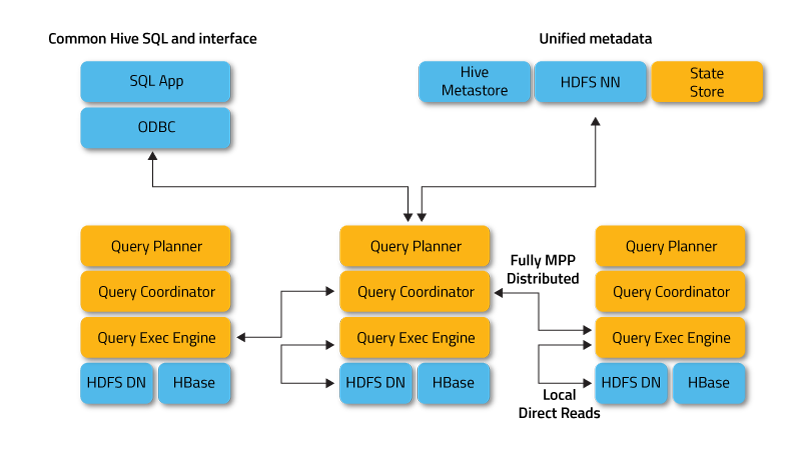
**Overview**

Impala raises the bar for SQL query performance on Apache Hadoop while retaining a familiar user experience. With Impala, you can query data, whether stored in HDFS or Apache HBase – including SELECT, JOIN, and aggregate functions – in real time. Furthermore, Impala uses the same metadata, SQL syntax (Hive SQL), ODBC driver, and user interface (Hue Beeswax) as Apache Hive, providing a familiar and unified platform for batch-oriented or real-time queries. (For that reason, Hive users can utilize Impala with little setup overhead.)

**Architecture**

To avoid latency, Impala circumvents MapReduce to directly access the data through a specialized distributed query engine that is very similar to those found in commercial parallel RDBMSs. The result is order-of-magnitude faster performance than Hive, depending on the type of query and configuration.



There are many advantages to this approach over alternative approaches for querying Hadoop data, including::

* Thanks to local processing on data nodes, network bottlenecks are avoided.
* A single, open, and unified metadata store can be utilized.
* Costly data format conversion is unnecessary and thus no overhead is incurred.
* All data is immediately query-able, with no delays for ETL.
* All hardware is utilized for Impala queries as well as for MapReduce.
* Only a single machine pool is needed to scale.

We encourage you to read [this paper](http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf) for details about Impala's architecture.

# Impala Tutorials

This section includes tutorial scenarios that demonstrate how to begin using Impala once the software is installed. It focuses on techniques for loading data, because once you have some data in tables and can query that data, you can quickly progress to more advanced Impala features.

Note:

Where practical, the tutorials take you from "ground zero" to having the desired Impala tables and data. In some cases, you might need to download additional files from outside sources, set up additional software components, modify commands or scripts to fit your own configuration, or substitute your own sample data.

Before trying these tutorial lessons, install Impala using one of these procedures:

* If you already have some CDH environment set up and just need to add Impala to it, follow the installation process described in [Impala Installation](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_install.html#install). Make sure to also install the Hive metastore service if you do not already have Hive configured.
* To set up Impala and all its prerequisites at once, in a minimal configuration that you can use for small-scale experiments, set up the Cloudera QuickStart VM, which includes CDH and Impala on CentOS. Use this single-node VM to try out basic SQL functionality, not anything related to performance and scalability. For more information, see [the Cloudera QuickStart VM](http://www.cloudera.com/content/support/en/downloads/quickstart_vms.html).

Continue reading:

* [Tutorials for Getting Started](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tut_beginner)
* [Advanced Tutorials](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tut_advanced)
* [Dealing with Parquet Files with Unknown Schema](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tut_parquet_schemaless)

## Tutorials for Getting Started

These tutorials demonstrate the basics of using Impala. They are intended for first-time users, and for trying out Impala on any new cluster to make sure the major components are working correctly.

Continue reading:

* [Explore a New Impala Instance](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tutorial_explore)
* [Load CSV Data from Local Files](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tutorial_csv_setup)
* [Point an Impala Table at Existing Data Files](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tutorial_create_table)
* [Describe the Impala Table](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tutorial_describe_impala)
* [Query the Impala Table](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tutorial_query_impala)
* [Data Loading and Querying Examples](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tutorial_etl)

### Explore a New Impala Instance

This tutorial demonstrates techniques for finding your way around the tables and databases of an unfamiliar (possibly empty) Impala instance.

When you connect to an Impala instance for the first time, you use the SHOW DATABASES and SHOW TABLES statements to view the most common types of objects. Also, call the version() function to confirm which version of Impala you are running; the version number is important when consulting documentation and dealing with support issues.

A completely empty Impala instance contains no tables, but still has two databases:

* default, where new tables are created when you do not specify any other database.
* \_impala\_builtins, a system database used to hold all the built-in functions.

The following example shows how to see the available databases, and the tables in each. If the list of databases or tables is long, you can use wildcard notation to locate specific databases or tables based on their names.

$ impala-shell -i localhost --quiet

Starting Impala Shell without Kerberos authentication

Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v...

[localhost:21000] > select version();

+-------------------------------------------

| version()

+-------------------------------------------

| impalad version ...

| Built on ...

+-------------------------------------------

[localhost:21000] > show databases;

+--------------------------+

| name |

+--------------------------+

| \_impala\_builtins |

| ctas |

| d1 |

| d2 |

| d3 |

| default |

| explain\_plans |

| external\_table |

| file\_formats |

| tpc |

+--------------------------+

[localhost:21000] > select current\_database();

+--------------------+

| current\_database() |

+--------------------+

| default |

+--------------------+

[localhost:21000] > show tables;

+-------+

| name |

+-------+

| ex\_t |

| t1 |

+-------+

[localhost:21000] > show tables in d3;

[localhost:21000] > show tables in tpc;

+------------------------+

| name |

+------------------------+

| city |

| customer |

| customer\_address |

| customer\_demographics |

| household\_demographics |

| item |

| promotion |

| store |

| store2 |

| store\_sales |

| ticket\_view |

| time\_dim |

| tpc\_tables |

+------------------------+

[localhost:21000] > show tables in tpc like 'customer\*';

+-----------------------+

| name |

+-----------------------+

| customer |

| customer\_address |

| customer\_demographics |

+-----------------------+

Once you know what tables and databases are available, you descend into a database with the USE statement. To understand the structure of each table, you use the DESCRIBE command. Once inside a database, you can issue statements such as INSERT and SELECT that operate on particular tables.

The following example explores a database named TPC whose name we learned in the previous example. It shows how to filter the table names within a database based on a search string, examine the columns of a table, and run queries to examine the characteristics of the table data. For example, for an unfamiliar table you might want to know the number of rows, the number of different values for a column, and other properties such as whether the column contains any NULL values. When sampling the actual data values from a table, use a LIMIT clause to avoid excessive output if the table contains more rows or distinct values than you expect.

[localhost:21000] > use tpc;

[localhost:21000] > show tables like '\*view\*';

+-------------+

| name |

+-------------+

| ticket\_view |

+-------------+

[localhost:21000] > describe city;

+-------------+--------+---------+

| name | type | comment |

+-------------+--------+---------+

| id | int | |

| name | string | |

| countrycode | string | |

| district | string | |

| population | int | |

+-------------+--------+---------+

[localhost:21000] > select count(\*) from city;

+----------+

| count(\*) |

+----------+

| 0 |

+----------+

[localhost:21000] > desc customer;

+------------------------+--------+---------+

| name | type | comment |

+------------------------+--------+---------+

| c\_customer\_sk | int | |

| c\_customer\_id | string | |

| c\_current\_cdemo\_sk | int | |

| c\_current\_hdemo\_sk | int | |

| c\_current\_addr\_sk | int | |

| c\_first\_shipto\_date\_sk | int | |

| c\_first\_sales\_date\_sk | int | |

| c\_salutation | string | |

| c\_first\_name | string | |

| c\_last\_name | string | |

| c\_preferred\_cust\_flag | string | |

| c\_birth\_day | int | |

| c\_birth\_month | int | |

| c\_birth\_year | int | |

| c\_birth\_country | string | |

| c\_login | string | |

| c\_email\_address | string | |

| c\_last\_review\_date | string | |

+------------------------+--------+---------+

[localhost:21000] > select count(\*) from customer;

+----------+

| count(\*) |

+----------+

| 100000 |

+----------+

[localhost:21000] > select count(distinct c\_birth\_month) from customer;

+-------------------------------+

| count(distinct c\_birth\_month) |

+-------------------------------+

| 12 |

+-------------------------------+

[localhost:21000] > select count(\*) from customer where c\_email\_address is null;

+----------+

| count(\*) |

+----------+

| 0 |

+----------+

[localhost:21000] > select distinct c\_salutation from customer limit 10;

+--------------+

| c\_salutation |

+--------------+

| Mr. |

| Ms. |

| Dr. |

| |

| Miss |

| Sir |

| Mrs. |

+--------------+

When you graduate from read-only exploration, you use statements such as CREATE DATABASE and CREATE TABLE to set up your own database objects.

The following example demonstrates creating a new database holding a new table. Although the last example ended inside the TPC database, the new EXPERIMENTS database is not nested inside TPC; all databases are arranged in a single top-level list.

[localhost:21000] > create database experiments;

[localhost:21000] > show databases;

+--------------------------+

| name |

+--------------------------+

| \_impala\_builtins |

| ctas |

| d1 |

| d2 |

| d3 |

| default |

| experiments |

| explain\_plans |

| external\_table |

| file\_formats |

| tpc |

+--------------------------+

[localhost:21000] > show databases like 'exp\*';

+---------------+

| name |

+---------------+

| experiments |

| explain\_plans |

+---------------+

The following example creates a new table, T1. To illustrate a common mistake, it creates this table inside the wrong database, the TPC database where the previous example ended. The ALTER TABLE statement lets you move the table to the intended database, EXPERIMENTS, as part of a rename operation. The USE statement is always needed to switch to a new database, and the current\_database() function confirms which database the session is in, to avoid these kinds of mistakes.

[localhost:21000] > create table t1 (x int);

[localhost:21000] > show tables;

+------------------------+

| name |

+------------------------+

| city |

| customer |

| customer\_address |

| customer\_demographics |

| household\_demographics |

| item |

| promotion |

| store |

| store2 |

| store\_sales |

| t1 |

| ticket\_view |

| time\_dim |

| tpc\_tables |

+------------------------+

[localhost:21000] > select current\_database();

+--------------------+

| current\_database() |

+--------------------+

| tpc |

+--------------------+

[localhost:21000] > alter table t1 rename to experiments.t1;

[localhost:21000] > use experiments;

[localhost:21000] > show tables;

+------+

| name |

+------+

| t1 |

+------+

[localhost:21000] > select current\_database();

+--------------------+

| current\_database() |

+--------------------+

| experiments |

+--------------------+

For your initial experiments with tables, you can use ones with just a few columns and a few rows, and text-format data files.

Note: As you graduate to more realistic scenarios, you will use more elaborate tables with many columns, features such as partitioning, and file formats such as Parquet. When dealing with realistic data volumes, you will bring in data using LOAD DATA or INSERT ... SELECT statements to operate on millions or billions of rows at once.

The following example sets up a couple of simple tables with a few rows, and performs queries involving sorting, aggregate functions and joins.

[localhost:21000] > insert into t1 values (1), (3), (2), (4);

[localhost:21000] > select x from t1 order by x desc;

+---+

| x |

+---+

| 4 |

| 3 |

| 2 |

| 1 |

+---+

[localhost:21000] > select min(x), max(x), sum(x), avg(x) from t1;

+--------+--------+--------+--------+

| min(x) | max(x) | sum(x) | avg(x) |

+--------+--------+--------+--------+

| 1 | 4 | 10 | 2.5 |

+--------+--------+--------+--------+

[localhost:21000] > create table t2 (id int, word string);

[localhost:21000] > insert into t2 values (1, "one"), (3, "three"), (5, 'five');

[localhost:21000] > select word from t1 join t2 on (t1.x = t2.id);

+-------+

| word |

+-------+

| one |

| three |

+-------+

After completing this tutorial, you should now know:

* How to tell which version of Impala is running on your system.
* How to find the names of databases in an Impala instance, either displaying the full list or searching for specific names.
* How to find the names of tables in an Impala database, either displaying the full list or searching for specific names.
* How to switch between databases and check which database you are currently in.
* How to learn the column names and types of a table.
* How to create databases and tables, insert small amounts of test data, and run simple queries.

### Load CSV Data from Local Files

This scenario illustrates how to create some very small tables, suitable for first-time users to experiment with Impala SQL features. TAB1 and TAB2 are loaded with data from files in HDFS. A subset of data is copied from TAB1 into TAB3.

Populate HDFS with the data you want to query. To begin this process, create one or more new subdirectories underneath your user directory in HDFS. The data for each table resides in a separate subdirectory. Substitute your own username for username where appropriate. This example uses the -p option with the mkdir operation to create any necessary parent directories if they do not already exist.

$ whoami

username

$ hdfs dfs -ls /user

Found 3 items

drwxr-xr-x - username username 0 2013-04-22 18:54 /user/username

drwxrwx--- - mapred mapred 0 2013-03-15 20:11 /user/history

drwxr-xr-x - hue supergroup 0 2013-03-15 20:10 /user/hive

$ hdfs dfs -mkdir -p /user/username/sample\_data/tab1 /user/username/sample\_data/tab2

Here is some sample data, for two tables named TAB1 and TAB2.

Copy the following content to .csv files in your local filesystem:

tab1.csv:

1,true,123.123,2012-10-24 08:55:00

2,false,1243.5,2012-10-25 13:40:00

3,false,24453.325,2008-08-22 09:33:21.123

4,false,243423.325,2007-05-12 22:32:21.33454

5,true,243.325,1953-04-22 09:11:33

tab2.csv:

1,true,12789.123

2,false,1243.5

3,false,24453.325

4,false,2423.3254

5,true,243.325

60,false,243565423.325

70,true,243.325

80,false,243423.325

90,true,243.325

Put each .csv file into a separate HDFS directory using commands like the following, which use paths available in the Impala Demo VM:

$ hdfs dfs -put tab1.csv /user/username/sample\_data/tab1

$ hdfs dfs -ls /user/username/sample\_data/tab1

Found 1 items

-rw-r--r-- 1 username username 192 2013-04-02 20:08 /user/username/sample\_data/tab1/tab1.csv

$ hdfs dfs -put tab2.csv /user/username/sample\_data/tab2

$ hdfs dfs -ls /user/username/sample\_data/tab2

Found 1 items

-rw-r--r-- 1 username username 158 2013-04-02 20:09 /user/username/sample\_data/tab2/tab2.csv

The name of each data file is not significant. In fact, when Impala examines the contents of the data directory for the first time, it considers all files in the directory to make up the data of the table, regardless of how many files there are or what the files are named.

To understand what paths are available within your own HDFS filesystem and what the permissions are for the various directories and files, issue hdfs dfs -ls / and work your way down the tree doing -ls operations for the various directories.

Use the impala-shell command to create tables, either interactively or through a SQL script.

The following example shows creating three tables. For each table, the example shows creating columns with various attributes such as Boolean or integer types. The example also includes commands that provide information about how the data is formatted, such as rows terminating with commas, which makes sense in the case of importing data from a .csv file. Where we already have .csv files containing data in the HDFS directory tree, we specify the location of the directory containing the appropriate .csv file. Impala considers all the data from all the files in that directory to represent the data for the table.

DROP TABLE IF EXISTS tab1;

-- The EXTERNAL clause means the data is located outside the central location

-- for Impala data files and is preserved when the associated Impala table is dropped.

-- We expect the data to already exist in the directory specified by the LOCATION clause.

CREATE EXTERNAL TABLE tab1

(

id INT,

col\_1 BOOLEAN,

col\_2 DOUBLE,

col\_3 TIMESTAMP

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/user/username/sample\_data/tab1';

DROP TABLE IF EXISTS tab2;

-- TAB2 is an external table, similar to TAB1.

CREATE EXTERNAL TABLE tab2

(

id INT,

col\_1 BOOLEAN,

col\_2 DOUBLE

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/user/username/sample\_data/tab2';

DROP TABLE IF EXISTS tab3;

-- Leaving out the EXTERNAL clause means the data will be managed

-- in the central Impala data directory tree. Rather than reading

-- existing data files when the table is created, we load the

-- data after creating the table.

CREATE TABLE tab3

(

id INT,

col\_1 BOOLEAN,

col\_2 DOUBLE,

month INT,

day INT

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

Note: Getting through these CREATE TABLE statements successfully is an important validation step to confirm everything is configured correctly with the Hive metastore and HDFS permissions. If you receive any errors during the CREATE TABLE statements:

* Make sure you followed the installation instructions closely, in [Impala Installation](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_install.html#install).
* Make sure the hive.metastore.warehouse.dir property points to a directory that Impala can write to. The ownership should be hive:hive, and the impala user should also be a member of the hive group.
* If the value of hive.metastore.warehouse.dir is different in the Cloudera Manager dialogs and in the Hive shell, you might need to [designate the hosts running impalad with the "gateway" role for Hive](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_managing_roles.html), and [deploy the client configuration files](http://www.cloudera.com/documentation/enterprise/latest/topics/cm_mc_client_config.html) to those hosts.

### Point an Impala Table at Existing Data Files

A convenient way to set up data for Impala to access is to use an external table, where the data already exists in a set of HDFS files and you just point the Impala table at the directory containing those files. For example, you might run in impala-shell a \*.sql file with contents similar to the following, to create an Impala table that accesses an existing data file used by Hive.

The following examples set up 2 tables, referencing the paths and sample data from the sample TPC-DS kit for Impala. For historical reasons, the data physically resides in an HDFS directory tree under /user/hive, although this particular data is entirely managed by Impala rather than Hive. When we create an external table, we specify the directory containing one or more data files, and Impala queries the combined content of all the files inside that directory. Here is how we examine the directories and files within the HDFS filesystem:

$ cd ~/username/datasets

$ ./tpcds-setup.sh

... Downloads and unzips the kit, builds the data and loads it into HDFS ...

$ hdfs dfs -ls /user/hive/tpcds/customer

Found 1 items

-rw-r--r-- 1 username supergroup 13209372 2013-03-22 18:09 /user/hive/tpcds/customer/customer.dat

$ hdfs dfs -cat /user/hive/tpcds/customer/customer.dat | more

1|AAAAAAAABAAAAAAA|980124|7135|32946|2452238|2452208|Mr.|Javier|Lewis|Y|9|12|1936|CHILE||Javie

r.Lewis@VFAxlnZEvOx.org|2452508|

2|AAAAAAAACAAAAAAA|819667|1461|31655|2452318|2452288|Dr.|Amy|Moses|Y|9|4|1966|TOGO||Amy.Moses@

Ovk9KjHH.com|2452318|

3|AAAAAAAADAAAAAAA|1473522|6247|48572|2449130|2449100|Miss|Latisha|Hamilton|N|18|9|1979|NIUE||

Latisha.Hamilton@V.com|2452313|

4|AAAAAAAAEAAAAAAA|1703214|3986|39558|2450030|2450000|Dr.|Michael|White|N|7|6|1983|MEXICO||Mic

hael.White@i.org|2452361|

5|AAAAAAAAFAAAAAAA|953372|4470|36368|2449438|2449408|Sir|Robert|Moran|N|8|5|1956|FIJI||Robert.

Moran@Hh.edu|2452469|

...

Here is a SQL script to set up Impala tables pointing to some of these data files in HDFS. (The script in the VM sets up tables like this through Hive; ignore those tables for purposes of this demonstration.) Save the following as customer\_setup.sql:

--

-- store\_sales fact table and surrounding dimension tables only

--

create database tpcds;

use tpcds;

drop table if exists customer;

create external table customer

(

c\_customer\_sk int,

c\_customer\_id string,

c\_current\_cdemo\_sk int,

c\_current\_hdemo\_sk int,

c\_current\_addr\_sk int,

c\_first\_shipto\_date\_sk int,

c\_first\_sales\_date\_sk int,

c\_salutation string,

c\_first\_name string,

c\_last\_name string,

c\_preferred\_cust\_flag string,

c\_birth\_day int,

c\_birth\_month int,

c\_birth\_year int,

c\_birth\_country string,

c\_login string,

c\_email\_address string,

c\_last\_review\_date string

)

row format delimited fields terminated by '|'

location '/user/hive/tpcds/customer';

drop table if exists customer\_address;

create external table customer\_address

(

ca\_address\_sk int,

ca\_address\_id string,

ca\_street\_number string,

ca\_street\_name string,

ca\_street\_type string,

ca\_suite\_number string,

ca\_city string,

ca\_county string,

ca\_state string,

ca\_zip string,

ca\_country string,

ca\_gmt\_offset float,

ca\_location\_type string

)

row format delimited fields terminated by '|'

location '/user/hive/tpcds/customer\_address';

We would run this script with a command such as:

impala-shell -i localhost -f customer\_setup.sql

### Describe the Impala Table

Now that you have updated the database metadata that Impala caches, you can confirm that the expected tables are accessible by Impala and examine the attributes of one of the tables. We created these tables in the database named default. If the tables were in a database other than the default, we would issue a command use db\_name to switch to that database before examining or querying its tables. We could also qualify the name of a table by prepending the database name, for example default.customer and default.customer\_name.

[impala-host:21000] > show databases

Query finished, fetching results ...

default

Returned 1 row(s) in 0.00s

[impala-host:21000] > show tables

Query finished, fetching results ...

customer

customer\_address

Returned 2 row(s) in 0.00s

[impala-host:21000] > describe customer\_address

+------------------+--------+---------+

| name | type | comment |

+------------------+--------+---------+

| ca\_address\_sk | int | |

| ca\_address\_id | string | |

| ca\_street\_number | string | |

| ca\_street\_name | string | |

| ca\_street\_type | string | |

| ca\_suite\_number | string | |

| ca\_city | string | |

| ca\_county | string | |

| ca\_state | string | |

| ca\_zip | string | |

| ca\_country | string | |

| ca\_gmt\_offset | float | |

| ca\_location\_type | string | |

+------------------+--------+---------+

Returned 13 row(s) in 0.01

### Query the Impala Table

You can query data contained in the tables. Impala coordinates the query execution across a single node or multiple nodes depending on your configuration, without the overhead of running MapReduce jobs to perform the intermediate processing.

There are a variety of ways to execute queries on Impala:

* Using the impala-shell command in interactive mode:
* $ impala-shell -i impala-host
* Connected to localhost:21000
* [impala-host:21000] > select count(\*) from customer\_address;
* 50000
* Returned 1 row(s) in 0.37s
* Passing a set of commands contained in a file:
* $ impala-shell -i impala-host -f myquery.sql
* Connected to localhost:21000
* 50000

Returned 1 row(s) in 0.19s

* Passing a single command to the impala-shell command. The query is executed, the results are returned, and the shell exits. Make sure to quote the command, preferably with single quotation marks to avoid shell expansion of characters such as \*.
* $ impala-shell -i impala-host -q 'select count(\*) from customer\_address'
* Connected to localhost:21000
* 50000

Returned 1 row(s) in 0.29s

### Data Loading and Querying Examples

This section describes how to create some sample tables and load data into them. These tables can then be queried using the Impala shell.

#### Loading Data

Loading data involves:

* Establishing a data set. The example below uses .csv files.
* Creating tables to which to load data.
* Loading the data into the tables you created.

#### Sample Queries

To run these sample queries, create a SQL query file query.sql, copy and paste each query into the query file, and then run the query file using the shell. For example, to run query.sql on impala-host, you might use the command:

impala-shell.sh -i impala-host -f query.sql

The examples and results below assume you have loaded the sample data into the tables as described above.

##### Example: Examining Contents of Tables

Let's start by verifying that the tables do contain the data we expect. Because Impala often deals with tables containing millions or billions of rows, when examining tables of unknown size, include the LIMIT clause to avoid huge amounts of unnecessary output, as in the final query. (If your interactive query starts displaying an unexpected volume of data, press Ctrl-C in impala-shell to cancel the query.)

SELECT \* FROM tab1;

SELECT \* FROM tab2;

SELECT \* FROM tab2 LIMIT 5;

Results:

+----+-------+------------+-------------------------------+

| id | col\_1 | col\_2 | col\_3 |

+----+-------+------------+-------------------------------+

| 1 | true | 123.123 | 2012-10-24 08:55:00 |

| 2 | false | 1243.5 | 2012-10-25 13:40:00 |

| 3 | false | 24453.325 | 2008-08-22 09:33:21.123000000 |

| 4 | false | 243423.325 | 2007-05-12 22:32:21.334540000 |

| 5 | true | 243.325 | 1953-04-22 09:11:33 |

+----+-------+------------+-------------------------------+

+----+-------+---------------+

| id | col\_1 | col\_2 |

+----+-------+---------------+

| 1 | true | 12789.123 |

| 2 | false | 1243.5 |

| 3 | false | 24453.325 |

| 4 | false | 2423.3254 |

| 5 | true | 243.325 |

| 60 | false | 243565423.325 |

| 70 | true | 243.325 |

| 80 | false | 243423.325 |

| 90 | true | 243.325 |

+----+-------+---------------+

+----+-------+-----------+

| id | col\_1 | col\_2 |

+----+-------+-----------+

| 1 | true | 12789.123 |

| 2 | false | 1243.5 |

| 3 | false | 24453.325 |

| 4 | false | 2423.3254 |

| 5 | true | 243.325 |

+----+-------+-----------+

##### Example: Aggregate and Join

SELECT tab1.col\_1, MAX(tab2.col\_2), MIN(tab2.col\_2)

FROM tab2 JOIN tab1 USING (id)

GROUP BY col\_1 ORDER BY 1 LIMIT 5;

Results:

+-------+-----------------+-----------------+

| col\_1 | max(tab2.col\_2) | min(tab2.col\_2) |

+-------+-----------------+-----------------+

| false | 24453.325 | 1243.5 |

| true | 12789.123 | 243.325 |

+-------+-----------------+-----------------+

##### Example: Subquery, Aggregate and Joins

SELECT tab2.\*

FROM tab2,

(SELECT tab1.col\_1, MAX(tab2.col\_2) AS max\_col2

FROM tab2, tab1

WHERE tab1.id = tab2.id

GROUP BY col\_1) subquery1

WHERE subquery1.max\_col2 = tab2.col\_2;

Results:

+----+-------+-----------+

| id | col\_1 | col\_2 |

+----+-------+-----------+

| 1 | true | 12789.123 |

| 3 | false | 24453.325 |

+----+-------+-----------+

##### Example: INSERT Query

INSERT OVERWRITE TABLE tab3

SELECT id, col\_1, col\_2, MONTH(col\_3), DAYOFMONTH(col\_3)

FROM tab1 WHERE YEAR(col\_3) = 2012;

Query TAB3 to check the result:

SELECT \* FROM tab3;

Results:

+----+-------+---------+-------+-----+

| id | col\_1 | col\_2 | month | day |

+----+-------+---------+-------+-----+

| 1 | true | 123.123 | 10 | 24 |

| 2 | false | 1243.5 | 10 | 25 |

+----+-------+---------+-------+-----+

## Advanced Tutorials

These tutorials walk you through advanced scenarios or specialized features.

Continue reading:

* [Attaching an External Partitioned Table to an HDFS Directory Structure](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tut_external_partition_data)
* [Switching Back and Forth Between Impala and Hive](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tutorial_impala_hive)
* [Cross Joins and Cartesian Products with the CROSS JOIN Operator](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_tutorial.html#tut_cross_join)

### Attaching an External Partitioned Table to an HDFS Directory Structure

This tutorial shows how you might set up a directory tree in HDFS, put data files into the lowest-level subdirectories, and then use an Impala external table to query the data files from their original locations.

The tutorial uses a table with web log data, with separate subdirectories for the year, month, day, and host. For simplicity, we use a tiny amount of CSV data, loading the same data into each partition.

First, we make an Impala partitioned table for CSV data, and look at the underlying HDFS directory structure to understand the directory structure to re-create elsewhere in HDFS. The columns field1, field2, and field3 correspond to the contents of the CSV data files. The year, month, day, and host columns are all represented as subdirectories within the table structure, and are not part of the CSV files. We use STRING for each of these columns so that we can produce consistent subdirectory names, with leading zeros for a consistent length.

create database external\_partitions;

use external\_partitions;

create table logs (field1 string, field2 string, field3 string)

partitioned by (year string, month string , day string, host string)

row format delimited fields terminated by ',';

insert into logs partition (year="2013", month="07", day="28", host="host1") values ("foo","foo","foo");

insert into logs partition (year="2013", month="07", day="28", host="host2") values ("foo","foo","foo");

insert into logs partition (year="2013", month="07", day="29", host="host1") values ("foo","foo","foo");

insert into logs partition (year="2013", month="07", day="29", host="host2") values ("foo","foo","foo");

insert into logs partition (year="2013", month="08", day="01", host="host1") values ("foo","foo","foo");

Back in the Linux shell, we examine the HDFS directory structure. (Your Impala data directory might be in a different location; for historical reasons, it is sometimes under the HDFS path /user/hive/warehouse.) We use the hdfs dfs -ls command to examine the nested subdirectories corresponding to each partitioning column, with separate subdirectories at each level (with = in their names) representing the different values for each partitioning column. When we get to the lowest level of subdirectory, we use the hdfs dfs -cat command to examine the data file and see CSV-formatted data produced by the INSERT statement in Impala.

$ hdfs dfs -ls /user/impala/warehouse/external\_partitions.db

Found 1 items

drwxrwxrwt - impala hive 0 2013-08-07 12:24 /user/impala/warehouse/external\_partitions.db/logs

$ hdfs dfs -ls /user/impala/warehouse/external\_partitions.db/logs

Found 1 items

drwxr-xr-x - impala hive 0 2013-08-07 12:24 /user/impala/warehouse/external\_partitions.db/logs/year=2013

$ hdfs dfs -ls /user/impala/warehouse/external\_partitions.db/logs/year=2013

Found 2 items

drwxr-xr-x - impala hive 0 2013-08-07 12:23 /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07

drwxr-xr-x - impala hive 0 2013-08-07 12:24 /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=08

$ hdfs dfs -ls /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07

Found 2 items

drwxr-xr-x - impala hive 0 2013-08-07 12:22 /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07/day=28

drwxr-xr-x - impala hive 0 2013-08-07 12:23 /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07/day=29

$ hdfs dfs -ls /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07/day=28

Found 2 items

drwxr-xr-x - impala hive 0 2013-08-07 12:21 /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07/day=28/host=host1

drwxr-xr-x - impala hive 0 2013-08-07 12:22 /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07/day=28/host=host2

$ hdfs dfs -ls /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07/day=28/host=host1

Found 1 items

-rw-r--r-- 3 impala hive 12 2013-08-07 12:21 /user/impala/warehouse/external\_partiti

ons.db/logs/year=2013/month=07/day=28/host=host1/3981726974111751120--8907184999369517436\_822630111\_data.0

$ hdfs dfs -cat /user/impala/warehouse/external\_partitions.db/logs/year=2013/month=07/day=28/\

host=host1/3981726974111751120--8 907184999369517436\_822630111\_data.0

foo,foo,foo

Still in the Linux shell, we use hdfs dfs -mkdir to create several data directories outside the HDFS directory tree that Impala controls (/user/impala/warehouse in this example, maybe different in your case). Depending on your configuration, you might need to log in as a user with permission to write into this HDFS directory tree; for example, the commands shown here were run while logged in as the hdfs user.

$ hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host1

$ hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host2

$ hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host1

$ hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=29/host=host1

$ hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=08/day=01/host=host1

We make a tiny CSV file, with values different than in the INSERT statements used earlier, and put a copy within each subdirectory that we will use as an Impala partition.

$ cat >dummy\_log\_data

bar,baz,bletch

$ hdfs dfs -mkdir -p /user/impala/data/external\_partitions/year=2013/month=08/day=01/host=host1

$ hdfs dfs -mkdir -p /user/impala/data/external\_partitions/year=2013/month=07/day=28/host=host1

$ hdfs dfs -mkdir -p /user/impala/data/external\_partitions/year=2013/month=07/day=28/host=host2

$ hdfs dfs -mkdir -p /user/impala/data/external\_partitions/year=2013/month=07/day=29/host=host1

$ hdfs dfs -put dummy\_log\_data /user/impala/data/logs/year=2013/month=07/day=28/host=host1

$ hdfs dfs -put dummy\_log\_data /user/impala/data/logs/year=2013/month=07/day=28/host=host2

$ hdfs dfs -put dummy\_log\_data /user/impala/data/logs/year=2013/month=07/day=29/host=host1

$ hdfs dfs -put dummy\_log\_data /user/impala/data/logs/year=2013/month=08/day=01/host=host1

Back in the impala-shell interpreter, we move the original Impala-managed table aside, and create a new external table with a LOCATION clause pointing to the directory under which we have set up all the partition subdirectories and data files.

use external\_partitions;

alter table logs rename to logs\_original;

create external table logs (field1 string, field2 string, field3 string)

partitioned by (year string, month string, day string, host string)

row format delimited fields terminated by ','

location '/user/impala/data/logs';

Because partition subdirectories and data files come and go during the data lifecycle, you must identify each of the partitions through an ALTER TABLE statement before Impala recognizes the data files they contain.

alter table logs add partition (year="2013",month="07",day="28",host="host1")

alter table log\_type add partition (year="2013",month="07",day="28",host="host2");

alter table log\_type add partition (year="2013",month="07",day="29",host="host1");

alter table log\_type add partition (year="2013",month="08",day="01",host="host1");

We issue a REFRESH statement for the table, always a safe practice when data files have been manually added, removed, or changed. Then the data is ready to be queried. The SELECT \* statement illustrates that the data from our trivial CSV file was recognized in each of the partitions where we copied it. Although in this case there are only a few rows, we include a LIMIT clause on this test query just in case there is more data than we expect.

refresh log\_type;

select \* from log\_type limit 100;

+--------+--------+--------+------+-------+-----+-------+

| field1 | field2 | field3 | year | month | day | host |

+--------+--------+--------+------+-------+-----+-------+

| bar | baz | bletch | 2013 | 07 | 28 | host1 |

| bar | baz | bletch | 2013 | 08 | 01 | host1 |

| bar | baz | bletch | 2013 | 07 | 29 | host1 |

| bar | baz | bletch | 2013 | 07 | 28 | host2 |

+--------+--------+--------+------+-------+-----+-------+

### Switching Back and Forth Between Impala and Hive

Sometimes, you might find it convenient to switch to the Hive shell to perform some data loading or transformation operation, particularly on file formats such as RCFile, SequenceFile, and Avro that Impala currently can query but not write to.

Whenever you create, drop, or alter a table or other kind of object through Hive, the next time you switch back to the impala-shell interpreter, issue a one-time INVALIDATE METADATA statement so that Impala recognizes the new or changed object.

Whenever you load, insert, or change data in an existing table through Hive (or even through manual HDFS operations such as the hdfs command), the next time you switch back to the impala-shell interpreter, issue a one-time REFRESH table\_name statement so that Impala recognizes the new or changed data.

For examples showing how this process works for the REFRESH statement, look at the examples of creating RCFile and SequenceFile tables in Impala, loading data through Hive, and then querying the data through Impala. See [Using the RCFile File Format with Impala Tables](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_rcfile.html#rcfile) and [Using the SequenceFile File Format with Impala Tables](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_seqfile.html#seqfile) for those examples.

For examples showing how this process works for the INVALIDATE METADATA statement, look at the example of creating and loading an Avro table in Hive, and then querying the data through Impala. See [Using the Avro File Format with Impala Tables](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_avro.html#avro) for that example.

Note:

Originally, Impala did not support UDFs, but this feature is available in Impala starting in Impala 1.2. Some INSERT ... SELECT transformations that you originally did through Hive can now be done through Impala. See [Impala User-Defined Functions (UDFs)](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_udf.html#udfs) for details.

Prior to Impala 1.2, the REFRESH and INVALIDATE METADATA statements needed to be issued on each Impala node to which you connected and issued queries. In Impala 1.2 and higher, when you issue either of those statements on any Impala node, the results are broadcast to all the Impala nodes in the cluster, making it truly a one-step operation after each round of DDL or ETL operations in Hive.

### Cross Joins and Cartesian Products with the CROSS JOIN Operator

Originally, Impala restricted join queries so that they had to include at least one equality comparison between the columns of the tables on each side of the join operator. With the huge tables typically processed by Impala, any miscoded query that produced a full Cartesian product as a result set could consume a huge amount of cluster resources.

In Impala 1.2.2 and higher, this restriction is lifted when you use the CROSS JOIN operator in the query. You still cannot remove all WHERE clauses from a query like SELECT \* FROM t1 JOIN t2 to produce all combinations of rows from both tables. But you can use the CROSS JOIN operator to explicitly request such a Cartesian product. Typically, this operation is applicable for smaller tables, where the result set still fits within the memory of a single Impala node.

The following example sets up data for use in a series of comic books where characters battle each other. At first, we use an equijoin query, which only allows characters from the same time period and the same planet to meet.

[localhost:21000] > create table heroes (name string, era string, planet string);

[localhost:21000] > create table villains (name string, era string, planet string);

[localhost:21000] > insert into heroes values

> ('Tesla','20th century','Earth'),

> ('Pythagoras','Antiquity','Earth'),

> ('Zopzar','Far Future','Mars');

Inserted 3 rows in 2.28s

[localhost:21000] > insert into villains values

> ('Caligula','Antiquity','Earth'),

> ('John Dillinger','20th century','Earth'),

> ('Xibulor','Far Future','Venus');

Inserted 3 rows in 1.93s

[localhost:21000] > select concat(heroes.name,' vs. ',villains.name) as battle

> from heroes join villains

> where heroes.era = villains.era and heroes.planet = villains.planet;

+--------------------------+

| battle |

+--------------------------+

| Tesla vs. John Dillinger |

| Pythagoras vs. Caligula |

+--------------------------+

Returned 2 row(s) in 0.47s

Readers demanded more action, so we added elements of time travel and space travel so that any hero could face any villain. Prior to Impala 1.2.2, this type of query was impossible because all joins had to reference matching values between the two tables:

[localhost:21000] > -- Cartesian product not possible in Impala 1.1.

> select concat(heroes.name,' vs. ',villains.name) as battle from heroes join villains;

ERROR: NotImplementedException: Join between 'heroes' and 'villains' requires at least one conjunctive equality predicate between the two tables

With Impala 1.2.2, we rewrite the query slightly to use CROSS JOIN rather than JOIN, and now the result set includes all combinations:

[localhost:21000] > -- Cartesian product available in Impala 1.2.2 with the CROSS JOIN syntax.

> select concat(heroes.name,' vs. ',villains.name) as battle from heroes cross join villains;

+-------------------------------+

| battle |

+-------------------------------+

| Tesla vs. Caligula |

| Tesla vs. John Dillinger |

| Tesla vs. Xibulor |

| Pythagoras vs. Caligula |

| Pythagoras vs. John Dillinger |

| Pythagoras vs. Xibulor |

| Zopzar vs. Caligula |

| Zopzar vs. John Dillinger |

| Zopzar vs. Xibulor |

+-------------------------------+

Returned 9 row(s) in 0.33s

The full combination of rows from both tables is known as the Cartesian product. This type of result set is often used for creating grid data structures. You can also filter the result set by including WHERE clauses that do not explicitly compare columns between the two tables. The following example shows how you might produce a list of combinations of year and quarter for use in a chart, and then a shorter list with only selected quarters.

[localhost:21000] > create table x\_axis (x int);

[localhost:21000] > create table y\_axis (y int);

[localhost:21000] > insert into x\_axis values (1),(2),(3),(4);

Inserted 4 rows in 2.14s

[localhost:21000] > insert into y\_axis values (2010),(2011),(2012),(2013),(2014);

Inserted 5 rows in 1.32s

[localhost:21000] > select y as year, x as quarter from x\_axis cross join y\_axis;

+------+---------+

| year | quarter |

+------+---------+

| 2010 | 1 |

| 2011 | 1 |

| 2012 | 1 |

| 2013 | 1 |

| 2014 | 1 |

| 2010 | 2 |

| 2011 | 2 |

| 2012 | 2 |

| 2013 | 2 |

| 2014 | 2 |

| 2010 | 3 |

| 2011 | 3 |

| 2012 | 3 |

| 2013 | 3 |

| 2014 | 3 |

| 2010 | 4 |

| 2011 | 4 |

| 2012 | 4 |

| 2013 | 4 |

| 2014 | 4 |

+------+---------+

Returned 20 row(s) in 0.38s

[localhost:21000] > select y as year, x as quarter from x\_axis cross join y\_axis where x in (1,3);

+------+---------+

| year | quarter |

+------+---------+

| 2010 | 1 |

| 2011 | 1 |

| 2012 | 1 |

| 2013 | 1 |

| 2014 | 1 |

| 2010 | 3 |

| 2011 | 3 |

| 2012 | 3 |

| 2013 | 3 |

| 2014 | 3 |

+------+---------+

Returned 10 row(s) in 0.39s

## Dealing with Parquet Files with Unknown Schema

As data pipelines start to include more aspects such as NoSQL or loosely specified schemas, you might encounter situations where you have data files (particularly in Parquet format) where you do not know the precise table definition. This tutorial shows how you can build an Impala table around data that comes from non-Impala or even non-SQL sources, where you do not have control of the table layout and might not be familiar with the characteristics of the data.

The data used in this tutorial represents airline on-time arrival statistics, from October 1987 through April 2008. See the details on the [2009 ASA Data Expo web site](http://stat-computing.org/dataexpo/2009/). You can also see the [explanations of the columns](http://stat-computing.org/dataexpo/2009/the-data.html); for purposes of this exercise, wait until after following the tutorial before examining the schema, to better simulate a real-life situation where you cannot rely on assumptions and assertions about the ranges and representations of data values.

We will download Parquet files containing this data from the Ibis blog. First, we download and unpack the data files. There are 8 files totalling 1.4 GB. Each file is less than 256 MB.

$ wget -O airlines\_parquet.tar.gz https://www.dropbox.com/s/ol9x51tqp6cv4yc/airlines\_parquet.tar.gz?dl=0

...

Length: 1245204740 (1.2G) [application/octet-stream]

Saving to: “airlines\_parquet.tar.gz”

2015-08-12 17:14:24 (23.6 MB/s) - “airlines\_parquet.tar.gz” saved [1245204740/1245204740]

$ tar xvzf airlines\_parquet.tar.gz

airlines\_parquet/

airlines\_parquet/93459d994898a9ba-77674173b331fa9a\_2073981944\_data.0.parq

airlines\_parquet/93459d994898a9ba-77674173b331fa99\_1555718317\_data.1.parq

airlines\_parquet/93459d994898a9ba-77674173b331fa99\_1555718317\_data.0.parq

airlines\_parquet/93459d994898a9ba-77674173b331fa96\_2118228804\_data.0.parq

airlines\_parquet/93459d994898a9ba-77674173b331fa97\_574780876\_data.0.parq

airlines\_parquet/93459d994898a9ba-77674173b331fa96\_2118228804\_data.1.parq

airlines\_parquet/93459d994898a9ba-77674173b331fa98\_1194408366\_data.0.parq

airlines\_parquet/93459d994898a9ba-77674173b331fa9b\_1413430552\_data.0.parq

$ cd airlines\_parquet/

$ du -kch \*.parq

253M 93459d994898a9ba-77674173b331fa96\_2118228804\_data.0.parq

65M 93459d994898a9ba-77674173b331fa96\_2118228804\_data.1.parq

156M 93459d994898a9ba-77674173b331fa97\_574780876\_data.0.parq

240M 93459d994898a9ba-77674173b331fa98\_1194408366\_data.0.parq

253M 93459d994898a9ba-77674173b331fa99\_1555718317\_data.0.parq

16M 93459d994898a9ba-77674173b331fa99\_1555718317\_data.1.parq

177M 93459d994898a9ba-77674173b331fa9a\_2073981944\_data.0.parq

213M 93459d994898a9ba-77674173b331fa9b\_1413430552\_data.0.parq

1.4G total

Next, we put the Parquet data files in HDFS, all together in a single directory, with permissions on the directory and the files so that the impala user will be able to read them.

Note: After unpacking, we saw the largest Parquet file was 253 MB. When copying Parquet files into HDFS for Impala to use, for maximum query performance, make sure that each file resides in a single HDFS data block. Therefore, we pick a size larger than any single file and specify that as the block size, using the argument -Ddfs.block.size=256m on the hdfs dfs -put command.

$ hdfs dfs -mkdir -p hdfs://demo\_host.example.com:8020/user/impala/staging/airlines

$ hdfs dfs -Ddfs.block.size=256m -put \*.parq /user/impala/staging/airlines

$ hdfs dfs -ls /user/impala/staging

Found 1 items

drwxrwxrwx - hdfs supergroup 0 2015-08-12 13:52 /user/impala/staging/airlines

$ hdfs dfs -ls hdfs://demo\_host.example.com:8020/user/impala/staging/airlines

Found 8 items

-rw-r--r-- 3 jrussell supergroup 265107489 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa96\_2118228804\_data.0.parq

-rw-r--r-- 3 jrussell supergroup 67544715 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa96\_2118228804\_data.1.parq

-rw-r--r-- 3 jrussell supergroup 162556490 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa97\_574780876\_data.0.parq

-rw-r--r-- 3 jrussell supergroup 251603518 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa98\_1194408366\_data.0.parq

-rw-r--r-- 3 jrussell supergroup 265186603 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa99\_1555718317\_data.0.parq

-rw-r--r-- 3 jrussell supergroup 16663754 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa99\_1555718317\_data.1.parq

-rw-r--r-- 3 jrussell supergroup 185511677 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa9a\_2073981944\_data.0.parq

-rw-r--r-- 3 jrussell supergroup 222794621 2015-08-12 17:18 /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa9b\_1413430552\_data.0.parq

With the files in an accessible location in HDFS, we create a database table that uses the data in those files. The CREATE EXTERNAL syntax and the LOCATION attribute point Impala at the appropriate HDFS directory. The LIKE PARQUET 'path\_to\_any\_parquet\_file' clause means we skip the list of column names and types; Impala automatically gets the column names and data types straight from the data files. (Currently, this technique only works for Parquet files.) We ignore the warning about lack of READ\_WRITE access to the files in HDFS; the impala user can read the files, which will be sufficient for us to experiment with queries and perform some copy and transform operations into other tables.

$ impala-shell -i localhost

Starting Impala Shell without Kerberos authentication

Connected to localhost:21000

Server version: impalad version 2.2.0-cdh5 RELEASE (build 2ffd73a4255cefd521362ffe1cfb37463f67f75c)

Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.1.2-cdh5 (92438b7) built on Tue Feb 24 12:36:33 PST 2015)

[localhost:21000] > create database airline\_data;

[localhost:21000] > use airline\_data;

[localhost:21000] > create external table airlines\_external

> like parquet 'hdfs://demo\_host.example.com:8020/user/impala/staging/airlines/93459d994898a9ba-77674173b331fa96\_2118228804\_data.0.parq'

> stored as parquet location 'hdfs://demo\_host.example.com:8020/user/impala/staging/airlines';

WARNINGS: Impala does not have READ\_WRITE access to path 'hdfs://demo\_host.example.com:8020/user/impala/staging'

With the table created, we examine its physical and logical characteristics to confirm that the data is really there and in a format and shape that we can work with. The SHOW TABLE STATS statement gives a very high-level summary of the table, showing how many files and how much total data it contains. Also, it confirms that the table is expecting all the associated data files to be in Parquet format. (The ability to work with all kinds of HDFS data files in different formats means that it is possible to have a mismatch between the format of the data files, and the format that the table expects the data files to be in.) The SHOW FILES statement confirms that the data in the table has the expected number, names, and sizes of the original Parquet files. The DESCRIBE statement (or its abbreviation DESC) confirms the names and types of the columns that Impala automatically created after reading that metadata from the Parquet file. The DESCRIBE FORMATTED statement prints out some extra detail along with the column definitions; the pieces we care about for this exercise are the containing database for the table, the location of the associated data files in HDFS, the fact that it's an external table so Impala will not delete the HDFS files when we finish the experiments and drop the table, and the fact that the table is set up to work exclusively with files in the Parquet format.

[localhost:21000] > show table stats airlines\_external;

+-------+--------+--------+--------------+-------------------+---------+-------------------+

| #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Incremental stats |

+-------+--------+--------+--------------+-------------------+---------+-------------------+

| -1 | 8 | 1.34GB | NOT CACHED | NOT CACHED | PARQUET | false |

+-------+--------+--------+--------------+-------------------+---------+-------------------+

[localhost:21000] > show files in airlines\_external;

+----------------------------------------------------------------------------------------+----------+-----------+

| path | size | partition |

+----------------------------------------------------------------------------------------+----------+-----------+

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa96\_2118228804\_data.0.parq | 252.83MB | |

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa96\_2118228804\_data.1.parq | 64.42MB | |

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa97\_574780876\_data.0.parq | 155.03MB | |

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa98\_1194408366\_data.0.parq | 239.95MB | |

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa99\_1555718317\_data.0.parq | 252.90MB | |

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa99\_1555718317\_data.1.parq | 15.89MB | |

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa9a\_2073981944\_data.0.parq | 176.92MB | |

| /user/impala/staging/airlines/93459d994898a9ba-77674173b331fa9b\_1413430552\_data.0.parq | 212.47MB | |

+----------------------------------------------------------------------------------------+----------+-----------+

[localhost:21000] > describe airlines\_external;

+---------------------+--------+---------------------------------------------------+

| name | type | comment |

+---------------------+--------+---------------------------------------------------+

| year | int | inferred from: optional int32 year |

| month | int | inferred from: optional int32 month |

| day | int | inferred from: optional int32 day |

| dayofweek | int | inferred from: optional int32 dayofweek |

| dep\_time | int | inferred from: optional int32 dep\_time |

| crs\_dep\_time | int | inferred from: optional int32 crs\_dep\_time |

| arr\_time | int | inferred from: optional int32 arr\_time |

| crs\_arr\_time | int | inferred from: optional int32 crs\_arr\_time |

| carrier | string | inferred from: optional binary carrier |

| flight\_num | int | inferred from: optional int32 flight\_num |

| tail\_num | int | inferred from: optional int32 tail\_num |

| actual\_elapsed\_time | int | inferred from: optional int32 actual\_elapsed\_time |

| crs\_elapsed\_time | int | inferred from: optional int32 crs\_elapsed\_time |

| airtime | int | inferred from: optional int32 airtime |

| arrdelay | int | inferred from: optional int32 arrdelay |

| depdelay | int | inferred from: optional int32 depdelay |

| origin | string | inferred from: optional binary origin |

| dest | string | inferred from: optional binary dest |

| distance | int | inferred from: optional int32 distance |

| taxi\_in | int | inferred from: optional int32 taxi\_in |

| taxi\_out | int | inferred from: optional int32 taxi\_out |

| cancelled | int | inferred from: optional int32 cancelled |

| cancellation\_code | string | inferred from: optional binary cancellation\_code |

| diverted | int | inferred from: optional int32 diverted |

| carrier\_delay | int | inferred from: optional int32 carrier\_delay |

| weather\_delay | int | inferred from: optional int32 weather\_delay |

| nas\_delay | int | inferred from: optional int32 nas\_delay |

| security\_delay | int | inferred from: optional int32 security\_delay |

| late\_aircraft\_delay | int | inferred from: optional int32 late\_aircraft\_delay |

+---------------------+--------+---------------------------------------------------+

[localhost:21000] > desc formatted airlines\_external;

+------------------------------+-------------------------------

| name | type

+------------------------------+-------------------------------

...

| # Detailed Table Information | NULL

| Database: | airline\_data

| Owner: | jrussell

...

| Location: | /user/impala/staging/airlines

| Table Type: | EXTERNAL\_TABLE

...

| # Storage Information | NULL

| SerDe Library: | parquet.hive.serde.ParquetHiveSerDe

| InputFormat: | parquet.hive.DeprecatedParquetInputFormat

| OutputFormat: | parquet.hive.DeprecatedParquetOutputFormat

...

Now that we are confident that the connections are solid between the Impala table and the underlying Parquet files, we run some initial queries to understand the characteristics of the data: the overall number of rows, and the ranges and how many different values are in certain columns. For convenience in understanding the magnitude of the COUNT(\*) result, we run another query dividing the number of rows by 1 million, demonstrating that there are 123 million rows in the table.

[localhost:21000] > select count(\*) from airlines\_external;

+-----------+

| count(\*) |

+-----------+

| 123534969 |

+-----------+

Fetched 1 row(s) in 1.32s

[localhost:21000] > select count(\*) / 1e6 as 'millions of rows' from airlines\_external;

+------------------+

| millions of rows |

+------------------+

| 123.534969 |

+------------------+

Fetched 1 row(s) in 1.24s

The NDV() function stands for "number of distinct values", which for performance reasons is an estimate when there are lots of different values in the column, but is precise when the cardinality is less than 16 K. Use NDV() calls for this kind of exploration rather than COUNT(DISTINCT colname), because Impala can evaluate multiple NDV() functions in a single query, but only a single instance of COUNT DISTINCT. Here we see that there are modest numbers of different airlines, flight numbers, and origin and destination airports. Two things jump out from this query: the number of tail\_num values is much smaller than we might have expected, and there are more destination airports than origin airports. Let's dig further. What we find is that most tail\_num values are NULL. It looks like this was an experimental column that wasn't filled in accurately. We make a mental note that if we use this data as a starting point, we'll ignore this column. We also find that certain airports are represented in the ORIGIN column but not the DEST column; now we know that we cannot rely on the assumption that those sets of airport codes are identical.

Note: A slight digression for some performance tuning. Notice how the first SELECT DISTINCT DEST query takes almost 40 seconds. We expect all queries on such a small data set, less than 2 GB, to take a few seconds at most. The reason is because the expression NOT IN (SELECT origin FROM airlines\_external) produces an intermediate result set of 123 million rows, then runs 123 million comparisons on each data node against the tiny set of destination airports. The way the NOT IN operator works internally means that this intermediate result set with 123 million rows might be transmitted across the network to each data node in the cluster. Applying another DISTINCT inside the NOT IN subquery means that the intermediate result set is only 340 items, resulting in much less network traffic and fewer comparison operations. The more efficient query with the added DISTINCT is approximately 7 times as fast.

[localhost:21000] > select ndv(carrier), ndv(flight\_num), ndv(tail\_num),

> ndv(origin), ndv(dest) from airlines\_external;

+--------------+-----------------+---------------+-------------+-----------+

| ndv(carrier) | ndv(flight\_num) | ndv(tail\_num) | ndv(origin) | ndv(dest) |

+--------------+-----------------+---------------+-------------+-----------+

| 29 | 9086 | 3 | 340 | 347 |

+--------------+-----------------+---------------+-------------+-----------+

[localhost:21000] > select tail\_num, count(\*) as howmany from airlines\_external

> group by tail\_num;

+----------+-----------+

| tail\_num | howmany |

+----------+-----------+

| 715 | 1 |

| 0 | 406405 |

| 112 | 6562 |

| NULL | 123122001 |

+----------+-----------+

Fetched 1 row(s) in 5.18s

[localhost:21000] > select distinct dest from airlines\_external

> where dest not in (select origin from airlines\_external);

+------+

| dest |

+------+

| LBF |

| CBM |

| RCA |

| SKA |

| LAR |

+------+

Fetched 5 row(s) in 39.64s

[localhost:21000] > select distinct dest from airlines\_external

> where dest not in (select distinct origin from airlines\_external);

+------+

| dest |

+------+

| LBF |

| RCA |

| CBM |

| SKA |

| LAR |

+------+

Fetched 5 row(s) in 5.59s

[localhost:21000] > select distinct origin from airlines\_external

> where origin not in (select distinct dest from airlines\_external);

Fetched 0 row(s) in 5.37s

Next, we try doing a simple calculation, with results broken down by year. This reveals that some years have no data in the AIRTIME column. That means we might be able to use that column in queries involving certain date ranges, but we cannot count on it to always be reliable. The question of whether a column contains any NULL values, and if so what is their number, proportion, and distribution, comes up again and again when doing initial exploration of a data set.

[localhost:21000] > select year, sum(airtime) from airlines\_external

> group by year order by year desc;

+------+--------------+

| year | sum(airtime) |

+------+--------------+

| 2008 | 713050445 |

| 2007 | 748015545 |

| 2006 | 720372850 |

| 2005 | 708204026 |

| 2004 | 714276973 |

| 2003 | 665706940 |

| 2002 | 549761849 |

| 2001 | 590867745 |

| 2000 | 583537683 |

| 1999 | 561219227 |

| 1998 | 538050663 |

| 1997 | 536991229 |

| 1996 | 519440044 |

| 1995 | 513364265 |

| 1994 | NULL |

| 1993 | NULL |

| 1992 | NULL |

| 1991 | NULL |

| 1990 | NULL |

| 1989 | NULL |

| 1988 | NULL |

| 1987 | NULL |

+------+--------------+

With the notion of NULL values in mind, let's come back to the TAILNUM column that we discovered had a lot of NULLs. Let's quantify the NULL and non-NULL values in that column for better understanding. First, we just count the overall number of rows versus the non-NULL values in that column. That initial result gives the appearance of relatively few non-NULL values, but we can break it down more clearly in a single query. Once we have the COUNT(\*) and the COUNT(colname) numbers, we can encode that initial query in a WITH clause, then run a followon query that performs multiple arithmetic operations on those values. Seeing that only one-third of one percent of all rows have non-NULL values for the TAILNUM column clearly illustrates that that column is not of much use.

[localhost:21000] > select count(\*) as 'rows', count(tail\_num) as 'non-null tail numbers'

> from airlines\_external;

+-----------+-----------------------+

| rows | non-null tail numbers |

+-----------+-----------------------+

| 123534969 | 412968 |

+-----------+-----------------------+

Fetched 1 row(s) in 1.51s

[localhost:21000] > with t1 as

> (select count(\*) as 'rows', count(tail\_num) as 'nonnull'

> from airlines\_external)

> select `rows`, `nonnull`, `rows` - `nonnull` as 'nulls',

> (`nonnull` / `rows`) \* 100 as 'percentage non-null'

> from t1;

+-----------+---------+-----------+---------------------+

| rows | nonnull | nulls | percentage non-null |

+-----------+---------+-----------+---------------------+

| 123534969 | 412968 | 123122001 | 0.3342923897119365 |

+-----------+---------+-----------+---------------------+

By examining other columns using these techniques, we can form a mental picture of the way data is distributed throughout the table, and which columns are most significant for query purposes. For this tutorial, we focus mostly on the fields likely to hold discrete values, rather than columns such as ACTUAL\_ELAPSED\_TIME whose names suggest they hold measurements. We would dig deeper into those columns once we had a clear picture of which questions were worthwhile to ask, and what kinds of trends we might look for. For the final piece of initial exploration, let's look at the YEAR column. A simple GROUP BY query shows that it has a well-defined range, a manageable number of distinct values, and relatively even distribution of rows across the different years.

[localhost:21000] > select min(year), max(year), ndv(year) from airlines\_external;

+-----------+-----------+-----------+

| min(year) | max(year) | ndv(year) |

+-----------+-----------+-----------+

| 1987 | 2008 | 22 |

+-----------+-----------+-----------+

Fetched 1 row(s) in 2.03s

[localhost:21000] > select year, count(\*) howmany from airlines\_external

> group by year order by year desc;

+------+---------+

| year | howmany |

+------+---------+

| 2008 | 7009728 |

| 2007 | 7453215 |

| 2006 | 7141922 |

| 2005 | 7140596 |

| 2004 | 7129270 |

| 2003 | 6488540 |

| 2002 | 5271359 |

| 2001 | 5967780 |

| 2000 | 5683047 |

| 1999 | 5527884 |

| 1998 | 5384721 |

| 1997 | 5411843 |

| 1996 | 5351983 |

| 1995 | 5327435 |

| 1994 | 5180048 |

| 1993 | 5070501 |

| 1992 | 5092157 |

| 1991 | 5076925 |

| 1990 | 5270893 |

| 1989 | 5041200 |

| 1988 | 5202096 |

| 1987 | 1311826 |

+------+---------+

Fetched 22 row(s) in 2.13s

We could go quite far with the data in this initial raw format, just as we downloaded it from the web. If the data set proved to be useful and worth persisting in Impala for extensive queries, we might want to copy it to an internal table, letting Impala manage the data files and perhaps reorganizing a little for higher efficiency. In this next stage of the tutorial, we copy the original data into a partitioned table, still in Parquet format. Partitioning based on the YEAR column lets us run queries with clauses such as WHERE year = 2001 or WHERE year BETWEEN 1989 AND 1999, which can dramatically cut down on I/O by ignoring all the data from years outside the desired range. Rather than reading all the data and then deciding which rows are in the matching years, Impala can zero in on only the data files from specific YEAR partitions. To do this, Impala physically reorganizes the data files, putting the rows from each year into data files in a separate HDFS directory for each YEAR value. Along the way, we'll also get rid of the TAIL\_NUM column that proved to be almost entirely NULL.

The first step is to create a new table with a layout very similar to the original AIRLINES\_EXTERNAL table. We'll do that by reverse-engineering a CREATE TABLE statement for the first table, then tweaking it slightly to include a PARTITION BY clause for YEAR, and excluding the TAIL\_NUM column. The SHOW CREATE TABLE statement gives us the starting point.

[localhost:21000] > show create table airlines\_external;

+-------------------------------------------------------------------------------------

| result

+-------------------------------------------------------------------------------------

| CREATE EXTERNAL TABLE airline\_data.airlines\_external (

| year INT COMMENT 'inferred from: optional int32 year',

| month INT COMMENT 'inferred from: optional int32 month',

| day INT COMMENT 'inferred from: optional int32 day',

| dayofweek INT COMMENT 'inferred from: optional int32 dayofweek',

| dep\_time INT COMMENT 'inferred from: optional int32 dep\_time',

| crs\_dep\_time INT COMMENT 'inferred from: optional int32 crs\_dep\_time',

| arr\_time INT COMMENT 'inferred from: optional int32 arr\_time',

| crs\_arr\_time INT COMMENT 'inferred from: optional int32 crs\_arr\_time',

| carrier STRING COMMENT 'inferred from: optional binary carrier',

| flight\_num INT COMMENT 'inferred from: optional int32 flight\_num',

| tail\_num INT COMMENT 'inferred from: optional int32 tail\_num',

| actual\_elapsed\_time INT COMMENT 'inferred from: optional int32 actual\_elapsed\_time',

| crs\_elapsed\_time INT COMMENT 'inferred from: optional int32 crs\_elapsed\_time',

| airtime INT COMMENT 'inferred from: optional int32 airtime',

| arrdelay INT COMMENT 'inferred from: optional int32 arrdelay',

| depdelay INT COMMENT 'inferred from: optional int32 depdelay',

| origin STRING COMMENT 'inferred from: optional binary origin',

| dest STRING COMMENT 'inferred from: optional binary dest',

| distance INT COMMENT 'inferred from: optional int32 distance',

| taxi\_in INT COMMENT 'inferred from: optional int32 taxi\_in',

| taxi\_out INT COMMENT 'inferred from: optional int32 taxi\_out',

| cancelled INT COMMENT 'inferred from: optional int32 cancelled',

| cancellation\_code STRING COMMENT 'inferred from: optional binary cancellation\_code',

| diverted INT COMMENT 'inferred from: optional int32 diverted',

| carrier\_delay INT COMMENT 'inferred from: optional int32 carrier\_delay',

| weather\_delay INT COMMENT 'inferred from: optional int32 weather\_delay',

| nas\_delay INT COMMENT 'inferred from: optional int32 nas\_delay',

| security\_delay INT COMMENT 'inferred from: optional int32 security\_delay',

| late\_aircraft\_delay INT COMMENT 'inferred from: optional int32 late\_aircraft\_delay'

| )

| STORED AS PARQUET

| LOCATION 'hdfs://a1730.example.com:8020/user/impala/staging/airlines'

| TBLPROPERTIES ('numFiles'='0', 'COLUMN\_STATS\_ACCURATE'='false',

| 'transient\_lastDdlTime'='1439425228', 'numRows'='-1', 'totalSize'='0',

| 'rawDataSize'='-1')

+-------------------------------------------------------------------------------------

Fetched 1 row(s) in 0.03s

[localhost:21000] > quit;

Although we could edit that output into a new SQL statement, all the ASCII box characters make such editing inconvenient. To get a more stripped-down CREATE TABLE to start with, we restart the impala-shell command with the -B option, which turns off the box-drawing behavior.

[localhost:21000] > quit;

Goodbye jrussell

$ impala-shell -i localhost -B -d airline\_data;

Starting Impala Shell without Kerberos authentication

Connected to localhost:21000

Server version: impalad version 2.2.0-cdh5 RELEASE (build 2ffd73a4255cefd521362ffe1cfb37463f67f75c)

Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.1.2-cdh5 (92438b7) built on Tue Feb 24 12:36:33 PST 2015)

[localhost:21000] > show create table airlines\_external;

"CREATE EXTERNAL TABLE airline\_data.airlines\_external (

year INT COMMENT 'inferred from: optional int32 year',

month INT COMMENT 'inferred from: optional int32 month',

day INT COMMENT 'inferred from: optional int32 day',

dayofweek INT COMMENT 'inferred from: optional int32 dayofweek',

dep\_time INT COMMENT 'inferred from: optional int32 dep\_time',

crs\_dep\_time INT COMMENT 'inferred from: optional int32 crs\_dep\_time',

arr\_time INT COMMENT 'inferred from: optional int32 arr\_time',

crs\_arr\_time INT COMMENT 'inferred from: optional int32 crs\_arr\_time',

carrier STRING COMMENT 'inferred from: optional binary carrier',

flight\_num INT COMMENT 'inferred from: optional int32 flight\_num',

tail\_num INT COMMENT 'inferred from: optional int32 tail\_num',

actual\_elapsed\_time INT COMMENT 'inferred from: optional int32 actual\_elapsed\_time',

crs\_elapsed\_time INT COMMENT 'inferred from: optional int32 crs\_elapsed\_time',

airtime INT COMMENT 'inferred from: optional int32 airtime',

arrdelay INT COMMENT 'inferred from: optional int32 arrdelay',

depdelay INT COMMENT 'inferred from: optional int32 depdelay',

origin STRING COMMENT 'inferred from: optional binary origin',

dest STRING COMMENT 'inferred from: optional binary dest',

distance INT COMMENT 'inferred from: optional int32 distance',

taxi\_in INT COMMENT 'inferred from: optional int32 taxi\_in',

taxi\_out INT COMMENT 'inferred from: optional int32 taxi\_out',

cancelled INT COMMENT 'inferred from: optional int32 cancelled',

cancellation\_code STRING COMMENT 'inferred from: optional binary cancellation\_code',

diverted INT COMMENT 'inferred from: optional int32 diverted',

carrier\_delay INT COMMENT 'inferred from: optional int32 carrier\_delay',

weather\_delay INT COMMENT 'inferred from: optional int32 weather\_delay',

nas\_delay INT COMMENT 'inferred from: optional int32 nas\_delay',

security\_delay INT COMMENT 'inferred from: optional int32 security\_delay',

late\_aircraft\_delay INT COMMENT 'inferred from: optional int32 late\_aircraft\_delay'

)

STORED AS PARQUET

LOCATION 'hdfs://a1730.example.com:8020/user/impala/staging/airlines'

TBLPROPERTIES ('numFiles'='0', 'COLUMN\_STATS\_ACCURATE'='false',

'transient\_lastDdlTime'='1439425228', 'numRows'='-1', 'totalSize'='0',

'rawDataSize'='-1')"

Fetched 1 row(s) in 0.01s

After copying and pasting the CREATE TABLE statement into a text editor for fine-tuning, we quit and restart impala-shell without the -B option, to switch back to regular output.

Next we run the CREATE TABLE statement that we adapted from the SHOW CREATE TABLE output. We kept the STORED AS PARQUET clause because we want to rearrange the data somewhat but still keep it in the high-performance Parquet format. The LOCATION and TBLPROPERTIES clauses are not relevant for this new table, so we edit those out. Because we are going to partition the new table based on the YEAR column, we move that column name (and its type) into a new PARTITIONED BY clause.

[localhost:21000] > CREATE TABLE airline\_data.airlines

> (

> month INT,

> day INT,

> dayofweek INT,

> dep\_time INT,

> crs\_dep\_time INT,

> arr\_time INT,

> crs\_arr\_time INT,

> carrier STRING,

> flight\_num INT,

> actual\_elapsed\_time INT,

> crs\_elapsed\_time INT,

> airtime INT,

> arrdelay INT,

> depdelay INT,

> origin STRING,

> dest STRING,

> distance INT,

> taxi\_in INT,

> taxi\_out INT,

> cancelled INT,

> cancellation\_code STRING,

> diverted INT,

> carrier\_delay INT,

> weather\_delay INT,

> nas\_delay INT,

> security\_delay INT,

> late\_aircraft\_delay INT

> )

> STORED AS PARQUET

> PARTITIONED BY (year INT);

Fetched 0 row(s) in 0.10s

Next, we copy all the rows from the original table into this new one with an INSERT statement. (We edited the CREATE TABLE statement to make an INSERT statement with the column names in the same order.) The only change is to add a PARTITION(year) clause, and move the YEAR column to the very end of the SELECT list of the INSERT statement. Specifying PARTITION(year), rather than a fixed value such as PARTITION(year=2000), means that Impala figures out the partition value for each row based on the value of the very last column in the SELECT list. This is the first SQL statement that legitimately takes any substantial time, because the rows from different years are shuffled around the cluster; the rows that go into each partition are collected on one node, before being written to one or more new data files.

[localhost:21000] > INSERT INTO airline\_data.airlines

> PARTITION (year)

> SELECT

> month,

> day,

> dayofweek,

> dep\_time,

> crs\_dep\_time,

> arr\_time,

> crs\_arr\_time,

> carrier,

> flight\_num,

> actual\_elapsed\_time,

> crs\_elapsed\_time,

> airtime,

> arrdelay,

> depdelay,

> origin,

> dest,

> distance,

> taxi\_in,

> taxi\_out,

> cancelled,

> cancellation\_code,

> diverted,

> carrier\_delay,

> weather\_delay,

> nas\_delay,

> security\_delay,

> late\_aircraft\_delay,

> year

> FROM airline\_data.airlines\_external;

Inserted 123534969 row(s) in 202.70s

Once partitioning or join queries come into play, it's important to have statistics that Impala can use to optimize queries on the corresponding tables. The COMPUTE INCREMENTAL STATS statement is the way to collect statistics for partitioned tables. Then the SHOW TABLE STATS statement confirms that the statistics are in place for each partition, and also illustrates how many files and how much raw data is in each partition.

[localhost:21000] > compute incremental stats airlines;

+-------------------------------------------+

| summary |

+-------------------------------------------+

| Updated 22 partition(s) and 27 column(s). |

+-------------------------------------------+

[localhost:21000] > show table stats airlines;

+-------+-----------+--------+----------+--------------+------------+---------+-------------------+

| year | #Rows | #Files | Size | Bytes Cached | Cache Repl | Format | Incremental stats |

+-------+-----------+--------+----------+--------------+------------+---------+-----

| 1987 | 1311826 | 1 | 9.32MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1988 | 5202096 | 1 | 37.04MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1989 | 5041200 | 1 | 36.25MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1990 | 5270893 | 1 | 38.39MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1991 | 5076925 | 1 | 37.23MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1992 | 5092157 | 1 | 36.85MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1993 | 5070501 | 1 | 37.16MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1994 | 5180048 | 1 | 38.31MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1995 | 5327435 | 1 | 53.14MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1996 | 5351983 | 1 | 53.64MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1997 | 5411843 | 1 | 54.41MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1998 | 5384721 | 1 | 54.01MB | NOT CACHED | NOT CACHED | PARQUET | true

| 1999 | 5527884 | 1 | 56.32MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2000 | 5683047 | 1 | 58.15MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2001 | 5967780 | 1 | 60.65MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2002 | 5271359 | 1 | 57.99MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2003 | 6488540 | 1 | 81.33MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2004 | 7129270 | 1 | 103.19MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2005 | 7140596 | 1 | 102.61MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2006 | 7141922 | 1 | 106.03MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2007 | 7453215 | 1 | 112.15MB | NOT CACHED | NOT CACHED | PARQUET | true

| 2008 | 7009728 | 1 | 105.76MB | NOT CACHED | NOT CACHED | PARQUET | true

| Total | 123534969 | 22 | 1.30GB | 0B | | |

+-------+-----------+--------+----------+--------------+------------+---------+-----

At this point, we go through a quick thought process to sanity check the partitioning we did. All the partitions have exactly one file, which is on the low side. A query that includes a clause WHERE year=2004 will only read a single data block; that data block will be read and processed by a single data node; therefore, for a query targeting a single year, all the other nodes in the cluster will sit idle while all the work happens on a single machine. It's even possible that by chance (depending on HDFS replication factor and the way data blocks are distributed across the cluster), that multiple year partitions selected by a filter such as WHERE year BETWEEN 1999 AND 2001 could all be read and processed by the same data node. The more data files each partition has, the more parallelism you can get and the less probability of "hotspots" occurring on particular nodes, therefore a bigger performance boost by having a big CDH cluster.

However, the more data files, the less data goes in each one. The overhead of dividing the work in a parallel query might not be worth it if each node is only reading a few megabytes. 50 or 100 megabytes is a decent size for a Parquet data block; 9 or 37 megabytes is on the small side. Which is to say, the data distribution we ended up with based on this partitioning scheme is on the borderline between sensible (reasonably large files) and suboptimal (few files in each partition). The way to see how well it works in practice is to run the same queries against the original flat table and the new partitioned table, and compare times.

Spoiler: in this case, with my particular 4-node cluster with its specific distribution of data blocks and my particular exploratory queries, queries against the partitioned table do consistently run faster than the same queries against the unpartitioned table. But I could not be sure that would be the case without some real measurements. Here are some queries I ran to draw that conclusion, first against AIRLINES\_EXTERNAL (no partitioning), then against AIRLINES (partitioned by year). The AIRLINES queries are consistently faster. Changing the volume of data, changing the size of the cluster, running queries that did or didn't refer to the partition key columns, or other factors could change the results to favor one table layout or the other.

Note: If you find the volume of each partition is only in the low tens of megabytes, consider lowering the granularity of partitioning. For example, instead of partitioning by year, month, and day, partition by year and month or even just by year. The ideal layout to distribute work efficiently in a parallel query is many tens or even hundreds of megabytes per Parquet file, and the number of Parquet files in each partition somewhat higher than the number of data nodes.

[localhost:21000] > select sum(airtime) from airlines\_external;

+--------------+

| sum(airtime) |

+--------------+

| 8662859484 |

+--------------+

Fetched 1 row(s) in 2.02s

[localhost:21000] > select sum(airtime) from airlines;

+--------------+

| sum(airtime) |

+--------------+

| 8662859484 |

+--------------+

Fetched 1 row(s) in 1.21s

[localhost:21000] > select sum(airtime) from airlines\_external where year = 2005;

+--------------+

| sum(airtime) |

+--------------+

| 708204026 |

+--------------+

Fetched 1 row(s) in 2.61s

[localhost:21000] > select sum(airtime) from airlines where year = 2005;

+--------------+

| sum(airtime) |

+--------------+

| 708204026 |

+--------------+

Fetched 1 row(s) in 1.19s

[localhost:21000] > select sum(airtime) from airlines\_external where dayofweek = 1;

+--------------+

| sum(airtime) |

+--------------+

| 1264945051 |

+--------------+

Fetched 1 row(s) in 2.82s

[localhost:21000] > select sum(airtime) from airlines where dayofweek = 1;

+--------------+

| sum(airtime) |

+--------------+

| 1264945051 |

+--------------+

Fetched 1 row(s) in 1.61s

Now we can finally do some serious analysis with this data set that, remember, a few minutes ago all we had were some raw data files and we didn't even know what columns they contained. Let's see whether the "air time" of a flight tends to be different depending on the day of the week. We can see that the average is a little higher on day number 6; perhaps Saturday is a busy flying day and planes have to circle for longer at the destination airport before landing.

[localhost:21000] > select dayofweek, avg(airtime) from airlines

> group by dayofweek order by dayofweek;

+-----------+-------------------+

| dayofweek | avg(airtime) |

+-----------+-------------------+

| 1 | 102.1560425016671 |

| 2 | 102.1582931538807 |

| 3 | 102.2170009256653 |

| 4 | 102.37477661846 |

| 5 | 102.2697358763511 |

| 6 | 105.3627448363705 |

| 7 | 103.4144351202054 |

+-----------+-------------------+

Fetched 7 row(s) in 2.25s

To see if the apparent trend holds up over time, let's do the same breakdown by day of week, but also split up by year. Now we can see that day number 6 consistently has a higher average air time in each year. We can also see that the average air time increased over time across the board. And the presence of NULL for this column in years 1987 to 1994 shows that queries involving this column need to be restricted to a date range of 1995 and higher.

[localhost:21000] > select year, dayofweek, avg(airtime) from airlines

> group by year, dayofweek order by year desc, dayofweek;

+------+-----------+-------------------+

| year | dayofweek | avg(airtime) |

+------+-----------+-------------------+

| 2008 | 1 | 103.1821651651355 |

| 2008 | 2 | 103.2149301386094 |

| 2008 | 3 | 103.0585076622796 |

| 2008 | 4 | 103.4671383539038 |

| 2008 | 5 | 103.5575385182659 |

| 2008 | 6 | 107.4006306562128 |

| 2008 | 7 | 104.8648851041755 |

| 2007 | 1 | 102.2196114337825 |

| 2007 | 2 | 101.9317791906348 |

| 2007 | 3 | 102.0964767689043 |

| 2007 | 4 | 102.6215927201686 |

| 2007 | 5 | 102.4289399000661 |

| 2007 | 6 | 105.1477448215756 |

| 2007 | 7 | 103.6305945644095 |

...

| 1996 | 1 | 99.33860750862108 |

| 1996 | 2 | 99.54225446396656 |

| 1996 | 3 | 99.41129336113134 |

| 1996 | 4 | 99.5110373340348 |

| 1996 | 5 | 99.22120745027595 |

| 1996 | 6 | 101.1717447111921 |

| 1996 | 7 | 99.95410136133704 |

| 1995 | 1 | 96.93779698300494 |

| 1995 | 2 | 96.93458674589712 |

| 1995 | 3 | 97.00972311337051 |

| 1995 | 4 | 96.90843832024412 |

| 1995 | 5 | 96.78382115425562 |

| 1995 | 6 | 98.70872826057003 |

| 1995 | 7 | 97.85570478374616 |

| 1994 | 1 | NULL |

| 1994 | 2 | NULL |

| 1994 | 3 | NULL |

...

| 1987 | 5 | NULL |

| 1987 | 6 | NULL |

| 1987 | 7 | NULL |

+------+-----------+-------------------+

**Categories:** [Data Analysts](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_data_analysts.html) | [Developers](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_developers.html) | [File Formats](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_file_formats.html) | [Getting Started](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_getting_started.html) | [Impala](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_impala.html) | [Parquet](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_parquet.html) | [Querying](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_querying.html) | [SQL](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_sql.html) | [Schemas](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_schemas.html) | [Tables](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_tables.html) | [Tutorials](https://www.cloudera.com/documentation/enterprise/latest/categories/hub_tutorials.html) | [All Categories](https://www.cloudera.com/documentation/enterprise/latest/categories/hub.html)

## Impala笔记

### Explore a New Impala Instance

* How to tell which version of Impala is running on your system.
* How to find the names of databases in an Impala instance, either displaying the full list or searching for specific names.
* How to find the names of tables in an Impala database, either displaying the full list or searching for specific names.
* How to switch between databases and check which database you are currently in.
* How to learn the column names and types of a table.
* How to create databases and tables, insert small amounts of test data, and run simple queries.

Select version;

Show databases;

Select current\_database();

Show tables;

Show tables in d3;

Show tables in tpc;

Show tables in tpc like ‘customer\*’;

Use tpc;

Show tables like ‘\*view\*’;(查询含有view字符串的表)

Describe city;

Select count(\*) from city;

Desc customer;

Select count(\*) from customer;

Select count(\*) from customer where c\_email\_address is null;

Select distinct c\_salutation from customer limit 10;

Create database experiments;

Show databases;

Show databases like ‘exp\*’;

Create table t1(x int);

Show tables;

Alter table t1 rename to experiments.t1; (更改表到另一databases下)

#### Note: As you graduate to more realistic scenarios, you will use more elaborate tables with many columns, features such as partitioning, and file formats such as Parquet. When dealing with realistic data volumes, you will bring in data using LOAD DATA or INSERT ... SELECT statements to operate on millions or billions of rows at once.

#### The following example sets up a couple of simple tables with a few rows, and performs queries involving sorting, aggregate functions and joins.

insert into t1 values (1), (3), (2), (4);

select x from t1 order by x desc;

select min(x), max(x), sum(x), avg(x) from t1;

[localhost:21000] > create table t2 (id int, word string);

[localhost:21000] > insert into t2 values (1, "one"), (3, "three"), (5, 'five');

[localhost:21000] > select word from t1 join t2 on (t1.x = t2.id);

### Load CSV Data from Local Files

$ hdfs dfs -mkdir -p /user/username/sample\_data/tab1 /user/username/sample\_data/tab2

$ hdfs dfs -put tab1.csv /user/username/sample\_data/tab1

$ hdfs dfs -ls /user/username/sample\_data/tab1

Found 1 items

-rw-r--r-- 1 username username 192 2013-04-02 20:08 /user/username/sample\_data/tab1/tab1.csv

$ hdfs dfs -put tab2.csv /user/username/sample\_data/tab2

$ hdfs dfs -ls /user/username/sample\_data/tab2

Found 1 items

-rw-r--r-- 1 username username 158 2013-04-02 20:09 /user/username/sample\_data/tab2/tab2.csv

DROP TABLE IF EXISTS tab1;

-- The EXTERNAL clause means the data is located outside the central location

-- for Impala data files and is preserved when the associated Impala table is dropped.

-- We expect the data to already exist in the directory specified by the LOCATION clause.

CREATE EXTERNAL TABLE tab1

(

id INT,

col\_1 BOOLEAN,

col\_2 DOUBLE,

col\_3 TIMESTAMP

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/user/username/sample\_data/tab1';

DROP TABLE IF EXISTS tab2;

-- TAB2 is an external table, similar to TAB1.

CREATE EXTERNAL TABLE tab2

(

id INT,

col\_1 BOOLEAN,

col\_2 DOUBLE

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/user/username/sample\_data/tab2';

DROP TABLE IF EXISTS tab3;

-- Leaving out the EXTERNAL clause means the data will be managed

-- in the central Impala data directory tree. Rather than reading

-- existing data files when the table is created, we load the

-- data after creating the table.

CREATE TABLE tab3

(

id INT,

col\_1 BOOLEAN,

col\_2 DOUBLE,

month INT,

day INT

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

### Point an Impala Table at Existing Data Files

Here is a SQL script to set up Impala tables pointing to some of these data files in HDFS. (The script in the VM sets up tables like this through Hive; ignore those tables for purposes of this demonstration.) Save the following as customer\_setup.sql:

--

-- store\_sales fact table and surrounding dimension tables only

--

create database tpcds;

use tpcds;

drop table if exists customer;

create external table customer

(

c\_customer\_sk int,

c\_customer\_id string,

c\_current\_cdemo\_sk int,

c\_current\_hdemo\_sk int,

c\_current\_addr\_sk int,

c\_first\_shipto\_date\_sk int,

c\_first\_sales\_date\_sk int,

c\_salutation string,

c\_first\_name string,

c\_last\_name string,

c\_preferred\_cust\_flag string,

c\_birth\_day int,

c\_birth\_month int,

c\_birth\_year int,

c\_birth\_country string,

c\_login string,

c\_email\_address string,

c\_last\_review\_date string

)

row format delimited fields terminated by '|'

location '/user/hive/tpcds/customer';

drop table if exists customer\_address;

create external table customer\_address

(

ca\_address\_sk int,

ca\_address\_id string,

ca\_street\_number string,

ca\_street\_name string,

ca\_street\_type string,

ca\_suite\_number string,

ca\_city string,

ca\_county string,

ca\_state string,

ca\_zip string,

ca\_country string,

ca\_gmt\_offset float,

ca\_location\_type string

)

row format delimited fields terminated by '|'

location '/user/hive/tpcds/customer\_address';

We would run this script with a command such as:

impala-shell -i localhost -f customer\_setup.sql

### Data Loading and Querying Examples

##### Example: Aggregate and Join

SELECT tab1.col\_1, MAX(tab2.col\_2), MIN(tab2.col\_2)

FROM tab2 JOIN tab1 USING (id)

GROUP BY col\_1 ORDER BY 1 LIMIT 5;

##### Example: Subquery, Aggregate and Joins

SELECT tab2.\*

FROM tab2,

(SELECT tab1.col\_1, MAX(tab2.col\_2) AS max\_col2

FROM tab2, tab1

WHERE tab1.id = tab2.id

GROUP BY col\_1) subquery1

WHERE subquery1.max\_col2 = tab2.col\_2;

##### Example: INSERT Query

INSERT OVERWRITE TABLE tab3

SELECT id, col\_1, col\_2, MONTH(col\_3), DAYOFMONTH(col\_3)

FROM tab1 WHERE YEAR(col\_3) = 2012;

### Advanced Tutorials

#### Attaching an External Partitioned Table to an HDFS Directory Structure

create database external\_partitions;

use external\_partitions;

create table logs (field1 string, field2 string, field3 string)

partitioned by (year string, month string , day string, host string)

row format delimited fields terminated by ',';

insert into logs partition (year="2013", month="07", day="28", host="host1") values ("foo","foo","foo");

insert into logs partition (year="2013", month="07", day="28", host="host2") values ("foo","foo","foo");

insert into logs partition (year="2013", month="07", day="29", host="host1") values ("foo","foo","foo");

insert into logs partition (year="2013", month="07", day="29", host="host2") values ("foo","foo","foo");

insert into logs partition (year="2013", month="08", day="01", host="host1") values ("foo","foo","foo");

This tutorial shows how you might set up a directory tree in HDFS, put data files into the lowest-level subdirectories, and then use an Impala external table to query the data files from their original locations.

The tutorial uses a table with web log data, with separate subdirectories for the year, month, day, and host. For simplicity, we use a tiny amount of CSV data, loading the same data into each partition.

#### Switching Back and Forth Between Impala and Hive

#### Cross Joins and Cartesian Products with the CROSS JOIN Operator

## Impala比hive快的原因