Big Data Technologies- Part 3

Aakash Ahuja
aakash@itmtb.com
+91 7709 009634

Course overview

By the end of this course, you will know about:

- Hadoop
- Hadoop ecosystem (HBase, Hive, Spark etc)
- HDFS, MapReduce, YRAN
- Distributed data storage and access
- Hadoop administration
- Initial mapreduce programming
- Some advanced Hadoop concepts
- And more...

What to expect

- A lot of topics to be discussed
- More focus on industry practises
- Some of them will be at an introduction level given the course/time constraints
- Suitable time for labs so you can cement as many topics as possible
- Guided labs - assignment comes from me
- Autonomous labs - time for you to explore the topics at your own pace
- Assignments
- Graded evaluation at the end of this course - to be counted towards your internals
- Surprise tests

ETL - source typesetl

- Depending on source
  - Data warehouses
  - Data marts
  - Data lakes
  - flat files

- Depending on type of data
  - Structured
  - Unstructured
  - Semi structured

ETL - Source data

- Ensure metadata is known
- Ensure connectivity exists between your etl server and all source servers
- Ensure access is available

How to perform:

- Always start by referring to your high level and low level design document for metadata information.
- Check with Data Stewards of your project.
- Ask if there is a Metadata Management tool that you can have access to.

ETL - data quality checks

Types: Generic constraints, Case specific constraints

- Check for nullability
- Check for empty values (NULL is not the same as blank)
- Check for data types
- Check for negativity in case of non character data types
- Check for special characters in case of text types
- Check for specific range constraints
- Check for specific value constraints
- Check for presence or absence of values

How to perform - always start by referring to your high level and low level design document for case specific constraints.

ETL - Lookups

- Performed on common dimension tables (there can be many dimension tables)
- Dimension tables are updated each time a new entry is found (unless business rules explicitly ask for record rejection in such cases)
- A new surrogate key is generated if a new record is accepted into a dimension tables
- In mature setups, common lookup functions might exist - check.

Lookup process - Discuss.

ETL - Surrogate Key

- Machine identifiers for real world entities
- For example, You have a customer id in Swiggy's database. Your customer id is a **surrogate key**, your name is a **natural key**
- Generated using specific tools in a database or specifically written logic
- Logic:
    - Check if a **natural key** exists in lookup
    - If yes then return the corresponding **surrogate key**
    - If no then store the natural key in the corresponding **dimension table**, and generate a new surrogate key **by incrementing the last used key value**
    - Return the **newly generated surrogate key**
    - **Increment the key counter** for next use
    - Repeat for every incoming row

Surrogate keys - Discuss.

ETL - Performance constraints

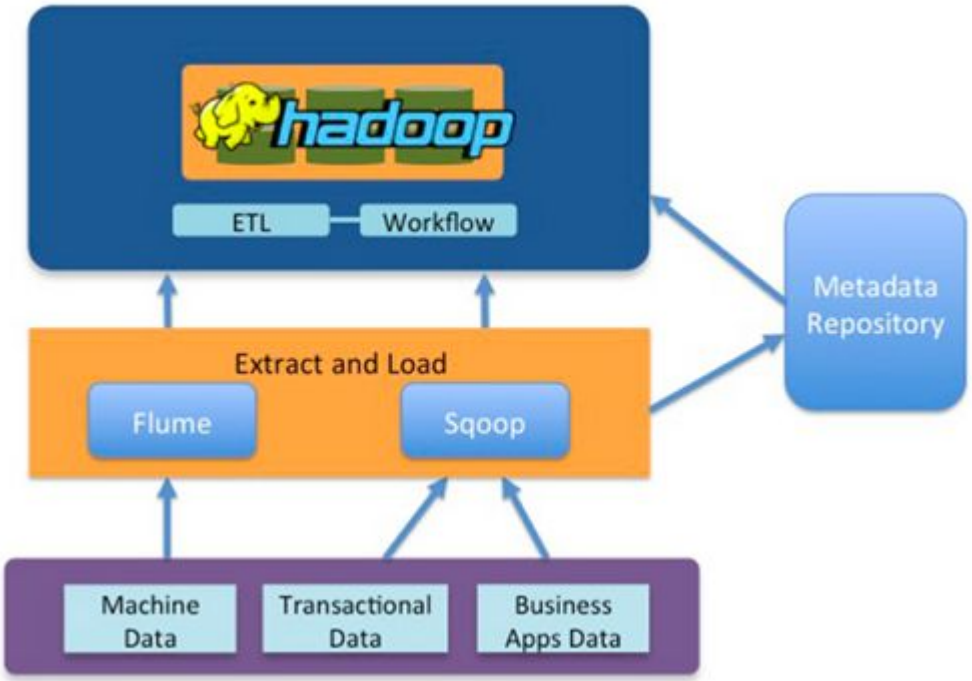
- Different best practises for ETL and database side (both read and write)
- Figure out the bottleneck - IO/CPU/Memory/Network
- Execute in parallel
- Read and write only required columns
- Use indexes and most importantly know which commands use indexes and which don't
- Drop indexes before large inserts/updates, recreate afterwards
- Truncate vs delete
- Use EXPLAIN on sqls - avoid full table scans and large, repeated joins
- Do not commit frequently

Suggested reading: SQL best practises

Hadoop for ETL

- Hadoop itself can be as the warehouse
- Ingesting data from all sources into a centralized Hadoop repository is future proof: as business scales and the data grows rapidly, the Hadoop infrastructure can scale easily
- Hadoop platform has tools that can extract the data from the source systems, whether they are log files, machine data or online databases and load them to Hadoop in record time
- It is possible to do transformations on the fly as well, although more elaborate processing is better done after the data is loaded into Hadoop
- Programming and scripting frameworks allow complex ETL jobs to be deployed and executed in a distributed manner. Rapid improvements in interactive SQL tools make Hadoop an ideal choice
- Low cost data warehouse

Hadoop for ETL



One set of typical steps to setup Hadoop for ETL:

1. Set up a Hadoop cluster
2. Connect data sources
3. Define the metadata
4. Create the ETL jobs
5. Create the workflow

Cluster setup

On the public cloud if allowed using Amazon EMR, Rackspace CBD, or other cloud Hadoop offerings.

If however your data sources happen to be in an in-house data center, there are a couple of things to take into consideration:

- Can the data be moved to the cloud? Legal, security, privacy and cost considerations apply.
- Can test data be used for development?

If the answer is No to both questions, then a cluster will need to be provisioned in the data center.

Setting data sources

The Hadoop eco-system includes several technologies such as Apache Flume and Apache Sqoop to connect various data sources such as log files, machine data and RDBMS. Depending on the amount of data and the rate of new data generation, a data ingestion architecture and topology must be planned. Start small and iterate just like any other development project. The goal is to move the data into Hadoop at a frequency that meets analytics requirements.

Metadata

Hadoop is a “schema-on-read” platform and there is no need to create a schema before loading data as databases typically require. That does not mean one can throw in any kind of data and expect some magic to happen. It is still important to clearly define the semantics and structure of data (the “metadata”) that will be used for analytics purposes. This definition will then help in the next step of data transformation.

With a clear design and documentation, there is no ambiguity in what a particular field means or how it was generated. Investing up front in getting this right will save a lot of angst later on.

With the metadata defined, this can be easily transposed to Hadoop using Apache HCatalog, a technology provides a relational table view of data in Hadoop. HCatalog also allows this view to be shared by different type of ETL jobs, Pig, Hive, or MapReduce

Create ETL jobs

MapReduce and Pig are some of the most common used frameworks for developing ETL jobs.

A word of caution – Too much cleansing can get rid of the very insights that big data promises. A thoughtful approach is required to get the most value from your data.

Create workflow

- Data mappings/transformations need to be executed in a specific order and/or there may be dependencies to check
- These dependencies and sequences are captured in workflows – parallel flows allow parallel execution that can speed up the ETL process
- Finally the entire workflow needs to be scheduled. They may have to run weekly, nightly, or perhaps even hourly
- Although technologies such as Oozie provide some workflow management, it is typically insufficient. Many organizations create their own workflow management tools. This can be a complex process as it is important to take care of failure scenarios and restart the workflow appropriately
- A smooth workflow will result in the source data being ingested and transformed based on the metadata definition and stored in Hadoop. At this point, the data is ready for analysis
- There are many different ways to do that with Hadoop; Hive, Impala and Lingual provide SQL-on-Hadoop functionality while several commercial BI tools can connect to Hadoop to explore the data visually and generate reports

Hive

In this session, you will learn about:

- Introduction to Hive
- Uses, comparisons, architecture
- Hive Query Language
- Data manipulation etc.

Hive

The Apache Hive data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage and queried using SQL syntax.

- Tools to enable easy access to data via SQL
- Good for ETL, reporting, and data analysis.
- Access to files stored either directly in Apache HDFS or in other data storage systems such as Apache HBase
- Query execution via Apache Spark, or MapReduce
- Hive's SQL can also be extended with user code via user defined functions (UDFs)

Hive

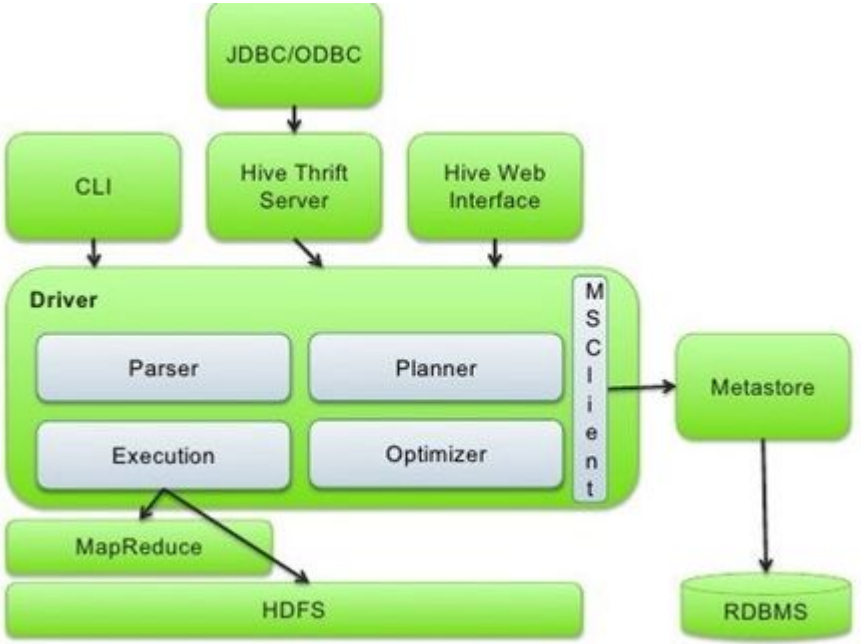
- Hive is not designed for OLTP workloads.
- Hive is designed to maximize scalability (scale out with more machines added dynamically to the Hadoop cluster), performance, extensibility, fault-tolerance, and loose-coupling with its input formats

Hive - Capabilities

Hive's SQL provides the basic SQL operations. These operations work on tables or partitions. These operations are:

- Ability to filter rows from a table using a WHERE clause.
- Ability to select certain columns from the table using a SELECT clause.
- Ability to do equi-joins between two tables.
- Ability to evaluate aggregations on multiple "group by" columns for the data stored in a table.
- Ability to store the results of a query into another table.
- Ability to download the contents of a table to a local (for example,, nfs) directory.
- Ability to store the results of a query in a hadoop dfs directory.
- Ability to manage tables and partitions (create, drop and alter).
- Ability to plug in custom scripts in the language of choice for custom map/reduce jobs.

Hive - Architecture



Hive - Components

Hadoop core components:

- HDFS: When we load the data into a Hive Table it internally stores the data in HDFS path i.e by default in hive warehouse directory.
- The hive default warehouse location can be found at hive.metastore.warehouse.dir property in hive-site.xml.
- MapReduce: When we run a query, it will run a Map Reduce job by converting or compiling the query into a java class file, building a jar and execute this jar file.
- **Metastore**: is a namespace for tables. This is a crucial part for the hive as all the metadata information related to the hive such as details related to the table, columns, partitions, location is present as part of it. Usually, the Metastore is available as part of the Relational databases eg: MySql
- You can check the DataBase configuration via hive-site.xml

Hive - Components

- Metastore has an many tables describing various properties related to the table and they in-turn helps hive infer the schema properties to execute the hive commands on a table.
- i) TBLS — Store all table information (Table name, Owner, Type of Table(Managed|External)
- ii) DBS — Database information (Location of database, Database name, Owner)
- iii) COLUMNS_V2 — column name and the datatype

- Parser: Performs semantic checks. Generates an execution plan.
- Optimizer: Optimizes the execution plan
- To check if the hive can talk to the appropriate cluster. i.e for hive to interact, query or execute with the existing cluster. You can check details under core-site.xml.

Hive - Components

- Hive Clients: It is the interface through which we can submit the hive queries. eg: hive CLI, beeline are some of the terminal interfaces, we can also use the Web-interface like Hue, Ambari to perform the same.
- On connecting to Hive via CLI or Web-Interface it establishes a connection to Metastore as well.

Hive - Installation

wget https://archive.apache.org/dist/hive/hive-2.3.7/apache-hive-2.3.7-bin.tar.gz

tar -zxvf apache-hive-2.3.7-bin.tar.gz

ln -s apache-hive-2.3.7-bin hive

update ~/.bashrc

export HIVE_HOME=/usr/local/hive

export PATH=$HIVE_HOME/bin:$PATH
export CLASSPATH=$CLASSPATH:/usr/local/hive/lib/*:.

source ~/.bashrc

hdfs dfs -mkdir /tmp

hdfs dfs -mkdir -p /user/hive/warehouse

hdfs dfs -chmod g+w /tmp

hdfs dfs -chmod -R g+w /user/hive

cp hive*xml* hive-site.xml - run from /usr/local/hive/conf

– RUN below command from your home directory
$HIVE_HOME/bin/schematool -dbType derby -initSchema

Assumptions:
Hadoop is installed and configured.
HADOOP_VERSION set in bashrc as x.y.z (2.7.7, for example)

Before we start...

Addition in hive-site.xml

```
<property>
  <name>system:java.io.tmpdir</name>
  <value>/tmp/hive/java</value>
 </property>
 <property>
  <name>system:user.name</name>
  <value>${user.name}</value>
 </property>
```

Hive - best practises

Partitioning Tables

- Hive partitioning is an effective method to improve the query performance on large tables.
- Partitioning allows you to store data in separate sub-directories under table location.
- It greatly helps the queries which are queried upon the partition key(s).
- Partition key should always be a low cardinal attribute.

Hive - best practises

De-normalizing data

- Normalization is a standard process used to model your data tables with certain rules to deal with redundancy of data and anomalies.
- If you normalize your data sets, you end up creating multiple relational tables which can be joined at the run time to produce the results.
- Joins are expensive and difficult operations to perform and are one of the common reasons for performance issues.

Hive - best practises

Compress map/reduce output

- Compression techniques significantly reduce the intermediate data volume, which internally reduces the amount of data transfers between mappers and reducers. All this generally occurs over the network.
- Compression can be applied on the mapper and reducer output individually. Keep in mind that gzip compressed files are not splittable.
- A compressed file size should not be larger than a few hundred megabytes.
    - For map output compression set **mapred.compress.map.output** to true
    - For job output compression set **mapred.output.compress** to true

Hive - best practises

Map join

- Map joins are really efficient if a table on the other side of a join is small enough to fit in the memory.
- Hive supports a parameter, **hive.auto.convert.join**, which when it's set to "true" suggests that Hive try to map join automatically.

Hive - best practises

Bucketing

- Bucketing improves the join performance if the bucket key and join keys are common.
- Bucketing in Hive distributes the data in different buckets based on the hash results on the bucket key. It also reduces the I/O scans during the join process if the process is happening on the same keys (columns).
- Set the bucketing flag(**SET hive.enforce.bucketing=true;**) every time before writing data to the bucketed table. For bucketing in the join operation we should **SET hive.optimize.bucketmapjoin=true**. With this Hive does bucket level join during the map stage join.
- It reduces the scans to find a particular key.

Hive - best practises

Input Format Selection

- Input formats play a critical role in Hive performance.
- JSON is not a good choice for a large production system where data volume is really high. These type of readable formats actually take a lot of space and have some overhead of parsing (e.g JSON parsing).
- To address these problems, Hive comes with columnar input formats like RCFile, ORC etc.
- Columnar formats allow you to reduce the read operations in analytics queries by allowing each column to be accessed individually. There are some other binary formats like Avro, sequence files which can be helpful in various use cases too.

Hive - best practises

Parallel execution

- Hadoop can execute MapReduce jobs in parallel, and several queries executed on Hive automatically use this parallelism.
- However, single, complex Hive queries commonly are translated to a number of MapReduce jobs that are executed by default sequencing.
- Often though, some of a query's MapReduce stages are not interdependent and could be executed in parallel. The configuration in Hive to change this behavior is merely switching a single flag **SET hive.exec.parallel=true**.

Hive Data Units

In the order of granularity - Hive data is organized into:

- **Databases**: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on. Databases can also be used to enforce security for a user or group of users.
- **Tables**: Homogeneous units of data which have the same schema.
- **Partitions**: Each Table can have partition Keys which determines how the data is stored.
- **Buckets** (or **Clusters**): Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table. For example the page_views table may be bucketed by userid, which is one of the columns, other than the partitions columns, of the page_view table. These can be used to efficiently sample the data.

Note that it is not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.

Hive - Primitive data types

- Integers
  - TINYINT—1 byte integer
  - SMALLINT—2 byte integer
  - INT—4 byte integer
  - BIGINT—8 byte integer
- Boolean type
  - BOOLEAN—TRUE/FALSE
- Floating point numbers
  - FLOAT—single precision
  - DOUBLE—Double precision
- Fixed point numbers
  - DECIMAL—a fixed point value of user defined scale and precision
- String types
  - STRING—sequence of characters in a specified character set
  - VARCHAR—sequence of characters in a specified character set with a maximum length
  - CHAR—sequence of characters in a specified character set with a defined length
- Date and time types
  - TIMESTAMP — A date and time without a timezone ("LocalDateTime" semantics)
  - TIMESTAMP WITH LOCAL TIME ZONE — A point in time measured down to nanoseconds ("Instant" semantics)
  - DATE—a date
- Binary types
  - BINARY—a sequence of bytes

Hive - Complex data types

Complex Types can be built up from primitive types and other composite types using:

- Structs: the elements within the type can be accessed using the DOT (.) notation. For example, for a column c of type STRUCT {a INT; b INT}, the a field is accessed by the expression c.a
- Maps (key-value tuples): The elements are accessed using ['element name'] notation. For example in a map M comprising of a mapping from 'group' -> gid the gid value can be accessed using M['group']
- Arrays (indexable lists): The elements in the array have to be in the same type. Elements can be accessed using the [n] notation where n is an index (zero-based) into the array. For example, for an array A having the elements ['a', 'b', 'c'], A[1] retruns 'b'.

Using the primitive types and the constructs for creating complex types, types with arbitrary levels of nesting can be created. For example, a type User may comprise of the following fields:

- gender—which is a STRING.
- active—which is a BOOLEAN.

Hive - Timestamps

Supports traditional UNIX timestamp with optional nanosecond precision.

Supported conversions:

- Integer numeric types: Interpreted as UNIX timestamp in seconds
- Floating point numeric types: Interpreted as UNIX timestamp in seconds with decimal precision
- Strings: JDBC compliant java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)

Timestamps are interpreted to be timezoneless and stored as an offset from the UNIX epoch. Convenience UDFs for conversion to and from timezones are provided (to_utc_timestamp, from_utc_timestamp).

- All existing datetime UDFs (month, day, year, hour, etc.) work with the TIMESTAMP data type.
- Timestamps in text files have to use the format yyyy-mm-dd hh:mm:ss[.f...]. If they are in another format, declare them as the appropriate type (INT, FLOAT, STRING, etc.) and use a UDF to convert them to timestamps.

Hive - Dates

DATE values describe a particular year/month/day, in the form YYYY-MM-DD. The range of values supported for the Date type is 0000-01-01 to 9999-12-31, dependent on support by the primitive Java Date type.

Date types can only be converted to/from Date, Timestamp, or String types. Casting with user-specified formats is documented here.

cast(date as date)

Same date value

cast(date as string)

The year/month/day represented by the Date is formatted as a string in the form 'YYYY-MM-DD'.

cast(timestamp as date)

The year/month/day of the timestamp is determined, based on the local timezone, and returned as a date value.

cast(string as date)

If the string is in the form 'YYYY-MM-DD', then a date value corresponding to that year/month/day is returned. If the string value does not match this format, then NULL is returned.

cast(date as timestamp)

A timestamp value is generated corresponding to midnight of the year/month/day of the date value, based on the local timezone.

Hive - Literals

Floating Point Types

Floating point literals are assumed to be DOUBLE.

Decimal Types

Decimal literals provide precise values and greater range for floating point numbers than the DOUBLE type.

Decimal data types store exact representations of numeric values, while DOUBLE data types store very close approximations of numeric values.

Decimal types are needed for use cases in which the (very close) approximation of a DOUBLE is insufficient, such as financial applications, equality and inequality checks, and rounding operations.

They are also needed for use cases that deal with numbers outside the DOUBLE range.

The precision of a Decimal type is limited to 38 digits in Hive.

Hive - Intervals

Supported Interval Description	Example	Meaning
Intervals of time units: SECOND / MINUTE / DAY / MONTH / YEAR	INTERVAL '1' DAY	an interval of 1 day(s)
Year to month intervals, format: SY-M S: optional sign (+/-) Y: year count M: month count	INTERVAL '1-2' YEAR TO MONTH	shorthand for: INTERVAL '1' YEAR + INTERVAL '2' MONTH
Day to second intervals, format: SD H:M:S.nnnnnn S: optional sign (+/-) D: day countH: hours M: minutes S: seconds nnnnnn: optional nanotime	INTERVAL '1 2:3:4.000005' DAY	shorthand for: INTERVAL '1' DAY+ INTERVAL '2' HOUR + INTERVAL '3' MINUTE + INTERVAL '4' SECOND + INTERVAL '5' NANO
Support for intervals with constant numbers	INTERVAL 1 DAY	aids query readability / portability
Support for intervals with expressions: this may involve other functions/columns. The expression must return with a number (which is not floating-point) or with a string.	INTERVAL (1+dt) DAY	enables dynamic intervals

Hive - Commands

quit /exit

Use quit or exit to leave the interactive shell.

reset

Resets the configuration to the default values.

set <key>=<value>

Sets the value of a particular configuration variable (key). If you misspell the variable name, the CLI will not show an error.

set

Prints a list of configuration variables that are overridden by the user or Hive.

set -v

Prints all Hadoop and Hive configuration variables.

add [FILE[S] | ARCHIVE[S] | JAR[S]] <filepath> <filepath>*

Adds one or more files, jars, or archives to the list of resources in the distributed cache.

list FILE[S] | JAR[S] | ARCHIVE[S]

Lists the resources already added to the distributed cache.

delete FILE[S] | JAR[S] | ARCHIVE[S] <filepath>*

Removes the resource(s) from the distributed cache.

Hive - Commands

`! <command>`

Executes a shell command from the Hive shell.

`dfs <dfs command>`

Executes a dfs command from the Hive shell.

`<query string>`

Executes a Hive query and prints results to standard output.

`source FILE <filepath>`

Executes a script file inside the CLI.

Hive - Resources

Hive can manage the addition of resources to a session where those resources need to be made available at query execution time.

The resources can be files, jars, or archives. Any locally accessible file can be added to the session.

Once a resource is added to a session, Hive queries can refer to it by its name (in map/reduce/transform clauses) and the resource is available locally at execution time on the entire Hadoop cluster.

Hive uses Hadoop's Distributed Cache to distribute the added resources to all the machines in the cluster at query execution time.

```
hive> add FILE /tmp/tt.py;

 hive> list FILES;

 /tmp/tt.py

 hive> select from networks a

        MAP a.networkid
        USING 'python tt.py' as nn where a.ds = '2009-01-04'
limit 10;
```

Hive - DDL

Create database

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS]
database_name
 [COMMENT database_comment]
 [LOCATION hdfs_path]
 [WITH DBPROPERTIES (property_name=property_value,
...)];
```

The uses of SCHEMA and DATABASE are interchangeable – they mean the same thing.

Drop database

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name
[RESTRICT|CASCADE];
```

This fails if the database is not empty. To drop the tables in the database as well, use DROP DATABASE ... CASCADE.

Hive - DDL

Alter database

```
ALTER (DATABASE|SCHEMA) database_name SET
DBPROPERTIES (property_name=property_value, ...);

ALTER (DATABASE|SCHEMA) database_name SET
OWNER [USER|ROLE] user_or_role; .0)

ALTER (DATABASE|SCHEMA) database_name SET
LOCATION hdfs_path;
```

The ALTER DATABASE ... SET LOCATION statement does not move the contents of the database's current directory to the newly specified location.

It does not change the locations associated with any tables/partitions under the specified database.

It only changes the default parent-directory where new tables will be added for this database.

No other metadata about a database can be changed.

Hive - DDL

Use database

```
USE database_name;
USE DEFAULT;
```

USE sets the current database for all subsequent HiveQL statements.

To revert to the default database, use the keyword "default" instead of a database name.

To check which database is currently being used: SELECT current_database().

Hive - DDL

Create table

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT
EXISTS] [db_name.]table_name
 [(col_name data_type [column_constraint_specification]
[COMMENT col_comment], ... [constraint_specification])]
 [COMMENT table_comment]
 [PARTITIONED BY (col_name data_type [COMMENT
col_comment], ...)]
 [CLUSTERED BY (col_name, col_name, ...) [SORTED BY
(col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS]
 [SKEWED BY (col_name, col_name, ...) ]
   ON ((col_value, col_value, ...), (col_value, col_value, ...), ...)
   [STORED AS DIRECTORIES]
[
  [ROW FORMAT row_format]
  [STORED AS file_format]
    | STORED BY 'storage.handler.class.name' [WITH
SERDEPROPERTIES (...)]
 ]
 [LOCATION hdfs_path]
 [TBLPROPERTIES (property_name=property_value, ...)]
-- (Note: Available in Hive 0.6.0 and later)
 [AS select_statement];
```

Hive - DDL

Create table contd…

row_format
 : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS TERMINATED BY char]
 [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]
 [NULL DEFINED AS char]
 | SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value, ...)]

file_format:
 : SEQUENCEFILE
 | TEXTFILE -- (Default, depending on hive.default.fileformat configuration)
 | RCFILE
 | ORC
 | PARQUET
 | AVRO
 | JSONFILE
 | INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

column_constraint_specification:
 : [PRIMARY KEY|UNIQUE|NOT NULL|DEFAULT [default_value]|CHECK [check_expression] ENABLE|DISABLE NOVALIDATE RELY/NORELY]

Hive - DDL

Create table contd…

Managed tables

- A managed table is stored under the hive.metastore.warehouse.dir path property, by default.
- Hive assumes that it owns the data for managed tables. That means that the data, its properties and data layout will and can only be changed via Hive command.
- The data still lives in a normal file system.
- Whenever you change an entity (e.g. drop a table) the data is also changed (in this case the data is deleted).
- Use managed tables when Hive should manage the lifecycle of the table, or when generating temporary tables.

Hive - DDL

Create table contd…

External tables

- For external tables Hive assumes that it does not manage the data.
- An external table describes the metadata / schema on external files.
- External table files can be accessed and managed by processes outside of Hive.
- Use external tables when files are already present or in remote locations, and the files should remain even if the table is dropped.

Hive - DDL

Create table contd…

Storage formats

INPUTFORMAT and OUTPUTFORMAT

In the file_format to specify the name of a corresponding InputFormat and OutputFormat class as a string literal.

For example, 'org.apache.hadoop.hive.contrib.fileformat.base64.Base64TextInputFormat'.

STORED AS TEXTFILE

Stored as plain text files. TEXTFILE is the default file format, unless the configuration parameter hive.default.fileformat has a different setting.

Hive - DDL

Create table contd…

continued on next slide.

Partitioned Tables

Partitioned tables can be created using the PARTITIONED BY clause.

A table can have one or more partition columns and a separate data directory is created for each distinct value combination in the partition columns.

Further, tables or partitions can be bucketed using CLUSTERED BY columns, and data can be sorted within that bucket via SORT BY columns. This can improve performance on certain kinds of queries.

Hive - DDL

Create table contd…

Partitioned Tables

If, when creating a partitioned table, you get this error: "FAILED: Error in semantic analysis: Column repeated in partitioning columns," it means you are trying to include the partitioned column in the data of the table itself. You probably really do have the column defined. However, the partition you create makes a pseudocolumn on which you can query, so you must rename your table column to something else (that users should not query on!).

For example, suppose your original unpartitioned table had three columns: id, date, and name.

```
id    int,
date   date,
name   varchar
```

```
create table table_name (
 id              int,
 dtDontQuery     string,
 name            string
)
partitioned by (date string)
```

Hive - DDL

Create table contd…

Create Table As Select (CTAS)

Tables can also be created and populated by the results of a query in one create-table-as-select (CTAS) statement.

The table created by CTAS is atomic, meaning that the table is not seen by other users until all the query results are populated.

So other users will either see the table with the complete results of the query or will not see the table at all.

CTAS has these restrictions:

- The target table cannot be an external table.
- The target table cannot be a bucketing table.

```
CREATE TABLE new_key_value_store
  AS
SELECT (key % 1024) new_key, concat(key, value)
key_value_pair
FROM key_value_store
SORT BY new_key, key_value_pair;
```

Hive - DDL

Create table contd…

Create Table Like

The LIKE form of CREATE TABLE allows you to copy an existing table definition exactly (without copying its data). The new table contains no rows.

CREATE TABLE empty_key_value_store

LIKE key_value_store [TBLPROPERTIES

(property_name=property_value, ...)];

Hive - DDL

Create table contd…

Bucketed Sorted Tables

Here the table is bucketed (clustered by) and within each bucket the data is sorted in increasing order of viewTime.

Such an organization allows the user to do efficient sampling on the clustered column - in this case userid.

The CLUSTERED BY and SORTED BY creation commands do not affect how data is inserted into a table – only how it is read. This means that users must be careful to insert data correctly by specifying the number of reducers to be equal to the number of buckets, and using CLUSTER BY and SORT BY commands in their query.

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
    page_url STRING, referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User')
 COMMENT 'This is the page view table'
 PARTITIONED BY(dt STRING, country STRING)
 CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS
 ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\001'
  COLLECTION ITEMS TERMINATED BY '\002'
  MAP KEYS TERMINATED BY '\003'
 STORED AS SEQUENCEFILE;
```

Hive - DDL

Create table contd…

Skewed Tables

This feature can be used to improve performance for tables where one or more columns have skewed values.

By specifying the values that appear very often (heavy skew) Hive will split those out into separate files (or directories in case of list bucketing) automatically and take this fact into account during queries so that it can skip or include the whole file (or directory in case of list bucketing) if possible.

This can be specified on a per-table level during table creation.

```
CREATE TABLE list_bucket_multiple (col1 STRING, col2
int, col3 STRING)
 SKEWED BY (col1, col2) ON (('s1',1), ('s3',3), ('s13',13),
('s78',78)) [STORED AS DIRECTORIES];
```

Hive - DDL

Create table contd…

Temporary Tables

A table that has been created as a temporary table will only be visible to the current session. Data will be stored in the user's scratch directory, and deleted at the end of the session.

If a temporary table is created with a database/table name of a permanent table which already exists in the database, then within that session any references to that table will resolve to the temporary table, rather than to the permanent table. The user will not be able to access the original table within that session without either dropping the temporary table, or renaming it to a non-conflicting name.

Temporary tables have the following limitations:

- Partition columns are not supported.
- No support for creation of indexes.

```
CREATE TEMPORARY TABLE list_bucket_multiple (col1 STRING, col2 int, col3 STRING);
```

Hive - DDL

Create table contd…

Constraints

Hive includes support for non-validated primary and foreign key constraints. Some SQL tools generate more efficient queries when constraints are present. Since these constraints are not validated, an upstream system needs to ensure data integrity before it is loaded into Hive.

```
create table constraints1(id1 integer UNIQUE disable
novalidate, id2 integer NOT NULL,
 usr string DEFAULT current_user(), price double CHECK
(price > 0 AND price <= 1000));

create table constraints2(id1 integer, id2 integer,
 constraint c1_unique UNIQUE(id1) disable novalidate);

create table constraints3(id1 integer, id2 integer,
 constraint c1_check CHECK(id1 + id2 > 0));
```

Hive - DDL

DROP table

DROP TABLE removes metadata and data for this table. The data is actually moved to the .Trash/Current directory if Trash is configured (and PURGE is not specified). The metadata is completely lost.

When dropping an EXTERNAL table, data in the table will *NOT* be deleted from the file system.

When dropping a table referenced by views, no warning is given (the views are left dangling as invalid and must be dropped or recreated by the user).

In many cases, this results in the table data being moved into the user's .Trash folder in their home directory; users who mistakenly DROP TABLEs may thus be able to recover their lost data by recreating a table with the same schema, recreating any necessary partitions, and then moving the data back into place manually using Hadoop.

This solution is subject to change over time.

DROP TABLE [IF EXISTS] table_name [PURGE];

Hive - DML

Inserts

Standard syntax:

INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select_statement1 FROM from_statement;

INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement;

Hive - DDL

Alter table

Rename Table

```
ALTER TABLE table_name RENAME TO new_table_name;
```

Alter Table Properties

```
ALTER TABLE table_name SET TBLPROPERTIES
table_properties;

table_properties:
 : (property_name = property_value, property_name =
property_value, ... )
```

Alter Table Comment

```
ALTER TABLE table_name SET TBLPROPERTIES
('comment' = new_comment);
```

Hive - DDL

Alter Column

```
ALTER TABLE table_name [PARTITION partition_spec]
CHANGE [COLUMN] col_old_name col_new_name
column_type
 [COMMENT col_comment] [FIRST|AFTER column_name]
[CASCADE|RESTRICT];
```

// First change column a's name to a1.

```
ALTER TABLE test_change CHANGE a a1 INT;
```

// Next change column a1's name to a2, its data type to string, and put it after column b.

```
ALTER TABLE test_change CHANGE a1 a2 STRING AFTER
b;
```

// The new table's structure is: b int, a2 string, c int.

// Then change column c's name to c1, and put it as the first column.

```
ALTER TABLE test_change CHANGE c c1 INT FIRST;
```

// The new table's structure is: c1 int, b int, a2 string.

// Add a comment to column a1

```
ALTER TABLE test_change CHANGE a1 a1 INT
COMMENT 'this is column a1';
```

Hive - DDL

Create View

```
CREATE VIEW [IF NOT EXISTS] [db_name.]view_name
[(column_name [COMMENT column_comment], ...) ]
 [COMMENT view_comment]
 [TBLPROPERTIES (property_name = property_value, ...)]
 AS SELECT ...;
```

Example:

```
CREATE VIEW onion_referrers(url COMMENT 'URL of
Referring page')
 COMMENT 'Referrers to The Onion website'
 AS
 SELECT DISTINCT referrer_url
 FROM page_view
 WHERE page_url='http://www.theonion.com';
```

- CREATE VIEW creates a view with the given name. An error is thrown if a table or view with the same name already exists. You can use IF NOT EXISTS to skip the error.
- Use SHOW CREATE TABLE to display the CREATE VIEW statement that created a view. As of Hive 2.2.0, SHOW VIEWS displays a list of views in a database.
- Drop View
- DROP VIEW [IF EXISTS] [db_name.]view_name;
- DROP VIEW removes metadata for the specified view.

Hive - DDL

Alter view

```
ALTER VIEW [db_name.]view_name SET
TBLPROPERTIES table_properties;

table_properties:
 : (property_name = property_value,
property_name = property_value, ...)


ALTER VIEW [db_name.]view_name AS
select_statement;
```

Hive - DDL

Create Temporary Macro

```
CREATE TEMPORARY MACRO macro_name([col_name col_type, ...]) expression;
```

CREATE TEMPORARY MACRO creates a macro using the given optional list of columns as inputs to the expression. Macros exist for the duration of the current session.

Examples:

```
CREATE TEMPORARY MACRO fixed_number() 42;
CREATE TEMPORARY MACRO string_len_plus_two(x string) length(x) + 2;
CREATE TEMPORARY MACRO simple_add (x int, y int) x + y;
```

Drop Temporary Macro

```
DROP TEMPORARY MACRO [IF EXISTS] macro_name;
```

Create Temporary Function

```
CREATE TEMPORARY FUNCTION function_name AS class_name;
```

This statement lets you create a function that is implemented by the class_name. You can use this function in Hive queries as long as the session lasts.

Hive - DDL

Drop Temporary Function

DROP TEMPORARY FUNCTION [IF EXISTS] function_name;

CREATE FUNCTION [db_name.]function_name AS class_name [USING JAR|FILE|ARCHIVE 'file_uri' [, JAR|FILE|ARCHIVE 'file_uri']];

This statement lets you create a function that is implemented by the class_name. Jars, files, or archives which need to be added to the environment can be specified with the USING clause; when the function is referenced for the first time by a Hive session, these resources will be added to the environment as if ADD JAR/FILE had been issued.

Drop Function

DROP FUNCTION [IF EXISTS] function_name;

Reload Function

RELOAD (FUNCTIONS|FUNCTION);

Issuing RELOAD FUNCTIONS within a HiveServer2 or HiveCLI session will allow it to pick up any changes to the permanent functions that may have been done by a different HiveCLI session.

Hive - DDL

Show Databases

SHOW (DATABASES|SCHEMAS) [LIKE 'identifier_with_wildcards'];

SHOW DATABASES or SHOW SCHEMAS lists all of the databases defined in the metastore.
The uses of SCHEMAS and DATABASES are interchangeable – they mean the same thing.

The optional LIKE clause allows the list of databases to be filtered using a regular expression. Wildcards in the regular expression can only be '*' for any character(s) or '|' for a choice.

Show Tables

SHOW TABLES [IN database_name] ['identifier_with_wildcards'];

Show Views

SHOW VIEWS [IN/FROM database_name] [LIKE 'pattern_with_wildcards'];

SHOW TBLPROPERTIES tblname;

SHOW TBLPROPERTIES tblname("foo");

Hive - DML

Loading files into tables

Hive does not do any transformation while loading data into tables. Load operations are currently pure copy/move operations that move datafiles into locations corresponding to Hive tables.

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE]
INTO TABLE tablename [PARTITION (partcol1=val1,
partcol2=val2 ...)]
```

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE]
INTO TABLE tablename [PARTITION (partcol1=val1,
partcol2=val2 ...)] [INPUTFORMAT 'inputformat' SERDE
'serde'] (3.0 or later)
```

- *filepath* can be:
  - a relative path, such as project/data1
  - an absolute path, such as /user/hive/project/data1
  - a full URI with scheme and (optionally) an authority, such as hdfs://namenode:9000/user/hive/project/data1
- The target being loaded to can be a table or a partition. If the table is partitioned, then one must specify a specific partition of the table by specifying values for all of the partitioning columns.
- *filepath* can refer to a file (in which case Hive will move the file into the table) or it can be a directory (in which case Hive will move all the files within that directory into the table). In either case, *filepath* addresses a set of files.

Hive - DML

Insert contd…

- INSERT OVERWRITE will overwrite any existing data in the table or partition
  - unless IF NOT EXISTS is provided for a partition.
- INSERT INTO will append to the table or partition, keeping the existing data intact.

Hive - DML

Insert values

Standard Syntax:
INSERT INTO TABLE tablename [PARTITION (partcol1[=val1], partcol2[=val2] ...)] VALUES values_row [, values_row ...]

Where values_row is:
(value [, value ...])
where a value is either null or any valid SQL literal

- Each row listed in the VALUES clause is inserted into table *tablename*.
- Values must be provided for every column in the table. The standard SQL syntax that allows the user to insert values into only some columns is not yet supported.
- Dynamic partitioning is supported in the same way as for INSERT...SELECT.
- Hive does not support literals for complex types (array, map, struct, union), so it is not possible to use them in INSERT INTO...VALUES clauses. This means that the user cannot insert data into a complex datatype column using the INSERT INTO...VALUES clause.

Hive - DDL

Update table

UPDATE tablename SET column = value [, column = value ...] [WHERE expression]

- The referenced column must be a column of the table Partitioning columns cannot be updated.
- Bucketing columns cannot be updated.

Hive - DDL

Import/Export

The EXPORT command exports the data of a table or partition, along with the metadata, into a specified output location.

IMPORT will create target table/partition if it does not exist.

```
EXPORT TABLE tablename [PARTITION
(part_column="value"[, ...])]
 TO 'export_target_path' [ FOR
replication('eventid') ]
```

```
IMPORT [[EXTERNAL] TABLE
new_or_original_tablename [PARTITION
(part_column="value"[, ...])]]
 FROM 'source_path'
 [LOCATION 'import_target_path']
```

Hive - Data Retrieval

```
[WITH CommonTableExpression (, CommonTableExpression)*]    (Note: Only available starting with Hive 0.13.0)
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
  FROM table_reference
  [WHERE where_condition]
  [GROUP BY col_list]
  [ORDER BY col_list]
  [CLUSTER BY col_list
    | [DISTRIBUTE BY col_list] [SORT BY col_list]
  ]
 [LIMIT [offset,] rows]
```

- A SELECT statement can be part of a union query or a subquery of another query.
- table_reference indicates the input to the query. It can be a regular table, a view, a join construct or a subquery.
- Table names and column names are case insensitive.

Hive - Data Retrieval

Group by

groupByClause: GROUP BY groupByExpression (, groupByExpression)

groupByQuery: SELECT expression (, expression)* FROM src groupByClause

```
SELECT pv_users.gender, count(DISTINCT pv_users.userid), count(*), sum(DISTINCT pv_users.userid)
FROM pv_users
GROUP BY pv_users.gender;
```

Hive - Data Retrieval

Order by

colOrder: (ASC | DESC)
colNullOrder: (NULLS FIRST | NULLS LAST) -- (Note: Available in Hive 2.1.0 and later)
orderBy: ORDER BY colName colOrder? colNullOrder? (',' colName colOrder? colNullOrder?)*
query: SELECT expression (',' expression)* FROM src orderBy

Sort by

colOrder: (ASC | DESC)
sortBy: SORT BY colName colOrder (',' colName colOrder)*
query: SELECT expression (',' expression)* FROM src sortBy

The difference between "order by" and "sort by" is that the former guarantees total order in the output while the latter only guarantees ordering of the rows within a reducer. If there are more than one reducer, "sort by" may give partially ordered final results.

Hive - Data Retrieval

Union

select_statement UNION [ALL | DISTINCT]
select_statement UNION [ALL | DISTINCT]
select_statement …

UNION is used to combine the result from multiple SELECT statements into a single result set.

You can mix UNION ALL and UNION DISTINCT in the same query.

The number and names of columns returned by each *select_statement* have to be the same. Otherwise, a schema error is thrown.

Union within a from clause

```
SELECT *
FROM (
  select_statement
  UNION ALL
  select_statement
) unionResult
```

```
SELECT key FROM (SELECT key FROM src ORDER BY key
LIMIT 10)subq1
UNION
SELECT key FROM (SELECT key FROM src1 ORDER BY
key LIMIT 10)subq2
```

Hive - Data Retrieval

Join

join_table:
  table_reference [INNER] JOIN table_factor [join_condition]
 | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference join_condition
 | table_reference LEFT SEMI JOIN table_reference join_condition
 | table_reference CROSS JOIN table_reference [join_condition] (as of Hive 0.10)

table_reference:
  table_factor
 | join_table

table_factor:
  tbl_name [alias]
 | table_subquery alias
 | (table_references)

join_condition:
  ON expression

Hive converts joins over multiple tables into a single map/reduce job if for every table the same column is used in the join clauses

Hive - Data Retrieval

Explain

EXPLAIN
[EXTENDED|CBO|AST|DEPENDENCY|AUTHORIZATION|LOCKS|VECTORIZATION|ANALYZE] query

The use of EXTENDED in the EXPLAIN statement produces extra information about the operators in the plan. This is typically physical information like file names.

A Hive query gets converted into a sequence (it is more a Directed Acyclic Graph) of stages. These stages may be map/reduce stages or they may even be stages that do metastore or file system operations like move and rename. The explain output has three parts:

- The Abstract Syntax Tree for the query
- The dependencies between the different stages of the plan
- The description of each of the stages