Cloudera Runtime 7.1.4

Using Apache Impala with Apache Kudu

Date published: 2020-02-28
Date modified: 2020-10-09

https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Using Apache Impala with Apache Kudu 4
Impala database containment model 4

Internal and external Impala tables 4

Using Impala to query Kudu tables 4
Querying an existing Kudu table from Impala 5
Creating a new Kudu table from Impala 5
CREATE TABLE AS SELECT 7
Partitioning tables 7
Basic partitioning 7
Advanced partitioning 9
Non-covering range partitions 10
Partitioning guidelines 11
Optimizing performance for evaluating SQL predicates 11
Inserting a row 11
Inserting in bulk 11
INSERT and primary key uniqueness violations 12
Updating a row 12
Updating in bulk 12
Upserting a row 12
Altering a table 14
Deleting a row 14
Deleting in bulk 14
Failures during INSERT, UPDATE, UPSERT, and DELETE operations 14
Altering table properties 14
Dropping a Kudu table using Impala 15
Security considerations 15
Known issues and limitations 15
Next steps 16

Using Apache Impala with Apache Kudu

Apache Kudu has tight integration with Apache Impala, allowing you to use Impala to insert, query, update, and delete data from Kudu tablets using Impala's SQL syntax, as an alternative to using the Kudu APIs to build a custom Kudu application. In addition, you can use JDBC or ODBC to connect existing or new applications written in any language, framework, or business intelligence tool to your Kudu data, using Impala as the broker.

Impala database containment model

Every Impala table is contained within a namespace called a database. The default database is called default, and you may create and drop additional databases as desired. To create the database, use a CREATE DATABASE statement. To use the database for further Impala operations such as CREATE TABLE, use the USE statement.

For example, to create a table in a database called impala_kudu, use the following statements:

```
CREATE DATABASE impala_kudu;
USE impala_kudu;
CREATE TABLE my_first_table (
...
```

The my_first_table table is created within the impala_kudu database.

The prefix impala:: and the Impala database name are appended to the underlying Kudu table name: impala::<database>.<table>

For example, to specify the my_first_table table in database impala_kudu, as opposed to any other table with the same name in another database, refer to the table as impala::impala_kudu.my_first_table. This also applies to INSERT, UPDATE, DELETE, and DROP statements.

Internal and external Impala tables

When creating a new Kudu table using Impala, you can create the table as an internal table or an external table.

Internal

An internal table (created by CREATE TABLE) is managed by Impala, and can be dropped by Impala. When you create a new table using Impala, it is generally a internal table. When such a table is created in Impala, the corresponding Kudu table will be named impala::database_name.table_name. The prefix is always impala::, and the database name and table name follow, separated by a dot.

External

An external table (created by CREATE EXTERNAL TABLE) is not managed by Impala, and dropping such a table does not drop the table from its source location (here, Kudu). Instead, it only removes the mapping between Impala and Kudu. This is the mode used in the syntax provided by Kudu for mapping an existing table to Impala.

Using Impala to query Kudu tables

Neither Kudu nor Impala need special configuration in order for you to use the Impala Shell or the Impala API to insert, update, delete, or query Kudu data using Impala. However, you do need to create a mapping between the Impala and Kudu tables. Kudu provides the Impala query to map to an existing Kudu table in the web UI.

- Make sure you are using the impala-shell binary provided by the default CDP Impala binary. The following example shows how you can verify this using the alternatives command on a RHEL 6 host. Do not copy and paste the alternatives --set command directly, because the file names are likely to differ.

```
$ sudo alternatives --display impala-shell

impala-shell - status is auto.
 link currently points to /opt/cloudera/parcels/<current_CDP_parcel>/bin/
impala-shell
/opt/cloudera/parcels/<current_CDP_parcel>/bin/impala-shell - priority 10
Current `best' version is opt/cloudera/parcels/<current_CDP_parcel>/bin/i
mpala-shell
```

- Although not necessary, it is recommended that you configure Impala with the locations of the Kudu Masters using the --kudu_master_hosts=<master1>[:port] flag. If this flag is not set, you will need to manually provide this configuration each time you create a table by specifying the kudu.master_addresses property inside a TBLP ROPERTIES clause. If you are using Cloudera Manager, no such configuration is needed. The Impala service will automatically recognize the Kudu Master hosts. However, if your Impala queries don't work as expected, use the following steps to make sure that the Impala service is set to be dependent on Kudu:

  1. Go to the Impala service.
  2. Click the Configuration tab and search for kudu.
  3. Make sure that the Kudu Service property is set to the right Kudu service.
  4. Click Save Changes.

  Before you carry out any of the operations listed within this section, make sure that this configuration has been set.
- Start Impala Shell using the impala-shell command. By default, impala-shell attempts to connect to the Impala daemon on localhost on port 21000. To connect to a different host, use the -i <host:port> option.

  To automatically connect to a specific Impala database, use the -d <database> option. For instance, if all your Kudu tables are in Impala in the database impala_kudu, use -d impala_kudu to use this database.
- To quit the Impala Shell, use the following command: quit;

Querying an existing Kudu table from Impala

Tables created through the Kudu API or other integrations such as Apache Spark are not automatically visible in Impala.

To query them, you must first create an external table within Impala to map the Kudu table into an Impala database:

```
CREATE EXTERNAL TABLE my_mapping_table
STORED AS KUDU
TBLPROPERTIES (
  'kudu.table_name' = 'my_kudu_table'
);
```

Creating a new Kudu table from Impala

Creating a new table in Kudu from Impala is similar to mapping an existing Kudu table to an Impala table, except that you need to specify the schema and partitioning information yourself. Use the examples in this section as a guideline. Impala first creates the table, then creates the mapping.

In the CREATE TABLE statement, the columns that comprise the primary key must be listed first. Additionally, primary key columns are implicitly considered NOT NULL.

When creating a new table in Kudu, you must define a partition schema to pre-split your table. The best partition schema to use depends upon the structure of your data and your data access patterns. The goal is to maximize parallelism and use all your tablet servers evenly.

Note: In Impala included in CDH 5.13 and higher, the PARTITION BY clause is optional for Kudu tables. If the clause is omitted, Impala automatically constructs a single partition that is not connected to any column. Because such a table cannot take advantage of Kudu features for parallelized queries and query optimizations, omitting the PARTITION BY clause is only appropriate for small lookup tables.

The following CREATE TABLE example distributes the table into 16 partitions by hashing the id column, for simplicity.

```
CREATE TABLE my_first_table
(
  id BIGINT,
  name STRING,
  PRIMARY KEY(id)
)
PARTITION BY HASH PARTITIONS 16
STORED AS KUDU;
```

By default, Kudu tables created through Impala use a tablet replication factor of 3. To specify the replication factor for a Kudu table, add a TBLPROPERTIES clause to the CREATE TABLE statement as shown below where n is the replication factor you want to use:

```
TBLPROPERTIES ('kudu.num_tablet_replicas' = 'n')
```

A replication factor must be an odd number.

Changing the kudu.num_tablet_replicas table property using the ALTER TABLE currently has no effect.

The Impala SQL Reference *CREATE TABLE* topic has more details and examples.

The mapping between Kudu and Impala data types is summarized in the the following table:

Kudu type	Impala type
BOOL (boolean)	BOOLEAN
INT8 (8-bit signed integer)	TINYINT
INT16 (16-bit signed integer)	SMALLINT
INT32 (32-bit signed integer)	INT
INT64 (64-bit signed integer)	BIGINT
DATE (32-bit days since the Unix epoch)	NA (Not supported by Impala)
UNIXTIME_MICROS (64-bit microseconds Unix epoch)	TIMESTAMP (For details, see *Notes*.)
FLOAT (single-precision 32-bit IEEE-754 floating-point)	FLOAT
DOUBLE (double-precision 64-bit IEEE-754 floating-point)	DOUBLE
DECIMAL	DECIMAL
VARCHAR	VARCHAR
STRING	STRING
BINARY	BINARY (Not yet supported. For details, see IMPALA-5323.)

Note: For more details about Timestamp data type, see *TIMESTAMP data type*. For more details on handling Kudu types, column attributes, partitioning, and using Impala to create and query Kudu tables, see Column Design and Handling Date, Time, or Timestamp Data with Kudu.

Related Information

Partitioning tables

TIMESTAMP data type

CREATE TABLE AS SELECT

You can create a table by querying any other table or tables in Impala, using a CREATE TABLE ... AS SELECT statement.

The following example imports all rows from an existing table, old_table, into a new Kudu table, new_table. The columns in new_table will have the same names and types as the columns in old_table, but you will need to additionally specify the primary key and partitioning schema.

```
CREATE TABLE new_table
PRIMARY KEY (ts, name)
PARTITION BY HASH(name) PARTITIONS 8
STORED AS KUDU
AS SELECT ts, name, value FROM old_table;
```

You can refine the SELECT statement to only match the rows and columns you want to be inserted into the new table. You can also rename the columns by using syntax like SELECT name as new_col_name.

Partitioning tables

Tables are partitioned into tablets according to a partition schema on the primary key columns. Each tablet is served by at least one tablet server. Ideally, a table should be split into tablets that are distributed across a number of tablet servers to maximize parallel operations. The details of the partitioning schema you use will depend entirely on the type of data you store and how you access it.

Kudu currently has no mechanism for splitting or merging tablets after the table has been created. Until this feature has been implemented, you must provide a partition schema for your table when you create it. When designing your tables, consider using primary keys that will allow you to partition your table into tablets which grow at similar rates.

You can partition your table using Impala's PARTITION BY clause, which supports distribution by RANGE or HASH. The partition scheme can contain zero or more HASH definitions, followed by an optional RANGE definition. The RANGE definition can refer to one or more primary key columns. Examples of basic and advanced partitioning are shown below.

Monotonically Increasing Values - If you partition by range on a column whose values are monotonically increasing, the last tablet will grow much larger than the others. Additionally, all data being inserted will be written to a single tablet at a time, limiting the scalability of data ingest. In that case, consider distributing by HASH instead of, or in addition to, RANGE.

Note: Impala keywords, such as group, are enclosed by back-tick characters when they are used as identifiers, rather than as keywords.

In general, be mindful the number of tablets limits the parallelism of reads, in the current implementation. Increasing the number of tablets significantly beyond the number of cores is likely to have diminishing returns.

Basic partitioning

In basic partitioning, you can either partition by range, or partition by hash.

PARTITION BY RANGE

You can specify range partitions for one or more primary key columns. Range partitioning in Kudu allows splitting a table based on specific values or ranges of values of the chosen partition keys. This allows you to balance parallelism in writes with scan efficiency.

For instance, if you have a table that has the columns state, name, and purchase_count, and you partition the table by state, it will create 50 tablets, one for each US state.

```
CREATE TABLE customers (
  state STRING,
  name STRING,
  purchase_count int,
  PRIMARY KEY (state, name)
)
PARTITION BY RANGE (state)
(
  PARTITION VALUE = 'al',
  PARTITION VALUE = 'ak',
  PARTITION VALUE = 'ar',
  ...
  ...
  PARTITION VALUE = 'wv',
  PARTITION VALUE = 'wy'
)
STORED AS KUDU;
```

PARTITION BY HASH

Instead of distributing by an explicit range, or in combination with range distribution, you can distribute into a specific number of partitions by hash. You specify the primary key columns you want to partition by, and the number of partitions you want to use. Rows are distributed by hashing the specified key columns. Assuming that the values being hashed do not themselves exhibit significant skew, this will serve to distribute the data evenly across all partitions.

You can specify multiple definitions, and you can specify definitions which use compound primary keys. However, one column cannot be mentioned in multiple hash definitions. Consider two columns, a and b:

- HASH(a), HASH(b) -- will succeed
- HASH(a,b) -- will succeed
- HASH(a), HASH(a,b) -- will fail

Note: PARTITION BY HASH with no column specified is a shortcut to create the desired number of partitions by hashing all primary key columns.

Hash partitioning is a reasonable approach if primary key values are evenly distributed in their domain and no data skew is apparent, such as timestamps or serial IDs.

The following example creates 16 tablets by hashing the id column. A maximum of 16 tablets can be written to in parallel. In this example, a query for a range of sku values is likely to need to read from all 16 tablets, so this may not be the optimum schema for this table. See the Advanced partitioning on page 9 section for an extended example.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
```

```
)
PARTITION BY HASH PARTITIONS 16
STORED AS KUDU;
```

Advanced partitioning

You can combine HASH and RANGE partitioning to create more complex partition schemas. You can also specify zero or more HASH definitions, followed by zero or one RANGE definitions. Each schema definition can encompass one or more columns. While enumerating every possible distribution schema is out of the scope of this topic, the following examples illustrate some of the possibilities.

PARTITION BY HASH and RANGE

Consider the basic PARTITION BY HASH example above. If you often query for a range of sku values, you can optimize the example by combining hash partitioning with range partitioning.

The following example still creates 16 tablets, by first hashing the id column into 4 partitions, and then applying range partitioning to split each partition into four tablets, based upon the value of the sku string. At least four tablets (and possibly up to 16) can be written to in parallel, and when you query for a contiguous range of sku values, there's a good chance you only need to read a quarter of the tablets to fulfill the query.

By default, the entire primary key (id, sku) will be hashed when you use PARTITION BY HASH. To hash on only part of the primary key, and use a range partition on the rest, use the syntax demonstrated below.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
PARTITION BY HASH (id) PARTITIONS 4,
RANGE (sku)
(
  PARTITION VALUES < 'g',
  PARTITION 'g' <= VALUES < 'o',
  PARTITION 'o' <= VALUES < 'u',
  PARTITION 'u' <= VALUES
)
STORED AS KUDU;
```

Multiple PARTITION BY HASH Definitions

Once again expanding on the example above, let's assume that the pattern of incoming queries will be unpredictable, but you still want to ensure that writes are spread across a large number of tablets. You can achieve maximum distribution across the entire primary key by hashing on both primary key columns.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
```

```
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
PARTITION BY HASH (id) PARTITIONS 4,
             HASH (sku) PARTITIONS 4
STORED AS KUDU;
```

The example creates 16 partitions. You could also use HASH (id, sku) PARTITIONS 16. However, a scan for sku values would almost always impact all 16 partitions, rather than possibly being limited to 4.

Non-covering range partitions

Kudu supports the use of non-covering range partitions, which can be used to address the following scenarios:

- In the case of time-series data or other schemas which need to account for constantly-increasing primary keys, tablets serving old data will be relatively fixed in size, while tablets receiving new data will grow without bounds.
- In cases where you want to partition data based on its category, such as sales region or product type, without non-covering range partitions you must know all of the partitions ahead of time or manually recreate your table if partitions need to be added or removed, such as the introduction or elimination of a product type.

Note: See Range partitioning for the caveats of non-covering range partitions.

The following example creates a tablet per year (5 tablets total), for storing log data. The table only accepts data from 2012 to 2016. Keys outside of these ranges will be rejected.

```
CREATE TABLE sales_by_year (
  year INT, sale_id INT, amount INT,
  PRIMARY KEY (sale_id, year)
)
PARTITION BY RANGE (year) (
  PARTITION VALUE = 2012,
  PARTITION VALUE = 2013,
  PARTITION VALUE = 2014,
  PARTITION VALUE = 2015,
  PARTITION VALUE = 2016
)
STORED AS KUDU;
```

When records start coming in for 2017, they will be rejected. At that point, the 2017 range should be added as follows:

```
ALTER TABLE sales_by_year ADD RANGE PARTITION VALUE = 2017;
```

In use cases where a rolling window of data retention is required, range partitions may also be dropped. For example, if data from 2012 should no longer be retained, it may be deleted in bulk:

```
ALTER TABLE sales_by_year DROP RANGE PARTITION VALUE = 2012;
```

Note that just like dropping a table, this irrecoverably deletes all data stored in the dropped partition.

Partitioning guidelines

Here are some partitioning guidelines for small and large tables.

- For large tables, such as fact tables, aim for as many tablets as you have cores in the cluster.
- For small tables, such as dimension tables, ensure that each tablet is at least 1 GB in size.

Optimizing performance for evaluating SQL predicates

If the WHERE clause of your query includes comparisons with the operators =, <=, <, >, >=, BETWEEN, or IN, Kudu evaluates the condition directly and only returns the relevant results. This provides optimum performance, because Kudu only returns the relevant results to Impala.

For predicates such as !=, LIKE, or any other predicate type supported by Impala, Kudu does not evaluate the predicates directly. Instead, it returns all results to Impala and relies on Impala to evaluate the remaining predicates and filter the results accordingly. This may cause differences in performance, depending on the delta of the result set before and after evaluating the WHERE clause. In some cases, creating and periodically updating materialized views may be the right solution to work around these inefficiencies.

Inserting a row

The syntax for inserting one or more rows using Impala is shown below:

```
INSERT INTO my_first_table VALUES (99, "sarah");
INSERT INTO my_first_table VALUES (1, "john"), (2, "jane"), (3, "jim");
```

The primary key must not be null.

Inserting in bulk

When inserting in bulk, there are at least three common choices. Each may have advantages and disadvantages, depending on your data and circumstances.

Multiple single INSERT statements

This approach has the advantage of being easy to understand and implement. This approach is likely to be inefficient because Impala has a high query start-up cost compared to Kudu's insertion performance. This will lead to relatively high latency and poor throughput.

Single INSERT statement with multiple VALUES subclauses

If you include more than 1024 VALUES statements, Impala batches them into groups of 1024 (or the value of batch_size) before sending the requests to Kudu. This approach may perform slightly better than multiple sequential INSERT statements by amortizing the query start-up penalties on the Impala side. To set the batch size for the current Impala Shell session, use the following syntax:

```
set batch_size=10000;
```

Note: Increasing the Impala batch size causes Impala to use more memory. You should verify the impact on your cluster and tune accordingly.

Batch insert

The approach that usually performs best, from the standpoint of both Impala and Kudu, is usually to import the data using a SELECT FROM subclause in Impala.

1. If your data is not already in Impala, one strategy is to import it from a text file, such as a TSV or CSV file.
2. Create the Kudu table, being mindful that the columns designated as primary keys cannot have null values.
3. Insert values into the Kudu table by querying the table containing the original data, as in the following example:

```
INSERT INTO my_kudu_table
SELECT * FROM legacy_data_import_table;
```

Ingest using the C++ or Java API

In many cases, the appropriate ingest path is to use the C++ or Java API to insert directly into Kudu tables. Unlike other Impala tables, data inserted into Kudu tables using the API becomes available for query in Impala without the need for any INVALIDATE METADATA statements or other statements needed for other Impala storage types.

Related Information

Using Text Data Files

Creating a new Kudu table from Impala

INSERT and primary key uniqueness violations

In many relational databases, if you try to insert a row that has already been inserted, the operation can fail because the primary key gets duplicated. Impala, however, does not fail the query. Instead, it generates a warning and continues to execute the remainder of the insert statement.

If you want to replace existing rows from the table, use the UPSERT statement instead.

```
INSERT INTO my_first_table VALUES (99, "sarah");
UPSERT INTO my_first_table VALUES (99, "zoe");
```

The current value of the row is now zoe.

Updating a row

The syntax for updating one or more rows using Impala is shown below:

```
UPDATE my_first_table SET name="bob" where id = 3;
```

You cannot change or null the primary key value.

Important: The UPDATE statement only works in Impala when the underlying data source is Kudu.

Updating in bulk

You can update in bulk using the same approaches outlined in the *Insert in bulk* topic.

Related Information

Inserting in bulk

Upserting a row

The UPSERT command acts as a combination of the INSERT and UPDATE statements.

For each row processed by the UPSERT statement:

- If another row already exists with the same set of primary key values, the other columns are updated to match the values from the row being 'UPSERTed'.
- If there is no row with the same set of primary key values, the row is created, the same as if the INSERT statement was used.

UPSERT Example

The following example demonstrates how the UPSERT statement works. We start by creating two tables, foo1 and foo2.

```
CREATE TABLE foo1 (
 id INT PRIMARY KEY,
 col1 STRING,
 col2 STRING
)
PARTITION BY HASH(id) PARTITIONS 3
STORED AS KUDU;
```

```
CREATE TABLE foo2 (
 id INT PRIMARY KEY,
 col1 STRING,
 col2 STRING
)
PARTITION BY HASH(id) PARTITIONS 3
STORED AS KUDU;
```

Populate foo1 and foo2 using the following INSERT statements. For foo2, we leave column col2 with NULL values to be upserted later:

```
INSERT INTO foo1 VALUES (1, "hi", "alice");
```

```
INSERT INTO foo2 select id, col1, NULL from foo1;
```

The contents of foo2 will be:

```
SELECT * FROM foo2;
...
+----+------+------+
| id | col1 | col2 |
+----+------+------+
| 1  | hi   | NULL |
+----+------+------+
Fetched 1 row(s) in 0.15s
```

Now use the UPSERT command to now replace the NULL values in foo2 with the actual values from foo1.

```
UPSERT INTO foo2 (id, col2) select id, col2 from foo1;
```

```
SELECT * FROM foo2;
...
+----+------+-------+
| id | col1 | col2  |
+----+------+-------+
| 1  | hi   | alice |
+----+------+-------+
Fetched 1 row(s) in 0.15s
```

Altering a table

You can the ALTER TABLE statement to change the default value, encoding, compression, or block size of existing columns in a Kudu table.

The Impala SQL Reference ALTER TABLE includes a Kudu Considerations section with examples and a list of constraints relevant to altering a Kudu table in Impala.

Deleting a row

You can delete Kudu rows in near real time using Impala.

```
DELETE FROM my_first_table WHERE id < 3;
```

You can even use more complex joins when deleting rows. For example, Impala uses a comma in the FROM sub-clause to specify a join query.

```
DELETE c FROM my_second_table c, stock_symbols s WHERE c.name = s.symbol;
```

Important: The DELETE statement only works in Impala when the underlying data source is Kudu.

Deleting in bulk

You can delete in bulk using the same approaches outlined in the *Inserting in bulk* topic.

Related Information

Inserting in bulk

Failures during INSERT, UPDATE, UPSERT, and DELETE operations

INSERT, UPDATE, and DELETE statements cannot be considered transactional as a whole. If one of these operations fails part of the way through, the keys may have already been created (in the case of INSERT) or the records may have already been modified or removed by another process (in the case of UPDATE or DELETE). You should design your application with this in mind.

Altering table properties

You can change Impala's metadata relating to a given Kudu table by altering the table's properties. These properties include the table name, the list of Kudu master addresses, and whether the table is managed by Impala (internal) or externally. You cannot modify a table's split rows after table creation.

Important: Altering table properties only changes Impala's metadata about the table, not the underlying table itself. These statements do not modify any Kudu data.

Rename an Impala mapping table:

```
ALTER TABLE my_table RENAME TO my_new_table;
```

Renaming a table using the ALTER TABLE ... RENAME statement only renames the Impala mapping table, regardless of whether the table is an internal or external table. This avoids disruption to other applications that may be accessing the underlying Kudu table.

Rename the underlying Kudu table for an internal table:

In CDH 5.14 and lower, if a table is an internal table, the underlying Kudu table may be renamed by changing the kudu.table_name property:

```
ALTER TABLE my_internal_table
SET TBLPROPERTIES('kudu.table_name' = 'new_name')
```

Remapping an external table to a different Kudu table:

If another application has renamed a Kudu table under Impala, it is possible to re-map an external table to point to a different Kudu table name.

```
ALTER TABLE my_external_table_
SET TBLPROPERTIES('kudu.table_name' = 'some_other_kudu_table')
```

Change the Kudu master addresses:

```
ALTER TABLE my_table SET TBLPROPERTIES('kudu.master_addresses' = 'kudu-origi
nal-master.example.com:7051,kudu-new-master.example.com:7051');
```

Change an internally-managed table to external:

```
ALTER TABLE my_table SET TBLPROPERTIES('EXTERNAL' = 'TRUE');
```

Dropping a Kudu table using Impala

If the table was created as an internal table in Impala, using CREATE TABLE, the standard DROP TABLE syntax drops the underlying Kudu table and all its data. If the table was created as an external table, using CREATE EXTERNAL TABLE, the mapping between Impala and Kudu is dropped, but the Kudu table is left intact, with all its data. To change an external table to internal, or vice versa, see Altering table properties on page 14.

```
DROP TABLE my_first_table;
```

Security considerations

Kudu 1.3 (and higher) includes security features that allow Kudu clusters to be hardened against access from unauthorized users. Kudu uses strong authentication with Kerberos, while communication between Kudu clients and servers can now be encrypted with TLS. Kudu also allows you to use HTTPS encryption to connect to the web UI. These features should work seamlessly in Impala as long as Impala's user is given permission to access Kudu.

For instructions on how to configure a secure Kudu cluster, see *Kudu Security Overview*.

Related Information

Kudu security

Known issues and limitations

Here are the limitations when integrating Kudu with Impala, and list of Impala keywords that are not supported for creating Kudu tables.

- When creating a Kudu table, the CREATE TABLE statement must include the primary key columns before other columns, in primary key order.
- Impala cannot update values in primary key columns.
- Impala cannot create Kudu tables with VARCHAR or nested-typed columns.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when used as an external table in Impala.

- Kudu tables with a column name containing upper case or non-ASCII characters cannot be used as an external table in Impala. Columns can be renamed in Kudu to work around this issue.
- != and LIKE predicates are not pushed to Kudu, and instead will be evaluated by the Impala scan node. This may decrease performance relative to other types of predicates.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or use large tables.

Impala keywords not supported for creating Kudu tables

- PARTITIONED
- LOCATION
- ROWFORMAT

Next steps

The examples above have only explored a fraction of what you can do with Impala Shell. You can explore more by visiting the resources listed in this topic.

Related Information

Impala project

Impala Wiki

Using Impala to query Kudu tables