Flex Tables Guide

HP Vertica Analytic Database

Software Version: 7.0.x

Document Release Date: 2/24/2014

Legal Notices

Warranty

The only warranties for HP 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. HP 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 HP 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.

Contents

Getting Started

Welcome to creating, loading, and querying flex tables in your database. Using Flex tables and their associated helper and map functions, along with their general integration into HP Vertica, you can create and manage flex tables.

Flex tables:

- Don't require schema definitions
- Don't need column definitions
- Have full Unicode support
- Support full SQL queries

In addition to promoting data directly from exploration to operation, the following features make HP Vertica flex tables a key part of your data management toolkit:

- Ability to put multiple formats into one flex table — lets you handle changing structure over time
- Full support of delimited and JSON data
- Extensive SQL and built-in analytics for the data you load
- Usability functions — explore unstructured data before materializing, and then materialize using built-in functions

Once you create a flex table, you can quickly load data, including social media output in JSON, log files, delimited files, and other information not typically considered ready for your HP Vertica database. Previously, working with such data required significant schema design and preparation. Now, you can load and query flex tables in a few steps.

This Flex Basics section guides you through several consecutive scenarios to illustrate creating and loading data into a flex table. Then, continues with extracting and querying the loaded data.

The rest of this guide presents the details beyond the basics using simple examples. Ready to start?

Create a Simple JSON File

Here is the JSON data we'll use in the rest of the Flex Basics:

```
{"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. Swipe the JSON content into your favorite editor.
2. Save the file in any convenient location for loading.

Create a Flex Table and Load Data

1. Create a flex table table `mountains`:

```
Vmart=> create flex table mountains();
CREATE TABLE
```

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

```
Vmart=> copy mountains from '/home/dbadmin/Downloads/mountains.json' parser fjsonpars
er();
 Rows Loaded
-------------
           5
(1 row)
```

3. Query values from the sample file:

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

That's it! You're ready to learn more about using flex table data in your database.

Query the Flex Table

1. Query your flex table to see the data you loaded as it is stored in the `__raw__` column:

```
Vmart=> select * from mountains;
 __identity__ |                                          __raw__

--------------+------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
           5 | \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\00
0\000\000\037\000\000\000hike_safetynametype
           1 | \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
           4 | \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\0001400022.8Kilimanjaromountain\004\000\000\000\024
\000\000\000\032\000\000\000%\000\000\000)\000\000\000heighthike_safetynametype
           2 | \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
           3 | \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\000\000%\000\000\000)\000\000\000heighthike_safetynametype
(5 rows)
```

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

```
Vmart=> 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. Use `compute_flextable_keys` to populate the `mountain_keys` table, which HP Vertica created automatically when you created your flex table. Query the keys table `mountains_keys`):

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

Vmart=> 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 `build_flextable_view` to populate a view generated from the `mountains_keys` table. Query the view `mountains_view`):

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

Vmart=> 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)
```

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

```
Vmart=> 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)
```

3. 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 notice the `data_type_guess` column when you queried the `mountains_keys` table after invoking that function?

4. Using the `data_type` information from `mountains_view`, change the `data_type_guess`for the `hike_safety` virtual column, `COMMIT` the change, and build the view again with `build_flextable_view()`:

```
Vmart=> update mountains_keys set data_type_guess = 'float' where key_name = 'hike_sa
fety';
 OUTPUT
--------
      1
(1 row)

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

5. Query the `view_columns` system table again to see the data type change for the `hike_safety` key:

```
Vmart=> select column_name, data_type from view_columns where table_name = 'mountain
s_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 unstructured data you load and query regularly is important enough to require full HP Vertica performance and support, create a *hybrid* flex table — one with as many

column definitions as you need. You can specify default values for the columns. Because you are creating a flex table with column definitions, the flex table automatically has a __raw__ column to store any unstructured data you load:

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

```
Vmart=> create flex table mountains_hybrid(name varchar(41) default name::varchar(4
1), hike_safety float default hike_safety::float);
CREATE TABLE
Vmart=> copy mountains_hybrid from '/home/dbadmin/Downloads/mountains.json' parser fj
sonparser();
 Rows Loaded
-------------
           5
(1 row)
```

2. Use the compute_flextable_keys_and_build_view helper function to populate the keys table and build the view for mountains_hybrid:

```
Vmart=> 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 keys table for mountains_hybrid. Notice the data_type_guesses column values again. These reflect the column definitions you declared:

```
Vmart=> 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)
```

Promote Virtual Columns in a Hybrid Flex Table

You can promote any virtual columns (keys) in a flex (or hybrid) table to real columns — no need to create a separate columnar table.

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

```
Vmart=> 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 = 'p
ublic' and table_name = 'mountains_hybrid';

(1 row)
```

2. Since you specified three (`3`) columns to materialize, but the table was created with two real columns (`name` and `hike_safety`), the function promotes only one other column, `type`. The example has expanded display so you can see columns listed 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:

```
Vmart=> \x
Expanded display is on.
Vmart=> select * from materialize_flextable_columns_results where table_name = 'mount
ains_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. Query 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 `NOT NULL` constraint (by default):

```
Vmart=> \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 |
```

That's it for getting started Flex table basics, hybrid flex tables, and an introduction to using the helper functions.

New Terms for Flex Tables

This handbook uses the following terms when describing and working with flex tables:

Term	Meaning
Map data	The data in the `__raw__` column after you load data. The __raw__ column gets populated using an internal Map type. The Map type consists of a varbinary blob containing raw data that is processed for structure or interpretation during loading. After processing, the map data includes a dictionary-style lookup of `key::value` pairs. Map keys are the virtual columns (described next) that exist in the map data.
Virtual columns	Undeclared (unmaterialized) columns, also called *map keys*, that exist within the `__raw__` column of a flex table after loading data. Virtual columns act as lookups into the Map data. You can query the values that exist within virtual columns. However, you cannot see virtual columns if you list a flex table with \d *`flextable`*. Restrictions also exist on virtual columns, as described in Altering Flex Tables.
Real (materialized) columns	Columns copied out of flex table virtual columns into real, fully-featured columns.
Promote	Materialize a virtual column to a real, fully-featured column.
Map keys	Within the Map data, Map keys are the names of the virtual columns. For example, by loading data from a tweet file (JSON) into a FlexTable `darkdata`, one of the map keys is `"user.lang"`, which you can query directly: `select "user.lang" from darkdata;` If you do not know what keys exist in the map data, the Working with Flex Table Map Functions and Flex Table Data Functions can compute them.

Understanding Flex Tables

Creating flex tables is similar to creating other tables, except column definitions are optional. When you create flex tables, with or without column definitions, HP Vertica implicitly adds a special column to your table, called `__raw__`. This is the column that stores loaded data. The __raw__ column type is LONG VARBINARY, and its default maximum width is 130000 bytes (with an absolute maximum of 32000000 bytes). You can change the width default with the `FlexTablesRawSize` configuration parameter.

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

Is There Structure in Flex Tables?

The term *unstructured data* (sometimes called *semi-structured* or *Dark Data*) does not indicate that the data you'll load into flex tables is entirely without structure. However, you may not know the data's composition, or the inconsistencies of its design, and data may not be relational. Our friends at Wikipedia define unstructured data as follows:

Unstructured Data (or unstructured information) refers to information that either does not have a pre-defined data model or is not organized in a pre-defined manner. Unstructured information is typically text-heavy, but may contain data such as dates, numbers, and facts as well. and...
Structure, while not formally defined, can still be implied. *Data with some form of structure may still be characterized as unstructured if its structure is not helpful for the processing task at hand.*

Your data may have some structure (like JSON and delimited data), be semi-structured or stringently-structured, but in ways that you either don't know about, or don't expect. We're using the term *flexible data* to encompass all of this sort of data, and to indicate that you can load that data directly into a flex table, and query its contents with your favorite SQL SELECT and other statements.

To summarize, you can load data first, without knowing its structure, and then query its content after a few simple transformations. If you already know the data's structure, such as some tweet map keys, like user.lang, user.screen_name, and user.url, you can query these values as soon as you've loaded 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 HP Vertica functionality writes the table and its contents to disk (ROS) to maintain K-safety in your cluster, and to support standard recovery processes should your site have a node failure. Flex tables are included in full backups (or in object-level backups if you choose).

What Happens When You Create Flex Tables?

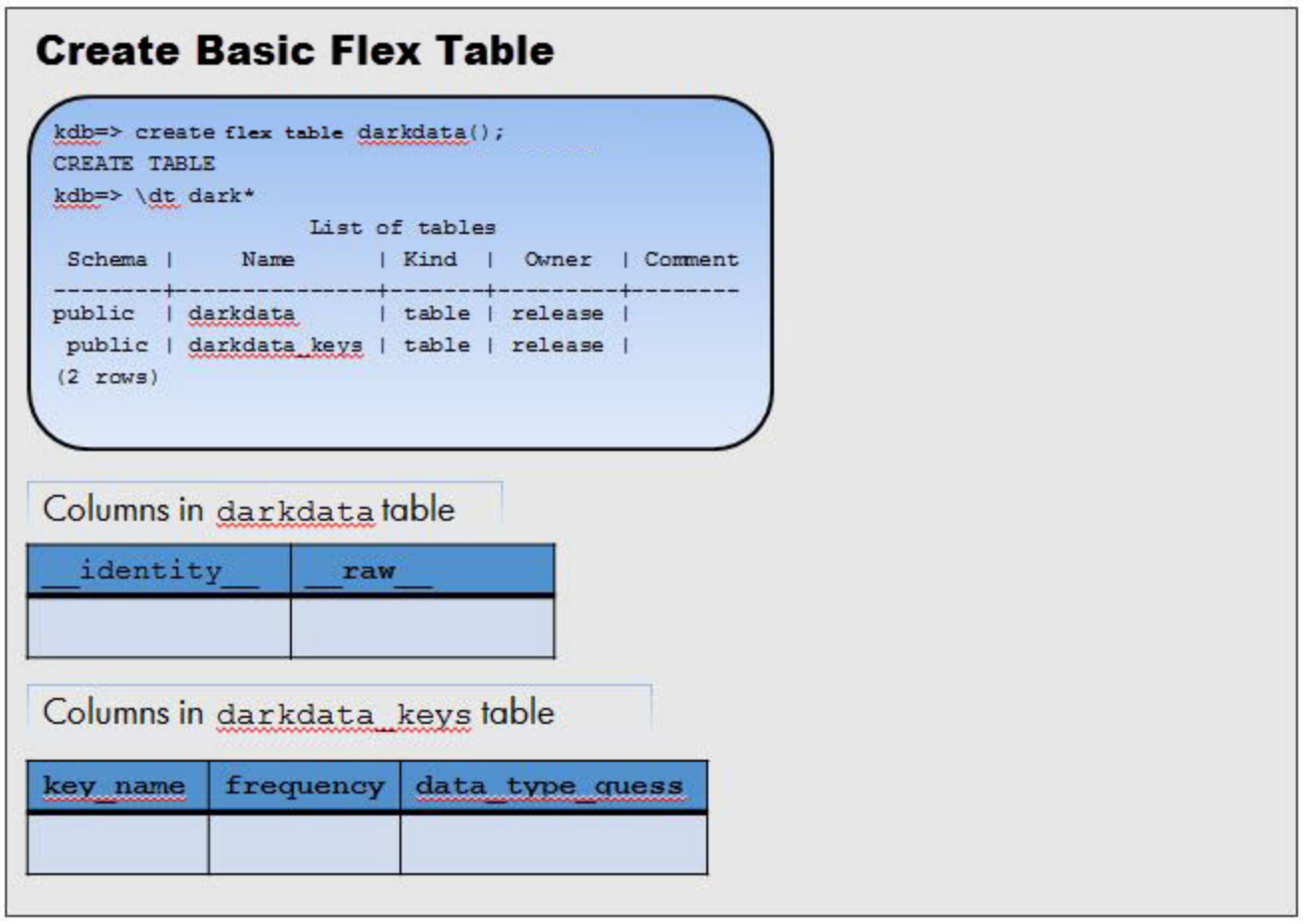
Whenever you execute a `CREATE FLEX TABLE` statement, HP Vertica creates three objects, as follows:

1. The flexible table (*flex_table*)
2. An associated keys table (*flex_table*_keys)
3. A default view for the main table (*flex_table*_view)

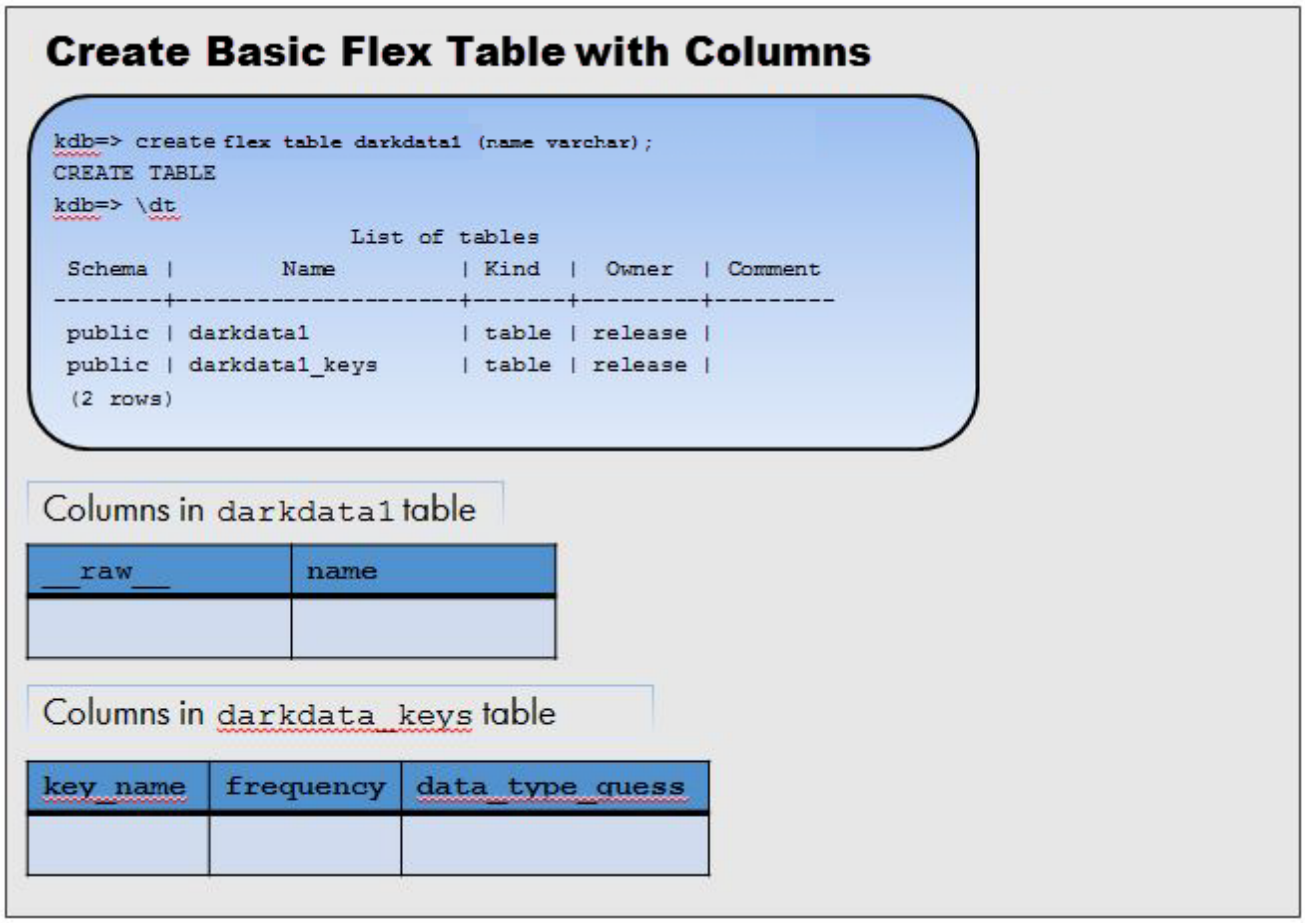
The _keys and _view objects are dependents of the parent *flex_table*. Dropping the flex table also removes its dependents, although you can drop the `_keys` or `_view` objects independently.

Without specifying any column definitions, creating a flex table (`darkdata` in the next example) creates two columns automatically, `__raw__` and `__identity__`. The `__raw__` column exists in every flex table to hold the data you load. The __identity__ column is auto-incrementing and used for segmentation and sort order when no other columns exist (other than __raw__).

Two tables are created automatically, the named flex table (such as darkdata) and its associated keys table, darkdata_keys, which has three columns, as shown in the `darkdata` table example:



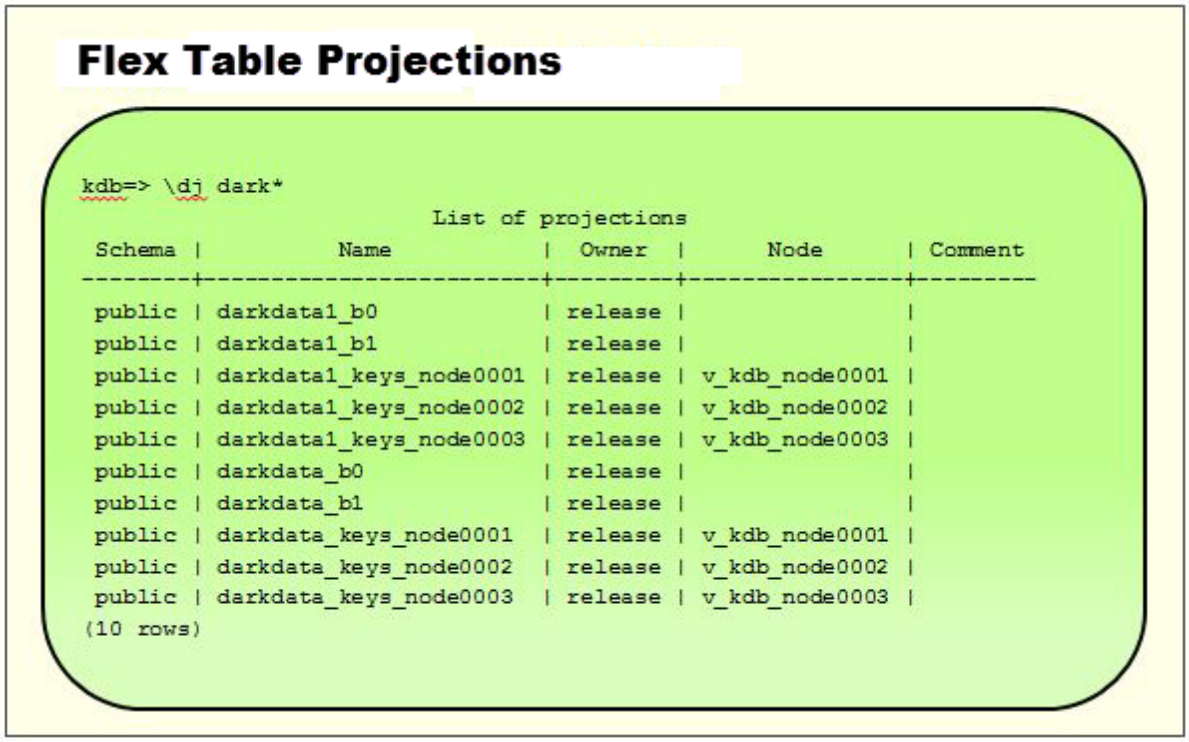
Creating a flex table with column definitions (darkdata1 in the next example) also creates a table with the __raw__ column, but not an __identity__ column, since the columns you specify can be used for segmentation and sort order. Two tables are also created automatically, as shown in the following example:



For more examples, see Creating Flex Tables.

Creating Superprojections Automatically

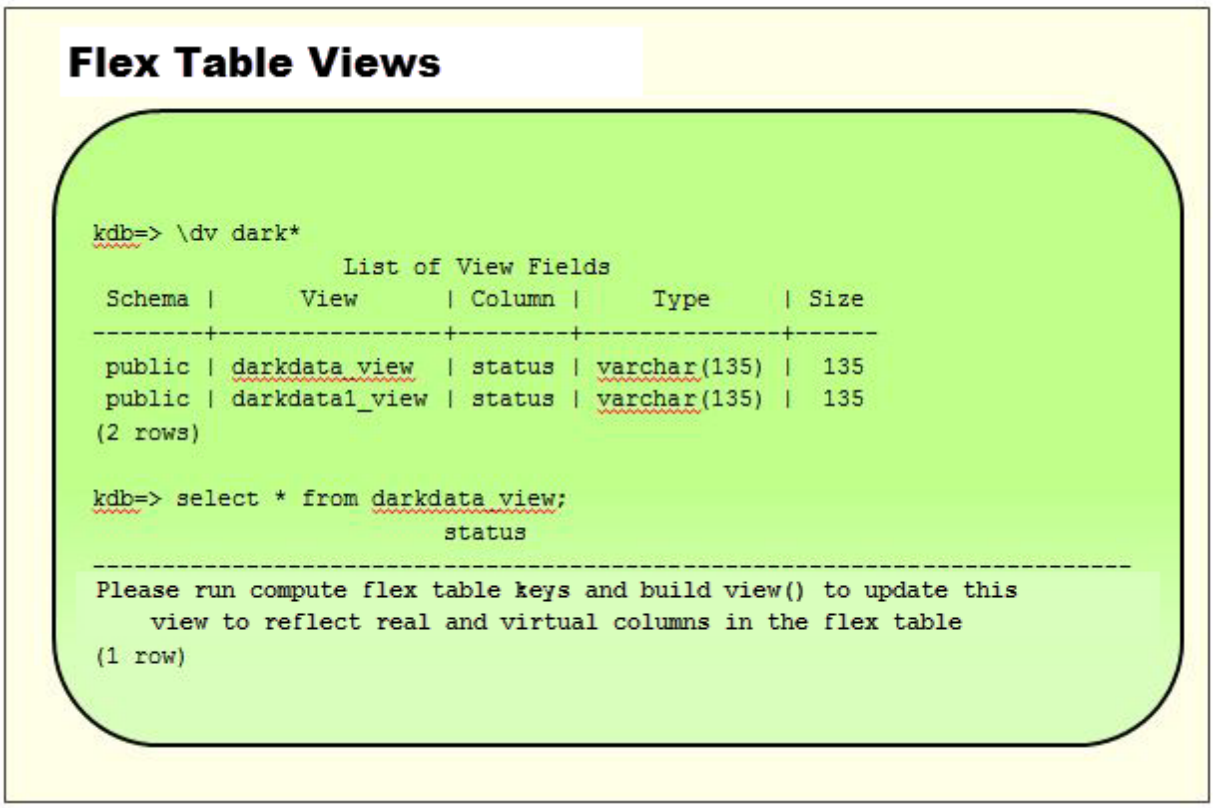
In addition to creating the two tables, HP Vertica also creates superprojections for both tables. This example shows the projections that were automatically created for the darkdata and darkdata1 tables:



Note: You cannot create pre-join projections from flex tables.

Default Flex Table View

Creating a flex table also creates a default view, using the table name with a `_view` suffix, as listed in the next example showing the list of views for darkdata and darkdata1. Querying the default view prompts you to use the COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW function to update the view so it includes all keys and values after you've loaded data:



For more information, see Updating Flex Table Views.

Library Map Functions

The flex table map functions give you access to map data from the virtual columns in a flex table. For more information, see Working with Flex Table Map Functions.

Flex Table Data Functions

The flex table data load and query facility includes a set of *helper* meta-functions. The functions compute keys and create views to aid in querying flex table data and SELECT * support.

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.

Function	Description
COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW	Performs both of the functions described above 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.

For more information on using these meta-functions, see Flex Table Data Functions.

You can also customize two configuration parameters for flex table usage. See Setting Flex Table Parameters.

Using Clients with Flex Tables

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

1. `INSERT` statements are not supported on any client (or from vsql), and the drivers' batch insert APIs do not support loading into flex tables. To load data from a client, use `COPY LOCAL` with either the `fjsonparser` or the `fdelimitedparser` options.
2. The driver metadata APIs only report materialized columns from a flex table, not virtual columns that exist within the `__raw__` column. For example, when asked for the columns of a flex table with a single materialized column (`name`), the drivers return the materialized column `name` and `__raw__`.

License Considerations

Flex tables are licensed under a separate Flex Zone license. You can purchase a Flex Zone license as a standalone product, or in addition to your HP VerticaEnterprise Edition (EE) license. For more information, see the *Managing Licenses* section information in the Administrator's Guide.

Creating Flex Tables

You can create a flex table (or external flex tables) without column definitions or other parameters, or use your favorite create parameters as usual.

Note: You cannot create temporary local or global flex tables, or temporary local or global external flex tables.

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:

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

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

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

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

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

Selecting from this table lists the default `__raw__` column, followed by *n* columns you defined. No `__identity__` column exists, because other columns you specify can be used for segmentation and sorting:

```
kdb=> select * from darkdata1;
 __raw__ | name
---------+------
(0 rows)
```

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

Creating Temporary Flex Tables

You can create temporary global and local flex tables with the following caveats:

- 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. 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 local temp flex tables to function correctly, you must also specify the `ON COMMIT PRESERVE ROWS` clause. The `ON COMMIT` clause is required for the flex table helper functions, which rely on commits. Create a local temp table as follows:

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

After creating a local temporary flex table in this way, you can then load data into the table, create table keys, and a flex table view, as follows:

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

dbt=> 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 warning messages when you create the flex table:

```
dbt=> 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 pol
icy,
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 t
able to create
```

```
a view, but involves a DDL commit which will delete the table's rows
CREATE TABLE
```

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:

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

Promoting Flex Table Virtual Columns

After you create your flex table and load data, you'll compute keys from virtual columns. After those tasks, you'll probably want to promote some keys into real table columns. By promoting virtual columns, you can query real columns, rather than querying the raw data.

Materializing one or more virtual columns — promoting those keys from within the `__raw__` data to real columns — is recommended to get the best flex table performance for all important keys. You don't need to create new columnar tables from your flex table. Materializing flex table columns results in a hybrid table. Hybrid tables maintain the convenience of a flex table for loading unstructured data, and improve query performance for any materialized columns.

If you have only a few columns to materialize, try altering your flex table progressively, adding columns whenever it's necessary. You can use the `ALTER TABLE...ADD COLUMN` statement to do that, 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 HP Vertica table from a flex table. You cannot use one flex table to create another. Typically, you create a columnar table from a flex table after loading data. Then, specify the virtual column data you want in a regular table. You must cast virtual columns to regular data types.

For example, to create a columnar table from the flex table `darkdata`, which has two virtual columns (`user.lang` and `user.name`), enter the following command:

```
kdb=> create table darkdata_full as select "user.lang"::varchar, "user.name"::varchar fro
m darkdata;
CREATE TABLE
kdb=> select * from darkdata_full;
 user.lang |      user.name
-----------+---------------------
```

```
 en        | Avita Desai
 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:

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

After creating an external flex table, two regular tables exist, as with other flex tables, the named table, and its associated _keys table, which is not an external table:

```
kdb=> \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:

```
kdb=> 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:

   ```
   kdb=> 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:

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

Note: While external tables are fully supported for both flex and columnar tables, using external flex (or columnar) tables is less efficient than using their resident counterparts, whose data is stored in the HP Vertica database. Any data maintained externally must be loaded each time you query such data.

Partitioning Flex Tables

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

The next example queries `user.location`, which is a virtual column in the map data, and then tries to partition on that column:

```
kdb=> select "user.location" from darkdata;
 user.location
---------------
 chicago
```

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

Loading Flex Table Data

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

- `fdelimitedparser`
- `fjsonparser`
- `fcefparser`

All parsers store the data as a single-value map type in the `VARBINARY __raw__` column. The map type data is encoded into a single binary value for storage in the `__raw__` varbinary column. The encoding places the value strings in a contiguous block, followed by the key strings.

If a flex table data row would be too large to fit in the column, it is rejected. Null values are supported for loading data with `NULL`-specified columns.

Basic Flex Table Load and Query

Loading data into your flex table is similar to loading other data into a regular table, except that it requires the `parser` argument, followed by `fjsonparser`, `fdelimitedparser`, or `fcefparser`.

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

Loading Flex Table Data into Same-Name Columns

If you create a flex table with column definitions, and names that are identical to key names in the data being loaded, those columns will be populated with the key values automatically, in addition to being loaded into the __raw__ column map data.

For example, continuing with the JSON data:

1. Create a flex table, darkdata1, with a column definition of one of the key values in the data:

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

2. Load data into darkdata1:

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

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

```
kdb=> 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 mapLookup().

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

```
kdb=> 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)
```

Note: 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 small test data), 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. In the next example, the user.lang column declares another virtual column (user.name) as its default value:

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

If you load data without specific directives, COPY uses values from the flex table data, ignoring the default column definition. Why? Because loading flex table data evaluates same-name key values (user.lang for example) against declared table column names. Loading data into flex table uses mapLookup to find keys that match any real column names. If a match exists (the incoming data has a virtual column that matches a real column), COPY populates the column with key values. COPY returns either a key value or NULL for each row, so same-name columns always have values.

For example, after creating the darkdata1 flex table, described above, load data with COPY:

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

Querying the darkdata1 table after loading data shows that the user.lang values were extracted from the data being loaded (values for the user.lang virtual column, or NULL for rows without values). The table column default value for user.lang was ignored:

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

Using COPY to Specify Default Values

You can add an expression to the 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. In the following example, which uses an expression for the default column value, loading populates the defined `user.lang` column with data from the input data's `user.name` values:

```
kdb=> copy darkdata1(__raw__, "user.lang" as "user.name"::varchar)from '/test/vertica/fle
xtable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
kdb=> 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)
```

A rather different behavior exists if you specify default values when adding columns, as described in Altering Flex Tables. 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 Table Parsers

These parsers support loading flex tables:

- `fdelimitedparser`
- `fjsonparser`
- `fcefparser`

All parsers store the data as a single-value map type in the `LONG` VARBINARY __raw__ column. If a flex table data row would be too large to fit in the column, it is rejected. Null values are supported for loading data with NULL-specified columns.

Note: Specifying rejected data and exceptions files is not currently supported while loading flex tables.

Loading Delimited Data

The `fdelimitedparser` loads delimited data, storing it in a single-value map. You can use this parser to load data into columnar and flex tables.

The delimited parser does not handle CSV escaping.

Parameters

`delimiter`	*=char* Required parameter. Default value is |.
`record_terminator`	*=char* [Optional] Default terminator is `newline`.
`header`	*=bool* [Optional] Default `true`. Uses `col###` as the table column names if no header ROW exists.
`trim`	*=bool* [Optional] Default `true`. Trims whitespace from header names and field values.
`reject_on_duplicate`	*=bool* [Optional] Default `false`.Causes the load to halt if the file being loaded includes a header row with duplicate column names, with different case.

fdelimitedparser Example

For example:

1. Create a flex table for delimited data:

```
kdb=> create flexible_table my_test();
CREATE TABLE
```

2. Use the `fdelimitedparser` to load the data from a `.csv` file, specifying a comma (,) delimiter:

```
kdb=> copy my_test from '/test/vertica/DATA/a.csv' parser fdelimitedparser (delimiter
=',');
 Rows Loaded-------------
           3
(1 row)
```

Loading JSON Data

Loads a bare file of repeated JSON data objects, including nested maps, or a file with an outer list of JSON elements. The `fjsonparser` loads values directly into any table column with a column name that matches a source data key. The parser stores the data in a single-value map.

Using the parameters `flatten_maps` and `flatten_arrays` is recursive, and flattens all data.

Checking JSON Integrity

You can check the integrity of the JSON data you are loading by using a web tool such as JSONLint. Copying your JSON data into the tool returns information if anything is invalid, as in this example:

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

Parameters

`flatten_maps`	*=bool*
	[Optional]
	Default `true`. Flattens sub-maps within the JSON data, separating map **`levels`** with a period (`.`). For example if the input file contains a submap such as the following :
	`{ foo: { bar: 4 } }`
	Using `flatten_maps=true` produces the following map :
	`{ "foo.bar" -> "4" }`

`flatten_arrays`	*=bool*
	[Optional] Default `false`. Converts lists to sub-maps with integer keys. By default, to prevent key space explosion, does not flatten lists. For example if the input file contains the following array:
	`{ foo: [ 1 2 ] }`
	Using `flatten_arrays=false` (the default), results in the following array:
	`{ "foo": { "0" -> "1", "1" -> "2" } }`

`reject_on_duplicate`	*=bool* [Optional] Default `false`.Causes the load to halt if the file being loaded includes duplicate key names, with different case.

Loading HP ArcSight Data

Loads a file of HP ArcSight data objects. The `fcefparser` loads values directly into any table column with a column name that matches a source data key. The parser stores the data in a single-value map.

Parameters

`delimiter`	*=char* Required parameter. Default value is |.
`record_terminator`	*=char* [Optional] Default terminator is `newline`.
`trim`	*=bool* [Optional] Default `true`. Trims whitespace from header names and field values.

fcefparser Example

The following example illustrates creating a sample flex table for CEF data, with two real columns, `eventId` and `priority`.

1. Create a table:

```
dbs=> create flex table CEFData(eventId int default(eventId::int), priority int defau
lt(priority::int) );
CREATE TABLE
```

2. Load a sample HP ArcSight log file into the `CEFData` table, using the `fcefparser`:

```
dbs=> copy CEFData from '/home/release/kmm/flextables/sampleArcSight.txt' parser fcef
parser();
Rows Loaded | 200
```

3. After loading the sample data file, use `maptostring()` to display the virtual columns in the `__raw__` column of `CEFData`:

```
dbs=> 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`:

```
dbs=>  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)
```

Using Flex Parsers for Columnar Tables

While the `fjsonparser` and `fdelimited` parsers are available to load raw JSON and delimited data into flex tables, you can also the parsers to load data into columnar tables. Using the flex table parsers for column tables gives you the capability to mix data loads in one table — you can load JSON data in one session, and delimited data in another.

The following basic examples illustrate this usage.

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

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

```
CREATE TABLE
```

2. Enter values using the `fjsonparser()`, and query the results:

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

3. Enter delimited values using the `fdelimitedparser()`:

```
dbt=> 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
>> \.
```

4. Query the table:

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

Notice that both JSON data and delimited data are saved in the same columnar table, `super`.

Computing Flex Table Keys

After loading data into a flex table, one of the first tasks to complete is to know what key value pairs exist as populated virtual columns in the data. Two helper functions compute keys from map data. The second function performs the same functionality as the first, but also builds a view, as described in Updating Flex Table Views:

- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

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, casting the values it computes to VARCHAR, LONG VARCHAR, or LONG VARBINARY, depending on the length of the key, and whether the key includes nested maps.

The following examples illustrate this function and the results of populating the _keys table, once you've created a flex table (darkdata1) and loaded data:

```
kdb=> create flex table darkdata1();
CREATE TABLE
kdb=> copy darkdata1 from '/test/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
kdb=> select compute_flextable_keys('darkdata1');
            compute_flextable_keys
--------------------------------------------------
 Please see public.darkdata1_keys for updated keys
(1 row)
kdb=> select * from darkdata1_keys;
                        key_name                         | frequency |   data_type_guess
---------------------------------------------------------+-----------+------------------
----
 contributors                                            |         8 | varchar(20)
 coordinates                                             |         8 | varchar(20)
```

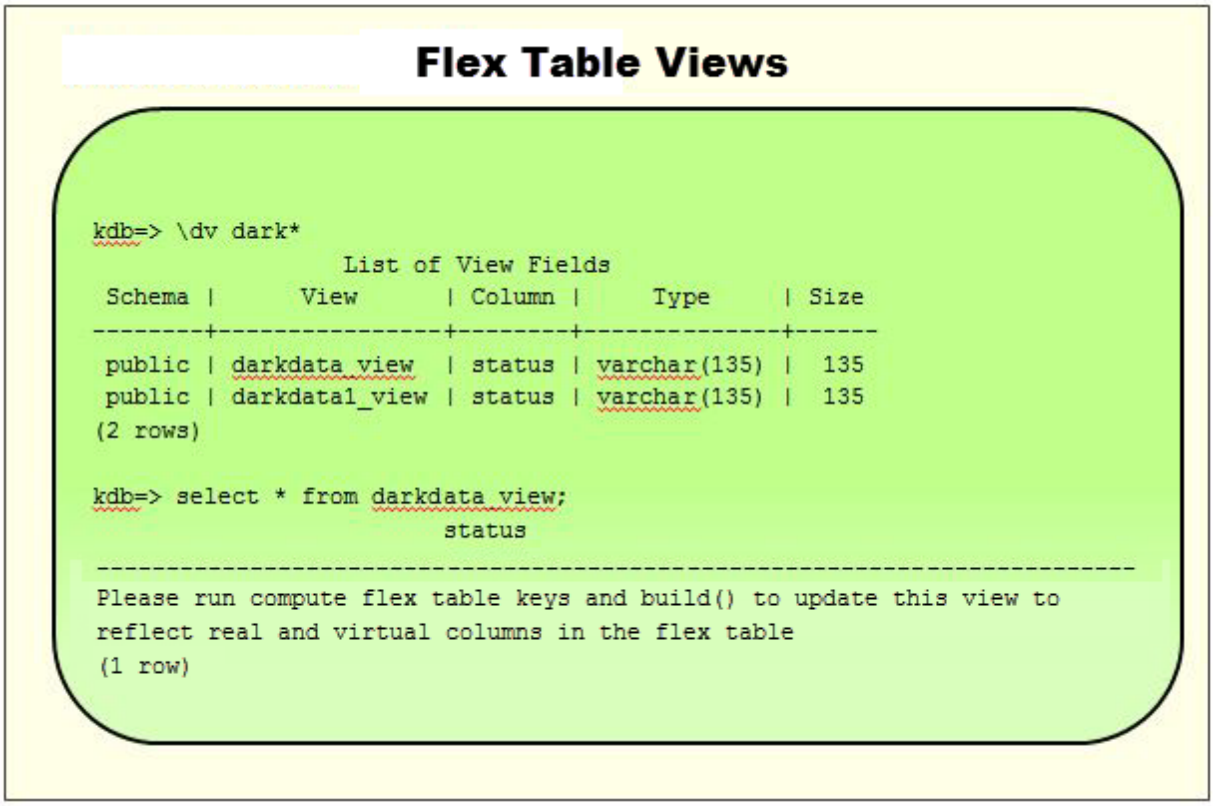
```
 created_at                                          |         8 | varchar(60)
 entities.hashtags                                   |         8 | long varbinary(18
6)
 entities.urls                                       |         8 | long varbinary(3
2)
 entities.user_mentions                              |         8 | long varbinary(67
4)
.
.
.
 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, cast to VARCHAR, LONG VARCHAR or LONG VARBINARY, depending 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, as calculated by the longest field, multiplied by the FlexTableDataTypeGuessMultiplier configuration parameter value. For more information, see Setting Flex Table Parameters.

Updating Flex Table Views

Creating Flex tables also creates an associated, default view at the same time. Selecting from the view prompts you to run a helper function, as shown in this example:



Two helper functions create views. Because the second function also computes keys, the hint is to use that one:

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

Using BUILD_FLEXTABLE_VIEW

After computing 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 that exist for the key, or NULL if they do not.

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.
`build_flextable_view ('flexible_table', 'view_name')`	Changes the view you specify with *view_name* from 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* with the current contents of the *flexible_table_keys* table. Use this function to change a view of your choice with the contents of keys you are interested in.

Handling Duplicate Key Names in JSON

SQL is case-insensitive, so the names `TEST`, `test`, and `TeSt` are identical.

JSON data is case sensitive, and 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:

```
dbt=> 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 leng
th 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 keys 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. The values from all duplicate keys are saved in that column. 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.

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

```
dbt=> \x
Expanded display is on.
dbt=> select * from dupe_keys;
-[ RECORD 1 ]---+-----------------------------------------------------------------------------
-----------------------------------------------------------
key_name        | TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttes
tTesttestTesttestTesttestTesttestTesttestTesttestTesttest
frequency       | 2
data_type_guess | varchar(20)
-[ RECORD 2 ]---+-----------------------------------------------------------------------------
-----------------------------------------------------------
key_name        | TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttes
tTesttestTesttestTesttestTesttestTesttestTesttestTest12345
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`), illustrating the consolidated `test` and `testtesttest...` virtual columns, with all of the `test`, `Test`, and `tESt` key values in the `test` column:

```
dbt=> select * from dupe_view;
  test  | testtesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttes
ttesttesttesttesttesttesttesttesttesttesttest
--------+-----------------------------------------------------------------------------
-----------------------------------------------
 upper2 |
 half4  |
```

```
 lower1 |
 upper1 |
 half1  |
 half4  |
        |
 lower1 |
 half1  |
 upper2 |
        |
 lower2 |
 lower3 |
 upper1 |
 lower2 |
 lower3 |
(16 rows)
```

Creating a Flex Table View

The following example creates a view, `dd_view`, from the Flex table `darkdata`, which contains JSON data.

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

Querying the view shows the key names you specified, and their values:

```
kdb=> 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 calls `build_flextable_view` with the original table, and the view you previously created, `dd_view`:

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

```
(1 row)
```

Querying the view again shows that the function populated the view with the contents of the darkdata_keys table. The next example shows a snippet from the results, with the `key_name` columns and their values:

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

```
kdb=> 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.
.
.
.
```

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.

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:

```
kdb=> 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)
```

Altering Flex Tables

Once flex tables exist, you can alter the table structure and contents. This section describes some aspects of adding columns, constraints, and default table values.

Note: Adding standard named columns with map key names materializes those virtual columns. HP Vertica strongly recommends that you materialize virtual columns before you start running large queries. Materializing virtual columns can significantly increase query performance, and is recommended, even at the cost of slightly increasing load times.

Adding Columns to Flex Tables

Add columns to your flex tables to materialize virtual columns:

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

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

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

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

3. Query the materialized column from the flex table:

```
kdb=> 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 Loading Flex Table Data describes the use of default values, and how they are evaluated during loading.

When you add columns with default values to a flex table (recommended), subsequently loading data overrides any column-defined default value, unless you specify a COPY value expression.

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`"):

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

2. Load some JSON data:

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

3. Query the talker column values, 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:

```
kdb=> 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__`:

```
kdb=> 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):

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

7. Load data again, this time without __raw__:

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

8. Query the two real columns and see that `talker` is `NULL` (__raw__ not specified), but `user.lang` has the key values from the data you loaded:

```
kdb=> 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`:

```
kdb=> 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:

```
kdb=> 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 ignores the default value of the column definition. However, you do not require an explicit COPY expression to cause the default expression to be ignored.

Changing the Default Size of __raw__ Columns

You can change the default size of the `__raw__` column for flex tables, 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 General Parameters):

```
VMart=> select set_config_parameter ('FlexTableRawSize',120000);
    set_config_parameter
----------------------------
 Parameter set successfully
(1 row)
```

Changing the configuration parameter will affect 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:

```
VMart=> 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 the column already contains.

Changing Standard and Virtual Table Columns

You can make the following changes to the default flex table columns (__raw__ and __identity__), and the virtual columns in the map data:

Actions	__raw__	__identity__	Virtual
Add or change NOT NULL constraints	Yes	Yes	No
Add PK/FK constraints	Yes	Yes	No
Create projections	Yes	Yes	No
Segment	No	Yes	No
Partition	No	Yes	No
Specify a user-defined scalar function (UDSF) as a default column expression in `ALTER TABLE x ADD COLUMN y` statement	No	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 materialized columns exist, flex tables are segmented on the __identity__ column.

Querying Flex Tables

Once you've created your flex table, with or without additional columns, and loaded data (JSON twitter data in these examples) what kinds of queries can you perform?

- SELECT
- COPY
- TRUNCATE
- DELETE

You can use SELECT queries for real columns if they exist, and from virtual columns that exist in the table's `__raw__` column, if same-name real columns do not exist. Column names are case insensitive.

Unsupported DDL and DML Commands for Flex Tables

You cannot use the following DDL commands with flex table and doing so results in an error:

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

Getting Key Values From Flex Table Data

Two helper functions compute keys from a flex table:

- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

The second function also builds a view from the data loaded into the table's `__raw__` column.

You can use the mapKeys and mapKeysInfo functions directly. See Working with Flex Table Map Functions for examples.

To compute the key values:

1. Call the function as follows:

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

2. Query the `darkdata_keys` table to see the computed key names:

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

```
                         key_name                         | frequency |   data_type_g
uess
----------------------------------------------------------+-----------+--------------
--------
 contributors                                             |         8 | varchar(20)
 coordinates                                              |         8 | varchar(20)
 created_at                                               |         8 | varchar(60)
 entities.hashtags                                        |         8 | long varbinar
y(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 Key Values

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

```
kdb=> 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 key values as required, and use functions in your select queries. The next example queries the darkdata1 flex table for the created_at and retweet_count key values, casting them in the process:

```
kdb=> select "created_at"::TIMESTAMP, "retweet_count"::INT from darkdata1 order by 1 des
c;
     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:

```
kdb=> select "user.lang", count (*), avg(length("text"))::int from darkdata1 group by 1 o
rder by 2 desc;
 user.lang | count | avg
-----------+-------+-----
 en        |     4 |  42
           |     4 |
 es        |     2 |  96
 it        |     1 |  50
 tr        |     1 |  16
(5 rows)
```

Comparing Queries With and Without Casting

The following query gets created_at data without casting:

```
kdb=> 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 data to a TIMESTAMP, resulting in different output and the regional time:

```
kdb=> 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
```

Querying Under the Covers

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

```
maplookup(_raw_, 'which_column')
```

The maplookup() function searches the data map 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 NULL handling, see mapLookup().

Accessing an Epoch Key

The term EPOCH is reserved in HP Vertica for internal use.

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

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 flex table. The `__raw__` column contains the map data you load into the table. The column data type is a `LONG VARBINARY`. 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. Because there is no way to determine the column width of future loads, the 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: • 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.

Working with Flex Table Map Functions

The flex table map functions let you extract and manipulate the map data contents of any flex tables you create. All of the map functions are available after you upgrade your site.

The flex table map functions are applicable for use with flex tables, their associated *{flextable}*_keys table and automatically generated *{flextable}*_view views. Using these functions is not applicable for use with standard HP Vertica tables.

All map functions (except for emptyMap), accept either `LONG VARBINARY` or `LONG VARCHAR` as the map argument, and return `LONG VARCHAR` values to support conversion.

mapAggregate

Returns a `LONG VARBINARY` `raw_map` with key/value pairs supplied from two VARCHAR input columns of an existing columnar table. Using this function requires using an `over()` clause for the source table, as shown in the example.

Usage

`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 `raw_map` data.
`source_column2`	Table column with values to use as the values in the key/value pair of the returned `raw_map` data.

Examples

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

```
dbs=> create table btest(keys varchar(10), values varchar(10));
CREATE TABLE
dbs=>  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
>> \.
```

Once the `btest` table is populated, call `mapaggregate()` as follows to return the `raw_map` data:

```
dbs=>  select mapaggregate(keys, values) over() from btest;
                                                                                                raw_m
ap
-------------------------------------------------------------------------------------------------
-----------
-------------------------------------------------------------------------
 \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:

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

mapContainsKey

Scalar function to determine whether the map data contains a virtual column (key). 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, which that function uses for both a non-existent key, and an existing key with a NULL value.

Usage

```
mapcontainskey(raw_map_data, 'virtual_column_name')
```

Arguments

`raw_map_data`	Table column name of raw map data (usually from the `__raw__` column in a flex table).
`virtual_column_name`	The name of the key to check.

Examples

This example uses both mapLookup and `mapcontainskey()` to determine whether the empty fields indicate a NULL value for the row, listed as (t), or no value. This example color codes the keys with no value (f):

```
kdb=> select maplookup(__raw__, 'user.location'), mapcontainskey(__raw__, 'user.locatio
n') 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)
```

mapContainsValue

Scalar function to determine whether the map data contains a specific value. Returns true (`t`) if the value exists, or false (`f`) if it does not.

Usage

```
mapcontainsvalue(raw_map_data, 'virtual_column_value')
```

Arguments

`raw_map_data`	Table column name of raw map data (usually from the `__raw__` column in a flex table).
`virtual_column_value`	The value whose existence you want to confirm.

Examples

This example shows a flex table (`ftest`), populated with some values keys and values. Both keys (virtual columns) are named `one`:

```
dbs=> create flex table ftest();
```

```
CREATE TABLE
dbs=> 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"}
>> \.
```

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

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

mapItems

Transform function that returns the virtual columns and their values present in the raw map data of a flex table. This function requires an `over()` clause, as shown in the example.

Usage

```
mapItems(map_data)
```

Arguments

`map_data`	Table column name of map data (usually from the `__raw__` column in a flex table).

Examples

This example uses a flex table `darkmountain`, populated with JSON data. This query returns the number of virtual columns found in the map data:

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

This example shows a snippet of the return data querying a flex table of all items in the map data:

```
dbt=> select * from (select mapitems(darkmountain.__raw__) over() 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)
```

mapKeys

Transform function that returns the virtual columns present in the map data of a flex table. This function requires an `over()` clause, as shown in the example.

Usage

```
mapkeys(map_data)
```

Arguments

`map_data`	Table column name of map data (usually from the `__raw__` column in a flex table).

Examples

This example uses a flex table `darkdata`, populated with JSON tweet data. This query returns the number of virtual columns found in the map data:

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

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

```
kdb=> select * from (SELECT mapkeys(darkdata.__raw__) OVER() 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)
```

mapKeysInfo

Transform function that returns virtual column information in a given map. This function requires an `over()` clause, as shown in the example. This function is a superset of the mapKeys() function, returning 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, stored as a `LONG VARBINARY`.
`row_num`	The number of rows in which the key was found. Currently, this is always `1`.
`field_num`	The field number in which the key exists.

Usage

```
mapkeysinfo(map_data)
```

Arguments

map_data	Table column name of map data (usually from the __raw__ column in a flex table).

Examples

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

```
kdb=> select * from (SELECT mapkeysinfo(darkdata.__raw__) OVER() 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)
```

mapLookup

Transform function that returns values associated with a single key. Returns a `LONG VARCHAR` with key values, or `NULL` if the key does not exist. Column names are case insensitive.

Before using maplookup, use these functions to find out about your map data:

- Call maptostring() to return the contents of your map data in a formatted text output
- Call mapContainsKey() to determine whether a key exists in the map data

You can control the behavior for non-scalar values when loading data with the fjsonparser and its `flatten-arrays` argument. See Using FlexTable Parsers.

Usage

```
maplookup (map_data, 'virtual_column_name' [USING PARAMETERS case_sensitive=
{false | true}] )
```

Arguments and Parameter

`map_data`	Flex table column containing map data (usually from the `__raw__` column in a flex table).
`virtual_column_name`	The name of the virtual column to locate in the map.
`case_sensitive`	[Optional parameter] Default=`false` Specifies whether to return virtual columns if keys with difference cases exist, as illustrated in the examples. Use as follows: `(... USING PARAMETERS case_sensitive=true)`

Examples

This example shows how to return the values of one key column, `user.location`, including some empty fields:

```
kdb=>  select maplookup(__raw__, 'user.location') from darkdata order by 1;
 maplookup
-----------
 Chile
 Narnia
 Uptown
```

```
 .
 .
 chicago
(12 rows)
```

Interpreting Empty Fields

When maplookup returns key values for each row, the vsql display has empty fields in these cases:

- The key does not exist
- Key exists but row contains a NULL
- Key exists but row does not contain a value

To determine what empty fields indicate when looking up a key, use the mapContainsKey function, to narrow the possibilities.

For example, if you use maplookup without first checking a key's existence, the following output (for 12 rows of JSON) is ambiguous. Does a `WinstonChurchill` key exist, or does it exist with `NULL` values for each row?

```
kdb=> select maplookup(__raw__, 'WinstonChurchill') from darkdata;
 maplookup
-----------
(12 rows)
```

After determining that a key exists, you can interpret what empty fields indicate using the mapContainsKey() function in conjunction with `maplookup()`.

The next example uses the functions together to disambiguate empty field values. Empty mapLookup rows with mapcontainskey `t` indicate a NULL key value. Empty maplookup rows with mapcontainskey `f` do not contain a value:

```
kdb=> select maplookup(__raw__, 'user.location'), mapcontainskey(__raw__, 'user.locatio
n') 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)
```

Querying Data From Nested Maps

If your map data consists of nested maps of arbitrary depth, you can call `maplookup()` recursively. The innermost map will always be __raw__, just as a single invocation is.

The next example first uses `maptostring()` to return the map contents of the table `bake`, so you can see the contents of the map data in the examples that follow:

```
kdb=> 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)
```

The next examples illustrate using several invocations of `maplookup()` to return values from the map __raw__ column (after creating the table and loading data). Compare the returned results to the `maptostring()` output in the previous example.

```
kdb=> create flex table bake();
CREATE TABLE
kdb=> copy bake from '/vertica/test/flextable/DATA/bake.json' parser ujsonparser(flatten_
arrays=1,flatten_maps=0);
 Rows Loaded
```

```
-------------
           1
(1 row)
kdb=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw__,'items'),'item.0')
,'batters'),'batter.0'),'type') from bake;
 maplookup
-----------
 Regular
(1 row)
kdb=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw__,'items'),'item.0')
,'batters'),'batter.1'),'type') from bake;
 maplookup
-----------
 Chocolate
(1 row)
kdb=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw__,'items'),'item.0')
,'batters'),'batter.2'),'type') from bake;
 maplookup
-----------
 Blueberry
(1 row)
kdb=> select maplookup(maplookup(maplookup(maplookup(maplookup(__raw__,'items'),'item.0')
,'batters'),'batter.3'),'type') from bake;
  maplookup
--------------
 Devil's Food
(1 row)
```

Checking 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"
}
{
  "TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTes
ttestTesttestTesttestTesttestTesttestTesttest":"1",
    "TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestT
esttestTesttestTesttestTesttestTesttestTest12345":"2"
}
```

2. Create a flex table, `dupe`, and load the JSON file:

```
dbt=> create flex table dupe();
CREATE TABLE
dbt=> copy dupe from '/home/release/KData/repeated_key_name.json' parser fjsonparser()
;
 Rows Loaded
-------------
           8
(1 row)
```

mapSize

Scalar function that returns the number of key virtual columns present in the map data.

Usage

`mapsze(map_data)`

Arguments

`map_data`	Table column name of map data (usually from the `__raw__` column in a flex table).

Examples

This example shows the returned sizes from the number of keys in the flex table `darkmountain`:

```
dbt=> select mapsize(__raw__) from darkmountain;
 mapsize
```

```
---------
       3
       4
       4
       4
       4
(5 rows)
```

mapToString

Transform function that recursively builds a string representation of flex table map data, including nested JSON maps, and displays the contents in a readable format. The function returns a `LONG VARCHAR`. Use `maptostring` to see how map data is nested before querying virtual columns with mapLookup().

Usage

`maptostring(`*`map_data`* `[using parameters canonical_json={`*`true`* `|` *`false`*`}])`

Arguments

`canonical_json`	=*bool* [Optional parameter] Default `canonical-json=true` 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.

Examples

The following example creates a sample flex table, `boo`. Then, after loading sample JSON data from STDIN, continues by calling `maptostring()` twice with both values for the `canonical_json` parameter, illustrating the different results on the flex table `__raw__` column data.

1. Create sample table:

```
dbs=> create flex table boo();
CREATE TABLE
```

2. Load data from STDIN:

```
dbs=> copy boo 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, which you do not need to specify. Notice the flex table contents using canonical JSON output. The function returns the first duplicate key and its value (`"aaa": "1"`), and omits any remaining duplicate keys (`"aaa": "2"`):

```
dbs=>  select maptostring(__raw__) from boo;
                        maptostring
------------------------------------------------------------
 {
   "AAA" : "3",
   "aaa" : "1",
   "bbb" : "aaa\"bbb"
}

(1 row)
```

4. Call `maptostring()` with its non-default behavior (`canonical_json=false`). Using the optional parameter specified as `false`, the function returns duplicate keys and their values:

```
dbs=> select maptostring(__raw__ using parameters canonical_json=false) from boo;
                         maptostring
-------------------------------------------------------------

{
        "aaa":  "1",
        "aaa":  "2",
        "AAA":  "3",
        "bbb":  "aaa"bbb"
 }
(1 row)
```

mapValues

Transform function to return a string representation of the top-level map data values. This function requires an `over()` clause when you use it, as shown in the example.

Usage

```
mapvalues(map_data)
```

Arguments

`map_data`	Flex table column containing map data (usually from the `__raw__` column in a flex table).

Examples

The following example uses `mapvalues()` with the `darkmountain` flex table, returning the values.

```
VMart=> select * from (select mapvalues(darkmountain.__raw__) over() from darkmountain) a
s 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)
```

mapVersion

Scalar function that returns whether the map data is a valid map, and if so, what version. Returns either the map version (such as `1`), or `-1` if the raw data is not valid.

Usage

```
mapversion(map_data)
```

Arguments

`map_data`	Flex table column containing map data (usually from the `__raw__` column in a flex table).

Examples

The following example uses `mapversion()` with the `darkmountain`flex table, returning the version 1 for the map data:

```
dbt=> select mapversion(__raw__) from darkmountain;
 mapversion
------------
          1
          1
          1
          1
          1
(5 rows)
```

emptyMap

Transform function to construct a new empty map with one row, but without keys or data.

Usage

`emptymap()`

Arguments

None

Examples

To create an empty map:

```
kdb=> select emptymap();                             emptymap
-------------------------------------------------------------------
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
(1 row)
```

If you create an empty map from an existing flex table, the new map will have the same number of rows as the table from which it was created. For example, creating an empty map from the `darkdata` table, which has 12 rows of JSON data has the following result:

```
kdb=> 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)
```

Flex Table Data Functions

The Flex table data helper functions supply information you'll need to query the data you've loaded. For example, if you don't know what keys are available in the map data, you can use the COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW function to populate a keys table and build a view.

While the functions are available to all users, they are applicable only to flex table, their associated *flex_table*_keys table and *flex_table*_view views. By computing keys and creating views from flex table data, the functions facilitate SELECT queries. One function restores the original keys table and view that were made when you first created the flex table.

Flex Table Dependencies

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

1. *flextable*_keys
2. *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.

Dropping Flex Tables and Views

The helper functions automatically use the dependent table and view if they are internally linked with the parent table, which both are when you create the flex table. If you drop either the _keys table or the _view, and recreate objects of the same name, 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 by dropping the _keys table and the _view before calling the RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW function. Calling this function recreates 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 recreates, a view for a default or user-defined `_keys` table. 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 do the following:

1. Drop the default-named view
2. Create your own view of the same name

Usage

```
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 with 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

Following are examples of calling `build_flextable_view` with 1, 2, or 3 arguments.

Creating a Default View

To create, or recreate, a default view:

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

```
kdb=> 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`):

```
kdb=> 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 recreate, 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, in this example:

```
kdb=> 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`):

```
kdb=> 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`).

Following are a couple of ways to create a custom keys table:

1. Create a table with the all keys from the keys table:

```
kdb=> 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:

```
kdb=> 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:

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

Continue once your custom keys table exists.

1. Call the function with all arguments, a flex table, the name of the view to create, and the custom keys table:

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

2. 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 map 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 build a view as well, use COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW.

Usage

`compute_flextable_keys('`*`flex_table`*`')`

Arguments

flex_table	The name of the flex table.

Examples

During execution, this function determines a data type for each virtual column, casting the values it computes to `VARCHAR`, `LONG VARCHAR`, or `LONG VARBINARY`, depending on the length of the key, and whether the key includes nested maps.

The following examples illustrate this function and the results of populating the _keys table, once you've created a flex table (darkdata1) and loaded data:

```
kdb=> create flex table darkdata1();
CREATE TABLE
kdb=> copy darkdata1 from '/test/flextable/DATA/tweets_12.json' parser fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
kdb=> select compute_flextable_keys('darkdata1');
            compute_flextable_keys
-----------------------------------------------
 Please see public.darkdata1_keys for updated keys
(1 row)
kdb=> 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(18
6)
 entities.urls                                            |         8 | long varbinary(3
2)
 entities.user_mentions                                   |         8 | long varbinary(67
4)
.
.
.
 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, cast to `VARCHAR`, `LONG VARCHAR` or `LONG VARBINARY`, depending 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, as calculated by 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 map data of a flex table , and construct a view. If you don't need to perform both operations together, use one of the single-operation functions.

Usage

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

Arguments

flex_table	The name of a flex table.

Examples

The following example calls the function for the darkdata flex table.

```
kdb=> 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 that are listed as *key_names* in the *flextable_keys* table. You can optionally indicate the number of columns to materialize, and use a keys table other than the default. If you do not specify the number of columns, the function materializes up to 50 virtual column key names. Calling this function requires that you first compute flex table keys using either COMPUTE_FLEXTABLE_KEYS or COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW .

Note: Materializing any virtual column into a real column with this function affects data storage limits. Each materialized column counts against the data storage limit of your HP Vertica Enterprise Edition (EE) license. This increase is reflected when HP Vertica next performs a license compliance audit. To manually check your EE license compliance, call the `audit()` function, described in the SQL Reference Manual.

Usage

```
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, skipping any columns already materialized. 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. HP VERTICA tables support a total of 1600 columns, which is the greatest 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 attempts to materialize the number of columns (value of *n-columns*) from *keys_table_name*, skipping any columns already materialized. The function orders the materialized results by frequency, descending, *key_name*when materializing the first `n` columns.

Examples

The following example loads 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, the example calls `materialize_flextable_columns` to materialize the first four columns:

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

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

dbt=> 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 = 'publi
c' 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. Following is an example of running that query:

```
dbt=> 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, linking them with their associated flex table if either is dropped. This function notes whether it restores one or both.

Usage

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

Arguments

flex_table	The name of the flex table .

Examples

This example invokes the function with an existing flex table, restoring both the _keys table and _view:

```
kdb=> 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 the function restoring darkdata_view, but noting that darkdata_keys does not need restoring:

```
kdb=> 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:

```
kdb=> 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

We appreciate your 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 Flex Tables Guide (Vertica Analytic Database 7.0.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@hp.com.