values of the function and stores them in the index. The database server can locate the return value of the function in an appropriate index without executing the function for each qualifying column.

For more information about indexing user-defined functions, see Using a functional index on page 233.

- User-defined selectivity functions

  You can write a function that calculates the expected fraction of rows that qualify for the function. For a brief description of user-defined selectivity functions, see Selectivity and cost functions on page 419. For more information about how to write and register user-defined selectivity functions, see *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

Avoid some filters

For best performance, avoid filters for certain difficult regular expressions and filters for noninitial strings.

Avoid difficult regular expressions

The MATCHES and LIKE keywords support *wildcard* matches, which are technically known as *regular expressions*. Some regular expressions are more difficult than others for the database server to process.

A wildcard in the initial position, as in the following example (find customers whose first names do not end in *y*), forces the database server to examine every value in the column:

```
SELECT * FROM customer WHERE fname NOT LIKE '%y'
```

You cannot use an index with such a filter, so the table in this example must be accessed sequentially.

If a difficult test for a regular expression is essential, avoid combining it with a join. If necessary, process the single table and apply the test for a regular expression to select the desired rows. Save the result in a temporary table and join that table to the others.

Regular-expression tests with wildcards in the middle or at the end of the operand do not prevent the use of an index when one exists.

Avoid noninitial substrings

For best performance, avoid filters for noninitial strings. A filter based on a noninitial substring of a column requires the database server to test every value in the column.

For example, in the following code, a noninitial substring requires the database server to test every value in the column:

```
SELECT * FROM customer
   WHERE zipcode[4,5] > '50'
```

The database server cannot use an index to evaluate such a filter.

The optimizer uses an index to process a filter that tests an initial substring of an indexed column. However, the presence of the substring test can interfere with the use of a composite index to test both the substring column and another column.

Use join filters and post-join filters

The database server provides support for using a subset of the ANSI join syntax.

This syntax that includes the following keywords:

- ON keyword to specify the join condition and any optional join filters
- LEFT OUTER JOIN keywords to specify which table is the dominant table (also referred to as *outer table*)

For more information about this ANSI join syntax, see the *HCL OneDB™ Guide to SQL: Syntax*.

In an ANSI outer join, the database server takes the following actions to process the filters:

- Applies the join condition in the ON clause to determine which rows of the subordinate table (also referred to as *inner table*) to join to the outer table
- Applies optional join filters in the ON clause before and during the join

  If you specify a join filter on a base inner table in the ON clause, the database server can apply it prior to the join, during the scan of the data from the inner table. Filters on a base subordinate table in the ON clause can provide the following additional performance benefits:

  - Fewer rows to scan from the inner table prior to the join
  - Use of index to retrieve rows from the inner table prior to the join
  - Fewer rows to join
  - Fewer rows to evaluate for filters in the WHERE clause

  For information about what occurs when you specify a join filter on an outer table in the ON clause, see the *HCL OneDB™ Guide to SQL: Syntax*.
- Applies filters in the WHERE clause after the join

  Filters in the WHERE clause can reduce the number of rows that the database server needs to scan and reduce the number of rows returned to the user.

  The term *post-join filters* refers to these WHERE clause filters.

When distributed queries that use ANSI-compliant LEFT OUTER syntax for specifying joined tables and nested loop joins are executed, the query is sent to each participating database server for operations on local tables of those servers.

For example, the demonstration database has the **customer** table and the **cust_calls** table, which tracks customer calls to the service department. Suppose a certain call code had many occurrences in the past, and you want to see if calls of this kind have decreased. To see if customers no longer have this call code, use an outer join to list all customers.

Figure 64: SET EXPLAIN ON output for an ANSI join on page 369 shows a sample SQL statement to accomplish this ANSI join query and the SET EXPLAIN ON output for it.

Figure 64. SET EXPLAIN ON output for an ANSI join

```
QUERY:
------
SELECT c.customer_num, c.lname, c.company,
c.phone, u.call_dtime, u.call_code, u.call_descr
FROM customer c
LEFT JOIN cust_calls u ON c.customer_num = u.customer_num
ORDER BY u.call_dtime

Estimated Cost: 14
Estimated # of Rows Returned: 29
Temporary Files Required For: Order By

1) virginia.c: SEQUENTIAL SCAN

2) virginia.u: INDEX PATH

   (1) Index Keys: customer_num call_dtime (Serial, fragments: ALL)
     Lower Index Filter: virginia.c.customer_num = virginia.u.customer_num

  ON-Filters:virginia.c.customer_num = virginia.u.customer_num
  NESTED LOOP JOIN(LEFT OUTER JOIN)
```

Look at the following lines in the SET EXPLAIN ON output in Figure 64: SET EXPLAIN ON output for an ANSI join on page 369:

- The `ON-Filters:` line lists the join condition that was specified in the ON clause.
- The last line of the SET EXPLAIN ON output shows all three keywords `(LEFT OUTER JOIN)` for the ANSI join even though this query specifies only the LEFT JOIN keywords in the FROM clause. The OUTER keyword is optional.

Figure 65: SET EXPLAIN ON output for a join filter in an ANSI join on page 370 shows the SET EXPLAIN ON output for an ANSI join with a join filter that checks for calls with the `I` call_code.

Figure 65. SET EXPLAIN ON output for a join filter in an ANSI join

```
QUERY:
------
SELECT c.customer_num, c.lname, c.company,
c.phone, u.call_dtime, u.call_code, u.call_descr
FROM customer c LEFT JOIN cust_calls u
ON c.customer_num = u.customer_num
AND u.call_code = 'I'
ORDER BY u.call_dtime

Estimated Cost: 13
Estimated # of Rows Returned: 25
Temporary Files Required For: Order By

  1) virginia.c: SEQUENTIAL SCAN

  2) virginia.u: INDEX PATH

     Filters: virginia.u.call_code = 'I'

      (1) Index Keys: customer_num call_dtime   (Serial, fragments: ALL)
          Lower Index Filter: virginia.c.customer_num = virginia.u.customer_num

ON-Filters:(virginia.c.customer_num = virginia.u.customer_num
               AND virginia.u.call_code = 'I' )
NESTED LOOP JOIN(LEFT OUTER JOIN)
```

The main differences between the output in Figure 64: SET EXPLAIN ON output for an ANSI join on page 369 and Figure 65: SET EXPLAIN ON output for a join filter in an ANSI join on page 370 are as follows:

- The optimizer chooses a different index to scan the inner table.

  This new index exploits more filters and retrieves a smaller number of rows. Consequently, the join operates on fewer rows.
- The ON clause join filter contains an additional filter.

The value in the `Estimated # of Rows Returned` line is only an estimate and does not always reflect the actual number of rows returned. The sample query in Figure 65: SET EXPLAIN ON output for a join filter in an ANSI join on page 370 returns fewer rows than the query in Figure 64: SET EXPLAIN ON output for an ANSI join on page 369 because of the additional filter.

Figure 66: SET EXPLAIN ON output for the WHERE clause filter in an ANSI join on page 371 shows the SET EXPLAIN ON output for an ANSI join query that has a filter in the WHERE clause.

Figure 66. SET EXPLAIN ON output for the WHERE clause filter in an ANSI join

```
QUERY:
------
SELECT c.customer_num, c.lname, c.company,
   c.phone, u.call_dtime, u.call_code, u.call_descr
FROM customer c LEFT JOIN cust_calls u
ON c.customer_num = u.customer_num
   AND u.call_code = 'I'
WHERE c.zipcode = "94040"
ORDER BY u.call_dtime

Estimated Cost: 3
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

  1) virginia.c: INDEX PATH

    (1) Index Keys: zipcode   (Serial, fragments: ALL)
        Lower Index Filter: virginia.c.zipcode = '94040'

  2) virginia.u: INDEX PATH

     Filters: virginia.u.call_code = 'I'

    (1) Index Keys: customer_num call_dtime   (Serial, fragments: ALL)
        Lower Index Filter: virginia.c.customer_num = virginia.u.customer_num

ON-Filters:(virginia.c.customer_num = virginia.u.customer_num
      AND virginia.u.call_code = 'I' )
NESTED LOOP JOIN(LEFT OUTER JOIN)

PostJoin-Filters:virginia.c.zipcode = '94040'
```

The main differences between the output in Figure 65: SET EXPLAIN ON output for a join filter in an ANSI join on page 370 and Figure 66: SET EXPLAIN ON output for the WHERE clause filter in an ANSI join on page 371 are as follows:

- The index on the **zipcode** column in the post-join filter is chosen for the dominant table.
- The `PostJoin-Filters` line shows the filter in the WHERE clause.

Automatic statistics updating

The database server updates statistics automatically according to a predefined schedule and a set of expiration policies. The Auto Update Statistics (AUS) maintenance system identifies tables and indexes that require new optimizer statistics and runs the appropriate UPDATE STATISTICS statements to optimize query performance.

The AUS maintenance system updates the statistics for tables that are in logged databases, regardless of the database locale. By making current table statistics available to the query optimizer, the AUS maintenance system can reduce the risk of performance degradation from inefficient query plans.

Depending on your system, you might need to adjust the AUS expiration policies or schedule. The AUS maintenance system resides in the **sysadmin** database.

Related information

Update statistics when they are not generated automatically on page 378

How AUS works

The Auto Update Statistics (AUS) maintenance system uses a combination of Scheduler sensors, tasks, thresholds, and tables to evaluate and update statistics.

The Scheduler tasks, sensors, thresholds, and tables reside in the **sysadmin** database. By default, only user **informix** is granted access to the **sysadmin** database.

The following sequence of events describes how statistics are automatically updated:

1. The **mon_table_profile** sensor of the Scheduler runs every day to read data from the **systables** table in the **sysmaster** database. The sensor updates the **mon_table_profile** table in the **sysadmin** database with information about how much each table has changed.
2. The Auto Update Statistics Evaluation task gathers information every day from the **mon_table_profile** table and the **systable**, **sysdistrib**, **syscolumns**, and **sysindices** tables in the **sysmaster** database.
3. The Auto Update Statistics Evaluation task determines which tables need updates based on the expiration policies.
4. The Auto Update Statistics Evaluation task generates UPDATE STATISTICS statements and inserts them into the **aus_command** table in the **sysadmin** database.
5. The Auto Update Statistics Refresh task runs the UPDATE STATISTICS statements from the **aus_command** table on Saturday and Sunday mornings between 1:00 AM and 5:00 AM and inserts the results back into the **aus_command** table. Any UPDATE STATISTICS statements that did not complete before 5:00 AM remain in the **aus_command** table.

The following table describes the tasks, sensors, thresholds, tables, and views in the **sysadmin** database that comprise the AUS maintenance system.

Table 17. AUS components

Component	Type	Description
mon_table_profile	sensor	Compiles table profile information, including the total number of updates, inserts, and deletes that occurred on each table. Defined in the **ph_task** table.
mon_table_profile	table	Stores table profile information gathered by its sensor. Many other Scheduler tasks use information from this table.
Auto Update Statistics Evaluation	task	Identifies tables with stale statistics, based on expiration policies, and generates UPDATE STATISTICS statements for those tables. Defined in the **ph_task** table.

Table 17. AUS components (continued)

Component	Type	Description
aus_command	table	Stores a list of prioritized UPDATE STATISTICS statements that are scheduled to be run, and the results of those statements after they are run. The **aus_cmd_state** column indicates the status of each UPDATE STATISTICS statement: • P = Pending • I = In progress • E = Error • C = Complete without errors If the command status is E, the associated SQL error code is listed in the **aus_cmd_err_sql** column and the associated ISAM error code is listed in the **aus_cmd_err_isam** column. The **aus_cmd_runtime** shows the time that is elapsed for the update statistics command to complete. The **aus_cmd_time** shows the start time for the update statistics command.
Auto Update Statistics Refresh	task	Runs the prepared UPDATE STATISTICS statements on Saturdays and Sundays between 1:00 AM and 5:00 AM. Defined in the **ph_task** table.
expiration policies	thresholds	Define the criteria for when to update statistics. Defined in the **ph_threshold** table.
aus_cmd_comp	view	Shows information from the **aus_command** table about UPDATE STATISTICS statements that were run successfully.
aus_cmd_list	view	Shows information from the **aus_command** table about UPDATE STATISTICS statements that are scheduled to be run.

For information about other features of the Scheduler, see its description in the *HCL OneDB™ Administrator's Guide*. For information about the **sysadmin** database, see the *HCL OneDB™ Administrator's Reference*.

AUS expiration policies

The Auto Update Statistics (AUS) maintenance system uses expiration policies as criteria for identifying user tables that have changed to the extent that their statistics need to be recalculated.

Internally, the AUS maintenance system automatically skips any tables or fragments that have current statistics and prioritizes tables or fragments that have more changes. Therefore, all tables are scheduled for updating statistics. For more information, see Automatic management of data distribution statistics on page .

The **ph_threshold** table of the **sysadmin** database stores the following configurable thresholds for defining AUS expiration policies.

Table 18. AUS expiration policy thresholds

Threshold Name	Default Value	Description
AUS_AGE	30 (days)	A time-based expiration policy. Statistics or distributions are updated for a table after this amount of time regardless of how much data has changed.
AUS_AUTO_RULES	1 (enabled)	If enabled, statistics are updated using the higher of the following default minimum guidelines or user-created distribution options: • All tables are updated in LOW mode. • All the leading index keys are updated in HIGH mode. • All non-leading index keys are updated in MEDIUM mode. • The minimum resolution for MEDIUM mode is 2.0. • The minimum confidence for MEDIUM mode is 0.95. • The minimum resolution for HIGH mode is 0.5. If the UPDATE STATISTICS statement was run manually for a table, the UPDATE STATISTICS statements generated by the AUS maintenance system do not reduce the level, resolution, confidence, or sampling size options. If disabled by being set to 0, the AUS maintenance system checks which columns have existing distributions and generates update statistics statements to refresh them.
AUS_CHANGE	10 (percent)	A modification-based expiration policy. Statistics or distributions are updated for a table after this percentage of data is changed.

Table 18. AUS expiration policy thresholds (continued)

Threshold Name	Default Value	Description
AUS_PDQ	10 (priority)	The PDQ priority for UPDATE STATISTICS statements run by the AUS maintenance system. By default, all fragments for each table are analyzed in parallel. For more information about PDQ priority, see Update statistics in parallel on very large databases on page 384.
AUS_SMALL_TABLES	100 (rows)	Statistics or distributions are updated every time for a table that has fewer than this number of rows.

Changing AUS expiration policies

You can change AUS expiration policies to customize how often statistics are updated based on how old the statistics are, how much data has changed, or how large the table is.

Before you begin

You must be connected to the **sysadmin** database as user **informix** or another authorized user.

About this task

To change the value of an expiration policy, update the **value** column in the **ph_threshold** table in the **sysadmin** database.

Example

For example, if you find that queries against small tables with 1000 rows or fewer run faster if their statistics are updated more frequently, you can change the expiration policy to ensure that their statistics are updated every week. The following example changes the value of the AUS_SMALL_TABLES threshold to 1000:

```
UPDATE ph_threshold
SET value = 1000
WHERE name = "AUS_SMALL_TABLES";
```

The new threshold takes effect the next time the Auto Update Statistics Evaluator task runs.

Viewing AUS statements

You can view the UPDATE STATISTICS statements generated by the AUS maintenance system in the **aus_cmd_list** view before they are run and in the **aus_cmd_comp** view after they are run successfully. Both tables are in the **sysadmin** database.

Before you begin

You must be connected to the **sysadmin** database as user **informix** or another authorized user.

About this task

To view all scheduled UPDATE STATISTICS statements, run this statement:

```
SELECT * FROM aus_cmd_list;
```

To see all UPDATE STATISTICS statements that were run successfully in the previous 30 days, run this statement:

```
SELECT * FROM aus_cmd_comp;
```

To view all UPDATE STATISTICS statements that failed, run this statement:

```
SELECT aus_cmd_exe, aus_cmd_err_sql, aus_cmd_err_isam
FROM aus_command
WHERE aus_cmd_state = "E";
```

Prioritizing databases in AUS

You can assign a priority to each of your databases in the AUS maintenance system.

About this task

By default all databases have a medium priority. You can assign specific databases a high or a low priority to ensure that statistics for your most important databases are updated first. Statistics for low priority databases are updated after high and medium priority databases, if time and resources permit. For example, if you have a system with a production and a test database, you can assign the production database a high priority and the test database a low priority. You can also disable AUS for a database.

You must be connected to the **sysadmin** database as user **informix** or another authorized user.

To assign a priority to a database in AUS, add a row to the **ph_threshold** table in the **sysadmin** database:

Choose from:

- High priority: Add a row with the **name** column set to AUS_DATABASE_HIGH and the **value** column set to the name of the database.
- Low priority: Add a row with the **name** column set to AUS_DATABASE_LOW and the **value** column set to the name of the database.
- Disable: Add a row with the **name** column set to AUS_DATABASE_DISABLE and the **value** column set to the name of the database.

If you assign more than one priority to a database, the higher priority takes precedence.

Example

Example

The following statement sets the priority for the database that is named **my_database** to high:

```
INSERT INTO ph_threshold(id, name, task_name, value, value_type, description)
     VALUES(0,
            "AUS_DATABASE_HIGH",
            "Auto Update Statistics Evaluation",
            "my_database",
            "STRING",
            "Rank this database as high priority to get its tables done first");
```

Rescheduling AUS

You can change when and for how long the Auto Update Statistics Refresh task runs.

Before you begin

Updating statistics is resource-intensive. Therefore, by default, statistics are automatically updated on Saturdays and Sundays between 1:00 AM and 5:00 AM. If you find that not all pending UPDATE STATISTICS statements can be run in this time period, or you want statistics to be refreshed more often, you can change the start time, the end time, and the days of the week to perform this task.

You must be connected to the **sysadmin** database as user **informix** or another authorized user.

About this task

To change the schedule of the Auto Update Statistics Refresh task, update the **ph_task** table where the value of the **tk_name** column is Auto Update Statistics Refresh.

Example

The following example changes the ending time of the task to 6:00 AM:

```
UPDATE ph_task
SET tk_stop_time = "06:00:00"
WHERE tk_name = "Auto Update Statistics Refresh";
```

The following example changes the days that the task is run to every day of the week (Saturday and Sunday are enabled by default):

```
UPDATE ph_task
SET tk_monday = "T",
tk_tuesday = "T",
tk_wednesday = "T",
tk_thursday = "T",
tk_friday = "T"
WHERE tk_name = "Auto Update Statistics Refresh";
```

Disabling AUS

You can prevent statistics from being updated automatically by disabling the AUS maintenance system.

Before you begin

You must be connected to the **sysadmin** database as user **informix** or another authorized user.

About this task

To disable AUS, you must disable both the Auto Update Statistics Evaluation task and the Auto Update Statistics Refresh task:

1. Update the value of the **tk_enable** column of the **ph_task** table to F where the value of the **tk_name** column is Auto Update Statistics Evaluation.
2. Update the value of the **tk_enable** column of the **ph_task** table to F where the value of the **tk_name** column is Auto Update Statistics Refresh.

Example

The following example disables both tasks:

```
UPDATE ph_task
SET tk_enable = "F"
WHERE tk_name = "Auto Update Statistics Evaluation";

UPDATE ph_task
SET tk_enable = "F"
WHERE tk_name = "Auto Update Statistics Refresh";
```

Update statistics when they are not generated automatically

The UPDATE STATISTICS statement updates the statistics in the system catalog tables that the optimizer uses to determine the lowest-cost query plan.

Important: You do not need to run UPDATE STATISTICS operations when the statistics are generated automatically.

The following statistics are generated automatically by the CREATE INDEX statement, with or without the ONLINE keyword:

- Index-level statistics, equivalent to the statistics gathered in the UPDATE STATISTICS operation in LOW mode, for B-tree indexes.
- Column-distribution statistics, equivalent to the distribution generated in the UPDATE STATISTICS operation in HIGH mode, for a non-opaque leading indexed column of an ordinary B-tree index.

To ensure that the optimizer selects a query plan that best reflects the current state of your tables, run UPDATE STATISTICS at regular intervals when the statistics are not generated automatically.

Tip: If you run UPDATE STATISTICS LOW on the **sysutils** database before you use ON-Bar, the time ON-BAR needs for processing is reduced.

The following table summarizes when to run different UPDATE STATISTICS statements if the statistics are not generated automatically. If you need to run UPDATE STATISTICS statements and you have many tables, you can write a script to generate these UPDATE STATISTICS statements.

When to Execute	UPDATE STATISTICS Statement	Reference for Details and Examples
Number of rows has changed significantly	UPDATE STATISTICS LOW DROP DISTRIBUTIONS	Update the statistics for the number of rows on page 379 or Drop data distributions if necessary when upgrading on page 380
For all columns that are not the leading column of any index	UPDATE STATISTICS LOW	Creating data distributions on page 380

When to Execute	UPDATE STATISTICS Statement	Reference for Details and Examples
Queries have non-indexed join columns or filter columns	UPDATE STATISTICS MEDIUM DISTRIBUTIONS ONLY	Creating data distributions on page 380
Queries have an indexed join columns or filter columns	UPDATE STATISTICS HIGH table (leading column in index)	Creating data distributions on page 380
Queries have a multicolumn indexed defined on join columns or filter columns	UPDATE STATISTICS HIGH table (first differing column in multicolumn index)	Creating data distributions on page 380
Queries have a multicolumn indexed defined on join columns or filter columns	UPDATE STATISTICS LOW table (all columns in multicolumn index)	Creating data distributions on page 380
Queries have many small tables (fit into one extent)	UPDATE STATISTICS HIGH on small tables	Creating data distributions on page 380
Queries use SPL routines	UPDATE STATISTICS FOR PROCEDURE	Reoptimizing SPL routines on page 324

For information about the specific statistics that the database server keeps in the system catalog tables, see Statistics held for the table and index on page 311.

Related information

Automatic statistics updating on page 371

UPDATE STATISTICS statement on page

Update the statistics for the number of rows

When you run UPDATE STATISTICS LOW, the database server updates the statistics in the table, row, and page counts in the system catalog tables. You should run UPDATE STATISTICS LOW as often as necessary to ensure that the statistic for the number of rows is as current as possible.

If the cardinality of a table changes often, run the statement more often for that table.

LOW is the default mode for UPDATE STATISTICS.

The following sample SQL statement updates the statistics in the **systables**, **syscolumns**, and **sysindexes** system catalog tables but does not update the data distributions:

```
UPDATE STATISTICS FOR TABLE tab1;
```

Drop data distributions if necessary when upgrading

When you upgrade to a new version of the database server, you might need to drop distributions to remove the old distribution structure in the **sysdistrib** system catalog table.

To drop the old distribution structure in the **sysdistrib** system catalog table, run this statement:

```
UPDATE STATISTICS DROP DISTRIBUTIONS;
```

Drop distributions in LOW mode without gathering statistics

You can remove distribution information from the **sysdistrib** table and update the **systables.version** column in the system catalog for those tables whose distributions were dropped, without gathering any LOW mode table and index statistics.

You do this using the DROP DISTRIBUTIONS ONLY option in the UPDATE STATISTICS statement. Using the DROP DISTRIBUTIONS ONLY option can result in faster performance because the database server does not gather the table and index statistics that the LOW mode option generates when the ONLY keyword does not follow the DROP DISTRIBUTIONS keywords.

For detailed information about how to use the DROP DISTRIBUTIONS ONLY option, see the *HCL OneDB™ Guide to SQL: Syntax*.

Creating data distributions

You can generate statistics for a table and you can build data distributions for each table that your query accesses.

About this task

(You do not need to run UPDATE STATISTICS operations when the statistics are generated automatically.)

The database server creates data distributions, which provide information to the optimizer, any time the UPDATE STATISTICS MEDIUM or UPDATE STATISTICS HIGH command is executed.

Important:

The database server creates data distributions by sampling a column's data, sorting the data, building distributions bins, and inserting the results into the **sysdistrib** system catalog table.

You can control the sample size for the scan through the keyword HIGH or MEDIUM. The difference between UPDATE STATISTICS HIGH and UPDATE STATISTICS MEDIUM is the number of rows sampled. UPDATE STATISTICS HIGH scans the entire table, while UPDATE STATISTICS MEDIUM samples only a subset of rows, based on the confidence and resolution used by the UPDATE STATISTICS statement.

You can use the LOW keyword with the UPDATE STATISTICS statement only for fully qualified index keys.

If a distribution has been generated for a column, the optimizer uses that information to estimate the number of rows that match a query against a column. Data distributions in **sysdistrib** supersede values in the **colmin** and **colmax** column of the **syscolumns** system catalog table when the optimizer estimates the number of rows returned.

When you use data-distribution statistics for the first time, try to update statistics in MEDIUM mode for all your tables and then update statistics in HIGH mode for all columns that head indexes. This strategy produces statistical query estimates for the columns that you specify. These estimates, on average, have a margin of error less than *percent* of the total number of rows in the table, where *percent* is the value that you specify in the RESOLUTION clause in the MEDIUM mode. The default percent value for MEDIUM mode is 2.5 percent. (For columns with HIGH mode distributions, the default resolution is 0.5 percent.)

With the DISTRIBUTIONS ONLY option, you can execute UPDATE STATISTICS MEDIUM at the table level or for the entire system because the overhead of the extra columns is not large.

The database server uses the storage locations that the DBSPACETEMP environment variable specifies only when you use the HIGH option of UPDATE STATISTICS.

You can prevent UPDATE STATISTICS operations from using indexes when sorting rows by setting the third parameter of the DBUPSPACE environment variable to a value of `1`.

For each table that your query accesses, build data distributions according to the following guidelines. Also see the examples below the guidelines.

To generate statistics on a table:

1. Identify the set of all columns that appear in any single-column or multi-column index on the table.
2. Identify the subset that includes all columns that are not the leading column of any index.
3. Run UPDATE STATISTICS LOW on each column in that subset.

Results

To build data distributions for each table that your query accesses:

1. Run a single UPDATE STATISTICS MEDIUM for all columns in a table that do not head an index.

   Use the default parameters unless the table is very large, in which case you should use a resolution of `1.0` and confidence of `0.99`.
2. Run the following UPDATE STATISTICS statement to create distributions for non-index join columns and non-index filter columns:

   ```
   UPDATE STATISTICS MEDIUM DISTRIBUTIONS ONLY;
   ```
3. Run UPDATE STATISTICS HIGH for all columns that head an index. For the fastest execution time of the UPDATE STATISTICS statement, you must execute one UPDATE STATISTICS HIGH statement for each column, as shown in the example below this procedure.
4. If you have indexes that begin with the same subset of columns, run UPDATE STATISTICS HIGH for the first column in each index that differs, as shown in the second example below this procedure.
5. For each single-column or multi-column index on the table:

a. Identify the set of all columns that appear in the index.
b. Identify the subset that includes all columns that are not the leading column of any index.
c. Run UPDATE STATISTICS LOW on each column in that subset. (LOW is the default.)

6. For the tables on which indexes were created in Step 3, run an UPDATE STATISTICS statement to update the **sysindexes** and **syscolumns** system catalog tables, as shown in the following example:

```
UPDATE STATISTICS FOR TABLE t1(a,b,e,f);
```

7. For small tables, run UPDATE STATISTICS HIGH, for example:

```
UPDATE STATISTICS HIGH FOR TABLE t2;
```

Because the statement constructs the statistics only once for each index, these steps ensure that UPDATE STATISTICS executes rapidly.

Example

Examples

Example of UPDATE STATISTICS HIGH statements for all columns that head an index

Suppose you have a table **t1** with columns **a**, **b**, **c**, **d**, **e**, and **f** with the following indexes:

```
CREATE INDEX ix_1 ON t1 (a, b, c, d) ...
CREATE INDEX ix_3 ON t1 (f) ...
```

Run the following UPDATE STATISTICS statements for the columns that head an index:

```
UPDATE STATISTICS HIGH FOR TABLE t1(a);
UPDATE STATISTICS HIGH FOR TABLE t1(f);
```

These UPDATE STATISTICS HIGH statements replace the distributions created with the UPDATE STATISTICS MEDIUM statements with high distributions for index columns.

Example of UPDATE STATISTICS HIGH statements for the first column in each index that differs:

For example, suppose you have the following indexes on table **t1:**

```
CREATE INDEX ix_1 ON t1 (a, b, c, d) ...
CREATE INDEX ix_2 ON t1 (a, b, e, f) ...
CREATE INDEX ix_3 ON t1 (f) ...
```

Step 3 on page 381 executes UPDATE STATISTICS HIGH on column **a** and column **f**. Then run UPDATE STATISTICS HIGH on columns **c** and **e**.

```
UPDATE STATISTICS HIGH FOR TABLE t1(c);
UPDATE STATISTICS HIGH FOR TABLE t1(e);
```

In addition, you can run UPDATE STATISTICS HIGH on column **b**, although this is usually not necessary.

Related information

Virtual portion of shared memory on page 62

UPDATE STATISTICS statement on page

Updating statistics for join columns

In some situations, you might want to run the UPDATE STATISTICS statement with the HIGH keyword for specific join columns.

About this task

Because of improvements and adjusted cost estimates to establish better query plans, the optimizer depends greatly on an accurate understanding of the underlying data distributions in certain cases. You might still think that a complex query does not execute quickly enough, even though you followed the guidelines in Creating data distributions on page 380. If your query involves equality predicates, take one of the following actions:

- Run the UPDATE STATISTICS statement with the HIGH keyword for specific join columns that appear in the WHERE clause of the query. If you followed the guidelines in Creating data distributions on page 380, columns that head indexes already have HIGH mode distributions.
- Determine whether HIGH mode distribution information about columns that do not head indexes can provide a better execution path, take the following steps:

To determine if UPDATE STATISTICS HIGH on join columns might make a difference:

1. Issue the SET EXPLAIN ON statement and rerun the query.
2. Note the estimated number of rows in the SET EXPLAIN output and the actual number of rows that the query returns.
3. If these two numbers are significantly different, run UPDATE STATISTICS HIGH on the columns that participate in joins, unless you have already done so.

Results

Important: If your table is very large, UPDATE STATISTICS with the HIGH mode can take a long time to execute.

The following example shows a query that involves join columns:

```
SELECT employee.name, address.city
   FROM employee, address
   WHERE employee.ssn = address.ssn
   AND employee.name = 'James'
```

In this example, the join columns are the **ssn** fields in the **employee** and **address** tables. The data distributions for both of these columns must accurately reflect the actual data so that the optimizer can correctly determine the best join plan and execution order.

You cannot use the UPDATE STATISTICS statement to create data distributions for a table that is external to the current database. For additional information about data distributions and the UPDATE STATISTICS statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

Updating statistics for columns with user-defined data types

Programmers can write functions that gather statistics for columns with user-defined data types. You can store the data distributions for user-defined data types in an sbspace.

About this task

Because information about the nature and use of a user-defined data type (UDT) is not available to the database server, it cannot collect the **colmin** and **colmax** column of the **syscolumns** system catalog table for user-defined data types. To gather statistics for columns with user-defined data types, programmers must write functions that extend the UPDATE STATISTICS statement. For more information, see the performance chapter in *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

Because the data distributions for user-defined data types can be large, you can optionally store them in an sbspace instead of the **sysdistrib** system catalog table.

To store data distributions for user-defined data types in an sbspace:

1. Use the **onspaces -c -S** command to create an sbspace.

   To ensure recoverability of the data distributions, specify `LOGGING=ON` in the **-Df** option, as the following sample shows:

   ```
   % onspaces -c -S distrib_sbsp -p /dev/raw_dev1 -o 500 -s
         20000
      -m /dev/raw_dev2 500 -Ms 150 -Mo 200 -Df
         "AVG_LO_SIZE=32,LOGGING=ON"
   ```

   For information about sizing an sbspace, see Estimating pages that smart large objects occupy on page 166.

   For more information about specifying storage characteristics for sbspaces, see Configuration parameters that affect sbspace I/O on page 127.

2. Specify the sbspace that you created in step 1 in the configuration parameter SYSSBSPACENAME.
3. Specify the column with the user-defined data type when you run the UPDATE STATISTICS statement with the MEDIUM or HIGH keywords to generate data distributions.

Results

To print the data distributions for a column with a user-defined data type, use the **dbschema -hd** option.

Update statistics in parallel on very large databases

If you have an extremely large database and indexes are fragmented, UPDATE STATISTICS LOW can automatically run statements in parallel.

To enable statements to automatically run in parallel, you must run UPDATE STATISTICS LOW with the PDQ priority set to a value that is between `1` and `10`. If the PDQ priority is set to `1`, 10 percent of the index fragments are analyzed at one time for the current table. If the PDQ priority is set to `5`, 50 percent of the index fragments are analyzed at one time for the current table. If the PDQ priority is set to `10`, all fragments are analyzed at one time for the current table. (If the PDQ priority is set to a value that is higher than `10`, HCL OneDB™ operates as if the PDQ priority is set to `10`, analyzing all fragments at one time for the current table.)

If you run UPDATE STATISTICS MEDIUM or HIGH, you can set the PDQ priority to a value that is higher than `10`. Because the UPDATE STATISTICS MEDIUM and HIGH statements perform a large amount of sorting operations, increasing the PDQ priority to a value that is higher than `10` provides additional memory than can improve the speed of the sorting operations.

Adjust the amount of memory and disk space for UPDATE STATISTICS

When you execute the UPDATE STATISTICS statement, the database server uses memory and disk to sort and construct data distributions. You can affect the amount of memory and disk space available for UPDATE STATISTICS operations.

You can affect the amount of memory and disk space available for UPDATE STATISTICS with the following methods:

- PDQ priority

  You can obtain more memory for sorting when you set PDQ priority greater than 0. The default value for PDQ priority is `0`. To set PDQ priority, use either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

  For more information about PDQ priority, see The allocation of resources for parallel database queries on page 352.

- **DBUPSPACE** environment variable

  You can use the **DBUPSPACE** environment variable to specify the amount of system disk space (and the amount of memory for sorting values) that UPDATE STATISTICS MEDIUM or UPDATE STATISTICS HIGH statements can use in each pass to construct column distributions. If you specify too small a value, the database server instead uses enough space to write the largest column to disk.

  For more information about this environment variable, see the *HCL OneDB™ Guide to SQL: Reference*.

Data sampling during update statistics operations

If you have a large b-tree index with more than 100 K leaf pages, you can generate index statistics based on sampling when you run UPDATE STATISTICS statements in LOW mode. The gathering of statistics through sampling can increase the speed of the update statistics operation.

By default, when UPDATE STATISTICS statements run, the database server reads all index leaf pages in sequence to gather statistics such as the number of leaf pages, the number of unique lead key values, and cluster information. For a large index this can take a long time. With sampling, the database server reads a fraction of the index leaf pages (the sample) and then deduces index statistics based on statistics gathered from the sample.

A possible trade-off for less time in gathering statistics is the accuracy of the statistics gathered. If there are significant skews in the data distribution for the lead index key, the sampling approach can result in a large error margin for the statistics gathered, which in turn might affect optimizer decisions in query plan generation.

You cannot control how much data is in the sample.

To enable or disable sampling, use the USTLOW_SAMPLE configuration parameter or the USTLOW_SAMPLE environment option of the SET ENVIRONMENT statement.

Related information

USTLOW_SAMPLE configuration parameter on page

USTLOW_SAMPLE environment option on page

Display data distributions

You can use the **dbschema** utility to display data distributions.

Unless column values change considerably, you do not need to regenerate the data distributions. To verify the accuracy of the distribution, compare **dbschema -hd** output with the results of appropriately constructed SELECT statements.

For example, the following **dbschema** command produces a list of distributions for each column of table **customer** in database **vjp_stores** with the number of values in each bin, and the number of distinct values:

```
dbschema -hd customer -d vjp_stores
```

Figure 67: Displaying Data Distributions with dbschema -hd on page 387 shows the data distributions for the column **zipcode** that this **dbschema -hd** command produces. Because this column heads the **zip_ix** index, UPDATE STATISTICS HIGH was run on it, as the following output line indicates:

```
High Mode, 0.500000 Resolution
```

Figure 67: Displaying Data Distributions with dbschema -hd on page 387 shows 17 bins with one distinct **zipcode** value in each bin.

Figure 67. Displaying Data Distributions with dbschema -hd

```
dbschema -hd customer -d vjp_stores

...
Distribution for virginia.customer.zipcode

Constructed on 09/18/2000

High Mode, 0.500000 Resolution

--- DISTRIBUTION ---

     (          02135 )
  1: (  1,   1, 02135 )
  2: (  1,   1, 08002 )
  3: (  1,   1, 08540 )
  4: (  1,   1, 19898 )
  5: (  1,   1, 32256 )
  6: (  1,   1, 60406 )
  7: (  1,   1, 74006 )
  8: (  1,   1, 80219 )
  9: (  1,   1, 85008 )
 10: (  1,   1, 85016 )
 11: (  1,   1, 94026 )
 12: (  1,   1, 94040 )
 13: (  1,   1, 94085 )
 14: (  1,   1, 94117 )
 15: (  1,   1, 94303 )
 16: (  1,   1, 94304 )
 17: (  1,   1, 94609 )


--- OVERFLOW ---

  1: (  2,      94022 )
  2: (  2,      94025 )
  3: (  2,      94062 )
  4: (  3,      94063 )
  5: (  2,      94086 )
```

The OVERFLOW portion of the output shows the duplicate values that might skew the distribution data, so **dbschema** moves them from the distribution to a separate list. The number of duplicates in this overflow list must be greater than a critical amount that the following formula determines. Figure 67: Displaying Data Distributions with dbschema -hd on page 387 shows a resolution value of `.0050`. Therefore, this formula determines that any value that is duplicated more than one time is listed in the overflow section.

```
Overflow = .25 * resolution * number_rows
      = .25 * .0050 * 28
      = .035
```

For more information about the **dbschema** utility, see the *HCL OneDB™ Migration Guide*.

Improve performance by adding or removing indexes

You can often improve the performance of a query by adding or, in some cases, removing indexes. You can also enable the optimizer to automatically fetch a set of keys from an index buffer.

To improve the performance of a query, consider using some of the methods that the following topics describe.

In addition:

- Consider using the CREATE INDEX ONLINE and DROP INDEX ONLINE statements to create and drop an index in an online environment, when the database and its associated tables are continuously available. These SQL statements enable you to create and drop indexes without having an access lock placed over the table during the duration of the index builds or drops. For more information, see Creating and dropping an index in an online environment on page 223.
- Set the BATCHEDREAD_INDEX configuration parameter to enable the optimizer to automatically fetch a set of keys from an index buffer. This reduces the number of times a buffer is read.

Related information

BATCHEDREAD_INDEX configuration parameter on page

Replace autoindexes with permanent indexes

If the query plan includes an *autoindex* path to a large table, you can generally improve performance by adding an index on that column. If you perform a query regularly, you can save time by creating a permanent index.

If you perform the query occasionally, you can reasonably let the database server build and discard an index.

Use composite indexes

The optimizer can use a composite index (one that covers more than one column) in several ways.

The database server can use an index on columns **a**, **b**, and **c** (in that order) in the following ways:

- To locate a particular row

  The database server can use a composite index when the first filter is an equality filter and subsequent columns have range (<, <=, >, >=) expressions. The following examples of filters use the columns in a composite index:

  ```
  WHERE a=1
  WHERE a>=12 AND a<15
  WHERE a=1 AND b < 5
  WHERE a=1 AND b = 17 AND c >= 40
  ```

  The following examples of filters cannot use that composite index:

  ```
  WHERE b=10
  WHERE c=221
  WHERE a>=12 AND b=15
  ```

- To replace a table scan when all of the desired columns are contained within the index

  A scan that uses the index but does not reference the table is called a *key-only search*.
- To join column **a**, columns **ab**, or columns **abc** to another table
- To implement ORDER BY or GROUP BY on columns **a**, **ab**, or **abc** but not on **b**, **c**, **ac**, or **bc**

Execution is most efficient when you create a composite index with the columns in order from most to least distinct. In other words, the column that returns the highest count of distinct rows when queried with the DISTINCT keyword in the SELECT statement should come first in the composite index.

If your application performs several long queries, each of which contains ORDER BY or GROUP BY clauses, you can sometimes improve performance by adding indexes that produce these orderings without requiring a sort. For example, the following query sorts each column in the ORDER BY clause in a different direction:

```
SELECT * FROM t1 ORDER BY a, b DESC;
```

To avoid using temporary tables to sort column **a** in ascending order and column **b** in descending order, you must create a composite index on (**a**, **b** DESC) or on (**a** DESC, **b**). You need to create only one of these indexes because of the bidirectional-traverse capability of the database server. For more information about bidirectional traverse, see the *HCL OneDB™ Guide to SQL: Syntax*.

On the other hand, it can be less expensive to perform a table scan and sort the results instead of using the composite index when the following criteria are met:

- Your table is well ordered relative to your index.
- The number of rows that the query retrieves represents a large percentage of the available data.

Indexes for data warehouse applications

Many data warehouse databases use a *star schema*, which consists of a *fact* table and a number of *dimensional* tables. Queries that use tables in a star schema or snowflake schema can benefit from the proper index on the fact table.

The fact table is generally large and contains the quantitative or factual information about the subject. A dimensional table describes an attribute in the fact table.

When a dimension needs lower-level information, the dimension is modeled by a hierarchy of tables, called a *snowflake schema*.

Consider the example of a star schema with one fact table named **orders** and four dimensional tables named **customers**, **suppliers**, **products**, and **clerks**. The **orders** table describes the details of each sale order, which includes the customer ID, supplier ID, product ID, and sales clerk ID. Each dimensional table describes an ID in detail. The **orders** table is large, and the four dimensional tables are small.

The following query finds the total direct sales revenue in the Menlo Park region (postal code 94025) for hard drives supplied by the Johnson supplier:

```
SELECT sum(orders.price)
FROM orders, customers, suppliers,product,clerks
```

```
WHERE orders.custid = customers.custid
   AND customers.zipcode = 94025
   AND orders.suppid = suppliers.suppid
   AND suppliers.name = 'Johnson'
   AND orders.prodid = product.prodid
   AND product.type = 'hard drive'
   AND orders.clerkid = clerks.clerkid
   AND clerks.dept = 'Direct Sales'
```

This query uses a typical star join, in which the fact table joins with all dimensional tables on a foreign key. Each dimensional table has a selective table filter.

An optimal plan for the star join is to perform a cartesian product on the four dimensional tables and then join the result with the fact table. The following index on the fact table allows the optimizer to choose the optimal query plan:

```
CREATE INDEX ON orders(custid,suppid,prodid,clerkid)
```

Without this index, the optimizer might choose to first join the fact table with a single dimensional table and then join the result with the remaining dimensional tables. The optimal plan provides better performance.

For more information about star schemas and snowflake schemas, see the *HCL OneDB™ Database Design and Implementation Guide*.

Configure B-tree scanner information to improve transaction processing

You can improve the performance of transaction processing in logged databases by controlling how the B-tree scanner threads remove deletions from indexes.

The B-tree scanner improves transaction processing for logged databases when rows are deleted from a table with indexes. The B-tree scanner automatically determines which index partitions will be cleaned, based on a priority list. B-tree scanner threads remove deleted index entries and rebalance the index nodes. The B-tree scanner automatically determines which index items are to be deleted.

In a logged database, when a delete or an update operation is performed on a row, any corresponding index entry is not immediately deleted. Instead, the corresponding index entry is flagged as deleted until a B-tree scanner thread scans the index and removes the deleted items. An index containing many deleted items can cause a significant performance problem, because index searches need to scan a larger number of items before finding the first valid item.

The default setting for B-tree scanning provides the following type of scanning, depending on your indexes:

- If the table has more than one attached index, the B-tree scanner uses the leaf scan mode. Leaf scan mode is the only type of scanning possible with multiple attached indexes.
- If the table contains a single attached index or if the indexes are detached, the B-tree scanner uses alice (adaptive linear index cleaning) mode. The initial alice scan mode is optimized for small- to medium-sized systems with few or no indexes above 1 GB. However, if the database server detects that the alice mode is inefficient, the alice scan mode setting is adjusted automatically to accommodate larger indexes.

Depending on your application and the order in which the system adds and deletes keys from the index, the structure of an index can become inefficient.

You use the BTSCANNER configuration parameter to specify the following information, which defines the scan mode:

- The number of B-tree scanner threads to start when the database server starts

  The number of B-tree scanner threads is configurable to any positive number. One B-tree scanner thread will always clean an individual index partition, so if you occasionally or consistently have a higher number of index partitions requiring cleaning, you might want to use more than one B-tree scanner thread. At runtime, you can turn off any B-tree scanner activity by issuing an **onmode -C** command. This command stops all B-tree scanner threads.
- The threshold, which is the minimum number of deleted items an index must encounter before an index is placed on the priority list for eligibility for scanning and cleaning by a B-tree scanner thread

  For example, if you increase the threshold beyond 5000, you might be able to avoid frequent B-tree scanner activity on the indexes that receive the most updates and deletes.
- The range size, in kilobytes, that an index or index fragment must exceed before the index is cleaned with range scanning
- An alice mode value
- The level at which B-tree scanner threads compress indexes by merging two partially used index pages

The server treats a forest of trees index the same way it treats a B-tree index. Therefore, in a logged database, you can control how the B-tree scanner threads remove deletions from both forest of trees and B-tree indexes.

The following table summarizes the differences between the scan modes.

Table 19. Scan modes for B-tree scanner threads

Scan Mode	Description	Performance Advantages or Issues	More Information
Leaf scan mode	In this mode, the	This mode is only	Leaf and range scan mode settings on page 407

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	leaf level of an attached index is completely scanned for deleted items.	possible on attached indexes and is the only mode the server can use if more than one attached	

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
		cheded index exists in a partition.	
Alice (adaptive linear index cleaning) scan mode	If the BTSCANNER alice option is enabled, every in	You can greatly improve performance and reduce I/O when us	Alice scan mode values on page 406

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	dexpartition receives a bitmap that tracks where a deleted item was found in the index.	ing the al ice mo de. Gen eral ly, al ice m ode is 64 ti mes m ore effi ci ent t han ra nge sca nn ing	

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	This scan thatoccurs exclusdes all parts of the index where no deleteoperations are fo	and can automatic ally tune itself for unsatisfactory index es, which range scanning can not do.	

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	und. The initial size and granularity of these bitmaps depend on the size of the pa		

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	rtitions they represent and the current system-wide device level. The server periodically ch		

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	e cks e ach bit map for its eff ici e ncy by ch ec k ing the ra tio of pa ges to be cle a ned to		

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	pages read, adjusting scanning if necessary to get better information. This mode		

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	allocates additional resources (memory) to the index that is consuming exc		

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	essl /O.		
Ranges can mode	Range scanning, which is enabled with the `ranges ize` option, is perfor	Not recommended for OneDB Version 11.10 or higher. Alice scanning is exa	Leaf and range scan mode settings on page 407

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	medctly in the s the ra ame nge as bet ra w nge een sca the nni low ng, and but h is igh 64 p ti age mes ad m dre ore ss. effi The cie l nt, eaf u le ses vel the of s the ame in res dex our pa ces, rtit and		

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	ion is only scanned within this range. The server performs light scans, which do not im	has 128 equal ranges. When you set al ice mode scanning, range scanning does not have	

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	mediately use and strain the buffer pool, even though cleaning occurs through	an effect. If you decide to use range scanning for systems with only a lot of large ind	

Table 19. Scan modes for B-tree scanner threads (continued)

Scan Mode	Description	Performance Advantages or Issues	More Information
	the buffer pool.	extents, set the `ranges ize` option to the minimum partition size for range scanning.	

For more information about the BTSCANNER configuration parameter and for more information about how the database server maintains an index tree, see the chapter on configuration parameters and the chapter on disk structure and storage in the *HCL OneDB™ Administrator's Reference*.

Use the **onstat -C** option to monitor the B-tree scanner activities.

Use the **onmode -C** option to change the configuration of B-tree scanners during runtime.

For more information about **onstat -C** and **onmode -C**, see the *HCL OneDB™ Administrator's Reference*.

Alice scan mode values

You enable alice (adaptive linear index cleaning) mode by setting the `alice` option to any value between `1` and `12` (finest initial granularity). For small- to medium-sized systems with few or no indexes above 1 gigabyte, set the `alice` option to `6` or `7`. For systems with large indexes, set alice to a higher mode.

When you set alice mode, the higher the mode, the more memory is used per index partition. However, the memory used is not a huge amount. The advantage is less I/O, as shown in the following table.

Table 20. Alice mode settings

Alice Mode Setting	Memory or Block I/O
0	Turns off alice scanning.
1	Uses exactly 8 bytes of memory (no adjusting).
2	Uses exactly 16 bytes of memory (no adjusting).
3	Each block of pages will need 512 I/O operations for cleaning.
4	Each block of pages will need 256 I/O operations for cleaning.
5	Each block of pages will need 128 I/O operations for cleaning.
6 (default)	Each block of pages will need 64 I/O operations for cleaning.
7	Each block of pages will need 32 I/O operations for cleaning.
8	Each block of pages will need 16 I/O operations for cleaning.
9	Each block of pages will need 8 I/O operations for cleaning.
10	Each block of pages will need 4 I/O operations for cleaning.
11	Each block of pages will need 2 I/O operations for cleaning.
12	Each block of pages will need 1 I/O operations for cleaning.

When you set the alice mode, you need to consider memory usage versus I/O. The lower the alice mode setting, the less memory the index will use. The higher the alice mode setting, the more memory the index will use. `12` is the highest mode value, because it is a direct mapping of a single bit of memory to each instance of I/O.

Suppose you have an online page size of 2 KB and the default B-Tree Scanner I/O size of 256 pages. If you set the alice mode to `6`, each byte of memory can represent 131,072 index pages (256 MB). If you set the mode to `10`, each byte of memory can represent 8,192 index pages (16 MB). Thus, changing the mode setting from `6` to `10` requests 16 times the memory, but requires 16 times less I/O.

If you have an index partition that uses 1 GB, then an alice mode setting of `6` would take 4 bytes of memory, while an alice mode setting of `10` would consume 64 bytes of memory, as shown in this formula:

```
( {mode block size} io per bit  *   8 bits per byte  * 256 page per io )
```

Setting the alice mode to a value between 3 and 12 sets the initial amount of memory that is used for index cleaning. Subsequently, the B-tree scanners automatically adjust the mode based on the efficiency of past cleaning operations.

For example, if after five scans (by default), the I/O efficiency is below 75 percent, the server automatically adjusts to the next alice mode if you set the mode to a value above `2`. For example, if an index is currently operating in alice mode `6`, a B-tree scanner has cleaned the index at least 5 times, and the I/O efficiency is below 75 percent, the server automatically adjusts to mode `7`, the next higher mode. This doubles the memory required, but reduces the I/O by a factor of 2.

The server will re-evaluate the index after five more scans to determine the I/O efficiency again, and will continue to do this until mode `12`. The server stops making adjustments at mode `12`.

The following example sets the alice mode to `6`:

```
BTSCANNER num=2,threshold=10000,alice=6,compression=default
```

Leaf and range scan mode settings

If a table has more than one attached index, the B-tree scanner uses the leaf scan mode. If you want small indexes to be scanned by the leaf scan method, set the `rangesize` option of the BTSCANNER configuration parameter to `100`.

If you decide to enable range scan mode when a single index exists in the partition, set `rangesize` option of the BTSCANNER configuration parameter to the minimum size that a partition must have to be scanned using this mode. Specify the size in kilobytes.

The following example specifies that:

- The server will start two B-tree scanner threads.
- The server will consider cleaning indexes in the hot list (a list of indexes that caused the server to do extra work) when 50000 deleted items are found in the index.
- Indexes with a partition size that is equal to or larger than 100 KB will be cleaned using the range scan mode.
- Indexes with a partition size of less than 100 KB will be cleaned using the leaf scan mode.
- Index compression is set at the medium (default) level

```
BTSCANNER num=2,threshold=50000,rangesize=100,compression=default
```

B-tree scanner index compression levels and transaction processing performance

B-tree scanner threads compress indexes by merging two partially used index pages if the amount of data on those pages is below the level that is specified by the compression option. You can set the compression level to control the amount of I/O required to find and load data.

B-tree scanner threads look for index pages that can be compressed because they are below the specified level. The B-tree scanner can compress index pages with deleted items and pages that do not have deleted items.

By default, a B-tree scanner compresses at the medium level. The following table provides information about the performance benefits and trade-offs if you change the compression level to high or low.

Table 21. B-Tree Scanner Compression Level Benefits and Trade-offs

Compression Level	Performance Benefits and Trade-offs	When to Use
Low	The low compression level is beneficial for an index that is expected to grow quickly, with frequent B-tree node splits. When the compression level is set to low, the B-tree index will not require as many splits as indexes with medium or high compression levels, because more free space remains in the B-tree nodes.	You might want to change the compression level to low if you expect an index to grow quickly with frequent splits.
High	In general, if an index is read-only or 90 percent of it is read-only, the high compression level is beneficial because searching for data will require fewer pages (and less I/O) to traverse. Examples might be indexes that do not have frequent changes or indexes undergoing batch (block) delete operations. Using high level of compression also means a performance trade-off, because it takes more I/O to compress the index more aggressively. Select operations will have less I/O when the compression level is high.	You might want to change the compression level to high under these circumstances: • If an index is read most of the time, and delete and insert operations occur a small percentage of the time. • If tables are loaded and updated in a batch and are kept for a period of time as read-only tables.

If you do not need to change the compression level to high or low, set the compression option of the BTSCANNER configuration parameter to `med` or `default`.

Index Compression and the Index Fill Factor

In addition to the compression option that specifies when to attempt to join two partially used pages, you can use the FILL FACTOR configuration parameter to control when to add new index pages. The index fill factor, which you define with the FILLFACTOR configuration parameter or the FILLFACTOR option of the CREATE INDEX statement, is a percentage of each index page that will be filled during the index build.

Setting the level for B-tree scanner compression of indexes

HCL OneDB™ provides several ways to specify the level at which B-tree scanner threads will compress indexes pages. To optimize space and transaction processing, you can lower the compression level if your indexes grow quickly. You can increase the level if your indexes have few delete and insert operations or if batch updates are performed.

Before you begin

Prerequisites:

- Determine if adjusting the level for index compression will improve performance.
- Get statistics on the number of rows read, deleted, and inserted by running the **onstat -g ppf** command. You can also view information in the **sysptprof** table.
- Analyze the statistics to determine if you want to change the threshold.

For information about compression levels and the circumstances under which you might want to change the level, see B-tree scanner index compression levels and transaction processing performance on page 407.

Specify the compression level for an instance with any of the following options:

- Set the `compression` field of the BTSCANNER configuration parameter to `low`, `med` (medium), `high`, or `default`. (The system default value is `med`.)
- Run the **onmode -C compression *value*** command, where *value* is `low`, `med` (medium), `high`, and `default`. The system default value is `med`.
- Run an SQL administration API function with this command:

  ```
  SET INDEX COMPRESSION, partition number, compression level
  ```

Example

Examples

Set the compression option of the BTSCANNER configuration parameter to `default` as follows:

```
BTSCANNER num=4,threshold=10000,rangesize=-1,alice=6,compression=default
```

Set the compression option of the BTSCANNER configuration parameter to `high` as follows:

```
BTSCANNER num=4,threshold=5000,compression=high
```

Specify the compression level using **onmode -C**, as follows:

```
onmode -C compression high
```

Run either of the following SQL administration API functions to set the compression level for a single fragment of the index that has the partition number `1048960`:

```
EXECUTE FUNCTION TASK("SET INDEX COMPRESSION", 1048960, "DEFAULT");
```

```
EXECUTE FUNCTION ADMIN("SET INDEX COMPRESSION", 1048960, "LOW");
```

Run the following SELECT statement to execute the task function over all index fragments. This command sets the compression level for all fragments of an index named `idx1` in a database named `db1`.

```
SELECT sysadmin:TASK("SET INDEX COMPRESSION", partnum, "MED")
FROM sysmaster:systabnames
WHERE dbsname = 'dbs1' AND tabname = 'idx1';
```

You can also run the following SELECT TASK statement to execute the task function over all index fragments and set the compression level for all fragments.

```
SELECT TASK("SET INDEX COMPRESSION", partn, "MED")
FROM dbs1:systables t, dbs1:sysfragments f
WHERE f.tabid = t.tabid AND f.fragtype = 'I' AND indexname ='idx1';
```

Determine the amount of free space in an index page

You can use the **oncheck -pT** command to determine the amount of free space in each index page.

If your table has relatively low update activity and a large amount of free space exists, you might want to drop and re-create the index with a larger value for FILLFACTOR to make the unused disk space available.

Optimizer estimates of distributed queries

The optimizer assumes that access to a row from a remote database takes longer than access to a row in a local database. The optimizer estimates include the cost of retrieving the row from disk and transmitting it across the network.

For an example of this higher estimated cost, see The query plan of a distributed query on page 410.

Buffer data transfers for a distributed query

HCL OneDB™ uses several factors to determine the size of the buffer that sends and receives data to and from a remote server.

The server uses the following factors to determine the buffer size:

- The row size

  The database server calculates the row size by summing the average move size (if available) or the length (from the **syscolumns** system catalog table) of the columns.

- The setting of the **FET_BUF_SIZE** environment variable on the client

  You might be able to reduce the size and number of data transfers by using the **FET_BUF_SIZE** environment variable to increase the size of the buffer that the database server uses to send and receive rows to and from the remote database server.

  The minimum buffer size is 1024 or 2048 bytes, depending on the row size. If the row size is larger than either 1024 or 2048 bytes, the database server uses the **FET_BUF_SIZE** value.

  For more information about the **FET_BUF_SIZE** environment variable, see the *HCL OneDB™ Guide to SQL: Reference*.

The query plan of a distributed query

You can display the chosen query plan of a distributed query. The information displayed for a distributed join differs from information displayed for a local join.

The following figure shows the chosen query plan for the distributed query.

Figure 68. Selected Output of SET EXPLAIN ALL for Distributed Query, Part 3

```
QUERY:
------
select l.customer_num, l.lname, l.company,
         l.phone, r.call_dtime, r.call_descr
       from customer l, vjp_stores@gilroy:cust_calls r
        where l.customer_num = r.customer_num

Estimated Cost: 9
Estimated # of Rows Returned: 7

  1) informix.r: REMOTE PATH

  2) informix.l: INDEX PATH

    (1) Index Keys: customer_num   (Serial, fragments: ALL)
        Lower Index Filter: informix.l.customer_num = informix.r.customer_num
NESTED LOOP JOIN
```

The following table shows the main differences between the chosen query plans for the distributed join and the local join.

Output Line in Figure 68: Selected Output of SET EXPLAIN ALL for Distributed Query, Part 3 on page 411 for Distributed Query	**Output Line in Figure 58: Result of EXPLAIN AVOID_EXECUTE directives on page 337 for Local-Only Query**	**Description of Difference**
vjp_stores@gilroy: virginia.cust_calls	informix.cust_calls	The remote table name is prefaced with the database and server names.
Estimated Cost: 9	Estimated Cost: 7	The optimizer estimates a higher cost for the distributed query.
informix.r: REMOTE PATH	informix.r: SEQUENTIAL SCAN	The optimizer chose to keep the outer, remote **cust_calls** table at the remote site.
select x0.call_dtime,x0.call_descr,x0. customer_num from vjp_stores:”virginia”.cust_ calls x0		The SQL statement that the local database server sends to the remote site. The remote site reoptimizes this statement to choose the actual plan.

Improve sequential scans

You can improve the performance of sequential read operations on large tables by eliminating repeated sequential scans.

Sequential access to a table other than the first table in the plan is ominous because it threatens to read every row of the table once for every row selected from the preceding tables.

If the table is small, it is harmless to read it repeatedly because the table resides completely in memory. Sequential search of an in-memory table can be faster than searching the same table through an index, especially if maintaining those index pages in memory pushes other useful pages out of the buffers.

When the table is larger than a few pages, however, repeated sequential access produces poor performance. One way to prevent this problem is to provide an index to the column that is used to join the table.

Any user with the Resource privilege can build additional indexes. Use the CREATE INDEX statement to make an index.

An index consumes disk space proportional to the width of the key values and the number of rows. (See Estimating index pages on page 209.) Also, the database server must update the index whenever rows are inserted, deleted, or updated; the index update slows these operations. If necessary, you can use the DROP INDEX statement to release the index after a series of queries, which frees space and makes table updates easier.

Enable view folding to improve query performance

You can significantly improve the performance of a query that involves a view by enabling view folding.

You do this by setting the IFX_FOLDVIEW configuration parameter to `1`.

When view folding is enabled, views are folded into a parent query. Because the views are folded into the parent query, the query results are not placed in a temporary table.

You can use view folding in the following types of queries:

- Views that contain a UNION ALL clause and the parent query incldues a regular join, HCL OneDB™ join, ANSI join, or an ORDER BY clause

View folding does not occur for the following types of queries that perform a UNION ALL operation involving a view:

- The view has one of the following clauses: AGGREGATE, GROUP BY, ORDER BY, UNION, DISTINCT, or OUTER JOIN (either HCL OneDB™ or ANSI type).
- The parent query has a UNION or UNION ALL clause.

In these situations, a temporary table is created to hold query results.

Reduce the join and sort operations

After you understand what the query is doing, you can look for ways to obtain the same output with less effort.

The following suggestions can help you rewrite your query more efficiently:

- Avoid or simplify sort operations.
- Use parallel sorts.
- Use temporary tables to reduce sorting scope.

Avoid or simplify sort operations

In many situations you can determine how to avoid or reduce frequent or complex sort operations.

The sort algorithm is highly tuned and extremely efficient. It is as fast as any external sort program that you might apply to the same data. You do not need to avoid infrequent sorts or sorts of relatively small numbers of output rows.

However, you should try to avoid or reduce the scope of repeated sorts of large tables. The optimizer avoids a sort step whenever it can use an index to produce the output in its proper order automatically. The following factors prevent the optimizer from using an index:

- One or more of the ordered columns is not included in the index.
- The columns are named in a different sequence in the index and the ORDER BY or GROUP BY clause.
- The ordered columns are taken from different tables.

For another way to avoid sorts, see Use temporary tables to reduce sorting scope on page 413.

If a sort is necessary, look for ways to simplify it. As discussed in Sort-time costs on page 315, the sort is quicker if you can sort on fewer or narrower columns.

Related information

Ordering with fragmented indexes on page 418

Use parallel sorts

When you cannot avoid sorting, the database server takes advantage of multiple CPU resources to perform the required sort-and-merge operations in parallel. The database server can use parallel sorts for any query, not just PDQ queries. You can control the number of threads that the database server uses to sort rows.

To control the number of threads that the database server uses to sort rows, use the **PSORT_NPROCS** environment variable.

When PDQ priority is greater than 0 and **PSORT_NPROCS** is greater than 1, the query benefits both from parallel sorts and from PDQ features such as parallel scans and additional memory. Users can use the **PDQPRIORITY** environment variable to request a specific proportion of PDQ resources for a query. You can use the MAX_PDQPRIORITY configuration parameter to limit the number of such user requests. For more information, see Limiting PDQ resources in queries on page 44.

In some cases, the amount of data being sorted can overflow the memory resources allocated to the query, resulting in I/O to a dbspace or sort file. For more information, see Configure dbspaces for temporary tables and sort files on page 114.

Use temporary tables to reduce sorting scope

You can use a temporary, ordered subset of a table to increase the speed of a query. The temporary table can also simplify the work of the query optimizer, cause the optimizer to avoid multiple-sort operations, and simplify the work of the optimizer in other ways.

For example, suppose your application produces a series of reports on customers who have outstanding balances, one report for each major postal area, ordered by customer name. In other words, a series of queries occurs, each of the following form (using hypothetical table and column names):

```
SELECT cust.name, rcvbles.balance, ...other columns...
   FROM cust, rcvbles
   WHERE cust.customer_id = rcvbles.customer_id
      AND rcvbls.balance > 0
      AND cust.postcode LIKE '98_ _ _'
   ORDER BY cust.name
```

This query reads the entire **cust** table. For every row with the specified postal code, the database server searches the index on **rcvbles.customer_id** and performs a nonsequential disk access for every match. The rows are written to a temporary file and sorted. For more information about temporary files, see Configure dbspaces for temporary tables and sort files on page 114.

This procedure is acceptable if the query is performed only once, but this example includes a series of queries, each incurring the same amount of work.

An alternative is to select all customers with outstanding balances into a temporary table, ordered by customer name, as the following example shows:

```
SELECT cust.name, rcvbles.balance, ...other columns...
   FROM cust, rcvbles
   WHERE cust.customer_id = rcvbles.customer_id
      AND cvbls.balance > 0
   INTO TEMP cust_with_balance
```

You can then execute queries against the temporary table, as the following example shows:

```
SELECT *
   FROM cust_with_balance
   WHERE postcode LIKE '98_ _ _'
   ORDER BY cust.name
```

Each query reads the temporary table sequentially, but the table has fewer rows than the primary table.

Configuring memory for queries with hash joins, aggregates, and other memory-intensive elements

Certain configuration parameters can be set to provide more memory for queries that require sorting, hash joins, aggregates, and other memory-intensive elements.

How you configure the amount of memory that is available for a query depends on whether or not the query is a Parallel Database Query (PDQ).

Configuring memory for non-PDQ queries

If the PDQ priority is set to `0` (zero), you can change the amount of memory that is available for a query that is not a PDQ query by changing the setting of the DS_NONPDQ_QUERY_MEM configuration parameter. You can only use this parameter if the PDQ priority is set to zero. Its setting has no effect if the PDQ priority is greater than zero.

You can also change the value of DS_NONPDQ_QUERY_MEM with an **onmode -wm** or onmode -wf command.

For example, if you use the onmode utility, specify a value as shown in the following example:

```
onmode -wf DS_NONPDQ_QUERY_MEM=500
```

The minimum value for DS_NONPDQ_QUERY_MEM is 128 kilobytes. The maximum supported value is 25 percent of DS_TOTAL_MEMORY. 128 kilobytes is the default value of DS_NONPDQ_QUERY_MEM. If you specify a value for the DS_NONPDQ_QUERY_MEM parameter, determine and adjust the value based on the number and size of table rows involved in the query.

HCL OneDB™ might recalculate the value of DS_NONPDQ_QUERY_MEM initialization if the value is more than 25 percent of the DS_TOTAL_MEMORY value.

If HCL OneDB™ changes the value that you set, the server sends a message in this format:

```
DS_NONPDQ_QUERY_MEM recalculated and changed from old_value Kb to new_value Kb.
```

In the message, `old_value` represents the value that you assigned to DS_NONPDQ_QUERY_MEM in the user configuration file, and `new_value` represents the value determined by HCL OneDB™.

For formulas for estimating the amount of additional space to allocate for hash joins, see Estimating temporary space for dbspaces and hash joins on page 118.

Configuring memory for PDQ queries

The Memory Grant Manager (MGM) component of HCL OneDB™ coordinates the use of memory, CPU virtual processors (VPs), disk I/O, and scan threads among decision-support queries. The MGM uses the DS_MAX_QUERIES, DS_TOTAL_MEMORY, DS_MAX_SCANS, and MAX_PDQPRIORITY configuration parameter settings to determine the quantity of these PDQ resources that can be granted to a decision-support query. The MGM also grants memory to a query for such activities as hash joins. For more information about the MGM, see The Memory Grant Manager on page 351.

Optimize user-response time for queries

You can influence the amount of time that OneDB takes to optimize a query and to return rows to a user.

Optimization level

You normally obtain optimum overall performance with the default optimization level, HIGH. The time that it takes to optimize the statement is usually unimportant. However, if experimentation with your application reveals that your query is still taking too long, you can set the optimization level to LOW.

If you change the optimization level to LOW, check the SET EXPLAIN output to see if the optimizer chose the same query plan as before.

To specify a HIGH or LOW level of database server optimization, use the SET OPTIMIZATION statement.

Related information

SET OPTIMIZATION statement on page

Optimization goals

Optimizing total query time and optimizing user-response time are two optimization goals for improving query performance.

Total query time is the time it takes to return all rows to the application. Total query time is most important for batch processing or for queries that require all rows be processed before returning a result to the user, as in the following query:

```
SELECT count(*) FROM orders
WHERE order_amount > 2000;
```

User-response time is the time that it takes for the database server to return a screen full of rows back to an interactive application. In interactive applications, only a screen full of data can be requested at one time. For example, the user application can display only 10 rows at one time for the following query:

```
SELECT * FROM orders
WHERE order_amount > 2000;
```

Which optimization goal is more important can have an effect on the query path that the optimizer chooses. For example, the optimizer might choose a nested-loop join instead of a hash join to execute a query if user-response time is most important, even though a hash join might result in a reduction in total query time.

Specifying the query performance goal

You can optimize user response time for your entire database server system, within a session, or for individual queries.

The default behavior is for the optimizer to choose query plans that optimize the total query time. You can specify optimization of user-response time at several different levels:

- For the database server system

  To optimize user-response time, set the OPT_GOAL configuration parameter to `0`, as in the following example:

  ```
  OPT_GOAL 0
  ```

  Set OPT_GOAL to `-1` to optimize total query time.

- For the user environment

  The **OPT_GOAL** environment variable can be set before the user application starts.

  UNIX™ Only

  To optimize user-response time, set the **OPT_GOAL** environment variable to `0`, as in the following sample commands:

  ```
  Bourne shell                OPT_GOAL = 0
                 export OPT_GOAL
  ```

```
C shell               setenv OPT_GOAL 0
```

For total-query-time optimization, set the **OPT_GOAL** environment variable to `-1`.

- Within the session

  You can control the optimization goal with the SET OPTIMIZATION statement in SQL. The optimization goal set with this statement stays in effect until the session ends or until another SET OPTIMIZATION statement changes the goal.

  The following statement causes the optimizer to choose query plans that favor total-query-time optimization:

  ```
  SET OPTIMIZATION ALL_ROWS
  ```

  The following statement causes the optimizer to choose query plans that favor user-response-time optimization:

  ```
  SET OPTIMIZATION FIRST_ROWS
  ```

- For individual queries

  You can use FIRST_ROWS and ALL_ROWS optimizer directives to instruct the optimizer which query goal to use. For more information about these directives, see Optimization-goal directives on page 335.

The precedence for these levels is as follows:

- Optimizer directives
- SET OPTIMIZATION statement
- **OPT_GOAL** environment variable
- OPT_GOAL configuration parameter

For example, optimizer directives take precedence over the goal that the SET OPTIMIZATION statement specifies.

Preferred query plans for user-response-time optimization

When the optimizer chooses query plans to optimize user-response time, it computes the cost for retrieving the first row in the query for each plan and chooses the plan with the lowest cost. In some cases, the query plan with the lowest cost for retrieving the first row might not be the optimal path to retrieve all rows in the query.

The following sections explain some of the possible differences in query plans.

Nested-loop joins versus hash joins

Hash joins generally have a higher cost to retrieve the first row than nested-loop joins do. The database server must build the hash table before it retrieves any rows. However, in some cases, total query time is faster if the database server uses a hash join.

In the following example, **tab2** has an index on **col1**, but **tab1** does not have an index on **col1**. When you execute SET OPTIMIZATION ALL_ROWS before you run the query, the database server uses a hash join and ignores the existing index, as the following portion of SET EXPLAIN output shows:

```
QUERY:
------
SELECT * FROM tab1,tab2
WHERE tab1.col1 = tab2.col1
Estimated Cost: 125
Estimated # of Rows Returned: 510
1) lsuto.tab2: SEQUENTIAL SCAN
2) lsuto.tab1: SEQUENTIAL SCAN
DYNAMIC HASH JOIN
    Dynamic Hash Filters: lsuto.tab2.col1 = lsuto.tab1.col1
```

However, when you execute SET OPTIMIZATION FIRST_ROWS before you run the query, the database server uses a nested-loop join. The clause (FIRST_ROWS OPTIMIZATION) in the following partial SET EXPLAIN output shows that the optimizer used user-response-time optimization for the query:

```
QUERY:        (FIRST_ROWS OPTIMIZATION)
------
SELECT * FROM tab1,tab2
WHERE tab1.col1 = tab2.col1
Estimated Cost: 145
Estimated # of Rows Returned: 510
1) lsuto.tab1: SEQUENTIAL SCAN
2) lsuto.tab2: INDEX PATH
    (1) Index Keys: col1
        Lower Index Filter: lsuto.tab2.col1 = lsuto.tab1.col1
NESTED LOOP JOIN
```

Table scans versus index scans

In cases where the database server returns a large number of rows from a table, the lower-cost option for the total-query-time goal might be to scan the table instead of using an index. However, to retrieve the first row, the lower-cost option for the user-response-time goal might be to use the index to access the table.

Ordering with fragmented indexes

When an index is not fragmented, the database server can use the index to avoid a sort. However, when an index is fragmented, the ordering can be guaranteed only within the fragment, not between fragments.

Usually, the least expensive option for the total-query-time goal is to scan the fragments in parallel and then use the parallel sort to produce the proper ordering. However, this option does not favor the user-response-time goal.

Instead, if the user-response time is more important, the database server reads the index fragments in parallel and merges the data from all of the fragments. No additional sort is generally needed.

Related information

Avoid or simplify sort operations on page 413

Optimize queries for user-defined data types

Queries that access user-defined data types (UDTs) can take advantage of the same performance features that built-in data types use.

These features are:

- Indexes

  If a query accesses a small number of rows, an index speeds retrieval because the database server does not need to read every page in a table to find the rows. For more information, see Indexes on user-defined data types on page 229.

- Parallel database query (PDQ)

  Queries that access user-defined data can take advantage of parallel scans and parallel execution.

  To turn on parallel execution for a query, set the **PDQPRIORITY** environment variable or use the SQL statement SET PDQPRIORITY. For more information about how to set PDQ priority and configuration parameters that affect PDQ, see The allocation of resources for parallel database queries on page 352.

- Optimizer directives

In addition, programmers can write the following functions or UDRs to help the optimizer create an efficient query plan for your queries:

- Parallel UDRs that can take advantage of parallel database queries
- User-defined selectivity functions that calculate the expected fraction of rows that qualify for the function
- User-defined cost functions that calculate the expected relative cost to execute a user-defined routine
- User-defined statistical functions that the UPDATE STATISTICS statement can use to generate statistics and data distributions
- User-defined negator functions to allow more choices for the optimizer

The following sections summarize these techniques. For a more complete description of how to write and register user-defined selectivity functions and user-defined cost functions, see *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

Selectivity and cost functions

You can use the CREATE FUNCTION statement to create a UDR. Then, you can use routine modifiers to change the cost or selectivity that is specified in the statement.

After you create a UDR, you can place it in an SQL statement.

The following example shows how you can place a UDR in an SQL statement:

```
SELECT * FROM image
WHERE get_x1(image.im2) and get_x2(image.im1)
```

The optimizer cannot accurately evaluate the cost of executing a UDR without additional information. You can provide the cost and selectivity of the function to the optimizer. The database server uses cost and selectivity together to determine the best path. For more information about selectivity, see Filters with user-defined routines on page 366.

In the previous example, the optimizer cannot determine which function to execute first, the **get_x1** function or the **get_x2** function. If a function is expensive to execute, the DBA can assign the function a larger cost or selectivity, which can influence the optimizer to change the query plan for better performance. In the previous example, if **get_x1** costs more to execute, the DBA can assign a higher cost to the function, which can cause the optimizer to execute the **get_x2** function first.

You can add the following routine modifiers to the CREATE FUNCTION statement to change the cost or selectivity that the optimizer assigns to the function:

- **selfunc**=*function_name*
- **selconst**=*integer*
- **costfunc**=*function_name*
- **percall_cost**=*integer*

For more information about cost or selectivity modifiers, see the *HCL OneDB™ User-Defined Routines and Data Types Developer's Guide*.

User-defined statistics for UDTs

Because information about the nature and use of a user-defined data type (UDT) is not available to the database server, it cannot collect distributions or the **colmin** and **colmax** information (found in the **syscolumns** system catalog table) for a UDT. Instead, you can create a special function that populates these statistics.

The database server runs the statistics collection function when you execute UPDATE STATISTICS.

For more information about the importance of updating statistics, see Statistics held for the table and index on page 311. For information about improving performance, see Updating statistics for columns with user-defined data types on page 384.

Optimize queries with the SQL statement cache

Before the database server runs an SQL statement, it must first parse and optimize the statement. Optimizing statements can be time consuming, depending on the size of the SQL statement.

The database server can store the optimized SQL statement in the virtual portion of shared memory, in an area that is called the SQL statement cache. The SQL statement cache (SSC) can be accessed by all users, and it allows users to bypass the optimize step before they run the query. This capability can result in the following significant performance improvements:

- Reduced response times when users are running the same SQL statements.

  SQL statements that take longer to optimize (usually because they include many tables and many filters in the WHERE clause) run faster from the SQL statement cache because the database server does not optimize the statement.
- Reduced memory usage because the database server shares query data structures among users.

  Memory reduction with the SQL statement cache is greater when a statement has many column names in the select list.

For more information about the effect of the SQL statement cache on the performance of the overall system, see Monitor and tune the SQL statement cache on page 89.

When to use the SQL statement cache

Applications might benefit from use of the SQL statement cache if multiple users execute the same SQL statements. The database server considers statements to be the same if all characters match exactly.

For example, if 50 sales representatives execute the **add_order** application throughout the day, they all execute the same SQL statements if the application contains SQL statements that use host variables, such as the following example:

```
SELECT * FROM ORDERS WHERE order_num = :hostvar
```

This kind of application benefits from use of the SQL statement cache because users are likely to find the SQL statements in the SQL statement cache.

The database server does not consider the following SQL statements exact matches because they contain different literal values in the WHERE clause:

```
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
   AND order_date > "01/01/07"
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
   AND order_date > "01/01/2007"
```

Performance does not improve with the SQL statement cache in the following situations:

- If a report application is run once nightly, and it executes SQL statements that no other application uses, it does not benefit from use of the statement cache.
- If an application prepares a statement and then executes it many times, performance does not improve with the SQL statement cache because the statement is optimized just once during the PREPARE statement.

When a statement contains host variables, the database server replaces the host variables with placeholders when it stores the statement in the SQL statement cache. Therefore, the statement is optimized without the database server having access to the values of the host variables. In some cases, if the database server had access to the values of the host variables, the statement might be optimized differently, usually because the distributions stored for a column inform the optimizer exactly how many rows pass the filter.

If an SQL statement that contains host variables performs poorly with the SQL statement cache turned on, try flushing the SQL statement cache with the **onmode -e flush** command and running the query with values that are more frequently used across multiple executions of the query. When you flush the cache, the database server reoptimizes the query and generates a query plan that is optimized for these frequently used values.

Important: The database server flushes an entry from the SQL statement cache only if it is not in use. If an application prepares the statement and keeps it, the entry is still in use. In this case, the application needs to close the statement before the flush is beneficial.

Using the SQL statement cache

The DBA usually makes the decision to enable the SQL statement cache. If the SQL statement cache is enabled, individual users can decide whether or not to use the SQL statement cache for their specific environment or application.

The database server incurs some processing overhead in managing the SQL statement cache, so you should use the SQL statement cache only when multiple users want to share the SQL statements.

To enable the SQL statement cache, set the STMT_CACHE configuration parameter to a value that defines either of the following modes:

- Always use the SQL statement cache unless a user explicitly specifies do not use the cache.
- Use the SQL statement cache only when a user explicitly specifies use it.

For more information, see Enabling the SQL statement cache on page 422. For more information about the STMT_CACHE configuration parameter, see the *HCL OneDB™ Administrator's Reference*.

Enabling the SQL statement cache

The database server does not use the SQL statement cache when the STMT_CACHE configuration parameter is `0` (the default value). You can change this value to enable the SQL statement cache in one of two modes.

Use one of the following methods to change this STMT_CACHE default value:

- Update the ONCONFIG file to specify the STMT_CACHE configuration parameter and restart the database server.

  If you set the STMT_CACHE configuration parameter to `1`, the database server uses the SQL statement cache for an individual user when the user sets the **STMT_CACHE** environment variable to `1` or executes the SET STATEMENT CACHE ON statement within an application.

  ```
  STMT_CACHE 1
  ```

  If the STMT_CACHE configuration parameter is `2`, the database server stores SQL statements for all users in the SQL statement cache except when individual users turn off the feature with the **STMT_CACHE** environment variable or the SET STATEMENT CACHE OFF statement.

  ```
  STMT_CACHE 2
  ```

- Use the **onmode -e** command to override the STMT_CACHE configuration parameter dynamically.

  If you use the **enable** keyword, the database server uses the SQL statement cache for an individual user when the user sets the **STMT_CACHE** environment variable to `1` or executes the SET STATEMENT CACHE ON statement within an application.

```
onmode -e enable
```

If you use the **on** keyword, the database server stores SQL statements for all users in the SQL statement cache except when individual users turn off the feature with the **STMT_CACHE** environment variable or the SET STATEMENT CACHE OFF statement.

```
onmode -e on
```

The following table summarizes the use of the SQL statement cache, which depends on the setting of the STMT_CACHE configuration parameter (or the execution of **onmode -e**) and the use in an application of the **STMT_CACHE** environment variable and the SET STATEMENT CACHE statement.

STMT_ CACHE Configuration Parameter or onmode -e	**STMT_CACHE Environment Variable**	**SET STATEMENT CACHE Statement**	**Resulting Behavior**
0 (default)	Not applicable	Not applicable	Statement cache not used
1	0 (or not set)	OFF	Statement cache not used
1	1	OFF	Statement cache not used
1	0 (or not set)	ON	Statement cache used
1	1	ON	Statement cache used
1	1	Not executed	Statement cache used
1	0	Not executed	Statement cache not used
2	1 (or not set)	ON	Statement cache used
2	1 (or not set)	OFF	Statement cache not used
2	0	ON	Statement cache used
2	0	OFF	Statement cache not used by user
2	0	Not executed	Statement cache not used by user
2	1 (or not set)	Not executed	Statement cache used by user

Placing statements in the cache

SELECT, UPDATE, INSERT and DELETE statements can be placed in the SQL statement cache, with some exceptions. When the database server checks if an SQL statement is in the cache, it must find an exact match.

For a complete list of the exceptions and a list of requirements for an exact match, see SET STATEMENT CACHE in the *HCL OneDB™ Guide to SQL: Syntax*.

Monitoring memory usage for each session

You can use several onstat -g command options to obtain memory information for each session.

About this task

You obtain memory information by identifying the SQL statements that use a large amount of memory.

To identify SQL statements using large amount of memory:

1. Run the onstat -g ses command to display memory of all sessions and see which session has the highest memory usage.
2. Run the onstat -g ses *session-id* command to display more details on the session with the highest memory usage.
3. Run the onstat -g stm *session-id* command to display the memory used by the SQL statements.

Display all user threads and session memory usage

Use the onstat -g ses command to display all user sessions and memory usage by session ID.

When the session shares the memory structures in the SSC, the value in the **used memory** column should be lower than when the cache is turned off. For example, Figure 69: onstat -g ses output when the SQL statement cache is not enabled on page 424 shows sample onstat -g ses output when the SQL statement cache is not enabled. Figure 70: onstat -g ses output when the SQL statement cache is enabled on page 424 shows output after the SQL statement cache is enabled and the queries in Session 4 are run again. Figure 69: onstat -g ses output when the SQL statement cache is not enabled on page 424 shows that Session 4 has 45656 bytes of used memory. Figure 70: onstat -g ses output when the SQL statement cache is enabled on page 424 shows that Session 4 has less used bytes (36920) when the SQL statement cache is enabled.

Figure 69. onstat -g ses output when the SQL statement cache is not enabled

```
session                                     #RSAM    total    used     dynamic
id       user     tty      pid      hostname threads  memory   memory   explain
12       informix -        0        -        0        12288    7632     off
4        informix 11       5158     smoke    1        53248    45656    off
3        informix -        0        -        0        12288    8872     off
2        informix -        0        -        0        12288    7632     off
```

Figure 70. onstat -g ses output when the SQL statement cache is enabled

```
session                                     #RSAM    total    used     dynamic
id       user     tty      pid      hostname threads  memory   memory   explain
17       informix -        0        -        0        12288    7632     off
16       informix 12       5258     smoke    1        40960   38784     off
4        informix 11       5158     smoke    1        53248   36920     off
3        informix -        0        -        0        12288    8872     off
2        informix -        0        -        0        12288    7632     off
```

Figure 70: onstat -g ses output when the SQL statement cache is enabled on page 424 also shows the memory allocated and used for Session 16, which runs the same SQL statements as Session 4. Session 16 allocates less total memory (40960) and uses less memory (38784) than Session 4 (Figure 69: onstat -g ses output when the SQL statement cache is not enabled

on page 424 shows 53248 and 45656) because Session 16 uses the existing memory structures in the SQL statement cache.

Display detailed session information and memory usage

Use the onstat -g ses *session-id* command to display detailed information for a session, including memory usage.

The following onstat -g ses *session-id* output columns display memory usage:

- The `Memory pools` portion of the output
  - The **totalsize** column shows the number of bytes currently allocated
  - The **freesize** column shows the number of unallocated bytes
- The last line of the output shows the number of bytes allocated from the sscpool.

Figure 71: onstat -g ses session-id output on page 425 shows that Session 16 has currently allocated 69632 bytes, of which 11600 bytes are allocated from the sscpool.

Figure 71. onstat -g ses session-id output

```
onstat -g ses 14


session                                     #RSAM    total      used
id       user     tty      pid      hostname threads  memory     memory
14       virginia 7        28734    lyceum   1        69632      67384

tid      name     rstcb    flags    curstk   status
38       sqlexec  a3974d8  Y--P---  1656     cond wait(netnorm)

Memory pools    count 1
name         class addr     totalsize freesize #allocfrag #freefrag
14           V     a974020  69632     2248    156        2

...
Sess  SQL            Current            Iso Lock       SQL  ISAM F.E.
Id    Stmt type      Database           Lvl Mode       ERR  ERR  Vers
14    SELECT         vjp_stores         CR  Not Wait   0    0    9.03

Current statement name : slctcur

Current SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

Last parsed SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

11600 byte(s) of memory is allocated from the sscpool
```

Display information about session SQL statements

Use the onstat -g sql *session-id* or onstat -g spf commands to display information about the SQL statements executed by a session.

The following figure shows that onstat -g sql *session-id* displays the same information as the bottom portion of the onstat -g ses *session-id* command in Figure 71: onstat -g ses session-id output on page 425, which includes the number of bytes allocated from the sscpool.

Figure 72. onstat -g sql session-id output

```
onstat -g sql 14


Sess  SQL            Current            Iso Lock       SQL  ISAM F.E.
Id    Stmt type      Database           Lvl Mode       ERR  ERR  Vers
14    SELECT         vjp_stores         CR  Not Wait   0    0    9.03

Current statement name : slctcur

Current SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

Last parsed SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

11600 byte(s) of memory is allocated from the sscpool
```

Monitoring usage of the SQL statement cache

If you notice a sudden increase in response time for a query that had been using the SQL statement cache, the entry might have been dropped or deleted. You can monitor the usage of the SQL statement cache and check for a dropped or deleted entry by displaying onstat -g ssc command output.

The database server drops an entry from the cache when one of the objects that the query depends on is altered so that it invalidates the data dictionary cache entry for the query. The following operations cause a dependency check failure:

- Execution of any data definition language (DDL) statement (such as ALTER TABLE, DROP INDEX, or CREATE INDEX) that might alter the query plan
- Alteration of a table that is linked to another table with a referential constraint (in either direction)
- Execution of UPDATE STATISTICS FOR TABLE for any table or column involved in the query
- Renaming a column, database, or index with the RENAME statement

When an entry is marked as dropped or deleted, the database server must reparse and reoptimize the SQL statement the next time it executes. For example, Figure 73: Sample onstat -g ssc command output for a dropped entry on page 427 shows the entries that the onstat -g ssc command displays after UPDATE STATISTICS was executed on the **items** and **orders** table between the execution of the first and second SQL statements.

The `Statement Cache Entries` portion of the onstat -g ssc output in Figure 73: Sample onstat -g ssc command output for a dropped entry on page 427 displays a **flag** field that indicates whether or not an entry has been dropped or deleted from the SQL statement cache.

- The first entry has a **flag** column with the value `DF`, which indicates that the entry is fully cached, but is now dropped because its entry was invalidated.
- The second entry has the same statement text as the third entry, which indicates that it was reparsed and reoptimized when it was executed after the UPDATE STATISTICS statement.

Figure 73. Sample onstat -g ssc command output for a dropped entry

```
onstat -g ssc

...
Statement Cache Entries:

lru hash ref_cnt hits flag heap_ptr  database     user
--------------------------------------------------------------------------------
...
  2  232       1    1   DF  aa3d020  vjp_stores   virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num

  3  232       1    0   -F  aa8b020  vjp_stores   virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num
...
```

Invalidating a statement

You can selectively invalidate entries of your choice by setting the sysmaster:syssscelem:valid column to 0 as user Informix

For example, Figure 2 on page 428 shows the entries that the **onstat -g ssc** command displays before and after invalidating a query from the items table

The `Statement Cache Entries` portion of the **onstat -g ssc** output in Figure 2 on page 428 displays a **flag** field that indicates whether or not an entry has been invalidated in the SQL statement cache.

Figure 74. Sample onstat -g ssc command output for an invalidate entry

```
onstat -g ssc snipit

...
Statement Cache Entries:

uniqid lru hash ref_cnt hits flag heap_ptr       database           user
--------------------------------------------------------------------------------
...
    7   1 2404       0    0     F 463d0438       stores_demo        informix

select count(*) from items

...
```

```
Invalidate it:

update syssscelem set valid = 0 where uniqid = 7;

Confirm it is invalid with onstat -g ssc:

Statement Cache Entries:

uniqid lru hash ref_cnt hits flag heap_ptr       database           user
--------------------------------------------------------------------------------
...
    7   1 2404       0    0    DF 463d0438       stores_demo        informix

select count(*) from items
```

The user can confirm a flag of 'D' in 'onstat -g ssc' output and can query sysmaster:syssscelem to confirm 'valid' column is 0.

Note: Invalid entry cannot be changed to valid.

Locking a statement

You can lock an entry of your choice in the Statement Cache even when UPDATE STATISTICS is executed on tables in the sql statement.

For example, Figure 3 on page 429 shows the entries that the onstat -g ssc command displays after UPDATE STATISTICS was executed on the **items** table between the execution of the first and second SQL statements.

The `Statement Cache Entries` portion of the onstat -g ssc output in Figure 3 on page 429 displays a **flag** field that indicates whether or not an entry has been locked in the SQL statement cache.

Figure 75. Sample onstat -g ssc command output for locking an entry

```
onstat -g ssc snipit

...
Statement Cache Entries:

uniqid lru hash ref_cnt hits flag heap_ptr       database           user
--------------------------------------------------------------------------------
...
    7   1 2404       0    0     F 463d0438       stores_demo        informix
  select count(*) from items

  3  232       1    0   -F  aa8b020  vjp_stores   virginia

...
```

```
Lock it:

update syssscelem set locked = 1 where uniqid = 7;

Confirm it is locked with onstat -g ssc:

Statement Cache Entries:

uniqid lru hash ref_cnt hits flag heap_ptr       database           user
--------------------------------------------------------------------------------
...
    7   1 2404       0    0    FL 463d0438       stores_demo        informix

select count(*) from items
```

The user can confirm a flag of 'L' in 'onstat -g ssc' output and can query sysmaster:syssscelem to confirm 'locked' column is 1.

Note: Statements can be locked and unlocked as many times as desired.

Monitor sessions and threads

You can monitor the number of active sessions and threads and the amount of resources that they are using. Monitoring sessions and threads is important for sessions that perform queries as well as sessions that perform inserts, updates, and deletes.

Some of the information that you can monitor for sessions and threads allows you to determine if an application is using a disproportionate amount of the resources.

Note: Session threads for a stored procedure with a PDQ priority setting and a GROUP BY clause are not released until a session is completed.

Monitor sessions and threads with onstat commands

You can use several onstat utility commands to monitor active sessions and threads.

Use the following **onstat** utility commands to monitor sessions and threads:

- onstat -u
- onstat -g ath
- onstat -a act
- onstat -a cpu
- onstat -a ses
- onstat -g mem
- onstat -g stm

Monitor blocking threads with the onstat -g bth and onstat -g BTH commands

Running threads take ownership of various objects and resources; for example, buffers, locks, mutexes, decision support memory, and others. Contention for these resources among hundreds or thousands of threads can result in chains of dependencies. Use the onstat -g bth command to display the dependencies between blocking and waiting threads. Use the onstat -g BTH command to display session and stack information for the blocking threads.

For example, a thread that is blocked waiting to enter a critical section might own a row lock for which another thread is waiting. The second thread might be blocking a third thread that is waiting in the MGM query queue. Usually, the duration of such contention is short. However, if a thread is blocked long enough to be noticed, you might need to identify the source of the contention. The onstat -g bth command discovers the chains of dependency and displays blocker threads followed by waiting threads, in order. You can use the resulting picture of contentions to diagnose and correct the issues.

The following example of the onstat -g bth command output has multiple threads that are waiting on resources.

Figure 76. The ouptut of the onstat -g bth command

```
This command attempts to identify any blocking threads.

Highest level blocker(s)
 tid      name                session
 48       sqlexec             26

Threads waiting on resources
 tid      name                blocking resource             blocker
 49       sqlexec             MGM                           48
 13       readahead_0         Condition (ReadAhead)          -
 50       sqlexec             Lock (0x4411e578)             49
 51       sqlexec             Lock (0x4411e578)             49
 52       sqlexec             Lock (0x4411e578)             49
 53       sqlexec             Lock (0x4411e578)             49
 57       bf_priosweep()      Condition (bp_cond)            -
 58       scan_1.0            Condition (await_MC1)          -
 59       scan_1.0            Condition (await_MC1)          -

Run 'onstat -g BTH' for more info on blockers.
```

In this example, four threads are waiting for a lock that is owned by thread 49. Thread 49 is waiting for MGM resources that are owned by thread 48. If you run the onstat -g BTH command, the output shows the session and stack information for the blocking thread, which in this case is thread 48.

Related reference

onstat -g bth and -g BTH: Print blocked and waiting threads

Monitor threads with onstat –u output

Use the onstat –u command to display information about active threads that require a database server task-control block.

Active threads include threads that belong to user sessions, as well as some that correspond to database server daemons (for example, page cleaners). Figure 77: onstat -u output on page 432 shows an example of onstat -u output.

Also use the onstat -u command to determine if a user is waiting for a resource or holding too many locks, or to get an idea of how much I/O the user has performed.

The utility output displays the following information:

- The address of each thread
- Flags that indicate the present state of the thread (for example, waiting for a buffer or waiting for a checkpoint), whether the thread is the primary thread for a session, and what type of thread it is (for example, user thread, daemon thread, and so on)

  For information on these flags, see the *HCL OneDB™ Administrator's Reference*.
- The session ID and user login ID for the session to which the thread belongs

A session ID of `0` indicates a daemon thread.

- Whether the thread is waiting for a specific resource and the address of that resource
- The number of locks that the thread is holding
- The number of read calls and the number of write calls that the thread has executed
- The maximum number of current, active user threads

If you execute onstat -u while the database server is performing fast recovery, several database server threads might appear in the display.

Figure 77. onstat -u output

```
Userthreads
address  flags   sessid   user     tty      wait     tout locks nreads   nwrites
80eb8c   ---P--D 0        informix -        0        0    0     33       19
80ef18   ---P--F 0        informix -        0        0    0     0        0
80f2a4   ---P--B 3        informix -        0        0    0     0        0
80f630   ---P--D 0        informix -        0        0    0     0        0
80fd48   ---P--- 45       chrisw   ttyp3    0        0    1     573      237
810460   ------- 10       chrisw   ttyp2    0        0    1     1        0
810b78   ---PR-- 42       chrisw   ttyp3    0        0    1     595      243
810f04   Y------ 10       chrisw   ttyp2    beacf8   0    1     1        0
811290   ---P--- 47       chrisw   ttyp3    0        0    2     585      235
81161c   ---PR-- 46       chrisw   ttyp3    0        0    1     571      239
8119a8   Y------ 10       chrisw   ttyp2    a8a944   0    1     1        0
81244c   ---P--- 43       chrisw   ttyp3    0        0    2     588      230
8127d8   ----R-- 10       chrisw   ttyp2    0        0    1     1        0
812b64   ---P--- 10       chrisw   ttyp2    0        0    1     20       0
812ef0   ---PR-- 44       chrisw   ttyp3    0        0    1     587      227
 15 active, 20 total, 17 maximum concurrent
```

Related reference

onstat -u command: Print user activity profile

Monitor threads with onstat -g ath output

Use the onstat -g ath command to view a list of all threads. Unlike the onstat –u command, this list includes internal daemon threads that do not have a database server task-control block.

The onstat -g ath command display does not include the session ID (because not all threads belong to sessions).

The `status` field contains information on the status of thread, such as `running`, `cond wait`, `IO Idle`, `IO Idle`, `sleeping secs: number_of_seconds`, or `sleeping forever`. The following output example identifies many threads as `sleeping forever`. To improve performance, you can remove or reduce the number of threads that are identified as `sleeping forever`.

Figure 78. onstat -g ath output

```
Threads:
tid  tcb          rstcb        prty  status             vp-class   name
2    10bbf36a8     0            2    sleeping forever      3lio    lio vp 0
3    10bc12218     0            2    sleeping forever      4pio    pio vp 0
4    10bc31218     0            2    sleeping forever      5aio    aio vp 0
5    10bc50218     0            2    sleeping forever      6msc    msc vp 0
6    10bc7f218     0            2    sleeping forever      7aio    aio vp 1
7    10bc9e540     10b231028    4    sleeping secs: 1      1cpu    main_loop()
8    10bc12548     0            2    running               1cpu    tlitcppoll
9    10bc317f0     0            3    sleeping forever      1cpu    tlitcplst
10   10bc50438     10b231780    2    sleeping forever      1cpu    flush_sub(0)
11   10bc7f740     0            2    sleeping forever      8aio    aio vp 2
12   10bc7fa00     0            2    sleeping forever      9aio    aio vp 3
13   10bd56218     0            2    sleeping forever     10aio    aio vp 4
14   10bd75218     0            2    sleeping forever     11aio    aio vp 5
15   10bd94548     10b231ed8    3    sleeping forever      1cpu    aslogflush
16   10bc7fd00     10b232630    1    sleeping secs: 26      1cpu    btscanner 0
32   10c738ad8     10b233c38    4    sleeping secs: 1       1cpu    onmode_mon
50   10c0db710     10b232d88    2    cond wait  netnorm     1cpu    sqlexec
```

Threads that a primary decision-support thread started have a name that indicates their role in the decision-support query. The following figure shows four scan threads that belong to a decision-support thread.

Figure 79. onstat -g ath output showing scan threads belonging to a decision-support thread

```
Threads:
tid  tcb          rstcb      prty  status               vp-class   name
11   994060       0           4    sleeping(Forever)       1cpu    kaio
12   994394       80f2a4      2    sleeping(secs: 51)      1cpu    btclean
26   99b11c       80f630      4    ready                   1cpu    onmode_mon
32   a9a294       812b64      2    ready                   1cpu    sqlexec
113  b72a7c       810b78      2    ready                   1cpu    sqlexec
114  b86c8c       81244c      2    cond wait(netnorm)      1cpu    sqlexec
115  b98a7c       812ef0      2    cond wait(netnorm)      1cpu    sqlexec
116  bb4a24       80fd48      2    cond wait(netnorm)      1cpu    sqlexec
117  bc6a24       81161c      2    cond wait(netnorm)      1cpu    sqlexec
118  bd8a24       811290      2    ready                   1cpu    sqlexec
119  beae88       810f04      2    cond wait(await_MC1)    1cpu    scan_1.0
120  a8ab48       8127d8      2    ready                   1cpu    scan_2.0
121  a96850       810460      2    ready                   1cpu    scan_2.1
122  ab6f30       8119a8      2    running                 1cpu    scan_2.2
```

Related reference

onstat -g ath command: Print information about all threads

Related information

Improve connection performance and scalability on page 49

Monitor threads with onstat -g act output

Use the onstat -g act command to obtain a list of active threads. The onstat -g act output shows a subset of the threads that are also listed in onstat -g ath output.

For sample output, see the *HCL OneDB™ Administrator's Reference*.

Related reference

onstat -g act command: Print active threads

Monitor threads with onstat -g cpu output

Use the onstat –g cpu command to display the last time the thread ran, how much CPU time the thread used, the number of times the thread ran, and other statistics about all the threads running in the server

The following output example shows the ID and name of each thread that is running, the ID of the virtual processor in which each thread is running, the day and time when each thread last ran, how much CPU time each thread used, the number of times each thread was scheduled to run, and the status of each thread.

Figure 80. onstat -g cpu command output

```
Thread CPU Info:
tid     name              vp       Last Run           CPU Time     #scheds    status
2       lio vp 0          3lio*    07/18 08:35:35        0.0000          1    IO Idle
3       pio vp 0          4pio*    07/18 08:35:36        0.0102          2    IO Idle
4       aio vp 0          5aio*    07/18 08:35:47        0.6876         68    IO Idle
5       msc vp 0          6msc*    07/18 11:47:24        0.0935         14    IO Idle
6       main_loop()       1cpu*    07/18 15:02:43        2.9365      23350    sleeping secs: 1
7       soctcppoll        7soc*    07/18 08:35:40        0.1150          1    running
8       soctcpio          8soc*    07/18 08:35:40        0.0037          1    running
9       soctcplst         1cpu*    07/18 11:47:24        0.1106         10    sleeping forever
10      soctcplst         1cpu*    07/18 08:35:40        0.0103          6    sleeping forever
11      flush_sub(0)      1cpu*    07/18 15:02:43        0.0403      23252    sleeping secs: 1
12      flush_sub(1)      1cpu*    07/18 15:02:43        0.0423      23169    sleeping secs: 1
13      flush_sub(2)      1cpu*    07/18 15:02:43        0.0470      23169    sleeping secs: 1
14      flush_sub(3)      1cpu*    07/18 15:02:43        0.0407      23169    sleeping secs: 1
15      flush_sub(4)      1cpu*    07/18 15:02:43        0.0307      23169    sleeping secs: 1
16      flush_sub(5)      1cpu*    07/18 15:02:43        0.0323      23169    sleeping secs: 1
17      flush_sub(6)      1cpu*    07/18 15:02:43        0.0299      23169    sleeping secs: 1
18      flush_sub(7)      1cpu*    07/18 15:02:43        0.0314      23169    sleeping secs: 1
19      kaio              1cpu*    07/18 14:56:42        1.4560    2375587    IO Idle
20      aslogflush        1cpu*    07/18 15:02:43        0.0657      23166    sleeping secs: 1
21      btscanner_0       1cpu*    07/18 15:00:53        0.0484        784    sleeping secs: 61
37      onmode_mon        1cpu*    07/18 15:02:43        0.3467      23165    sleeping secs: 1
43      dbScheduler       1cpu*    07/18 14:58:14        1.6613        320    sleeping secs: 31
44      dbWorker1         1cpu*    07/18 13:48:10        0.4264        399    sleeping forever
45      dbWorker2         1cpu*    07/18 14:48:11        1.9346       2936    sleeping forever
94      bf_priosweep()    1cpu*    07/18 15:01:42        0.0431         77    cond wait  bp_cond
```

Related reference

onstat -g cpu: Print runtime statistics

Monitor session resources with onstat -g ses output

Use the onstat -g ses command to monitor the resources allocated for and used by a session, in particular, a session that is running a decision-support query. The onstat -g ses command also shows information on recently terminated sessions.

For example, in Figure 81: onstat -g ses output on page 435, session number `49` is running five threads for a decision-support query.

Figure 81. onstat -g ses output

```
session                                  #RSAM    total      used
id       user     tty      pid   hostname threads  memory     memory
57       informix -        0     -        0        8192       5908
56       user_3   ttyp3    2318  host_10  1        65536      62404
55       user_3   ttyp3    2316  host_10  1        65536      62416
54       user_3   ttyp3    2320  host_10  1        65536      62416
53       user_3   ttyp3    2317  host_10  1        65536      62416
52       user_3   ttyp3    2319  host_10  1        65536      62416
51       user_3   ttyp3    2321  host_10  1        65536      62416
49       user_1   ttyp2    2308  host_10  5        188416     178936
2        informix -        0     -        0        8192       6780
1        informix -        0     -        0        8192       4796
```

```
session                                  #RSAM    total      used
id       user     tty      pid   hostname threads  memory     memory
57       informix -        0     -        0        8192       5908
56       user_3   ttyp3    2318  host_1   1        65536      62404
55       user_3   ttyp3    2316  host_1   1        65536      62416
54       user_3   ttyp3    2320  host_1   1        65536      62416
53       user_3   ttyp3    2317  host_1   1        65536      62416
52       user_3   ttyp3    2319  host_1   1        65536      62416
51       user_3   ttyp3    2321  host_1   1        65536      62416
49       user_1   ttyp2    2308  host_1   5        188416     178936
2        informix -        0     -        0        8192       6780
1        informix -        0     -        0        8192       4796

Last 20 Sessions Terminated

Ses ID  Username  Hostname  PID    Time              Reason
36      user_1    host_1    2122   01/19/2015.15:20  session limit txn time (60s)
40      user_1    host_1    2134   01/19/2015.15:14  session limit memory (5124 KB)
47      user_1    host_1    2140   01/19/2015.15:04  session limit logspace (10242 KB)
50      user_1    host_1    2145   01/19/2015.15:02  session limit txn time (39548 KB)
```

Related reference

onstat -g ses command: Print session-related information

Monitor session memory with onstat -g mem and onstat -g stm output

Use the onstat -g mem and onstat -g stm commands to obtain information about the memory used for each session.

You can determine which session to focus on by the **used memory** column of the onstat -g ses output.

Figure 82: onstat -g mem and onstat -g stm to determine session memory on page 436 shows sample onstat -g ses output and some of the onstat -g mem and onstat -g stm output for Session 16.

- The output of the onstat -g mem command shows the total amount of memory used by each session.

  The **totalsize** column of the **onstat -g mem 16** output shows the total amount of memory allocated to the session.
- The output of the onstat -g stm command shows the portion of the total memory allocated to the current prepared SQL statement.

  The **heapsz** column of the **onstat -g stm 16** output in the following figure shows the amount of memory allocated for the current prepared SQL statement.

Figure 82. onstat -g mem and onstat -g stm to determine session memory

```
onstat -g ses

session                                     #RSAM    total      used
id       user     tty      pid      hostname threads  memory     memory
18       informix -        0        -        0        12288      8928
17       informix 12       28826    lyceum   1        45056      33752
16       virginia 6        28743    lyceum   1        90112      79504
14       virginia 7        28734    lyceum   1        45056      33096
3        informix -        0        -        0        12288      10168
2        informix -        0        -        0        12288      8928



onstat -g mem 16

Pool Summary:
name         class addr     totalsize freesize #allocfrag #freefrag
16           V     a9ea020  90112     10608    159        5
...


onstat -g stm 16

session   16 ------------------------------------------------------------------
 sdblock  heapsz  statement ('*' = Open cursor)
 aa0d018   10056 *SELECT C.customer_num, O.order_num
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
AND O.order_num = I.order_num
```

Related information

onstat -g lap command: Print light appends status information on page

onstat -g mem command: Print pool memory statistics on page

Monitor sessions and threads with ON-Monitor (UNIX™)

On (UNIX™), you can use ON-Monitor to view information about sessions and user threads. This information is a subset of the information that the onstat -u command displays.

Choose **User** from the Status menu. The following information appears:

- The session ID
- The user ID
- The number of locks that the thread is holding
- The number of read calls and write calls that the thread has executed
- Flags that indicate the present state of the thread (for example, waiting for a buffer or waiting for a checkpoint), whether the thread is the primary thread for a session, and what type of thread it is (for example, user thread, daemon thread, and so on)

The following figure shows sample output.

Figure 83. Output from the User option of the ON-Monitor Status menu

```
                          USER THREAD INFORMATION

                        Locks       Disk       Disk       User thread
 Session   User         Held        Reads      Writes     Status

      0    informix        0          96          2       ------D
      0    informix        0           0          0       ------F
      0    informix        0           0          0       -------
     15    informix        0           0          0       Y-----M
      0    informix        0           0          0       ------D
     17    chrisw          1           3         34       Y------
```

Monitor sessions and threads with SMI tables

You can use the **syssessions** and the **syssesprof** system-monitoring interface (SMI) tables to obtain information about sessions and threads.

Query the **syssessions** table to obtain the following information.

Column

Description

sid

Session ID

username

Name (login ID) of the user

uid

User ID

pid

Process ID

connected

Time that the session started

feprogram

Absolute path of the executable program or application

In addition, some columns contain flags that show the following information;

- Whether the *primary* thread of the session is waiting for a latch, lock, log buffer, or transaction
- If the thread is in a critical section.

Important: The information in the **syssessions** table is organized by session, and the information in the **onstat -u** output is organized by thread. Also, unlike the **onstat -u** output, the **syssessions** table does not include information about daemon threads, only user threads.

Query the **syssesprof** table to obtain a profile of the activity of a session. This table contains a row for each session with columns that store statistics on session activity (for example, number of locks held, number of row writes, number of commits, number of deletes, and so on).

For a complete list of the **syssessions** columns and descriptions of **syssesprof** columns, see the chapter on the **sysmaster** database in the *HCL OneDB™ Administrator's Reference*.

Monitor transactions

You can monitor transactions to track open transactions and the locks that those transactions hold. You can use several **onstat** utility options to view transaction, lock, and session statistics.

ISA uses information that the following onstat command-line options generate to display session information, as the following table shows. Click the **Refresh** button to rerun the onstat command and display fresh information.

The following onstat command-line options display session information.

To monitor	Displays the output of	See
Transaction statistics	onstat -x	Display information about transactions on page 439

To monitor	Displays the output of	See
User session statistics	onstat -u	Display statistics on user sessions on page 441
Lock statistics	onstat -k	Display information about transaction locks on page 440
Sessions running SQL statements	onstat -g sql *session-id*	Display statistics on sessions executing SQL statements on page 442

Display information about transactions

The output of the onstat -x command contains information that you can use to monitor transactions.

The onstat -x output contains the following information for each open transaction:

- The address of the transaction structure in shared memory
- Flags that indicate the following information:
  - The present state of the transaction (user thread attached, suspended, waiting for a rollback)
  - The mode in which the transaction is running (loosely coupled or tight coupled)
  - The stage that the transaction is in (BEGIN WORK, prepared to commit, committing or committed, rolling back)
  - The nature of the transaction (global transaction, coordinator, subordinate, both coordinator and subordinate)
- The thread that owns the transaction
- The number of locks that the transaction holds
- The logical-log file in which the BEGIN WORK record was logged
- The current logical-log id and position
- The isolation level
- The number of attempts to start a recovery thread
- The coordinator for the transaction (if the subordinate is executing the transaction)
- The maximum number of concurrent transactions since you last started the database server

The **onstat** utility is especially useful for monitoring global transactions. For example, you can determine whether a transaction is executing in loosely coupled or tightly coupled mode. These transaction modes have the following characteristics:

- Loosely coupled mode

  Each branch in a global transaction has a separate transaction ID (XID). This mode is the default.

  - The different database servers coordinate transactions, but do not share resources. No two transaction branches, even if they access the same database, can share locks.
  - The records from all branches of a global transaction display as separate transactions in the logical log.
- Tightly coupled mode

In a single global transaction, all branches that access the same database share the same transaction ID (XID). This mode only occurs with the Microsoft™ Transaction Server (MTS) transaction manager.

- The different database servers coordinate transactions and share resources such as locks and log records. The branches with the same XID share locks and can never wait on another branch with the same XID because only one branch is active at one time.
- Log records for branches with the same XID appear under the same transaction in the logical log.

Figure 84: onstat -x output on page 440 shows sample output from onstat -x. The last transaction listed is a global transaction, as the `G` value in the fifth position of the **flags** column indicates. The `T` value in the second position of the **flags** column indicates that the transaction is running in tightly coupled mode.

Figure 84. onstat -x output

```
Transactions
address flags userthread locks beginlg curlog logposit isol    retrys coord
ca0a018 A---- c9da018    0     0       5      0x18484c COMMIT  0
ca0a1e4 A---- c9da614    0     0       0      0x0      COMMIT  0
ca0a3b0 A---- c9dac10    0     0       0      0x0      COMMIT  0
ca0a57c A---- c9db20c    0     0       0      0x0      COMMIT  0
ca0a748 A---- c9db808    0     0       0      0x0      COMMIT  0
ca0a914 A---- c9dbe04    0     0       0      0x0      COMMIT  0
ca0aae0 A---- c9dcff8    1     0       0      0x0      COMMIT  0
ca0acac A---- c9dc9fc    1     0       0      0x0      COMMIT  0
ca0ae78 A---- c9dc400    1     0       0      0x0      COMMIT  0
ca0b044 AT--G c9dc9fc    0     0       0      0x0      COMMIT  0
10 active, 128 total, 10 maximum concurrent
```

The output in Figure 84: onstat -x output on page 440 shows that this transaction branch is holding 13 locks. When a transaction runs in tightly coupled mode, the branches of this transaction share locks.

Display information about transaction locks

The output of the onstat -k command contains details on the locks that a transaction holds.

To find the relevant locks, match the address in the **userthread** column in onstat -x output to the address in the **owner** column of onstat -k output.

Figure 85: onstat -k and onstat -x output on page 441 shows sample output from onstat -x and the corresponding onstat -k command. The `a335898` value in the **userthread** column in the onstat -x output matches the value in the **owner** column of the two lines of onstat -k output.

Figure 85. onstat -k and onstat -x output

```
onstat -x

Transactions
address  flags userthread locks   beginlg curlog  logposit   isol    retrys coord
a366018  A---- a334018    0      0       1       0x22b048  COMMIT  0
a3661f8  A---- a334638    0      0       0       0x0       COMMIT  0
a3663d8  A---- a334c58    0      0       0       0x0       COMMIT  0
a3665b8  A---- a335278    0      0       0       0x0       COMMIT  0
a366798  A---- a335898    2      0       0       0x0       COMMIT  0
a366d38  A---- a336af8    0      0       0       0x0       COMMIT  0
 6 active, 128 total, 9 maximum concurrent

onstat -k

Locks
address  wtlist   owner    lklist   type     tblsnum  rowid    key#/bsiz
a09185c  0        a335898  0        HDR+S    100002   20a         0
a0918b0  0        a335898  a09185c  HDR+S    100002   204         0
 2 active, 2000 total, 2048 hash buckets, 0 lock table overflows
```

In the example in Figure 85: onstat -k and onstat -x output on page 441, a user is selecting a row from two tables. The user holds the following locks:

- A shared lock on one database
- A shared lock on another database

Display statistics on user sessions

The output of the onstat -u command contains statistics on user sessions.

You can find the session-id of the transaction by matching the address in the **userthread** column of the onstat -x output with the **address** column in the onstat -u output. The **sessid** column of the same line in the onstat -u output provides the session id.

For example, Figure 86: Obtaining session-id for userthread in onstat -x on page 442 shows the address `a335898` in the **userthread** column of the onstat -x output. The output line in onstat -u with the same address shows the session id `15` in the **sessid** column.

Figure 86. Obtaining session-id for userthread in onstat -x

```
onstat -x

Transactions
address  flags userthread locks   beginlg curlog  logposit   isol    retrys coord
a366018  A---- a334018    0      0       1       0x22b048  COMMIT  0
a3661f8  A---- a334638    0      0       0       0x0       COMMIT  0
a3663d8  A---- a334c58    0      0       0       0x0       COMMIT  0
a3665b8  A---- a335278    0      0       0       0x0       COMMIT  0
a366798  A---- a335898    2      0       0       0x0       COMMIT  0
a366d38  A---- a336af8    0      0       0       0x0       COMMIT  0
 6 active, 128 total, 9 maximum concurrent


onstat -u

address  flags    sessid  user     tty     wait     tout locks nreads   nwrites
a334018  ---P--D 1        informix -        0        0    0     20       6
a334638  ---P--F 0        informix -        0        0    0     0        1
a334c58  ---P--- 5        informix -        0        0    0     0        0
a335278  ---P--B 6        informix -        0        0    0     0        0
a335898  Y--P--- 15       informix 1        a843d70  0    2     64       0
a336af8  ---P--D 11       informix -        0        0    0     0        0
 6 active, 128 total, 17 maximum concurrent
```

For a transaction executing in loosely coupled mode, the first position of the **flags** column in theonstat -u output might display a value of `T`. This `T` value indicates that one branch within a global transaction is waiting for another branch to complete. This situation could occur if two different branches in a global transaction, both using the same database, tried to work on the same global transaction simultaneously.

For a transaction executing in tightly coupled mode, this `T` value does not occur because the database server shares one transaction structure for all branches that access the same database in the global transaction. Only one branch is attached and active at one time and does not wait for locks because the transaction owns all the locks held by the different branches.

Display statistics on sessions executing SQL statements

The output of the onstat -g sql command contains statistics on the SQL statements executed by the session

To obtain information about the last SQL statement that each session executed, issue the onstat -g sql command with the appropriate session ID.

Figure 87: onstat -g sql output on page 443 shows sample output for this option using the same session ID obtained from the onstat -u sample in Figure 86: Obtaining session-id for userthread in onstat -x on page 442.

Figure 87. onstat -g sql output

```
onstat -g sql 15

Sess  SQL            Current            Iso Lock       SQL  ISAM F.E.
Id    Stmt type      Database           Lvl Mode       ERR  ERR  Vers
15    SELECT         vjp_stores         CR  Not Wait   0    0    9.03

Current statement name : slctcur

Current SQL statement :
  select l.customer_num, l.lname, l.company,   l.phone, r.call_dtime,
    r.call_descr from customer l, vjp_stores@gilroy:cust_calls r where
    l.customer_num = r.customer_num

Last parsed SQL statement :
  select l.customer_num, l.lname, l.company,   l.phone, r.call_dtime,
    r.call_descr from customer l, vjp_stores@gilroy:cust_calls r where
    l.customer_num = r.customer_num
```

The onperf utility on UNIX™

The **onperf** utility is a windowing environment that you can use to monitor the database server performance. The **onperf** utility monitors the database server running on the UNIX™ operating system.

Related reference

Database server tools on page 22

Overview of the onperf utility

The **onperf** utility is a graphical tool that you can use for displaying most of the same database server metrics that you can view on **onstat** utility reports.

The **onperf** utility provides the following advantages over the **onstat** utility:

- Displays metric values graphically in real time
- Allows you to choose which metrics to monitor
- Allows you to scroll back to previous metric values to analyze a trend
- Allows you to save performance data to a file for review at a later time

You cannot use the **onperf** utility on High-Availability Data Replication (HDR) secondary servers, remote standalone (RS) secondary servers, or shared disk (SD) secondary servers.

Basic onperf utility functions

The **onperf** utility displays the values of the database server metrics in a tool window and saves the database server metric values to a file. You can review the contents of this file.

Display metric values

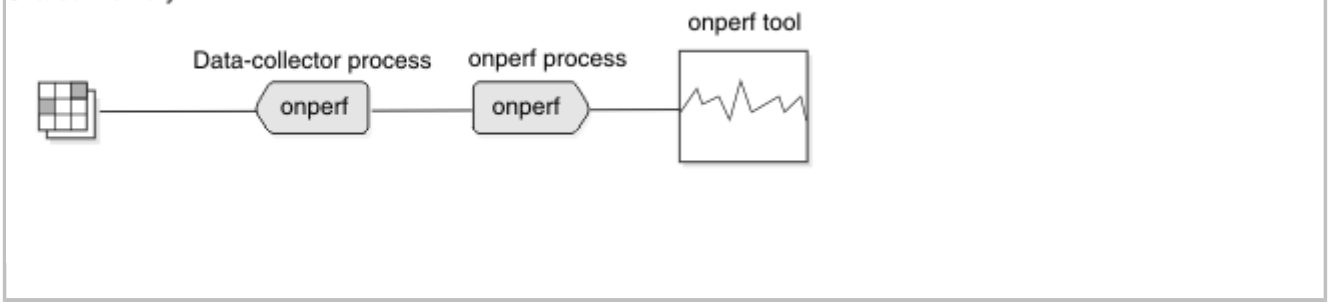
The **onperf** utility displays database server metrics obtained from shared memory.

When **onperf** starts, it activates the following processes:

- **The onperf process.** This process controls the display of **onperf** tools.
- **The data-collector process.** This process attaches to shared memory and passes performance information to the **onperf** process for display in an **onperf** tool.

An **onperf** tool is a Motif window that an **onperf** process manages, as Figure 88: Data flow from shared memory to an onperf tool window on page 444 shows.

Figure 88. Data flow from shared memory to an onperf tool window

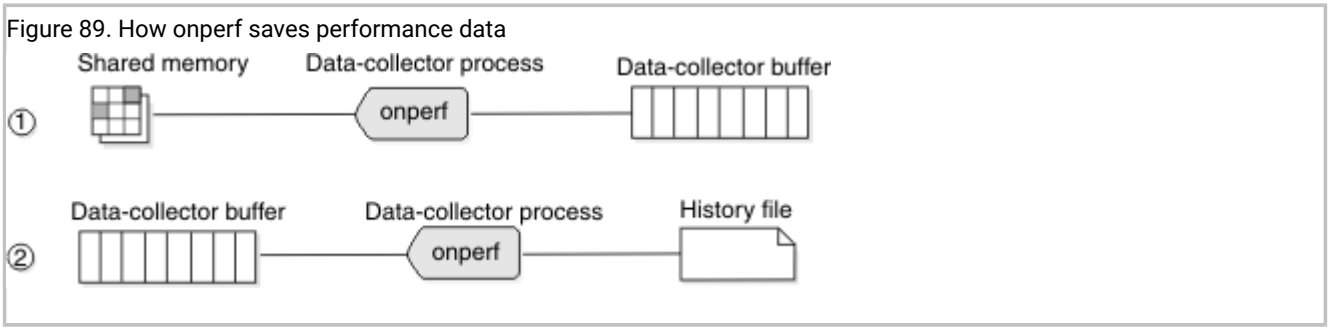


Save metric values to a file

The **onperf** utility saves collected metrics in a history file.

The **onperf** utility allows designated metrics to be continually buffered. The data collector writes these metrics to a circular buffer called the *data-collector buffer*. When the buffer becomes full, the oldest values are overwritten as the data collector continues to add data. The current contents of the data-collector buffer are saved to a history file, as Figure 89: How onperf saves performance data on page 444 illustrates.

Figure 89. How onperf saves performance data



The **onperf** utility uses either a binary format or an ASCII representation for data in the history file. The binary format is host-dependent and allows data to be written quickly. The ASCII format is portable across platforms.

You have control over the set of metrics stored in the data-collector buffer and the number of samples. You could buffer all metrics; however, this action might consume more memory than is feasible. A single metric measurement requires 8 bytes of memory. For example, if the sampling frequency is one sample per second, then to buffer 200 metrics for 3,600 samples

requires approximately 5.5 megabytes of memory. If this process represents too much memory, you must reduce the depth of the data-collector buffer, the sampling frequency, or the number of buffered metrics.

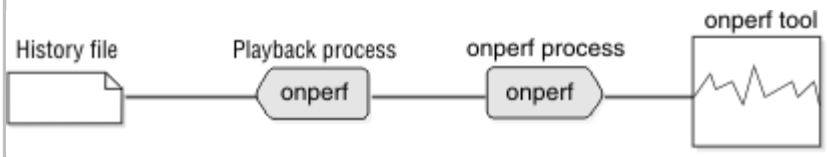
To configure the buffer depth or the sampling frequency, you can use the Configuration dialog box. For more information about the Configuration dialog box, see The graph-tool Configure menu and the Configuration dialog box on page 451.

Review metric measurements

You can review the contents of a history file in a tool window. When you request a tool to display a history file, the **onperf** utility starts a playback process that reads the data from disk and sends the data to the tool for display.

The playback process is similar to the data-collector process mentioned under Save metric values to a file on page 444. However, instead of reading data from shared memory, the playback process reads measurements from a history file. Figure 90: Flow of data from a history file to an onperf tool window on page 445 shows the playback process.

Figure 90. Flow of data from a history file to an onperf tool window



onperf utility tools

The **onperf** utility provides Motif windows, called *tools*, which display metric values.

Table 22. onperf utility tools

Tool	Description
Graph tool	This tool allows you to monitor general performance activity. You can use this tool to display any combination of metrics that **onperf** supports and to display the contents of a history file. For more information, see Graph tool on page 447.
Query-tree tool	This tool displays the progress of individual queries. For more information, see Query-tree tool on page 454.
Status tool	This tool displays status information about the database server and allows you to save the data that is currently held in the data-collector buffer to a file. For more information, see Status tool on page 454.
Activity tools	These tools display specific database server activities. Activity tools include disk, session, disk-capacity, physical-processor, and virtual-processor tools. The physical-processor and virtual-processor tools, respectively, display information about all CPUs and VPs. The other activity tools each display the top 10 instances of a resource ranked by a suitable activity measurement. For more information, see Activity tools on page 455.

Requirements for running the onperf utility

The computer that is running the **onperf** utility must support the X terminal and the **mwm** window manager.

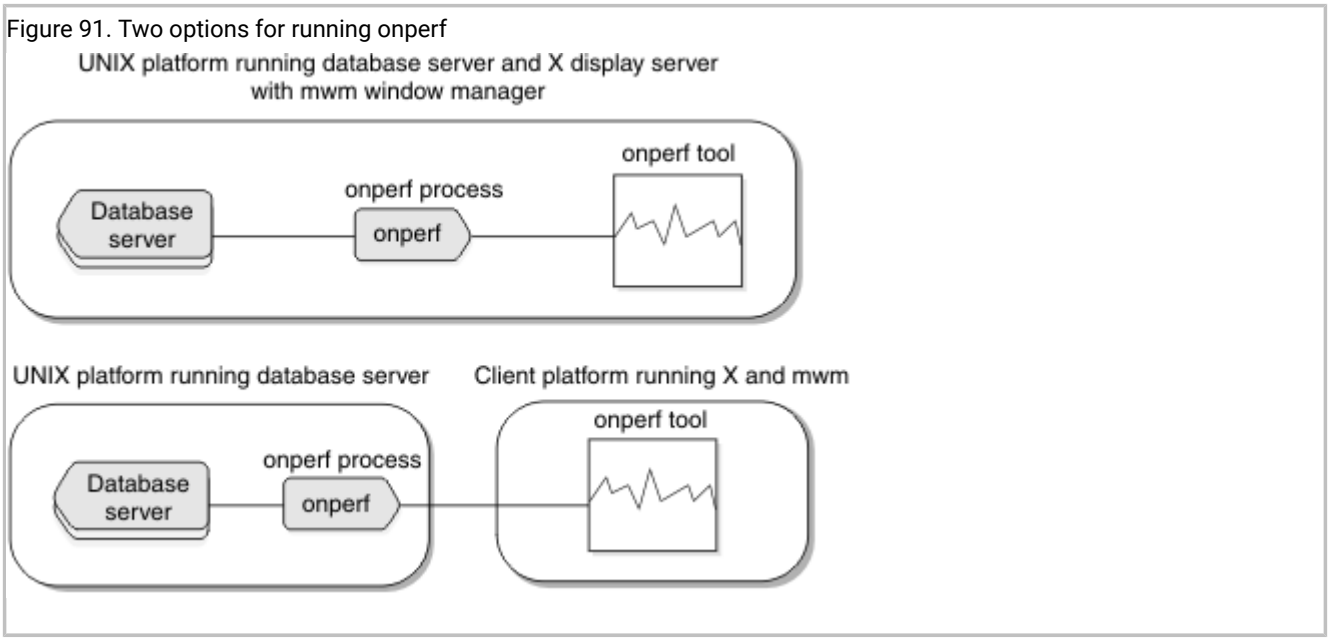
When you install the database server, the following executable files are written to the **$INFORMIXDIR/bin** directory:

- **onperf**
- **onedcu**
- **onedpu**
- **xtree**

In addition, the **onperf.hlp** online help file is placed in the directory **$INFORMIXDIR/hhelp**.

When the database server is installed and running in online mode, you can bring up **onperf** tools either on the computer that is running the database server or on a remote computer or terminal that can communicate with your database server instance. Figure 91: Two options for running onperf on page 446 illustrates both possibilities. In either case, the computer that is running the **onperf** tools must support the X terminal and the **mwm** window manager.

Figure 91. Two options for running onperf



Starting the onperf utility and exiting from it

Before you start the **onperf** utility, set the **DISPLAY** and **LD_LIBRARY_PATH** environment variables.

Prerequisite: Set the **DISPLAY** environment variable as follows:

```
C shell         setenv DISPLAY displayname0:0 #

Bourne shell    DISPLAY=displayname0:0 #
```

In these commands, *displayname* is the name of the computer or X terminal where the **onperf** window should appear.

Set the **LD_LIBRARY_PATH** environment variable to the appropriate value for the Motif libraries on the computer that is running **onperf**.

With the environment properly set up, you can enter **onperf** to bring up a graph-tool window, as described in The onperf user interface on page 447.

You can monitor multiple database server instances from the same Motif client by invoking **onperf** for each database server, as the following example shows:

```
INFORMIXSERVER=instance1 ; export INFORMIXSERVER; onperf
INFORMIXSERVER=instance2 ; export INFORMIXSERVER; onperf
...
```

Exiting from the onperf Utility

To exit from the **onperf** utility, use the **Close** option to close each tool window, use the **Exit** option of a tool, or choose **Window Manager > Close**.

The onperf user interface

When you invoke the **onperf** utility, it displays an initial graph-tool window. From this graph-tool window, you can display additional graph-tool windows as well as the query-tree, data-collector, and activity tools.

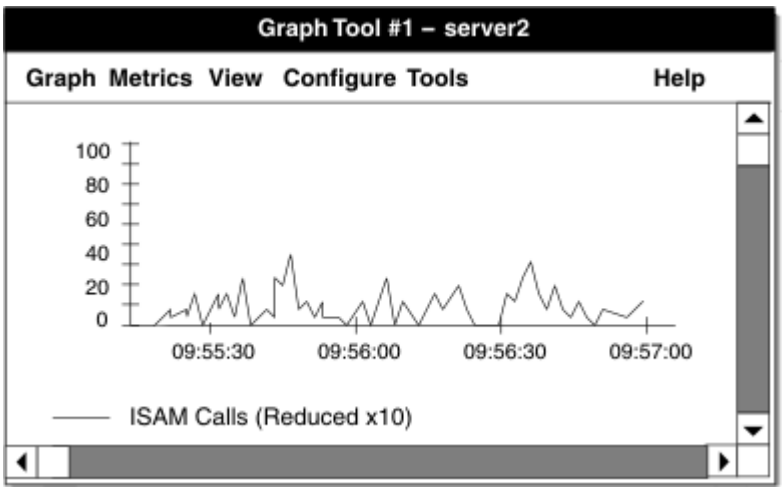
The graph-tool windows have no hierarchy; you can create and close these windows in any order.

Graph tool

The graph tool is the principal **onperf** interface. Use the graph tool to display any set of database server metrics that the **onperf** data collector obtains from shared memory.

The Figure 92: Graph-Tool window on page 447 shows a diagram of a graph tool that displays a graph of metrics for ISAM calls.

Figure 92. Graph-Tool window



You cannot bring up a graph-tool window from a query-tree tool, a status tool, or one of the activity tools.

Graph-tool title bar

When you invoke **onperf**, the initial graph-tool window displays **serverName**, the database server that the **INFORMIXSERVER** environment variable specifies, in the title bar. The data comes from the shared memory of the indicated database server instance.

If the configuration of an initial graph-tool has not yet been saved or loaded from disk, **onperf** does not display the name of a configuration file in the title bar.

If you open a historical data file, for example named **caselog.23April.2PM**, in this graph-tool window, the title bar displays **caselog.23.April.23.April.2PM**.

Graph-tool graph menu

The **Graph** menu contains options for creating, opening, saving the contents of, printing the contents of, annotating, and closing a graph tool.

The **Graph** menu provides the following options.

Option

Use

New

Creates a new graph tool. All graph tools that you create using this option share the same data-collector and **onperf** processes. To create new graph tools, use this option rather than invoke **onperf** multiple times.

Open History File

Loads a previously saved file of historical data into the graph tool for viewing. If the file does not exist, **onperf** prompts you for one. When you select a file, **onperf** starts a playback process to view the file.

Save History File

Saves the contents of the data-collector buffer to either an ASCII or a binary file, as specified in the Configuration dialog box.

Save History File As

Specifies the filename in which to save the contents of the data-collector buffer.

Annotate

Brings up a dialog box in which you can enter a header label and a footer label. Each label is optional. The labels are displayed on the graph. When you save the graph configuration, **onperf** includes these labels in the saved configuration file.

Print

Brings up a dialog box that allows you to select a destination file. You cannot send the contents of the graph tool directly to a printer; you must use this option to specify a file and subsequently send the file to a printer.

Close

Closes the tool. When a tool is the last remaining tool of the **onperf** session, this menu item behaves in the same way as the **Exit** option.

Exit

Exits **onperf**.

Important: To save your current configuration before you load a new configuration from a file, you must choose **Configure > Save Configuration** or **Configure > Save Configuration As**.

Graph-tool metrics menu

The **Metrics** menu contains options for choosing the class of metrics to display in the graph tool.

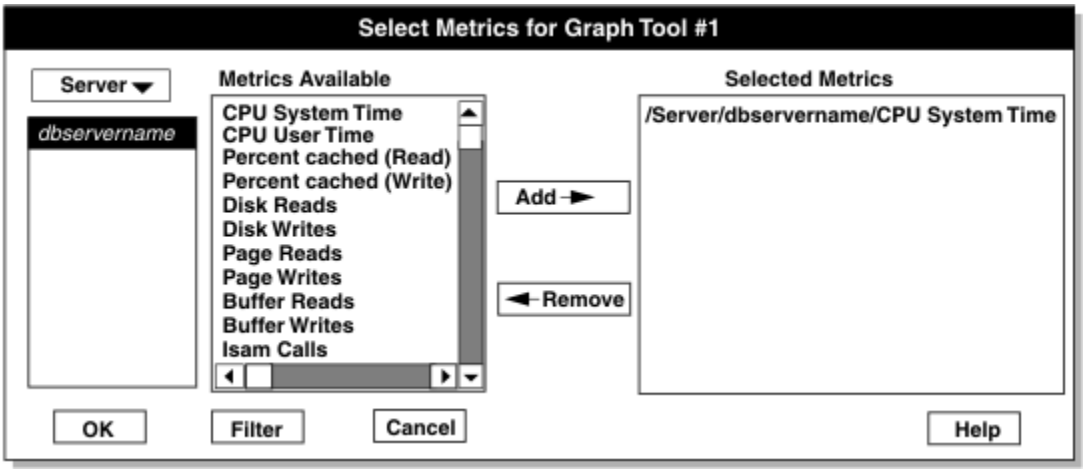
Metrics are organized by *class* and *scope*. When you select a metric for the graph tool to display, you must specify the metric class, the metric scope, and the name of the metric.

The *metric class* is the generic database server component or activity that the metric monitors. The *metric scope* depends on the metric class. In some cases, the metric scope indicates a particular component or activity. In other cases, the scope indicates all activities of a given type across an instance of the database server.

The **Metrics** menu has a separate option for each class of metrics. For more information about metrics, see Why you might want to use onperf on page 455.

When you choose a class, such as **Server**, you see a dialog box like the one in Figure 93: The Select Metrics dialog box on page 449.

Figure 93. The Select Metrics dialog box



The Select Metrics dialog box contains three list boxes. The list box on the left displays the valid scope levels for the selected metrics class. For example, when the scope is set to **Server**, the list box displays the dbservername of the database server instance that is being monitored. When you select a scope from this list, **onperf** displays the individual metrics that

are available within that scope in the middle list box. You can select one or more individual metrics from this list and add them to the display by clicking **Add**. To remove them from the display, click **Remove**.

Tip: You can display metrics from more than one class in a single graph-tool window. For example, you might first select **ISAM Calls**, **Opens**, and **Starts** from the **Server** class. When you choose the **Option** menu in the same dialog box, you can select another metric class without exiting the dialog box. For example, you might select the Chunks metric class and add the **Operations**, **Reads**, and **Writes** metrics to the display.

The **Filter** button in the dialog box brings up an additional dialog box in which you can filter long text strings shown in the Metrics dialog box. The Filter dialog box also lets you select tables or fragments for which metrics are not currently displayed.

After you make your selections, you can click OK to proceed, or Cancel if you choose not to proceed.

Graph-tool view menu

The **View** menu contains options for changing how the graph tool appears.

The **View** menu provides the following options.

Line

Changes the graph tool to the line format. Line format includes horizontal and vertical scroll bars. The vertical scroll bar adjusts the scale of the horizontal time axis. When you raise this bar, **onperf** reduces the scale and vice versa. The horizontal scroll bar allows you to adjust your view along the horizontal time axis.

To change the color and width of the lines in the line format, click the legend in the graph tool. When you do, **onperf** generates a Customize Metric dialog box that provides a choice of line color and width.

Horizontal Bar Graph

Changes the graph tool to the horizontal bar format.

Vertical Bar Graph

Changes the graph tool to the vertical bar format.

Pie

Changes the graph tool to the pie-chart format. To display a pie chart, you must select at least two metrics.

Quick Rescale Axis

Rescales the axis to the largest point that is currently visible on the graph. This button turns off automatic rescaling.

Configure Axis

Displays the Axis Configuration dialog box. Use this dialog box to select a fixed value for the y-axis on the graph or select automatic axis scaling.

The graph-tool Configure menu and the Configuration dialog box

The **Configure** menu contains options of opening, editing, and saving **onperf** configuration information.

The **Configure** menu provides the following options.

Edit Configuration

Brings up the Configuration dialog box, which allows you to change the settings for the current data-collector buffer, graph-tool display options, and data-collector options. The Configuration dialog box appears in Figure 94: The Configuration dialog box on page 451.

Open Configuration

Restarts **onperf** with the settings that are stored in the configuration file. Unsaved data in the data-collector buffer is lost. If no configuration file is specified and the default does not exist, the following error message appears:

```
Open file filename failed.
```

If the specified configuration file does not exist, **onperf** prompts for one.

Save Configuration

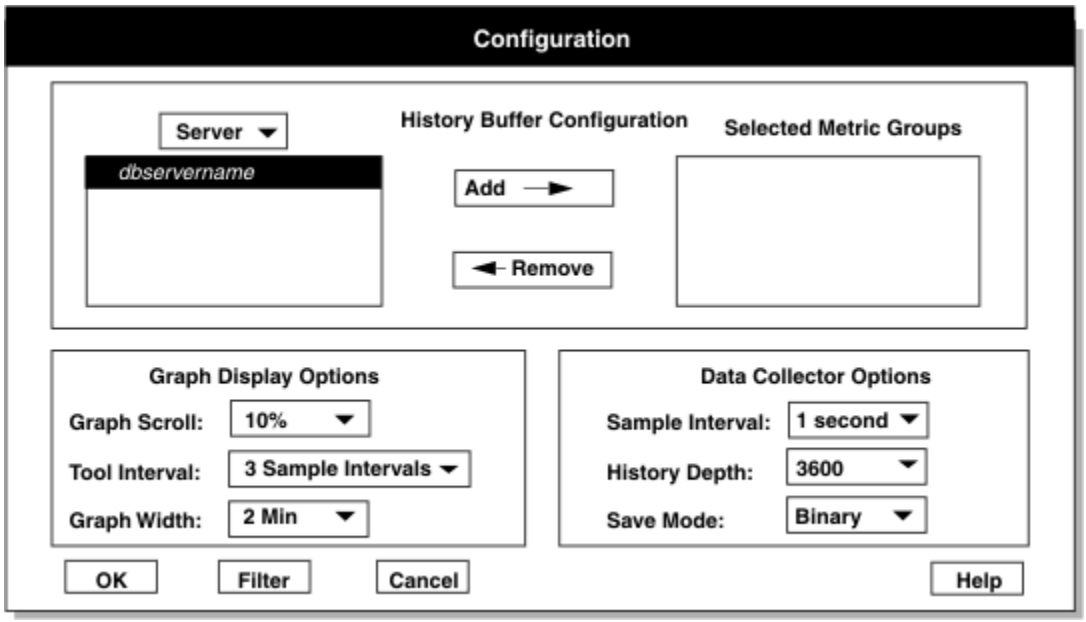
Saves the current configuration to a file. If no configuration file is currently specified, **onperf** prompts for one.

Save Configuration As

Saves a configuration file. This option always prompts for a filename.

To configure data-collector options, graph-display options, and metrics about which to collect data, choose the **Edit Configuration** option to bring up the Configuration dialog box.

Figure 94. The Configuration dialog box



The Configuration dialog box provides the following options for configuring display.

Option

Use

History Buffer Configuration

Allows you to select a metric class and metric scope to include in the data-collector buffer. The data collector gathers information about all metrics that belong to the indicated class and scope.

Graph Display Options

Allows you to adjust the size of the graph portion that scrolls off to the left when the display reaches the right edge, the initial time interval that the graph is to span, and the frequency with which the display is updated.

Data Collector Options

Controls the collection of data. The sample interval indicates the amount of time to wait between recorded samples. The history depth indicates the number of samples to retain in the data-collector buffer. The save mode indicates the data-collector data should be saved in binary or ASCII format.

Graph-tool Tools menu

The **Tools** menu contains options that start additional **onperf** tools.

This menu provides the following options.

Query Tree

Starts a query-tree tool. For more information, see Query-tree tool on page 454.

Status

Starts a status tool. For more information, see Status tool on page 454.

Disk Activity

Starts a disk-activity tool. For more information, see Activity tools on page 455.

Session Activity

Starts a session-activity tool. For more information, see Activity tools on page 455.

Disk Capacity

Starts a disk-capacity tool. For more information, see Activity tools on page 455.

Physical Processor Activity

Starts a physical-processor tool. For more information, see Activity tools on page 455.

Virtual Processor Activity

Starts a virtual-processor tool. For more information, see Activity tools on page 455.

Changing the scale of metrics

When **onperf** displays metrics, it automatically adjusts the scale of the y-axis to accommodate the largest value. You can use the Customize Metric dialog box to establish one for the current display.

For more information, see Graph-tool view menu on page 450.

Displaying recent-history values

When you use the **onperf** utility, you can scroll back over previous metric values that are displayed in a line graph. This is useful for analyzing recent trends.

The time interval to which you can scroll back is the *lesser* of the following intervals:

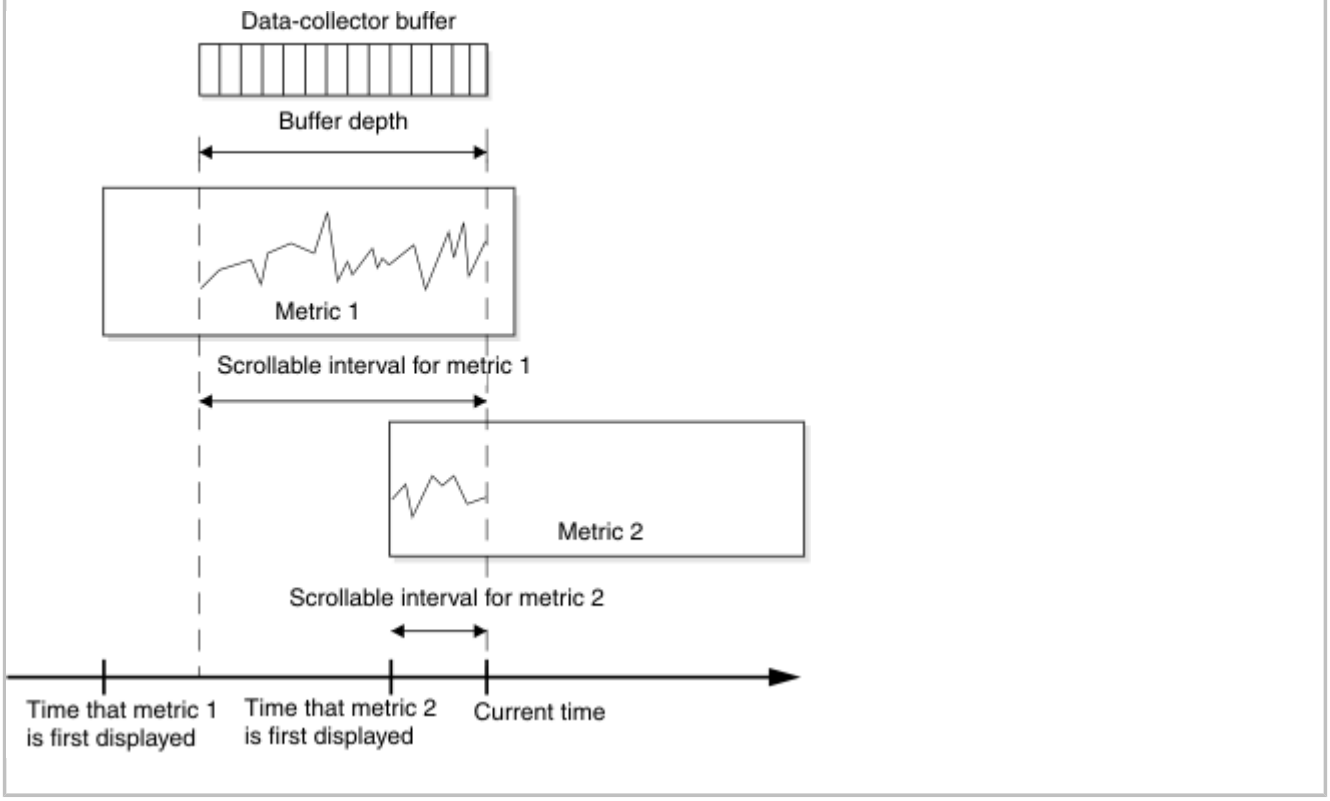
- The time interval over which the metric has been displayed
- The history interval that the graph-tool Configuration dialog box specifies

The length of time you can scroll back through cannot exceed the depth of the data-collector buffer.

For more information, see The graph-tool Configure menu and the Configuration dialog box on page 451.

Figure 95: Maximum scrollable intervals for metrics that span different time periods on page 453 illustrates the maximum scrollable intervals for metrics that span different time periods.

Figure 95. Maximum scrollable intervals for metrics that span different time periods



Query-tree tool

The query-tree tool contains options for monitoring the performance of individual queries.

The query-tree tool is a separate executable tool that does not use the data-collector process. You cannot save query-tree tool data to a file.

This tool includes a **Select Session** button and a **Quit** button. When you select a session that is running a query, the large detail window displays the SQL operators that constitute the execution plan for the query. The query-tree tool represents each SQL operator with a box. Each box includes a dial that indicates rows per second and a number that indicates input rows. In some cases, not all the SQL operators can be represented in the detail window. The smaller window shows the SQL operators as small icons.

The **Quit** button allows you to exit from the query-tree tool.

Status tool

The status tool enables you to select metrics to store in the data-collector buffer. In addition, you can use this tool to save the data currently held in the data-collector buffer to a file.

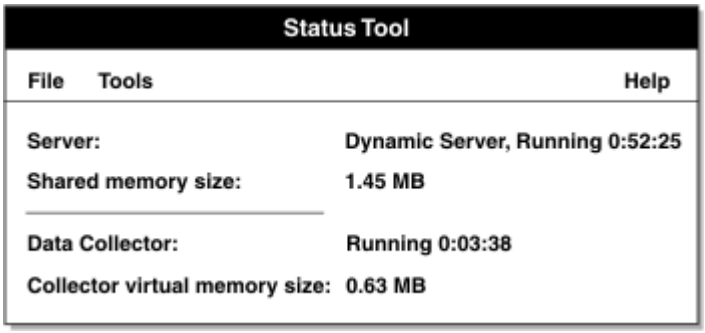
Figure 96: Status Tool window on page 454 shows a status tool.

The status tool displays:

- The length of time that the data collector has been running
- The size of the data-collector process area, called the *collector virtual memory size*

  When you select different metrics to store in the data-collector buffer, you see different values for the collector virtual memory size.

  Figure 96. Status Tool window



The status tool **File** menu provides the following options.

Option

Use

Close

This option closes the tool. When it is the last remaining tool of the **onperf** session, Close behaves in the same way as Exit.

Exit

This option exits **onperf**.

Activity tools

Activity tools are specialized forms of the graph tool that display instances of the specific activity, based on a ranking of the activity by some suitable metric.

You can choose from among the following activity tools:

- The disk-activity tool, which displays the top 10 activities ranked by total operations
- The session-activity tool, which displays the top 10 activities ranked by ISAM calls plus PDQ calls per second
- The disk-capacity tool, which displays the top 10 activities ranked by free space in megabytes
- The physical-processor-activity tool, which displays all processors ranked by nonidle CPU time
- The virtual-processor-activity tool, which displays all VPs ranked by VP user time plus VP system time

The activity tools use the bar-graph format. You cannot change the scale of an activity tool manually; **onperf** always sets this value automatically.

The **Graph** menu provides you with options for closing, printing, and exiting the activity tool.

Why you might want to use onperf

You can use the **onperf** utility for routine monitoring, diagnosing sudden performance loss, and diagnosing performance degradation.

The following sections provide suggestions for different ways to use the **onperf** utility.

Routine monitoring with onperf

You can use the **onperf** utility to facilitate routine monitoring. For example, you can display several metrics in a graph-tool window and run this tool throughout the day.

Displaying these metrics allows you to monitor database server performance visually at any time.

Diagnosing sudden performance loss

When you detect a sudden performance dip, you can use the **onperf** utility to examine the recent history of the database server metrics values to identify any trend.

The **onperf** utility allows you to scroll back over a time interval, as explained in Displaying recent-history values on page 453.

Diagnosing performance degradation

You can save the metrics that the **onperf** utility displays, so you can analyze it and compare it to other saved information. This can be useful when analyzing performance problems that gradually develop and might be difficult to diagnose.

For example, if you detect a degradation in database server response time, it might not be obvious from looking at the current metrics which value is responsible for the slowdown. The performance degradation might also be sufficiently gradual that you cannot detect a change by observing the recent history of metric values. To allow for comparisons over longer intervals, **onperf** allows you to save metric values to a file, as explained in Status tool on page 454.

onperf utility metrics

When you use the **onperf** utility, you can view various metric classes.

The following sections describe these metric classes. Each section indicates the scope levels available and describes the metrics within each class.

Database server performance depends on many factors, including the operating-system configuration, the database server configuration, and the workload. It is difficult to describe relationships between **onperf** metrics and specific performance characteristics.

The approach taken here is to describe each metric without speculating on what specific performance problems it might indicate. Through experimentation, you can determine which metrics best monitor performance for a specific database server instance.

Database server metrics

The **onperf** utility displays metrics for the named database server, rather than a component of the database server or disk space.

The **onperf** utility displays the following database server metrics.

Metric Name	Description
CPU System Time	System time, as defined by the platform vendor
CPU User Time	User time, as defined by the platform vendor
Percent Cached (Read)	Percentage of all read operations that are read from the buffer cache without requiring a disk read, calculated as follows: `100 * ((buffer_reads – disk_reads) / (buffer_reads))`
Percent Cached (Write)	Percentage of all write operations that are buffer writes, calculated as follows: `100 * ((buffer_writes – disk_writes) / (buffer_writes))`
Disk Reads	Total number of read operations from disk

Metric Name	Description
Disk Writes	Total number of write operations to disk
Page Reads	Number of pages read from disk
Page Writes	Number of pages transferred to disk
Buffer Reads	Number of reads from the buffer cache
Buffer Writes	Number of writes to the buffer cache
Calls	Number of calls received at the database server
Reads	Number of read calls received at the database server
Writes	Number of write calls received at the database server
Rewrites	Number of rewrite calls received at the database server
Deletes	Number of delete calls received at the database server
Commits	Number of commit calls received at the database server
Rollbacks	Number of rollback calls received at the database server
Table Overflows	Number of times that the tblspace table was unavailable (overflowed)
Lock Overflows	Number of times that the lock table was unavailable (overflowed)
User Overflows	Number of times that the user table was unavailable (overflowed)
Checkpoints	Number of checkpoints written since database server shared memory began
Buffer Waits	Number of times that a thread waited to access a buffer
Lock Waits	Number of times that a thread waited for a lock
Lock Requests	Number of times that a lock was requested
Deadlocks	Number of deadlocks detected
Deadlock Timeouts	Number of deadlock timeouts that occurred (Deadlock timeouts involve distributed transactions.)
Checkpoint Waits	Number of checkpoint waits; in other words, the number of times that threads have waited for a checkpoint to complete
Index to Data Pages Read-aheads	Number of read-ahead operations for index keys
Index Leaves Read-aheads	Number of read-ahead operations for index leaf nodes
Data-path-only Read-aheads	Number of read-ahead operations for data pages
Latch Requests	Number of latch requests
Network Reads	Number of ASF messages read

Metric Name	Description
Network Writes	Number of ASF messages written
Memory Allocated	Amount of database server virtual-address space in kilobytes
Memory Used	Amount of database server shared memory in kilobytes
Temp Space Used	Amount of shared memory allocated for temporary tables in kilobytes
PDQ Calls	The total number of parallel-processing actions that the database server performed
DSS Memory	Amount of memory currently in use for decision-support queries

Disk-chunk metrics

The **onperf** utility can display metrics for a specific disk chunk.

The disk-chunk metrics take the path name of a chunk as the metric scope.

Metric Name	Description
Disk Operations	Total number of I/O operations to or from the indicated chunk
Disk Reads	Total number of reads from the chunk
Disk Writes	Total number of writes to the chunk
Free Space (MB)	The amount of free space available in megabytes

Disk-spindle metrics

The **onperf** utility can display metrics for a disk spindle.

The disk-spindle metrics take the path name of a disk device or operation-system file as the metric scope.

Metric Name	Description
Disk Operations	Total number of I/O operations to or from the indicated disk or buffered operating-system file
Disk Reads	Total number of reads from the disk or operating-system file
Disk Writes	Total number of writes to the disk or operating-system file
Free Space	The amount of free space available in megabytes

Physical-processor metrics

The **onperf** utility can display CPU metrics.

The physical-processor metrics take either a physical-processor identifier (for example, `0` or `1`) or **Total** as the metric scope.

Metric Name	Description
Percent CPU System Time	CPU system time for the physical processors
Percent CPU User Time	CPU user time for the physical processors
Percent CPU Idle Time	CPU idle time for the physical processors
Percent CPU Time	The sum of CPU system time and CPU user time for the physical processors

Virtual-processor metrics

The **onperf** utility can display metrics for a virtual-processor class.

These metrics take a virtual-processor class as a metric scope (cpu, aio, kaio, and so on). Each metric value represents a sum across all instances of this virtual-processor class.

Metric Name	Description
User Time	Accumulated user time for a class
System Time	Accumulated system time for a class
Semaphore Operations	Total count of semaphore operations
Busy Waits	Number of times that virtual processors in class avoided a context switch by spinning in a loop before going to sleep
Spins	Number of times through the loop
I/O Operations	Number of I/O operations per second
I/O Reads	Number of read operations per second
I/O Writes	Number of write operations per second

Session metrics

The **onperf** utility can display metrics for an active session.

These metrics take the active session as the metric scope.

Metric Name	Description
Page Reads	Number of pages read from disk on behalf of a session
Page Writes	Number of pages written to disk on behalf of a session
Number of Threads	Number of threads currently running for the session
Lock Requests	Number of lock requests issued by the session
Lock Waits	Number of lock waits for session threads

Metric Name	Description
Deadlocks	Number of deadlocks involving threads that belong to the session
Deadlock timeouts	Number of deadlock timeouts involving threads that belong to the session
Log Records	Number of log records written by the session
ISAM Calls	Number of ISAM calls by session
ISAM Reads	Number of ISAM read calls by session
ISAM Writes	Number of ISAM write calls by session
ISAM Rewrites	Number of ISAM rewrite calls by session
ISAM Deletes	Number of ISAM delete calls by session
ISAM Commits	Number of ISAM commit calls by session
ISAM Rollbacks	Number of ISAM rollback calls by session
Long Transactions	Number of long transactions owned by session
Buffer Reads	Number of buffer reads performed by session
Buffer Writes	Number of buffer writes performed by session
Log Space Used	Amount of logical-log space used
Maximum Log Space Used	High-watermark of logical-log space used for this session
Sequential Scans	Number of sequential scans initiated by session
PDQ Calls	Number of parallel-processing actions performed for queries initiated by the session
Memory Allocated	Memory allocated for the session in kilobytes
Memory Used	Memory used by the session in kilobytes

Tblspace metrics

The **onperf** utility can display metrics for a particular tblspace.

A tblspace name is composed of the database name, a colon, and the table name (*database:table*).

For fragmented tables, the tblspace represents the sum of all fragments in a table. To obtain measurements for an individual fragment in a fragmented table, use the Fragment Metric class.

Metric Name	Description
Lock Requests	Total requests to lock tblspace
Lock Waits	Number of times that threads waited to obtain a lock for the tblspace
Deadlocks	Number of times that a deadlock involved the tblspace

Metric Name	Description
Deadlock Timeouts	Number of times that a deadlock timeout involved the tblspace
Reads	Number of read calls that involve the tblspace
Writes	Number of write calls that involve the tblspace
Rewrites	Number of rewrite calls that involve the tblspace
Deletes	Number of delete calls that involve the tblspace
Calls	Total calls that involve the tblspace
Buffer Reads	Number of buffer reads that involve tblspace data
Buffer Writes	Number of buffer writes that involve tblspace data
Sequential Scans	Number of sequential scans of the tblspace
Percent Free Space	Percent of the tblspace that is free
Pages Allocated	Number of pages allocated to the tblspace
Pages Used	Number of pages allocated to the tblspace that have been written
Data Pages	Number of pages allocated to the tblspace that are used as data pages

Fragment metrics

The **onperf** utility can display metrics for an individual table fragment.

These metrics take the dbspace of an individual table fragment as the metric scope.

Metric Name	Description
Lock Requests	Total requests to lock fragment
Lock Waits	Number of times that threads have waited to obtain a lock for the fragment
Deadlocks	Number of times that a deadlock involved the fragment
Deadlock Timeouts	Number of times that a deadlock timeout involved the fragment
Reads	Number of read calls that involve the fragment
Writes	Number of write calls that involve the fragment
Rewrites	Number of rewrite calls that involve the fragment
Deletes	Number of delete calls that involve the fragment
Calls	Total calls that involve the fragment
Buffer Reads	Number of buffer reads that involve fragment data
Buffer Writes	Number of buffer writes that involve fragment data

Metric Name	Description
Sequential Scans	Number of sequential scans of the fragment
Percent Free Space	Percent of the fragment that is free
Pages Allocated	Number of pages allocated to the fragment
Pages Used	Number of pages allocated to the fragment that have been written to
Data Pages	Number of pages allocated to the fragment that are used as data pages

Appendix

Case studies and examples

This appendix contains a case study with examples of performance-tuning methods that this publication describes.

Case study of a situation in which disks are overloaded

You can identify overloaded disks and the dbspaces that reside on those disks. After you identify the overloaded disks, you can correct the problem.

About this task

The following case study illustrates a situation in which the disks are overloaded. This study shows the steps taken to isolate the symptoms and identify the problem based on an initial report from a user, and it describes the needed correction.

A database application that does not have the wanted throughput is being examined to see how performance can be improved. The operating-system monitoring tools reveal that a high proportion of process time was spent idle, waiting for I/O. The database server administrator increases the number of CPU VPs to make more processors available to handle concurrent I/O. However, throughput does not increase, which indicates that one or more disks are overloaded.

To verify the I/O bottleneck, the database server administrator must identify the overloaded disks and the dbspaces that reside on those disks.

To identify overloaded disks and the dbspaces that reside on those disks:

1. To check the asynchronous I/O (AIO) queues, use **onstat -g ioq**. Figure 97: Output from the onstat -g ioq option on page 463 shows the output.

Figure 97. Output from the onstat -g ioq option

```
AIO I/O queues:
q name/id    len maxlen totalops  dskread dskwrite  dskcopy
opt   0      0      0        0        0        0        0
msc   0      0      0        0        0        0        0
aio   0      0      0        0        0        0        0
pio   0      0      1        1        0        1        0
lio   0      0      1      341        0      341        0
gfd   3      0      1      225        2      223        0
gfd   4      0      1      225        2      223        0
gfd   5      0      1      225        2      223        0
gfd   6      0      1      225        2      223        0
gfd   7      0      0        0        0        0        0
gfd   8      0      0        0        0        0        0
gfd   9      0      0        0        0        0        0
gfd  10      0      0        0        0        0        0
gfd  11      0     28    32693    29603     3090        0
gfd  12      0     18    32557    29373     3184        0
gfd  13      0     22    20446    18496     1950        0
```

In Figure 97: Output from the onstat -g ioq option on page 463, the **maxlen** and **totalops** columns show significant results:

- The **maxlen** column shows the largest backlog of I/O requests to accumulate within the queue. The last three queues are much longer than any other queue in this column listing.
- The **totalops** column shows 100 times more I/O operations completed through the last three queues than for any other queue in the column listing.

The **maxlen** and **totalops** columns indicate that the I/O load is severely unbalanced.

Another way to check I/O activity is to use **onstat -g iov**. This option provides a slightly less detailed display for all VPs.

2. To check the AIO activity for each disk device associated with each queue, use **onstat -g iof**, as Figure 98: Partial output from the onstat -g iof option on page 463 shows.

Figure 98. Partial output from the onstat -g iof option

```
gfd pathname         bytes read     page reads  bytes write    page writes io/s
3  /dev/infx5      85456896       41727       207394816      101267      572.9
 op type     count          avg. time
 seeks       0              N/A
 reads       13975          0.0015
 writes      51815          0.0018
 kaio_reads  0              N/A
 kaio_writes 0              N/A
```

Depending on how your chunks are arranged, several queues can be associated with the same device.

3. To determine the dbspaces that account for the I/O load, use **onstat -d,** as Figure 99: Output from the onstat -d option on page 464 shows.

Results

Figure 99. Output from the onstat -d option

```
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
c009ad00 1        1        1        1        N        informix rootdbs
c009ad44 2        2001     2        1        N T      informix tmp1dbs
c009ad88 3        1        3        1        N        informix oltpdbs
c009adcc 4        1        4        1        N        informix histdbs
c009ae10 5        2001     5        1        N T      informix tmp2dbs
c009ae54 6        1        6        1        N        informix physdbs
c009ae98 7        1        7        1        N        informix logidbs
c009aedc 8        1        8        1        N        informix runsdbs
c009af20 9        1        9        3        N        informix acctdbs
 9 active, 32 total

Chunks
address  chk/dbs offset   size     free     bpages   flags pathname
c0099574 1   1   500000   10000    9100               PO-   /dev/infx2
c009960c 2   2   510000   10000    9947               PO-   /dev/infx2
c00996a4 3   3   520000   10000    9472               PO-   /dev/infx2
c009973c 4   4   530000   250000   242492             PO-   /dev/infx2
c00997d4 5   5   500000   10000    9947               PO-   /dev/infx4
c009986c 6   6   510000   10000    2792               PO-   /dev/infx4
c0099904 7   7   520000   25000    11992              PO-   /dev/infx4
c009999c 8   8   545000   10000    9536               PO-   /dev/infx4
c0099a34 9   9   250000  450000    4947               PO-   /dev/infx5
c0099acc 10  9   250000  450000    4997               PO-   /dev/infx6
c0099b64 11  9   250000  450000    169997             PO-   /dev/infx7
 11 active, 32 total
```

In the **Chunks** output, the **pathname** column indicates the disk device. The **chk/dbs** column indicates the numbers of the chunk and dbspace that reside on each disk. In this case, only one chunk is defined on each of the overloaded disks. Each chunk is associated with dbspace number 9.

The **Dbspaces** output shows the name of the dbspace that is associated with each dbspace number. In this case, all three of the overloaded disks are part of the **acctdbs** dbspace.

Although the original disk configuration allocated three entire disks to the **acctdbs** dbspace, the activity within this dbspace suggests that three disks are not enough. Because the load is about equal across the three disks, it does not appear that the tables are necessarily laid out badly or improperly fragmented. However, you might get better performance by adding fragments on other disks to one or more large tables in this dbspace or by moving some tables to other disks with lighter loads.

Related information

onstat -g iof command: Print asynchronous I/O statistics on page

onstat -g ioa command: Print combined onstat -g information on page

onstat -g ioq command: Print I/O queue information on page

onstat -g iov command: Print AIO VP statistics on page

onstat -d command: Print chunk information on page

Index

C

D

E

M