WEEK 5 HONORS

LEARNING OBJECTIVES

- Use views to simplify queries
- Identify and use some common strategies for examining and optimizing query performance
- Use partitions to organize data and improve query performance
- Create and query tables with complex data structures when appropriate
- Describe the difference between storage engines and file systems

SIMPLIFYING QUERIES WITH VIEWS

What to Do When Queries Are Too Complex

Anyone who has written SQL queries for real-world applications should be familiar with how lengthy and complex these queries can be. For example, the query shown here combines data from the flights, airlines, and planes tables to show information about the kinds of aircraft flying into or out of a particular airport. This query is not even really that long by many standards, but still working with queries like this, can become cumbersome. With the example here, it could be that you need to repeatedly enter the same query, substituting only the origin and dest values in the where clause. Using views can help you to simplify complex queries especially ones that you need to run repeatedly. A view is like a saved query which can be used as if it were a table itself. You can hide the complex part of the query in the definition of the view, allowing you to run a simpler query against that view each time. Views can be useful not only for convenience but also for security. For example, if a table contains sensitive information, you can restrict access to the table and create a view that exposes only the rows and columns of the table that are not sensitive. In this lesson, you'll learn how to create, query, modify, and remove views, and you'll learn about the limitations of views in Hive and Impala.

CREATING AND QUERYING VIEWS

A ***view*** is a saved query, which then can be queried as if it were a table. The syntax for creating a view is the same as using **CREATE TABLE AS SELECT** (CTAS), but with **VIEW** instead of **TABLE**:

```
CREATE VIEW viewname AS
   SELECT col1, col2, ... FROM tablename ... ;
```

Views appear in lists of the tables within a database (in Hue interfaces and in the result of **SHOW TABLES**) exactly as if they were tables.

For example, this query to show information about the aircraft flying into or out of a particular airport might be something you want to explore for different airports:

```
SELECT f.carrier, name, origin,
      dest, type, manufacturer, model,
      p.year, engines, seats, engine
   FROM flights f
      JOIN airlines a
         ON (f.carrier = a.carrier)
      JOIN planes p
         ON (f.tailnum = p.tailnum)
   WHERE origin='BOS' OR dest='BOS';
```

Rather than running this every time you want to look at a different airport, you could save a view with all the information for all airports:

```
CREATE VIEW craft_information AS
   SELECT f.carrier, name, origin,
         dest, type, manufacturer, model,
         p.year, engines, seats, engine
      FROM flights f
      JOIN airlines a
         ON (f.carrier = a.carrier)
      JOIN planes p
         ON (f.tailnum = p.tailnum);
```

Then you can query the view:

```
SELECT * FROM craft_information WHERE origin='BOS' or dest='BOS';
```

Note that only the columns specified in the CREATE VIEW statement will be returned when you query the view. It's also possible to limit which rows can be returned by using a WHERE clause in the view definition. By limiting which columns and rows can be returned, views can be used to prevent users from accessing sensitive information. So views can be used both for convenience ***and*** for security.

Create the craft_information view described above.
Use SHOW TABLES; to get a list of tables in your active database. See if craft_information is included, and if it is, does it appear any differently from the actual tables in the list?
Use DESCRIBE craft_information; and then DESCRIBE FORMATTED craft_information; Note that there is nothing in the basic DESCRIBE results that indicates this is a view rather than a table, then find what there ***is*** in DESCRIBE FORMATTED that indicates this.

Run a query to return sample craft information for BOS (Logan International Airport in Boston), or another U.S. airport that you have flown from. (Not all airports will be included in the database, but you can certainly try! If you get 0 results, try a larger airport.) ***Note:*** Most airports are likely to return a large number of rows—BOS returns over 2 million rows, for example—so if you are using the command line, you should limit the number of rows returned. If you're interested, you could also limit the results by picking a particular carrier as well as the airport for your WHERE clause, but even then, you might get thousands of rows.

If this were a table, there would be a craft_information storage directory in the file system (for example, in /user/hive/warehouse/ or /user/hive/warehouse/fly.db, depending on which database you had as your active database). Check HDFS for such a directory. (There should be none.) The view uses the same data as the source tables, so no storage directory is created.
Do not try dropping the view; you'll use it in the next reading.

MODIFYING AND REMOVING VIEWS

As with tables, you can modify and remove views—but there are distinct differences in what you can do, and in the results.

MODIFYING VIEWS

You can alter a table in several different ways: rename the table, move it to a different database, change column names or types, change column order (Hive only), add or remove columns (including replacing ***all*** columns at once), or change table properties such as whether the table is managed or unmanaged. The ways you can modify a view are quite different. In each case you use **ALTER VIEW**, which different additional clauses. Hive and Impala support different clauses, so please note which engine supports which modifications.
Here are two examples:

Associate with a different query (Hive or Impala): Using either Hive or Impala, you can keep the view name but change the underlying query. To do this, use this syntax:

ALTER VIEW *viewname* AS SELECT …;

Supply the new query in the **SELECT** statement after the **AS** keyword.
Rename or move to a different database (Impala only): In Impala, you can rename the view or move it to a different database using this syntax:

ALTER VIEW *db.name* RENAME TO *newdb.newname*

To keep the view in the same database, repeat the same database name, or if the active database is the database which has the view, you can omit the database name entirely. To keep the same view name when changing the database, repeat the same view name.
Dropping Views
You can drop a view in Hive or Impala simply by using **DROP VIEW *dbname.viewname*;** The ***dbname.*** is optional if the view is in the active database.

Dropping views makes no changes to any data in the file system. Any tables used for the underlying query will still have their data.
Try It!

Try modifying and then dropping the craft_information view you created in the previous reading, "Creating and Querying Views." If you didn't do the activities with that reading, go back and use the instructions to create the view before preceding.

First try renaming the view using Impala. Be sure your active database is the one with the view, then use ALTER VIEW craft_information RENAME TO *viewname.* You can rename it whatever you like. Run a SHOW TABLES; command to see verify that the name has changed.

Change the underlying query, using either Impala or Hive (your choice). Make it something simple, such as ALTER VIEW *viewname* AS SELECT DISTINCT carrier FROM flights; Use DESCRIBE *viewname*; to verify that the underlying query has changed.

Finally drop the view. Check that the table you specified in the SELECT statement in the view definition in previous step still has its data!

MATERIALIZED AND NON-MATERIALIZED VIEWS

Views in Hive and Impala are typically ***non-materialized.*** This means that a view does not store or persist any data. It stores only a query.

When you query a view, Hive or Impala generates the result set on the fly by running the view's stored query on the underlying tables, then running your query on the result of the stored query.* It does this each time you run the query. For a complicated and computationally expensive query, this can take some time—that's the major disadvantage of non-materialized views. The major advantage is that whenever the data is changed in any of the underlying tables used in the stored query, the view will use the new data.

A ***materialized*** view, on the other hand, would store the data, so that the SQL engine does not need to run a query on the underlying tables every time the view is queried. This can save time. However, if the data is changed in any of the underlying tables, a materialized view will ***not*** use the new data; you would need to rebuild the view first.

At this time, Impala does not support materialized views, though this is something the developers are considering. (See [IMPALA-3446] for updates on this.) Hive has recently added materialized views with Hive 3.0.0. (See [HIVE-10459] and the Hive Materialized views page.) However, the VM provided for this course uses an older version of Hive, so you cannot create materialized views on the VM.
* This is essentially what happens, but in practice Hive and Impala might optimize this process by combining the operations in your query with the operations in the stored query into one single set of operations so that the result can be generated as efficiently as possible.

The ORDER BY Clause in Views

The stored query for a view can be any query—it can use any of the allowed clauses of a SELECT statement. However, using the ORDER BY clause in a view's stored query is not recommended. Sorting (arranging) result rows in order is an action best performed in the query on the view, not in the view's stored query. To understand why this is, recall that Hive and Impala are designed to distribute query processing work across large clusters of computers. Some tasks (like filtering rows) can easily be performed in parallel on many rows distributed across these many computers. But the task of efficiently ***sorting*** many rows typically requires consolidating all the rows on just one or a few computers. This makes sorting rows a slow and inefficient operation; sorting is typically the bottleneck of any query that uses an ORDER BY clause. Furthermore, preserving the sort order through later query operations forces those later operations also to be slow and inefficient.

So if it is necessary to sort a result set, the sort operation should be performed ***last,*** after the other operations such as filtering. Because the queries ***on*** a view will often perform further operations, including filtering, the query stored ***in*** a view should ***not*** perform sorting; doing this would cause major inefficiencies.

Impala and newer versions of Hive (version 3.0.0 and higher, which is newer than the version on the course VM) prevent these inefficiencies from occurring by ignoring the ORDER BY clause when it is used in a view's stored query. Impala will issue a warning to inform you of this when you query a view that uses ORDER

BY in its stored query. However, some applications do not display this warning. Impala Shell displays it prominently, but Hue's Impala query editor does not; you need to click **Show Logs** to see it in Hue. Some other applications do not display the warning at all.

Newer versions of Hive will silently ignore the **ORDER BY** clause in a view's stored query and will not issue any warning. Older versions of Hive (like the one on the VM) will respect the **ORDER BY** clause in a view's stored query and will incur the associated inefficiencies.

The exception to all of this is when the **ORDER BY** clause is used together with the **LIMIT** clause in a view's stored query. If a view's stored query uses **ORDER BY** and **LIMIT** ***n***, then the sorting operation is much less likely to be a bottleneck, because Hive and Impala can efficiently identify the top ***n*** or bottom ***n*** rows (if ***n*** is fairly small—and it typically is).

So if a view's stored query uses **ORDER BY** together with **LIMIT**, then Impala and newer versions of Hive will ***not*** ignore the **ORDER BY** clause.

Try It!

1. Using Hive (either in Hue or using Beeline on the command line), make the **fly** database your active database.
2. Do a quick **SELECT * FROM planes LIMIT 20;** to see what the first 20 rows of the **planes** table looks like.
3. Create a view of the **planes** table with all the columns, ordering by **year** in descending order but without a **LIMIT** clause (and omitting any rows where **year** is **NULL**). You can name it whatever you like. Here's the syntax, just to remind you:

```
CREATE VIEW viewname AS
  SELECT * FROM planes
    WHERE year IS NOT NULL
    ORDER BY year DESC;
```

4. Query the view just using:

```
SELECT * FROM viewname LIMIT 20;
```

Notice that the result set is indeed sorted by year (all the results should be from 2018). Also note how long it took to finish (this will matter in the next step). The time is reported in the top right of the query window, next to the active database.

5. Query the view again, but this time sort your query by **tailnum:**

SELECT * FROM *viewname* ORDER BY tailnum LIMIT 20;

Notice that the results are not all from 2018, and the query took probably almost twice as long, because it had to sort twice!

6. Now try it in Impala. First, go to Impala in Hue or using Impala Shell on the command line, and make **fly** the active database. Then execute:

INVALIDATE METADATA *viewname*;

so Impala will see the view you created in Hive.

7. Query the view just using:

SELECT * FROM *viewname* LIMIT 20;

a. When you did this in Hive, you got only planes from 2018; what are the results this time?

b. Can you see the warning message indicating that the ORDER BY clause in the view has no effect on the query result? If you are using Hue, click the **Show Logs** button on the upper right; the warning should be visible at the bottom of the logs.

8. You can drop the view if you like.

Improving Query Performance

When you work with large-scale data, you'll often come across the problem of queries that take too long to complete. While they're running, these queries might use an enormous amount of the shared compute resources on your cluster. Part of the issue is just the sheer size of the data-set, but the specifics of a particular query and the different stages that the SQL engine must go through to provide the results can have a significant effect on query performance. In this lesson, you'll learn a few general strategies for improving query performance, darting with choosing which query engine to use. You'll also see how to view the execution plan for processing a query to see where the problem areas might be and you'll learn a few common ways to address these problems. Then in the next two lessons after this one, you'll learn about some more specific strategies for improving query performance in two particular types of cases.

What to Do When Queries Take Too Long

CHOOSING WHICH QUERY ENGINE TO USE

With different query engines available to you, and with different options available in those engines, you'll have to decide which to use for a particular task. Here are some guidelines to help you decide.

First, if you're considering whether you should be using a big data system or to a relational database system, remember that relational databases are optimized to store relatively small amounts of data, to provide immediate query results, and to allow for in-place modification of data. Big data engines such as Hive and Impala, on the other hand, are better optimized for large amounts of read-only data. They provide excellent scalability at low cost. So if you're working with smaller amounts of data (no more than a terabyte or two) and you need in-place modification of data, you probably don't need or even want to use a big data system.

If you do need a big data system, Impala is typically faster than Hive and is good for interactive and ad-hoc queries when you're exploring data.
However, Impala lacks some of the features that Hive provides. For example, a very long-running query that experiences failures (due to computers in the cluster failing, for example) will fail in Impala, but Hive has fault tolerance and likely will still complete the query. This makes Hive a good choice for batch processing and ETL jobs using SQL. Hive also offers extensible record formats and file formats; Impala is more limited in the accepted file formats.

Hive developers have found ways to improve Hive's query performance in recent versions. Hive works by creating jobs that run in a different engine (originally MapReduce, which can be rather slow) and the underlying engine can be changed. Rather than MapReduce, you may be able to use Apache Spark or Apache Tez, both of which are faster than MapReduce. Newer versions of Hive also support an architecture called LLAP (Live Long And Process) which caches metadata similarly to Impala, reducing query latency. You may want to test some typical queries against your own tables to see if one of these works better for you than Impala for interactive and ad-hoc queries.

There are a few other considerations when deciding how to complete tasks in big data systems. While both Hive and Impala can insert individual records into a table, this is not a recommended way to populate a table, because it tends to create small files that are inefficient to process. Recent versions of Hive do include some limited support for updating and deleting records, but full transactions (including **COMMIT** and **ROLLBACK**) are not yet implemented. Hive and Impala do not support stored procedures, as relational databases do. Relational database engines also typically have extensive support for indexing, while Hive has only limited support for indexing, and Impala does not support it. However, if you need the speed for searching massive datasets that indexing provides, other tools such as Cloudera Search, which uses Apache Solr, can be used instead.

You might also be curious about Apache Spark, which is another powerful large-scale data processing engine. It provides APIs for writing custom data processing code, but working directly with it requires programming skill. If you don't already have skills with Spark, Hive and Impala are typically a better choice for data analysis and data processing tasks.

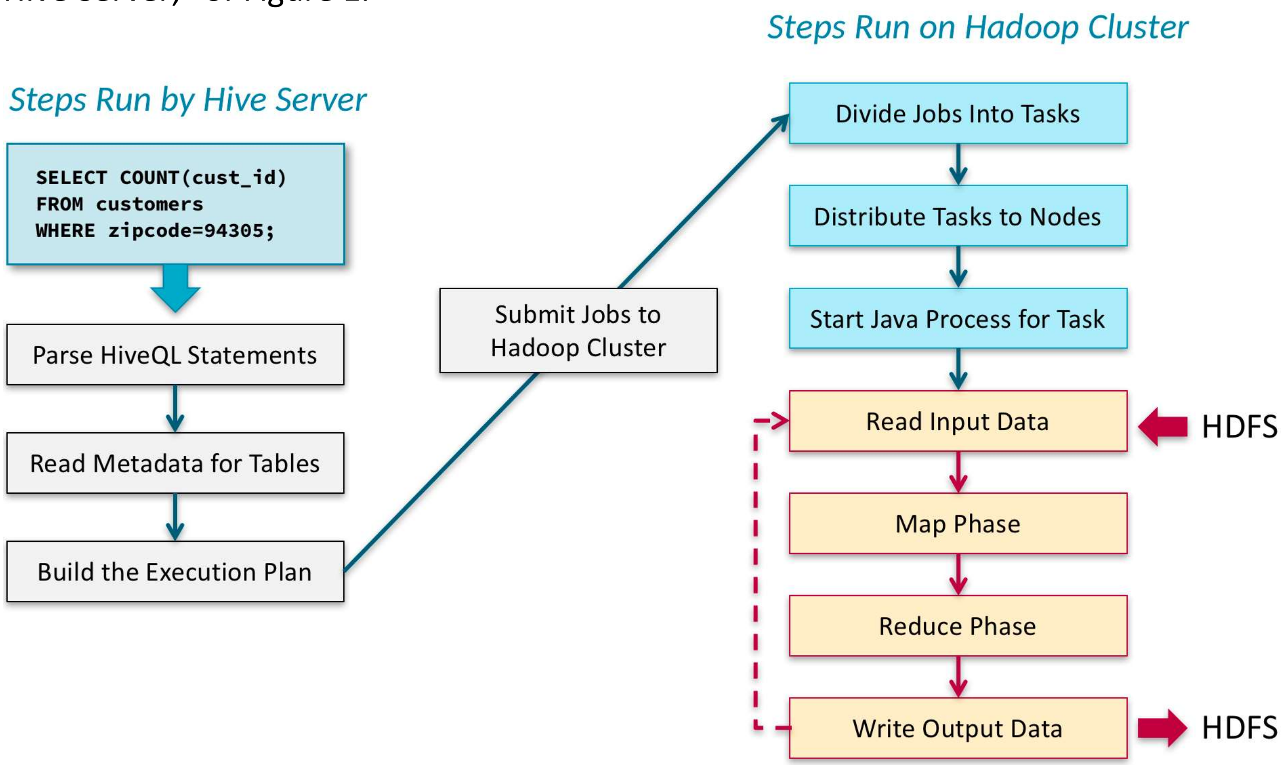
UNDERSTANDING MAP TASKS AND REDUCE TASKS

If you've been using the Hive engine, you may have noticed that some types of queries are significantly slower than others. Hive is designed to hide the complexity of distributed data processing from the user. To use Hive, you need only issue SQL queries. But to understand why some types of queries finish faster than others, you need to know what happens when you run a Hive query.

HIVE QUERY PROCESS

Hive does not have its own data processing engine; instead, it converts a query into one or more jobs that run on the cluster using a different engine. Originally, MapReduce was the exclusive data processing engine for Hive. Newer versions of Hive include support for Apache Spark or Apache Tez as an alternative engine. When you use MapReduce as Hive’s execution engine, this is called ***Hive on MapReduce***. There are similar terms for Spark and Tez. In this reading, and as the default execution engine in the course VM, we're using Hive on MapReduce.
A Hive client application, such as Beeline or Hue, connects to a Hive server. When you run a query from one of these clients, the Hive server performs several operations. It parses the SQL, retrieves metadata from the metastore, and plans

the execution of the query. These steps are relatively fast; they might take only a small fraction of a second for a simple query. See the first column, “Steps Run by Hive Server,” of Figure 1.



```
SELECT COUNT(cust_id)
FROM customers
WHERE zipcode=94305;
```

Figure 1: Hive process overview

Then the Hive server submits one or more jobs to the cluster. On the cluster, the jobs are divided into tasks, the tasks are distributed across nodes, and processes are started to execute the tasks (“Steps Run on Hadoop Cluster” in Figure 1). Tasks are executed in a specific sequence. First the input data is read, then the data is processed through one or more map and reduce phases, and finally the result is generated. The steps that run on the cluster account for a large majority of a query’s total running time.

Map and Reduce Tasks

To understand how a job on the cluster is divided up into tasks, you need to understand how the map-reduce data processing model works. The MapReduce engine used by Hive is an implementation of the map-reduce data processing model.

This model provides a way to divide a large data processing job into a sequence of smaller tasks that can run in parallel across a large number of computers. A MapReduce job is divided into two types of tasks: map tasks and reduce tasks. These tasks are sequenced in phases.

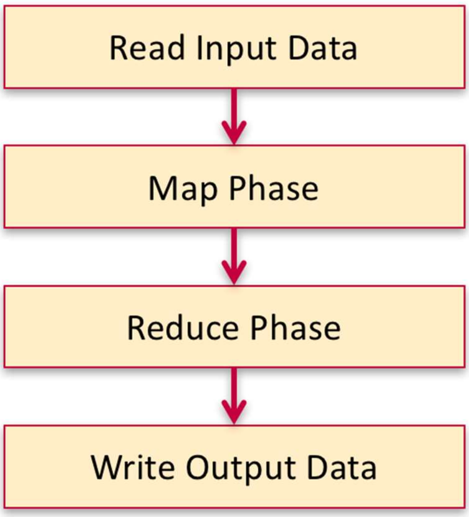


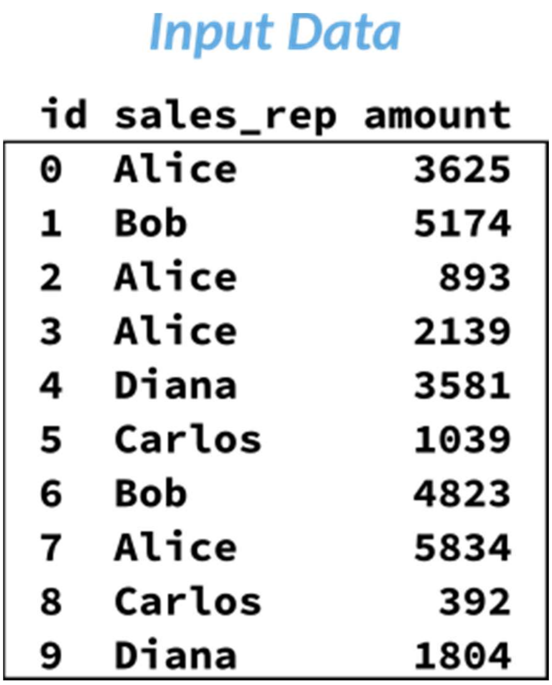
Figure 2: Map-reduce job within the Hive process

A map phase runs first. It is used to filter, transform, or parse data. In a map phase, each record of data is processed independently. The output from a map phase becomes the input to a reduce phase. A reduce phase is used to summarize or aggregate data, combining multiple records together. Some types of jobs don't perform any aggregation so they don't require a reduce phase; these are called map-only jobs.

Example MapReduce Job

This example illustrates how a Hive query executes as a MapReduce job. The query in this example selects data from a table named **order_info**. This table has three columns representing the order ID, the name of the salesperson, and the order amount.

```
SELECT upper(sales_rep), SUM(amount) AS high_sales
   FROM order_info
   WHERE amount > 1000
   GROUP BY upper(sales_rep);
```



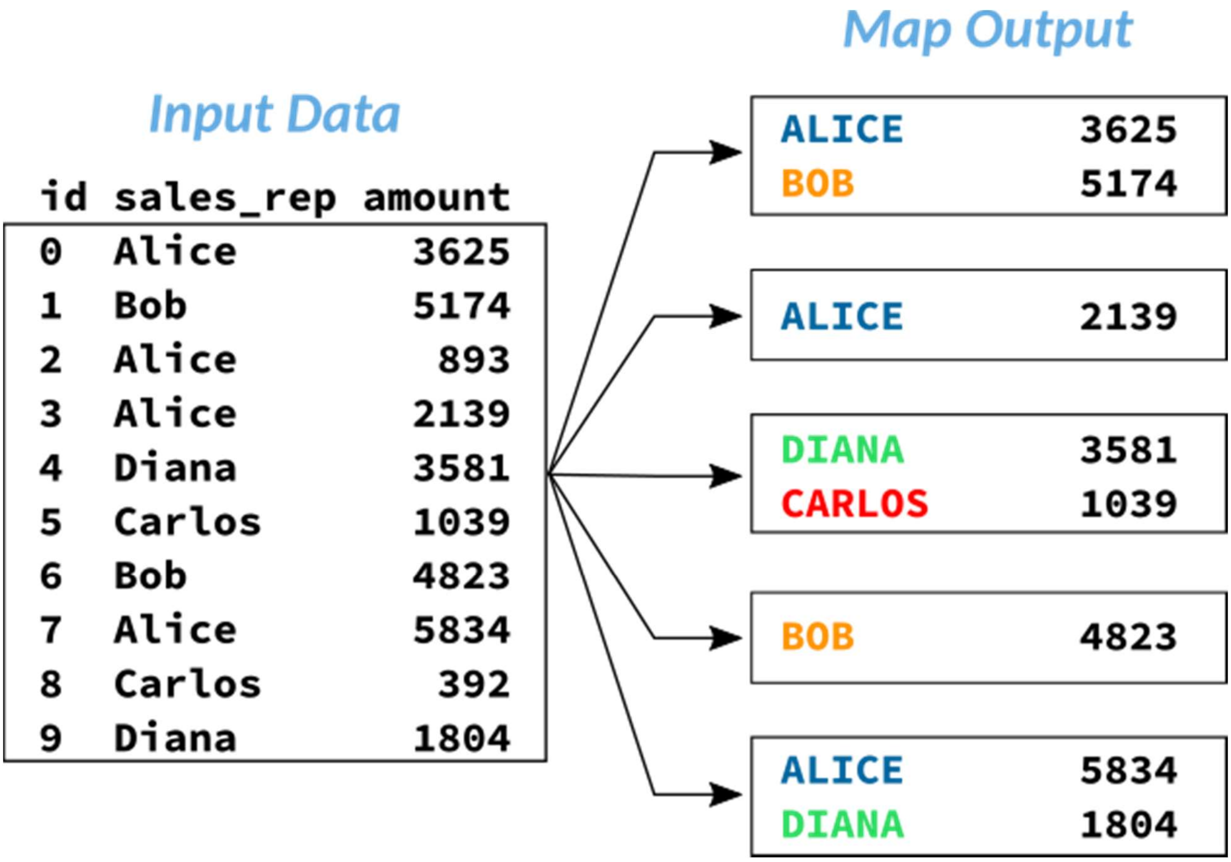
Input Data

id	sales_rep	amount
0	Alice	3625
1	Bob	5174
2	Alice	893
3	Alice	2139
4	Diana	3581
5	Carlos	1039
6	Bob	4823
7	Alice	5834
8	Carlos	392
9	Diana	1804

Figure 3: Example input data

Notice that salespeople can have multiple orders. The query groups by the sales_rep column, adjusted for case sensitivity, and calculates the sum of the order amounts for each salesperson. Executing this query requires both a map phase and a reduce phase.

In the map phase of the MapReduce job, the individual map tasks each receive a portion of the input data. The number of map tasks is determined primarily by the total size of the input data. The example in Figure 4 shows five parallel map tasks, but with a very large input dataset, there could be hundreds or thousands.



id	sales_rep	amount
0	Alice	3625
1	Bob	5174
2	Alice	893
3	Alice	2139
4	Diana	3581
5	Carlos	1039
6	Bob	4823
7	Alice	5834
8	Carlos	392
9	Diana	1804

Figure 4: Map output

The map tasks process the input records. They may filter, transform, or parse the input data. The map tasks can also project the input data, which means returning only a subset of the columns. That's what the map tasks do in this example; each map task simply reads a portion of the input data and outputs the sales_rep and amount fields, discarding the order ID field because it's not needed.

The output from the map tasks goes through an intermediate process called ***shuffle and sort***. This process merges together the output from all the map tasks to create the input to the reduce phase, one input dataset for each reduce task. (See Figure 5.) The process also sorts the data by the column or columns that the data is grouped by, which in this example is the sales_rep column. Notice here that the records for Alice are grouped together, the records for Carlos are grouped together, and so on. But the result is not globally ordered—notice here that Carlos comes before Bob.

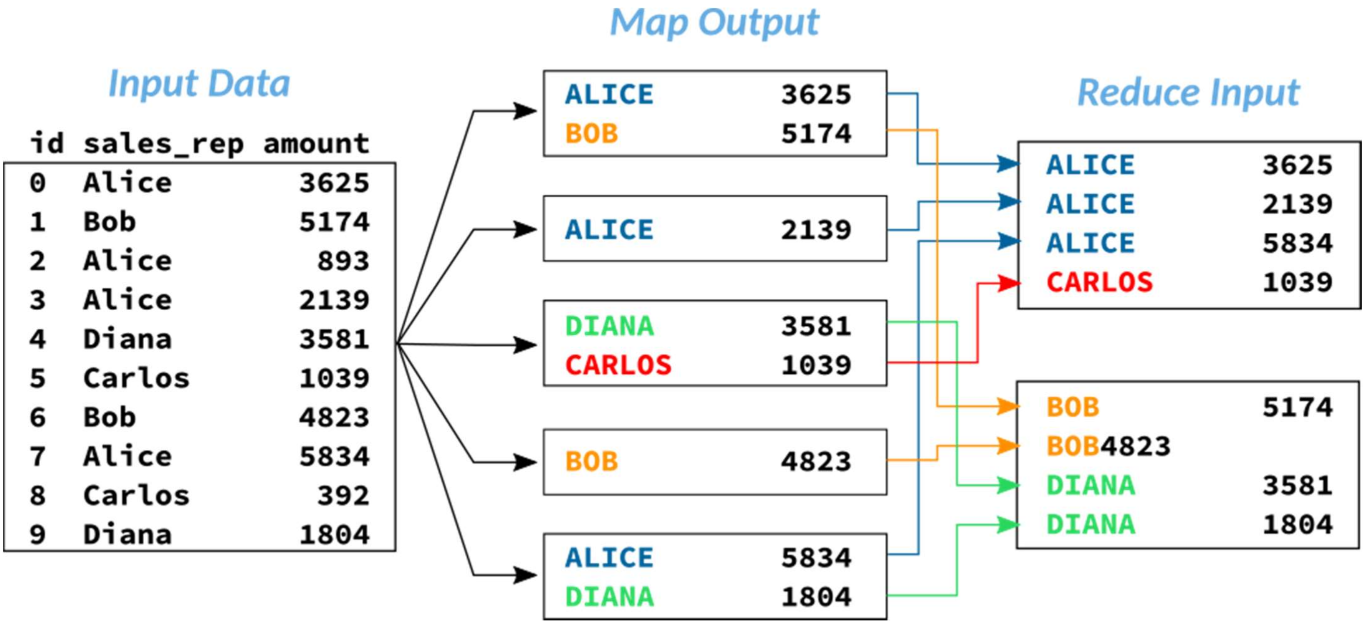


Figure 5: Shuffle and sort

The reduce tasks are where aggregation is performed; in this example, they compute the sum of the order amounts for each salesperson. The number of reduce tasks is determined by the configuration of Hive or MapReduce, and it's almost always much smaller than the number of map tasks. This example (Figure 6) shows two reduce tasks. The outputs from the reduce tasks are appended together to produce the query result.

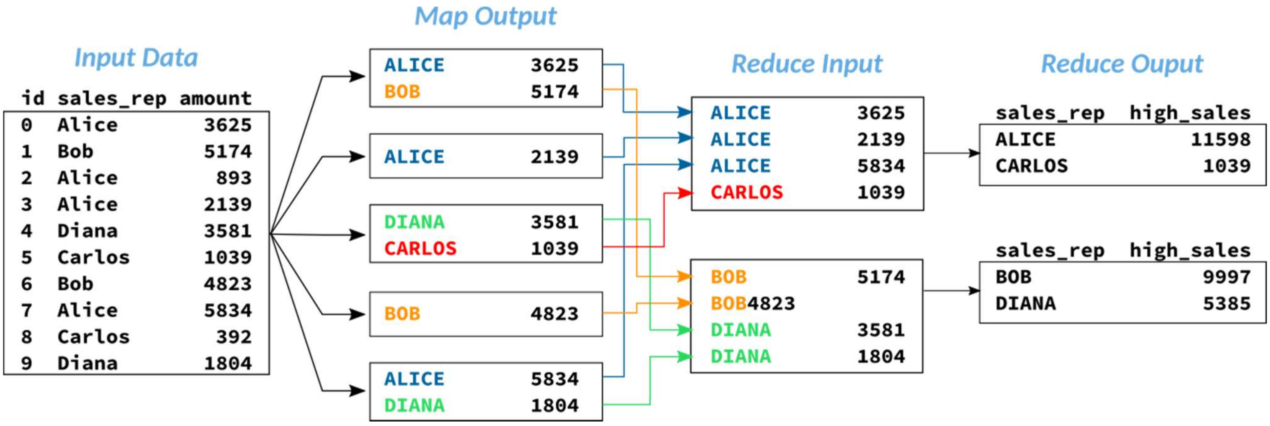


Figure 6: Reduce output

Hive Query Performance Patterns

The speed with which a query completes depends upon what operations Hive must perform to execute the query. The sections below present query patterns in order from the fastest to the slowest.
Understanding these patterns requires an understanding of map and reduce phases, so if you have not yet read “Understanding Map Tasks and Reduce Tasks,” you should do so before continuing. How to know what phases a particular query requires may seem mysterious at this time. With practice, you'll get better at distinguishing map tasks from reduce tasks. The next reading, “Understanding Execution Plans,” will give you some tools to help you figure this out, too.

1. Only Metadata

The fastest type of query requires Hive to retrieve only metadata from the metastore, not data from the file system. An example of this is a **DESCRIBE** command.

```
DESCRIBE customers;
```

2. Fetch Tasks

The next fastest type is a query that executes as a ***fetch task***. These are **SELECT**queries that do not require the underlying data processing engine. The Hive server executes these queries by fetching data directly from the file system and processing it internally. This eliminates the overhead of starting separate processes to execute the job, which reduces query latency.

```
SELECT * FROM customers LIMIT 10;
```

A fetch task can do more than simply fetching the data and returning it, but there are limitations. To execute as a fetch task, a **SELECT** statement must not include the **DISTINCT** keyword and must not use aggregation, windowing, or joins. Also, the input data must be smaller than one gigabyte. Some of these requirements can be changed using Hive configuration properties.

3. Only Map Phase

Next fastest is the type of query that requires only a map phase and no reduce phase. For example, when a query inserts data into another table, Hive executes the query as a map-only job.

```
INSERT INTO TABLE ny_customers
   SELECT * FROM customers
      WHERE state = 'NY';
```

4. Map and Reduce Phases

Slower yet is the type of query that requires both map and reduce phases, such as a query that performs aggregation. To execute this example, Hive projects and filters the data in the map phase, then aggregates it using the COUNT function in the reduce phase.

```
SELECT COUNT(cust_id)
   FROM customers
   WHERE zipcode=94305;
```

5. Multiple Map and Reduce Phases

The slowest type of query is one that requires multiple map and reduce phases. This example is similar to the previous one, but it also sorts the results and returns only the first 10 rows. Executing this query requires a sequence of multiple map and reduce phases.

```
SELECT zipcode, COUNT(cust_id) AS num

   FROM customers

   GROUP BY zipcode
```

```
ORDER BY num DESC

LIMIT 10;
```

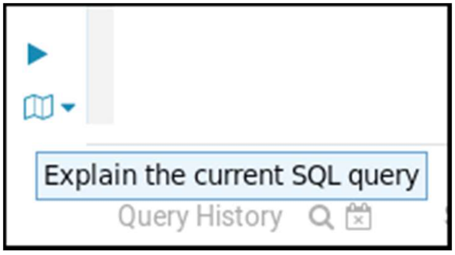
UNDERSTANDING EXECUTION PLANS

To get a true grasp on what causes a query to take a long time, you need to understand what operations Hive or Impala will perform when it executes a query. To find this out, you can view the ***execution plan*** for a query. The execution plan is a description of the tasks required for a query, the order in which they'll be executed, and some details about each task.

To see an execution plan for a query, you can do either of these:

Prefix the query with the keyword EXPLAIN, then run it.

In Hue, simply click the Explain button, which has an icon of a folding map:



Execution plans can be long and complex. Fully understanding them requires a deep knowledge of MapReduce, which is beyond the scope of this course. However, the basics covered here provide a useful introduction that can help you identify trouble areas in your queries' execution plans.

The execution plans provided by Hive and by Impala look slightly different, but at a basic level, they provide more or less the same information. (Hive's execution plans provide much more detail, and understanding it all is beyond the scope of this course.)

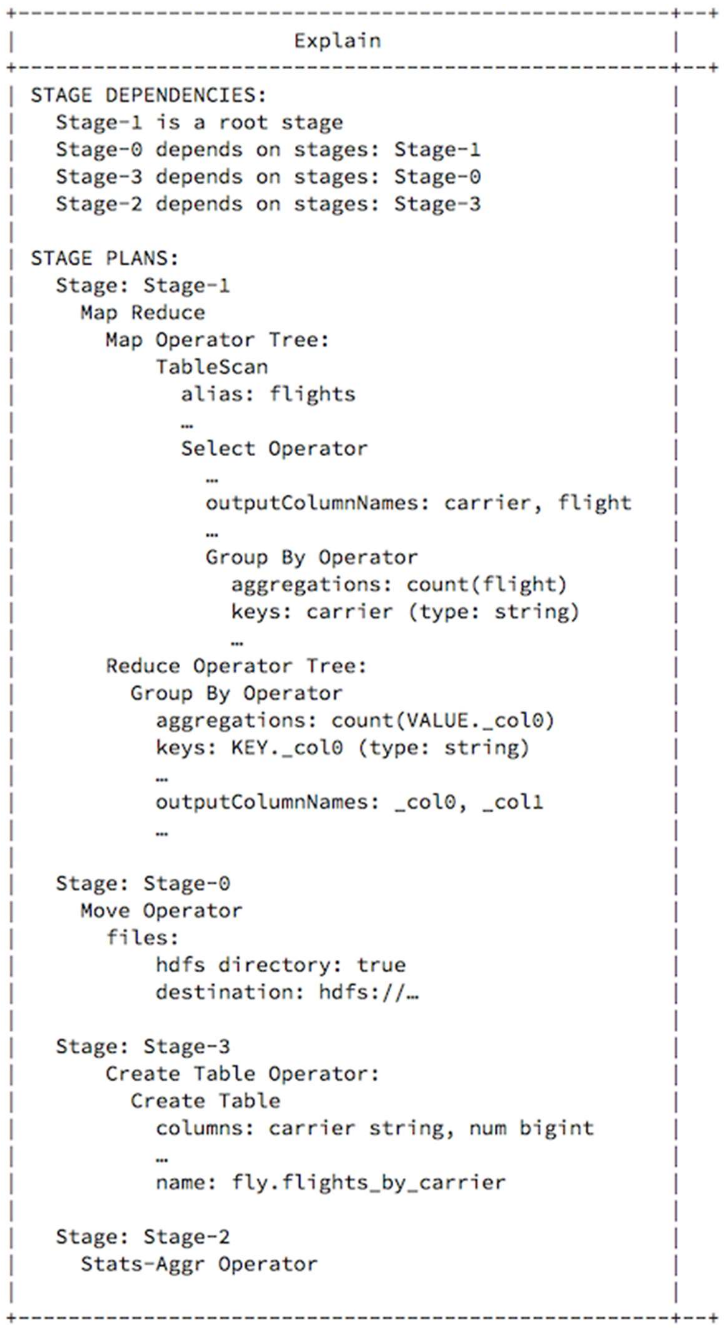
The next several sections show these parts of the query plan for this example query:

```
CREATE TABLE flights_by_carrier AS
  SELECT carrier, COUNT(flight) AS num
    FROM flights GROUP BY carrier;
```

This query is a CTAS statement that creates a new table named flights_by_carrier and populates it with the result of a SELECT query. The SELECT query groups the rows of the flights table by carrier and returns each carrier and the number of flights for that carrier.

HIVE EXECUTION

Hive's output of the EXPLAIN statement for the example is shown here, with some details removed:



```
+----------------------------------------------------+--+
|                      Explain                       |
+----------------------------------------------------+--+
| STAGE DEPENDENCIES:                                |
|   Stage-1 is a root stage                          |
|   Stage-0 depends on stages: Stage-1               |
|   Stage-3 depends on stages: Stage-0               |
|   Stage-2 depends on stages: Stage-3               |
|                                                    |
| STAGE PLANS:                                       |
|   Stage: Stage-1                                   |
|     Map Reduce                                     |
|       Map Operator Tree:                           |
|           TableScan                                |
|             alias: flights                         |
|             …                                      |
|             Select Operator                        |
|               …                                    |
|               outputColumnNames: carrier, flight   |
|               …                                    |
|               Group By Operator                    |
|                 aggregations: count(flight)        |
|                 keys: carrier (type: string)       |
|                 …                                  |
|       Reduce Operator Tree:                        |
|         Group By Operator                          |
|           aggregations: count(VALUE._col0)         |
|           keys: KEY._col0 (type: string)           |
|           …                                        |
|           outputColumnNames: _col0, _col1          |
|           …                                        |
|                                                    |
|   Stage: Stage-0                                   |
|     Move Operator                                  |
|       files:                                       |
|           hdfs directory: true                     |
|           destination: hdfs://…                    |
|                                                    |
|   Stage: Stage-3                                   |
|       Create Table Operator:                       |
|         Create Table                               |
|           columns: carrier string, num bigint      |
|           …                                        |
|           name: fly.flights_by_carrier             |
|                                                    |
|   Stage: Stage-2                                   |
|     Stats-Aggr Operator                            |
|                                                    |
+----------------------------------------------------+--+
```

Stage Dependencies

The example query will execute in four stages, Stage-0 to Stage-3. Each stage could be a MapReduce job, an HDFS action, a metastore action, or some other action performed by the Hive server.

The numbering does not imply an order of execution or dependency. The dependencies between stages determine the order in which they must execute, and Hive specifies these dependencies explicitly at the start of the EXPLAIN results.

A root stage, like Stage-1 in this example, has no dependencies and is free to run first. Non-root stages cannot run until the stages upon which they depend have completed.

Stage Plans

The stage plans part of the output shows descriptions of the stages. For Hive, read them by starting at the top and then going down.

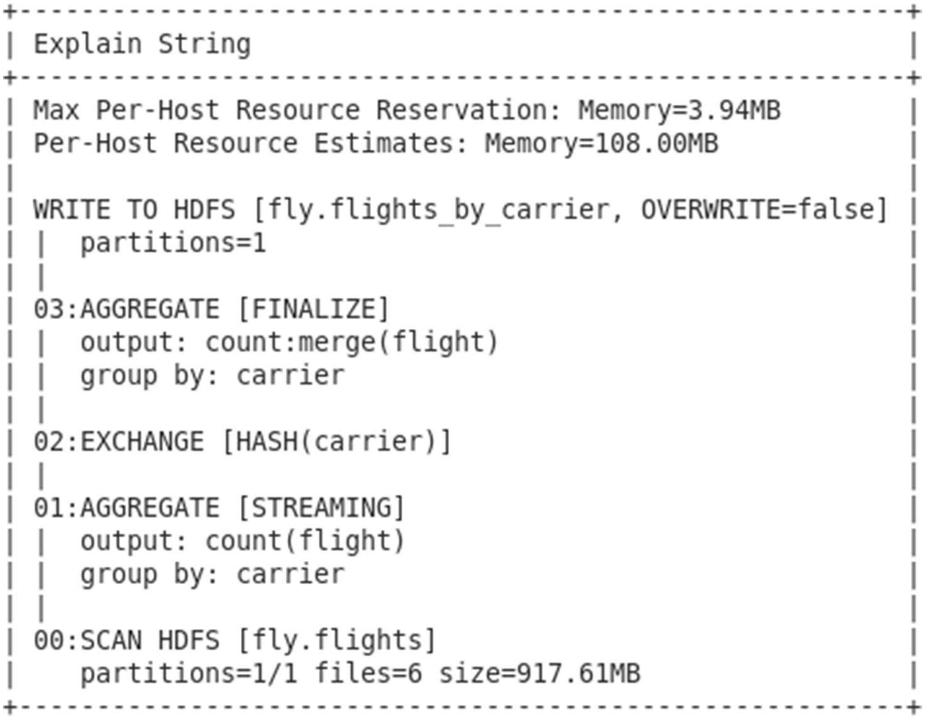
Stage-1 is identified as a MapReduce job. The query plan shows that this job includes both a map phase (described by the Map Operator Tree) and a reduce phase (described by the Reduce Operator Tree). In the map phase, the map tasks read the flights table and select the carrier and flights columns. This data is passed to the reduce phase, in which the reduce tasks group the data by carrier and aggregate it by counting flights.

Following Stage-1 is Stage-0, which is an HDFS action (Move). In this stage, Hive moves the output of the previous stage to a new subdirectory in the warehouse directory in HDFS. This is the storage directory for the new table that will be named flights_by_carrier. (The actual HDFS path is too long to show here.) Following Stage-0 is Stage-3, which is a metastore action: Create Table. In this stage, Hive creates a new table named flights_by_carrier in the fly database. The table has two columns: a STRING column named carrier and a BIGINT column named num.

The final stage, Stage-2, collects statistics. The details of this final stage are not important, but it gathers information such as the number of rows in the table, the number of files that store the table data in HDFS, and the number of unique values in each column in the table. These statistics can be used to optimize Hive queries, but further discussion of that is beyond the scope of this course.

IMPALA EXECUTION

Impala's output of the **EXPLAIN** statement for the example is shown here:



```
+-----------------------------------------------------------+
| Explain String                                            |
+-----------------------------------------------------------+
| Max Per-Host Resource Reservation: Memory=3.94MB          |
| Per-Host Resource Estimates: Memory=108.00MB              |
|                                                           |
| WRITE TO HDFS [fly.flights_by_carrier, OVERWRITE=false]   |
| |  partitions=1                                           |
| |                                                         |
| 03:AGGREGATE [FINALIZE]                                   |
| |  output: count:merge(flight)                            |
| |  group by: carrier                                      |
| |                                                         |
| 02:EXCHANGE [HASH(carrier)]                               |
| |                                                         |
| 01:AGGREGATE [STREAMING]                                  |
| |  output: count(flight)                                  |
| |  group by: carrier                                      |
| |                                                         |
| 00:SCAN HDFS [fly.flights]                                |
|    partitions=1/1 files=6 size=917.61MB                   |
+-----------------------------------------------------------+
```

For Impala's results, the stages are executed from the bottom up (so the first stage is **00** and the final stage is **WRITE TO HDFS**. There are four stages labeled **00** to **03**, and a final unnumbered stage. The numbering normally does not imply execution order, although in this case, the stages will execute in order from **00** to **03**. (Although this example does not show it, additional root stages can be shown within one of the other stages that depends on it, by indenting the line for the root stage.)

The stages are a bit different for Impala; it does not use the map-reduce phases that Hive does with the MapReduce engine. Impala's output is also a bit easier to read.

In Stage 00, Impala reads in the data for the **flights** table in the **fly** database.
In Stage 01, each daemon working on this task groups its data by **carrier** and counts the **flight** column. At this stage, it's probable that a particular carrier's data will be distributed across daemons.

Stage 02 is similar to the shuffle and sort in a MapReduce job. Data is exchanged, using **carrier** to determine which data goes to which daemon.
In Stage 03, the daemons again group the data by carrier and merge the individual counts for each carrier.

The final stage writes the results to the new table, **flights_by_carrier** in the **fly** database.
Try It!
Try running these **EXPLAIN** statements on the example query, in both Hive and Impala. (You do not need to run the query itself.)
See what you can understand from a few other examples. You can try different examples, from something simple (like **SELECT * FROM fun.games;**) to something a bit more complicated, like

```
SELECT COUNT(f.flight) FROM flights f
  JOIN planes p ON (f.tailnum = p.tailnum)
  WHERE p.year < 1968
```

Table and Column Statistics

The SQL engines you use do a certain amount of optimizing of the queries on their own—they look for the best way to proceed with your query, when possible. When the query uses joins, the optimizers can do a better job when they have ***table statistics*** and ***column statistics***. For the table as a whole, these statistics include the number of rows, the number of files used to store the data, and the total size of the data. The column statistics includes the ***approximate*** number of distinct values and the maximum and average sizes of the values (***not*** the maximum or average value, but rather the size used in storage). The optimizers use this information when deciding how to perform the join tasks. Statistics also help your system prevent issues due to memory usage and resource limitations.

These statistics are not automatically calculated—you have to manually trigger it using a SQL command (see below). Once statistics are computed, both Hive and Impala can use them, though if you compute them in Hive, you need to refresh Impala's metadata cache. If you make any changes to the table, such as adding or deleting data, you'll need to recompute the statistics.

Both Hive and Impala can use the statistics, even when calculated by the other machine. However, when you have both Impala and Hive available, Cloudera recommends using Impala's **COMPUTE STATS** command to calculate and view the statistics. The method for Hive (see below) is a bit more difficult to use. If you ***do*** use Hive, you must refresh Impala's metadata cache for the table if you want Impala to use the statistics.

Statistics in Impala

Impala's syntax for calculating statistics for a table (including statistics for all columns) is **COMPUTE STATS *dbname.tablename*;** If the table is in the active database, you can omit ***dbname.*** from the command.

To see the statistics in Impala, run **SHOW TABLE STATS *dbname.tablename*;** or **SHOW COLUMN STATS *dbname.tablename*;**

Note: If the statistics have not yet been computed, **#Rows** for the table shows **-1**. The **#Nulls** statistics for each column will always be **-1**; old versions of Impala would calculate this statistic, but it is not used for optimization, so newer versions skip it.

Statistics in Hive

Hive's syntax for calculating statistics for a table is **ANALYZE TABLE *dbname.tablename* COMPUTE STATISTICS;** If the table is in the active database, you can omit ***dbname.*** from the command. To calculate column statistics, add **FOR COLUMNS** at the end of the command.

To see the table statistics in Hive, run **DESCRIBE FORMATTED *dbname.tablename*;** The **Table Parameters** section will include **numFIles**, **numRows**, **rawDataSize**, and **totalSize**. To see the statistics for

a column, include the column name at the end: **DESCRIBE FORMATTED *dbname.tablename columnname;*** You can only display column statistics one column at a time.
Try It!

1. The **planes** table in the **fly** database has not had any statistics computed for it. In Impala, run **SHOW TABLE STATS fly.planes;** Notice that **#Rows** says **-1**. Then run **SHOW COLUMN STATS** for the same table. Most of the statistics there are also **-1**. (The **INT** types have max and average size of 4, because all integer types have a fixed size.)
2. Use **COMPUTE STATS fly.planes;** to compute the table and column statistics. Check the table and column statistics for these, and note that there is information where there wasn't before. (The **#Null** column will still be all **-1** though, as noted above.)
3. Compare the number of rows from the table statistics to the **#Distinct Values** statistics for the **tailnum** column. Most likely it appears that there are more distinct values in that column than there are rows in the table! This isn't unusual—remember, **#Distinct Values** is an ***approximation*** of the number of distinct values in the column, not an actual count.

Other Strategies for Query Optimization

There are some other techniques for improving Hive and Impala query performance that are not described in this course. Two of these techniques are briefly described below. Further details of these methods are beyond the scope of this course, but you can follow the links below to learn more.

Bucketing (Hive only)

Bucketing is a Hive-only technique that is similar to partitioning. Recall that partitioning is an approach for improving Hive query performance. Partitioning divides a table's data into separate subdirectories based on the values from one or more partition columns, which have a limited number of discrete values. Hive also offers another way of subdividing data, called bucketing.

Bucketing stores data in separate ***files***, not separate subdirectories like partitioning. It divides the data in an effectively random way, not in a predictable way like partitioning. When records are inserted into a bucketed table, Hive computes hash codes of the values in the specified bucketing column and uses these hash codes to divide the records into buckets. For this reason, bucketing is sometimes called ***hash partitioning***. The goal of bucketing is to distribute records evenly across a predefined number of buckets. Bucketing can improve the performance of joins if all the joined tables are bucketed on the join key column. For more on bucketing, see the page of the Hive Language Manual describing bucketed tables, at https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL+BucketedTables.

INDEXING (HIVE ONLY)

If you have worked with relational databases, you may be familiar with ***indexes***. Indexes can greatly improve the speed of queries that search for specific values in certain columns. By indexing the columns you're filtering by, you can avoid the need to do a full table scan at query time. However, relational database implementations of indexing typically depend on the database system controlling all data that is added to tables. Since Hive and Impala do not work this way, the use of indexing would not confer the same benefits with Hive and Impala.
Early versions of Hive (but not Impala) include a limited implementation of indexing. As in relational databases, indexes in Hive can improve the speed of some queries. However, the speedup from indexing is typically not as dramatic, and building and maintaining indexes with Hive has high costs in terms of disk space and CPU utilization. In fact, indexing is no longer supported in Hive as of version 3.0.0. Cloudera recommends using Cloudera Search (an implementation of Apache Solr) if you need extensive indexing.

For more on indexing, see the page of the Hive Language Manual describing indexing,
at https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Indexing.

TABLE PARTITIONING

Table partitioning, is one approach for improving Hive and Impala query performance. Recall that the data for Hive and Impala tables typically is stored in a file system like HDFS or S3. Breach table, there is a single directory in the file system containing the files that store that tables data. Typically, there are no sub-directories within a table storage directory and all the data files are stored directly in the storage directory. But this poses a problem. If the table contains a huge amount of data, stored across many files, then it can take the query engine an awfully long time to scan through all those files. Hive and Impala are pretty good at scanning files quickly, especially if they're in an optimized file format like Parquet. But still, when the data gets very large, queries can become slow and inefficient. Table partitioning offers help with this problem. It divides a table's data into multiple subdirectories within the tables directory.

The data is divided into these subdirectories when it is loaded. Each record is stored in a partition subdirectory, based on the values of one or more columns called partition columns. Then when you run a query that filters on a partition column, the query does not need to scan all the tables data, it only needs to scan the relevant partition subdirectories. This allows the query to run faster. A query that does not filter on a partition column, will still need to scan all the data, so it will not be any faster. But to be clear, partitioning does not prevent you from running any query that you could run on a non-partitioned table. When used appropriately, table partitioning can greatly improve the performance of commonly used queries. But in some cases, partitioning may not be worthwhile. In other cases, it may actually worsen query performance.

So it's important to understand when table partitioning is appropriate. Typically, partitioning is a good idea under the following criteria. If the table is very large, an Hive and Impala will necessarily take a long time to scan all the data in the tables directory for all your queries. Table partitioning can allow Hive and Impala to scan only parts of the data, resulting in improved performance. Your table partitioning will favor certain queries. You want to know that these are queries that you will tend to run frequently. The queries that will be helped by table partitioning, are those in which you filter two specific values on a certain column or columns. This is how you'll choose to organize your table into partitions by that column or columns.

Don't worry if this is not perfectly clear right now. You will see working examples in the lesson. Third, the partition column should have a reasonable number of different values, and not to make partitioning worthwhile, but not so many that queries become inefficient. For example, you would not want a customer table to be partitioned on different values of say customer ID, because then you would have many partitions each with very little data. This huge number of partition directories would greatly penalize the performance of all your queries. Under these criteria, partitioning can be appropriate. If the criteria I've given here do not apply, you should think twice about whether to use partitioning. In a partition table, files are stored in different directories based on different values of some categorical variable like transaction date, or customer region. Since a partition table will organize data this way, it can be especially sensible to use table partitioning if your data processes already generate files that are divided by category. For example, maybe you receive log records from a log web server, with a different set of files for each new date. It's easy to place these files into a table partitioned by date. Having this kind of organization, can help provide great performance benefits for many of your analytic queries. Another aspect of the organization of partition tables is that one, or perhaps more than one column in your table, will not be stored in the data files at all. But instead will be the tag for different subdirectories where your files are stored.

For example, if you receive a set of files for sales transactions in your northeast region, and a second set of files for your northwest region, you can easily place these files into partitions, where one subdirectory has data for your table column region equal to northeast. There was a different subdirectory for the column region, equal to northwest. You can set up these directories, and then easily run queries for distinct regions and their partition table will help with your query performance. If you choose partition tables, you will transform your data so that it will be organized to meet these two criteria I've just given, whether or not the data comes to you organized that way originally, though it can make your table setup simpler if these criteria are met, but it is not critical that these criteria are met by your data to begin with. It's important to avoid partitioning data into numerous small files because this will worsen query performance instead of improving it.

This small files problem occurs when their partition columns contain too many unique values. An example of a poor choice for partition column is, first name. There could be thousands of different first names in a table of customers. In the remainder of this lesson, you will learn how to create and load data into partition tables. You'll learn more about the risk of using partitioning. A final note to anyone who's familiar with partitioning from the world of relational database systems. Many relational databases do support other types of table partitioning. Such as range, hash, and list partitioning. In general, Hive and Impala do not support these. There are some ways to achieve these more specialized types of partitioning with Hive and Impala, but that topic is outside the scope of this course.

When to Use Table Partitioning

To create a partitioned table, use the **PARTITIONED BY** clause in the **CREATE TABLE** statement. The names and types of the partition columns must be specified in the **PARTITIONED BY** clause, and only in the **PARTITIONED BY** clause. They must not also appear in the list of all the other columns.

```
CREATE TABLE customers_by_country
    (cust_id STRING, name STRING)
  PARTITIONED BY (country STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

The example **CREATE TABLE** statement shown above creates the table **customers_by_country**, which is partitioned by the **STRING** column named **country**. Notice that the **country** column appears only in the **PARTITIONED BY** clause, and not in the column list above it. This example specifies only one partition column, but you can specify more than one by using a comma-separated column list in the **PARTITIONED BY** clause. Aside from these specific differences, this **CREATE TABLE** statement is the same as the statement used to create an equivalent non-partitioned table.

Table partitioning is implemented in a way that is mostly transparent to a user issuing queries with Hive and Impala. A partition column is what’s known as a ***virtual column***, because its values are not stored within the data files. Following is the result of the **DESCRIBE** command on **customers_by_country**; it displays the partition column **country** just as if it were a normal column within the table. You can refer to partition columns in any of the usual clauses of a **SELECT** statement.

```
name    type   comment
cust_id string
name    string
country string
```

Note: In the previous lesson, you learned about using COMPUTE STATS in Impala and ANALYZE TABLE … COMPUTE STATISTICS in Hive. If the table is partitioned, Impala cannot use Hive-generated column statistics, so for partitioned tables, it's best to compute the statistics with the engine you'll be using to query the table.

Try It!

Use the CREATE TABLE command above to create a customers_by_country table on your VM, then use DESCRIBE to show the columns. Notice that it's just as described above: there is no apparent difference between the partition column and the other columns.

You'll use this table in the next readings too, so don't drop it yet.

CREATING PARTITIONED TABLES

Loading Data with Dynamic Partition

One way to load data into a partitioned table is to use ***dynamic partitioning,*** which automatically defines partitions when you load the data, using the values in the partition column**.**(The other way is to manually define the partitions. See "Loading Data with Static Partitioning" for this method.)

To use dynamic partitioning, you must load data using an INSERT statement. In the INSERT statement, you must use the PARTITION clause to list the partition columns. The data you are inserting must include values for the partition columns. The partition columns ***must*** be the rightmost columns in the data you are inserting, and they must be in the same order as they appear in the PARTITION clause.

```
INSERT OVERWRITE TABLE customers_by_country
   PARTITION(country)
   SELECT cust_id, name, country FROM customers;
```

The example shown above uses an INSERT … SELECT statement to load data into the customers_by_country table with dynamic partitioning. Notice that the partition column, country, is included in the PARTITION clause and is specified last in the SELECT list.

When Hive or Impala executes this statement, it automatically creates partitions for the country column and loads the data into these partitions based on the values in the country column. The resulting data files in the partition subdirectories do not include values for the country column. Since the country is known based on which subdirectory a data file is in, it would be redundant to include country values in the data files as well.

Note: Hive includes a safety feature that prevents users from accidentally creating or overwriting a large number of partitions. (See "Risks of Using Partitioning" for more about this.) By default, Hive sets the property hive.exec.dynamic.partition.mode to strict. This prevents you from using dynamic partitioning, though you can still use static partitions.
You can disable this safety feature in Hive by setting the property hive.exec.dynamic.partition.mode to nonstrict:

```
   SET hive.exec.dynamic.partition.mode=nonstrict;
```

Then you can use the INSERT statement to load the data dynamically.
Hive properties set in Beeline are for the current session only, so the next time you start a Hive session this property will be set back to strict. Your system administrator can configure properties permanently, if necessary.

Try It!
If you did not create the customers_by_country table in the "Creating Partitioned Tables" reading, do so before continuing. If you did the exercises in the "Loading Data with Static Partitioning" reading before doing this one, drop the table and recreate it (without loading the data).
1. First, look at the contents of the customers_by_country table directory in HDFS. (Where this directory exists depends on which database holds the table.) You can use Hue or an hdfs dfs -ls command to list the contents of the directory. Since you haven't loaded any data, it should be empty.
2. If you want to use Hive, disable the partition safety feature by running

```
   SET hive.exec.dynamic.partition.mode=nonstrict;
```

   If you want to use Impala for the rest of these exercises, you don't need to run that command.
3. The customers table has only four rows, and each has a different code in the country column. Use the following command to insert the data into the partitioned customers_by_country table:

```
   INSERT OVERWRITE TABLE customers_by_country
```

```
    PARTITION(country)
    SELECT cust_id, name, country FROM default.customers;
```

4. Look at the contents of the customers_by_country directory. It should now have one subdirectory for each value in the country column.
5. Look at the file in one (or more, if you like) of those directories, using Hue or an hdfs dfs -cat command. Notice that the file contains the row for the customer from that country, and no others; notice also that the country value is not included.
6. Run some SELECT queries on the partitioned table. Try one that does no filtering (like SELECT * FROM customers_by_country;) and one that filters on country. It's a small table so there won't be a significant difference in the time it takes to run; the point is to notice that you will not query the table any differently than you would query the customers table.

Loading Data with Static Partitioning

One way to load data into a partitioned table is to use ***static partitioning,*** in which you manually define the different partitions**.**(The other way is to have the partitions automatically defined when you load the data. See “Loading Data with Dynamic Partitioning” for this method.)

With static partitioning, you create a partition manually, using an ALTER TABLE … ADD PARTITION statement, and then load the data into the partition.

For example, this ALTER TABLE statement creates the partition for Pakistan (pk):

```
  ALTER TABLE customers_by_country
    ADD PARTITION (country='pk');
```

Notice how the partition column name, which is country, and the specific value that defines this partition, which is pk, are both specified in the ADD PARTITION clause. This creates a partition directory named country=pk inside the customers_by_country table directory.

After the partition for Pakistan is created, you can add data into the partition using an INSERT … SELECT statement:

```
  INSERT OVERWRITE TABLE customers_by_country
    PARTITION(country='pk')
    SELECT cust_id, name FROM customers WHERE country='pk'
```

Notice how in the PARTITION clause, the partition column name, which is country, and the specific value, which is pk, are both specified, just like in the ADD PARTITION command used to create the partition. Also notice that in the SELECT statement, the partition column is not included in the SELECT list. Finally, notice that the WHERE clause in the SELECT statement selects only customers from Pakistan.

With static partitioning, you need to repeat these two steps for each partition: first create the partition, then add data. You can actually use any method to load the data; you need not use an INSERT statement. You could instead use hdfs dfs commands or a LOAD DATA INPATH command. But however you load the data, it's your responsibility to ensure that data is stored in the correct partition subdirectories. For example, data for customers in Pakistan must be stored in the Pakistan partition subdirectory, and data for customers in other countries must be stored in those countries' partition subdirectories.

Static partitioning is most useful when the data being loaded into the table is already divided into files based on the partition column, or when the data grows in a manner that coincides with the partition column: For example, suppose your company opens a new store in a different country, like New Zealand ('nz'), and you're given a file of data for new customers, all from that country. You could easily add a new partition and load that file into it.

Try It!

If you did not create the customers_by_country table in the "Creating Partitioned Tables" reading, do so before continuing. If you did the exercises in the "Loading Data with Dynamic Partitioning" reading before doing this one, drop the table and recreate it (without loading the data). (Hint: In Hue, you probably don't have to retype the command, it should be in your Query History. Find it and click on it.)

1. First, look at the contents of the customers_by_country table directory in HDFS. (Where this directory exists depends on which database holds the table.) You can use Hue or an hdfs dfs -ls command to list the contents of the directory. Since you haven't loaded any data, it should be empty.
2. Add the partition for Pakistan (pk) using the ALTER TABLE command:

```
ALTER TABLE customers_by_country
   ADD PARTITION (country='pk');
```

3. Check the contents of the customers_by_countrytable directory in HDFS and see that it now has a subdirectory for the partition you just created.
4. Now load ***only*** the customers from Pakistan into that partition:

```
INSERT OVERWRITE TABLE customers_by_country
  PARTITION(country='pk')
  SELECT cust_id, name FROM default.customers WHERE country='pk';
```

5. ***Optional:*** Modify and run both commands to create a partition for one of the other countries (us, ja, or ug) and load the data from the customers table into that partition. You can do it for all three if you like. Check that the customers_by_country directory has one subdirectory for each partition you added.
6. Look at the file in one (or more, if you like) of those directories, using Hue or an hdfs dfs -cat command. Notice that the file contains the row for the customer from that country, and no others; notice also that the country value is not included (because you didn't include it in the SELECT list).
7. Run some SELECT queries on the partitioned table. Try one that does no filtering (like SELECT * FROM customers_by_country;) and one that filters on country. It's a small table so there won't be a significant difference in the time it takes to run; the point is just to notice that you will not query the table any differently than you would query the customers table.

Risks of Using Partitioning

A major risk when using partitioning is creating partitions that lead you into the small files problem. When this happens, partitioning a table will actually worsen query performance (the opposite of the goal when using partitioning) because it causes too many small files to be created. This is more likely when using dynamic partitioning, but it could still happen with static partitioning—for example if you added a new partition to a **sales** table on a daily basis containing the sales from the previous day, and each day’s data is not particularly big.

When choosing your partitions, you want to strike a happy balance between too many partitions (causing the small files problem) and too few partitions (providing performance little benefit). The partition column or columns should have a reasonable number of values for the partitions—but what you should consider *reasonable* is difficult to quantify.

Using dynamic partitioning is particularly dangerous because if you're not careful, it's easy to partition on a column with too many distinct values. Imagine a use case where you are often looking for data that falls within a time frame that you would specify in your query. You might think that it's a good idea to partition on a column that pertains to time. But a **TIMESTAMP** column could have the time *to the nanosecond,* so every row could have a unique value; that would be a terrible choice for a partition column! Even to the minute or hour could create far too many partitions, depending on the nature of your data; partitioning by larger time units like day, month, or even year might be a better choice.

As another example, consider the **default.employees** table on the VM. This has five columns: **empl_id**, **first_name**, **last_name**, **salary**, and **office_id**. Before reading on, think for a moment, which of these might be reasonable for partitioning (assuming the table will eventually be much larger than the five rows in our sample table)?

The column **empl_id** is a unique identifier. If that were your partition column, you would have a separate partition for each employee, and each would have exactly one row. In addition, it's not likely you'll be doing a lot of queries looking for a particular value, or even a particular range of values. This is a poor choice.
The column **first_name** will not have one per employee, but there will likely be many columns that have only one row. This is also true for **last_name**. Also, like **empl_id**, it's not likely you'll need filter queries based on these columns. These are also poor choices.

The column **salary** also will have many divisions (and even more so if your salaries go to the cent rather than to the dollar as our sample table does). While it may be that you'll sometimes want to query on salary ranges, it's not likely you'll want to use individual salaries. So **salary** is a poor choice. A more limited **salary_grades** specification, like the ones in the **salary_grades** table, might be reasonable *if* your use case involves looking at the data by salary grade frequently.

The **office_id** column identifies the office where an employee works. This will have a much smaller number of unique values, even if you have a large company with offices in many cities. It's imaginable that your use case might be to frequently filter your employee data based on office location, too. So this would be a good choice.

You also can use multiple columns and create nested partitions. For example, a dataset of customers might include **country** and **state_or_province** columns. You can partition by **country** and then partition those further by **state_or_province**, so customers from Ontario, Canada would be in the **country=ca/state_or_province=on/** partition directory. This can be extremely helpful for large amounts of data that you want to access either by country or by state or province. However, using multiple columns increases the danger of creating too many partitions, so you must take extra care when doing so.
The risk of creating too many partitions is why Hive includes the property **hive.exec.dynamic.partition.mode**, set to **strict** by default, which must be reset to **nonstrict** before you can create a partition. (See the note about this, near the end of the “Loading Data Using Dynamic Partitioning” reading.) Rather than automatically and mechanically resetting that property when you're about to load data dynamically, take it as an opportunity to think about the partitioning columns and maybe check the number of unique values you would get when you load the data.

COMPLEX DATA AND DENORMALIZATION

When to Use Complex Columns

A particularly computationally expensive class of queries for Hive and Impala is joins. Queries with joins can take a long time. The main strategy for making these queries faster is to actually eliminate the need to join multiple tables together by denormalizing tables and storing data in complex columns where suitable. Hive and Impala support the use of three complex column types, array, map, and struct. These column types combine multiple values into a single column. While avoiding joins and improving performance is our main motivation to look at complex column types now, there are other reasons why you might want to use complex types. For example, complex types help organize related data. You can store related fields in a single column rather than multiple columns with similar column names. Complex types afford flexibility. They allow you to store an arbitrary amount of data in a single row. Sometimes the data files you want to query already contain complex or nested structures. This is common for data produced by tools like Apache Pig and Spark, and languages like Java and Python. By using Hive and Impala's complex types, you can avoid the need to flatten these nested structures. Although these three column types are the same in Hive and Impala, these two engines use different syntax to access them. The readings in this lesson we'll explain the types and then provide guidance in using them in both engines.

Complex Data Types

Recall that Hive and Impala support a number of simple data types, similar to the types found in relational databases. These simple data types represent a single value within a single row-column position.

In addition, Hive and Impala also support several ***complex data types***, which represent multiple values within a single row-column position. Complex types are also referred to by several other names, including ***nested types*** and ***collection types***.

Hive and Impala both support three different complex data types: ARRAY, MAP, and STRUCT.

ARRAY

An ARRAY represents an ordered list of values, all having the same data type. For example, people often have multiple phone numbers, such as home (landline), work, and mobile. An array could hold several phone numbers. In the table below, the column phones is an ARRAY in which each element is a STRING:

name	**phones**
Alice	**[555-1111, 555-2222, 555-3333]**
Bob	**[555-4444]**
Carlos	**[555-5555, 555-6666]**

The elements of an ARRAY can be other simple data types, but all elements of an ARRAY must be of the same type.

MAP

A MAP represents key-value pairs, with all keys having the same data type, and all values having the same type. With the phones example, this allows you to specify which phone number is for what purpose (such as home, work, or mobile):

name	phones
Alice	{home:555-1111, work:555-2222, mobile:555-3333}
Bob	{mobile:555-4444}
Carlos	{work:555-5555, home:555-6666}

Here the key is a STRING and the value is also a STRING. Each could be other simple data types; for example, if you don't use the dash in the phone numbers, you could make the key STRING and the value INT.

STRUCT

A STRUCT represents named fields, which can have different data types. For example, you could use a STRUCT to store addresses, with each part of the address a different field:

name	address
Alice	**{street:742 Evergreen Terrace, city:Springfield, state:OR, zipcode:97477}**
Bob	**{street:1600 Pennsylvania Ave NW, city:Washington, state:DC, zipcode:20500}**
Carlos	**{street:342 Gravelpit Terrace, city:Bedrock}**

Here, the STRUCT is defined to have four fields: street, city, and state are STRING types; zipcode is an INT type (though it could also be STRING). Notice that Carlos's address is missing the state and zipcode fields. When this table is queried (see the next readings), those fields would show as NULL.

Nested Complex Types

It's also possible to ***nest*** complex types, for example, to have an ARRAY in which each element is an ARRAY, or a MAP for which the value is a STRUCT element. For the tables above, you might create a contacts column which is a STRUCT with two named fields: phones is a MAP and address is another STRUCT.

As you'll see in the next readings, working with a single layer of complex data can be difficult; working with a nested layer will be much more difficult. If you do need to use nested complex types, we recommend using no more than one nested layer. If you find yourself using more (for example, an ARRAY whose elements are MAPs, and the values of that MAP are themselves ARRAYs), consider whether a different schema design could provide the same information in a more digestible way.

CREATING TABLES WITH COMPLEX DATA

The syntax for creating tables that use complex data types is very similar in Hive and Impala, but mostly you will be creating tables in Hive. The reason for that is ***Impala only supports the use of complex data in Parquet files***, and you cannot load complex data into a table using INSERT or LOAD statements in Impala. If you don't have the data file in Parquet format, you can create the table in Hive, then create a copy using CREATE TABLE … AS SELECT, with STORED AS PARQUET. You then can query the table in Impala.

The examples below assume you are using text files to store the data, so the delimiters are specified in the ROW FORMAT clause of the CREATE TABLE statement. For file formats such as Parquet and Avro, you do not use the ROW FORMAT clause; the details of how these formats represent complex values are determined by the file format, not by the user.

If you're using Impala to create the tables, you ***must*** be using Parquet files, so Impala will never use the ROW FORMAT clause for tables with complex data, and you will need to specify STORED AS PARQUET. The CREATE TABLE statements otherwise will be the same as the examples shown here.

(Querying data can be very different, though, so the next two readings will cover basic queries in both engines, one at a time.)

ARRAY

An ARRAY type is declared in the column list of the CREATE TABLE statement using ARRAY<*type*>, where *type* is the simple data type that each element of the array will have.

The following shows the contents of a data file called customers_phones_array.csv with three columns: cust_id, name, and phones. This will be used as the data for a table using an ARRAY data type for phones using ARRAY<STRING> because the phone numbers are given as STRINGs.

```
a,Alice,555-1111|555-2222|555-3333
b,Bob,555-4444
c,Carlos,555-5555|555-6666
```

Commas separate the customer ID, the customer name, and a list of their phone numbers. The phone numbers themselves are separated using the pipe character (the vertical bar). Both delimiters need to be declared in the CREATE TABLE statement:

```
CREATE TABLE customers_phones_array
    (cust_id STRING,
    name STRING,
    phones ARRAY<STRING>)
  ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    COLLECTION ITEMS TERMINATED BY '|';
```

Recall that if you omit the FIELDS TERMINATED BY subclause, Hive and Impala use the default delimiter, which is the ASCII Control-A character. For the COLLECTION ITEMS TERMINATED BY subclause, the default collection item terminator is the ASCII Control-B character.

Remember: If you're creating the table using Parquet or Avro data files (and, again, Parquet is the only format Impala supports with complex data) omit the ROW FORMAT clause and the subclauses specifying the terminators, and include a STORED AS clause.

MAP

A MAP type is declared in the column list of the CREATE TABLE statement using MAP<*keytype, valuetype*>. Notice that the keys—in the phones example, this would be home, work, and mobile—are not defined in the CREATE TABLE statement. This means new keys could be added to the data without updating the table definition.

The following shows the contents of a data file called customers_phones_map.csv with the same three columns as in the ARRAY example: cust_id, name, and phones. In this case, though, phones will use MAP<STRING,STRING> because both the key (type of number) and the value (the phone number itself) are both given as STRINGs.

```
a,Alice,home:555-1111|work:555-2222|mobile:555-3333
b,Bob,mobile:555-4444
c,Carlos,work:555-5555|home:555-6666
```

Again, as with the ARRAY example, commas separate the customer ID, the customer name, and the list of their phone numbers. The key-value pairs are separated using the pipe character (the vertical bar) again, and colons are used to separate the key from the value in each pair. All three delimiters need to be declared in the CREATE TABLE statement:

```
CREATE TABLE customers_phones_map
    (cust_id STRING,
    name STRING,
    phones MAP<STRING,STRING>)
  ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    COLLECTION ITEMS TERMINATED BY '|'
    MAP KEYS TERMINATED BY ':';
```

The default terminators are the same as with ARRAY, but now you also have the MAP KEYS terminator. The default (if you omit the MAP KEYS TERMINATED BY subclause) is the ASCII Control-C character.

Remember: If you're creating the table using Parquet or Avro data files (and, again, Parquet is the only format Impala supports with complex data) omit the ROW FORMAT clause and the subclauses specifying the terminators, and include a STORED AS clause.

STRUCT

A STRUCT type is declared in the column list of the CREATE TABLE statement using STRUCT<*field1:TYPE1, field2:TYPE, …*>. The order of the STRUCT fields in the table definition ***must*** match the order in the data files.

The following shows the contents of a data file called customers_addr.csv with the three columns: cust_id, name, and address. Here, address will use a STRUCT type with four named fields: street, city, state, and zipcode. All are STRINGs except zipcode, which is an INT.

```
a,Alice,742 Evergreen Terrace|Springfield|OR|97477
b,Bob,1600 Pennsylvania Ave NW|Washington|DC|20500
c,Carlos,342 Gravelpit Terrace|Bedrock
```

A STRUCT contains a predefined number of named fields, but fields can be missing. In this example, Carlos's address is missing the state and zipcode fields, so queries will return NULL for these missing fields.

Again, as with the ARRAY example, commas separate the columns. The fields in the STRUCT are separated using the pipe character (the vertical bar). Both delimiters need to be declared in the CREATE TABLE statement:

```
CREATE TABLE customers_addr
      (cust_id STRING,
      name STRING,
      address STRUCT<street:STRING,
                     city:STRING,
                     state:STRING,
                     zipcode:INT>)
  ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    COLLECTION ITEMS TERMINATED BY '|';
```

The default terminators are the same as with ARRAY.

Note that unlike MAPs, the “keys” of a STRUCT (the names of its fields) are ***not*** part of the actual data. So if we changed the name from zipcode to postalcode, we would ***not*** need to update the underlying data in the data file.

Remember: If you're creating the table using Parquet or Avro data files (and, again, Parquet is the only format Impala supports with complex data) omit the ROW FORMAT clause and the subclauses specifying the terminators, and include a STORED AS clause.

Try It!

Do the following to create tables that you can query in the next readings.

1. Use Hive to create the example tables for each of the three types, and load the data with your preferred method. The data for each example is on the VM in /home/training/training_materials/analyst/data/. The data files are named customers_phones_array.csv, customers_phones_map.csv, and customers_addr.csv.

Use Hive to create a Parquet version of each table so you can also query the data with Impala. For each table, run a CTAS statement and use STORED AS PARQUET. To make things easier when you query these tables with Impala, name the tables phones_array_parquet, phones_map_parquet, and customers_addr_parquet.

If you prefer to use something shorter, please do, but you'll need to make adjustments when you complete the exercises in the "Querying Complex Data with Impala" reading.

For example:

```
CREATE TABLE phones_array_parquet
  STORED AS PARQUET
  AS SELECT * FROM customers_phones_array;
```

Querying Complex Data with Hive

A Hive query can select a full complex column simply by including the bare name of the column in the SELECT list. For example, this query selects the full ARRAY column named phones. Hive displays the full phones column in the results, using square brackets and commas to represent the ARRAY structure:

```
SELECT name, phones FROM customer_phones_array;
```

```
name   phones
Alice  [555-1111, 555-2222, 555-3333]
Bob    [555-4444]
Carlos [555-5555, 555-6666]
```

For access to the elements within the different complex data types, you have to use different syntax depending on the complex type.

Note: The syntax in this reading is for Hive only. See “Querying Complex Data with Impala” for the syntax to use with Impala.

Querying ARRAYs with Hive

To query an element within an ARRAY, use an array index number in square brackets. The array index starts at 0. For example, to get the first and second phone numbers in phones, use this query:

```
SELECT name, phones[0], phones[1]
  FROM customers_phones_array;
```

Since Bob has only one phone number, the query returns NULL for Bob’s second phone number.

Querying MAPs with Hive

Querying an element within a MAP is similar to querying the ARRAY element, except you use the key instead of the index. For example, to get the home phone numbers, use this query:

```
SELECT name, phones['home'] AS home
  FROM customers_phones_map;
```

In this example, the MAP keys are strings, so you must quote the literal string within the square brackets. MAP keys are case-sensitive, so 'HOME' or 'Home' would not work in this case.

Since Bob has only a mobile phone number, the query returns NULL for Bob’s home phone number.

Querying STRUCTs with Hive

To query a field from a STRUCT column, use the column name, a dot, and the field name (similar to how you can use the database name, a dot, and the table name to refer to a table in a different database from the active one). For example, this query selects the name column, and the state and zipcode fields from the address column:

```
SELECT name, address.state, address.zipcode
  FROM customers_addr;
```

In this example, Carlos’s address is missing the state and zipcode fields, so the query returns NULL for these missing fields.

Try It!

Do the following to run two queries on each of the three (non-Parquet) tables you created in "Creating Tables with Hive and Impala." Use Hive for these exercises. First, run **SELECT * FROM *tablename*;** for each table and note how the complex column appears in the results.

Then, run each of the examples in the reading above. Notice the **NULL** fields in each case.
Optional: Try some other queries for each table.

QUERYING COMPLEX DATA WITH IMPALA

As noted previously, while Impala does support the use of complex data types in tables, it does so with some limitations. Remember, Impala supports the use of complex columns only in Parquet tables. Also, Impala does not support selecting a full complex column simply by including the bare name of the column in the **SELECT** list, as Hive does. If you issue **SELECT *** queries on a table with complex columns, the query will run but the complex columns will be omitted from the results.

To access elements within a complex column using Impala, you have to use different syntax depending on the complex type. The syntax for accessing **ARRAY**s and **MAP**s also is different from Hive's syntax.
Note: The syntax in this reading is for Impala only. See "Querying Complex Data with Hive" for the syntax to use with Hive.

Pseudocolumns
An important concept in working with complex data using Impala is ***pseudocolumns***. To understand pseudocolumns, think of an **ARRAY** or **MAP** complex column as a table within a table. This inner table then has columns within it—those are the pseudocolumns. Every **ARRAY** has two columns named **item** and **pos**, and every **MAP** has two columns named **key** and **value**. As you'll see, you then treat the complex column as if it were a table, and use these pseudocolumns to access the elements within the column.

QUERYING ARRAYS WITH IMPALA

To query an element within an ARRAY, treat the array as a table named *tablename.arrayname*with the two pseudocolumns mentioned above: item and pos. The item pseudocolumn gives the value of each ARRAY element. The pos pseudocolumn gives the index of each element within the array. The array index starts at 0. For example, to get the first and second phone numbers in phones from the phones_array_parquet table, use this query:

```
SELECT item
  FROM phones_array_parquet.phones
  WHERE pos = 0 OR pos = 1;
```

Notice that you can use the item and pos pseudocolumns in the WHERE clause as well as the SELECT clause. You can also use them in other clauses, just as if they were regular columns.

Just as if this phones column were a regular table, the query results for this example will include one row for each phone number. This is different than Hive's behavior, which uses a comma-separated list, enclosed in brackets. In the "Complex Data in Practice" reading, you'll see how to produce similar output for Hive.

You often will want your query results to include the values from other columns in the actual table (such as the name column in phones_array_parquet) along with the items in the ARRAY column. The list of phone numbers, without the person who has each number, will probably not be useful! You can use join notation to return ARRAY elements along with scalar column values from corresponding table rows. Typically you use ***implicit*** join notation (also called ***SQL-89-style*** join notation) as shown in the following example. It's also possible to use ***explicit*** join notation and to specify different join types, such as LEFT OUTER JOIN, but that's used less often and is not described here. (For more about implicit join syntax, see Course 2, ***Analyzing Big Data with SQL,*** Week 6, "Alternative Join Syntax.")

```
SELECT name, phones.item AS phone
  FROM phones_array_parquet, phones_array_parquet.phones;
```

In this example, the FROM clause includes the base table name, which is phones_array_parquet, and the qualified name of the ARRAY column, which is phones_array_parquet.phones. These are separated with a comma, which indicates an implicit join. Notice that no join condition is specified, because the join condition is implied: the ARRAY elements are joined with the rows they came from.

The SELECT list can then include any columns from the base table along with the item and pos pseudocolumns from the ARRAY column. This example includes the name column and the item pseudocolumn in the SELECT list. It's a good practice to qualify the item and pos pseudocolumns with the ARRAY column name as shown here (phones.item) but this is not strictly required.
This may seem a bit complicated, but users often find Impala's familiar join syntax to be more straightforward than what's needed with Hive to get a separate row for each ARRAY element (again, see the "Complex Data in Practice" reading).

QUERYING MAPS WITH IMPALA

Querying a MAP column is similar to querying an ARRAY column, but the pseudocolumns are key and value, representing the keys and values of the MAP elements. For example, to get the home phone numbers, use this query:

```
SELECT value AS home
  FROM phones_map_parquet.phones
  WHERE key = 'home';
```

In this example, the Parquet table named phones_map_parquet contains a MAP column named phones. The MAP keys hold the label (home, work, or mobile) for each phone number, and the associated values hold the phone numbers themselves. To query this MAP column, you use the column name qualified with the table name in the FROM clause: FROM phones_map_parquet.phones. Then you can include one or both of the pseudocolumns in the SELECT list or in other clauses such as WHERE. This example returns the phone numbers (the values), and only phone numbers with the label 'home' will be returned.
As with ARRAY columns, use join notation to return MAP elements along with scalar column values from corresponding table rows.

```
SELECT name, phones.value AS home
  FROM phones_map_parquet, phones_map_parquet.phones
  WHERE phones.key = 'home';
```

In this example, the FROM clause includes the base table name, which is **phones_map_parquet**, and the qualified name of the MAP column, which is **phones_map_parquet.phones**. They are separated with a comma to indicate an implicit join. The SELECT list includes the **name** column and the **phones.value** pseudocolumn. As with ARRAY, it's a good practice to qualify the **value** and **key** pseudocolumns with the name of the MAP column, **phones**.

Querying STRUCTs with Impala

The Impala query syntax for **STRUCT** columns is exactly the same as Hive's:

To select a field from a **STRUCT** column, use the column name, a dot, and the field name (similar to how you can use the database name, a dot, and the table name to refer to a table in a different database from the active one). For example, this query selects the **name** column, and the **state** and **zipcode** fields from the **address** column:

```
SELECT name, address.state, address.zipcode
  FROM customers_addr_parquet;
```

Try It!

Do the following to run two queries on each of the three ***Parquet*** tables you created in "Creating Tables with Hive and Impala." Use Impala for these exercises.

First, recall that you created these tables using Hive, so there's something you need to do before you can query the tables with Impala. Do you remember what that is? Figure that out and do it.

Try running **SELECT * FROM *tablename*;** for each table. Notice that the complex column is omitted from the results.

Then, run each of the examples in the reading above. Notice the **NULL** fields in each case. (Why does Bob have no row in the MAP example?)

Optional: Try some other queries for each table.

Complex Data in Practice

While there are several things you may want to do with complex data, here are three examples of practical applications. The techniques used for these applications are not immediately apparent from understanding basic queries, and they must be handled differently between Hive and Impala, so look for the new information provided here!

Counting Items in a Collection

ARRAYs and MAPs can contain any number of items; they do not have a fixed size. You can use the size function in a Hive query to return the number of items in an ARRAY or MAP, but Impala does not have such a function. The examples here show how to find the number of items in both engines.

Using Hive

Using the size function with Hive is fairly straightforward:

```
SELECT name, size(phones) AS num
   FROM customers_phones_array;
```

For this example, the column named phones is an ARRAY column. Using our example data from the previous readings, the ARRAY in each row of this column contains a different number of items. In the row for Alice, the ARRAY has three items; in the row for Bob, the ARRAY has one item; and in the row for Carlos, the ARRAY has two items.

The query uses the size function to return these numbers of items as a column with the alias num. Similarly, when you use the size function with a MAP column, it returns the numbers of key-value pairs. The size function is an example of what's called a ***collection function***, because it's a function that operates on a ***collection type***, which is another name for a complex type.

Using Impala

As noted above, Impala does not have a size function, nor does Impala support any other collection functions. To count the number of elements in each ARRAY or MAP using Impala, you need to use join notation and a GROUP BY clause to group by a column or columns that have unique row values. You can then use the COUNT function, or indeed any other aggregation function, to aggregate the elements in each ARRAY or MAP.

```
SELECT name, COUNT(*) AS num
   FROM phones_array_parquet, phones_array_parquet.phones
   GROUP BY name;
```

In this example, the goal is to return each customer's name and the number of phone numbers they have. In the SELECT list, the expression COUNT(*) AS num returns a column named num giving the number of records in each group, which is the number of phone numbers each customer has.

CONVERTINGARRAYS AND MAPS TO RECORDS WITH HIVE

Recall that Impala's method of querying ARRAY and MAP types provides a separate row for each element in the complex column, and you must use a join to include values from the other columns in the table. Hive's behavior is very different, whichmay seem unusual, because typically Impala aims for a high degree of compatibility with Hive’s query syntax. In this case, Impala’s syntax is intended to provide greater flexibility.

If you want to use Hive to break the individual items within an ARRAY or MAP into a table of results with one item per row, you can do this using the explode function. The explode function is an example of what’s called a ***table-generating*** function; this is a class of functions that can transform a single input row into multiple output rows.

```
SELECT explode(phones) AS phone
  FROM customers_phones_array;
```

In this example, the explode function is applied to the ARRAY column named phones, and it returns a column with the alias phone. It returns one output record for each item in the ARRAY column. Using the same example data as in the previous readings, there are a total of six phone numbers in the ARRAY column (three for Alice, one for Bob, and two for Carlos). This means the output contains six records.
Since the MAP type has two parts to it, the key and the value, explode returns two values. You can use the same syntax ***except***for the alias—if you use an alias, you must supply two values:

```
SELECT explode(phones) AS (type, number)
  FROM customers_phones_map;
```

The result set would again have six rows, each with two columns: type and number. If you omit the alias, the columns would be called key and value.

When you use a table-generating function like explode, you cannot include any other columns in the SELECT list of your query. However, you can overcome this limitation by using a ***lateral view***, which first applies the table-generating function to the ARRAY or MAP column, then joins the resulting output with the rows of the table. Lateral view syntax is similar to explicit join syntax; in the FROM clause, include the name of the base table, then the keywords LATERAL VIEW, followed by the explode function applied to the ARRAY or MAP column.

```
SELECT name, phone
  FROM customers_phones_array
  LATERAL VIEW
    explode(phones) p AS phone;
```

In the example here, a lateral view is used to return values from the name column along with the individual phone numbers. ***The table alias,*** *p* ***in this example, is required,*** even if you don't use it anywhere else in the query**.**The column alias, AS phone in this example, is optional.
For the MAP version, again you need two column aliases—but without parentheses in this case:

```
SELECT name, type, number
  FROM customers_phones_map
  LATERAL VIEW
    explode(phones) p AS type, number;
```

DENORMALIZING TABLES USING COMPLEX DATA

A potential use for complex data is denormalizing tables with a one-to-many relationship. Normalized tables (using Third Normal Form) cannot have a repeating groups—that is, a single row should not have multiple values for one type of data. For example, a toy company likely has many products; a table listing different toy makers (like the makers table in the toy database) would not include all Hasbro's products in the same row.

Instead, you typically have a second table in which each row holds one of those values along with a foreign key that identifies which row in the first table that value belongs with. In the toy company in example, the toys table has one row for each toy, and the maker_id column identifies which company from the makers table makes that particular toy. Joining the columns using the maker's identification number allows you to identify all the toys made by a particular company.

The complex column types allow analysts to reshape tables into a denormalized form. The resulting revised data model can support queries of the combined data elements from one table, without any join required. In big data, you can expect a query that does not require a join to be significantly faster than one that does require a join.

The rest of this section describes how to set up and query such a table, using the toy example. You don't need to memorize these steps or the functions involved.

Creating the Denormalized Table

The makers_with_toys table defined below could hold each toy maker with its information, ***including*** a list of its toys (and the MSRP, manufacturer's suggested retail price) in the same row, using an ARRAY with STRUCTs as its elements. (The table uses Parquet files so you can query it with Impala. You can create this table in either Hive or Impala.)

```
  CREATE EXTERNAL TABLE toy.makers_with_toys (
      id INT,
      name STRING,
      city STRING,
      toys ARRAY<STRUCT <toy_name:STRING, price:DECIMAL(5,2)>>)
    STORED AS PARQUET;
```

Populating the Denormalized Table

The next step is to load the data into the table. Because Impala can't load data into Parquet files, this step ***must*** be completed with Hive.
Use the named_struct function to cast a row of the detail table (in this case, toys) into the STRUCT. Then use the collect_list function to collect multiple rows into the ARRAY. (These functions are probably new to you, because none of the courses in this specialization have introduced them before now. Consult the Hive documentation if you want to learn more about these functions.)

```
INSERT OVERWRITE TABLE toy.makers_with_toys
  SELECT m.id, m.name, m.city,
      collect_list(named_struct('toy_name', t.name,
                                'price', t.price))
    FROM toy.makers m LEFT OUTER JOIN toy.toys t
      ON (m.id = t.maker_id)
    GROUP BY m.id, m.name, m.city;
```

Querying the Denormalized Table

You now can query the table with Hive or Impala, using the syntax for the engine you're using. For example, you can find the price of the most expensive toy for each maker using the following query with Impala. Remember to invalidate metadata before attempting to run this query with Impala:

```
SELECT name, MAX(toys.item.price) AS max_price
  FROM toy.makers_with_toys, toy.makers_with_toys.toys
  GROUP BY name;
```

Note that to query the elements in the ARRAY column (toys), you need to reference the pseudocolumn toys.item as the column, but that element is a STRUCT. So you then need to use .price to identify the element within that STRUCT.

Try It!

Try each of the examples above, using both the ARRAY and MAP tables when appropriate.

Storage Engines

File Systems versus Storage Engines

Throughout this course, we've talked about how the data for Hive and Impala tables is stored in a file system like HDFS or S3. However, that is not the whole truth. The vast majority of the time, yes, the data is stored in a file system. However, Hive and Impala can also work with data stored in some other systems.

These include Apache HBase and Apache Kudu. These are not file systems. They do not store data in files that you can list and access. Instead, these systems encapsulate the data storage. They manage the data for you and they abstract away the details of how it's stored and accessed. They provide a higher level interface to the data. The name we use for these systems, systems like HBase and Kudu is storage engines. Hive and Impala have what are called storage handlers that allow them to interact with these storage engines. What you can do with these storage engines varies depending on whether you're using Hive or Impala. Hive can create tables with data stored in HBase and it can query those tables. Hive also offers some limited options for managing the data that's in these tables, but mostly, you would manage the data in HBase tables from outside of Hive.

Impala can query HBase tables but it cannot create them. You need to create them in Hive. Impala can create tables with data stored in Kudu and it can query those tables. Impala also has a rich set of options for loading data into Kudu tables and managing Kudu tables. Hive currently does not support Kudu tables. Also, besides HBase, there are a few other storage engines that it's possible to use with Hive, but they're not as widely used and I will not discuss them here. In general, there are two main reasons why you would use a storage engine instead of a file system with Hive and Impala. One reason is that the data you need to query is already stored in one of these storage engines, and using Hive or Impala is an easy way to query it. This is a common reason for using HBase with Hive or Impala. The other reason is that you need to overcome some of the limitations of distributed file systems like HDFS and S3. This is a common reason for using Kudu with Impala. So what limitations am I talking about? Well, typically the biggest limitation is that files in HDFS and in S3 are immutable. They cannot be directly modified. You can delete a file or completely overwrite a file to replace it with a new version, but you cannot modify a file in place. You cannot directly modify some part of a file.

This makes file systems like HDFS and S3 a poor choice for applications in which data needs to be updated frequently. If you took the first course in this specialization, you might recall that this is one of the trade-offs between RDBMSs and engines like Hive and Impala. RDBMSs are typically a better fit for rapidly changing data. However, by using a storage engine like HBase or Kudu to store the data, you can overcome this limitation, and you can have frequent updates like in RDBMS and the scalability and lower cost of open source Big Data systems. Kudu in particular, is an attractive choice when you need to enable real-time analytics on rapidly changing data. It's designed specifically to enable this and you can query Kudu tables with Impala the same way you query other Impala tables. The remainder of this lesson provides some further details about Kudu.

OVERVIEW OF APACHE KUDU

Apache Kudu is an open source storage engine designed to handle very large amounts of data—terabytes or even petabytes. Kudu stores data directly on the file system of the machine it's running on. In other words, it's ***not*** built on top of an underlying distributed file system, as some storage systems are. (Apache HBase, for example, is built on top of HDFS.)

Kudu structures data as tables much like those in a relational database, but with the ability to scale to support more data and higher throughput than traditional RDBMSs. This means it provides excellent performance for analytical queries through mechanisms like SQL, while also offering great performance for workloads common in online databases, such as random access and lookups and updates by key.

Many storage systems excel at one or the other of these types of access patterns. Kudu excels at both, making applications simpler to develop and more scalable.

FEATURES

In order to provide performant storage for applications that use a variety of access patterns, Kudu offers high throughput for scans through large portions of a table's data, as required by analytical use cases. It also has low latency to support quick access to individual rows or groups of rows.

Another key Kudu feature is its support for atomic transactions, much like traditional databases. This means that when you write data to a row, if any part of the operation fails, the entire operation fails, so you don't end up with data in an inconsistent state. Kudu supports atomic transactions for single-row operations, but not for multi-row or multi-table transactions.

Kudu Architecture

Kudu runs on a cluster of machines, which can range in number from just a few machines to thousands. These machines collectively store and manage Kudu's data. (Note that the Kudu cluster is distinct from the Hadoop cluster that includes HDFS, although hosts can be part of both.) The cluster architecture is designed for fault tolerance and high availability—meaning that the system will always be able to respond to client requests to read and write data. Kudu clusters are also easily scaled by adding more nodes to the cluster as your storage and throughput needs increase.

Kudu's architecture is based on ***tablets,*** each containing a subset of the data in a table. Instead of storing the data in a table in a single location, Kudu distributes the table's tablets to multiple machines in the cluster. Those machines store the tablet's data on their local disks and respond to client requests to read and write table data. Tables must have primary keys, and the table data is distributed using hash or range partitioning (and possibly both), with columnar storage. (See the reading on Parquet files in Week 3 for more on columnar formats.)

Kudu and Impala

Apache Impala is tightly integrated with Kudu to enable data analysts, data scientists, and developers to access data in Kudu using common SQL syntax. Kudu supports most Impala SQL commands, which you can use to create and modify Kudu tables; insert, modify, and delete data; and issue queries. The details are beyond the scope of this course.

For More Information

The following offer more information on Apache Kudu:
Apache Kudu Overview (on the official website)
Introduction to Apache Kudu (a training course available for purchase from Cloudera)
Kudu: Storage for Fast Analytics on Fast Data
Documentation for using Impala with Kudu
Impala Guide: Using Impala to Query Kudu Tables