HCL Informix® VectorBlade 1.0.0

Contents

Chapter 1. Introduction

Imagine a complex object like a digital image, a sound file, or a short paragraph of text containing medical jargon. How can software quickly compare two images for similarity, or determine whether a question posed in plain English, more nuanced than a simple collection of keywords, was best answered by this paragraph or that paragraph? This is the engineering problem that has thus far been addressed by *vectors*.

A *vector* in this context is an array of floating-point numbers called *dimensions*. Vectors describe complex data. The process of generating a vector from an image or a chunk of text is called *embedding*. Once you have a method of succinctly describing an image, for example, in terms of various attributes, you can compare these descriptions and do some very useful database operations such as indexing and searching. To successfully compare two vectors, they must have the same number of dimensions. They must also be generated by the same *embedding model*.

With the HCL Informix VectorBlade, you can now store vectors, index them, compare them for similarity (calculate the *distance* between the two vectors), and perform hybrid searches involving them. You can even generate the embeddings you are storing, but this is not required.

Vectors are stored in your HCL Informix Dynamic Server using a new opaque data type: *lvector_embedding*. In order to index and perform searches on these vectors they must be copied, using one of several UDRs provided with the blade, to special file system files outside of Informix chunks. The location of these external files is configurable.

Chapter 2. Prerequisites

1. Currently the HCL Informix VectorBlade is supported only on the Linux x86_64 platform.
2. Redhat 8.8 or higher O/S is required.
3. The blade will run only with the HCL Informix server version 15.0.1.10 and later releases.

Chapter 3. Getting Started

Installing the blade

As either root or informix:

cd $INFORMIXDIR/extend

tar xvf <path to lvector.1.0.0_EAP.tar>

Configuring the blade

---- IFX_LVECTOR_URI ----

In order to index vectors and perform *nearest neighbor* searches on them efficiently, the HCL Informix VectorBlade must store copies of your vectors in file system files outside of Informix chunks. You can set the location of this storage area using the IFX_LVECTOR_URI configuration parameter in your $INFORMIXDIR/etc/$ONCONFIG file:

```
IFX_LVECTOR_URI <path to storage directory>
Default value:
$INFORMIXDIR/lvector_storage.<DBSERVERNAME>
```

Informix recommends that you ensure IFX_LVECTOR_URI is set to a unique directory for each instance using a particular $INFORMIXDIR. The directory should have ownerships of informix:informix, permissions 770.

```
---- IFX_LVECTOR_OPENAI_KEY ----
```

For convenience, the blade has a built-in interface with OpenAI for embedding operations. To successfully use the blade to generate an embedding with OpenAI you will need a valid OpenAI API Key. Set that key for all databases in your instance by setting the IFX_LVECTOR_OPENAI_KEY configuration parameter. Example:

```
IFX_LVECTOR_OPENAI_KEY sk-proj--xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx -D0GbcqdKRkX
```

Alternatively you may set the OpenAPI API key on a per-database basis by executing the following routine as user informix while connected to your database:

```
EXECUTE PROCEDURE lvector_set_embedding_api_key(
'sk-proj--
xxxxx9_V4YfRXV3aWOEw9Al0fviZg_VmLqRCHc_gIMAFRAIDTHISISNOTAWORKINGKEYVJQlRHu_5gH5GlaRD2lbkFJpZv_lw5b5Rxpu_dl6msz
DcdO_SD11vWp73IMAFRAIDTHISISNOTAWORKINGKEY-D0GbcqdKRkX');
```

Warning: If you use this method to set your key(s), instead of the configuration parameter, you will need to re-execute the lvector_set_embedding_api_key() routine for each database each time the server is restarted.

To create an OpenAI API key for generating vector embeddings, you must access the OpenAI API platform , separate from the standard ChatGPT interface.

Steps to Create Your API Key

1. Log In to the Platform: Go to the OpenAI Platform and log in with your existing OpenAI account or create a new one.
2. Navigate to API Keys: On the left-hand sidebar of the Dashboard, click on the API keys tab.
3. Generate a New Key: Click the "+ Create new secret key" button.
4. Label Your Key: Give your key a name (e.g., "Embeddings Project") to identify its purpose later.
5. Copy and Save: Your full secret key will only be shown once. Copy it immediately and store it in a secure location (like a password manager or environment file).

Essential Requirements for Vector Embeddings

- Billing Information: You must add a payment method and purchase credits (starting at $5) for the API key to function; the API is not free beyond limited initial trial credits.
- Models for Embeddings: When using your key, the blade is explicitly using text-embedding-3-small.
- Security: Never share your key or commit it to public code repositories like GitHub.

---- IFX_LVECTOR_MAX_CONN ----

You can limit the number of simultaneous users who perform ANN (approximate nearest neighbor) searches using the blade by configuring the IFX_LVECTOR_MAX_CONN parameter. This parameter's maximum value is 16, which is also the default:

IFX_LVECTOR_MAX_CONN 16

---- STACKSIZE ----

Whenever data blades or User Defined Routines (UDRs) are used heavily in an Informix instance it is a good idea to increase the thread stack size from its default of 64 KB. One does this using the STACKSIZE parameter in your $INFORMIXDIR/etc/$ONCONFIG file:

STACKSIZE 128

Informix recommends a thread stack size of 128 when using the

VectorBlade. ---- VPCLASS ----

The UDRs bundled with this blade are designed to run on the 'lvec' VP class. When your blade is registered for the first time, one 'lvec' VP will be created if none exists. However, it is better to indicate the existence of this VP class in your config file, and to configure the initial number of VPs in the class. For example:

VPCLASS lvec,num=4

---- PRELOAD_DLL_FILE ----

Include $INFORMIXDIR/extend/lvector.1.0.0_EAP/lvector.bld in the value of your PRELOAD_DLL_FILE configuration parameter and the Vector Blade's shared library will be automatically loaded each time the server is started. Otherwise it will load the first time one of the blade's UDRs is executed, which may cause a delay of several seconds.

Manual Registration of the Blade

With your Informix instance on-line and your blade installed in $INFORMIXDIR/extend you may manually register the blade for a particular database by connecting to that database as user informix and performing the following SQL statement:

execute function sysbldprepare('lvector.*','create');

A successful execution of that statement will load information into the current database's system catalogs about all

UDTs and UDRs required by the VectorBlade. That database will then be primed to work with the blade.

Now, if you have not already done so, add the 'lvec' VP class to your $INFORMIXDIR/etc/$ONCONFIG file. For example:

VPCLASS lvec,num=4

This change will take effect the next time you restart your server instance. If you do not wish to restart the instance you may also add your preferred number of lvec VPs dynamically using the onmode -p command. For example:

onmode -p +3 lvec

When run by informix or root on the Unix command line that command will dynamically add 3 lvec VPs to your running instance, which can then be seen in the output of either of these commands:

onstat -g sch

onstat -g glo

The UDRs in your VectorBlade will now execute on an 'lvec' VP instead of a CPU VP. Informix recommends running them

on this VP class.

Automatic Registration o the Blade

Alternatively you may auto-register the VectorBlade for a particular database by connecting to that database and executing any of the UDRs included in the blade. For example you could set your OpenAI API key:

```
EXECUTE PROCEDURE lvector_set_embedding_api_key(
 'sk-proj--xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx9_V4YfRXV3aWOEw9Al0fviZg_VmLqRCHc_gIMAFRAIDTHISISNOTAWORKINGKEYVJQlRHu_5gH5GlaRD2lbkFJpZv_lw5b5Rxpu_dl6msz
DcdO_SD11vWp73IMAFRAIDTHISISNOTAWORKINGKEY-D0GbcqdKRkX ');
```

The first time a blade is auto-registered for a database:

1. That database's system catalogs are populated with information about all UDTs and UDRs required by the blade.
2. A single 'lvec' VP will be dynamically added, but note that no associated VPCLASS entry will be added to your $INFORMIXDIR/etc/$ONCONFIG file. That addition will still need to be done manually at some point.
3. The lvector.bld shared library, which contains the UDR code, will load. You should see messages to this effect in the server message log.

You will find more information below on all UDTs and UDRs included with the VectorBlade.

Chapter 4. Using the Blade

Example 1:

Let's start with a simple use case. We'll create a special 'lance' table that can be used to store vectors and perform "nearest neighbor" searches on them. This lance table has three columns of the following types:

(bigint, embedding, lvarchar)

The bigint column stores vector IDs. These must be unique numbers.

The embedding column can store vectors with any number of dimensions, but that number must be specified at table creation time.

The lvarchar column is for storing textual metadata. This metadata can be anything you like and can be useful for filtering (see example 5).

```
-- Create a database
create database examples with log;
-- Create a lance table called 'sample_tab1' to store vectors of 8
-- dimensions. This operation should also auto-register your vector
-- blade.
EXECUTE PROCEDURE lvector_create_table('sample_tab1', 8);
-- Insert some vectors. This must be done inside an explicit
-- transaction.
EXECUTE PROCEDURE lvector_sync_insert
    (
    'sample_tab1',
    1,
    lvector_in('[1,2,3,4,5,6,7,8]'),
    'This is vector 1'
    );

EXECUTE PROCEDURE lvector_sync_insert
    (
    'sample_tab1',
    2,
    lvector_in('[8,7,6,5,4,3,2,1]'),
    'This is vector 2'
    );

EXECUTE PROCEDURE lvector_sync_insert
    (
    'sample_tab1',
    3,
    lvector_in('[1,1,2,2,3,3,4,4]'),
    'This is vector 3'
    );

EXECUTE PROCEDURE lvector_sync_insert
    (
    'sample_tab1',
    4,
    lvector_in('[5,5,6,6,7,7,8,8]'),
    'This is vector 4'
```

```
    );

EXECUTE PROCEDURE lvector_sync_insert
    (
    'sample_tab1',
    5,
    lvector_in('[3,3,3,3,3,3,3,3]'),
    'This is vector 5'
    );

COMMIT;

-- Now let's find the vector in our table that is most like a given
-- vector: [1,1,2,2,3,3,3,3]

SELECT *
FROM TABLE
    (
    lvector_search
        (
        'sample_tab1',
        lvector_in('[1,1,2,2,3,3,3,3]'),
        1
        )
    );

The lvector_search() routine returns a row type with the following structure:
create row type lvector_search_result
    (
    id bigint,
    distance float
    );
```

Note that it does not return a vector. It returns the *ID* of the vector judged to be most similar to the given vector, along with a distance value indicating *how similar* they were. The lower the distance value, the more similar the vectors are. The distance between two identical vectors is 0.

In the case of the query above, the results are:

unnamed_col_1 ROW(3 ,2.000000000000)

This means the nearest neighbor to the given vector had an ID of 3, and the distance between the vectors was 2.000000000000. In other words, given a vector of [1,1,2,2,3,3,3,3], the following vector was judged to be the nearest neighbor:

[1,1,2,2,3,3,4,4]

We could also do the following to find the 3 nearest neighbors:

```
SELECT *
FROM TABLE
    (
    lvector_search
        (
```

```
        'sample_tab1',
        lvector_in('[1,1,2,2,3,3,3,3]'),
        3
        )
    );
```

The result of that search is:

unnamed_col_1 ROW(3 ,2.000000000000)

unnamed_col_1 ROW(5 ,10.00000000000)

unnamed_col_1 ROW(1 ,60.00000000000)

In this case the distance values indicate that the first nearest neighbor is significantly more similar to the given vector than the second and third nearest neighbors.

Let's make this output a little more convenient to deal with by teasing apart the returned elements:

```
SELECT results.unnamed_col_1.id id,
       results.unnamed_col_1.distance distance
FROM TABLE
    (
    lvector_search
        (
        'sample_tab1',
        lvector_in('[1,1,2,2,3,3,3,3]'),
        3
        )::lvector_search_result
    ) results;
```

The result of that statement is:

id distance

3 2.000000000000

5 10.00000000000

1 60.00000000000

Note that the IDs returned by lvector_search() are guaranteed to be sorted by increasing distance. Sorting on the distance column in the query itself is not necessary.

If our only column of interest was the id, we could use a different function:

```
SELECT *
FROM TABLE
    (
    lvector_id_search
        (
        'sample_tab1',
        lvector_in('[1,1,2,2,3,3,3,3]'),
```

```
        3
        )
    );
```

As with lvector_search() the set of IDs returned by lvector_id_search() is guaranteed to be sorted by increasing distance.

Example 2:

Let's make the previous example a little more interesting by associating some meaningful text with the vectors. In this case there is no direct connection between the text and the associated vector, but we'll get to that.

This example will involve a combination of a traditional Informix table along with the lance table used for vector comparisons.

```
-- Create a traditional table in the same database
database examples;
create table ifx_sample_tab2
    (
    id bigint,
    embedding lvector_embedding,
    metadata lvarchar,
    unused datetime year to second,
    primary key (id)
    );

-- Insert some rows into this table

insert into ifx_sample_tab2 values
    (
    1,
    '[1,2,3,4,5,6,7,8]',
    'This vector is somehow related to squirrels',
    '2026-01-26 18:43:20'
    );

insert into ifx_sample_tab2 values
    (
    2,
    '[8,7,6,5,4,3,2,1]',
    'This vector is somehow related to car racing',
    '2026-02-26 18:43:20'
    );

insert into ifx_sample_tab2 values
    (
    3,
    '[1,1,2,2,3,3,4,4]',
    'This vector is somehow related to architecture',
    '2026-03-26 18:43:20'
    );

insert into ifx_sample_tab2 values
    (
    4,
    '[5,5,6,6,7,7,8,8]',
```

```
    'This vector is somehow related to wrestling',
    '2026-04-26 18:43:20'
    );

insert into ifx_sample_tab2 values
    (
    5,
    '[3,3,3,3,3,3,3,3]',
    'This vector is somehow related to sound',
    '2026-05-26 18:43:20'
    );
```

-- Now let's create a lance table to help us search on these vectors

```
EXECUTE PROCEDURE lvector_create_table('sample_tab2', 8);
-- Copy the id, vector, and metadata columns into the lance table.
-- Note that the name of the metadata column must be 'metadata'. Also
-- note that we are not copying the datetime data.
EXECUTE PROCEDURE lvector_bulk_sync('ifx_sample_tab2', 'sample_tab2', 'id', 'embedding');
-- Now let's find the metadata string associated with the vector that
-- is most similar to [9,8,7,6,5,4,3,2]:
SELECT metadata
FROM ifx_sample_tab2
WHERE id IN
    (SELECT * FROM TABLE(lvector_id_search('sample_tab2',
        lvector_in('[9,8,7,6,5,4,3,2]'), 1)));
```

According to our vector comparison, the given vector is most similar to the vector associated with car racing. Since this association was completely contrived in this case, these results aren't particularly useful. But we will improve upon that in the next example, which uses OpenAI to create actual embeddings from text. In order to run this SQL you will need an active OpenAI key, and you will need to provide your key to Informix one of two ways:

1. Set your IFX_LVECTOR_OPENAI_KEY configuration parameter to your key and bounce the instance, or
2. Execute a procedure to set the key on the fly, as in:

database examples;

EXECUTE PROCEDURE lvector_set_embedding_api_key('sk-proj--9_V4YfRXV3xxxxxxbcqdKRkA');

Once this key is set you can test it with the following statement:

EXECUTE FUNCTION lvector_embed('Hello world');

If that function returns an array of 1536 floating point numbers, your OpenAI key is working beautifully. However, the following error indicates that the setting is unfortunately not yet working:

(U0001) - Failed to generate embedding

Error in line 1

Near character position 45

The most likely reasons for this error:

1) Your IFX_LVECTOR_OPENAI_KEY configuration parameter is not set correctly in your $INFORMIXDIR/etc/$ONCONFIG file. The line should look something like this:

IFX_LVECTOR_OPENAI_KEY sk-proj--xxxxxxxxxxxxxxxxx9_V4YfRXV3aWOEw9Al0fviZg_D0GbcqdKRkX

2) You did not bounce (shut down and restart) your instance after setting IFX_LVECTOR_OPENAI_KEY in your config file

3) Instead of setting the IFX_LVECTOR_OPENAI_KEY configuration parameter you chose to set the key on the fly using the lvector_set_embedding_api_key() routine, but since doing so your current database has changed.

4) The key you have provided to Informix is not an active OpenAI key.

5) The machine on which you are running your Informix instance does not have access to the internet.

Assuming your OpenAI key is set properly and you are able to successfully execute the lvector_embed() routine, the following example will be of interest.

Example 3:

```
-- Create a new traditional table
database examples;
create table ifx_sample_tab3
    (
    ident bigint,
    embedding_data lvector_embedding,
    text_data lvarchar,
    metadata lvarchar,
    primary key (ident)
    );

-- Generate some real embeddings and insert them into our tableinsert into ifx_sample_tab3 values
    (
    1,
    lvector_embed('This text is about squirrels'),
    'This text is about squirrels',
    'This metadata will not be used in this example'
    );

insert into ifx_sample_tab3 values
    (
    2,
    lvector_embed('This text is about car racing'),
    'This text is about car racing',
    'This metadata will not be used in this example'
    );
insert into ifx_sample_tab3 values
    (
    3,
    lvector_embed('This text is about architecture'),
    'This text is about architecture',
    'This metadata will not be used in this example'
    );

insert into ifx_sample_tab3 values
```

```
    (
    4,
   lvector_embed('This text is about wrestling'),
    'This text is about wrestling',
    'This metadata will not be used in this example'
    );

insert into ifx_sample_tab3 values
    (
    5,
    lvector_embed('This text is about sound'),
    'This text is about sound',
    'This metadata will not be used in this example'
    );
```

-- Say we want to find the text most relevant to the question, "Tell me about building design". First, we have to create a lance table that we can use to perform this search. We indicate that the vectors we will be storing have 1536 dimensions, because this is the format of the vectors returned by OpenAI:

```
EXECUTE PROCEDURE lvector_create_table('sample_tab3', 1536);
```

-- Now we synchronize this lance table with our traditional table:

```
EXECUTE PROCEDURE lvector_bulk_sync('ifx_sample_tab3', 'sample_tab3', 'ident', 'embedding_data');
```

-- Now we can perform a true vector search:

```
SELECT text_data
FROM ifx_sample_tab3
WHERE ident IN
    (SELECT * FROM TABLE(lvector_id_search('sample_tab3',
        lvector_embed('Tell me about building design'), 1)));

You should see that the results of this vector search are:

text_data  This text is about architecture


How about this one:

SELECT text_data
FROM ifx_sample_tab3
WHERE ident IN
    (SELECT * FROM TABLE(lvector_id_search('sample_tab3',
        lvector_embed('Tell me about cute rodents'), 1)));

Expected results:

text_data  This text is about squirrels


Or maybe this one:

SELECT text_data
FROM ifx_sample_tab3
WHERE ident IN
```

```
    (SELECT * FROM TABLE(lvector_id_search('sample_tab3',
        lvector_embed('Tell me about sports'), 2)));

Expected results:
text_data  This text is about car racing
text_data  This text is about wrestling
```

Example 4:

In this example we'll insert 304 vectors and index them before searching on them. Once the number of vectors that you are searching grows into the hundreds of thousands, using an index for the search is a lot more efficient.

```
First, execute the following at the UNIX command line:awk 'BEGIN {
 print "BEGIN WORK;";
 for (i=1; i<=300; i++) {
    printf "insert into ifx_sample_tab4 values (%d, lvector_embed('\''sample text %d'\''), '\''{\"sample text
 %d\"}'\'', '\''{\"This metadata will not be used in this example\"}'\'');\n", i, i, i;
 }

printf "insert into ifx_sample_tab4 values (301, lvector_embed('\''This text is about the new york cycle
 club'\''), '\''{\"This text is about the new york cycle club\"}'\'', '\''{\"This metadata will not be used in
 this example\"}'\'');\n";
printf "insert into ifx_sample_tab4 values (302, lvector_embed('\''This text is about the new york citrus
 club'\''), '\''{\"This text is about the new york citrus club\"}'\'', '\''{\"This metadata will not be used in
 this example\"}'\'');\n";
printf "insert into ifx_sample_tab4 values (303, lvector_embed('\''This text is about the los angeles penal
 code'\''), '\''{\"This text is about the los angeles penal code\"}'\'', '\''{\"This metadata will not be used
 in this example\"}'\'');\n";
printf "insert into ifx_sample_tab4 values (304, lvector_embed('\''This text can be about anything you want.
 Let us say... monkeys.'\''), '\''{\"This text can be about anything you want. Let us say... monkeys.\"}'\'',
 '\''{\"This metadata will not be used in this example\"}'\'');\n";
print "COMMIT WORK;";
}' > /tmp/load_ifx_sample_tab4.sql
Now create our traditional table:
database examples;
create table ifx_sample_tab4
    (
    ident bigint,
    embedding_data lvector_embedding,
    text_data lvarchar,
    metadata lvarchar,
    primary key (ident)
    );
```

Exit out of dbaccess. Now insert vectors and their associated text into our traditional table using the script:

```
dbaccess examples /tmp/load_ifx_sample_tab4.sql
Back into dbaccess now.
-- Create our lance table that will be used for vector searches
database examples;
EXECUTE PROCEDURE lvector_create_table('sample_tab4', 1536);
-- Synchronize the traditional table with the lance table
database examples;
EXECUTE PROCEDURE lvector_bulk_sync('ifx_sample_tab4', 'sample_tab4', 'ident', 'embedding_data');
-- Create an index on the lance table
EXECUTE PROCEDURE lvector_create_index('sample_tab4', 'ivf_hnsw_pq ', 1);
-- Now let's do a symantic search that will make use of the vector
-- index
```

```
SELECT text_data
FROM ifx_sample_tab4WHERE ident IN
    (SELECT * FROM TABLE(lvector_id_search('sample_tab4',
        lvector_embed('Tell me about the NYCC'), 2)));

Results:
text_data {"This text is about the new york cycle club"}
text_data {"This text is about the new york citrus club"}
```

Example 5:

In our final example we will demonstrate vector search with metadata filtering.

```
-- Create a new traditional table
database examples;
create table ifx_sample_tab5
  (
    ident bigint,
    embedding_data lvector_embedding,
    text_data lvarchar,
    metadata lvarchar,
    primary key (ident)
    );

-- Generate some real embeddings and insert them into our table
insert into ifx_sample_tab5 values  (
    1,
    lvector_embed('This text is about squirrels'),
    'This text is about squirrels',
    'Category: Animals'
    );

insert into ifx_sample_tab5 values
    (
    2,
    lvector_embed('This text is about car racing'),
    'This text is about car racing',
    'Category: Modern Sports'
    );

insert into ifx_sample_tab5 values
      (
      3,
      lvector_embed('This text is about architecture'),
      'This text is about architecture',
      'Category: The Arts'
      );

      insert into ifx_sample_tab5 values
      (
      4,
      lvector_embed('This text is about wrestling'),
      'This text is about wrestling',
      'Category: Ancient Sports'
      );

insert into ifx_sample_tab5 values
      (
```

```
      5,
      lvector_embed('This text is about sound'),
      'This text is about sound',
      'Category: Music'
      );
-- Create a lance table and synchronize it with the traditional one
EXECUTE PROCEDURE lvector_create_table('sample_tab5', 1536);
EXECUTE PROCEDURE lvector_bulk_sync('ifx_sample_tab5', 'sample_tab5', 'ident', 'embedding_data');
-- Now in our previous example we ran a query that returned two rows relevant to sports:
SELECT text_data
FROM ifx_sample_tab5
WHERE ident IN
    (SELECT * FROM TABLE(lvector_id_search('sample_tab5',
        lvector_embed('Tell me about sports'), 2)));
```

Expected results:

```
text_data This text is about car racing
```

```
text_data This text is about wrestling
```

But what if we only want sports-related text returned if it definitely pertains to a *modern* sport, based on a keyword search of the metadata. We could filter on the metadata while performing the vector search:

```
SELECT text_data
FROM ifx_sample_tab5
WHERE ident IN
    (SELECT results.unnamed_col_1.id
     FROM TABLE
         (
         lvector_search_filtered
             (
             'sample_tab5',
             lvector_embed('Tell me about sports'),
             2,
             'metadata like "%Modern%"'
             )::lvector_search_result
         ) results
    );
```

Additional example scripts have been provided with your blade. Find them in $INFORMIXDIR/extend/lvector*/samples

Chapter 5. Using the Blade (Continued)

Indexing your vectors

You do not have to create indexes on your lance tables in order to perform nearest-neighbor searches on them. However, these searches will perform significantly better with indexes.

Indexes on these tables are static. However, vectors inserted after the index is built can still be searched. The search will automatically use the index and then pivot to a brute-force search on the unindexed data.

You do not need to drop an index on a lance table in order to rebuild it. Simply create the index again and the original index will automatically be dropped.

See the lvector_create_index() UDR for more information

Archiving lance tables and indexes

Your lance tables and indexes are stored outside of Informix-managed chunks, in a directory specified by the IFX_LVECTOR_URI configuration parameter. When archiving your Informix instance it is recommended that you manually archive the contents of IFX_LVECTOR_URI. Modifications to lance tables are not recorded in the Informix logical logs. Therefore these modifications cannot be logically recovered or replicated using IDS. If your Informix instance is part of a grid, you will need to replicate the contents of IFX_LVECTOR_URI to the other nodes through some other mechanism.

Because Informix does not archive or replicate lance tables or indexes, best practice is to store copies of all vectors, vector ids, and metadata in one or more informix tables, then copy that data into your lance tables via the lvector_bulk_sync() UDR. Your Informix tables represent the "source of truth" in case you lose your lance tables or they somehow become out of sync with your informix tables. Lance tables and indexes can be recreated relatively quickly from the source of truth using bulk synchronization.

If you restore your Informix instance from an archive you will need to either separately restore the contents of IFX_LVECTOR_URI or recreate your lance tables using your Informix tables as the source of truth.

Chapter 6. Resources Included With Your HCL Informix VectorBlade

User-Defined Types:

create opaque type lvector_embedding(

internallength = variable,

maxlen = 6148,

alignment = 4,

cannothash

);

User Defined Routines:

Setup and con iguration routines
1. lvector_set_embedding_api_key

```
Signature:
CREATE PROCEDURE lvector_set_embedding_api_key(api_key LVARCHAR);
Purpose: Stores or configures the embedding provider API key used by embedding generation routines.
Example:
EXECUTE PROCEDURE lvector_set_embedding_api_key('YOUR_API_KEY');
```

Note: Run this before calling lvector_embed() or lvector_generate_embedding() if the embedding provider requires an API key.

2. lvector_set_option

```
Signature:
CREATE PROCEDURE lvector_set_option(key LVARCHAR, value LVARCHAR);
Purpose: Sets runtime options for the lvector blade.
Examples:
EXECUTE PROCEDURE lvector_set_option('default_uri', '<new path>');
EXECUTE PROCEDURE lvector_set_option('log_level', 'debug');
EXECUTE PROCEDURE lvector_set_option('log_level', 'info');
EXECUTE PROCEDURE lvector_set_option('log_level', 'warn');
EXECUTE PROCEDURE lvector_set_option('log_level', 'error');
```

Note: Enable debug logging before reproducing insert, delete, search, or index issues.

3. lvector_status

```
Signature:
CREATE FUNCTION lvector_status();
```

```
Purpose: Returns lvector runtime status information.
Example:
EXECUTE FUNCTION lvector_status();
```

Note: Use as a smoke test to confirm the blade is installed, registered, and callable.

4. lvector_version

```
Signature:
CREATE FUNCTION lvector_version();
Purpose: Returns version information for the lvector blade/runtime.
Example:
EXECUTE FUNCTION lvector_version();
```

Embedding and vector math routines

5. lvector_embed

```
Signature:
CREATE FUNCTION lvector_embed(txt LVARCHAR) RETURNS lvector_embedding;
Purpose: Generates an embedding vector for text input using the configured embedding provider.
Example:
SELECT lvector_dimensions(lvector_embed('hello world')) AS dims
FROM systables
WHERE tabid = 1;
```

Note: Expected dimension is typically 1536 depending on the configured embedding model. Failures can indicate API key, quota, network, or model configuration problems.

6. lvector_generate_embedding

```
Signature:

CREATE FUNCTION lvector_generate_embedding(txt LVARCHAR, model LVARCHAR, dimensions INTEGER) RETURNS
 lvector_embedding;
Purpose: Generates an embedding using explicit model and dimension arguments.
Example:
SELECT lvector_dimensions(
lvector_generate_embedding('hello world', 'text-embedding-3-small', 1536)
) AS dims
FROM systables
WHERE tabid = 1;
```

Note: Use lvector_embed() for simpler demos unless testing a specific model/dimension path.

7. lvector_dimensions

```
Signature:
CREATE FUNCTION lvector_dimensions(lvec lvector_embedding);
Purpose: Returns the dimension count of a lvector_embedding value.
Example:
```

```
SELECT lvector_dimensions(lvector_in('[1,2,3,4]')) AS dims
FROM systables
WHERE tabid = 1;
```

Note: Tested output for the example above: dims = 4.

7. lvector_distance

```
Signature:
CREATE FUNCTION lvector_distance(v1 lvector_embedding, v2 lvector_embedding, metric LVARCHAR DEFAULT 'l2');
Purpose: Calculates distance or similarity between two lvector values using a metric such as l2, cosine, or
dot/ip.
Example:
SELECT
lvector_distance( lvector
_in('[1,2,3,4]'),
lvector_in('[1,2,3,4]'),
'l2'
) AS distance
FROM systables
WHERE tabid = 1;
```

Note: Distance between identical vectors should be 0 or very close to 0 depending on metric and implementation.

9. lvector_normalize

```
Signature:
CREATE FUNCTION lvector_normalize(lvec lvector_embedding) RETURNS lvector_embedding;
Purpose: Normalizes a lvector_embedding value.
Example:
SELECT lvector_out(lvector_normalize(lvector_in('[1,2,3,4]'))) AS normalized
FROM systables
WHERE tabid = 1;
```

10. lvector_compare

```
Signature:
CREATE FUNCTION lvector_compare(v1 lvector_embedding, v2 lvector_embedding);
Purpose: Compares two lvector_embedding values. It is also related to type comparison support.
Example:SELECT lvector_compare(lvector_in('[1,2,3,4]'), lvector_in('[1,2,3,4]')) AS cmp
FROM systables
WHERE tabid = 1;
```

Note: There is also a generic compare(lvector_embedding,lvector_embedding) support function in objects.sql for the opaque type. If direct catalog queries trigger compare resolution errors, use objects.sql as the source of truth for signatures.

lvector type I/O routines

11. lvector_in

```
Signature:
CREATE FUNCTION lvector_in(input LVARCHAR) RETURNS lvector_embedding;
Purpose: Converts text representation into the internal lvector_embedding type.
Example:SELECT lvector_dimensions(lvector_in('[1,2,3,4]')) AS dims
FROM systables
WHERE tabid = 1;
```

Note: This is user-testable and useful in demos. Tested output: dims = 4.

12. lvector_out

```
Signature:
CREATE FUNCTION lvector_out(lvec lvector_embedding) RETURNS LVARCHAR;
Purpose: Converts the internal lvector_embedding value back to text.
Example:
SELECT lvector_out(lvector_in('[1,2,3,4]')) AS roundtrip
FROM systables
WHERE tabid = 1;
```

Note: Tested output: roundtrip = [1,2,3,4].

13. lvector_recv

```
Signature:
CREATE FUNCTION lvector_recv(data SENDRECV) RETURNS lvector_embedding;
Purpose: Binary receive function for the lvector type.
```

Note: Type-system support routine. Usually not called directly by application SQL.

14. lvector_send

```
Signature:
CREATE FUNCTION lvector_send(lvec lvector_embedding) RETURNS SENDRECV;
Purpose: Binary send function for the lvector type.
```

Note: Type-system support routine. Usually not called directly by application SQL.

Table lifecycle routines

15. lvector_create_table

```
Signature:
CREATE PROCEDURE lvector_create_table(table_name LVARCHAR, dimensions INTEGER);
Purpose: Creates or initializes a corresponding LanceDB/lvector table.
Example:
EXECUTE PROCEDURE lvector_create_table('docs_demo', 16);
```

Note: Informix source data and the LanceDB backing table must stay synchronized through explicit sync calls or triggers.

Note: The hard-coded table schema is as follows:

```
(
Id <64-bit integer>,
lvector <array[dimensions] of 32-bit floats>,
metadata <long variable-length UTF8>
)
```

16. lvector_drop_table

```
Signature:
CREATE PROCEDURE lvector_drop_table(table_name LVARCHAR);
Purpose: Drops a specific LanceDB/lvector table.
Example:
EXECUTE PROCEDURE lvector_drop_table('docs_demo');
```

17. lvector_drop_all_tables

```
Signature:
CREATE FUNCTION lvector_drop_all_tables();
Purpose: Drops all LanceDB/lvector tables managed by the extension.
Example:
EXECUTE FUNCTION lvector_drop_all_tables();
```

Note: Destructive operation. Use only for test cleanup or controlled reset scenarios.

18. lvector_list_tables

```
Signature:
CREATE FUNCTION lvector_list_tables();
Purpose: Lists LanceDB/lvector tables known to the extension.
Example:
EXECUTE FUNCTION lvector_list_tables();
```

19. lvector_table_info

```
Signature:
CREATE FUNCTION lvector_table_info(table_name LVARCHAR);
Purpose: Returns metadata for a lvector table.
Example:
EXECUTE FUNCTION lvector_table_info('docs_demo');
```

Note: Note: Expected fields include table_name, version, row_count, and index_count. If index_count stays 0 after index creation, the index did not complete successfully.

20. lvector_count

```
Signature:
CREATE FUNCTION lvector_count(table_name LVARCHAR);
Purpose: Returns the row/vector count for a lvector table.
Example:
EXECUTE FUNCTION lvector_count('docs_demo');
```

Index lifecycle routines

15. lvector_create_index

```
Signature:
CREATE PROCEDURE lvector_create_index(table_name LVARCHAR, index_type lvarchar, num_partitions INTEGER);
Purpose: Creates or recreates a corresponding LanceDB/lvector index.
Examples:
EXECUTE PROCEDURE lvector_create_index('docs_demo', '', 1);
EXECUTE PROCEDURE lvector_create_index('docs_demo', 'ivf_flat', 1);
EXECUTE PROCEDURE lvector_create_index('docs_demo', 'ivf_pq', 16);
EXECUTE PROCEDURE lvector_create_index('docs_demo', 'ivf_hnsw_pq', 1);
EXECUTE PROCEDURE lvector_create_index('docs_demo', 'ivf_hnsw_sq', 1);
EXECUTE PROCEDURE lvector_create_index('docs_demo', 'btree', 0);
EXECUTE PROCEDURE lvector_create_index('docs_demo', 'bitmap', 0);
Additional details:
Lower num_partition values seem to perform best.
Specifying a blank index type ('') will result in a type being automatically chosen by the system.
All ivf_* index types are created on the 'lvector' column.
All other index types are created on the 'metadata' column.
Your mileage may vary with index type choices, but speaking generally:
 Ivf_pq is typically used with large numbers of vectors (greater than half a million).
 Ivf_hnsw* is typically used when accuracy is critical.
```

Note: Choose your index type based on the number of vectors in your table and your accuracy vs. speed requirements.

Sync routines

21. lvector_sync_insert

```
Signature:
CREATE PROCEDURE lvector_sync_insert(table_name LVARCHAR, id BIGINT, embedding lvector_embedding, metadata
 LVARCHAR);
Purpose: Syncs an inserted Informix row into the LanceDB/lvector backing table.
Example:
BEGIN WORK;
EXECUTE PROCEDURE lvector_sync_insert(
'docs_demo',
1,
lvector_in('[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]'),
'sample document'
);
COMMIT WORK;
```

Note: Can be called directly or from an insert trigger.

22. lvector_sync_update

```
Signature:
CREATE PROCEDURE lvector_sync_update(table_name LVARCHAR, id BIGINT, embedding lvector_embedding, metadata
 LVARCHAR);
Purpose: Syncs an updated Informix row into the lvector backing table.
Example:
BEGIN WORK;
EXECUTE PROCEDURE lvector_sync_update(
'docs_demo',
1,
lvector_in('[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]'),
'updated document'
);
COMMIT WORK;
```

23. lvector_sync_delete

```
Signature:
CREATE PROCEDURE lvector_sync_delete(table_name LVARCHAR, id BIGINT);
Purpose: Deletes the corresponding row from the lvector backing table.
Example:
BEGIN WORK;
EXECUTE PROCEDURE lvector_sync_delete('docs_demo', 1);
COMMIT WORK;
```

Note: If Informix IDs and LanceDB IDs do not match, delete sync may not remove the expected row. Large delete flushes are now batched into 1000-ID LanceDB calls to avoid OOM kills observed during 10K/20K delete tests.

24. lvector_bulk_sync

```
Signature:
CREATE PROCEDURE lvector_bulk_sync(ifx_table LVARCHAR, lvector_table LVARCHAR, id_column LVARCHAR,
 lvector_column LVARCHAR);
Purpose: Bulk syncs existing Informix table data into the lvector/LanceDB backing table.
Example:
EXECUTE PROCEDURE lvector_bulk_sync('src_docs', 'docs_demo', 'id', 'embedding');
```

Note: Use after initial load or after recreating a backing LanceDB table. Column names should be safe SQL identifiers.

Search routines

Routine	Signature
lvector_search	lvector_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER DEFAULT 10)

Routine	Signature
lvector_search	lvector_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, nprobes INTEGER, ef INTEGER DEFAULT 0, refine_factor INTEGER DEFAULT 0)
lvector_search	lvector_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, metric LVARCHAR)
lvector_search	lvector_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, nprobes INTEGER, ef INTEGER, refine_factor INTEGER, metric LVARCHAR)
lvector_id_search	lvector_id_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER DEFAULT 10)
lvector_id_search	lvector_id_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, nprobes INTEGER, ef INTEGER DEFAULT 0, refine_factor INTEGER DEFAULT 0)
lvector_id_search	lvector_id_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, metric LVARCHAR)
lvector_id_search	lvector_id_search(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, nprobes INTEGER, ef INTEGER, refine_factor INTEGER, metric LVARCHAR)
lvector_search_filtered	lvector_search_filtered(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, filter LVARCHAR)
lvector_search_filtered	lvector_search_filtered(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, filter LVARCHAR, nprobes INTEGER, ef INTEGER DEFAULT 0, refine_factor INTEGER DEFAULT 0)
lvector_search_filtered	lvector_search_filtered(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, filter LVARCHAR, metric LVARCHAR)
lvector_search_filtered	lvector_search_filtered(table_name LVARCHAR, query_lvector lvector_embedding, k INTEGER, filter LVARCHAR, nprobes INTEGER, ef INTEGER, refine_factor INTEGER, metric LVARCHAR)