UNIT V
BIG DATA FRAMEWORKS

Introduction to NoSQL – Aggregate Data Models – Hbase: Data Model and Implementations – Hbase Clients – Examples – .Cassandra: Data Model – Examples – Cassandra Clients – Hadoop Integration. Pig – Grunt – Pig Data Model – Pig Latin – developing and testing Pig Latin scripts. Hive – Data Types and File Formats – HiveQL Data Definition – HiveQL Data Manipulation – HiveQL Queries

NoSQL

- NoSQL is a non-relational database management systems, different from traditional relational database management systems in some significant ways.
- It is designed for distributed data stores where very large scale of data storing needs (for example Google or Facebook which collects terabits of data every day for their users).
- These type of data storing may not require fixed schema, avoid join operations and typically scale horizontally.
- It provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases.
- NoSQL database doesn't use tables for storing data. It is generally used to store big data and real-time web applications.

NoSQL Advantages :

- High scalability
- Distributed Computing
- Lower cost
- Schema flexibility, semi-structure data
- No complicated Relationships

Disadvantages

- No standardization
- Limited query capabilities (so far)
- Eventual consistent is not intuitive to program for

NoSQL Categories

There are four general types (most common categories) of NoSQL databases. Each of these categories has its own specific attributes and limitations. There is not a single solutions which is better than all the others, however there are some databases that are better to solve specific problems.

To clarify the NoSQL databases, lets discuss the most common categories :

- Key-value stores
- Column-oriented
- Graph
- Document oriented

Aggregate Data Models

A data model is the model through which we perceive and manipulate our data. For people using a database, the data model describes how we interact with the data in the database. This is distinct from a storage model, which describes how the database stores and manipulates the data internally. In an ideal world, we should be ignorant of the storage model.

In conversation, the term "data model" often means the model of the specific data in an application. A developer might point to an entity-relationship diagram of their database and refer to that as their data model containing customers, orders, products, and the like. However, in this book we'll mostly be using "data model" to refer to the model by which the database organizes data—what might be more formally called a metamodel.

The dominant data model of the last couple of decades is the relational data model, which is best visualized as a set of tables, rather like a page of a spreadsheet. Each table has rows, with each row representing some entity of interest. We describe this entity through columns, each having a single value. A column may refer to another row in the same or different table, which constitutes a relationship between those entities. (We're using informal but common terminology when we speak of tables and rows; the more formal terms would be relations and tuples.)

One of the most obvious shifts with NoSQL is a move away from the relational model. Each NoSQL solution has a different model that it uses, which we put into four categories widely used in the NoSQL ecosystem: key-value, document, column-family, and graph. Of these, the first three share a common characteristic of their data models which we will call aggregate orientation

Key-Value and Document Data Models

The key-value and document databases were strongly aggregate-oriented. What we meant by this was that we think of these databases as primarily constructed through aggregates. Both of these types of databases consist of lots of aggregates with each aggregate having a key or ID that's used to get at the data.

The two models differ in that in a key-value database, the aggregate is opaque to the database—just some big blob of mostly meaningless bits. In contrast, a document database is able to see a structure in the aggregate. The advantage of opacity is that we can store whatever we like in the aggregate. The database may impose some general size limit, but other than that we have complete freedom. A document database imposes limits on what we can place in it, defining allowable structures and types. In return, however, we get more flexibility in access.

With a key-value store, we can only access an aggregate by lookup based on its key. With a document database, we can submit queries to the database based on the fields in the aggregate, we can retrieve part of the aggregate rather than the whole thing, and database can create indexes based on the contents of the aggregate.

Column-Family Stores

One of the early and influential NoSQL databases was Google's BigTable. Its name conjured up a tabular structure which it realized with sparse columns and no schema. It is a two-level map. But, however think about the structure, it has been a model that influenced later databases such as HBase and Cassandra.

Hbase
HBase is a column-oriented database that's an open-source implementation of Google's Big Table storage architecture. It can manage structured and semi-structured data and has some built-in features such as scalability, versioning, compression and garbage collection.

Since its uses write-ahead logging and distributed configuration, it can provide fault-tolerance and quick recovery from individual server failures. HBase built on top of Hadoop / HDFS and the data stored in HBase can be manipulated using Hadoop's MapReduce capabilities.

Let's now take a look at how HBase (a column-oriented database) is different from some other data structures and concepts that we are familiar with **Row-Oriented vs. Column-Oriented data stores.** As shown below, in a row-oriented data store, a row is a unit of data that is read or written together. In a column-oriented data store, the data in a column is stored together and hence quickly retrieved.

Row-oriented data stores –

- Data is stored and retrieved one row at a time and hence could read unnecessary data if only some of the data in a row is required.
- Easy to read and write records

- Well suited for OLTP systems
- Not efficient in performing operations applicable to the entire dataset and hence aggregation is an expensive operation
- Typical compression mechanisms provide less effective results than those on column-oriented data stores

Column-oriented data stores –

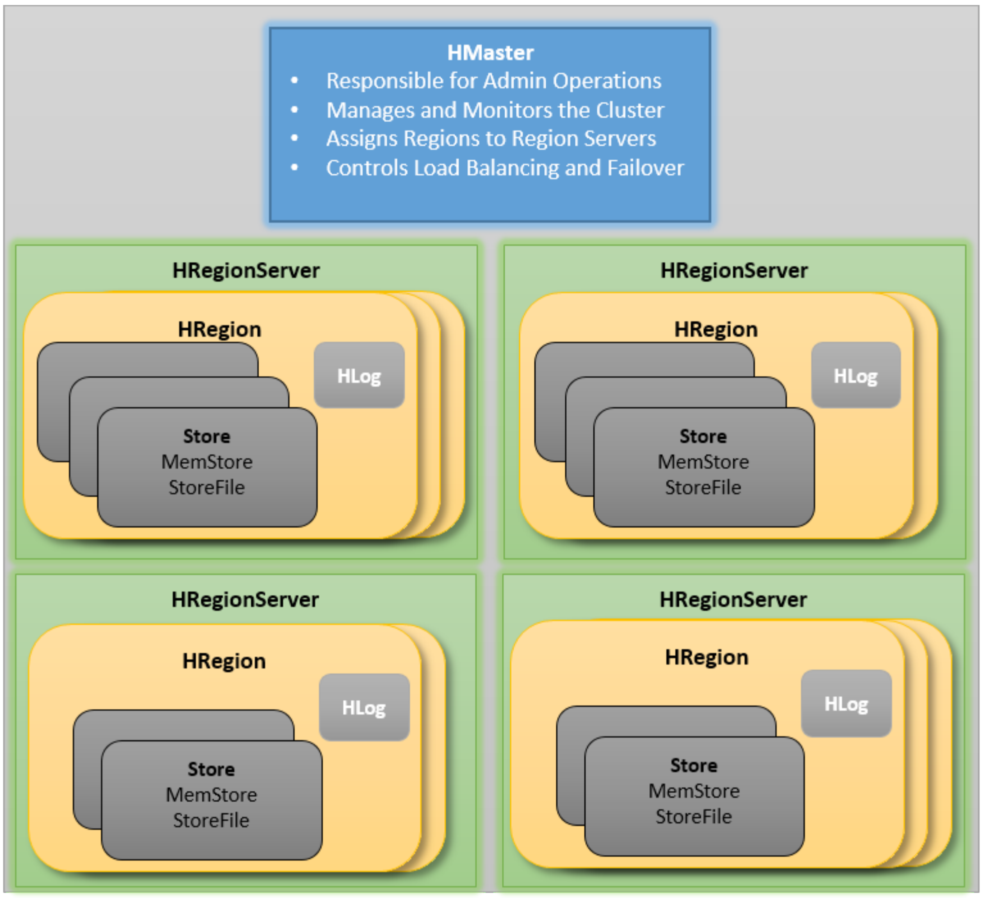
- Data is stored and retrieved in columns and hence can read only relevant data if only some data is required
- Read and Write are typically slower operations
- Well suited for OLAP systems
- Can efficiently perform operations applicable to the entire dataset and hence enables aggregation over many rows and columns
- Permits high compression rates due to few distinct values in columns

HBase –

- Is Schema-less
- Is a Column-oriented datastore
- Is designed to store Denormalized Data
- Contains wide and sparsely populated tables
- Supports Automatic Partitioning

HBase Architecture

The HBase Physical Architecture consists of servers in a Master-Slave relationship as shown below. Typically, the HBase cluster has one Master node, called HMaster and multiple Region Servers called HRegionServer. Each Region Server contains multiple Regions – HRegions.



Just like in a Relational Database, data in HBase is stored in Tables and these Tables are stored in Regions. When a Table becomes too big, the Table is partitioned into multiple Regions. These Regions are assigned to Region Servers across the cluster. Each Region Server hosts roughly the same number of Regions.

The HMaster in the HBase is responsible for

•Performing Administration

•Managing and Monitoring the Cluster

•Assigning Regions to the Region Servers

•Controlling the Load Balancing and Failover

On the other hand, the HRegionServer perform the following work

•Hosting and managing Regions

•Splitting the Regions automatically

•Handling the read/write requests

•Communicating with the Clients directly

Each Region Server contains a Write-Ahead Log (called HLog) and multiple Regions. Each Region in turn is made up of a MemStore and multiple StoreFiles (HFile). The data lives in these StoreFiles in the form of Column Families (explained below). The MemStore holds in-memory modifications to the Store (data).

The mapping of Regions to Region Server is kept in a system table called .META. When trying to read or write data from HBase, the clients read the required Region information from the .META table and directly communicate with the appropriate Region Server. Each Region is identified by the start key (inclusive) and the end key (exclusive)

HBase Data Model

The Data Model in HBase is designed to accommodate semi-structured data that could vary in field size, data type and columns. Additionally, the layout of the data model makes it easier to partition the data and distribute it across the cluster. The Data Model in HBase is made of different logical components such as Tables, Rows, Column Families, Columns, Cells and Versions.

Tables – The HBase Tables are more like logical collection of rows stored in separate partitions called Regions. As shown above, every Region is then served by exactly one Region Server. The figure above shows a representation of a Table.

Rows – A row is one instance of data in a table and is identified by a rowkey. Rowkeys are unique in a Table and are always treated as a byte[].

Column Families – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it's important that proper care be taken when designing Column Families in table.

Columns – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily:columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied number of Columns.

Cell – A Cell stores data and is essentially a unique combination of rowkey, Column Family and the Column (Column Qualifier). The data stored in a Cell is called its value and the data type is always treated as byte[].

Version – The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

Example

Example: HBase APIs for Java "Hello World" Application

This example is a very simple "hello world" application, written in Java, that illustrates how to:

• Connect to a Cloud Bigtable instance.

• Create a new table.

• Write data to the table.

• Read the data back.

• Delete the table.

Running the sample

The sample uses the HBase APIs to communicate with Cloud Bigtable. The code for this sample is in the GitHub repository GoogleCloudPlatform/cloud-bigtable-examples, in the directory java/hello-world.

To run this sample program, follow the instructions for the sample on GitHub.

Using the HBase APIs

The sample application connects to Cloud Bigtable and demonstrates some simple operations.

Installing and importing the client library

This samples uses the Cloud Bigtable HBase client for Java. This sample uses Maven. See the instructions for how to include it as a Maven dependency.

The sample uses the following imports:

```
import com.google.cloud.bigtable.hbase.BigtableConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.util.Bytes;
import java.io.IOException;
```

Connecting to Cloud Bigtable

Connect to the Cloud Bigtable using the BigtableConfiguration class.

```
// Create the Bigtable connection, use try-with-resources to make sure it gets closed
```

```
try (Connection connection = BigtableConfiguration.connect(projectId, instanceId)) {
  // The admin API lets us create, manage and delete tables
  Admin admin = connection.getAdmin();
```

Creating a table

```
// Create a table with a single column family
HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf(TABLE_NAME));
descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));
print("Create table " + descriptor.getNameAsString());
admin.createTable(descriptor);
```

Reading a row by its key

Get a row directly using its key.

```
// Get the first greeting by row key
String rowKey = "greeting0";
Result getResult = table.get(new Get(Bytes.toBytes(rowKey)));
String greeting = Bytes.toString(getResult.getValue(COLUMN_FAMILY_NAME,
COLUMN_NAME));
System.out.println("Get a single greeting by row key");
System.out.printf("\t%s = %s\n", rowKey, greeting);
```

Scanning all table rows

Use the Scan class to get a range of rows.

```
// Now scan across all rows.
Scan scan = new Scan();
print("Scan for all greetings:");
ResultScanner scanner = table.getScanner(scan);
for (Result row : scanner) {
  byte[] valueBytes = row.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME);
  System.out.println('\t' + Bytes.toString(valueBytes));
}
```

Deleting a table

```
// Clean up by disabling and then deleting the table
print("Delete the table");
admin.disableTable(table.getName());
admin.deleteTable(table.getName());
```

Cassandra

The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data.Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages.

Data Modeling

Conceptual Data Modeling

First, let's create a simple domain model that is easy to understand in the relational world, and then see how we might map it from a relational to a distributed hashtable model in Cassandra.

Our conceptual domain includes hotels, guests that stay in the hotels, a collection of rooms for each hotel, the rates and availability of those rooms, and a record of reservations booked for guests. Hotels typically also maintain a collection of "points of interest," which are parks, museums, shopping galleries, monuments, or other places near the hotel that guests might want to visit during their stay. Both hotels and points of interest need to maintain geolocation data so that they can be found on maps for mashups, and to calculate distances.

Logical Data Modeling

Now that after defining our queries, we're ready to begin designing our Cassandra tables. First, we'll create a logical model containing a table for each query, capturing entities and relationships from the conceptual model.

To name each table, identify the primary entity type for which we are querying and use that to start the entity name. If we are querying by attributes of other related entities, we append those to the table name, separated with `_by_`. For example, `hotels_by_poi`.

Next, identify the primary key for the table, adding partition key columns based on the required query attributes, and clustering columns in order to guarantee uniqueness and support desired sort ordering.

Complete each table by adding any additional attributes identified by the query. If any of these additional attributes are the same for every instance of the partition key, we mark the column as static.

Physical Data Modeling

Once we have a logical data model defined, creating the physical model is a relatively simple process.

Each of our logical model tables, assigning types to each item. Use any data types, including the basic types, collections, and user-defined types. We may identify additional user-defined types that can be created to simplify our design.

After assigning our data types, we analyze our model by performing size calculations and testing out how the model works. We may make some adjustments based on our findings.

Cassandra Clients

Cassandra allows you to secure the client transport (CQL) as well as the cluster transport (storage transport).

SSL/TLS have some overhead. This is especially true in the JVM world which is not as performant for handling SSL/TLS unless you are using Netty/OpenSSl integration.

If possible, use no encryption for the cluster transport (storage transport), and deploy your Cassandra nodes in a private subnet, and limit access to this subnet to the client transport. Also if possible avoid using TLS/SSL on the client transport and do client operations from your app tier, which is located in a non-public subnet.

Industry that requires the use of encrypted transports like the U.S. Health Insurance Portability and Accountability Act (HIPAA), Germany's Federal Data Protection Act, The Payment Card Industry Data Security Standard (PCI DSS), or U.S. Sarbanes-Oxley Act of 2002. Or you might work for a bank or other financial institution. Or it just might be a corporate policy to encrypt such transports.

Another area of concern is for compliance is authorization, and encrypted data at rest. Cassandra's has essential security features: authentication, role-based authorization, transport encryption (JMX, client transport, cluster transport), as well as data at rest encryption (encrypting SSTables).

Encrypting the transports

Data that travels over the client transport across a network could be accessed by someone you don't want accessing said data with tools like wire shark. If data includes private information, SSN number, credentials (password, username), credit card numbers or account numbers, then we want to make that data unintelligible (encrypted) to any and all 3rd parties. This is especially important if we don't control the network. You can also use TLS to make sure the data has not been tampered with whilst traveling the network. The Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols are designed to provide these features (SSL is the old name for what became TLS but many people still refer to TLS as SSL).

Cassandra is written in Java. Java defines the JSSE framework which in turn uses the Java Cryptography Architecture (JCA). JSSE uses cryptographic service providers from JCA.

Cassandra Hadoop Integration

A Cassandra/Hadoop integration can provide remarkable performance for companies using big data to drive business improvement. As the leading platform for processing big data, the Hadoop framework provides the distributed processing and massive scalability needed to process enormous amounts of data from widely disparate sources. As a leading NoSQL database, Cassandra delivers linear scalability and high availability without compromising performance. Together, Cassandra and Hadoop provide the big data tools and processing power it takes to manage big data effectively.

But deploying and managing a Cassandra/Hadoop integration is no picnic. Odds are, for most organizations, legacy architecture can't adequately handle a Cassandra/Hadoop integration. Existing integration tools likely won't cut it when it comes to connecting with all the new and emerging data sources that big data requires. And since very few developers have the training to manage big data technologies, most enterprises probably don't have the skills they need right now to trust business-critical data to a Cassandra/Hadoop integration.

Fortunately, Talend provides a simple solution, delivering easy-to-use tools that let developers work with big data technologies and manage a Cassandra/Hadoop integration using the skills they already have.

Talend software for Cassandra/Hadoop integration.

Talend Big Data is a powerful, open source platform that that makes managing a Cassandra/Hadoop integration easier and less costly. Running 100% natively on Hadoop, Talend lets developers use existing skills to quickly load, extract and transform big data sets with technologies like YARN (MapReduce 2.0), HBase, Hive, HCatalog, Oozie, Pig and Sqoop. With Talend, organizations can get a Cassandra/Hadoop integration up and running in hours instead of days or months.

Talend delivers easy-to-use tools for Hadoop/Cassandra integration. With Talend organizations can:

• Connect quickly to any data source using more than 800 pre-built connectors. Developers can visually map data sources to targets in an easy-to-use graphical environment. NoSQL connectivity provides access to Cassandra and other NoSQL and Hadoop database technologies, to speed development without requiring specific knowledge of NoSQL.

• Perform complex transformations using the skills developers already have to map, compare, filter, evaluate and group massive data sets.

• Quickly scale as needed, relying on Talend's massive scalability. Once a Hadoop connector is configured, the underlying code is automatically generated as new data clusters are added.

• Ensure data quality with tools that provide clearer visibility into the accuracy, integrity and completeness of data.

• Govern big data projects with ease using a simple and intuitive environment to schedule, monitor and deploy any job on a Cassandra/Hadoop integration.

Pig

Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger sets of data representing them as data flows. Pig is generally used with **Hadoop**; we can perform all the data manipulation operations in Hadoop using Apache Pig.

To write data analysis programs, Pig provides a high-level language known as **Pig Latin**. This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.

To analyze data using **Apache Pig**, programmers need to write scripts using Pig Latin language. All these scripts are internally converted to Map and Reduce tasks. Apache Pig has a component known as **Pig Engine** that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

Grunt

After invoking the Grunt shell, you can run your Pig scripts in the shell. In addition to that, there are certain useful shell and utility commands provided by the Grunt shell. This chapter explains the shell and utility commands provided by the Grunt shell.

Invoking the Grunt Shell

Invoke the Grunt shell in a desired mode (local/MapReduce) using the −**x** option as shown below.

Local mode	MapReduce mode
Command −	**Command −**
$ ./pig –x local	$ ./pig -x mapreduce

Either of these commands gives you the Grunt shell prompt as shown below.

```
grunt>
```

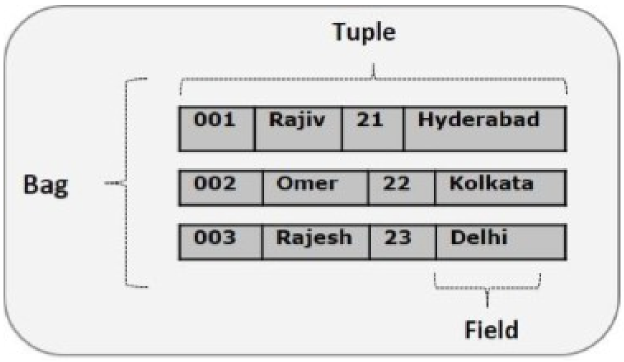
Exit the Grunt shell using **'ctrl + d'.**

After invoking the Grunt shell, you can execute a Pig script by directly entering the Pig Latin statements in it.

```
grunt> customers = LOAD 'customers.txt' USING PigStorage(',');
```

Pig Data Model

The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**. Given below is the diagrammatical representation of Pig Latin's data model.



Atom

Any single value in Pig Latin, irrespective of their data, type is known as an **Atom**. It is stored as string and can be used as string and number. int, long, float, double, chararray, and bytearray are the atomic values of Pig. A piece of data or a simple atomic value is known as a **field**.

Example − 'raja' or '30'

Tuple

A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type. A tuple is similar to a row in a table of RDBMS.

Example − (Raja, 30)

Bag

A bag is an unordered set of tuples. In other words, a collection of tuples (non-unique) is known as a bag. Each tuple can have any number of fields (flexible schema). A bag is represented by '{}'. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Example − {(Raja, 30), (Mohammad, 45)}

A bag can be a field in a relation; in that context, it is known as **inner bag**.

Example − {Raja, 30, **{9848022338, raja@gmail.com,}**}

Map

A map (or data map) is a set of key-value pairs. The **key** needs to be of type chararray and should be unique. The **value** might be of any type. It is represented by '[]'

Example − [name#Raja, age#30]

Relation

A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

Developing and Testing Pig Latin Scripts

Development Tools

Pig provides several tools and diagnostic operators to help you develop your applications. It is easier to develop Pig with standard editors and integrated development environments (IDEs).

Syntax Highlighting and Checking

Syntax highlighting often helps users write code correctly, at least syntactically, the first

time around. Syntax highlighting packages exist for several popular editors.
describe

describe shows you the schema of a relation in your script. This can be very helpful for developing scripts. It is especially useful as you are learning Pig Latin and understanding how various operators change the data. describe can be applied to any
relation in script.
explain

There are two ways to use explain. You can explain any alias in your Pig Latin script, which will show the execution plan Pig would use if you stored that relation. You can also take an existing Pig Latin script and apply explain to the whole script in Grunt.
illustrate

Often one of the best ways to debug your Pig Latin script is to run your data through it. But if you are using Pig, the odds are that you have a large data set. If it takes several hours to process your data, this makes for a very long debugging cycle. One obvious solution is to run your script on a sample of your data. For simple scripts this works fine. But sampling has another problem: it is not always trivial to pick a sample that will exercise your script properly.
Testing Your Scripts with PigUnit

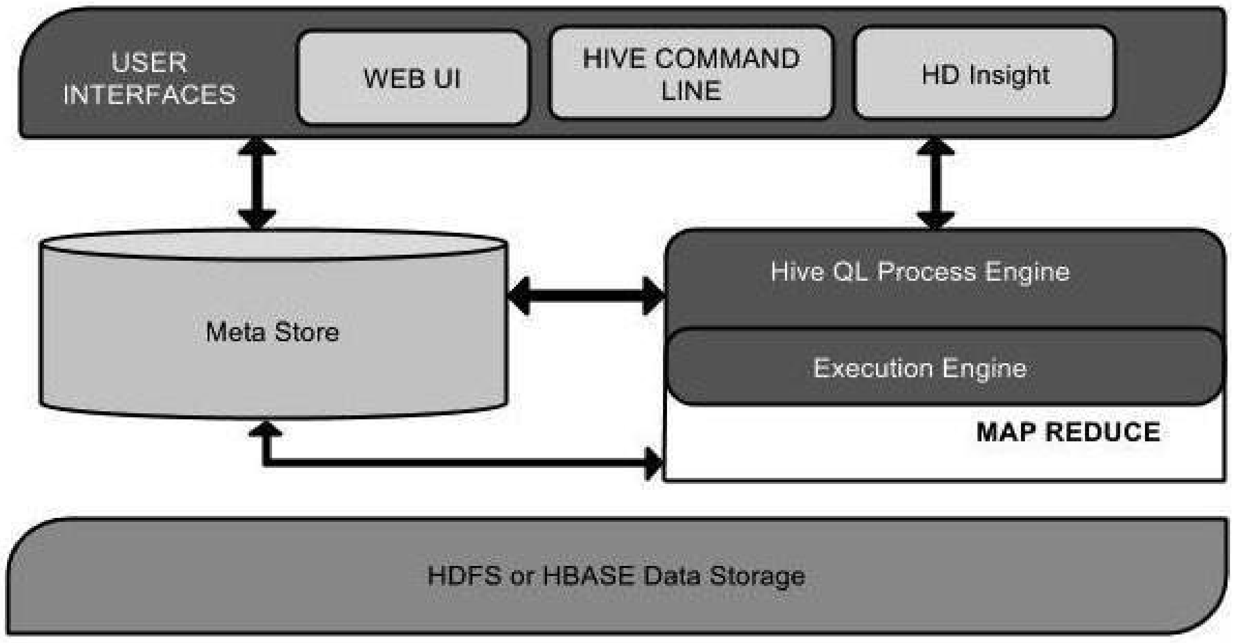
As part of your development, you will want to test your Pig Latin scripts. Even once they are finished, regular testing helps assure that changes to your UDFs, to your scripts, or in the versions of Pig and Hadoop that you are using do not break your code.

Second, you will need the pigunit.jar JAR file. This is not distributed as part of the standard Pig distribution, but you can build it from the source code included in your distribution. To do this, go to the directory your distribution is in and type ant jar pigunit-jar. Once this is finished, there should be two files in the directory: pig.jar and pigunit.jar. You will need to place these in your classpath when running PigUnit tests.

Third, you need data to run through your script. You can use an existing input file, or can manufacture some input in your test and run that through your script.

Finally, you need to write a Java class that JUnit can be used to run the test.

Hive
Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.
Apache Hive is a data ware house system for Hadoop that runs SQL like queries called HQL (Hive query language) which gets internally converted to map reduce jobs.
Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive.
It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.
The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.
• Sqoop: It is used to import and export data to and from between HDFS and RDBMS.
• Pig: It is a procedural language platform used to develop a script for MapReduce operations.
• Hive: It is a platform used to develop SQL type scripts to do MapReduce operations.
Features of Hive
• It stores schema in a database and processed data into HDFS.
• It is designed for OLAP.
• It provides SQL type language for querying called HiveQL or HQL.
• It is familiar, fast, scalable, and extensible.
Architecture of Hive
The following component diagram depicts the architecture of Hive:



This component diagram contains different units. The following table describes each unit:
User Interface
Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store
Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine
HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine

The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE
Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Hive Data Types anf File Formats
Hive supports many of the *primitive* data types you find in relational databases, as well as three *collection* data types that are rarely found in relational databases, for reasons we'll discuss shortly.

A related concern is how these types are represented in text files, as well as alternatives to text storage that address various performance and other concerns. A unique feature of Hive, compared to most databases, is that it provides great flexibility in how data is encoded in files. Most databases take total control of the data, both how it is persisted to disk and its life cycle. By letting you control all these aspects, Hive makes it easier to manage and process data with a variety of tools.

Primitive Data Types
Hive supports several sizes of integer and floating-point types, a Boolean type, and character strings of arbitrary length. Hive v0.8.0 added types for timestamps and binary fields.

The list of *primitive* types supported by Hive are

Primitive data types

Type	**Size**
`TINYINT`	1 byte signed integer.
`SMALLINT`	2 byte signed integer.
`INT`	4 byte signed integer.
`BIGINT`	8 byte signed integer.
`BOOLEAN`	Boolean true or false.
`FLOAT`	Single precision floating point.
`DOUBLE`	Double precision floating point.
`STRING`	Sequence of characters. The character set can be specified. Single or double quotes can be used.
`TIMESTAMP`	Integer, float, or string.
`BINARY`	Array of bytes.

Collection Data Types
Hive supports columns that are `structs`, `maps`, and `arrays`.

Collection data types

Type	**Description**
`STRUCT`	Analogous to a C `struct` or an "object." Fields can be accessed using the "dot" notation. For example, if a column `name` is of type `STRUCT {first STRING; last STRING}`, then the first name field can be referenced using `name.first`.
`MAP`	A collection of key-value tuples, where the fields are accessed using array notation (e.g., ['key']). For example, if a column `name` is of type `MAP` with key→value pairs `'first'→'John'` and `'last'→'Doe'`, then the last name can be referenced using

Type	Description
	`name['last']`.
`ARRAY`	Ordered sequences of the *same* type that are indexable using zero-based integers. For example, if a column `name` is of type `ARRAY` of strings with the value `['John', 'Doe']`, then the second element can be referenced using `name[1]`.

File Formats

Text files delimited with commas or tabs, the so-called *comma-separated values* (CSVs) or *tab-separated values* (TSVs), respectively. Hive can use those formats. However, there is a drawback to both formats; commas or tabs embedded in text and not intended as field or column delimiters. For this reason, Hive uses various control characters by default, which are less likely to appear in value strings. Hive uses the term `field` when overriding the default delimiter.

Hive's default record and field delimiters

Delimiter	Description
`\n`	For text files, each line is a record, so the line feed character separates records.
`^A` ("control" A)	Separates all fields (columns). Written using the octal code `\001` when explicitly specified in `CREATE TABLE` statements.
`^B`	Separate the elements in an `ARRAY` or `STRUCT`, or the key-value pairs in a `MAP`. Written using the octal code `\002` when explicitly specified in `CREATE TABLE` statements.
`^C`	Separate the key from the corresponding value in `MAP` key-value pairs. Written using the octal code `\003` when explicitly specified in `CREATE TABLE` statements

Example

```
CREATE TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Hive has many ways to create, modify, and even damage the data that Hive will query. Therefore, Hive can only enforce queries on *read*. This is called *schema on read*.

`If the schema doesn't match the file contents, Hive does the best that it can to read the data.  If some fields are numbers and Hive encounters nonnumeric strings, it will return nulls for those fields. Above all else, Hive tries to recover from all errors.`

HiveQL: Data Definition

HiveQL is the Hive query language. Like all SQL dialects in widespread use, it doesn't fully conform to any particular revision of the ANSI SQL standard. It is perhaps closest to MySQL's dialect, but with significant differences. Hive offers no support for row-level inserts, updates, and deletes. Hive doesn't support transactions. Hive adds extensions to provide better performance in the context of Hadoop and to integrate with custom extensions and even external programs.

Databases in Hive

The Hive concept of a database is essentially just a *catalog* or *namespace* of tables. However, they are very useful for larger clusters with multiple teams and users, as a way of avoiding table name collisions. It's also common to use databases to organize production tables into logical groups.

If you don't specify a database, the default database is used.

The simplest syntax for creating a database is shown in the following example:

```
hive> CREATE DATABASE databasename;
```

drop a database:

```
hive> DROP DATABASE IF EXISTS databasename;
```

The IF EXISTS is optional

Alter Database

You can set key-value pairs in the `DBPROPERTIES` associated with a database using the `ALTER DATABASE` command. No other metadata about the database can be changed, including its name and directory location:

```
hive> ALTER DATABASE financials SET DBPROPERTIES ('edited-by' = 'Joe Dba');
```

There is no way to delete or "unset" a `DBPROPERTY`.

Creating Tables

The `CREATE TABLE` statement follows SQL conventions, but Hive's version offers significant extensions to support a wide range of flexibility where the data files for tables are stored, the formats used, etc.

Example

```
CREATE TABLE IF NOT EXISTS mydb.employees (
  name         STRING COMMENT 'Employee name',
  salary       FLOAT  COMMENT 'Employee salary',
  subordinates ARRAY<STRING> COMMENT 'Names of subordinates',
  deductions   MAP<STRING, FLOAT>
               COMMENT 'Keys are deductions names, values are percentages',
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
               COMMENT 'Home address')
COMMENT 'Description of the table'
TBLPROPERTIES ('creator'='me', 'created_at'='2012-01-02 10:00:00', ...)
LOCATION '/user/hive/warehouse/mydb.db/employees';
```

If not currently working in the target database, databse prefix can be used(`mydb)` .

If the option `IF NOT EXISTS is added`, Hive will silently ignore the statement if the table already exists. This is useful in scripts that should create a table the first time they run.

If the schema specified differs from the schema in the table that already exists, Hive won't warn.

Alter Table

Most table properties can be altered with `ALTER TABLE` statements, which change *metadata* about the table but not the data itself. These statements can be used to fix mistakes in schema.

`ALTER TABLE` modifies table metadata *only*. The data for the table is untouched. It's up to you to ensure that any modifications are consistent with the actual data.

Renaming a Table

Use this statement to rename the table `log_messages` to `logmsgs`:

```
ALTER TABLE log_messages RENAME TO logmsgs;
```

Adding, Modifying, and Dropping a Table Partition

`ALTER TABLE table ADD PARTITION` … is used to add a new partition to a table (usually an *external* table).

```
ALTER TABLE log_messages ADD IF NOT EXISTS
PARTITION (year = 2011, month = 1, day = 1) LOCATION '/logs/2011/01/01'
PARTITION (year = 2011, month = 1, day = 2) LOCATION '/logs/2011/01/02'
PARTITION (year = 2011, month = 1, day = 3) LOCATION '/logs/2011/01/03'
...;
```

Multiple partitions can be added in the same query when using Hive v0.8.0 and later. As always, `IF NOT EXISTS` is optional and has the usual meaning.

Similarly, you can change a partition location, effectively moving it:

```
ALTER TABLE log_messages PARTITION(year = 2011, month = 12, day = 2)
SET LOCATION 's3n://ourbucket/logs/2011/01/02';
```

This command does not move the data from the old location, nor does it delete the old data.

Finally, drop a partition:

```
ALTER TABLE log_messages DROP IF EXISTS PARTITION(year = 2011, month = 12, day =
2);
```

The `IF EXISTS` clause is optional, as usual. For managed tables, the data for the partition is *deleted*, along with the metadata, even if the partition was created using `ALTER TABLE … ADD PARTITION`. For external tables, the data is not deleted.

Changing Columns

Rename a column, change its position, type, or comment:

```
ALTER TABLE log_messages
CHANGE COLUMN hms hours_minutes_seconds INT
COMMENT 'The hours, minutes, and seconds part of the timestamp'
AFTER severity;
```

Specify the old name, a new name, and the type, even if the name or type is not changing. The keyword `COLUMN` is optional as is the `COMMENT` clause. If not moving the column, the `AFTER other_column` clause is not necessary. If want to move the column to the first position, use `FIRST` instead of `AFTER other_column`.

As always, this command changes metadata only. If you are moving columns, the data must already match the new schema or you must change it to match by some other means.

Adding Columns

Add new columns to the end of the existing columns, before any partition columns.

```
ALTER TABLE log_messages ADD COLUMNS (
 app_name   STRING COMMENT 'Application name',
 session_id LONG   COMMENT 'The current session id');
```

The `COMMENT` clauses are optional, as usual. If any of the new columns are in the wrong position, use an `ALTER COLUMN table CHANGE COLUMN` statement for each one to move it to the correct position.

Deleting or Replacing Columns

The following example removes *all* the existing columns and replaces them with the new columns specified:

```
ALTER TABLE log_messages REPLACE COLUMNS (
 hours_mins_secs INT    COMMENT 'hour, minute, seconds from timestamp',
 severity        STRING COMMENT 'The message severity'
 message         STRING COMMENT 'The rest of the message');
```

This statement effectively renames the original `hms` column and removes the `server` and `process_id` columns from the original schema definition. As for all `ALTER` statements, only the table metadata is changed.

The `REPLACE` statement can only be used with tables.

Alter Table Properties

Add additional table properties or modify existing properties, but not remove them:

```
ALTER TABLE log_messages SET TBLPROPERTIES (
 'notes' = 'The process id is no longer captured; this column is always NULL');
```

Alter Storage Properties

There are several `ALTER TABLE` statements for modifying format and SerDe properties.

The following statement changes the storage format for a partition to be `SEQUENCEFILE.`

```
ALTER TABLE log_messages
PARTITION(year = 2012, month = 1, day = 1)
SET FILEFORMAT SEQUENCEFILE;
```

The `PARTITION` clause is required if the table is partitioned.

Alter the storage properties

```
ALTER TABLE stocks
CLUSTERED BY (exchange, symbol)
SORTED BY (symbol)
INTO 48 BUCKETS;
```

The `SORTED BY` clause is optional, but the `CLUSTER BY` and `INTO … BUCKETS` are required.

Dynamic Partition Inserts

If lot of partitions are there to create, you have to write a lot of SQL! Fortunately, Hive also supports a *dynamic partition* feature, where it can infer the partitions to create based on query parameters. By comparison, up until now we have considered only *static partitions*.

Consider this change to the previous example:

```
INSERT OVERWRITE TABLE employees
PARTITION (country, state)
```

```
SELECT ..., se.cnty, se.st
FROM staged_employees se;
```
Hive determines the values of the partition keys, `country` and `state`, from the last two columns in the `SELECT` clause. This is why we used different names in `staged_employees`, to emphasize that the relationship between the source column values and the output partition values is by *position* only and not by matching on names.
Suppose that `staged_employees` has data for a total of 100 country and state pairs. After running this query, `employees` will have 100 partitions!
You can also mix *dynamic* and *static* partitions. This variation of the previous query specifies a *static* value for the `country` (`US`) and a *dynamic* value for the `state`:
```
INSERT OVERWRITE TABLE employees
PARTITION (country = 'US', state)
SELECT ..., se.cnty, se.st
FROM staged_employees se
WHERE se.cnty = 'US';
```
The static partition keys must come before the dynamic partition keys.
Dynamic partitioning is not enabled by default. When it is enabled, it works in "strict" mode by default, where it expects at least some columns to be static. This helps protect against a badly designed query that generates a gigantic number of partitions.Dynamic partitions properties

Name	Default	Description
`hive.exec.dynamic.partition`	`false`	Set to `true` to enable dynamic partitioning.
`hive.exec.dynamic.partition.mode`	`strict`	Set to `nonstrict` to enable all partitions to be determined dynamically.
`hive.exec.max.dynamic.partitions.pernode`	`100`	The maximum number of dynamic partitions that can be created by each mapper or reducer. Raises a fatal error if one mapper or reducer attempts to create more than the threshold.
`hive.exec.max.dynamic.partitions`	`+1000`	The total number of dynamic partitions that can be created by one statement with dynamic partitioning. Raises a fatal error if the limit is exceeded.
`hive.exec.max.created.files`	`100000`	The maximum total number of files that can be created globally. A Hadoop counter is used to track the number of files created. Raises a fatal error if the limit is exceeded.

So, for example, using dynamic partitioning for all partitions might actually look this, where we set the desired properties just before use:
```
hive> set hive.exec.dynamic.partition=true;
hive> set hive.exec.dynamic.partition.mode=nonstrict;
hive> set hive.exec.max.dynamic.partitions.pernode=1000;

hive> INSERT OVERWRITE TABLE employees
  > PARTITION (country, state)
  > SELECT ..., se.cty, se.st
  > FROM staged_employees se;
```
Creating Tables and Loading Them in One Query
A table can be created and insert query results into it in one statement:
```
CREATE TABLE ca_employees
```

```
AS SELECT name, salary, address
FROM employees
WHERE se.state = 'CA';
```

This table contains just the `name`, `salary`, and `address` columns from the `employee` table records for employees in California. The schema for the new table is taken from the `SELECT` clause.

A common use for this feature is to extract a convenient subset of data from a larger, more unwieldy table.

This feature can't be used with external tables. Recall that "populating" a partition for an external table is done with an `ALTER TABLE` statement, where we aren't "loading" data, per se, but pointing metadata to a location where the data can be found.

Exporting Data

If the data files are already formatted the way you want, then it's simple enough to copy the directories or files:

hadoop fs -cp source_path target_path

Otherwise, you can use `INSERT … DIRECTORY …`, as in this example:

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/ca_employees'
SELECT name, salary, address
FROM employees
WHERE se.state = 'CA';
```

`OVERWRITE` and `LOCAL` have the same interpretations as before and paths are interpreted following the usual rules.

Just like inserting data to tables, multiple inserts to directories can be specified:

```
FROM staged_employees se
INSERT OVERWRITE DIRECTORY '/tmp/or_employees'
 SELECT * WHERE se.cty = 'US' and se.st = 'OR'
INSERT OVERWRITE DIRECTORY '/tmp/ca_employees'
 SELECT * WHERE se.cty = 'US' and se.st = 'CA'
INSERT OVERWRITE DIRECTORY '/tmp/il_employees'
 SELECT * WHERE se.cty = 'US' and se.st = 'IL';
```

Hive Queries

SELECT … FROM Clauses

SELECT is the *projection operator* in SQL. The FROM clause identifies from which table, view, or nested query we select records.

```
CREATE TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
PARTITIONED BY (country STRING, state STRING);
```

Here are queries of this table and the output they produce:

```
hive> SELECT name, salary FROM employees;
John Doe    100000.0
Mary Smith   80000.0
Todd Jones   70000.0
Bill King    60000.0
```

The following two queries are identical. The second version uses a table alias e.

```
hive> SELECT   name,   salary FROM employees;
hive> SELECT e.name, e.salary FROM employees e;
```

The deductions is a MAP, where the JSON representation for maps is used, namely a comma-separated list of key:value pairs, surrounded with {...}:

```
hive> SELECT name, deductions FROM employees;
John Doe    {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}
Mary Smith  {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}
Todd Jones  {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}
Bill King   {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}
```

Finally, the address is a STRUCT, which is also written using the JSON map format:

```
hive> SELECT name, address FROM employees;
John Doe    {"street":"1 Michigan Ave.","city":"Chicago","state":"IL","zip":60600}
Mary Smith  {"street":"100 Ontario St.","city":"Chicago","state":"IL","zip":60601}
Todd Jones  {"street":"200 Chicago Ave.","city":"Oak Park","state":"IL","zip":60700}
Bill King   {"street":"300 Obscure Dr.","city":"Obscuria","state":"IL","zip":60100}
```

Specify Columns with Regular Expressions

Use *regular expressions* to select the columns.

The following query selects the symbol column and all columns from stocks whose names start with the prefix price

```
hive> SELECT symbol, `price.*` FROM stocks;
AAPL    195.69  197.88  194.0   194.12  194.12
AAPL    192.63  196.0   190.85  195.46  195.46
AAPL    196.73  198.37  191.57  192.05  192.05
AAPL    195.17  200.2   194.42  199.23  199.23
AAPL    195.91  196.32  193.38  195.86  195.86
...
```

Computing with Column Values

Manipulate column values using function calls and arithmetic expressions.

```
hive> SELECT upper(name), salary, deductions["Federal Taxes"],
    > round(salary * (1 - deductions["Federal Taxes"])) FROM employees;
JOHN DOE    100000.0  0.2   80000
MARY SMITH   80000.0  0.2   64000
TODD JONES   70000.0  0.15  59500
```

```
BILL KING    60000.0  0.15  51000
```

Using Functions

Our tax-deduction example also uses a built-in mathematical function, round(), for finding the nearest integer for a DOUBLE value.

Mathematical functions

Return type	Signature	Description
BIGINT	round(d)	Return the BIGINT for the rounded value of DOUBLE d.
DOUBLE	round(d, N)	Return the DOUBLE for the value of d, a DOUBLE, rounded to N decimal places.
BIGINT	floor(d)	Return the largest BIGINT that is <= d, a DOUBLE.
BIGINT	ceil(d), ceiling(DOUBLE d)	Return the smallest BIGINT that is >= d.
DOUBLE	rand(), rand(seed)	Return a pseudorandom DOUBLE that changes for each row. Passing in an integer seed makes the return value deterministic.
DOUBLE	exp(d)	Return e to the d, a DOUBLE.
DOUBLE	ln(d)	Return the natural logarithm of d, a DOUBLE.
DOUBLE	log10(d)	Return the base-10 logarithm of d, a DOUBLE.
DOUBLE	log2(d)	Return the base-2 logarithm of d, a DOUBLE.
DOUBLE	log(base, d)	Return the base-base logarithm of d, where base and d are DOUBLEs.
DOUBLE	pow(d, p), power(d, p)	Return d raised to the power p, where d and p are DOUBLEs.
DOUBLE	sqrt(d)	Return the square root of d, a DOUBLE.
STRING	bin(i)	Return the STRING representing the binary value of i, a BIGINT.
STRING	hex(i)	Return the STRING representing the hexadecimal value of i, a BIGINT.
STRING	hex(str)	Return the STRING representing the hexadecimal value of s, where each two characters in the STRING s is converted to its hexadecimal representation.
STRING	unhex(i)	The inverse of hex(str).
STRING	conv(i, from_base, to_base)	Return the STRING in base to_base, an INT, representing the value of i, a BIGINT, in base from_base, an INT.
STRING	conv(str, from_base, to_base)	Return the STRING in base to_base, an INT, representing the value of str, a STRING, in base from_base, an INT.
DOUBLE	abs(d)	Return the DOUBLE that is the absolute value of d, a DOUBLE.
INT	pmod(i1, i2)	Return the positive module INT for two INTs, i1 mod i2.
DOUBLE	pmod(d1, d2)	Return the positive module DOUBLE for two DOUBLEs, d1 mod d2.
DOUBLE	sin(d)	Return the DOUBLE that is the *sin* of d, a DOUBLE, in *radians*.
DOUBLE	asin(d)	Return the DOUBLE that is the *arcsin* of d, a DOUBLE, in *radians*.
DOUBLE	cos(d)	Return the DOUBLE that is the *cosine* of d, a DOUBLE, in *radians*.
DOUBLE	acos(d)	Return the DOUBLE that is the *arccosine* of d, a DOUBLE, in

Return type	Signature	Description
		radians.
DOUBLE	tan(d)	Return the DOUBLE that is the *tangent* of d, a DOUBLE, in *radians*.
DOUBLE	atan(d)	Return the DOUBLE that is the *arctangent* of d, a DOUBLE, in *radians*.
DOUBLE	degrees(d)	Return the DOUBLE that is the value of d, a DOUBLE, converted from radians to degrees.
DOUBLE	radians(d)	Return the DOUBLE that is the value of d, a DOUBLE, converted from degrees to radians.
INT	positive(i)	Return the INT value of i (i.e., it's effectively the expression \+i).
DOUBLE	positive(d)	Return the DOUBLE value of d (i.e., it's effectively the expression \+d).
INT	negative(i)	Return the negative of the INT value of i (i.e., it's effectively the expression -i).
DOUBLE	negative(d)	Return the negative of the DOUBLE value of d; effectively, the expression -d.
FLOAT	sign(d)	Return the FLOAT value 1.0 if d, a DOUBLE, is positive; return the FLOAT value -1.0 if d is negative; otherwise return 0.0.
DOUBLE	e()	Return the DOUBLE that is the value of the constant e, 2.718281828459045.
DOUBLE	pi()	Return the DOUBLE that is the value of the constant pi, 3.141592653589793.

Note the functions floor, round, and ceil ("ceiling") for converting DOUBLE to BIGINT, which is floating-point numbers to integer numbers. These functions are the preferred technique, rather than using the *cast* operator we mentioned above.
Also, there are functions for converting integers to strings in different bases (e.g., hexadecimal).
Aggregate functions
A special kind of function is the *aggregate* function that returns a single value resulting from some computation over many rows.
More precisely, this is the *User Defined Aggregate Function*, as we'll see in Aggregate Functions.
Perhaps the two best known examples are count, which counts the number of rows (or values for a specific column), and avg, which returns the average value of the specified column values.
Here is a query that counts the number of our example employees and averages their salaries:

```
hive> SELECT count(*), avg(salary) FROM employees;
4  77500.0
```

Aggregate functions

Return type	Signature	Description
BIGINT	count(*)	Return the total number of retrieved rows, including rows containing NULL values.
BIGINT	count(expr)	Return the number of rows for which the supplied expression is not NULL.
BIGINT	count(DISTINCT expr[, expr_.])	Return the number of rows for which the supplied expression(s) are unique and not NULL.
DOUBLE	sum(col)	Return the sum of the values.

Return type	Signature	Description
DOUBLE	sum(DISTINCT col)	Return the sum of the distinct values.
DOUBLE	avg(col)	Return the average of the values.
DOUBLE	avg(DISTINCT col)	Return the average of the distinct values.
DOUBLE	min(col)	Return the minimum value of the values.
DOUBLE	max(col)	Return the maximum value of the values.
DOUBLE	var_samp(col)	Return the sample variance of a set of numbers.

Improve the performance of aggregation by setting the following property to true, hive.map.aggr, as shown here:
hive> SET hive.map.aggr=true;

hive> SELECT count(*), avg(salary) FROM employees;
This setting will attempt to do “top-level” aggregation in the map phase, as in this example. (An aggregation that isn’t top-level would be aggregation after performing a GROUP BY.) However, this setting will require more memory.

Table generating functions

Return type	Signature	Description
N rows	explode(array)	Return 0 to many rows, one row for each element from the input array.
N rows	explode(map)	(v0.8.0 and later) Return 0 to many rows, one row for each map key-value pair, with a field for each map key and a field for the map value.
tuple	json_tuple(jsonStr, p1, p2, …, pn)	Like get_json_object, but it takes multiple names and returns a tuple. All the input parameters and output column types are STRING.
tuple	parse_url_tuple(url, partname1, partname2, …, partnameN) where N >= 1	Extract N parts from a URL. It takes a URL and the partnames to extract, returning a tuple. All the input parameters and output column types are STRING. The valid partnames are case-sensitive and should only contain a minimum of white space: HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, USERINFO, QUERY:<KEY_NAME>.
N rows	stack(n, col1, …, colM)	Convert M columns into N rows of size M/N each.

Here is an example that uses parse_url_tuple where we assume a url_table exists that
Other built-in functions

Return type	Signature	Description
BOOLEAN	test in(val1, val2, …)	Return true if test equals one of the values in the list.
INT	length(s)	Return the length of the string.
STRING	reverse(s)	Return a reverse copy of the string.
STRING	concat(s1, s2, …)	Return the string resulting from s1 joined with s2, etc. For example, concat('ab', 'cd') results in 'abcd'. You can pass an arbitrary number of string arguments and the result will contain all of them joined together.
STRING	concat_ws(separator, s1, s2, …)	Like concat, but using the specified separator.

Return type	Signature	Description
STRING	substr(s, start_index)	Return the substring of s starting from the start_index position, where 1 is the index of the first character, until the end of s. For example, substr('abcd', 3) results in 'cd'.
STRING	substr(s, int start, int length)	Return the substring of s starting from the start position with the given length, e.g., substr('abcdefgh', 3, 2) results in 'cd'.
STRING	upper(s)	Return the string that results from converting all characters of s to upper case, e.g., upper('hIvE') results in 'HIVE'.
STRING	ucase(s)	A synonym for upper().
STRING	lower(s)	Return the string that results from converting all characters of s to lower case, e.g., lower('hIvE') results in 'hive'.
STRING	lcase(s)	A synonym for lower().
STRING	trim(s)	Return the string that results from removing whitespace from both ends of s, e.g., trim(' hive ') results in 'hive'.
STRING	ltrim(s)	Return the string resulting from trimming spaces from the beginning (lefthand side) of s, e.g., ltrim(' hive ') results in 'hive '.
STRING	rtrim(s)	Return the string resulting from trimming spaces from the end (righthand side) of s, e.g., rtrim(' hive ') results in ' hive'.

LIMIT Clause

The results of a typical query can return a large number of rows. The LIMIT clause puts an upper limit on the number of rows returned:

```
hive> SELECT upper(name), salary, deductions["Federal Taxes"],
   > round(salary * (1 - deductions["Federal Taxes"])) FROM employees
   > LIMIT 2;
JOHN DOE    100000.0  0.2   80000
MARY SMITH   80000.0  0.2   64000
```

Column Aliases

It’s sometimes useful to give those anonymous columns a name, called a *column alias*. Here is the previous query with column aliases for the third and fourth columns returned by the query, fed_taxes and salary_minus_fed_taxes, respectively:

```
hive> SELECT upper(name), salary, deductions["Federal Taxes"] as fed_taxes,
   > round(salary * (1 - deductions["Federal Taxes"])) as salary_minus_fed_taxes
   > FROM employees LIMIT 2;
JOHN DOE    100000.0  0.2   80000
MARY SMITH   80000.0  0.2   64000
```

Nested SELECT Statements

The column alias feature is especially useful in *nested select statements*.

```
hive> FROM (
   >   SELECT upper(name), salary, deductions["Federal Taxes"] as fed_taxes,
   >   round(salary * (1 - deductions["Federal Taxes"])) as salary_minus_fed_taxes
   >   FROM employees
   > ) e
   > SELECT e.name, e.salary_minus_fed_taxes
   > WHERE e.salary_minus_fed_taxes > 70000;
JOHN DOE    100000.0  0.2   80000
```

The previous result set is aliased as e, from which we perform a second query to select the name and the salary_minus_fed_taxes, where the latter is greater than 70,000. (We'll cover WHERE clauses in WHERE Clauses below.)

CASE … WHEN … THEN Statements

The CASE … WHEN … THEN clauses are like if statements for individual columns in query results. For example:

```
hive> SELECT name, salary,
   >   CASE
   >     WHEN salary <  50000.0 THEN 'low'
   >     WHEN salary >= 50000.0 AND salary <  70000.0 THEN 'middle'
   >     WHEN salary >= 70000.0 AND salary < 100000.0 THEN 'high'
   >     ELSE 'very high'
   >   END AS bracket FROM employees;
John Doe         100000.0   very high
Mary Smith        80000.0   high
Todd Jones        70000.0   high
Bill King        60000.0   middle
Boss Man          200000.0   very high
Fred Finance      150000.0   very high
Stacy Accountant  60000.0   middle
...
```

WHERE Clauses

While SELECT clauses select columns, WHERE clauses are filters; they select which records to return. Like SELECT clauses, we have already used many simple examples of WHERE clauses before defining the clause, on the assumption you have seen them before. Now we'll explore them in a bit more detail.

WHERE clauses use *predicate expressions*, applying *predicate operators.* Several predicate expressions can be joined with AND and OR clauses. When the predicate expressions evaluate to true, the corresponding rows are retained in the output.

Example

```
SELECT * FROM employees
WHERE country = 'US' AND state = 'CA';
```

GROUP BY Clauses

The GROUP BY statement is often used in conjunction with aggregate functions to group the result set by one or more columns and then perform an aggregation over each group.

```
hive> SELECT year(ymd), avg(price_close) FROM stocks
   > WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
   > GROUP BY year(ymd);
1984    25.578625440597534
1985    20.193676221040867
1986    32.46102808021274
1987    53.88968399108163
1988    41.540079275138766
1989    41.65976212516664
1990    37.56268799823263
1991    52.49553383386182
1992    54.80338610251119
1993    41.02671956450572
1994    34.0813495847914
...
```

HAVING Clauses

The HAVING clause lets you constrain the groups produced by GROUP BY in a way that could be expressed with a subquery, using a syntax that’s easier to express.

```
hive> SELECT year(ymd), avg(price_close) FROM stocks
   > WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
   > GROUP BY year(ymd)
      > HAVING avg(price_close) > 50.0;
1987    53.88968399108163
1991    52.49553383386182
1992    54.80338610251119
1999    57.77071460844979
2000    71.74892876261757
2005    52.401745992993554
...
```

Without the HAVING clause, this query would require a nested SELECT statement:

```
hive> SELECT s2.year, s2.avg FROM
   > (SELECT year(ymd) AS year, avg(price_close) AS avg FROM stocks
   > WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'
   > GROUP BY year(ymd)) s2
   > WHERE s2.avg > 50.0;
1987    53.88968399108163
...
```