Hewlett Packard Enterprise

Using Flex Tables

HPE Vertica Analytic Database

Software Version: 7.2.x

Document Release Date: 12/7/2015

Legal Notices

Warranty

The only warranties for Hewlett Packard Enterprise products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HPE shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HPE required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

This product includes an interface of the 'zlib' general purpose compression library, which is Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.

Contents

Getting Started

Getting Started describes the basics of creating, exploring, and using flex tables. The rest of this guide presents *beyond the basics* details using simple examples.

Create a Simple JSON File

Use this JSON data for the exercises in the rest of this section:

```
{"name": "Everest", "type":"mountain", "height":29029, "hike_safety": 34.1}
{"name": "Mt St Helens", "type":"volcano", "height":29029, "hike_safety": 15.4}
{"name": "Denali", "type":"mountain", "height":17000, "hike_safety": 12.2}
{"name": "Kilimanjaro", "type":"mountain", "height":14000 }
{"name": "Mt Washington", "type":"mountain", "hike_safety": 50.6}
```

1. Copy and paste the JSON data into your favorite editor.
2. Save the file in any convenient location for loading into your Vertica database.

Create a Flex Table and Load Data

1. Create a flex table called `mountains`:

```
=> CREATE flex table mountains();
```

2. Load the JSON file you saved, using the flex table parser `fjsonparser`:

```
=> COPY mountains from '/home/dbadmin/data/flex/mountains.json' parser fjsonparser();
 Rows Loaded
-------------
           5
(1 row)
```

3. Query values from the sample file:

```
=> SELECT name, type, height from mountains;
     name      |   type   | height
---------------+----------+--------
 Everest       | mountain | 29029
 Mt St Helens  | volcano  | 29029
 Denali        | mountain | 17000
 Kilimanjaro   | mountain | 14000
 Mt Washington | mountain |
(5 rows)
```

You have now created a flex table and loaded data. Next, learn more about using flex table data in your database.

Query More of Your Flex Table

1. Query your flex table to see the data you loaded as it is stored in the `__raw__` column. The example illustrates the table contents, with Return characters added for illustration:

```
=> \x
Expanded display is on.
=> SELECT * from mountains;
[ RECORD 1 ]+---------------------------------------------------------------------------
__identity__ | 1
__raw__      | \001\000\000\000,\000\000\000\004\000\000\000\024\000\000\000\031\000\000\000\
035\000\000\000$\000\000\0002902934.1Everestmountain\004\000\000\000\024\000\000\000\032\000\
000\000%\000\000\000)\000\000\000heighthike_safetynametype
[ RECORD 2 ]+---------------------------------------------------------------------------
__identity__ | 2
__raw__      | \001\000\000\0000\000\000\000\004\000\000\000\024\000\000\000\031\000\000\000\
035\000\000\000)\000\000\0002902915.4Mt St
Helensvolcano\004\000\000\000\024\000\000\000\032\000\
000\000%\000\000\000)\000\000\000heighthike_safetynametype
[ RECORD 3 ]+---------------------------------------------------------------------------
__identity__ | 3
__raw__      | \001\000\000\000+\000\000\000\004\000\000\000\024\000\000\000\031\000\000\000\

035\000\000\000#\000\000\0001700012.2Denalimountain\004\000\000\000\024\000\000\000\032\000\00
0
\000%\000\000\000)\000\000\000heighthike_safetynametype
[ RECORD 4 ]+---------------------------------------------------------------------------
__identity__ | 4
__raw__      | \001\000\000\000(\000\000\000\003\000\000\000\020\000\000\000\025\000\000\000\
000\000\00014000Kilimanjaromountain\003\000\000\000\020\000\000\000\026\000\000\000\032\000\
000\000heightnametype
[ RECORD 5 ]+---------------------------------------------------------------------------
__identity__ | 5
__raw__      | \001\000\000\000)\000\000\000\003\000\000\000\020\000\000\000\024\000\000\000\
000\000\00050.6Mt Washingtonmountain\003\000\000\000\020\000\000\000\033\000\000\000\037\000\
000\000hike_safetynametype
```

2. Use the `mapToString()` function (with the `__raw__` column of `mountains`) to inspect its contents in readable JSON text format:

```
=> SELECT maptostring(__raw__) from mountains;
                                         MAPTOSTRING
------------------------------------------------------------------------------------------
 {
        "hike_safety":  "50.6",
        "name": "Mt Washington",
        "type": "mountain"
```

```
}
{
       "height":       "29029",
       "hike_safety":  "34.1",
       "name": "Everest",
       "type": "mountain"
}
{
       "height":       "14000",
       "hike_safety":  "22.8",
       "name": "Kilimanjaro",
       "type": "mountain"
}
{
       "height":       "29029",
       "hike_safety":  "15.4",
       "name": "Mt St Helens",
       "type": "volcano"
}
{
       "height":       "17000",
       "hike_safety":  "12.2",
       "name": "Denali",
       "type": "mountain"
}
```

3. Now, use the `compute_flextable_keys` function to populate the `mountain_keys` table.Vertica generates this table automatically when you create your flex table.

```
=> SELECT compute_flextable_keys('mountains');
       compute_flextable_keys
---------------------------------------------------
Please see public.mountains_keys for updated keys
(1 row)
```

4. Query the keys table `mountains_keys`), and examine the results:

```
=> SELECT * from public.mountains_keys;
  key_name   | frequency | data_type_guess
-------------+-----------+-----------------
 hike_safety |         5 | varchar(20)
 name        |         5 | varchar(26)
 type        |         5 | varchar(20)
 height      |         4 | varchar(20)
(4 rows)
```

Build a Flex Table View

1. Use the `build_flextable_view` function to populate a view generated from the `mountains_keys` table.

```
=> SELECT build_flextable_view('mountains');
                 build_flextable_view
------------------------------------------------------
 The view public.mountains_view is ready for querying
(1 row)
```

2. Query the view `mountains_view`:

```
=> SELECT * from public.mountains_view;
 hike_safety |     name      |   type   | height
-------------+---------------+----------+--------
 50.6        | Mt Washington | mountain |
 34.1        | Everest       | mountain | 29029
 22.8        | Kilimanjaro   | mountain | 14000
 15.4        | Mt St Helens  | volcano  | 29029
 12.2        | Denali        | mountain | 17000
(5 rows)
```

3. Use the `view_columns` system table to query the `column_name` and `data_type` columns for `mountains_view`:

```
=> SELECT column_name, data_type from view_columns where table_name = 'mountains_view';
 column_name |  data_type
-------------+-------------
 hike_safety | varchar(20)
 name        | varchar(26)
 type        | varchar(20)
 height      | varchar(20)
(4 rows)
```

4. Review the query results:

   - Notice the `data_type` column, its values and sizes. These are calculated when you compute keys for your flex table with `compute_flextable_keys()`.
   - Did you also notice the `data_type_guess` column when you queried the `mountains_keys` table after invoking that function?

5. With the `data_type` information from `mountains_view`, override the `data_type_guess` for `hike_safety`. Then, `COMMIT` the change, and rebuild the view with

build_flextable_view():

```
=> UPDATE mountains_keys SET data_type_guess = 'float' where key_name = 'hike_safety';
 OUTPUT
--------
      1
(1 row)

=> commit;
=> SELECT build_flextable_view('mountains');
              build_flextable_view
------------------------------------------------------
 The view public.mountains_view is ready for querying
(1 row)
```

6. Next, use the `view_columns` system table. Notice that `hike_safety` is now cast to a float data type:

```
=> SELECT column_name, data_type from view_columns where table_name = 'mountains_view';
 column_name |  data_type
-------------+-------------
 hike_safety | float
 name        | varchar(26)
 type        | varchar(20)
 height      | varchar(20)
(4 rows)
```

Create a Hybrid Flex Table

If you already know that some of the data you load and query regularly needs full Vertica performance and support, you can create a *hybrid* flex table. A hybrid flex table has one or more real columns that you define, and a `__raw__` column to store any unstructured data you load. Querying real columns is faster than querying flexible data in the __raw__ column. You can define default values for the columns.

1. Create a hybrid flex table, and load the same sample JSON file:

```
=> CREATE flex table mountains_hybrid(name varchar(41) default name::varchar(41), hike_safety
float
default hike_safety::float);
=> COPY mountains_hybrid from '/home/dbadmin/Downloads/mountains.json' parser fjsonparser();
 Rows Loaded
-------------
           5
(1 row)
```

2. Use the `compute_flextable_keys_and_build_view` function to populate the

keys table and build the view for `mountains_hybrid`:

```
=> SELECT compute_flextable_keys_and_build_view('mountains_hybrid');
                                        compute_flextable_keys_and_build_view

----------------------------------------------------------------------------------------------
--
 Please see public.mountains_hybrid_keys for updated keys
The view public.mountains_hybrid_view is ready for querying
(1 row)
```

3. Query the `mountains_hybrid` keys table. Review the `data_type_guesses` column values again. The types list the column definitions you declared when you created the hybrid table:

```
=> SELECT * from mountains_hybrid_keys;
  key_name   | frequency | data_type_guess
-------------+-----------+-----------------
 height      |         4 | varchar(20)
 name        |         5 | varchar(41)
 type        |         5 | varchar(20)
 hike_safety |         5 | float
(4 rows)
```

If you create a basic flex table, and later find you want to promote one or more virtual columns to real columns, see Materializing Flex Tables to add columns.

Promote Virtual Columns in a Hybrid Flex Table

After you explore your flex table data, you can promote one ore more virtual columns in your flex table to real columns. You do not need to create a separate columnar table.

1. Invoke the `materialize_flextable_columns` function on the hybrid table, specifying the number of virtual columns to materialize:

```
=> SELECT materialize_flextable_columns('mountains_hybrid', 3);
                                 materialize_flextable_columns

---------------------------------------------------------------------------------------------
 The following columns were added to the table public.mountains_hybrid:
        type
For more details, run the following query:
SELECT * FROM v_catalog.materialize_flextable_columns_results WHERE table_schema = 'public'
and
```

```
table_name = 'mountains_hybrid';

(1 row)
```

2. You specified three (`3`) columns to materialize, but the table was created with two real columns (`name` and `hike_safety`). To fulfill your three-column specification, the function promotes only one other column, `type`. The next example has expanded display to list the columns vertically. Notice the `ADDED` status for the column that was just materialized, rather than `EXISTS` for the two columns you defined when creating the table:

```
=> \x
Expanded display is on.
=> SELECT * from materialize_flextable_columns_results where table_name = 'mountains_hybrid';
-[ RECORD 1 ]-+-------------------------------------------------------
table_id      | 45035996273766044
table_schema  | public
table_name    | mountains_hybrid
creation_time | 2013-11-30 20:09:37.765257-05
key_name      | type
status        | ADDED
message       | Added successfully
-[ RECORD 2 ]-+-------------------------------------------------------
table_id      | 45035996273766044
table_schema  | public
table_name    | mountains_hybrid
creation_time | 2013-11-30 20:09:37.765284-05
key_name      | hike_safety
status        | EXISTS
message       | Column of same name already exists in table definition
-[ RECORD 3 ]-+-------------------------------------------------------
table_id      | 45035996273766044
table_schema  | public
table_name    | mountains_hybrid
creation_time | 2013-11-30 20:09:37.765296-05
key_name      | name
status        | EXISTS
message       | Column of same name already exists in table definition
```

3. Now, display the hybrid table definition, listing the `__raw__` column and the three materialized columns. Flex table data types are derived from the associated keys tables, so you can update them as necessary. Notice that the `__raw__` column has a default `NOT NULL` constraint:

```
=> \d mountains_hybrid
List of Fields by Tables
-[ RECORD 1 ]---------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | __raw__
Type        | long varbinary(130000)
```

```
Size        | 130000
Default     |
Not Null    | t
Primary Key | f
Foreign Key |
-[ RECORD 2 ]-------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | name
Type        | varchar(41)
Size        | 41
Default     | (MapLookup(mountains_hybrid.__raw__, 'name'))::varchar(41)
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 3 ]-------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | hike_safety
Type        | float
Size        | 8
Default     | (MapLookup(mountains_hybrid.__raw__, 'hike_safety'))::float
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 4 ]-------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | type
Type        | varchar(20)
Size        | 20
Default     | (MapLookup(mountains_hybrid.__raw__, 'type'))::varchar(20)
Not Null    | f
Primary Key | f
Foreign Key |
```

You have now completed getting started with flex table basics, hybrid flex tables, and using the flex functions.

Understanding Flex Tables

You can create flex tables and then manage them with their associated helper, data, and map functions. Flex tables:

- Do not require schema definitions
- Do not need column definitions
- Have full Unicode support
- Support SQL queries

You can use flex tables to promote data directly from exploration to analytic operations. flex tables features include:

- Ability to load different formats into one flex table, which lets you handle changing structure over time
- Full support of delimited and JSON data
- Extensive SQL queries and built-in analytics for the data you load
- Usability functions, which let you explore your unstructured data and then use built-in functions to materialize the data

Exploration to Promotion

After you create a flex table, you can quickly load data, including social media content in JSON, log files, delimited files, and other information. Previously, working with such data required significant schema design and preparation. Now, you can load and query flex tables in a few steps.

Creating flex tables is similar to creating other tables, except column definitions are optional. When you create flex tables, with or without column definitions, Vertica implicitly adds a real column to your table, called `__raw__`. This column stores loaded data. The `__raw__` column is a `LONG VARBINARY` column with a `NOT NULL` constraint. It contains the documented limits for its data type (see Long Data Types in the SQL Reference Manual. The `__raw__` column's default maximum width is 130,000 bytes (with an absolute maximum of 32,000,000 bytes). You can change the default width with the `FlexTablesRawSize` configuration parameter.

If you create a flex table without other column definitions, the table includes a second default column, `__identity__`, declared as an auto-incrementing IDENTITY (1,1) column. When no other columns are defined, flex tables use the `__identity__` column for segmentation and sort order.

Loading data into a flex table encodes the record into a VMap type and populates the `__raw__` column. The VMap is a standard dictionary type, pairing keys with string values as virtual columns.

Flex Table Terms

This guide uses the following terms when describing how you work with flex tables to explore and analyze flexible data:

- VMap: An internal map data format.
- Virtual Columns: Undeclared columns that have not been promoted to real columns.
- Real Columns: Fully featured columns in flex or columnar tables.
- Promoted Columns : Virtual columns that have been materialized to real columns.
- Map Keys: Map keys are the virtual column names within VMap data.

Is There Structure in a Flex Table?

The term *unstructured data* (sometimes called *semi-structured* or *Dark Data*) does not indicate that the data you load into flex tables is entirely without structure. However, you may not know the data's composition or the inconsistencies of its design. In some cases, the data may not be relational.

Your data may have some structure (like JSON and delimited data). Data may be semi-structured or stringently structured, but in ways that you either do not know about or do not expect. In this guide, the term *flexible data* encompasses these and other kinds of data. You can load your flexible data directly into a flex table, and query its contents with your favorite SQL SELECT or other statements.

To summarize, you can load data first, without knowing its structure, and then query its content after a few simple transformations. In some cases, you already know the data structure, such as some tweet map keys, like `user.lang, user.screen_name`, and `user.url`. If so, you can query these values explicitly as soon as you load the data.

Making Flex Table Data Persist

The underlying implementation of each flex table is one (or two) real columns. Because of this design, existing Vertica functionality writes the table and its contents to disk

(ROS). This approach maintains K-safety in your cluster and supports standard recovery processes should node failures occur. Flex tables are included in full backups (or, if you choose, in object-level backups).

What Happens When You Create Flex Tables?

Whenever you execute a `CREATE FLEX TABLE` statement, Vertica creates three objects, as follows:

- The flexible table (*flex_table*)
- An associated keys table (*flex_table*_keys)
- A default view for the main table (*flex_table*_view)

The _keys and _view objects are dependents of the parent, *flex_table*. Dropping the flex table also removes its dependents, although you can drop the `_keys` or `_view` objects independently.

You can create a flex table without specifying any column definitions (such as`darkdata`, in the next example). When you do so, Vertica automatically creates two tables, the named flex table (such as `darkdata`) and its associated keys table, `darkdata_keys`:

```
=> create flex table darkdata();
CREATE TABLE
=> \dt dark*
                List of tables
 Schema |     Name      | Kind  |  Owner  | Comment
--------+---------------+-------+---------+---------
 public | darkdata      | table | dbadmin |
 public | darkdata_keys | table | dbadmin |
(2 rows)
```

Each flex table has two default columns, `__raw__` and `__identity__`. The `__raw__` column exists in every flex table to hold the data you load. The `__identity__` column is auto-incrementing. Vertica uses the `__identity__` column for segmentation and sort order when no other column definitions exist. The flex keys table (`darkdata_keys`) has three columns, as shown:

```
=> select * from darkdata;
 __identity__ | __raw__
--------------+---------
(0 rows)

=> select * from darkdata_keys;
```

```
 key_name | frequency | data_type_guess
----------+-----------+-----------------
(0 rows)
```

Creating a flex table with column definitions (such as `darkdata1`, in the next example) automatically generates a table with the `__raw__` column. However, the table has no `__identity__` column because columns are specified for segmentation and sort order. Two tables are created automatically, as shown in the following example:

```
=> create flex table darkdata1 (name varchar);
CREATE TABLE

=> select * from darkdata1;
 __raw__ | name
---------+------
(0 rows)
=> \d darkdata1*
                                             List of Fields by Tables
 Schema |   Table   | Column  |          Type          |  Size  | Default | Not Null | Primary Key
| Foreign Key
--------+-----------+---------+------------------------+--------+---------+----------+------------
+-------------
 public | darkdata1 | __raw__ | long varbinary(130000) | 130000 |         | t        | f
|
 public | darkdata1 | name    | varchar(80)            |     80 |         | f        | f
|
(2 rows)
=> \dt darkdata1*
                   List of tables
 Schema |      Name      | Kind  |  Owner  | Comment
--------+----------------+-------+---------+---------
 public | darkdata1      | table | dbadmin |
 public | darkdata1_keys | table | dbadmin |
(2 rows)
```

Creating a flex table with at least one column definition (`darkdata1` in the next example) also generatesa table with the `__raw__` column, but not an `__identity__` column. Instead, the specified columns are used for segmentation and sort order. Two tables are also created automatically, as shown in the following example:

```
=> create flex table darkdata1 (name varchar);
CREATE TABLE
=> \d darkdata1*
                               List of Fields by Tables
 Schema |   Table   | Column  |          Type          |  Size  | Default | Not Null | Primary Key
| Foreign Key
--------+-----------+---------+------------------------+--------+---------+----------+------------
+-------------
 public | darkdata1 | __raw__ | long varbinary(130000) | 130000 |         | t        | f
|
 public | darkdata1 | name    | varchar(80)            |     80 |         | f        | f
|
(2 rows)
```

```
=> \dt darkdata1*
                   List of tables
 Schema |      Name      | Kind  |  Owner  | Comment
--------+----------------+-------+---------+---------
 public | darkdata1      | table | dbadmin |
 public | darkdata1_keys | table | dbadmin |
(2 rows)
```

For more examples, see Creating Flex Tables.

Creating Superprojections Automatically

In addition to creating two tables for each flex table, Vertica creates superprojections for both the main flex table and its associated keys table. Using the `\dj` command, you can display the projections created automatically for the `darkdata` and `darkdata1` tables in this set of examples:

```
=> \dj darkdata*
                           List of projections
 Schema |          Name          |  Owner  |       Node       | Comment
--------+------------------------+---------+------------------+---------
 public | darkdata1_b0           | dbadmin |                  |
 public | darkdata1_b1           | dbadmin |                  |
 public | darkdata1_keys_super   | dbadmin | v_vmart_node0001 |
 public | darkdata1_keys_super   | dbadmin | v_vmart_node0003 |
 public | darkdata1_keys_super   | dbadmin | v_vmart_node0004 |
 public | darkdata_b0            | dbadmin |                  |
 public | darkdata_b1            | dbadmin |                  |
 public | darkdata_keys__super   | dbadmin | v_vmart_node0001 |
 public | darkdata_keys_super    | dbadmin | v_vmart_node0003 |
 public | darkdata_keys_super    | dbadmin | v_vmart_node0004 |
(10 rows)
```

Note: You cannot create pre-join projections from flex tables. For information about projections and how they are used, see Vertica Concepts.

Default Flex Table View

When you create a flex table, you also create a default view. This default view uses the table name with a `_view` suffix, as listed in the next example, which shows the list of views for darkdata and darkdata1. If you query the default view, you are prompted to use the COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW function. This view enables you to update the view after you load data so that it includes all keys and values.

```
=> \dv darkdata*
                  List of View Fields
 Schema |      View      | Column |     Type      | Size
--------+----------------+--------+---------------+------
```

```
 public | darkdata_view  | status | varchar(124) |  124
 public | darkdata1_view | status | varchar(124) |  124
(2 rows)
```

For more information, see Updating Flex Table Views.

Flex Functions

There are three sets of functions to support flex tables and extracting data into VMaps. See the following sections for more information:

- Data (helper) functions (Flex Data Functions Reference)
- Extractor functions (Flex Extractor Functions Reference)
- Map functions (Flex Map Functions Reference)

Using Clients with Flex Tables

You can use the Vertica supported client drivers with flex tables as follows:

- Flex tables do not permit `INSERT` statements from vsql or from any client. To load data from a client, use `COPY LOCAL` with the appropriate flex table parser.
- The driver metadata APIs return only real columns. For example, using a `select * from myflex;` statement, when *myflex* has a single materialized column (`name`), returns the `__raw__` and `name` columns. However, it does not return virtual columns from within `__raw__`. To access virtual columns and their values, query the associated `flextable_keys` table, just as you would in vsql.

Creating Flex Tables

You can create a flex table, or an external flex table, without column definitions or other parameters. You can use any CREATE TABLE statement parameters you prefer, as usual.

Unsupported CREATE FLEX TABLE Statements

These statements are not currently supported:

- `CREATE FLEX TABLE AS...`
- `CREATE FLEX TABLE LIKE...`

Creating Basic Flex Tables

Here's how to create the table:

```
=> create flex table darkdata();
CREATE TABLE
```

Selecting from the table before loading any data into it reveals its two real columns, `__identity__` and `__raw__`:

```
=> select * from darkdata;
__identity__     | __raw__
--------------+---------
(0 rows)
```

Here's an example of creating a flex table with a column definition:

```
=> create flex table darkdata1(name varchar);
CREATE TABLE
```

When flex tables exist, you can add new columns (including those with default derived expressions), as described in Altering Flex Tables.

Creating Temporary Flex Tables

Before you create temporary global and local flex tables, be aware of the following considerations:

- GLOBAL TEMP flex tables are supported. Creating a temporary global flex table results in the `flextable_keys` table and the `flextable_view` having temporary table restrictions for their content.
- LOCAL TEMP flex tables must include at least one column definition. The reason for this requirement is that local temp tables do not support automatically-incrementing data (such as the flex table default `__identity__` column). Creating a temporary local flex table results in the `flextable_keys` table and the `flextable_view` existing in the local temporary object scope.
- LOCAL TEMP views are supported for flex and columnar temporary tables.

For global or local temp flex tables to function correctly, you must also specify the `ON COMMIT PRESERVE ROWS` clause. You must use the `ON COMMIT` clause for the flex table helper functions, which rely on commits. Create a local temp table as follows:

```
=> create flex local temp table good(x int) ON COMMIT PRESERVE ROWS;
CREATE TABLE
```

After creating a local temporary flex table using this approach, you can then load data into the table, create table keys, and a flex table view:

```
=> copy good from '/home/release/KData/bake.json' parser fjsonparser();
 Rows Loaded
-------------
           1
(1 row)

=> select compute_flextable_keys_and_build_view('good');
                            compute_flextable_keys_and_build_view
----------------------------------------------------------------------------------------------
 Please see v_temp_schema.good_keys for updated keys
The view good_view is ready for querying
(1 row)
```

However, creating temporary flex tables without an `ON COMMIT PRESERVE ROWS` clause results in the following warnings:

```
=> create flex local temp table bak1(id int, type varchar(10000), name varchar(1000));
WARNING 5860:  Due to the data isolation of temp tables with an on-commit-delete-rows
policy, the compute_flextable_keys() and compute_flextable_keys_and_build_view() functions
cannot access this table's data. The build_flextable_view() function can be used with a
user-provided keys table to create  a view, but involves a DDL commit which will delete
the table's rows
```

After loading data into a such a temporary flex table, computing keys or building a view for the flex table results in the following error:

```
=> select compute_flextable_keys('bak1');
ERROR 5859:  Due to the data isolation of temp tables with an on-commit-delete-rows policy,
```

```
the compute_flextable_keys() and compute_flextable_keys_and_build_view() functions cannot
access this table's data
HINT:  Make the temp table ON COMMIT PRESERVE ROWS to use this function
```

Similarly, you can create global temp tables as follows:

```
=> create flex global temp table good_global(x int) ON COMMIT PRESERVE ROWS;
```

After creating a global temporary flex table using this approach, you can then load data into the table, create table keys, and a flex table view:

```
=> copy good_global from '/home/dbadmin/data/flex/bake_single.json' parser fjsonparser();
Rows Loaded
-------------
5
(1 row)

=> select compute_flextable_keys_and_build_view('good_global');
                            compute_flextable_keys_and_build_view
----------------------------------------------------------------------------------------------
 Please see v_temp_schema.good_keys for updated keys
The view good_view is ready for querying
(1 row)
```

Similar to local temp flex tables, creating global flex tables without an `ON COMMIT PRESERVE ROWS` clause results in the following warnings:

```
=> create flex global temp table bak_global(id int, type varchar(10000), name varchar(1000));
WARNING 5860:  Due to the data isolation of temp tables with an on-commit-delete-rows
policy, the compute_flextable_keys() and compute_flextable_keys_and_build_view() functions
cannot access this table's data. The build_flextable_view() function can be used with a
user-provided keys table to create  a view, but involves a DDL commit which will delete
the table's rows
CREATE TABLE
```

Loading data into a such a temporary flex table, computing keys or building a view for the flex table results in the following error:

```
=> select compute_flextable_keys('bak_global');
ERROR 5859:  Due to the data isolation of temp tables with an on-commit-delete-rows policy, the
compute_flextable_keys() and
compute_flextable_keys_and_build_view() functions cannot access this table's data
HINT:  Make the temp table ON COMMIT PRESERVE ROWS to use this function
```

Materializing Flex Table Virtual Columns

After you create your flex table and load data, you compute keys from virtual columns. After completing those tasks, you can materialize some keys by promoting virtual columns to real table columns. By promoting virtual columns, you query real columns rather than the raw data.

You can promote one or more virtual columns — materializing those keys from within the `__raw__` data to real columns. Vertica recommends this approach to get the best query performance for all important keys. You don't need to create new columnar tables from your flex table.

Promoting flex table columns results in a hybrid table. Hybrid tables:

- Maintain the convenience of a flex table for loading unstructured data
- Improve query performance for any real columns

If you have only a few columns to materialize, try altering your flex table progressively, adding columns whenever necessary. You can use the `ALTER TABLE...ADD COLUMN` statement to do so, just as you would with a columnar table. See Altering Flex Tables for ideas about adding columns.

If you want to materialize columns automatically, use the helper function MATERIALIZE_FLEXTABLE_COLUMNS

Creating Columnar Tables from Flex Tables

You can create a regular columnar Vertica table from a flex table, but you cannot use one flex table to create another.

Typically, you create a columnar table from a flex table after loading data. Then, you specify the virtual column data you want in a regular table, casting virtual columns to regular data types.

To create a columnar table from a flex table, `darkdata`, select two virtual columns, (`user.lang` and `user.name`), for the new table. Use a command such as the following, which casts both columns to `varchars` for the new table:

```
=> create table darkdata_full as select "user.lang"::varchar, "user.name"::varchar from darkdata;
CREATE TABLE
=> select * from darkdata_full;
 user.lang |      user.name
-----------+---------------------
 en        | Frederick Danjou
 en        | The End
 en        | Uptown gentleman.
 en        | ~G A B R I E L A â¿
 es        | Flu Beach
 es        | I'm Toasterâ¥
 it        | laughing at clouds.
 tr        | seydo shi
           |
           |
           |
           |
(12 rows)
```

Creating External Flex Tables

To create an external flex table:

```
=> create flex external table mountains() as copy from 'home/release/KData/kmm_ountains.json'
parser fjsonparser();
CREATE TABLE
```

As with other flex tables, creating an external flex table produces two regular tables: the named table and its associated `_keys` table. The keys table is not an external table:

```
=> \dt mountains
                 List of tables
 Schema |   Name    | Kind  |  Owner  | Comment
--------+-----------+-------+---------+---------
 public | mountains | table | release |
(1 row)
```

You can use the helper function, COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW, to compute keys and create a view for the external table:

```
=> select compute_flextable_keys_and_build_view ('appLog');

                     compute_flextable_keys_and_build_view
--------------------------------------------------------------------------------------------------
Please see public.appLog_keys for updated keys
The view public.appLog_view is ready for querying
(1 row)
```

1. Check the keys from the `_keys` table for the results of running the helper application:

```
=> select * from appLog_keys;
                          key_name                           | frequency |   data_type_guess
-------------------------------------------------------------+-----------+------------------
contributors                                                 |         8 | varchar(20)
coordinates                                                  |         8 | varchar(20)
created_at                                                   |         8 | varchar(60)
entities.hashtags                                            |         8 | long varbinary(186)
.
.
.
retweeted_status.user.time_zone                              |         1 | varchar(20)
retweeted_status.user.url                                    |         1 | varchar(68)
retweeted_status.user.utc_offset                             |         1 | varchar(20)
retweeted_status.user.verified                               |         1 | varchar(20)
(125 rows)
```

2. Query from the external flex table view:

```
=> select "user.lang" from appLog_view;
```

```
 user.lang
-----------
it
en
es
en
en
es
tr
en
(12 rows)
```

Note: External tables are fully supported for both flex and columnar tables. However, using external flex (or columnar) tables is less efficient than using flex tables whose data is stored in the Vertica database. Data that is maintained externally requires reloading each time you query..

Partitioning Flex Tables

You cannot partition a flex table on any virtual column (key).

The next example shows a query on `user.location`, which is a virtual column in the map data. The example thenattempts tp partition that column:

```
=> select "user.location" from darkdata;
 user.location
---------------
 chicago
 Narnia
 Uptown..
 Chile
(12 rows)
=> alter table darkdata partition by "user.location" reorganize;
ROLLBACK 5371:  User defined function not allowed: MapLookup
```

Using COPY with Flex Tables

You load data into a flex table with a COPY statement, specifying one of the flex parsers:

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FJSONPARSER
- FREGEXPARSER

All flex parsers store the data as a single-value VMap. They reside in the VARBINARY __raw__ column, which is a real column with a NOT NULL constraint. The VMap is encoded into a single binary value for storage in the __raw__ column. The encoding places the value strings in a contiguous block, followed by the key strings. Vertica supports null values within the VMap for keys with NULL-specified columns. The key and value strings represent the virtual columns and their values in your flex table.

If a flex table data row is too large to fit in the VARBINARY __raw__ column, it is rejected. By default, the rejected data and exceptions files are stored in the standard CopyErrorLogs location, a subdirectory of the catalog directory:

```
v_mart_node003_catalog\CopyErrorLogs\trans-STDIN-copy-from-exceptions.1
v_mart_node003_catalog\CopyErrorLogs\trans-STDIN-copy-rejections.1
```

Flex tables do not copy any rejected data, due to disk space considerations. The rejected data file exists, but it contains only a new line character for every rejected record. The corresponding exceptions file lists the reason why each record was rejected.

You can specify a different path and file for the rejected data and exceptions files. To do so, use the COPY parameters REJECTED DATA and EXCEPTIONS, respectively. You can also save load rejections and exceptions in a table. See the *Bulk Loading* section of the Administrator's Guide, (*Capturing Load Rejections and Exceptions*).

Basic Flex Table Load and Query

Loading data into your flex table is similar to loading data into a regular columnar table. The difference is that you must use the `parser` argument with one of the flex parsers:

```
=> copy darkdata from '/home/dbadmin/data/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

Note: You can use many additional COPY parameters as required but not all are supported.

Loading Data into Flex Table Real Columns

If you create a hybrid flex table with one or more real column definitions, COPY evaluates each virtual column key name during data load. For each real column with a name that is identical to a virtual column key name, COPY does the following:

- Loads the keys and values as part of the VMap data in the `__raw__` column
- Automatically populates real columns with the values from their virtual column counterparts

Subsequent data loads continue loading same-name key-value pairs into both the `__raw__` column and the real column.

Note: Over time, storing values in both column types can impact your licensed data limits. For more information about Vertica licenses, see Managing Licenses in the Administrator's Guide.

For example, continuing with the JSON data:

1. Create a flex table, `darkdata1`, with a column definition of one of the keys in the data you will load:

   ```
   => create flex table darkdata1 ("user.lang" varchar);
   CREATE TABLE
   ```

2. Load data into `darkdata1`:

```
=> copy darkdata1 from '/test/vertica/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

3. Query the `user.lang` column of `darkdata1`. Loading the JSON data file populated the column you defined:

```
=> select "user.lang" from darkdata1;
 user.lang
-----------
 es
 es
 tr
 it
 en
 en
 en
 en
(12 rows)
```

   Empty column rows indicate NULL values. For more information about how NULLs are handled in flex table, see ().

4. You can query for other virtual columns (such as "`user.name`" in `darkdata1`), with similar results as for "`user.lang`":

```
=> select "user.name" from darkdata1;
      user.name
---------------------
 I'm Toasterâ¥
 Flu Beach
 seydo shi
 The End
 Uptown gentleman.
 ~G A B R I E L A â¿
 Frederick Danjou
 laughing at clouds.
(12 rows)
```

> **Note:** While the results for these two queries are similar, the difference in accessing the keys and their values is significant. Data for "`user.lang`" has been materialized into a real table column, while "`user.name`" remains a virtual column. For production-level data usage (rather than test data sets), materializing flex table data improves query performance significantly.

Handling Default Values During Loading

You can create your flex table with a real column, named for a virtual column that exists in your incoming data. For example, if the data you load has a `user.lang` virtual column, define the flex table with that column. You can also specify a default column value when creating the flex table with real columns. The next example shows how to define a real column (`user.lang`), which has a default value from a virtual column (`user.name`):

```
=> create flex table table darkdata1 ("user.lang" long varchar default "user.name");
```

When you load data into your flex table, COPY uses values from the flex table data, ignoring the default column definition. Loading data into a flex table requires MAPLOOKUP to find keys that match any real column names. A match exists when the incoming data has a virtual column with the same name as a real column. When COPY detects a match, it populates the column with values. COPY returns either a value or NULL for each row, so real columns always have values.

For example, after creating the `darkdata1` flex table, described in the previous example, load data with `COPY`:

```
=> copy darkdata1 from '/test/vertica/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

If you query the `darkdata1` table after loading, the data shows that the values for the `user.lang` column were extracted:

- From the data being loaded — Values for the `user.lang` virtual column
- With `NULL` — Rows without values

In this case, the table column default value for `user.lang` was ignored:

```
=> select "user.lang" from darkdata1;
 user.lang
-----------
 it
 en
 es
 en
 en
 es
 tr
 en
(12 rows)
```

Using COPY to Specify Default Column Values

You can add an expression to a COPY statement to specify default column values when loading data. For flex tables, specifying any column information requires that you list the `__raw__` column explicitly. The following example shows how to use an expression for the default column value. In this case, loading populates the defined `user.lang` column with data from the input data `user.name` values:

```
=> copy darkdata1(__raw__, "user.lang" as "user.name"::varchar)from
'/test/vertica/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
=> select "user.lang" from darkdata1;
      user.lang
---------------------
 laughing at clouds.
 Avita Desai
 I'm Toasterâ¥
 Uptown gentleman.
 ~G A B R I E L A â¿
 Flu Beach
 seydo shi
 The End
(12 rows)
```

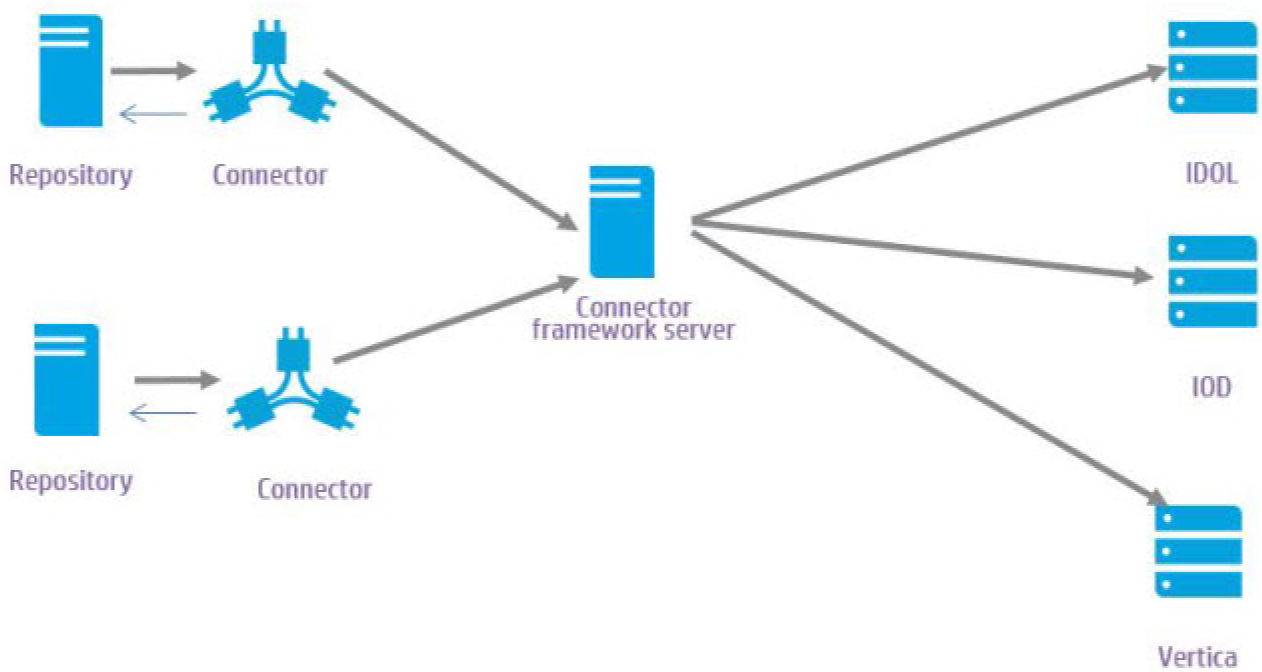
You can specify default values when adding columns, as described in Altering Flex Tables. When you do so, a different behavior results. For more information about using COPY, its expressions and parameters, see Bulk Loading Data in the Administrator's Guide and COPY in the SQL Reference Manual.

Using Flex Tables for IDOL Data

You can create flex tables to use with the HPE IDOL Connector Framework Server (CFS) and an ODBC client. The CFS VerticaIndexer module uses the connector to retrieve data. CFS then indexes the data into your Vertica database.

CFS supports many connectors for interfacing to different unstructured file types stored in repositories. Examples of repositories include Microsoft Exchange (email), file systems (including Word documents, images, and videos), Microsoft SharePoint, and Twitter (containing Tweets).

Connectors retrieve and aggregate data from repositories. CFS indexes the data, sending it to IDOL, IDOL OnDemand, or Vertica. The following figure illustrates a basic setup with a repository and a connector.



After you configure CFS and connect it to your Vertica database, the connector monitors the repository for changes and deletions to loaded documents, and for new files not previously added to the server. CFS then updates its server destinations automatically.

To achieve the best query results with ongoing CFS updates and deletes, HPE recommends using live aggregate projections and top-K projections. For more information about how these projections work, and for examples of using them, see Working with Projections in the Administrator's Guide.

ODBC Connection String for CFS

There are several steps to setting up the CFS VerticaIndexer to load IDOL metadata into your database. For a more complete example, see the blog post HPE IDOL CFS Vertica Module.

One of the first steps is to add information to the CFS configuration file. To do so, add an `Indexing` section to the configuration file that specifies the ODBC ConnectionString details.

Successfully loading data requires a valid database user with write permissions to the destination table. Two ODBC connection parameters (`UID` and `PWD`) specify the Vertica user and password. The following example shows a sample CFS `Indexing` section. The section includes a `ConnectionString` with the basic parameters, including a sample user (`UID=fjones`) and password (`PWD=fjones_password`):

```
[Indexing]
IndexerSections=vertica
IndexTimeInterval=30

[vertica]
IndexerType = Library
ConnectionString=Driver=Vertica;Server=123.456.478.900;Database=myDB;UID=fjones;PWD=fjones_password
TableName = marcomm.myFlexTable
LibraryDirectory = ./shared_library_indexers
LibraryName = verticaIndexer
```

For more information about ODBC connection parameters, see ODBC Configuration Parameters.

CFS COPY LOCAL Statement

CFS first indexes and processes metadata from a document repository to add to your database. Then, CFS uses the Indexing information you added to the configuration file to create an ODBC connection. After establishing a connection, CFS generates a standard `COPY LOCAL` statement, specifying the `fjsonparser`. CFS loads data directly into your pre-existing flex table with a statement such as the following:

```
=> COPY myFlexTable FROM LOCAL path_to_compressed_temporary_json_file parser fjsonparser();
```

```
=> select * from myavro;
 __identity__ | __raw__
--------------+---------
(0 rows)
```

When your IDOL metadata appears in a flex table, you can optionally add new table columns, or materialize other data, as described in Altering Flex Tables.

Using Flex Table Parsers

You can load flex tables with one of several parsers. You can load data using the options that the flex parsers support:

- Loading Avro Data
- Loading Common Event Format (CEF) Data
- Loading CSV Data
- Loading Delimited Data
- Loading JSON Data
- Loading Matches from Regular Expressions

Using Flex Parsers for Columnar Tables

You can use any of the flex parsers to load data into columnar tables. Using the flex table parsers to load columnar tables gives you the capability to mix data loads in one table. For example, you can load JSON data in one session and delimited data in another.

> **Note:** For Avro data, you can load only data into a columnar table, not the schema. For flex tables, Avro schema information is required to be embedded in the data.

The following basic examples illustrate how you can use flex parsers with columnar tables.

1. Create a columnar table, `super`, with two columns, `age` and `name`:

```
=> create table super(age int, name varchar);
CREATE TABLE
```

2. Enter JSON values from STDIN, using the `fjsonparser()`.

```
=> copy super from stdin parser fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"age": 5, "name": "Tim"}
>>  {"age": 3}
>>  {"name": "Fred"}
>>  {"name": "Bob", "age": 10}
>> \.
```

3. Query the table to see the values you entered:

```
=> select * from super;
 age | name
-----+------
     | Fred
  10 | Bob
   5 | Tim
   3 |
(4 rows)
```

4. Enter some delimited values from STDIN, using the `fdelimitedparser()`:

```
=> copy super from stdin parser fdelimitedparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> name |age
>> Tim|50
>> |30
>> Fred|
>> Bob|100
>> \.
```

5. Query the flex table. Both JSON and delimited data are saved in the same columnar table, `super`.

```
=> select * from super;
 age | name
-----+------
  50 | Tim
  30 |
   3 |
   5 | Tim
 100 | Bob
     | Fred
  10 | Bob
     | Fred
(8 rows)
```

Use the `reject_on_materialized_type_error` parameter to avoid loading data with type mismatch. If `reject_on_materialized_type_error` is set to `false`, the flex parser will accept the data with type mismatch. Consider the following example:

Assume that the CSV file to be loaded has the following sample contents:

```
$ cat json.dat
{"created_by":"system","site_source":"flipkart_india_kol","updated_by":"system1","invoice_
id":"INVDPKOL100","vendor_id":"VEN15731","total_quantity":12,"created_at":"2012-01-09 23:15:52.0"}
{"created_by":"system","site_source":"flipkart_india_kol","updated_by":"system2","invoice_
id":"INVDPKOL101","vendor_id":"VEN15732","total_quantity":14,"created_at":"hello"}
```

1. Create a columnar table.

```
=> CREATE TABLE hdfs_test (
site_source varchar(200),
total_quantity int ,
vendor_id varchar(200),
invoice_id varchar(200),
updated_by varchar(200),
created_by varchar(200),
created_at timestamp
);
```

2. Load JSON data.

```
=>COPY hdfs_test from '/home/dbadmin/json.dat' parser fjsonparser() ABORT ON ERROR;
Rows Loaded
-------------
2
(1 row)
```

3. View the contents.

```
=> select * from hdfs_test;
site_source | total_quantity | vendor_id | invoice_id | updated_by | created_by | created_at
--------------------+----------------+-----------+-------------+------------+------------+----
-----------------
flipkart_india_kol | 12 | VEN15731 | INVDPKOL100 | system1 | system | 2012-01-09 23:15:52
flipkart_india_kol | 14 | VEN15732 | INVDPKOL101 | system2 | system |
(2 rows)
```

4. If `reject_on_materialized_type_error` parameter is set to `true`, you will receive errors when loading the sample JSON data.

```
=> COPY hdfs_test from '/home/dbadmin/data/flex/json.dat' parser fjsonparser(reject_on_
materialized_type_error=true) ABORT ON ERROR;
ERROR 2035:  COPY: Input record 2 has been rejected (Rejected by user-defined parser)
```

Loading Avro Data

You can load Avro data files into flex tables and columnar tables using the parser, `favroparser`. Before loading, verify that Avro files are encoded in the Avro binary serialization encoding format, described in the Apache Avro standard. The parser also supports Snappy compression. You cannot load Avro data directly from STDIN.

> **Note:** The parser `favroparser` does not support Avro files with separate schema files. The Avro file must have its related schema in the file you are loading.

You can use the following data types and optional parameters for `favroparser`.

The `favroparser` supports two data types:

- Primitive Data Types for favroparser
- Complex Data Types for favroparser

Rejecting Data on Materialized Column Type Errors

The `favroparser` has a Boolean parameter, `reject_on_materialized_type_error`. If you set this parameter to `true`, Vertica rejects rows when the input data presents *both* of the following conditions:

- Includes keys matching an existing materialized column
- Has a value that cannot be coerced into the materialized column's data type

Suppose the flex table has a materialized column, `Temperature`, declared as a `FLOAT`. If you try to load a row with a `Temperature` key that has a `VARCHAR` value, `favroparser` rejects the data row.

See Also

- Using COPY with Kafka

Primitive Data Types for favroparser

The `favroparser` supports the following primitive data types:

AVRO Data Type	Vertica Data Type	Value
NULL	NULL	No value
boolean	BOOLEAN	A binary value
int	INTEGER	32-bit signed integer
long	INTEGER	64-bit signed integer
float	DOUBLE PRECISION (FLOAT) Synonymous with 64-bit IEEE FLOAT	Single precision

		(32-bit) IEEE 754 floating-point number
double	DOUBLE PRECISION (FLOAT)	Double precision (64-bit) IEEE 754 floating-point number
bytes	BYTES	Sequence of 8-bit unsigned bytes
string	VARCHAR	Unicode character sequence

Note: Vertica does not have an explicit 4-byte (32-bit integer) or smaller types. Instead, Vertica encoding and compression automatically eliminate the storage overhead of values that require less than 64 bits.

Vertica copies each primitive type into the `__raw__` column of the flex table. In this copy operation, the name of the primitive type becomes a virtual column key with its corresponding value as the value of the virtual column.

If the flex table has materialized columns, `favroparser` loads the primitive data type into the corresponding Vertica type for the column. If the parsing is successful, Vertica copies the data into the materialized column; otherwise, it rejects the row.

Complex Data Types for favroparser

You specify the data type of a record in the Avro file using the `type` parameter for `favroparser`. The `favroparser` supports these complex data types:

- Records
- Enums
- Arrays

- Maps
- Unions
- Fixed

This section describes attributes associated with the complex data types.

Records

Records have the following attributes:

Attribute	Description
name	A JSON string for the name of the record
fields	A JSON array used to list fields. Each field is a JSON object: • name: A JSON string for the name of the field • type: A JSON object used to define a schema or a JSON string used for naming a record definition

The `name` of each field is used as a virtual column name. If `flatten_records = true` and several nesting levels are present, Vertica concatenates the record names to create the `key_name`, as follows:

```
{
  "type": "record",
  "name": "Profile",
  "fields" : [
      {"name": "UserName", "type": "string"},
      {"name": "Address", "type": "string"}
   ]
}
```

```
{
  "type": "record",
  "name": "Profile",
  "fields" : [
      {VerticaUser},
      {VerticaUser Address}
]
}
```

Vertica creates virtual columns for the records as follows:

Names	Values
`UserName`	`VerticaUser`
`Address`	`VerticaUser Address`

Enums

Enums (enumerated values) use the type name `enum` and support the following attributes:

Attribute	Description
name	A JSON string for the name of the enum
symbols	A JSON array used to list symbols as JSON strings. All symbols in an enum must be unique and duplicates are prohibited

Example:

```
{
        "type": "enum",
        "name": "suit",
        "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
}
```

Consider the preceding Avro schema with a record that contains a field with the value `HEARTS`. In this case, the key value pair copied into the `__raw__` column has `suit` as the key and `HEARTS` as the value.

Arrays

Arrays use the type name `array` and support one attribute:

Attribute	Description
items	The schema of the array's items

For example, declare an array of strings:

```
{"type": "array", "items": "string"}
```

Similar to the capabilities for `Records`, you can nest and flatten `Arrays` using `flatten_arrays=true` :

```
{
       "__name__" : "Order",                            <-- artificial __name__ key for record
       "customer_id" : "111222",
       "order_details" : {                                    <-- array of records
       "0" : {                                                   <-- array index 0
               "__name__" : "OrderDetail",
               "product_detail" : {
                        "__name__" : "Product",
                        "price" : "46.21",
                        "product_category" : {           <- array of strings
                               "0" : "electronics",
                               "1" : "printers",
                               "2" : "computers"
                               },
               "product_name" : "hp printer X11ew",
               "product_status" : "ONLY_FEW_LEFT"
               }
       },
       "order_id" : "2389646",
       "total" : "132.43"
}
```

Here is the result of flattening the array:

```
{
       "0.order_details.__name__" : "OrderDetail",
       "0.order_details.product_detail.0.product_category" : "electronics",
       "0.order_details.product_detail.1.product_category" : "prnters",
       "0.order_details.product_detail.2.product_category" : "computers",
       "0.order_details.product_detail.__name__" : "Product",
       "0.order_details.product_detail.price" : "46.21",
       "0.order_details.product_detail.product_name" : "hp printer X11ew",
       "0.order_details.product_detail.product_status" : "ONLY_FEW_LEFT",
       "__name__" : "Order",
       "customer_id" : "111222",
       "order_id" : "2389646",
       "total" : "132.43"
}
```

Maps

Maps use the type name `map` and support one attribute:

Attribute	Description
values	The schema of the map's items

The `favroparser` treats map keys as strings. For example, you can declare the `map` type as a long as follows:

```
{"type": "map", "values": "long"}
```

Similar to `Records` types, `Maps` can also be nested and flattened using `flatten_ maps=true`.

The `favroparser` inserts key-value pairs from the Avro map as key-value pairs in the `__raw__` column. For an Avro record that has `KeyX` with value `10`, and `KeyY` with value `20`, `favroparser` loads the key-value pairs as virtual columns `KeyX` and `KeyY`, with values `10` and `20`, respectively.

Unions

Vertica uses JSON arrays to represent Avro `Unions`. Consider this example:

```
{"name":"TransactionID","type":["string","null"]}
```

The field `TransactionID` can be a `string` or `null`.

Fixed

Fixed (`fixed`) Avro types support two attributes:

Attribute	Description
name	A string for the name of this data type
size	An integer, specifying the number of bytes per value

For example, you can declare a 16-byte quantity:

```
{"type": "fixed", "size": 16, "name": "md5"}
```

With the preceding declaration is the Avro file schema, consider a record that contains a field with the following byte values for the key `md5`:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5]
```

The `favroparser` loads the key value pair as an `md5` key with the preceding byte values.

Loading Common Event Format (CEF) Data

Use the flex parser `fcefparser` to load HPE ArcSight or other Common Event Format (CEF) log file data into columnar and flexible tables. For more information, see the ArcSight Common Event Format (CEF) Guide.

When you use the parser to load arbitrary CEF-format files, it interprets key names in the data as virtual columns in your flex table. After loading, you can query your CEF data directly, regardless of which set of keys exist in each row. You can also use the associated flex table data and map functions to manage CEF data access.

Create a Flex Table and Load CEF Data

This section uses a sample set of CEF data. All IP addresses have been purposely changed to be inaccurate, and Return characters added for illustration.

To use this sample data, copy the following text and remove all Return characters. Save the file as `CEF_sample.cef`, which is the name used throughout these examples.

```
CEF:0|ArcSight|ArcSight|6.0.3.6664.0|agent:030|Agent [test] type [testalertng] started|Low|
eventId=1 mrt=1396328238973 categorySignificance=/Normal categoryBehavior=/Execute/Start
categoryDeviceGroup=/Application catdt=Security Mangement categoryOutcome=/Success
categoryObject=/Host/Application/Service art=1396328241038 cat=/Agent/Started
deviceSeverity=Warning rt=1396328238937 fileType=Agent
cs2=<Resource ID\="3DxKlG0UBABCAA0cXXAZIwA\=\="/> c6a4=fe80:0:0:0:495d:cc3c:db1a:de71
cs2Label=Configuration Resource c6a4Label=Agent
IPv6 Address ahost=SKEELES10 agt=888.99.100.1 agentZoneURI=/All Zones/ArcSight
System/Private Address Space
Zones/RFC1918: 888.99.0.0-888.200.255.255 av=6.0.3.6664.0 atz=Australia/Sydney
aid=3DxKlG0UBABCAA0cXXAZIwA\=\= at=testalertng dvchost=SKEELES10 dvc=888.99.100.1
deviceZoneURI=/All Zones/ArcSight System/Private Address Space Zones/RFC1918:
888.99.0.0-888.200.255.255 dtz=Australia/Sydney _cefVer=0.1
```

1. Create a flex table `logs`:

```
=> create flex table logs();
CREATE TABLE
```

2. Load the sample CEF file, using the flex parser `fcefparser`:

```
=> copy logs from '/home/dbadmin/data/CEF_sample.cef' parser fcefparser();
 Rows Loaded
-------------
           1
(1 row)
```

3. Use the `maptostring()` function to see the contents of the `logs` flex table:

```
=> select maptostring(__raw__) from logs;
                                  maptostring

-------------------------------------------------------------------------------------
  {
   "_cefver" : "0.1",
   "agentzoneuri" : "/All Zones/ArcSight System/Private Address
       Space Zones/RFC1918: 888.99.0.0-888.200.255.255",
   "agt" : "888.99.100.1",
   "ahost" : "SKEELES10",
   "aid" : "3DxKlG0UBABCAA0cXXAZIwA==",
   "art" : "1396328241038",
   "at" : "testalertng",
   "atz" : "Australia/Sydney",
```

```
   "av" : "6.0.3.6664.0",
   "c6a4" : "fe80:0:0:0:495d:cc3c:db1a:de71",
   "c6a4label" : "Agent IPv6 Address",
   "cat" : "/Agent/Started",
   "catdt" : "Security Mangement",
   "categorybehavior" : "/Execute/Start",
   "categorydevicegroup" : "/Application",
   "categoryobject" : "/Host/Application/Service",
   "categoryoutcome" : "/Success",
   "categorysignificance" : "/Normal",
   "cs2" : "<Resource ID=\"3DxKlG0UBABCAA0cXXAZIwA==\"/>",
   "cs2label" : "Configuration Resource",
   "deviceproduct" : "ArcSight",
   "deviceseverity" : "Warning",
   "devicevendor" : "ArcSight",
   "deviceversion" : "6.0.3.6664.0",
   "devicezoneuri" : "/All Zones/ArcSight System/Private Address Space
       Zones/RFC1918: 888.99.0.0-888.200.255.255",
   "dtz" : "Australia/Sydney",
   "dvc" : "888.99.100.1",
   "dvchost" : "SKEELES10",
   "eventid" : "1",
   "filetype" : "Agent",
   "mrt" : "1396328238973",
   "name" : "Agent [test] type [testalertng] started",
   "rt" : "1396328238937",
   "severity" : "Low",
   "signatureid" : "agent:030",
   "version" : "0"
}

(1 row)
```

Create a Columnar Table and Load CEF Data

This example lets you compare the flex table for CEF data with a columnar table. You do so by creating a new table and load the same `CEF_sample.cef` file used in the preceding flex table example.

1. Create a columnar table, `col_logs`, defining the prefix names that are hard coded in `fcefparser`:

```
=> create table col_logs(version int,
  devicevendor varchar,
  deviceproduct varchar,
  deviceversion varchar,
  signatureid varchar,
  name varchar,
  severity varchar);
CREATE TABLE
```

2. Load the sample file into `col_logs`, as you did for the flex table:

```
=> copy col_logs from '/home/dbadmin/data/CEF_sample.cef' parser fcefparser();
 Rows Loaded
-------------
           1
(1 row)
```

3. Query the table. You can find the identical information in the flex table output.

```
=> \x
Expanded display is on.
VMart=> select * from col_logs;
-[ RECORD 1 ]-+----------------------------------------
version       | 0
devicevendor  | ArcSight
deviceproduct | ArcSight
deviceversion | 6.0.3.6664.0
signatureid   | agent:030
name          | Agent [test] type [testalertng] started
severity      | Low
```

Compute Keys and Build a Flex Table View

In this example, you use a flex helper function to compute keys and build a view for the `logs` flex table.

1. Use the `compute_flextable_keys_and_build_view` function to compute keys and populate a view generated from the `logs` flex table:

```
=> select compute_flextable_keys_and_build_view('logs');
                            compute_flextable_keys_and_build_view
-------------------------------------------------------------------------------------
 Please see public.logs_keys for updated keys
The view public.logs_view is ready for querying
(1 row)
```

2. Query the `logs_keys` table to see what the function computed from the sample CEF data:

```
=> select * from logs_keys;
       key_name       | frequency | data_type_guess
----------------------+-----------+-----------------
 c6a4                 |         1 | varchar(60)
 c6a4label            |         1 | varchar(36)
 categoryobject       |         1 | varchar(50)
 categoryoutcome      |         1 | varchar(20)
 categorysignificance |         1 | varchar(20)
 cs2                  |         1 | varchar(84)
 cs2label             |         1 | varchar(44)
```

```
 deviceproduct         |         1 | varchar(20)
 deviceversion         |         1 | varchar(24)
 devicezoneuri         |         1 | varchar(180)
 dvchost               |         1 | varchar(20)
 version               |         1 | varchar(20)
 ahost                 |         1 | varchar(20)
 art                   |         1 | varchar(26)
 at                    |         1 | varchar(22)
 cat                   |         1 | varchar(28)
 catdt                 |         1 | varchar(36)
 devicevendor          |         1 | varchar(20)
 dtz                   |         1 | varchar(32)
 dvc                   |         1 | varchar(24)
 filetype              |         1 | varchar(20)
 mrt                   |         1 | varchar(26)
 _cefver               |         1 | varchar(20)
 agentzoneuri          |         1 | varchar(180)
 agt                   |         1 | varchar(24)
 aid                   |         1 | varchar(50)
 atz                   |         1 | varchar(32)
 av                    |         1 | varchar(24)
 categorybehavior      |         1 | varchar(28)
 categorydevicegroup   |         1 | varchar(24)
 deviceseverity        |         1 | varchar(20)
 eventid               |         1 | varchar(20)
 name                  |         1 | varchar(78)
 rt                    |         1 | varchar(26)
 severity              |         1 | varchar(20)
 signatureid           |         1 | varchar(20)
(36 rows)
```

3. Query several columns from the `logs_view`:

```
=> \x
Expanded display is on.
VMart=> select version, devicevendor, deviceversion, name, severity, signatureid
  from logs_view;
-[ RECORD 1 ]-+----------------------------------------
version       | 0
devicevendor  | ArcSight
deviceversion | 6.0.3.6664.0
name          | Agent [test] type [testalertng] started
severity      | Low
signatureid   | agent:030
```

Use the fcefparser Delimiter Parameter

In this example, you use the `fcefparser delimiter` parameter to query events located in California, New Mexico, and Arizona.

1. Create a new columnar table, `CEFData3`:

```
=> create table CEFData3(eventId int, location varchar(20));
CREATE TABLE
```

2. Using the `delimiter=','` parameter, load some CEF data into the table:

```
=> copy CEFData3 from stdin parser fcefparser(delimiter=',');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> eventId=1,location=California
>> eventId=2,location=New Mexico
>> eventId=3,location=Arizona
>> \.
```

3. Query the table:

```
=> select eventId, location from CEFData3;
 eventId |  location
---------+------------
       1 | California
       2 | New Mexico
       3 | Arizona
(3 rows)
```

Loading CSV Data

Use the `fcsvparser` to load data in CSV format (comma-separated values). Since no formal CSV standard exists, Vertica supports the RFC 4180 standard as the default behavior for `fcsvparser`. Other parser parameters simplify various combinations of CSV options into columnar or flex tables. `fcsvparser` parses the following CSV data formats:

- **RFC 4180:** The RFC4180 csv format parser for Vertica flex tables. The parameters for this format are fixed and cannot be changed.

- **Traditional:** The traditional csv parser, which allows the user to specify the parameter values such as delimiter or record terminator. For a detailed list of parameters, please refer the FCSVPARSER

Using Default Parser Settings

These fixed parameter settings apply to the RCF4180 format. You can change the default values for Traditional format type, if necessary.

Parameter	Data	Fixed	Default

	Type	Value (RCF4180)	Value (Traditional)
`delimiter`	`CHAR`	`,`	`,`
`enclosed_by`	`CHAR`	`"`	`\`
`escape`	`CHAR`	`"`	`"`
`record_terminator`	`CHAR`	`\n; \r\n`	`\n; \r\n`

Use the `type` parameter to indicate either an RFC 4180-compliant file or a traditional-compliant file. You can specify `type` as `RCF4180`. However, you must first verify that the data is compatible with the preceding fixed values for parameters of the RFC4180 format. The default value of the `type` parameter is `RFC4180.`

Loading CSV Data (RFC4180)

Follow these steps to use `fcsvparser` to load data in the RFC4180 CSV data format.

To perform this task, assume that the CSV file to be loaded has the following sample contents:

```
$ more /home/dbadmin/flex/flexData1.csv
sno,name,age,gender
1,John,14,male
2,Mary,23,female
3,Mark,35,male
```

1. Create a flex table:

   ```
   => CREATE FLEX TABLE csv_basic();
   CREATE TABLE
   ```

2. Load the data from csv file using `fcsvparser`:

   ```
   => COPY csv_basic FROM '/home/dbadmin/flex/flexData1.csv' PARSER fcsvparser();
   Rows Loaded
   -------------
   3(1 row)
   ```

3. View the data loaded in the flex table:

   ```
   => SELECT maptostring(__raw__) FROM csv_basic;
   maptostring
   ----------------------------------------------------------------------------------
   {
   "age" : "14",
   "gender" : "male",
   ```

```
"name" : "John",
"sno" : "1"
}
{
"age" : "23",
"gender" : "female",
"name" : "Mary",
"sno" : "2"
}
{
"age" : "35",
"gender" : "male",
"name" : "Mark",
"sno" : "3"
}
(3 rows)
```

Loading CSV Data (Traditional)

Follow these steps to use `fcsvparser` to load data in traditional CSV data format using `fcsvparser`.

In this example, the CSV file loaded has `delimiter` as `$` and `record_terminator` as `#`. Assume that the CSV file to be loaded has the following sample contents:

```
$ more /home/dbadmin/flex/flexData1.csv
sno$name$age$gender#
1$John$14$male#
2$Mary$23$female#
3$Mark$35$male#
```

1. Create a flex table:

```
=> CREATE FLEX TABLE csv_basic();
CREATE TABLE
```

2. Load the data in flex table using `fscvparser` with parameters `type='traditional'`, `delimiter='$'` and `record_terminator='#'` :

```
=> COPY csv_basic FROM '/home/dbadmin/flex/flexData2.csv' PARSER fcsvparser
(type='traditional',
delimiter='$', record_terminator='#');
Rows Loaded
-------------
3
(1 row)
```

3. View the data loaded in the flex table:

```
=> SELECT maptostring(__raw__) FROM csv_basic;
maptostring
----------------------------------------------------------------------------------
```

```
{
"age" : "14",
"gender" : "male",
"name" : "John",
"sno" : "1"
}
{
"age" : "23",
"gender" : "female",
"name" : "Mary",
"sno" : "2"
}
{
"age" : "35",
"gender" : "male",
"name" : "Mark",
"sno" : "3"
}
(3 rows)
```

Note: Refer to the following example to load new line characters using `fcsvparser`.

```
=> COPY foo_1 FROM STDIN PARSER fcsvparser(reject_on_materialized_type_error=true)
rejected data as table "m";
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> a,b
>> "10 <- we have quotes to escape the new line here
>> hi",20
>> 1,2
>> \.
```

Rejecting on Duplicate Values

You can prevent loading duplicate data by using the `reject_on_duplicate=true` option with the `fcsvparser`. The next example shows how to use this parameter and then displays the specified exception and rejected data files.

Note: The `fcsvparser` rejects the entire load if any duplicate exists. The rejected data file includes the duplicate record that caused the load to fail. The exceptions file contains the reason for the rejection.

```
=> CREATE FLEX TABLE csv_basic();
CREATE TABLE

=> COPY csv_basic FROM stdin PARSER fdelimitedparser(reject_on_duplicate=true)
exceptions '/home/dbadmin/load_errors/except.out' rejected data '/home/dbadmin/load_
errors/reject.out';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
```

```
>> A|A
>> 1|2
>> \.
=> \! cat /home/dbadmin/load_errors/reject.out
A|A
=> \! cat /home/dbadmin/load_errors/except.out
COPY: Input record 1 has been rejected (Processed a header row with duplicate keys with
reject_on_duplicate specified; rejecting.).  Please see /home/dbadmin/load_errors/reject.out,
record 1 for the rejected record.
COPY: Loaded 0 rows, rejected 1 rows.
```

Rejecting Data on Materialized Column Type Errors

The `fcsvparser` parser has a Boolean parameter, `reject_on_materialized_type_error`. Setting this parameter to `true` causes rows to be rejected if *both* the following conditions exist in the input data:

- Includes keys matching an existing materialized column
- Has a key value that cannot be coerced into the materialized column's data type

The following examples illustrate setting this parameter.

1. Create a table, `reject_true_false`, with two real columns:

    ```
    => CREATE FLEX TABLE reject_true_false(one int, two int);
    CREATE TABLE
    ```

2. Load CSV data into the table (from STDIN), using the `fcsvparser` with `reject_on_materialized_type_error=false`. While `false` is the default value,you can specify it explicitly, as shown. Additionally, set the parameter `header=true` to specify the columns for input values:

    ```
    => COPY reject_true_false FROM stdin PARSER fcsvparser(reject_on_materialized_type_
    error=false,header=true);
    Enter data to be copied followed by a newline.
    End with a backslash and a period on a line by itself.
    >> one,two
    >> 1,2
    >> "3","four"
    >> "five",6
    >> 7,8
    >> \.
    ```

3. Invoke `maptostring` to display the table values after loading data:

```
=> SELECT maptostring(__raw__), one, two FROM reject_true_false;
maptostring               | one | two
----------------------------------------+-----+-----
{
"one" : "1",
"two" : "2"
}
|   1 |   2
{
"one" : "3",
"two" : "four"
}
|   3 |
{
"one" : "five",
"two" : "6"
}
|     |   6
{
"one" : "7",
"two" : "8"
}
|   7 |   8
(4 rows)
```

4. Truncate the table to empty the data stored in the table:

```
=> TRUNCATE TABLE reject_true_false;
TRUNCATE TABLE
```

5. Reload the same data again, but this time, set `reject_on_materialized_type_error=true`:

```
=> COPY reject_true_false FROM stdin PARSER fcsvparser(reject_on_materialized_type_
error=true,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> one,two
>> 1,2
>> "3","four"
>> "five",6
>> 7,8
>> \.
```

6. Call `maptostring` to display the table contents. Only two rows are currently loaded, whereas the previous results had four rows. The rows having input values with incorrect data type have been rejected:

```
=> SELECT maptostring(__raw__), one, two FROM reject_true_false;
maptostring             | one | two
-------------------------------------+-----+-----
{
"one" : "1",
```

```
"two" : "2"
}
|   1 |   2
{
"one" : "7",
"two" : "8"
}
|   7 |   8
(2 rows)
```

Note: The parser `fcsvparser` uses `null` values if there is a type mismatch and you set the `reject_on_materialized_type_error` parameter to `false`.

Rejecting or Omitting Empty Rows

Valid CSV files can include empty key and value pairs. Such rows are invalid for SQL. You can control the behavior for empty rows by either rejecting or omitting them, using two boolean `FCSVPARSER` parameters:

- `reject_on_empty_key`
- `omit_empty_keys`

The following example illustrates how to set these parameters:

1. Create a flex table:

```
 => CREATE FLEX TABLE csv_basic();
CREATE TABLE
```

2. Load CSV data into the table (from STDIN), using the `fcsvparser` with `reject_on_empty_key=false`. While `false` is the default value, you can specify it explicitly, as shown. Additionally, set the parameter `header=true` to specify the columns for input values:

```
=> COPY csv_basic FROM stdin PARSER fcsvparser(reject_on_empty_key=false,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> ,num
>> 1,2
>> \.
```

3. Invoke `maptostring` to display the table values after loading data:

```
=>SELECT maptostring(__raw__) FROM csv_basic;
```

```
maptostring
----------------------------------
{
"" : "1",
"num" : "2"
}

(1 row)
```

4. Truncate the table to empty the data stored in the table:

```
=> TRUNCATE TABLE csv_basic;
TRUNCATE TABLE
```

5. Reload the same data again, but this time, set `reject_on_empty_key=true`:

```
=> COPY csv_basic FROM stdin PARSER fcsvparser(reject_on_empty_key=true,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> ,num
>> 1,2
>> \.
```

6. Call `maptostring` to display the table contents. No rows are loaded because one of the keys is empty:

```
=>SELECT maptostring(__raw__) FROM csv_basic;
maptostring
-------------
(0 rows)
```

7. Truncate the table to empty the data stored in the table:

```
=> TRUNCATE TABLE csv_basic;
TRUNCATE TABLE
```

8. Reload the same data again, but this time, set `omit_empty_keys=true`:

```
=> COPY csv_basic FROM stdin PARSER fcsvparser(omit_empty_keys=true,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> ,num
>> 1,2
>> \.
```

9. Call `maptostring` to display the table contents. One row is now loaded, and the rows with empty keys are omitted:

```
=> SELECT maptostring(__raw__) FROM csv_basic;
maptostring
```

```
--------------------
{
"num" : "2"
}
(1 row)
```

Note: The `fcsvparser` uses 'col###' as a column name if no header row is indicated. If a table column name and key name matches, the parser loads the table column with values associated with the matching key.

Using the NULL Parameter

Use the `NULL` parameter with `fcsvparser` to load `NULL` values into a flex table.

The next example uses this parameter:

1. Create a flex table.

```
=> CREATE FLEX TABLE fcsv(c1 int);
CREATE TABLE
```

2. Load CSV data in flex table using STDIN and NULL parameter.

```
=> COPY fcsv FROM STDIN PARSER fcsvparser() delimiter '|' NULL 'NULL' ;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> a,b,c1
>> 10,20,NULL
>> 20,30,50
>> 20,30,40
>> \.
```

3. Use `compute_flextable_keys_and_build_view` function to compute keys and build flex view.

```
=> SELECT compute_flextable_keys_and_build_view('fcsv');
compute_flextable_keys_and_build_view
--------------------------------------------------------------------------------

--------------
Please see public.fcsv_keys for updated keys
The view public.fcsv_view is ready for querying
(1 row)
```

4. View the flex view and replace the `NULL` values.

```
=> SELECT * FROM public.fcsv_view;
a  | b  | c1
----+----+----
20 | 30 | 50
```

```
10 | 20 |
20 | 30 | 40
(3 rows)

=> SELECT a,b, ISNULL(c1,-1) from public.fcsv_view;
a  | b  | ISNULL
----+----+--------
20 | 30 |     50
10 | 20 |     -1
20 | 30 |     40
(3 rows)
```

Loading Delimited Data

You can load flex tables with one of two delimited parsers, `fdelimitedparser` or `fdelimitedpairparser`.

- Use `fdelimitedpairparser` when the data specifies column names with the data in each row.
- Use `fdelimitedparser` when the data does not specify column names or has a header row for column names.

This section describes using some options that `fdelimitedpairparser` and fdelimitedparser support.

Rejecting Duplicate Values

You can prevent loading duplicate data by using the `reject_on_duplicate=true` option with the `fdelimitedparser`. The next example shows how to use this parameter and then displays the specified exception and rejected data files.

Note: For the `fdelimitedparser`, the entire load is rejected if any duplicate is found. The exceptions file contains the reason for the rejection. The rejected data file includes the duplicate record that caused the load to fail.

```
=> create flex table delim_dupes();
CREATE TABLE

=> copy delim_dupes from stdin parser fdelimitedparser(reject_on_duplicate=true)
exceptions '/home/dbadmin/load_errors/except.out' rejected data '/home/dbadmin/load_
errors/reject.out';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.

>> A|A
>> 1|2
>> \.
```

```
=> \! cat /home/dbadmin/load_errors/reject.out
A|A
=> \! cat /home/dbadmin/load_errors/except.out
COPY: Input record 1 has been rejected (Processed a header row with duplicate keys with
reject_on_duplicate specified; rejecting.).  Please see /home/dbadmin/load_errors/reject.out,
record 1 for the rejected record.
COPY: Loaded 0 rows, rejected 1 rows.
```

Rejecting Materialized Column Type Errors

Both the `fjsonparser` and `fdelimitedparser` parsers have a boolean parameter, `reject_on_materialized_type_error`. Setting this parameter to `true` causes rows to be rejected if *both* the following conditions exist in the input data:

- Includes keys matching an existing materialized column
- Has a value that cannot be coerced into the materialized column's data type

Suppose the flex table has a materialized column, `OwnerPercent`, declared as a `FLOAT`. Trying to load a row with an `OwnerPercent` key that has a `VARCHAR` value causes `fdelimitedparser` to reject the data row.

The following examples illustrate setting this parameter.

1. Create a table, `reject_true_false`, with two real columns:

```
=> create flex table reject_true_false(one varchar, two int);
CREATE TABLE
```

2. Load JSON data into the table (from `STDIN`), using the `fjsonparser` with `reject_on_materialized_type_error=false`. While `false` is the default value, the following example specifies it explicitly for illustration:

```
=> copy reject_true_false from stdin parser fjsonparser(reject_on_materialized_type_
error=false);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one": 1, "two": 2}
>> {"one": "one", "two": "two"}
>> {"one": "one", "two": 2}
>> \.
```

3. Invoke `maptostring` to display the table values after loading data:

```
=>  select maptostring(__raw__), one, two from reject_true_false;
               maptostring        | one | two
----------------------------------+-----+-----
 {
```

```
   "one" : "one",
   "two" : "2"
}
   | one |   2
{
   "one" : "1",
   "two" : "2"
}
   | 1   |   2
 {
   "one" : "one",
   "two" : "two"
}
   | one |
(3 rows)
```

4. Truncate the table:

```
=> truncate table reject_true_false;
```

5. Reload the same data again, but this time, set `reject_on_materialized_type_error=true`:

```
=> copy reject_true_false from stdin parser fjsonparser(reject_on_materialized_type_
error=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one": 1, "two": 2}
>> {"one": "one", "two": "two"}
>> {"one": "one", "two": 2}
>> \.
```

6. Call `maptostring` to display the table contents. Only two rows were loaded, whereas the previous results had three rows:

```
=> select maptostring(__raw__), one, two from reject_true_false;
              maptostring              | one | two
---------------------------------------+-----+-----
 {
   "one" : "1",
   "two" : "2"
}
   | 1   |   2
 {
   "one" : "one",
   "two" : "2"
}
 | one |   2
(2 rows)
```

Loading JSON Data

You can load JSON data into flex or columnar tables. This section describes some examples of using the `fjsonparser` with several of the parser options.

Checking JSON Integrity

Before loading any JSON data, be sure that the data is valid. You can verifyJSON data integrity using a web tool such as JSONLint. Copy your JSON data into the tool. If any data is invalid, the tool returns a message similar to the one in this example:

```
Parse error on line 170:...257914002502451200}{    "id_str": "257
----------------------^
Expecting 'EOF', '}', ',', ']'
```

Using flatten_maps and flatten_arrays Parameters

When loading JSON data, the `fjsonparser` uses the parameters `flatten_maps` and `flatten_arrays` to control how the parser handles the data it is loading. Here are the default settings for these two parameters:

Parameter	Default	Change Default
`flatten_maps`	`TRUE`: Flatten all maps.	`flatten_maps=FALSE`
`flatten_arrays`	`FALSE`: Do not flatten arrays.	`flatten_arrays=TRUE`

You control the default the behavior by using one or both `flatten_` parameters.

For JSON maps, the parser flattens all submaps, separating the levels with a period (`.`). Consider the following input data with a submap:

```
 { grade: { level: 4 } }
```

The default parser behavior results in the following map:

```
{ "grade.level" -> "4" }
```

For JSON arrays, the parser maintains the array. Consider the following input data containing a 2-element array, with values 1 and 2:

```
{ grade: [ 1 2 ] }
```

The default parser behavior results in the following array:

```
{ "grade": { "0" -> "1", "1" -> "2" } }
```

Note: Using the parameters `flatten_maps` and `flatten_arrays` is recursive, and affects all data.

Loading from a Specific Start Point

You can use the `fjsonparser start_point` parameter to load JSON data beginning at a specific key, rather than at the beginning of a file. Data is parsed from after the `start_point` key until the end of the file, or to the end of the first `start_point`'s value. The `fjsonparser` ignores any subsequent instance of the `start_point`, even if that key appears multiple times in the input file. If the input data contains only one copy of the `start_point` key, and that value is a list of JSON elements, the parser loads each element in the list as a row.

This section uses the following sample JSON data, saved to a file (`alphanums.json`):

```
 { "A": { "B": { "C": [ { "d": 1, "e": 2, "f": 3 }, { "g": 4, "h": 5, "i": 6 },
{ "j": 7, "k": 8, "l": 9 } ] } } }
```

1. Create a flex table, `start_json`:

   ```
   => create flex table start_json();
   CREATE TABLE
   ```

2. Load `alphanums.json` into `start_json` using the `fjsonparser` without any parameters:

   ```
   =>  copy start_json from '/home/dbadmin/data/flex/alphanums.json' parser fjsonparser();
    Rows Loaded
   -------------
              1
   (1 row)
   ```

3. Use `maptostring` to see the results of loading all of `alphanums.json`:

   ```
   => select maptostring(__raw__) from start_json;
                           maptostring
   -------------------------------------------------------------------------------
    {
      "A.B.C" : {
         "0.d" : "1",
         "0.e" : "2",
         "0.f" : "3",
         "1.g" : "4",
   ```

```
      "1.h" : "5",
      "1.i" : "6",
      "2.j" : "7",
      "2.k" : "8",
      "2.l" : "9"
   }
}

(1 row)
```

4. Truncate `start_json` and load `alphanums.json` with the `start_point` parameter:

```
=> truncate table start_json;
TRUNCATE TABLE
=>  copy start_json from '/home/dbadmin/data/flex/alphanums.json' parser
-> fjsonparser(start_point='B');
 Rows Loaded
-------------
           1
(1 row)
```

5. Next, call `maptostring` again to compare the results of loading `alphanums.json`from `start_point='B'`:

```
=> select maptostring(__raw__) from start_json;
                         maptostring
--------------------------------------------------------------------------------
 {
   "C" : {
      "0.d" : "1",
      "0.e" : "2",
      "0.f" : "3",
      "1.g" : "4",
      "1.h" : "5",
      "1.i" : "6",
      "2.j" : "7",
      "2.k" : "8",
      "2.l" : "9"
   }
}

(1 row)
```

Parsing From a Start Point Occurrence

If a `start_point` value occurs in multiple locations in your JSON data, you can use the `start_point_occurrence` integer parameter to specify the occurrence at which to start parsing. By defining `start_point_occurrence`, `fjsonparser` begins at the nth occurrence of `start_point`.

Controlling Column Name Separators

By default, `fjsonparser` produces column names by concatenating JSON field names with a period (`.`). You can change the default separator by specifying a different character with the `key_separator` parameter.

Handling Special Characters

Some input JSON data can have special characters in field names. You can replace these characters by setting the `suppress_nonalphanumeric_key_chars` to `TRUE`. With this parameter setting, all special characters are converted to an underscore (_) character.

Rejecting on Duplicate Values

You can avoid loading duplicate data by using the `reject_on_duplicate=true` option with the `fjsonparser`. The next example uses this option while loading data and then displays the specified exception and rejected data files.

Note: For the `fjsonparser`, the entire load is rejected if any duplicate is found. To save disk space, rejected data is not saved. The rejected data file includes one newline per rejected record, and the exceptions file includes the reason for the rejection.

```
=> create flex table json_dupes();
CREATE TABLE

=> copy json_dupes from stdin parser fjsonparser(reject_on_duplicate=true)
exceptions '/home/dbadmin/load_errors/json_e.out'
rejected data '/home/dbadmin/load_errors/json_r.out';

Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.

>> {"a":"1","a":"2","b":"3"}
>> \.

=>  \!cat /home/dbadmin/load_errors/json_e.out
COPY: Input record 1 has been rejected (Rejected by user-defined parser).
Please see /home/dbadmin/load_errors/json_r.out, record 1 for the rejected record.
COPY: Loaded 0 rows, rejected 1 rows.
```

Rejecting Data on Materialized Column Type Errors

Both the `fjsonparser` and `fdelimitedparser` parsers have a Boolean parameter, `reject_on_materialized_type_error`. Setting this parameter to `true` causes rows to be rejected if the input data:

- Includes keys matching an existing materialized column
- Has a key value that cannot be coerced into the materialized column's data type.

The following examples illustrate setting this parameter.

1. Create a table, `reject_true_false`, with two real columns:

```
=> create flex table reject_true_false(one varchar, two int);
CREATE TABLE
```

2. Load JSON data into the table (from STDIN), using the `fjsonparser` with `reject_on_materialized_type_error=false`. While `false` is the default value, the following example specifies it explicitly for illustration:

```
=> copy reject_true_false from stdin parser
-> fjsonparser(reject_on_materialized_type_error=false);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one": 1, "two": 2}
>> {"one": "one", "two": "two"}
>> {"one": "one", "two": 2}
>> \.
```

3. Invoke `maptostring` to display the table values after loading data:

```
=>  select maptostring(__raw__), one, two from reject_true_false;
               maptostring        | one | two
----------------------------------+-----+-----
 {
   "one" : "one",
   "two" : "2"
}
   | one |   2
{
   "one" : "1",
   "two" : "2"
}
   | 1   |   2
 {
   "one" : "one",
   "two" : "two"
}
   | one |
(3 rows)
```

4. Truncate the table:

```
=> truncate table reject_true_false;
```

5. Reload the same data again, but this time, set `reject_on_materialized_type_error=true`:

```
=> copy reject_true_false from stdin parser fjsonparser(reject_on_materialized_type_
error=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one": 1, "two": 2}
>> {"one": "one", "two": "two"}
>> {"one": "one", "two": 2}
>> \.
```

6. Call `maptostring` to display the table contents. Only two rows were loaded, whereas the previous results had three rows:

```
=> select maptostring(__raw__), one, two from reject_true_false;
              maptostring              | one | two
---------------------------------------+-----+-----
 {
   "one" : "1",
   "two" : "2"
}
   | 1   |   2
 {
   "one" : "one",
   "two" : "2"
}
 | one |   2
(2 rows)
```

Rejecting or Omitting Empty Rows

Valid JSON files HPE can include empty key and value pairs, such as this one:

```
{"": 1 "}
```

Such rows are invalid for SQL. To prevent this situation, you can control the behavior for empty rows, either rejecting or omitting them. You do so using two boolean parameters for the parsers `FDELIMITEDPARSER` or `FJSONPARSER`:

- `reject_on_empty_key`
- `omit_empty_keys`

See Also

- Using COPY with Kafka

Loading Matches from Regular Expressions

You can load flex or columnar tables with the matched results of a regular expression, using the `fregexparser`. This section describes some examples of using the options that the flex parsers support.

Sample Regular Expression

These examples use the following regular expression, which searches information that includes the `timestamp`, `date`, `thread_name`, and `thread_id`. For illustrative purposes, this regular expression uses new lines to break up long text. Remove any new line characters before using this example in your own tests.

```
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?
(?:\[(?<component>\w+)\] \<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @\[?(?<enode>\w+)\]?: )?
(?<text>.*)'
```

Using Regular Expression Matches for a Flex Table

You can load the results from a regular expression into a flex table, using the `fregexparser`. For a complete example of doing so, see FREGEXPARSER.

Using fregexparser for Columnar Tables

This section illustrates how to load the results of a regular expression used against a sample log file for a Vertica database. By using an external table definition, the This section presents an example of using the fregexparser to load data into a columnar table. Using a flex table parser for a columnar tables gives you the capability to mix data loads in one table, such as loading the results of a regular expression in one session, and JSON data in another.

The following basic examples illustrate this usage.

1. Create a columnar table, `vlog`, with the following columns:

```
=> CREATE TABLE vlog (
       "text"                  varchar(2322),
       thread_id               varchar(28),
       thread_name     varchar(44),
       "time"                  varchar(46),
       component               varchar(30),
```

```
       level                   varchar(20),
       transaction_id  varchar(32),
       elevel                  varchar(20),
       enode                   varchar(34)
);
```

2. Use COPY to load parts of a log file using the sample regular expression, with the `fregexparser`:

```
=> COPY v_log FROM '/home/dbadmin/data/flex/vertica.log' PARSER
FRegexParser(pattern='^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?
(?:\[(?<component>\w+)\] <(?<level>\w+)> )?(?:<(?<elevel>\w+)> @\[?(?<enode>\w+)\]?: )
?(?<text>.*)') rejected data as table fregex_reject;
```

3. Query the `time` column:

```
=> select time from flogs limit 10;
          time
-------------------------
 2014-04-02 04:02:02.613
 2014-04-02 04:02:02.613
 2014-04-02 04:02:02.614
 2014-04-02 04:02:51.008
 2014-04-02 04:02:51.010
 2014-04-02 04:02:51.012
 2014-04-02 04:02:51.012
 2014-04-02 04:02:51.013
 2014-04-02 04:02:51.014
 2014-04-02 04:02:51.017
(10 rows)
```

Using External Tables with fregexparser

By creating an external columnar table for your Vertica log file, querying the table will return updated log information. The following basic example illustrate this usage.

1. Create a columnar table, `vertica_log`, using the `AS COPY` clause and `fregexparser` to load matched results from the regular expression. For illustrative purposes, the regular expression has new lines to break up long text:

```
=> CREATE EXTERNAL TABLE public.vertica_log
(
    "text" varchar(2322),
    thread_id varchar(28),
    thread_name varchar(44),
    "time" varchar(46),
    component varchar(30),
    level varchar(20),
    transaction_id varchar(32),
```

```
    elevel varchar(20),
    enode varchar(34)
)
AS COPY
FROM '/home/dbadmin/data/vertica.log'
PARSER FRegexParser(pattern='^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?
(?:\[(?<component>\w+)\] \<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @\[?(?<enode>\w+)\]?: )?
(?<text>.*)');
```

2. Query from the external table to get updated results:

```
=> select component, thread_id, time from vertica_log limit 10;
 component | thread_id  |          time
-----------+------------+------------------------
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
           | 0x16321430 | 2014-04-02 04:02:02.614
           | 0x16321430 | 2014-04-02 04:02:02.614
           | 0x16321430 | 2014-04-02 04:02:02.614
(10 rows)
```

Computing Flex Table Keys

After loading data into a flex table, you must determine what key value pairs exist as populated virtual columns in the data. Two helper functions compute keys from map data:

- COMPUTE_FLEXTABLE_KEYS— See also COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW— Performs the same functionality as COMPUTE_FLEXTABLE_KEYS but also builds a new view. See also Updating Flex Table Views.

Using COMPUTE_FLEXTABLE_KEYS

Call this function with a flex table argument to compute a list of keys from the map data:

Calling the Function	Results
`compute_flextable_keys ('flex_table')`	Computes keys from the *flex_table* map data and populates the associated *flex_table*`_keys` with the virtual columns.

Calculating Key Value Column Widths

During execution, this function determines a data type for each virtual column. It casts the values computed to `VARCHAR`, `LONG VARCHAR`, or `LONG VARBINARY`. Casting choice depends on the length of the key and whether the key includes nested maps.

The following examples illustrate this function. It showsresults of populating the _keys table, after you create a flex table (darkdata1) and load data:

```
=> create flex table darkdata1();
CREATE TABLE
=> copy darkdata1 from '/test/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
=> select compute_flextable_keys('darkdata1');
            compute_flextable_keys
--------------------------------------------------
 Please see public.darkdata1_keys for updated keys
(1 row)
=> select * from darkdata1_keys;
                         key_name                         | frequency |   data_type_guess
----------------------------------------------------------+-----------+---------------------
```

```
 contributors                                            |         8 | varchar(20)
 coordinates                                             |         8 | varchar(20)
 created_at                                              |         8 | varchar(60)
 entities.hashtags                                       |         8 | long varbinary(186)
 entities.urls                                           |         8 | long varbinary(32)
 entities.user_mentions                                  |         8 | long varbinary(674)
.
.
.
 retweeted_status.user.time_zone                         |         1 | varchar(20)
 retweeted_status.user.url                               |         1 | varchar(68)
 retweeted_status.user.utc_offset                        |         1 | varchar(20)
 retweeted_status.user.verified                          |         1 | varchar(20)
(125 rows)
```

The flex keys table has these columns:

Column	Description
`key_name`	The name of the virtual column (key).
`frequency`	The number of times the virtual column occurs in the map.
`data_type_guess`	The data type for each virtual column. This value is cast to `VARCHAR`, `LONG VARCHAR` or `LONG VARBINARY`. Casting depends on the length of the key and whether the key includes one or more nested maps. In the `_keys` table output, the `data_type_guess` column values are also followed by a value in parentheses, such as `varchar(20)`. The value indicates the padded width of the key column. The width is determined by calculating the longest field, multiplied by the `FlexTableDataTypeGuessMultiplier` configuration parameter value. For more information, see Setting Flex Table Parameters.

Materializing Flex Tables

Once flex tables exist, you can change the table structure to promote virtual columns to materialized (real) columns. If your table is already a hybrid table, you can change existing real columns and promote other important virtual columns. This section describes some key aspects of promoting columns, adding columns, specifying constraints, and declaring default values. It also presents some differences when loading flex or hybrid tables, compared with columnar tables.

Note: Materializing virtual columns by promoting them to real columns can significantly improve query performance. Vertica recommends that you materialize important virtual columns before running large and complex queries. Promoted columns cause a small increase in load performance.

Adding Columns to Flex Tables

Add columns to your flex tables to promote virtual columns:

1. Add a column with the same name as a map key:

```
=> alter table darkdata1 add column "user.name" varchar;
ALTER TABLE
```

2. Loading data into a materialized column populates the new column automatically:

```
 => copy darkdata1 from '/vertica/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

3. Query the materialized column from the flex table:

```
=> select "user.name" from darkdata1;
      user.name
---------------------
 I'm Toasterâ¥
 Flu Beach
 seydo shi
 The End
 Uptown gentleman.
 ~G A B R I E L A â¿
 Avita Desai
 laughing at clouds.
(12 rows)
```

Adding Columns with Default Values

The section Using COPY with Flex Tables describes the use of default values, and how they are evaluated during loading. As with all tables, using COPY to load data ignores any column default values.

Note: Adding a table column default expression to a flex table requires casting the column to an explicit data type.

1. Create a darkdata1 table with some column definition, but a name that does not correspond to any key names in the JSON data you'll load. Assign a default value for a column you know exists in your data ("`user.lang`"):

```
=> create flex table darkdata1(talker long varchar default "user.lang");
CREATE TABLE
```

2. Load some JSON data:

```
=> copy darkdata1 from '/test/vertica/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

3. Query the `talker` column to see that the default value was not used. The column contains `NULL` values.

4. Load data again, specifying just the `__raw__` column to use the column's default value:

```
=> copy darkdata1 (__raw__)from '/test/vertica/flextable/DATA/tweets_12.json'
    parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

5. Query to see that the column's default expression was used ("`user.lang`"), because you specified `__raw__`:

```
=> select "talker" from darkdata1;
 talker
--------
 it
```

```
 en
 es
 en
 en
 es
 tr
 en
(12 rows)
```

6. Alter the table to add a row with a key value name, assigning the key name as the default value (recommended):

```
=> alter table darkdata1 add column "user.name" varchar default "user.name";
ALTER TABLE
```

7. Load data again, this time without `__raw__`:

```
=> copy darkdata1 from '/test/vertica/flextable/DATA/tweets_12.json' parser fjsonparser();
```

8. Query the two real columns and see that `talker` is `NULL`, since you did not specify the `__raw__` column. The `user.lang` column contains values from the data you loaded:

```
=> select "talker", "user.name" from darkdata1;
 talker |      user.name
--------+--------------------
        | laughing at clouds.
        | Avita Desai
        | I'm Toasterâ¥
        |
        |
        |
        | Uptown gentleman.
        | ~G A B R I E L A â¿
        | Flu Beach
        |
        | seydo shi
        | The End
(12 rows)
```

9. Load data once more, this time specifying a COPY statement default value expression for `user.name`:

```
=> copy darkdata1 (__raw__, "user.name" as 'QueenElizabeth'::varchar) from
'/test/vertica/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
```

```
(1 row)
```

10. Query once more. `talker` has its default values (you used `__raw__`), and the COPY value expression (`QueenElizabeth`) overrode the `user.name` default column value:

```
=> select "talker", "user.name" from darkdata1;
 talker |   user.name
--------+----------------
 it     | QueenElizabeth
 en     | QueenElizabeth
 es     | QueenElizabeth
        | QueenElizabeth
        | QueenElizabeth
        | QueenElizabeth
 en     | QueenElizabeth
 en     | QueenElizabeth
 es     | QueenElizabeth
        | QueenElizabeth
 tr     | QueenElizabeth
 en     | QueenElizabeth
(12 rows)
```

To summarize, you can set a default column value as part of the `ALTER TABLE...ADD COLUMN...` operation. For materializing columns, the default should reference the key name of the virtual column (as in `"user.lang"`). Subsequently loading data with a COPY value expression overrides the default value of the column definition.

Changing the __raw__ Column Size

You can change the default size of the `__raw__` column for flex tables you plan to create, the current size of an existing flex table, or both.

To change the default size for the flex table `__raw__` column, use the following configuration parameter (described in Setting Flex Table Parameters):

```
=> ALTER DATABASE mydb SET FlexTableRawSize = 120000;
```

Changing the configuration parameter affects all flex tables you create after making this change.

To change the size of the `_raw_` column in an existing flex table, use the ALTER TABLE statement as follows:

```
=> alter table tester alter column __raw__ set data type long varbinary(120000);
ALTER TABLE
```

Note: An error will occur if you try reducing the `__raw__` column size to a value smaller than the data that the column already contains.

Changing Flex Table Real Columns

You can make the following changes to the flex table real columns (`__raw__` and `__identity__`), but not to any virtual columns:

Actions	__raw__	__identity__
Change NOT NULL constraints (default)	Yes	Yes
Add primary key and foreign key (PK/FK) constraints	No	Yes
Create projections	No	Yes
Segment	No	Yes
Partition	No	Yes
Specify a user-defined scalar function (UDSF) as a default column expression in `ALTER TABLE x ADD COLUMN y` statement	No	No

Note: While segmenting and partitioning the `__raw__` column is permitted, it is not recommended due to its long data type. By default, if no real columns exist, flex tables are segmented on the `__identity__` column.

Dropping Flex Table Columns

There are two considerations about dropping columns:

- You cannot drop the last column in your flex table's sort order.
- If you have not created a flex table with any real columns, or materialized any columns, you cannot drop the `__identity__` column.

Updating Flex Table Views

Creating a flex table also creates a default view to accompany the table. The view has the name of the table with an underscore (`_view`) suffix. When you perform a `select` query from the default view, you get a prompt to run the helper function, as shown in this example:

```
=> \dv dark*
                  List of View Fields
 Schema |      View       | Column |     Type     | Size
--------+----------------+--------+-------------+------
 public | darkdata_view  | status | varchar(124) |  124
 public | darkdata1_view | status | varchar(124) |  124
(2 rows)

=> select * from darkdata_view;
                                        status
-----------------------------------------------------------------------------------------------
 Please run compute_flextable_keys_and_build_view() to update this view to reflect real and
virtual columns in the flex table
(1 row)
```

Two helper functions create views:

- COMPUTE_FLEXTABLE_KEYS— See also COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW— Performs the same functionality as BUILD_FLEXTABLE_KEYS but also computes keys. See also Using COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW.

Using BUILD_FLEXTABLE_VIEW

After the function computes keys for the Flex Table (Computing Flex Table Keys), call this function with one or more arguments. The records under the `key_name` column of the *`{flextable}_keys`* table are used as view columns, along with any values for the key. If no values exist, the column value is NULL.

Regardless of the number of arguments, calling this function replaces the contents of the existing view as follows:

Function Invocation	Results
`build_flextable_view ('flexible_table')`	Changes the existing view associated with *flexible_ table* with the current contents of the associated *flexible_table*_`keys` table.

Function Invocation	Results
`build_flextable_view ('flexible_table', 'view_name')`	Changes the view you specify with *view_name* by using the current contents of the *{flextable}_keys* table.
`build_flextable_view ('flexible_table', 'view_name', 'table_keys')`	Changes the view you specify with *view_name* to the current contents of the *flexible_table_keys* table. Use this function to change a view of your choice with the contents of the keys of interest.

If you do not specify a `view_name` argument, the default name is the flex table name with a `_view` suffix. For example, if you specify the table `darkdata` as the sole argument to this function, the default view is called `darkdata_view`.

You cannot specify a custom view name with the same name as the default view `flex_table_view`, unless you first drop the default-named view and then create your own view of the same name.

Creating a view stores a definition of the column structure at the time of creation. Thus, if you create a flex table view and then promote virtual columns to real columns, you must rebuild the view. Querying a rebuilt flex table view that has newly promoted real columns produces two results. These results reflect values from both virtual columns in the map data and real columns.

Handling JSON Duplicate Key Names in Views

SQL is a case-insensitive language, so the names `TEST`, `test`, and `TeSt` are identical. JSON data is case sensitive, so that it can validly contain key names of different cases with separate values.

When you build a flex table view, the function generates a warning if it detects same-name keys with different cases in the *`{flextable}_keys`* table. For example, calling `BUILD_FLEXTABLE_VIEW` or `COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW()` on a flex table with duplicate key names results in these warnings:

```
=> select compute_flextable_keys_and_build_view('dupe');
```

```
WARNING 5821:  Detected keys sharing the same case-insensitive key name
WARNING 5909:  Found and ignored keys with names longer than the maximum column-name length limit


                            compute_flextable_keys_and_build_view
----------------------------------------------------------------------------------------------
 Please see public.dupe_keys for updated keys
The view public.dupe_view is ready for querying
(1 row)
```

While a *{flextable}_keys* table can include duplicate key names with different cases, a view cannot. Creating a flex table view with either of the helper functions consolidates any duplicate key names to one column name, consisting of all lowercase characters. All duplicate key values for that column are saved. For example, if these key names exist in a flex table:

- `test`
- `Test`
- `tESt`

The view will include a virtual column `test` with values from the `test`, `Test`, and `tESt` keys.

> **Note:** The examples in this section include added Return characters to reduce line lengths. The product output may differ.

For example, consider the following query, showing the duplicate `test` key names:

```
=> \x
Expanded display is on.
dbt=> select * from dupe_keys;
-[ RECORD 1 ]---+-----------------------------------------------------------------------------
----------------
key_name        |
TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttest
TesttestTesttestTesttestTesttest
frequency       | 2
data_type_guess | varchar(20)
-[ RECORD 2 ]---+-----------------------------------------------------------------------------
----------------
key_name        |
TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttest
TesttestTesttestTesttestTest12345
frequency       | 2
data_type_guess | varchar(20)
-[ RECORD 3 ]---+-----------------------------------------------------------------------------
----------------
key_name        | test
frequency       | 8
data_type_guess | varchar(20)
-[ RECORD 4 ]---+-----------------------------------------------------------------------------
```

```
----------------
key_name        | TEst
frequency       | 8
data_type_guess | varchar(20)
-[ RECORD 5 ]---+--------------------------------------------------------------------------------
----------------
key_name        | TEST
frequency       | 8
data_type_guess | varchar(20)
```

The following query displays the `dupe` flex table (`dupe_view`). It shows the consolidated `test` and `testtesttest...` virtual columns. All the `test`, `Test`, and `tESt` virtual column values are in the `test` column:

```
=> select * from dupe_view;
  test  |
testtesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttest
testtesttesttesttesttesttesttest
--------+------------------------------------------------------------------------------------------
--------
 upper2 |
 half4  |
 lower1 |
 upper1 |
 half1  |
 half4  |
        |
 lower1 |
 half1  |
 upper2 |
        |
 lower2 |
 lower3 |
 upper1 |
 lower2 |
 lower3 |
(16 rows)
```

Creating a Flex Table View

The following example shows how to create a view, `dd_view`, from the flex table `darkdata`, which contains JSON data.

```
=> create view dd_view as select "user.lang"::varchar, "user.name"::varchar from darkdata;
CREATE VIEW
```

Query the key names you specified, and their values:

```
=> select * from dd_view;
 user.lang |      user.name
-----------+--------------------
 en        | Uptown gentleman.
 en        | The End
```

```
 it        | laughing at clouds.
 es        | I'm Toasterâ¥
           |
 en        | ~G A B R I E L A â¿
           |
 en        | Avita Desai
 tr        | seydo shi
           |
           |
 es        | Flu Beach
(12 rows)
```

This example shows how to call `build_flextable_view` with the original table and the view you previously created, `dd_view`:

```
=> select build_flextable_view ('darkdata', 'dd_view');
         build_flextable_view
-----------------------------------------------
 The view public.dd_view is ready for querying
(1 row)
```

Query the view again. You can see that the function populated the view with the contents of the `darkdata_keys` table. Next, review a snippet from the results, with the `key_name` columns and their values:

```
=> \x
Expanded display is on.
```

```
=> select * from dd_view;
.
.
.
user.following                                          |
user.friends_count                                      | 791
user.geo_enabled                                        | F
user.id                                                 | 164464905
user.id_str                                             | 164464905
user.is_translator                                      | F
user.lang                                               | en
user.listed_count                                       | 4
user.location                                           | Uptown..
user.name                                               | Uptown gentleman.
.
.
.
```

When building views, be aware that creating a view stores a definition of the column structure at the time the view is created. If you promote virtual columns to real columns after building a view, the existing view definition is not changed. Querying this view with a select statement such as the following, returns values from only the `__raw__` column:

```
=> select * from myflextable_view;
```

Also understand that rebuilding the view after promoting virtual columns changes the resulting value. Future queries return values from both virtual columns in the map data and from real columns.

Using COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

Call this function with a flex table to compute Flex table keys (see Computing Flex Table Keys), and create a view in one step.

Querying Flex Tables

After you create your flex table (with or without additional columns) and load data, you can perform four types of queries:

- SELECT
- COPY
- TRUNCATE
- DELETE

You can use SELECT queries for virtual columns that exist in the `__raw__` column and real columns in your flex tables. Column names are case insensitive.

Unsupported DDL and DML Statements

You cannot use the following DDL and DML statements with flex tables:

```
CREATE TABLE flex_table AS...CREATE TABLE flex_table LIKE...
SELECT INTO
UPDATE
MERGE
INSERT INTO
```

Querying Flex Table Keys

If you reference an undefined column (`'which_column'`) in a flex table query, Vertica converts the query to a call to the maplookup() function as follows:

```
maplookup(__raw__, 'which_column')
```

The maplookup() function searches the VMap data for the requested key and returns the following information:

- String values associated with the key for a row.
- `NULL` if the key is not found.

For more information about handling NULL values, see MAPCONTAINSKEY().

Determining Flex Table Data Contents

If you don't know what your flex table contains, two helper functions let you explore that data to determine contents. Use these functions to compute the keys in the flex table `__raw__` column and, optionally, build a view based on those keys:

- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

For more information about these and other helper functions, see Flex Data Functions Reference

To determine what virtual columns exist:

1. Call the function as follows:

```
            => select compute_flextable_keys('darkdata');
            compute_flextable_keys
--------------------------------------------------
 Please see public.darkdata_keys for updated keys(1 row)
```

2. View the computed key names by querying the `darkdata_keys` table:

```
=> select * from darkdata_keys;
```

```
                         key_name                        | frequency |   data_type_guess
---------------------------------------------------------+-----------+-------------
 contributors                                            |         8 | varchar(20)
 coordinates                                             |         8 | varchar(20)
 created_at                                              |         8 | varchar(60)
 entities.hashtags                                       |         8 | long varbinary(186)
 .
 .
 retweeted_status.user.time_zone                         |         1 | varchar(20)
 retweeted_status.user.url                               |         1 | varchar(68)
 retweeted_status.user.utc_offset                        |         1 | varchar(20)
 retweeted_status.user.verified                          |         1 | varchar(20)
(125 rows)
```

Querying Virtual Columns

Continuing with the JSON data example, use select queries to explore content from the virtual columns. Then, analyze what's most important to you. This example shows querying some common virtual columns in the map data:

```
=> select "user.name", "user.lang", "user.geo_enabled" from darkdata1;
      user.name      | user.lang | user.geo_enabled
---------------------+-----------+------------------
 laughing at clouds. | it        | T
 Avita Desai         | en        | F
 I'm Toasterâ¥       | es        | T
                     |           |
                     |           |
                     |           |
 Uptown gentleman.   | en        | F
 ~G A B R I E L A â¿ | en        | F
 Flu Beach           | es        | F
                     |           |
 seydo shi           | tr        | T
 The End             | en        | F
(12 rows)
```

Using Functions and Casting in Flex Table Queries

You can cast the virtual columns as required and use functions in your select queries. The next example queries the darkdata1 flex table for the `created_at` and `retweet_count` virtual columns, casting their values in the process:

```
=> select "created_at"::TIMESTAMP, "retweet_count"::INT from darkdata1 order by 1 desc;
     created_at      | retweet_count
---------------------+---------------
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:04 |             1
                     |
                     |
                     |
                     |
(12 rows)
```

The following query uses the COUNT and AVG functions to determine the average length of text in different languages:

```
=> select "user.lang", count (*), avg(length("text"))::int from darkdata1 group by 1 order by 2
desc;
 user.lang | count | avg
-----------+-------+-----
 en        |     4 |  42
           |     4 |
 es        |     2 |  96
 it        |     1 |  50
```

```
 tr        |     1 |  16
(5 rows)
```

Casting Data Types in a Query

The following query requests the values of the `created_at` virtual column, without casting to a specific data type:

```
=> select "created_at" from darkdata1;
           created_at
--------------------------------
 Mon Oct 15 18:41:04 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
(12 rows)
```

The next example queries the same virtual column, casting `created_at` to a TIMESTAMP. Casting results in different output and the regional time:

```
=> select "created_at"::TIMESTAMP from darkdata1 order by 1 desc;
     created_at
---------------------
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:04
```

Accessing an Epoch Key

The term *EPOCH* (all uppercase letters) is reserved in Vertica for internal use.

If your JSON data includes a virtual column called `epoch`, you can query it within your flex table. However, use the `maplookup()` function to do so.

Querying Flex Views

Flex tables offer the ability of dynamic schema through the application of query rewriting. Use flex views to support restricted access to flex tables. As with flex tables, each time you use a `select` query on a flex table view, internally, Vertica invokes the `maplookup()` function, to return information on all virtual columns. This query behavior occurs for any flex or columnar table that includes a `__raw__` column.

This example illustrates querying a flex view:

1. Create a flex table.

```
=> CREATE FLEX TABLE twitter();
```

2. Load JSON data into flex table using `fjsonparser`.

```
=> COPY twitter FROM '/home/dbadmin/data/flex/tweets_10000.json' PARSER fjsonparser();
Rows Loaded
-------------
10000
(1 row)
```

3. Create a flex view on top of flex table `twitter` with constraint `retweet_count>0`.

```
=> CREATE VIEW flex_view AS SELECT __raw__ FROM twitter WHERE retweet_count::int > 0;
CREATE VIEW
```

4. Query the view. First 5 rows are displayed.

```
=> SELECT retweeted,retweet_count,source FROM (select __raw__ from flex_view) t1 limit 5;
retweeted | retweet_count |                                         source
-----------+---------------+------------------------------------------------------------------
--------------------
F         | 1             | <a href="http://blackberry.com/twitter" rel="nofollow">Twitter for
BlackBerry®</a>
F         | 1             | web
F         | 1             | <a href="http://twitter.com/download/iphone"
rel="nofollow">Twitter for iPhone</a>
F         | 23            | <a href="http://twitter.com/download/android"
rel="nofollow">Twitter for Android</a>
F         | 7             | <a href="http://twitter.com/download/iphone"
rel="nofollow">Twitter for iPhone</a>
(5 rows)
```

Listing Flex Tables

You can determine which tables in your database are flex tables by querying the `is_flextable` column of the `v_catalog.tables` system table. For example, use a query such as the following to see all tables with a true (`t`) value in the `is_flextable` column:

```
=> select table_name, table_schema, is_flextable from v_catalog.tables;
     table_name      | table_schema | is_flextable
---------------------+--------------+-----------------
 bake1               | public       | t
 bake1_keys          | public       | f
 del                 | public       | t
 del_keys            | public       | f
 delicious           | public       | t
 delicious_keys      | public       | f
 bake                | public       | t
 bake_keys           | public       | f
 appLog              | public       | t
 appLog_keys         | public       | f
 darkdata            | public       | t
 darkdata_keys       | public       | f
(12 rows)
```

Setting Flex Table Parameters

Two configuration parameters affect flex table usage:

Name	Description and Use
`FlexTableRawSize`	Determines the default column width for the `__raw__` column of a new flex table. The `__raw__` column contains the map data you load into the table. The column data type is a `LONG VARBINARY`. Setting this configuration parameter does not affect any existing flex tables. However, doing so changes the default width for any flex tables you create after changing `FlexTableRawSize`. To change an existing flex table, use the ALTER TABLE statement, described in Materializing Flex Tables. **Default:** `130000` **Value range:** `1 - 32000000`
`FlexTableDataTypeGuessMultiplier`	Specifies the multiplier used to set column widths when casting columns from a `LONG VARBINARY` data type for flex table views. Multiplying the longest column member by the factor pads the column width to support subsequent data loads. There is no way to determine the column width of future loads. Thus,padding adds a buffer to support values at least twice that of the previously longest value. These functions update the column width with each invocation:

Name	Description and Use
	• COMPUTE_FLEXTABLE_KEYS • COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW **Default:** `2.0`: The column width multiplier. Must be a value within the following range. **Range (in bytes):** Any value that results in a column width neither less than 20 bytes nor greater than the `FlexTableRawSize` value. This range is a cap to round sizes up or down, accordingly.

Note: The `FlexTableDataTypeGuessMultiplier` value is not used to calculate the width of any real columns. If a flex table has defined columns, their width is set by their data type, such as `80` for a `VARCHAR`.

For more information, see General Parameters in the Administrator's Guide.

Flex Data Functions Reference

The flex table data helper functions supply information you need to query the data you load. For example, suppose you don't know what keys are available in the map data. If not, you can use the COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW function to populate a keys table and build a view. The functions aid in querying flex table and other VMap data.

Function	Description
COMPUTE_ FLEXTABLE_KEYS	Computes map keys from the map data in a `flextable_data` table, and populates the `flextable_data_keys` table with the computed keys. Use this function before building a view.
BUILD_ FLEXTABLE_VIEW	Uses the keys in the `flextable_data_keys` table to create a view definition (`flextable_data_view`) for the `flextable_ data` table. Use this function after computing flex table keys.
COMPUTE_ FLEXTABLE_ KEYS_AND_ BUILD_VIEW	Performs both of the preceding functions in one call.
MATERIALIZE_ FLEXTABLE_ COLUMNS	Materializes a default number of columns (50) or more or less, if specified.
RESTORE_ FLEXTABLE_ DEFAULT_KEYS_ TABLE_AND_VIEW	Replaces the `flextable_data_keys` table and the `flextable_data_view`, linking both the keys table and the view to the parent flex table.

While the functions are available to all users, they are applicable only to:

- Flex tables
- Associated *flex_table*_keys tables
- Associated *flex_table*_view views

By computing keys and creating views from flex table data, the functions allow you to perform SELECT queries. One function restores the original keys table and view that you specified when you first created the flex table.

Flex Table Dependencies

Each flex table (*flextable*) has two dependent objects:

- *flextable*_keys
- *flextable*_view

While both objects are dependent on their parent table, (*flextable*), you can drop either object independently. Dropping the parent table removes both dependents, without a CASCADE option.

Associating Flex Tables and Views

The helper functions automatically use the dependent table and view if they are internally linked with the parent table. You create both when you create the flex table. You can you drop either the _keys table or the _view, and re-create objects of the same name. However, if you do so, the new objects are not internally linked with the parent flex table.

In this case, you can restore the internal links of these objects to the parent table. To do so, drop the _keys table and the _view before calling the RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW function. Calling this function re-creates either, or both, the `_keys` table and the `_view`.

The remaining helper functions perform the tasks described in this section.

BUILD_FLEXTABLE_VIEW

Creates, or re-creates, a view for a default or user-defined `_keys` table, ignoring any empty keys.

Syntax

```
build_flextable_view('flex_table' [ [,'view_name'] [,'user_keys_table'] ])
```

Arguments

flex_table	The flex table name. By default, this function builds or rebuilds a view for the input table with the current contents of the associated `flex_table_keys` table.

view_name	[Optional] A custom view name. Use this option to build or rebuild a new or existing view of your choice for the input table. This option allows you to use the current contents of the associated *`flex_table_keys`* table, rather than the default view (*`flex_table_view`*).
user_keys_ table	[Optional] Specifies a keys table from which to create a view. Use this option if you created a custom `user_keys` table for keys of interest from the flex table map data, rather than the default `flex_table_keys` table. The function builds a view from the keys in *`user_keys`* table, rather than from the *`flex_table_keys`* table.

Examples

The following examples show how to call `build_flextable_view` with 1, 2, or 3 arguments.

Creating a Default View

To create, or re-create, a default view:

1. Call the function with a single argument of a flex table, `darkdata`:

```
=> select build_flextable_view('darkdata');
                  build_flextable_view
-----------------------------------------------------
 The view public.darkdata_view is ready for querying
(1 row)
```

   The function creates a view from the darkdata_keys table.

2. Query from the default view name, (darkdata_`view`):

```
=> select "user.id" from darkdata_view;
  user.id
-----------
 340857907
 727774963
 390498773
 288187825
 164464905
 125434448
 601328899
 352494946
(12 rows)
```

Creating a Custom Name View

To create, or re-create, a default view with a custom name:

1. Call the function with two arguments, a flex table, `darkdata`, and the name of the view to create, dd_view:

```
=> select build_flextable_view('darkdata', 'dd_view');
            build_flextable_view
-----------------------------------------------
 The view public.dd_view is ready for querying
(1 row)
```

2. Query from the custom view name (`dd_view`):

```
=> select "user.lang" from dd_view;
 user.lang
-----------
 tr
 en
 es
 en
 en
 it
 es
 en
(12 rows)
```

Creating a View from a Custom Keys Table

To create a view from a custom `_keys` table with `build_flextable_view`, the table must already exist. The custom table must have the same schema and table definition as the default table (`darkdata_keys`).

Create a custom keys table, using any of these three approaches: to :

1. Create a custom keys table, using any of these three approaches:

   - Create a table with the all keys from the keys table:

```
=> create table new_darkdata_keys as select * from darkdata_keys;
CREATE TABLE
```

2. - Alternatively, create a table based on the default keys table, but without content:

```
=> create table new_darkdata_keys as select * from darkdata_keys LIMIT 0;
CREATE TABLE
kdb=> select * from new_darkdata_keys;
 key_name | frequency | data_type_guess
----------+-----------+-----------------
(0 rows)
```

3. - Given an existing table (or creating one with no data), insert one or more keys:

```
=> create table dd_keys as select * from darkdata_keys limit 0;
CREATE TABLE
=> insert into dd_keys (key_name) values ('user.lang');
 OUTPUT
--------
      1
(1 row)
=> insert into dd_keys (key_name) values ('user.name');
 OUTPUT
--------
      1
(1 row)
=> select * from dd_keys;
 key_name  | frequency | data_type_guess
-----------+-----------+-----------------
 user.lang |           |
 user.name |           |
(2 rows)
```

2. After you create your custom keys table, call the function, as shown, including:
   - All arguments
   - A flex table
   - The name of the view to create
   - The custom keys table

```
=> select build_flextable_view('darkdata', 'dd_view', 'new_darkdata_keys');
            build_flextable_view
-----------------------------------------------
 The view public.dd_view is ready for querying
(1 row)
```

3. Query the new view:

```
SELECT * from dd_view;
```

See Also

- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- MATERIALIZE_FLEXTABLE_COLUMNS
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

COMPUTE_FLEXTABLE_KEYS

Computes the virtual columns (keys and values) from the VMap data of a flex table and repopulates the associated `_keys` table. The keys table has the following columns:

- `key_name`
- `frequency`
- `data_type_guess`

This function sorts the keys table by `frequency` and `key_name`.

Use this function to compute keys without creating an associated table view. To also build a view, use COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW.

Syntax

compute_flextable_keys('*flex_table*')

Arguments

flex_table	The name of a flex table.

Examples

During execution, this function determines a data type for each virtual column. It casts the values computed to `VARCHAR`, `LONG VARCHAR`, or `LONG VARBINARY`. Casting choice depends on the length of the key and whether the key includes nested maps.

The following examples illustrate this function. It showsresults of populating the _keys table, after you create a flex table (darkdata1) and load data:

```
=> create flex table darkdata1();
CREATE TABLE
=> copy darkdata1 from '/test/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
=> select compute_flextable_keys('darkdata1');
            compute_flextable_keys
--------------------------------------------------
 Please see public.darkdata1_keys for updated keys
(1 row)
=> select * from darkdata1_keys;
                         key_name                         | frequency |   data_type_guess
```

```
-----------------------------------------------------------+-----------+---------------------
 contributors                                               |         8 | varchar(20)
 coordinates                                                |         8 | varchar(20)
 created_at                                                 |         8 | varchar(60)
 entities.hashtags                                          |         8 | long varbinary(186)
 entities.urls                                              |         8 | long varbinary(32)
 entities.user_mentions                                     |         8 | long varbinary(674)
.
.
.
 retweeted_status.user.time_zone                            |         1 | varchar(20)
 retweeted_status.user.url                                  |         1 | varchar(68)
 retweeted_status.user.utc_offset                           |         1 | varchar(20)
 retweeted_status.user.verified                             |         1 | varchar(20)
(125 rows)
```

The flex keys table has these columns:

Column	Description
`key_name`	The name of the virtual column (key).
`frequency`	The number of times the virtual column occurs in the map.
`data_type_guess`	The data type for each virtual column. This value is cast to `VARCHAR`, `LONG VARCHAR` or `LONG VARBINARY`. Casting depends on the length of the key and whether the key includes one or more nested maps. In the `_keys` table output, the `data_type_guess` column values are also followed by a value in parentheses, such as `varchar(20)`. The value indicates the padded width of the key column. The width is determined by calculating the longest field, multiplied by the `FlexTableDataTypeGuessMultiplier` configuration parameter value. For more information, see Setting Flex Table Parameters.

See Also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- MATERIALIZE_FLEXTABLE_COLUMNS
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

Combines the functionality of BUILD_FLEXTABLE_VIEW and COMPUTE_FLEXTABLE_KEYS to compute virtual columns (keys) from the VMap data of a flex table and construct a view. Creating a view with this function ignores empty keys. If you don't need to perform both operations together, use one of the single-operation functions instead.

Syntax

`compute_flextable_keys_and_build_view('`*`flex_table`*`')`

Arguments

flex_table	The name of a flex table.

Examples

This example shows how to call the function for the darkdata flex table.

```
=> select compute_flextable_keys_and_build_view('darkdata');
              compute_flextable_keys_and_build_view
-----------------------------------------------------------------------
 Please see public.darkdata_keys for updated keys
The view public.darkdata_view is ready for querying
(1 row)
```

See Also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS
- MATERIALIZE_FLEXTABLE_COLUMNS
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

MATERIALIZE_FLEXTABLE_COLUMNS

Materializes virtual columns listed as *key_names* in the *flextable_keys* table you compute using either COMPUTE_FLEXTABLE_KEYS or COMPUTE_FLEXTABLE_

KEYS_AND_BUILD_VIEW.

> **Note:** Each column materialized with this function counts against the data storage limit of your VerticaPremium Edition license, affecting your next Vertica license compliance audit. To manually check your Premium Edition license compliance, call the `audit()` function.

Syntax

```
materialize_flextable_columns('flex_table' [, n-columns [, keys_
table_name] ])
```

Arguments

flex_table	The name of the flex table with columns to materialize. Specifying only the flex table name attempts to materialize up to 50 columns of key names in the default *flex_table_keys* table. When you use this argument, the function: • Skips any columns already materialized • Ignores any empty keys To materialize a specific number of columns, use the optional parameter `n_columns`, described next.
n-columns	[Optional] The number of columns to materialize. The function attempts to materialize the number of columns from the `flex_table_keys` table, skipping any columns already materialized. Vertica tables support a total of 1600 columns, which is the largest value you can specify for `n-columns`. The function orders the materialized results by frequency, descending, *key_name* when materializing the first `n` columns.
keys_table_name	[Optional] The name of a flex_keys_table from which to materialize columns. The function: • Materializes the number of columns (value of *n-columns*) from *keys_table_name* • Skips any columns already materialized • Orders the materialized results by frequency, descending, *key_name* when materializing the first `n` columns.

Examples

The following example shows how to call `materialize_flextable_columns` to materialize columns. First, load a sample file of tweets (`tweets_10000.json`) into the flex table `twitter_r`.

After loading data and computing keys for the sample flex table, then call `materialize_flextable_columns` to materialize the first four columns:

```
=> copy twitter_r from '/home/release/KData/tweets_10000.json' parser fjsonparser();
 Rows Loaded
-------------
       10000
(1 row)

=> select compute_flextable_keys ('twitter_r');
              compute_flextable_keys
---------------------------------------------------
 Please see public.twitter_r_keys for updated keys
(1 row)

=> select materialize_flextable_columns('twitter_r', 4);
    materialize_flextable_columns
-------------------------------------------------------------------------------
 The following columns were added to the table public.twitter_r:
        contributors
        entities.hashtags
        entities.urls
For more details, run the following query:
SELECT * FROM v_catalog.materialize_flextable_columns_results WHERE table_schema = 'public' and
table_name = 'twitter_r';

(1 row)
```

The last message in the example recommends querying the `materialize_flextable_columns_results` system table for the results of materializing the columns, as shown:

```
=> SELECT * FROM v_catalog.materialize_flextable_columns_results WHERE table_schema = 'public' and
table_name = 'twitter_r';
table_id           | table_schema | table_name |      creation_time           |     key_name      |
status |    message
-------------------+--------------+------------+------------------------------+-------------------
+--------+---------------------------
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.945484-05| contributors      |
ADDED  | Added successfully
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.94551-05 | entities.hashtags |
ADDED  | Added successfully
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.945519-05| entities.urls     |
ADDED  | Added successfully
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.945532-05| created_at        |
EXISTS | Column of same name already exists in table definition
(4 rows)
```

See the MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS system table in the SQL Reference Manual.

See Also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

Restores the `_keys` table and the `_view`. The function also links the `_keys`table with its associated flex table, in cases where either table is dropped, and indicates whether one or both is restored.

Syntax

`restore_flextable_default_keys_table_and_view('`*`flex_table`*`')`

Arguments

flex_table	The name of a flex table .

Examples

This example shows how to invoke this function with an existing flex table, restoring both the `_keys` table and `_view`:

```
        => select restore_flextable_default_keys_table_and_view('darkdata');
                            restore_flextable_default_keys_table_and_view

----------------------------------------------------------------------------------
The keys table public.darkdata_keys was restored successfully.
The view public.darkdata_view was restored successfully.
(1 row)
```

This example shows how you use the function to restore `darkdata_view`. However, the results indicate that `darkdata_keys` does not need restoring:

```
=> select restore_flextable_default_keys_table_and_view('darkdata');
                        restore_flextable_default_keys_table_and_view
-------------------------------------------------------------------------------------------------
 The keys table public.darkdata_keys already exists and is linked to darkdata.
The view public.darkdata_view was restored successfully.
(1 row)
```

The `_keys` table has no content after it is restored:

```
=> select * from darkdata_keys;
 key_name | frequency | data_type_guess
----------+-----------+-----------------
(0 rows)
```

See Also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- MATERIALIZE_FLEXTABLE_COLUMNS

Flex Extractor Functions Reference

The following extractor scalar functions process polystructured data:

- `MAPDELIMITEDEXTRACTOR`
- `MAPJSONEXTRACTOR`
- `MAPREGEXEXTRACTOR`

Each function accepts input data that is:

- Existing database content
- A table
- Returned from an expression
- Entered directly

These functions do not parse data from an external file source. All functions return a single VMap value. The extractor functions can return data with NULL-specified columns.

This section describes each extractor function.

MAPDELIMITEDEXTRACTOR

Extracts data with a delimiter character, and other optional arguments, returning a single VMap value. The `USING PARAMETERS` phrase specifies optional parameters for the function.

Parameters

`delimiter`	VARCHAR	Single delimiter character. **Default value:** |
`header_names`	VARCHAR	[Optional] Specifies header names for columns. **Default value:** `ucol`*n* Where *n* is the column offset number, starting with `0` for the first column. The function uses default values if you do not specify values for the `header_names` parameter.

trim	BOOLEAN	[Optional] Trims white space from header names and field values. **Default value:** `true`
treat_empty_val_as_null	BOOLEAN	[Optional] Specifies that empty fields become `NULLs`, rather than empty strings (`''`). **Default value:** `true`

Examples

These examples use a short set of delimited data:

```
Name|CITY|New city|State|zip
Tom|BOSTON|boston|MA|01
Eric|Burlington|BURLINGTON|MA|02
Jamie|cambridge|CAMBRIDGE|MA|08
```

To begin, save this data as `delim.dat`.

1. Create a flex table, `dflex`:

```
=> create flex table dflex();
CREATE TABLE
```

2. Use COPY to load the `delim.dat` file. Use the flex tables `fdelimitedparser` with the `header='false'` option:

```
=> copy dflex from '/home/release/kmm/flextables/delim.dat' parser fdelimitedparser
(header='false');
 Rows Loaded
-------------
           4
(1 row)
```

3. Create a columnar table, `dtab`, with an identity `id` column, a `delim` column, and a column to hold a VMap, named`vmap`:

```
=> create table dtab (id IDENTITY(1,1), delim varchar(128), vmap long varbinary(512));
CREATE TABLE
```

4. Use COPY to load the `delim.dat` file into the `dtab` table. For the `mapdelimitedextractor` function, add a header row with `USING PARAMETERS header_names=` option to specify the header row for the sample data, along with `delimiter '!'`:

```
=> copy dtab(delim, vmap as mapdelimitedextractor(delim
   USING PARAMETERS header_names='Name|CITY|New City|State|Zip')) FROM
'/home/dbadmin/data/delim.dat' DELIMITER '!';

 Rows Loaded
-------------
           4
(1 row)
```

5. Use `maptostring` for the flex table `dflex` to view the `__raw__` column contents. Notice the default header names in use (`ucol0` – `ucol4`), since you specified `header='false'` when you loaded the flex table:

```
=> select maptostring(__raw__) from dflex limit 10;
                             maptostring
-------------------------------------------------------------------------------------
 {
   "ucol0" : "Jamie",
   "ucol1" : "cambridge",
   "ucol2" : "CAMBRIDGE",
   "ucol3" : "MA",
   "ucol4" : "08"
}

 {
   "ucol0" : "Name",
   "ucol1" : "CITY",
   "ucol2" : "New city",
   "ucol3" : "State",
   "ucol4" : "zip"
}

 {
   "ucol0" : "Tom",
   "ucol1" : "BOSTON",
   "ucol2" : "boston",
   "ucol3" : "MA",
   "ucol4" : "01"
}

 {
   "ucol0" : "Eric",
   "ucol1" : "Burlington",
   "ucol2" : "BURLINGTON",
   "ucol3" : "MA",
   "ucol4" : "02"
}

(4 rows)
```

6. Use `maptostring` again, this time with the `dtab` table's `vmap` column. Compare the results of this output to those for the flex table. Note that `maptostring` returns the `header_name` parameter values you specified when you loaded the data:

```
=> select maptostring(vmap) from dtab;
                                                      maptostring
-----------------------------------------------------------------------------------------------
-------------------------
 {
   "CITY" : "CITY",
   "Name" : "Name",
   "New City" : "New city",
   "State" : "State",
   "Zip" : "zip"
}

 {
   "CITY" : "BOSTON",
   "Name" : "Tom",
   "New City" : "boston",
   "State" : "MA",
   "Zip" : "02121"
}

 {
   "CITY" : "Burlington",
   "Name" : "Eric",
   "New City" : "BURLINGTON",
   "State" : "MA",
   "Zip" : "02482"
}

 {
   "CITY" : "cambridge",
   "Name" : "Jamie",
   "New City" : "CAMBRIDGE",
   "State" : "MA",
   "Zip" : "02811"
}

(4 rows)
```

7. Query the `delim` column to view the contents differently:

```
=> select delim from dtab;
                delim
-------------------------------------
 Name|CITY|New city|State|zip
 Tom|BOSTON|boston|MA|02121
 Eric|Burlington|BURLINGTON|MA|02482
 Jamie|cambridge|CAMBRIDGE|MA|02811
(4 rows)
```

See Also

- MAPJSONEXTRACTOR
- MAPREGEXEXTRACTOR

MAPJSONEXTRACTOR

Extracts content of repeated JSON data objects, including nested maps, or data with an outer list of JSON elements. The `USING PARAMETERS` phrase specifies optional parameters for the function. Empty input does not generate a Warning or Error.

Parameters

`flatten_maps`	BOOLEAN	[Optional] Flattens sub-maps within the JSON data, separating map levels with a period (`.`). **Default value:** `true`
`flatten_arrays`	BOOLEAN	[Optional] Converts lists to sub-maps with integer keys. Lists are not flattened by default. **Default value:** `false`
`reject_on_duplicate`	BOOLEAN	[Optional] Halts the load process if the file being loaded includes duplicate key names, with different case. **Default value:** `false`
`reject_on_empty_key`	BOOLEAN	[Optional] Rejects any row containing a key without a value (`reject_on_empty_key=true`). **Default value:** `false`
`omit_empty_keys`	BOOLEAN	[Optional] Omits any key from the load data that does not have a value (`omit_empty_keys=true`). **Default value:** `false`
`start_point`	CHAR	[Optional] Specifies the name of a key in the JSON load data at which to begin parsing. The parser ignores all data before the `start_point` value.The parser processes data after the first instance, and up to the second, ignoring any remaining data. **Default value:** none

Examples

These examples use the following sample JSON data:

```
{ "id": "5001", "type": "None" }
{ "id": "5002", "type": "Glazed" }
{ "id": "5005", "type": "Sugar" }
{ "id": "5007", "type": "Powdered Sugar" }
{ "id": "5004", "type": "Maple" }
```

We save this content as `bake_single.json`, and load that file.

1. Create a flex table, `flexjson`:

   ```
   => create flex table flexjson();
   CREATE TABLE
   ```

2. Use COPY to load the `bake_single.json` file with the flex tables `fjsonparser` parser:

   ```
   => copy flexjson from '/home/dbadmin/data/bake_single.json' parser fjsonparser();
    Rows Loaded
   -------------
              5
   (1 row)
   ```

3. Create a columnar table, `coljson`, with an identity `id` column, a `json` column, and a column to hold a VMap, called `vmap`:

   ```
   => create table coljson(id IDENTITY(1,1), json varchar(128), vmap long varbinary(10000));
   CREATE TABLE
   ```

4. Use COPY to load the `bake_single.json` file into the `coljson` table, using the `mapjsonextractor` function:

   ```
   => copy coljson (json, vmap AS MapJSONExtractor(json)) FROM '/home/dbadmin/data/bake_
   single.json';
    Rows Loaded
   -------------
              5
   (1 row)
   ```

5. Use the `maptostring` function for the flex table `flexjson` to output the `__raw__` column contents as strings:

   ```
   => select maptostring(__raw__) from flexjson limit 5;
                        maptostring
   ```

```
----------------------------------------------------
 {
   "id" : "5001",
   "type" : "None"
}

 {
   "id" : "5002",
   "type" : "Glazed"
}

 {
   "id" : "5005",
   "type" : "Sugar"
}

 {
   "id" : "5007",
   "type" : "Powdered Sugar"
}

 {
   "id" : "5004",
   "type" : "Maple"
}

(5 rows)
```

6. Use the `maptostring` function again, this time with the `coljson` table's `vmap` column and compare the results. The element order differs:

```
=> select maptostring(vmap) from coljson limit 5;
                     maptostring
----------------------------------------------------
 {
   "id" : "5001",
   "type" : "None"
}

 {
   "id" : "5002",
   "type" : "Glazed"
}

 {
   "id" : "5004",
   "type" : "Maple"
}

 {
   "id" : "5005",
   "type" : "Sugar"
}

 {
   "id" : "5007",
   "type" : "Powdered Sugar"
```

```
}
(5 rows)
```

See Also

- MAPDELIMITEDEXTRACTOR
- MAPREGEXEXTRACTOR

MAPREGEXEXTRACTOR

Extracts data from a regular expression and returns the results as a VMap. Use the `USING PARAMETERS pattern=` phrase, followed by the regular expression.

Parameters

`pattern=`	`VARCHAR`	The regular expression as a string. **Default value:** An empty string ("").
`use_jit`	`BOOLEAN`	[Optional] Uses just-in-time compiling when parsing the regular expression. **Default value:** `false`.
`record_terminator`	`VARCHAR`	[Optional] The character used to separate input records. **Default value:** `\n`.
`logline_column`	`VARCHAR`	[Optional] The destination column containing the full string that the regular expression matched. **Default value:** An empty string ("").

Examples

This example uses the following regular expression to search a Vertica log file for information that includes the `timestamp`, `date`, `thread_name`, and `thread_id`. For illustrative purposes, the regular expression components are shown here on separate lines. Remove any new line characters before using this example in your own tests.

```
^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)(?<thread_name>[A-Za-z ]+):
(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?(?:\[(?<component>\w+)\]
<(?<level>\w+)> )?(?<(?<elevel>\w+)> @\[?(?<enode>\w+)\]?: )?(?<text>.*)
```

The output in the following examples include newline characters for display purposes.

1. Create a flex table, `flogs`:

```
=> create flex table flogs();
CREATE TABLE
```

2. Use COPY to load a sample log file (`vertica.log`), using the flex table `fregexparser`.

```
=> copy flogs from '/home/dbadmin/data/regdata/vertica.log' PARSER fregexparser(pattern=
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+) (?<thread_name>[A-Za-z ]+):
(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?
(?:\[(?<component>\w+)\] <(?<level>\w+)> )?(?:<(?<elevel>\w+)>
@\[?(?<enode>\w+)\]?: )?(?<text>.*)');
 Rows Loaded
-------------
        8434
(1 row)
```

3. Use `MapToString` to return the results from calling `MapRegexExtractor` with a regular expression. The output returns the results of the function in string format.

```
=> select maptostring(MapregexExtractor(E'2014-04-02 04:02:51.011
TM Moveout:0x2aab9000f860-a0000000002067 [Txn] <INFO>
Begin Txn: a0000000002067 \'Moveout: Tuple Mover\'' using PARAMETERS
pattern='^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?
(?:\[(?<component>\w+)\] <(?<level>\w+)> )?(?:<(?<elevel>\w+)> @\[?(?<enode>\w+)\]?: )?
(?<text>.*)')) FROM flogs where __identity__=13;

maptostring
----------------------------------------------------------------------------------------------
----
{
"component" : "Txn",
"level" : "INFO",
"text" : "Begin Txn: a0000000002067 'Moveout: Tuple Mover'",
"thread_id" : "0x2aab9000f860",
"thread_name" : "TM Moveout",
"time" : "2014-04-02 04:02:51.011",
"transaction_id" : "a0000000002067"
}
(1 row)
```

See Also

- MAPDELIMITEDEXTRACTOR
- MAPJSONEXTRACTOR

Flex Map Functions Reference

The flex map functions let you extract and manipulate nested map data.

The first argument of all flex map functions (except `emptymap()` and `mapaggregate()`) takes a VMap. The VMap can originate from the `__raw__` column in a flex table or be returned from a map or extraction function.

All map functions (except for `emptymap()` and `mapaggregate()`), accept either a `LONG VARBINARY` or a `LONG VARCHAR` map argument.

In the following example, the outer `maplookup()`function operates on the VMap data returned from the inner `maplookup()` function:

```
=> maplookup(maplookup(ret_map, 'batch'), 'scripts')
```

You can use flex map functions with:

- Flex tables
- Their associated *{flextable}*_keys table
- Automatically generated *{flextable}*_view views.

However, use of these functions does not apply to standard Vertica tables.

EMPTYMAP

Constructs a new VMap with one row but without keys or data. Use this transform function to populate a map without using a flex parser. Instead, you use either from SQL queries or from map data present elsewhere in the database.

Syntax

`emptymap()`

Arguments

None

Examples

Create an Empty Map

```
=> select emptymap();
```

```
                              emptymap
--------------------------------------------------------------------
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
(1 row)
```

Create an Empty Map from an Existing Flex Table

If you create an empty map from an existing flex table, the new map has the same number of rows as the table from which it was created.

This example shows the result if you create an empty map from the `darkdata` table, which has 12 rows of JSON data:

```
=> select emptymap() from darkdata;
                              emptymap
--------------------------------------------------------------------
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
(12 rows)
```

See Also

- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING

- MAPVALUES
- MAPVERSION

MAPAGGREGATE

Returns a LONG VARBINARY VMap with keys and value pairs supplied from two VARCHAR input columns of an existing columnar table. Using this function requires specifying an over() clause for the source table.

Syntax

mapaggregate(*source_column1*, *source_column2*)

Arguments

source_column1	Table column with values to use as the keys of the key/value pair of the returned VMap data.
source_column2	Table column with values to use as the values in the key/value pair of the returned VMap data.

Examples

This example creates a columnar table btest, with two VARCHAR columns, named keys and values, and adds three sets of values:

```
=> create table btest(keys varchar(10), values varchar(10));
CREATE TABLE
=>  copy btest from stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> one|1
>> two|2
>> three|3
>> \.
```

After populating the btest table, call mapaggregate(), using the the over (PARTITION BEST) clause. This call returns the raw_map data:

```
=>  select mapaggregate(keys, values) over(PARTITION BEST) from btest;
                 raw_map
-----------------------------------------------------------------------------------------------------
-
-----------------------------------------------------------------------------
```

```
\001\000\000\000\023\000\000\000\003\000\000\000\020\000\000\000\021\000\000\000\022\000\000\000132
\003\000\000\000\020\000\000\000\023\000\000\000\030\000\000\000onethreetwo
(1 row)
```

The next example illustrates using MAPTOSTRING() with the returned `raw_map` from `mapaggregate()` to see the values:

```
=> select maptostring(raw_map) from (select mapaggregate(keys, values) over(PARTITION BEST) from
btest) bit;
                 maptostring
----------------------------------------------
 {
        "one":  "1",
        "three":        "3",
        "two":  "2"
 }
(1 row)
```

See Also

- EMPTYMAP
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPCONTAINSKEY

Determines whether a VMap contains a virtual column (key). This scalar function returns true (`t`), if the virtual column exists, or false (`f`) if it does not. Determining that a key exists before calling `maplookup()` lets you distinguish between NULL returns. The `maplookup()` function uses for both a non-existent key and an existing key with a NULL value.

Syntax

mapcontainskey(*VMap_data*, 'virtual_column_name')

Arguments

`VMap_data`	Any VMap data. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content
`virtual_column_name`	The name of the key to check.

Examples

This example shows how to use the `mapcontainskey()` functions with `maplookup()`. View the results returned from both functions. Check whether the empty fields that `maplookup()` returns indicate a `NULL` value for the row (t) or no value (`f`):

You can use mapcontainskey( ) to determine that a key exists before calling maplookup (). The maplookup() function uses both NULL returns and existing keys with NULL values to indicate a non-existent key.

```
=> select maplookup(__raw__, 'user.location'), mapcontainskey(__raw__, 'user.location') from
darkdata order by 1;
 maplookup | mapcontainskey
-----------+----------------
           | t
           | t
           | t
           | t
 Chile     | t
 Narnia    | t
 Uptown..  | t
 chicago   | t
           | f
           | f
           | f
           | f

(12 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPCONTAINSVALUE

Determines whether a VMap contains a specific value. Use this scalar function to return true (`t`), if the value exists, or false (`f`), if it does not.

Syntax

```
mapcontainsvalue(VMap_data, 'virtual_column_value')
```

Arguments

`VMap_data`	Any VMap data. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content
`virtual_column_value`	The value whose existence you want to confirm.

Examples

This example shows how to use `mapcontainsvalue()` to determine whether or not a virtual column contains a particular value. Create a flex table (`ftest`), and populate it with some virtual columns and values. Name both virtual columns `one`:

```
=> create flex table ftest();
CREATE TABLE
=> copy ftest from stdin parser fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one":1, "two":2}
>> {"one":"one","2":"2"}
>> \.
```

Call `mapcontainsvalue()` on the `ftest` map data. The query returns false (`f`) for the first virtual column, and true (`t`) for the second , which contains the value `one`:

```
=> select mapcontainsvalue(__raw__, 'one') from ftest;
mapcontainsvalue
------------------
f
t
(2 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPITEMS

Returns information about items in a VMap. Use this transform function with one or more optional arguments to access polystructured values within the VMap data. This function requires an `over()` clause.

Syntax

`mapItems(VMap_data [, passthrough_arg [,...] ])`

Arguments

`VMap_data`	Any VMap data. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content
`passthrough_arg`	[Optional] One or more arguments indicating keys within the map data in `VMap_data`

Examples

These examples show how to use `mapItems()`in queries. All examples use `over (PARTITION BEST)`.

Determine the Number of Virtual Columns in the Map Data

Use a flex table, labeled `darkmountain`, populated with JSON data. Query using the `count()` function. This query returns the number of virtual columns found in the map data:

```
=> select count(keys) from (select mapitems(darkmountain.__raw__) over(PARTITION BEST) from
darkmountain) as a;
 count
-------
    19
(1 row)
```

Determine What Data a Map Contains

You can also query a flex table for all of all items in the map data, as this snippet shows:

```
=> select * from (select mapitems(darkmountain.__raw__) over(PARTITION BEST) from darkmountain) as
```

```
a;
    keys     |    values
-------------+---------------
 hike_safety | 50.6
 name        | Mt Washington
 type        | mountain
 height      | 17000
 hike_safety | 12.2
 name        | Denali
 type        | mountain
 height      | 29029
 hike_safety | 34.1
 name        | Everest
 type        | mountain
 height      | 14000
 hike_safety | 22.8
 name        | Kilimanjaro
 type        | mountain
 height      | 29029
 hike_safety | 15.4
 name        | Mt St Helens
 type        | volcano
(19 rows)
```

Directly Query a Key Value in a VMap

Review the following JSON input file, `simple.json`. In particular, notice the array called `three_Array`, and its four values:

```
{
  "one": "one",
  "two": 2,
  "three_Array":
  [
    "three_One",
    "three_Two",
    3,
    "three_Four"
  ],
  "four": 4,
  "five_Map":
  {
    "five_One": 51,
    "five_Two": "Fifty-two",
    "five_Three": "fifty three",
    "five_Four": 54,
    "five_Five": "5 x 5"
  },
  "six": 6
}
```

1. Create a flex table, mapper:

   ```
   => create flex table mapper();
   CREATE TABLE
   ```

2. Load `simple.json` into the flex table mapper:

```
=> copy mapper from '/home/dbadmin/data/simple.json' parser fjsonparser (flatten_arrays=false,
flatten_maps=false);
 Rows Loaded
-------------
           1
(1 row)
```

3. Call `mapkeys` on the flex table's `__raw__` column to see the flex table's keys, but not the key submaps. The return values indicate `three_Array` as one of the virtual columns:

```
=> select mapkeys(__raw__) over() from mapper;
    keys
-------------
 five_Map
 four
 one
 six
 three_Array
 two
(6 rows)
```

4. Call `mapitems` on flex table `mapper` with `three_Array` as a pass-through argument to the function. The call returns these array values:

```
=> select __identity__, mapitems(three_Array) OVER(PARTITION BY __identity__) from mapper;
 __identity__ | keys |   values
--------------+------+------------
            1 | 0    | three_One
            1 | 1    | three_Two
            1 | 2    | 3
            1 | 3    | three_Four
(4 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP

- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPKEYS

Returns the virtual columns (and values) present in any VMap data. This transform function requires an `over(PARTITION BEST)` clause.

Syntax

```
mapkeys(VMap_data)
```

Arguments

VMap_data	Any VMap data. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content

Examples

Determine Number of Virtual Columns in Map Data

This example shows how to create a query, using an `over(PARTITION BEST)` clause with a flex table, `darkdata` to find the number of virtual column in the map data. The table is populated with JSON tweet data.

```
=> select count(keys) from (SELECT mapkeys(darkdata.__raw__) over(PARTITION BEST) from darkdata) as
a;
 count
-------
   550
(1 row)
```

Query Ordered List of All Virtual Columns in the Map

This example shows a snippet of the return data when you query an ordered list of all virtual columns in the map data:

```
=> select * from (SELECT mapkeys(darkdata.__raw__) over(PARTITION BEST) from darkdata) as a;
    keys
-------------------------------------
 contributors
 coordinates
 created_ at
 delete.status.id
 delete.status.id_str
 delete.status.user_id
 delete.status.user_id_str
 entities.hashtags
 entities.media
 entities.urls
 entities.user_mentions
 favorited
 geo
 id
.
.
.
 user.statuses_count
 user.time_zone
 user.url
 user.utc_offset
 user.verified
(125 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING

- MAPVALUES
- MAPVERSION

MAPKEYSINFO

Returns virtual column information from a given map. This transform function requires an `over(PARTITION BEST)` clause.

Syntax

```
mapkeysinfo(VMap_data)
```

Arguments

`VMap_data`	Any VMap data. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content

Returns

This function is a superset of the MAPKEYS() function. It returns the following information about each virtual column:

Column	Description
`keys`	The virtual column names in the raw data.
`length`	The data length of the key name, which can differ from the actual string length.
`type_oid`	The OID type into which the value should be converted. Currently, the type is always `116` for a `LONG VARCHAR`, or `199` for a nested map that is stored as a `LONG VARBINARY`.
`row_num`	The number of rows in which the key was found.
`field_num`	The field number in which the key exists.

Examples

This example shows a snippet of the return data you receive if you query an ordered list of all virtual columns in the map data:

```
=> select * from (SELECT mapkeysinfo(darkdata.__raw__) over(PARTITION BEST) from darkdata) as a;
                          keys                                | length | type_oid | row_num | field_num
----------------------------------------------------------+--------+----------+---------+----------
-
 contributors                                              |      0 |      116 |       1 |         0
 coordinates                                               |      0 |      116 |       1 |         1
 created_at                                                |     30 |      116 |       1 |         2
 entities.hashtags                                         |     93 |      199 |       1 |         3
 entities.media                                            |    772 |      199 |       1 |         4
 entities.urls                                             |     16 |      199 |       1 |         5
 entities.user_mentions                                    |     16 |      199 |       1 |         6
 favorited                                                 |      1 |      116 |       1 |         7
 geo                                                       |      0 |      116 |       1 |         8
 id                                                        |     18 |      116 |       1 |         9
 id_str                                                    |     18 |      116 |       1 |        10
.
.
.
 delete.status.id                                          |     18 |      116 |      11 |         0
 delete.status.id_str                                      |     18 |      116 |      11 |         1
 delete.status.user_id                                     |      9 |      116 |      11 |         2
 delete.status.user_id_str                                 |      9 |      116 |      11 |         3
 delete.status.id                                          |     18 |      116 |      12 |         0
 delete.status.id_str                                      |     18 |      116 |      12 |         1
 delete.status.user_id                                     |      9 |      116 |      12 |         2
 delete.status.user_id_str                                 |      9 |      116 |      12 |         3
(550 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPLOOKUP
- MAPSIZE

- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPLOOKUP

Returns values associated with a single key. This scalar function returns a LONG VARCHAR, with virtual column values, or NULL, if the virtual column does not exist. Column names are case insensitive.

Before using maplookup(), you can use these two functions to find out about your VMap data:

- MAPTOSTRING returns the contents of map data in a formatted text output
- MAPCONTAINSKEY determines whether a key exists in the map data

You can control the behavior for non-scalar values when loading data with the fjsonparser parser and its flatten-arrays argument. See Loading JSON Data and the FJSONPARSER reference.

Syntax

```
maplookup (VMap_data, 'virtual_column_name' [USING PARAMETERS [case_
sensitive={false | true}] [, buffer_size=n] ] )
```

Arguments and Parameters

VMap_data	Any VMap data. The VMap can exist as: • The __raw__ column of a flex table • Data returned from a map function such as maplookup() • Other database content
virtual_column_name	The name of the virtual column whose values this function returns.
buffer_size	[Optional parameter] Specifies the maximum length (in bytes) of each value returned for *virtual_*

	column_name. To return all values for `virtual_column_name`, specify a `buffer_size` equal to or greater than (`=>`) the number of bytes for any returned value. Any returned values greater in length than `buffer_size` are rejected. **Default value:** `0` (No limit on `buffer_size`)
`case_sensitive`	[Optional parameter] Specifies whether to return values for *virtual_column_name* if keys with different cases exist. **Example:** `(... USING PARAMETERS case_sensitive=true)` **Default value:** `false`

Examples

The following examples show ways you can use `maplookup`.

Return Row Values of One Virtual Column

Return the row values of one virtual column, `user.location`:

```
=>  select maplookup(__raw__, 'user.location') from darkdata order by 1;
 maplookup
-----------
 Chile
 Narnia
 Uptown
 .
 .
 chicago
(12 rows)
```

Specify Buffer Size for Returned Values

Use the `buffer_size=` parameter to indicate the maximum length of any value that maplookup returns for the virtual column specified. If none of the returned key values can be greater than `n` bytes, you can use this parameter to allocate `n` bytes as the `buffer_size`.

In this example, simple JSON data is saved into a file, `simple_name.json`:

```
{
  "name": "sierra",
  "age": "63",
  "eyes": "brown",
```

```
  "weapon": "doggie"
}
{
  "name": "janis",
  "age": "10",
  "eyes": "blue",
  "weapon": "humor"
}
{
  "name": "ben",
  "age": "43",
  "eyes": "blue",
  "weapon": "sword"
}
{
  "name": "jen",
  "age": "38",
  "eyes": "green",
  "weapon": "shopping"
}
```

1. Create a flex table, `logs`.
2. Load the `simple_name.json` data into `logs`, using the `fjsonparser`. Specify the `flatten_arrays` option as `True`:

```
=> COPY logs FROM '/home/dbadmin/data/simple_name.json' PARSER fjsonparser(flatten_
arrays=True);
```

3. Query the file to display the `name` key:

```
=> select name from logs ORDER BY name;
  name
--------
 ben
 janis
 jen
 sierra
(4 rows)
```

4. Use `maplookup` with `buffer_size=0` for the `logs` table `name` key. This query returns all of the values:

```
=> SELECT MapLookup(__raw__, 'name' USING PARAMETERS buffer_size=0) FROM logs;
 MapLookup
-----------
 sierra
 ben
 janis
 jen
(4 rows)
```

5. Next, specify h `buffer_size` `3`, `5`, and `6`. Now, `maplookup()` returns only values with a byte length less than or equal to (<=), the specified `buffer_size`:

```
=> SELECT MapLookup(__raw__, 'name' USING PARAMETERS buffer_size=3) FROM logs;
 MapLookup
-----------

 ben

 jen
(4 rows)
=> SELECT MapLookup(__raw__, 'name' USING PARAMETERS buffer_size=5) FROM logs;
 MapLookup
-----------

 janis
 jen
 ben
(4 rows)
=> SELECT MapLookup(__raw__, 'name' USING PARAMETERS buffer_size=6) FROM logs;
 MapLookup
-----------
 sierra
 janis
 jen
 ben
(4 rows)
```

Interpreting Empty Fields

This example show that if you use `maplookup` without first checking the existence of a key, the output (12 empty rows) is ambiguous. When you review the following output, you cannot determine whether a `location` key:

- Exists
- Exists with a `NULL` value in a row
- Exists without a value

```
=> select maplookup(__raw__, 'user.location') from darkdata;
 maplookup
-----------










```

```
(12 rows)
```

To disambiguate empty rows, use the `mapcontainskey()` function in conjunction with `maplookup()`. When `maplookup` returns an empty field, the corresponding value from `mapcontainskey` indicates `t` for a `NULL` value, or `f`for no value.

The following vsql sample output from both functions indicates rows without values in purple. It also shows location `Narnia` as a value in the `user.location` virtual column:

```
=> select maplookup(__raw__, 'user.location'), mapcontainskey(__raw__, 'user.location')
from darkdata order by 1;
 maplookup | mapcontainskey
-----------+----------------
           | t
           | t
           | t
           | t
 Chile     | t
 Narnia    | t
 Uptown..  | t
 chicago   | t
           | f >>>>>>>>>>No value
           | f >>>>>>>>>>No value
           | f >>>>>>>>>>No value
           | f >>>>>>>>>>No value
(12 rows)
```

Query Data from Nested Maps

To access data contained in nested maps of arbitrary depth, you can call `maplookup()` recursively. The innermost map is always __raw__, just as a single invocation is.

The next example shows how to use `maptostring()` to return the map contents of the table `bake`, so you can see the contents of the map data:

```
=> select maptostring(__raw__) from bake;
           maptostring
--------------------------------------------------------------------------------
 items.item :
 0.batters.batter :
        0.id :  2001
                0.type : Regular
                1.id :  2002
                1.type : Chocolate
                2.id :  2003
                2.type : Blueberry
                3.id :  2004
                3.type : Devil's Food
        0.id :  0002
        0.name :        CupCake
        0.ppu : 0.55
```

```
        0.topping :
        0.id :  6001
                0.type : None
                1.id :  6002
                1.type : Glazed
                2.id :  6005
                2.type : Sugar
                3.id :  6007
                3.type : Powdered Sugar
                4.id :  6006
                4.type : Chocolate with Sprinkles
                5.id :  6003
                5.type : Chocolate
                6.id :  6004
                6.type : Maple
        0.type :        Muffin
(1 row)
```

Next, after creating a flex table and loading data, use several invocations of `maplookup()` to return values from the map `__raw__` column (). Compare the returned results to the `maptostring()` output in the previous example.

```
=> create flex table bake();
CREATE TABLE
kdb=> copy bake from '/vertica/test/flextable/DATA/bake.json' parser fjsonparser(flatten_
arrays=1,flatten_maps=0);
 Rows Loaded
-------------
           1
(1 row)
=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw_
_,'items'),'item.0'),'batters'),'batter.0'),'type') from bake;
 maplookup
-----------
 Regular
(1 row)
=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw_
_,'items'),'item.0'),'batters'),'batter.1'),'type') from bake;
 maplookup
-----------
 Chocolate
(1 row)
=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw_
_,'items'),'item.0'),'batters'),'batter.2'),'type') from bake;
 maplookup
-----------
 Blueberry
(1 row)
=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw_
_,'items'),'item.0'),'batters'),'batter.3'),'type') from bake;
  maplookup
--------------
 Devil's Food
(1 row)
```

Check for Case-Sensitive Virtual Columns

You can use `maplookup()` with the `case_sensitive` parameter to return results when key names with different cases exist.

1. Save the following sample content as a JSON file. This example saves the file as `repeated_key_name.json`:

```
{
  "test": "lower1"
}
{
  "TEST": "upper1"
}
{
  "TEst": "half1"
}
{
  "test": "lower2",
  "TEst": "half2"
}
{
  "TEST": "upper2",
  "TEst": "half3"
}
{
  "test": "lower3",
  "TEST": "upper3"
}
{
  "TEst": "half4",
  "test": "lower4",
  "TEST": "upper4"
}
{

"TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTestte
stTesttestTesttestTesttestTesttest":"1",

"TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTestte
stTesttestTesttestTesttestTest12345":"2"
}
```

2. Create a flex table, `dupe`, and load the JSON file:

```
=> create flex table dupe();
CREATE TABLE
dbt=> copy dupe from '/home/release/KData/repeated_key_name.json' parser fjsonparser();
 Rows Loaded
-------------
           8
(1 row)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPSIZE

Returns the number of virtual columns present in any VMap data. Use this scalar function to determine the size of keys.

Syntax

```
mapsize(VMap_data)
```

Arguments

`VMap_data`	Any VMap data. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content

Examples

This example shows the returned sizes from the number of keys in the flex table `darkmountain`:

```
=> select mapsize(__raw__) from darkmountain;
 mapsize
---------
       3
       4
       4
       4
       4
(5 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPTOSTRING

Recursively builds a string representation VMap data, including nested JSON maps. Use this transform function to display the VMap contents in a readable `LONG VARCHAR` format. Use `maptostring` to see how map data is nested before querying virtual columns with `mapvalues()`.

Syntax

```
maptostring(VMap_data [using parameters canonical_json={true
| false}])
```

Arguments

`VMap_data`	Any VMap data. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content

Parameters

`canonical_json`	*=bool* [Optional parameter] Produces canonical JSON output by default, using the first instance of any duplicate keys in the map data. Use this parameter as other UDF parameters, preceded by `using parameters`, as shown in the examples. Setting this argument to `false` maintains the previous behavior of `maptostring()` and returns same-name keys and their values. **Default value:** `canonical-json=true`

Examples

The following example shows how to create a sample flex table, `darkdata`and load JSON data from STDIN. By calling `maptostring()` twice with both values for the `canonical_json` parameter, you can see the different results on the flex table `__raw__` column data.

1. Create sample table:

```
=> create flex table darkdata();
CREATE TABLE
```

2. Load sample JSON data from STDIN:

```
=> copy darkdata from stdin parser fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"aaa": 1, "aaa": 2, "AAA": 3, "bbb": "aaa\"bbb"}
>> \.
```

3. Call `maptostring()` with its default behavior using canonical JSON output, and then review the flex table contents. The function returns the first duplicate key and its value (`"aaa": "1"`) but omits remaining duplicate keys (`"aaa": "2"`):

```
=> select maptostring (__raw__) from darkdata;
                        maptostring
------------------------------------------------------------
 {
   "AAA" : "3",
   "aaa" : "1",
   "bbb" : "aaa\"bbb"
}
(1 row)
```

4. Next, call `maptostring()` with `using parameters canonical_json=false`). This time, the function returns the first duplicate keys and their values:

```
=> select maptostring(__raw__ using parameters canonical_json=false) from darkdata;
                          maptostring
---------------------------------------------------------------

{
        "aaa":  "1",
        "aaa":  "2",
        "AAA":  "3",
        "bbb":  "aaa"bbb"
 }
(1 row)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS

- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPVALUES
- MAPVERSION

MAPVALUES

Returns a string representation of the top-level values from a VMap. This transform function requires an `over()` clause.

Syntax

`mapvalues(VMap_data)`

Arguments

`VMap_data`	The VMap from which values should be returned. The VMap can exist as: • The `__raw__` column of a flex table • Data returned from a map function such as `maplookup()` • Other database content

Examples

The following example shows how to query a `darkmountain` flex table, using an `over()` clause (in this case, the `over(PARTITION BEST)` clause) with `mapvalues()`.

```
=> select * from (select mapvalues(darkmountain.__raw__) over(PARTITION BEST) from darkmountain) as
a;
    values
---------------
 29029
 34.1
 Everest
 mountain
 29029
 15.4
 Mt St Helens
 volcano
```

```
 17000
 12.2
 Denali
 mountain
 14000
 22.8
 Kilimanjaro
 mountain
 50.6
 Mt Washington
 mountain
(19 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVERSION

MAPVERSION

Returns the version or invalidity of any map data. This scalar function returns the map version (such as `1`) or `-1`, if the map data is invalid.

Syntax

```
mapversion(VMap_data)
```

Arguments

VMap_data	The VMap data either from a `__raw__` column in a flex table or from the data returned from a map function such as `maplookup()`.

Examples

The following example shows how to use `mapversion()` with the `darkmountain`flex table, returning `mapversion 1` for the flex table map data:

```
=> select mapversion(__raw__) from darkmountain;
 mapversion
------------
          1
          1
          1
          1
          1
(5 rows)
```

See Also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES

Flex Parsers Reference

Verticasupports several parsers to load different types of data into flex tables:

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FJSONPARSER
- FREGEXPARSER

Unlike with columnar tables, you must specify which parser to use when loading flex tables. You can use each flex parser to load the parser's associated type of data into columnar tables.

All parsers store the data as a single Vmap in the `LONG VARBINARY __raw__` column. If a data row is too large to fit in the column, it is rejected. Vertica supports null values for loading data with NULL-specified columns.

For information about how you can use each type of flex parser, see Using Flex Table Parsers

FAVROPARSER

Parses data from an Avro file. The input file must use binary serialization encoding. Use this parser to load data into columnar, flex, and hybrid tables.

> **Note:** The parser `favroparser` does not support Avro files with separate schema files. The Avro file must have its related schema in the file you are loading.

Parameters

`flatten_arrays`	BOOLEAN	[Optional] Flattens all Avro arrays. Key names are concatenated with nested levels. **Default value:** `false` (Arrays are not flattened.)

`flatten_maps`	BOOLEAN	[Optional] Flattens all Avro maps. Key names are concatenated with nested levels **Default value:** `true`
`flatten_records`	BOOLEAN	[Optional] Flattens all Avro records. Key names are concatenated with nested levels. **Default value:** `true`
`reject_on_materialized_type_error`	BOOLEAN	[Optional] Indicates whether to reject any row value for a materialized column that the parser cannot coerce into a compatible data type. See Using Flex Table Parsers. **Default value:** `false`

Examples

This example shows how to create and load a flex table with Avro data using `favroparser`. After loading the data, you can query virtual columns.

1. Create a flex table for Avro data, `avro_basic`:

```
=> CREATE FLEX TABLE avro_basic();
CREATE TABLE
```

2. Use the `favroparser` to load the data from an Avro file (`weather.avro`).

```
=> COPY avro_basic FROM '/home/dbadmin/data/flexcsv/weather.avro' PARSER favroparser();
Rows Loaded
-------------
5
(1 row)
```

3. Query virtual columns from the `avro` flex table:

```
=> SELECT station, temp, time FROM avro_basic;
station | temp |     time
---------+------+---------------
mohali  | 0    | -619524000000
lucknow | 22   | -619506000000
norwich | -11  | -619484400000
ams     | 111  | -655531200000
```

```
baddi    | 78    | -655509600000
(5 rows)
```

For more information, see Loading Avro Data.

See Also

- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPARSER
- FDELIMITEDPAIRPARSER
- FJSONPARSER
- FREGEXPARSER

FCEFPARSER

Parses HPE ArcSight Common Event Format (CEF) log files. The `fcefparser` loads values directly into any table column with a column name that matches a source data key. The parser stores the data loaded into a flex table in a single VMap.

Parameters

`delimiter`	`CHAR`	[Optional] Specifies a single-character delimiter. **Default value:** `' '`
`record_terminator`	`CHAR`	[Optional] Specifies a single-character record terminator. **Default value:** `newline`
`trim`	`BOOLEAN`	[Optional] Trims white space from header names and key values. **Default value:** `true`
`reject_on_unescaped_delimiter`	`BOOLEAN`	[Optional] Determines whether to reject rows containing unescaped delimiters. The CEF standard does not permit them.

		Default value: `false`

Examples

The following example illustrates creating a sample flex table for CEF data, with two real columns, `eventId` and `priority`.

1. Create a flex table `cefdata`:

```
=> create flex table cefdata();
CREATE TABLE
```

2. Load some basic CEF data, using the flex parser `fcefparser`:

```
=> copy cefdata from stdin parser fcefparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> CEF:0|ArcSight|ArcSight|2.4.1|machine:20|New alert|High|
>> \.
```

3. Use the `maptostring()` function to view the contents of your `cefdata` flex table:

```
=> select maptostring(__raw__) from cefdata;
                  maptostring

-------------------------------------------------------------
 {
   "deviceproduct" : "ArcSight",
   "devicevendor" : "ArcSight",
   "deviceversion" : "2.4.1",
   "name" : "New alert",
   "severity" : "High",
   "signatureid" : "machine:20",
   "version" : "0"
}


(1 row)
```

4. Select some virtual columns from the `cefdata` flex table:

```
= select deviceproduct, severity, deviceversion from cefdata;
 deviceproduct | severity | deviceversion
---------------+----------+---------------
 ArcSight      | High     | 2.4.1
(1 row)
```

For more information, see Loading Common Event Format (CEF) Data

See Also

- FAVROPARSER
- FCSVPARSER
- FDELIMITEDPARSER
- FDELIMITEDPAIRPARSER
- FJSONPARSER
- FREGEXPARSER

FCSVPARSER

Parses CSV format (comma-separated values) data. Use this parser to load CSV data into columnar, flex, and hybrid tables. All data must be encoded in Unicode UTF-8 format. The parser `fcsvparser` supports the RFC 4180 de facto standard for CSV data, and other options, to accommodate variations in CSV file format definitions.

> The `fsvparser` does not support multibyte data. For more information about data formats, see Checking Data Format Before or After Loading.

Parameters

`type= {'traditional' | 'rfc4180'}`	CHAR	[Optional] Specifies the default parameter values for the parser. You do not have to use the type parameter when loading data that conforms to the RFC 4180 standard (such as MS Excel files). See Loading CSV Data for the `RFC4180` default parameters, and other options you can specify for non-default CSV files. **Default value:** `RFC4180`
`delimiter`	CHAR	[Optional] Indicates the single-character value used to separate fields in the CSV data. **Default value:** ,

escape	CHAR	[Optional] Specifies a single-character value. Use an `escape` character to interpret the next character in the data literally. **Default value:** "
enclosed_by	CHAR	[Optional] Specifies a single-character value. Use and `enclosed_by` value to include a value that is identical to the delimiter, but should be interpreted literally. For example, if the data delimiter is a comma (`,`), and you want to use a comma within the data ("`my name is jane, and his is jim`"). **Default value:** "
record_terminator	CHAR	[Optional] Indicates the single-character value used to specify the end of a record. **Default value:** `\n` or `\r\n`
header	BOOLEAN	[Optional] Specifies that a header column exists. If you specify `true`, but the data has no header, the parser uses `ucoln`, where *n* is the column offset number, starting with `0` for the first column. **Default value:** `true`
trim	BOOLEAN	[Optional] Indicates whether to trim white space from header names and key values. **Default value:** `true`
omit_empty_keys	BOOLEAN	[Optional] Indicates how the parser handles header keys without values. If `omit_empty_keys=true`, keys with an empty value in the `header` row are not loaded. **Default value:** `false`
reject_on_duplicate	BOOLEAN	[Optional] Specifies whether to halt the load process if the load file includes duplicate key names with different case (such as `BYPASS` and `BYpass`).

		Default value: `false`
`reject_on_empty_key`	BOOLEAN	[Optional] Specifies whether to reject any row containing a key without a value. **Default value:** `false`
`reject_on_materialized_type_error`	BOOLEAN	[Optional] Indicates whether to reject any materialized column value that the parser cannot coerce into a compatible data type. See Loading CSV Data. **Default value:** `false`

Examples

This example shows how you can use `fcsvparser` to load a flex table, build a view, and then query that view.

1. Create a flex table for csv data:

```
=> CREATE FLEX TABLE rfc();
CREATE TABLE
```

2. Use `fcsvparser` to load the data from STDIN. Specify that no header exists, and enter some data as shown:

```
=> COPY rfc FROM stdin PARSER fcsvparser(header='false');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 10,10,20
>> 10,"10",30
>> 10,"20""5",90
>> \.
```

3. Run the `compute_flextable_keys_and_build_view` function, and query the `rfc_view`. Notice that the default `enclosed_by` character permits an escape character (`"`) within a field (`"20""5"`). Thus, the resulting value was parsed correctly. Since no header existed in the input data, the function added `ucol`*n* for each column:

```
=> SELECT compute_flextable_keys_and_build_view('rfc');
                           compute_flextable_keys_and_build_view
--------------------------------------------------------------------------------------------
 Please see public.rfc_keys for updated keys
```

```
The view public.rfc_view is ready for querying
(1 row)

=> SELECT * FROM rfc_view;
 ucol0 | ucol1 | ucol2
-------+-------+-------
 10    | 10    | 20
 10    | 10    | 30
 10    | 20"5  | 90
(3 rows)
```

For more information and examples of using other parameters of this parser, see Loading CSV Data.

See Also

- FAVROPARSER
- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FDELIMITEDPAIRPARSER
- FJSONPARSER
- FREGEXPARSER

FDELIMITEDPAIRPARSER

Parses delimited data files. This parser provides a subset of the functionality in the parser `fdelimitedparser`. Use the `fdelimitedpairparser` when the data you are loading specifies pairs of column names with data in each row.

Parameters

`delimiter`	`CHAR`	[Optional] Specifies a single-character delimiter. **Default value:** `'  '`
`record_terminator`	`CHAR`	[Optional] Specifies a single-character record terminator.

		DDefault value: `newline`
trim	BOOLEAN	[Optional] Trims white space from header names and key values. **Default value:** `true`

Examples

The following example illustrates creating a sample flex table for simple delimited data, with two real columns, `eventId` and `priority`.

1. Create a table:

```
=> create flex table CEFData(eventId int default(eventId::int), priority int default
(priority::int) );
CREATE TABLE
```

2. Load a sample delimited HPE ArcSight log file into the `CEFData` table, using the `fcefparser`:

```
=> copy CEFData from '/home/release/kmm/flextables/sampleArcSight.txt' parser
fdelimitedpairparser();
Rows Loaded | 200
```

3. After loading the sample data file, use `maptostring()` to display the virtual columns in the `__raw__` column of `CEFData`:

```
=> select maptostring(__raw__) from CEFData limit 1;


maptostring
-----------------------------------------------------------
   "agentassetid" : "4-WwHuD0BABCCQDVAeX21vg==",
   "agentzone" : "3083",
   "agt" : "265723237",
   "ahost" : "svsvm0176",
   "aid" : "3tGoHuD0BABCCMDVAeX21vg==",
   "art" : "1099267576901",
   "assetcriticality" : "0",
   "at" : "snort_db",
   "atz" : "America/Los_Angeles",
   "av" : "5.3.0.19524.0",
   "cat" : "attempted-recon",
   "categorybehavior" : "/Communicate/Query",
   "categorydevicegroup" : "/IDS/Network",
   "categoryobject" : "/Host",
   "categoryoutcome" : "/Attempt",
   "categorysignificance" : "/Recon",
```

```
"categorytechnique" : "/Scan",
"categorytupledescription" : "An IDS observed a scan of a host.",
"cnt" : "1",
"cs2" : "3",
"destinationgeocountrycode" : "US",
"destinationgeolocationinfo" : "Richardson",
"destinationgeopostalcode" : "75082",
"destinationgeoregioncode" : "TX",
"destinationzone" : "3133",
"device product" : "Snort",
"device vendor" : "Snort",
"device version" : "1.8",
"deviceseverity" : "2",
"dhost" : "198.198.121.200",
"dlat" : "329913940429",
"dlong" : "-966644973754",
"dst" : "3334896072",
"dtz" : "America/Los_Angeles",
"dvchost" : "unknown:eth1",
"end" : "1364676323451",
"eventid" : "1219383333",
"fdevice product" : "Snort",
"fdevice vendor" : "Snort",
"fdevice version" : "1.8",
"fdtz" : "America/Los_Angeles",
"fdvchost" : "unknown:eth1",
"lblstring2label" : "sig_rev",
"locality" : "0",
"modelconfidence" : "0",
"mrt" : "1364675789222",
"name" : "ICMP PING NMAP",
"oagentassetid" : "4-WwHuD0BABCCQDVAeX21vg==",
"oagentzone" : "3083",
"oagt" : "265723237",
"oahost" : "svsvm0176",
"oaid" : "3tGoHuD0BABCCMDVAeX21vg==",
"oat" : "snort_db",
"oatz" : "America/Los_Angeles",
"oav" : "5.3.0.19524.0",
"originator" : "0",
"priority" : "8",
"proto" : "ICMP",
"relevance" : "10",
"rt" : "1099267573000",
"severity" : "8",
"shost" : "198.198.104.10",
"signature id" : "[1:469]",
"slat" : "329913940429",
"slong" : "-966644973754",
"sourcegeocountrycode" : "US",
"sourcegeolocationinfo" : "Richardson",
"sourcegeopostalcode" : "75082",
"sourcegeoregioncode" : "TX",
"sourcezone" : "3133",
"src" : "3334891530",
"start" : "1364676323451",
"type" : "0"
```

```
}
(1 row)
```

4. Select the `eventID` and `priority` real columns, along with two virtual columns, `atz` and `destinationgeoregioncode`:

```
=>  select eventID, priority, atz, destinationgeoregioncode from CEFData limit 10;
  eventID    | priority |         atz         | destinationgeoregioncode
------------+----------+---------------------+--------------------------
 1218325417 |        5 | America/Los_Angeles |
 1219383333 |        8 | America/Los_Angeles | TX
 1219533691 |        9 | America/Los_Angeles | TX
 1220034458 |        5 | America/Los_Angeles | TX
 1220034578 |        9 | America/Los_Angeles |
 1220067119 |        5 | America/Los_Angeles | TX
 1220106960 |        5 | America/Los_Angeles | TX
 1220142122 |        5 | America/Los_Angeles | TX
 1220312009 |        5 | America/Los_Angeles | TX
 1220321355 |        5 | America/Los_Angeles | CA
(10 rows)
```

See Also

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPARSER
- FJSONPARSER
- FREGEXPARSER

FDELIMITEDPARSER

Parses data using a delimiter character to separate values. The `fdelimitedparser` loads delimited data, storing it in a single-value VMap. You can use this parser to load data into columnar and flex tables.

Parameters

`delimiter`	CHAR	[Optional] Indicates a single-character delimiter.

		Default value: |
`record_terminator`	CHAR	[Optional] Indicates a single-character record terminator. **Default value:** `\n`
`trim`	BOOLEAN	[Optional] Determines whether to trim white space from header names and key values. **Default value:** `true`
`header`	BOOLEAN	[Optional] Specifies that a header column exists. The parser uses `col###` for the column names if you use this parameter but no header exists. **Default value:** `true`
`omit_empty_keys`	BOOLEAN	[Optional] Indicates how the parser handles header keys without values. If `omit_empty_keys=true`, keys with an empty value in the `header`row are not loaded. **Default value:** `false`
`reject_on_duplicate`	BOOLEAN	[Optional] Specifies whether to halt the load process if the file being loaded includes duplicate key names with different case (such as `BYPASS` and `BYpass`). **Default value:** `false`
`reject_on_empty_key`	BOOLEAN	[Optional] Specifies whether to reject any row containing a key without a value. **Default value:** `false`
`reject_on_materialized_type_error`	BOOLEAN	[Optional] Indicates whether to reject any row value for a materialized column that the parser cannot coerce into a compatible data type. See Using Flex Table Parsers. **Default value:** `false`

`treat_empty_val_as_null`	BOOLEAN	[Optional] Specifies that empty fields become `NULLs`, rather than empty strings (`''`). **Default value:** `true`

Examples

1. Create a flex table for delimited data:

```
=> create flexible table my_test();
CREATE TABLE
```

2. Use the `fdelimitedparser` to load the data from a `.csv` file. Specify a comma (,) column delimiter:

```
=> copy my_test from '/test/vertica/DATA/a.csv' parser fdelimitedparser (delimiter=',');
 Rows Loaded
-------------
           3
(1 row)
```

1. Select some virtual columns from the `cefdata` flex table:

```
= select deviceproduct, severity, deviceversion from cefdata;
 deviceproduct | severity | deviceversion
---------------+----------+--------------
 ArcSight      | High     | 2.4.1
(1 row)
```

For more information, see Loading Common Event Format (CEF) Data

See Also

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPAIRPARSER
- FJSONPARSER
- FREGEXPARSER

FJSONPARSER

Parses and loads a JSON file. This file can contain either repeated JSON data objects (including nested maps), or an outer list of JSON elements. For a flex table, the parser stores the JSON data in a single-value VMap. For a hybrid or columnar table, the parser loads data directly in any table column with a column name that matches a key in the JSON source data.

Parameters

`flatten_maps`	BOOLEAN	[Optional] Flattens sub-maps within the JSON data, separating map levels with a period (`.`). **Default value:** `true`
`flatten_arrays`	BOOLEAN	[Optional] Converts lists to sub-maps with integer keys. Lists are not flattened by default. **Default value:** `false`
`reject_on_duplicate`	BOOLEAN	[Optional] Halts the load process if the file being loaded includes duplicate key names, with different case. **Default value:** `false`
`reject_on_empty_key`	BOOLEAN	[Optional] Rejects any row containing a key without a value (`reject_on_empty_key=true`). **Default value:** `false`
`omit_empty_keys`	BOOLEAN	[Optional] Omits any key from the load data that does not have a value (`omit_empty_keys=true`). **Default value:** `false`
`reject_on_materialized_type_error`	BOOLEAN	[Optional] Rejects a data row that contains a materialized column value that cannot be coerced into a compatible data type (`reject_on_materialized_type_error=true`. **Default value:** `false`
`start_point`	CHAR	[Optional] Specifies the name of a key in the

		JSON load data at which to begin parsing. The parser ignores all data before the `start_point` value.The parser processes data after the first instance, and up to the second, ignoring any remaining data. **Default value:** none
`start_point_occurrence`	INTEGER	[Optional] Indicates the *n*th occurrence of the value you specify with `start_point`. Use in conjunction with `start_point` when load data has multiple start values and you know the occurrence at which to begin parsing. **Default value:** `1`
`suppress_nonalphanumeric_key_chars`	BOOLEAN	[Optional] Suppresses non-alphanumeric characters in JSON key values. The parser replaces these characters with an underscore (_) when this parameter is `true`. **Default value:** `false`
`key_separator`	CHAR	[Optional] Specifies a non-default character for the parser to use when concatenating key names. **Default value:** `'.'`

Examples

Load JSON Data Without Optional Parameters

1. Create a flex table, `super`, with two columns, `age` and `name`:

```
=> create table super(age int, name varchar);
CREATE TABLE
```

2. Enter values using the `fjsonparser()`, and query the results:

```
=> copy super from stdin parser fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"age": 5, "name": "Tim"}
>>  {"age": 3}
>>  {"name": "Fred"}
>>  {"name": "Bob", "age": 10}
>> \.
```

```
=> select * from super;
 age | name
-----+------
     | Fred
  10 | Bob
   5 | Tim
   3 |
(4 rows)
```

For other examples, see Loading JSON Data.

See Also

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPARSER
- FDELIMITEDPAIRPARSER
- FREGEXPARSER

FREGEXPARSER

Parses a regular expression, matching columns to the contents of the named regular expression groups.

Parameters

pattern	VARCHAR	Specifies the regular expression of data to match. **Default value:** Empty string ("")
use_jit	BOOLEAN	[Optional] Indicates whether to use just-in-time compiling when parsing the regular expression. **Default value:** `false`
record_terminator	VARCHAR	[Optional] Specifies the character used to separate input records. **Default value:** `\n`
logline_column	VARCHAR	[Optional] Captures the destination column containing the

		full string that the regular expression matched. **Default value:** Empty string ("")

Example

These examples use the following regular expression, which searches for information that includes the `timestamp`, `date`, `thread_name`, and `thread_id` strings. For illustrative purposes, the regular expression components are shown here on separate lines. Remove any new line characters before using this example in your own tests.

```
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?
(?:\[(?<component>\w+)\] \<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @\[?(?<enode>\w+)\]?: )?
(?<text>.*)'
```

1. Create a flex table (`vlog`) to contain the results of a Vertica log file. For this example, we made a copy of a log file in the directory `/home/dbadmin/data/vertica.log`:

```
=> create flex table vlog();
CREATE TABLE
```

2. Use the `fregexparser` with the sample regular expression to load data from the log file (removing line characters first):

```
=>  copy vlog from '/home/dbadmin/data/flex/vertica.log' PARSER FRegexParser(pattern=
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+) (?<thread_name>[A-Za-z ]+):
(?<thread_id>0x[0-9a-f]+)-?(?<transaction_id>[0-9a-f]+)?
(?:\[(?<component>\w+)\] \<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @\[?(?<enode>\w+)\]?: )?
(?<text>.*)');
 Rows Loaded
-------------
        8434
(1 row)
```

3. After successfully loading data, query columns directly from the flex table:

```
=> select text, time, thread_name from vlog limit 4;
                         text                        |          time           | thread_name
-------------------------------------------------------+-------------------------+------------
--
 Starting up Vertica Analytic Database v7.1.0-20140331 | 2014-04-02 04:02:02.613 | Main
 Checking for missed alter partition events           | 2014-04-02 04:02:51.017 |
ManageEpochs
 Found no missed alter partition events               | 2014-04-02 04:02:51.017 |
ManageEpochs
 Checking for missed replace node events              | 2014-04-02 04:02:51.017 |
```

```
ManageEpochs
(4 rows)
```

4. Use the MAPTOSTRING() function with the table's `__raw__` column. The four rows (`limt 4`) that the query returns are regular expression results of the `vertica.log` file, parsed with `fregexparser`:

```
=> select maptostring(__raw__) from vlog limit 4;
                        maptostring

-------------------------------------------------------------------------------
 {
   "component" : "Init",
   "level" : "INFO",
   "text" : "Log /scratch_b/qa/STDB/v_stdb_node0001_catalog/vertica.log opened; #2",
   "thread_id" : "0x16321430",
   "thread_name" : "Main",
   "time" : "2014-04-02 04:02:02.613"
}

 {
   "component" : "Init",
   "level" : "INFO",
   "text" : "Project Codename: Dragline",
   "thread_id" : "0x16321430",
   "thread_name" : "Main",
   "time" : "2014-04-02 04:02:02.613"
}

 {
   "component" : "Init",
   "level" : "INFO",
   "text" : "Processing command line: /opt/vertica/bin/vertica -D /scratch_b/qa/STDB/v_stdb_
node0001_catalog
     -C STDB -n v_stdb_node0001 -h 10.20.90.192 -p 6059 -P 6061",
   "thread_id" : "0x16321430",
   "thread_name" : "Main",
   "time" : "2014-04-02 04:02:02.613"
}

 {
   "component" : "Init",
   "level" : "INFO",
   "text" : "64-bit Optimized Build",
   "thread_id" : "0x16321430",
   "thread_name" : "Main",
   "time" : "2014-04-02 04:02:02.613"
}

(4 rows)
```

See Also

- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FJSONPARSER

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on Using Flex Tables (Vertica Analytic Database 7.2.x)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to vertica-docfeedback@hpe.com.

We appreciate your feedback!