



Web: Inceptez.com Mail: info@inceptez.com Call: 7871299810, 7871299817



Apache Hive

Introduction to Hive

Hive is a data warehousing infrastructure based on Hadoop.

- An Abstraction on top of MapReduce.
 - Allows users to query data in the Hadoop cluster without knowing Java or Map Reduce.
 - Structured/semi structured Data in HDFS logically into Tables.
 - Uses the HiveQL Language.
 - Very similar to SQL.
 - Turns HiveQL into Map Reduce Jobs.
- A system for querying and managing structured data built on top of Hadoop
 - Uses Map-Reduce for execution
 - HDFS for storage – but any system that implements Hadoop FS API
 - Key Building Principles:
 - Structured data with rich data types (structs, lists and maps)
 - Directly query data from different formats (text/binary) and file formats (Flat/Sequence)
 - SQL as a familiar programming tool and for standard analytics
 - Allow embedded scripts for extensibility and for nonstandard applications Rich Meta Data to allow data discovery and for optimization

Why Hive

Hive is designed to enable easy data summarization, ad-hoc querying and analysis of large volumes of data. It provides a simple query language called Hive QL, which is based on SQL and which enables users familiar with SQL to do ad-hoc querying, summarization and data analysis easily. At the same time, Hive QL also allows traditional map/reduce programmers to be able to plug in their custom mappers and reducers to do more sophisticated analysis that may not be supported by the built-in capabilities of the language. Comparison between ETL and ELT.

Mix of ETL and ELT

ELT	ETL
<ul style="list-style-type: none">▪ Generally better in Flexibility▪ More suitable for simpler and well-defined formats▪ More applicable for experimentation▪ XML data parsed on demand for every query	<ul style="list-style-type: none">▪ Generally better in Performance▪ More suitable when substantial cleansing and reformatting is needed▪ Repetitive queries and production workloads▪ XML Data pre-parsed to minimize resource usage

ETL vs. ELT

- Hadoop generally built for EL – T
 - aka Schema-on-Read
 - Load as-is
 - Transform on Access/Query
- Compare with Data Warehouse ETL
 - Aka Schema-on-Write
 - Transform and Load
 - Queries are lot simpler
 - Transformation and cleansing done *a priori*

Brief History

Team of engineers from Facebook started the development of Hive on 08/2007 to bring an alternative approach for analyzing data in HDFS other than writing core MapReduce program where most of the users are familiar in SQL than Java programming due to the complexity and huge coding efforts, and the open source release to ASF happened on 08/2008. Hive is the original SQL framework on Hadoop.

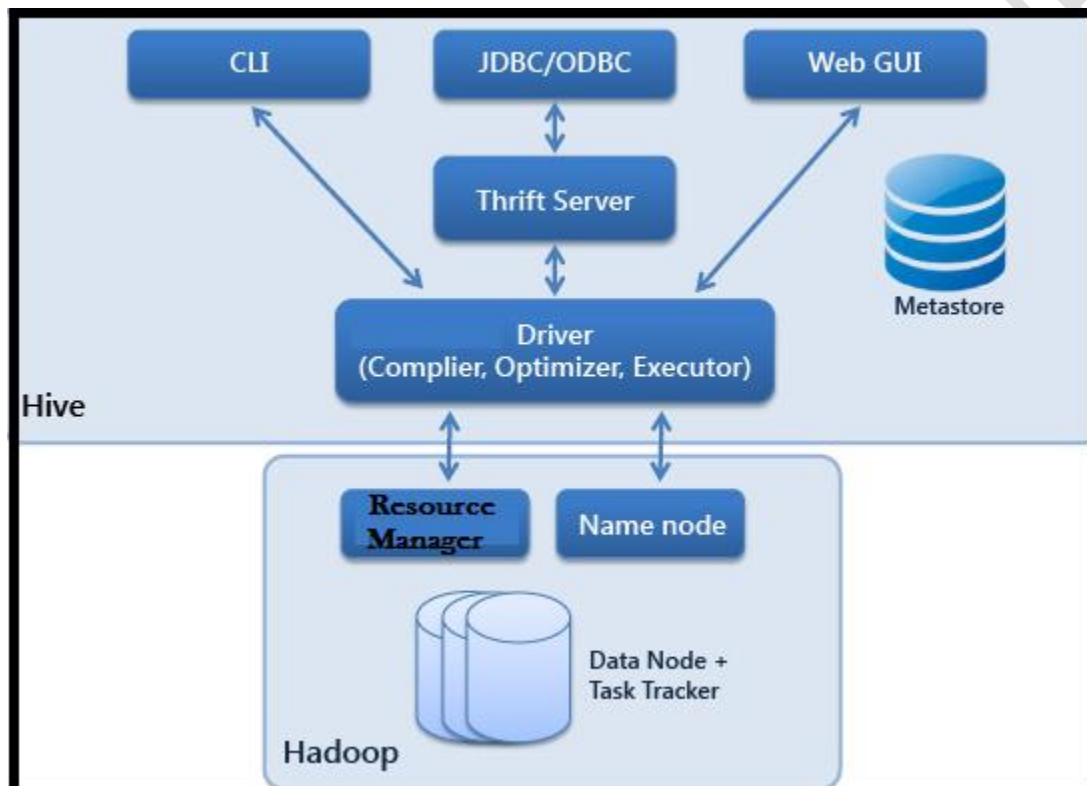
Hive is good for

- Works best for Batch processing and OLAP needs.
- Provides SQL like environment and easily adaptable.
- Capable of storing and processing huge volume of structure and semi structure data with minimal coding effort.
- Uses schema on read instead of schema on write, hence true raw data can be stored.
- It is familiar, fast, scalable, and extensible.

Hive is not good for

- OLTP requirements where frequent changes happen in data.
- Not a relational database, it is a complementary approach to data warehouse on top of Hadoop.
- Not good for real-time and interactive low latency queries.

Architecture Overview



Layers	Operation
User Interface	The user interfaces that Hive supports are Hive Web UI, Hive command line, MS XLS and Squirrel clients.
Meta Store	Hive chooses respective database servers (Derby or MYSQL) to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.

HiveQL Process Engine -Execution Engine	<p>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.</p> <p>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.</p>
Storage HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Hive Execution Engines

MR

The motivation to develop Hive was to provide an abstraction layer on top of Map Reduce (M/R) to make it easier for analysts and data scientists to query data on the Hadoop File System. Rather than write hundreds of lines of Java code to get answers to relatively simple questions the objective was to offer SQL, the natural choice of the data analyst. While this approach works well in a batch oriented environment it does not perform well for interactive workloads in near real time.

The problem with the original M/R framework was that it works in stages and at each stage the data is set down to disk and then again read from disk in the next phase.

Hive on Apache Tez

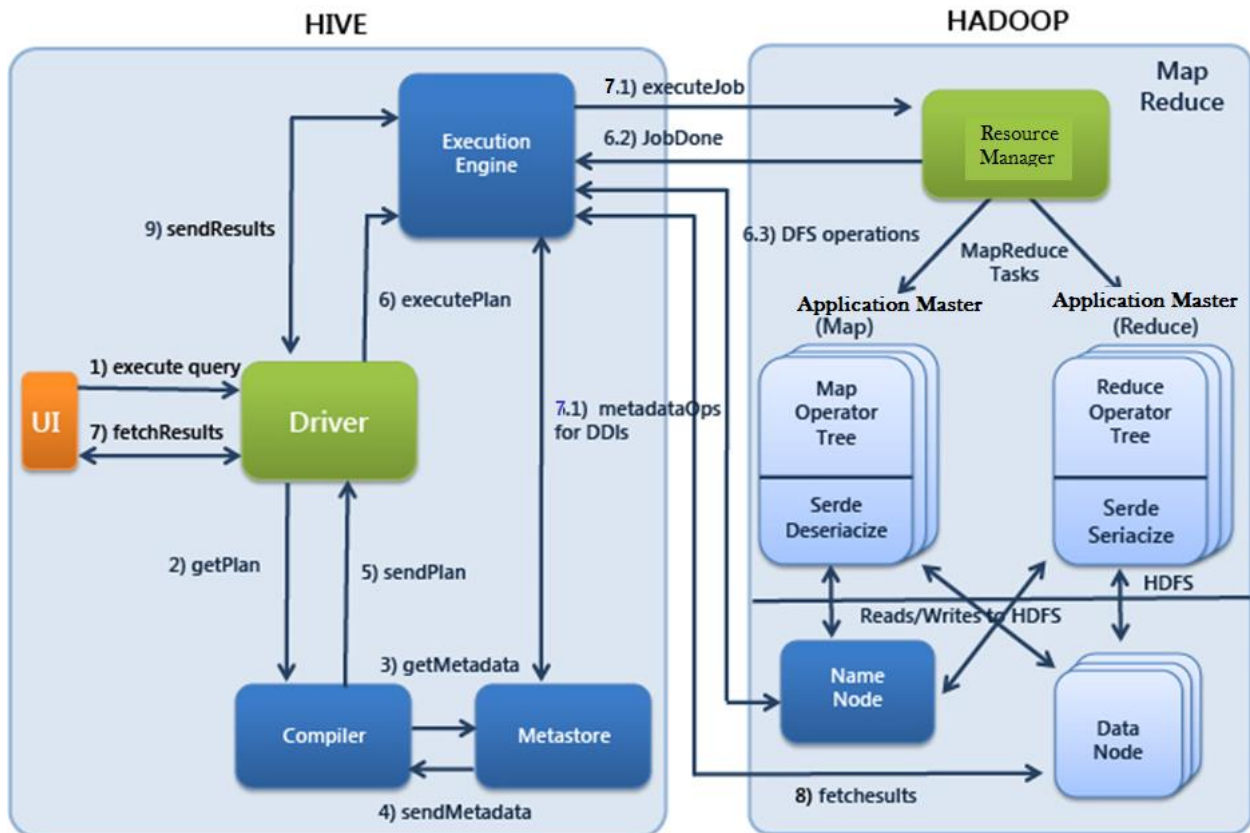
Similar to M/R, Tez is a Hive execution engine developed by Hortonworks (also committers from Facebook, Microsoft, and Yahoo). Tez is part of the Stinger initiative led by Hortonworks to make Hive enterprise ready and suitable for realtime SQL queries. The two main objectives of the initiative were to increase performance and offer a rich set of SQL features such as analytic functions, query optimization, and standard data types such as timestamp etc. Tez is the underlying engine that creates more efficient execution plans in comparison to Map Reduce. The Tez design is based on research done by Microsoft on parallel and distributed computing. The two main objectives were delivered as part of the

recent Hive 0.13 release. The roadmap for release 0.14 includes DML functionality such as Updates and Inserts for lookup tables.

Hive on Spark

Recently, Cloudera together with MapR, Intel, and Databricks spearheaded a new initiative to add a third execution engine to the mix. They propose to add Spark as a third Hive execution engine. Developers then will be able to choose between Map Reduce, Tez, and Spark as their execution engine for Hive. Based on the design document the three engines will be fully interchangeable and compatible. Cloudera see Spark as the next generation distributed processing engine, which has various advantages over the Map Reduce paradigm, e.g. intermediate result sets can be cached in memory. Going forward, Spark will underpin many of the components in the Cloudera platform. The rationale for Hive on Spark then is to make Spark available to the vast amount of Hive users and establish Hive on the Spark framework. It will also allow users to run faster Hive queries without having to install Tez. Contrary to Hortonworks, Cloudera don't see Hive on Spark (or Hive on Tez) to be suitable as a realtime SQL query engine. Their flagship product for interactive SQL queries is Impala, while Databricks see Spark SQL as the tool of choice for realtime queries.

HIVE - HADOOP Integration



Step No.	Operation
1	Execute Query <p>The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.</p>
2	Get Plan <p>The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.</p>
3	Get Metadata <p>The compiler sends metadata request to Metastore (any database).</p>

4	Send Metadata Metastore sends metadata as a response to the compiler.
5	Send Plan The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	Execute Plan The driver sends the execute plan to the execution engine.
7	Execute Job Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	Metadata Ops Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	Fetch Result The execution engine receives the results from Data nodes.
9	Send Results The execution engine sends those resultant values to the driver.

Data Units Hierarchy

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

- **Databases:** Namespaces that separate tables and other data units from naming confliction.
- **Tables:** Homogeneous units of data which have the same schema.
- **Partitions:** is the subdivision of a single table into multiple segments, each of which holds a subset of values. Each Table can have one or more partition Keys which determines how the data is stored. Partitions - apart from being storage units - also allow the user to efficiently identify the rows that satisfy a certain criteria.

- **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. In hive a partition is a directory but a bucket is a file.

set hive.enforce.bucketing=true;

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

Column Types

Column types are used as column data types of Hive. They are as follows:

Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

Type	Size
TINYINT	1 byte int
SMALLINT	2 byte int
INT	4 byte int
BIGINT	8 byte int

String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

Data Type	Length
VARCHAR	1 to 65355
CHAR	255

Timestamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format “YYYY-MM-DD HH:MM:SS.ffffff” and format “yyyy-mm-dd hh:mm:ss.ffffff”.

Dates

DATE values are described in year/month/day format in the form {{YYYY-MM-DD}}.

Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

```
DECIMAL(precision, scale)
decimal(10,0)
```

Union Types

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

```
UNIONTYPE <int, double, array<string>, struct<a:int,b:string>>
{0:1}
{1:2.0}
{2:["three","four"]}
{3:{"a":5,"b":"five"}}
{2:["six","seven"]}
{3:{"a":8,"b":"eight"}}
{0:9}
```

```
{1:10.0}
```

Literals

The following literals are used in Hive:

Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10^{-308} to 10^{308} .

Null Value

Missing values are represented by the special value NULL.

Complex Types

The Hive complex data types are as follows:

Arrays

Arrays are ordered sequence of similar data elements indexed using numeric values.

Syntax: ARRAY ('a','b','c')

the second value can be accessed using ARRAY[1]

Structs

Structs in Hive is similar to using complex data with comment.

Syntax: STRUCT <col_name:data_type , col_name:data_type>

example: STRUCT {a:int, b:char} which can be referred using '.' Notation like column.a

Maps

Maps are collection of key value pairs.

Syntax: MAP<key, value> ,

map('first','a','second','b','last','c'), here map['first'] can retrieve 'a'

Hive Query Language – HQL

Create Database is a statement used to create a database in Hive. A database in Hive is a namespace or a collection of tables.

```
CREATE DATABASE | SCHEMA [IF NOT EXISTS] <database name>
```

Eg:

```
Hive> create database if not exists ibdb;
```

Create Schema – Logical subset of DB objects,

```
Hive> use ibdb;
```

```
Hive> create schema if not exists ibschema;
```

Tables

A Hive table is logically made up of the data being stored and the associated metadata describing the layout of the data in the table. The data typically resides in HDFS, although it may reside in any Hadoop filesystem, including the local filesystem.

Managed Tables and External Tables

When you create a table in Hive, by default Hive will manage the data, which means that Hive moves the data into its warehouse directory. Alternatively, you may create an *external table*, which tells Hive to refer to the data that is at an existing location outside the warehouse directory.

The difference between the two table types is seen in the LOAD and DROP semantics. Let's consider a managed table first.

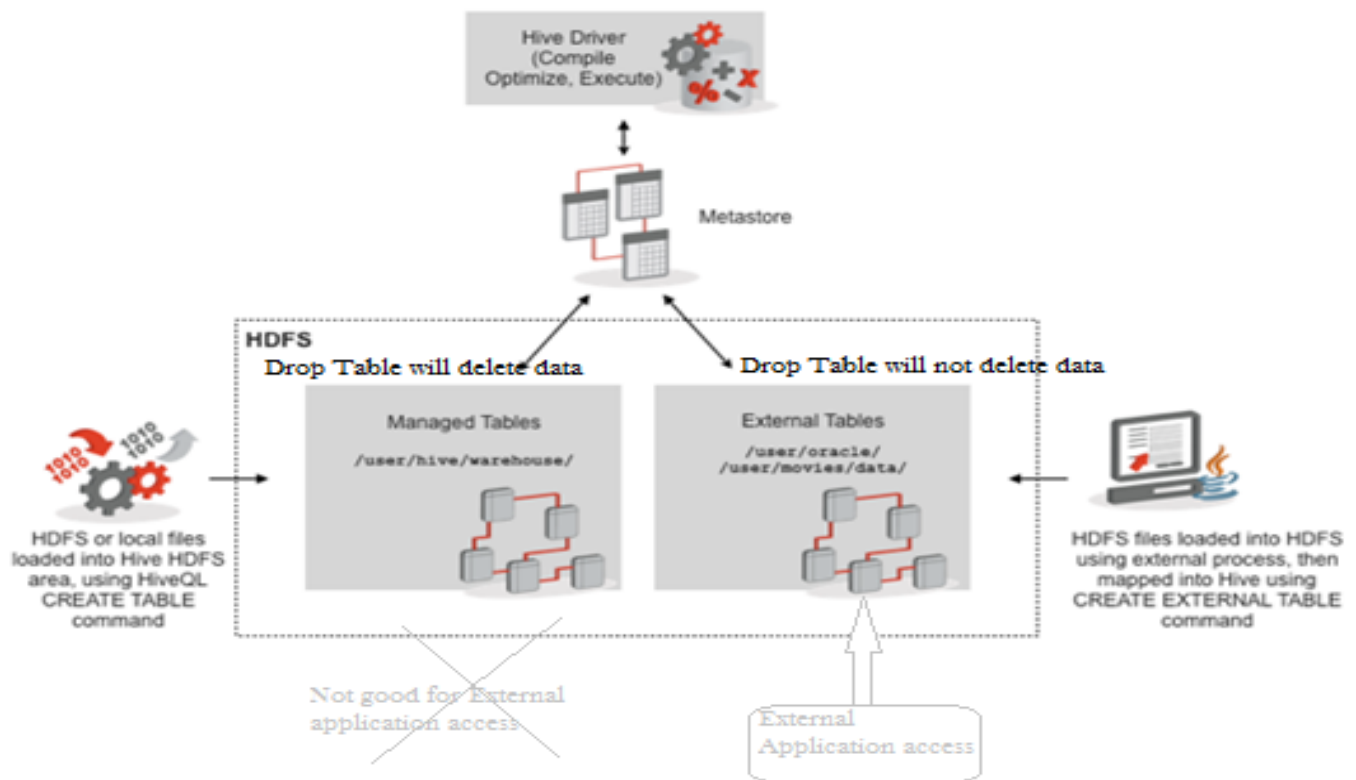
When you load data into a managed table, it is moved into Hive's warehouse directory.

For example:

```
CREATE TABLE managed_table (dummy STRING);
```

```
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

will move the file `hdfs://user/tom/data.txt` into Hive's warehouse directory for the `managed_table` table, which is `hdfs://user/hive/warehouse/managed_table`.



If the table is later dropped, using:

`DROP TABLE managed_table;`

the table, including its metadata and its data, is deleted. It bears repeating that since the initial LOAD performed a move operation, and the DROP performed a delete operation, the data no longer exists anywhere. This is what it means for Hive to manage the data. An external table behaves differently. You control the creation and deletion of the data.

The location of the external data is specified at table creation time:

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
LOCATION '/user/tom/external_table';
```

```
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

With the EXTERNAL keyword, Hive knows that it is not managing the data, so it doesn't move it to its warehouse directory. Indeed, it doesn't even check whether the external location exists at the time it is defined. This is a useful feature because it means you can create the data lazily after creating the table. When you drop an external table, Hive will leave the data untouched and only delete the metadata. So how do you choose which type of table to use? In most cases, there is not much difference between the two (except of course for the difference in DROP semantics), so it is just a matter of preference. As

a rule of thumb, if you are doing all your processing with Hive, then use managed tables, but if you wish to use Hive and other tools on the same dataset, then use external tables. A common pattern is to use an external table to access an initial dataset stored in HDFS (created by another process), then use a Hive transform to move the data into a managed Hive table. This works the other way around, too; an external table (not necessarily on HDFS) can be used to export data from Hive for other applications to use.

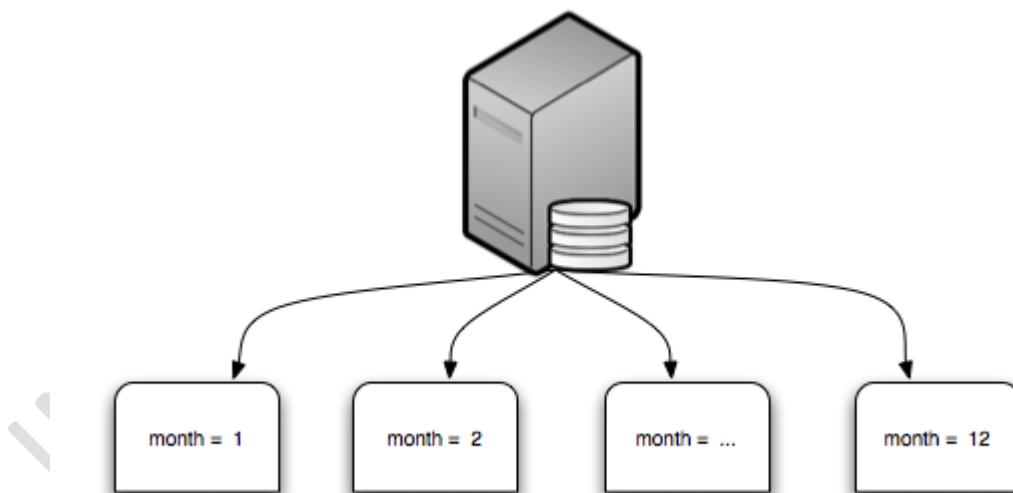
Partitions

Static Partition

Usually when loading files (big files) into Hive tables static partitions are preferred. That saves your time in loading data compared to dynamic partition. You "statically" add a partition in table and move the file into the partition of the table. Since the files are big they are usually generated in HDFS. You can get the partition column value from the filename, day of date etc without reading the whole big file.

Dynamic Partition

In case of dynamic partition whole big file i.e. every row of the data is read and data is partitioned through a MR job into the destination tables depending on certain field in file. So usually dynamic partition are useful when you are doing sort of a ETL flow in your data pipeline. e.g. you load a huge file through a move command into a Table X. then you run a insert query into a Table Y and partition data based on field in table X say day , country. You may want to further run a ETL step to partition the data in country partition in Table Y into a Table Z where data is partitioned based on cities for a particular country only. etc.



To take an example where partitions are commonly used, imagine logfiles where each record includes a timestamp. If we partition by date, then records for the same date will be stored in the same partition. The advantage to this scheme is that queries that are restricted to a particular date or set of dates can run much more efficiently, because they only need to scan the files in the partitions that the query pertains to. Notice that partitioning doesn't preclude more wide-ranging queries: it is still feasible to query the entire dataset across many partitions.

A table may be partitioned in multiple dimensions. For example, in addition to partitioning logs by date, we might also *subpartition* each date partition by country to permit efficient queries by location.

Partitions are defined at table creation time using the PARTITIONED BY clause, which takes a list of column definitions. For the hypothetical logfile example, we might define a table with records comprising a timestamp and the log line itself:

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED BY (dt STRING, country STRING);
```

When we load data into a partitioned table, the partition values are specified explicitly:

```
LOAD DATA LOCAL INPATH 'input/hive/partitions/file1'
INTO TABLE logs
PARTITION (dt='2001-01-01', country='GB');
```

At the filesystem level, partitions are simply nested subdirectories of the table directory. After loading a few more files into the logs table, the directory structure might look like this:

```
/user/hive/warehouse/logs
├── dt=2001-01-01/
│   ├── country=GB/
│   │   ├── file1
│   │   └── file2
│   └── country=US/
│       └── file3
└── dt=2001-01-02/
    ├── country=GB/
    │   └── file4
    └── country=US/
        ├── file5
        └── file6
```

The logs table has two date partitions (2001-01-01 and 2001-01-02, corresponding to subdirectories called *dt=2001-01-01* and *dt=2001-01-02*); and two country subpartitions (GB and US, corresponding to nested subdirectories called *country=GB* and *country=US*).

The datafiles reside in the leaf directories. We can ask Hive for the partitions in a table using SHOW PARTITIONS:

```
hive> SHOW PARTITIONS logs;
dt=2001-01-01/country=GB
dt=2001-01-01/country=US
dt=2001-01-02/country=GB
dt=2001-01-02/country=US
```

One thing to bear in mind is that the column definitions in the `PARTITIONED BY` clause are full-fledged table columns, called partition columns; however, the data files do not contain values for these columns, since they are derived from the directory names.

You can use partition columns in `SELECT` statements in the usual way. Hive performs *input pruning* to scan only the relevant partitions.

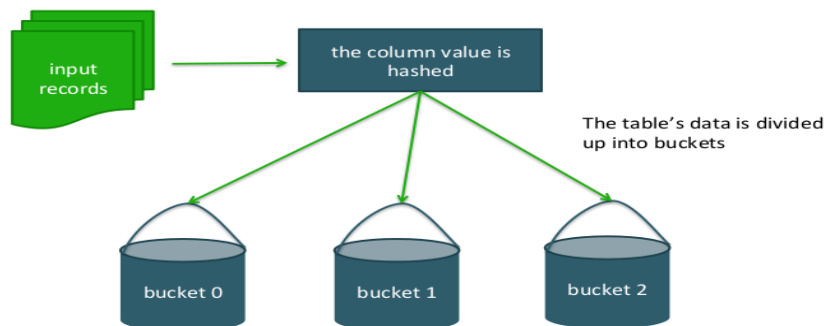
For example:

```
SELECT ts, dt, line FROM logs WHERE country='GB';
```

will only scan *file1*, *file2*, and *file4*. Notice, too, that the query returns the values of the `dt` partition column, which Hive reads from the directory names since they are not in the datafiles.

Buckets

Hive Buckets



There are two reasons why you might want to organize your tables (or partitions) into buckets. The first is to enable more efficient queries. Bucketing imposes extra structure on the table, which Hive can take advantage of when performing certain queries. In particular, a join of two tables that are bucketed on the same columns — which include the join columns — can be efficiently implemented as a map-side join.

The second reason to bucket a table is to make sampling more efficient. When working with large datasets, it is very convenient to try out queries on a fraction of your dataset while you are in the process of developing or refining them. We will see how to do efficient sampling at the end of this section. First, let's see how to tell Hive that a table should be bucketed. We use the `CLUSTERED BY` clause to specify the columns to bucket on and the number of buckets:

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

Here we are using the user ID to determine the bucket (which Hive does by hashing the value and reducing modulo the number of buckets), so any particular bucket will effectively have a random set of users in it. In the map-side join case, where the two tables are bucketed in the same way, a mapper processing a bucket of the left table knows that the matching rows in the right table are in its corresponding bucket, so it need only retrieve that bucket (which is a small fraction of all the data stored

in the right table) to effect the join. This optimization also works when the number of buckets in the two tables are multiples of each other; they do not have to have exactly the same number of buckets.

The data within a bucket may additionally be sorted by one or more columns. This allows even more efficient map-side joins, since the join of each bucket becomes an efficient merge sort. The syntax for declaring that a table has sorted buckets is:

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) SORTED BY (id ASC) INTO 4 BUCKETS;
```

How can we make sure the data in our table is bucketed? Although it's possible to load data generated outside Hive into a bucketed table, it's often easier to get Hive to do the bucketing, usually from an existing table.

WARNING

Hive does not check that the buckets in the datafiles on disk are consistent with the buckets in the table definition (either in number or on the basis of bucketing columns). If there is a mismatch, you may get an error or undefined behavior at query time. For this reason, it is advisable to get Hive to perform the bucketing.

Take an unbucketed users table:

```
hive> SELECT * FROM users;
```

```
0 Nat
2 Joe
3 Kay
4 Ann
```

To populate the bucketed table, we need to set the `hive.enforce.bucketing` property to true so that Hive knows to create the number of buckets declared in the table definition. Then it is just a matter of using the INSERT command:

```
INSERT OVERWRITE TABLE bucketed_users
SELECT * FROM users;
```

Physically, each bucket is just a file in the table (or partition) directory. The filename is not important, but bucket n is the nth file when arranged in lexicographic order. In fact, buckets correspond to MapReduce output file partitions: a job will produce as many buckets (output files) as reduce tasks. We can see this by looking at the layout of the `bucketed_users` table we just created. Running this command:

```
hive> dfs -ls /user/hive/warehouse/bucketed_users;
```

shows that four files were created, with the following names (the names are generated by Hive):

```
000000_0
000001_0
000002_0
```


000003_0

The first bucket contains the users with IDs 0 and 4, since for an INT the hash is the integer itself, and the value is reduced modulo the number of buckets — four, in this case:

```
hive> dfs -cat /user/hive/warehouse/bucketed_users/000000_0;
0Nat
4Ann
```

We can see the same thing by sampling the table using the TABLESAMPLE clause, which restricts the query to a fraction of the buckets in the table rather than the whole table:

```
hive> SELECT * FROM bucketed_users TABLESAMPLE (BUCKET 1 OUT OF 4 ON id);

4 Ann
0 Nat
```

Bucket numbering is 1-based, so this query retrieves all the users from the first of four buckets. For a large, evenly distributed dataset, approximately one-quarter of the table's rows would be returned. It's possible to sample a number of buckets by specifying a different proportion (which need not be an exact multiple of the number of buckets, as sampling is not intended to be a precise operation). For example, this query returns half of the buckets:

```
hive> SELECT * FROM bucketed_users TABLESAMPLE(BUCKET 1 OUT OF 2 ON id);
4 Ann
0 Nat
2 Joe
```

Sampling a bucketed table is very efficient because the query only has to read the buckets that match the TABLESAMPLE clause. Contrast this with sampling a nonbucketed table using the rand() function, where the whole input dataset is scanned, even if only a very small sample is needed:

```
hive> SELECT * FROM users TABLESAMPLE(BUCKET 1 OUT OF 4 ON rand());
2 Joe
```

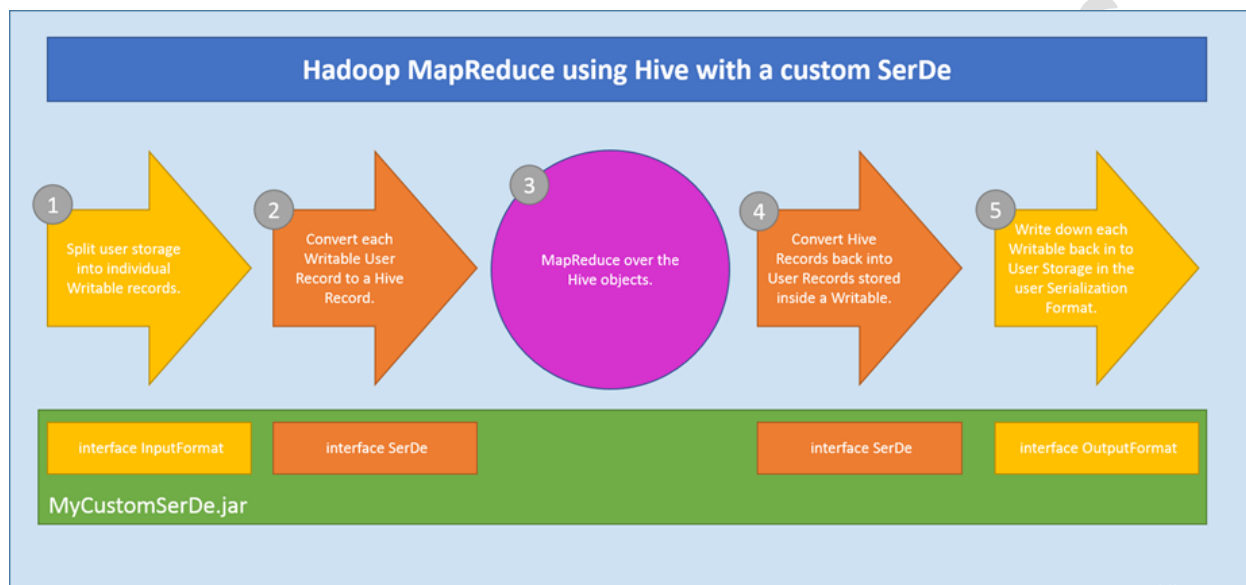
Serialization and De Serialization (SERDE)

The SerDe interface is extremely powerful for dealing with data with a complex schema. By utilizing SerDes, any dataset can be made queryable through Hive.

The SerDe interface allows you to instruct Hive as to how a record should be processed. A SerDe is a combination of a Serializer and a Deserializer (hence, Ser-De). The Deserializer interface takes a string or binary representation of a record, and translates it into a Java object that Hive can manipulate. The Serializer, however, will take a Java object that Hive has been working with, and turn it into something that Hive can write to HDFS or another supported system. Commonly, Deserializers are used

at query time to execute SELECT statements, and Serializers are used when writing data, such as through an INSERT-SELECT statement. Example of serde data for Hive are XML, JSON file etc.

- Hive uses SerDe (and FileFormat) to read and write table rows.
- HDFS files --> InputFileFormat --> <key, value> --> Deserializer --> Row object
- Row object --> Serializer --> <key, value> --> OutputFileFormat --> HDFS files



Comparison of Different File Formats

➤ Predicate Pushdown

- `hive.optimize.ppd=true`
- `hive.enforce.sorting`

First 10,000 Rows

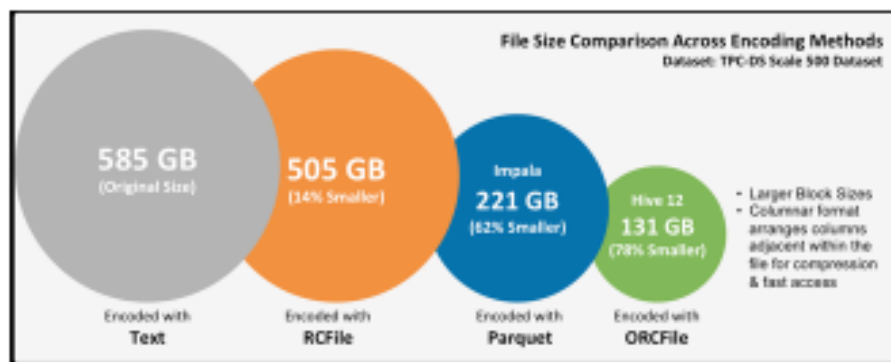
ID (min = 1, max=10000)	Name (dictionary, min, max)	State (dictionary, min, max)
1	Bob	NJ
2	Larry	CA
3	Sue	TX

Stride Index

➤ ORC Format

Second 10,000 Rows

ID (min = 10001, max=20000)	Name (dictionary, min, max)	State (dictionary, min, max)
10001	Steve	OR
10002	Alan	ND
10003	Mary	FL



INCEPTEZ TECHNOLOGIES

➤ Performance Tuning

- Parallel exec (hive.exec.parallel=true)

- Vectorization

```
set hive.vectorized.execution.enabled = true;  
set hive.vectorized.execution.reduce.enabled = true;
```

- CBO

```
set hive.cbo.enable=true;  
set hive.compute.query.using.stats=true;  
set hive.stats.fetch.column.stats=true;  
set hive.stats.fetch.partition.stats=true;  
analyze table tweets compute statistics;  
analyze table tweets compute statistics for columns sender, topic;
```

- Compression (set mapred.output.compress=true)
- Index, Partition and Bucket sampling
- Optimized joins (/* streamtable(table_name) */)



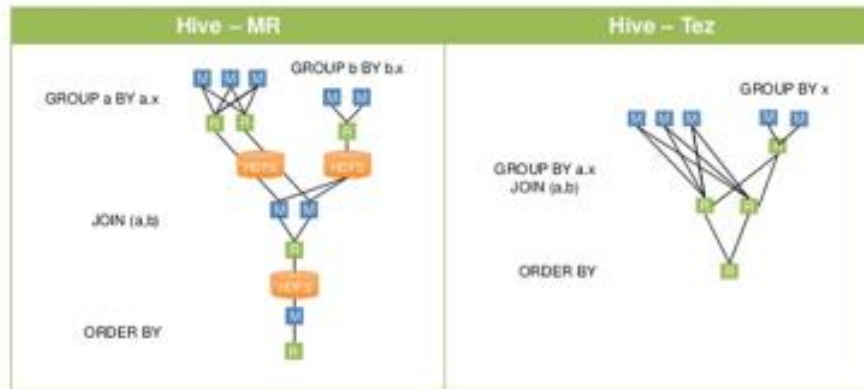
➤ TEZ Engine

- DAG
- Container Reusability
- Dynamic Graphs

Hive-on-MR vs. Hive-on-Tez










```
SELECT g1.x, g1.avg, g2.cnt  
FROM (SELECT a.x, AVERAGE(a.y) AS avg FROM a GROUP BY a.x) g1  
JOIN (SELECT b.x, COUNT(b.y) AS cnt FROM b GROUP BY b.x) g2  
ON (g1.x = g2.x)  
ORDER BY avg;
```

Tez avoids
unnecessary writes
to HDFS



INCEPTEZ TECHNOLOGIES

BIG DATA FORMATS COMPARISON

	Avro	Parquet	ORC
Schema Evolution Support			
Compression			
Splitability			
Most Compatible Platforms	Kafka, Druid	Impala, Arrow Drill, Spark	Hive, Presto
Row or Column	Row	Column	Column
Read or Write	Write	Read	Read

Source: Nexla analysis, April 2018

	AVRO	Parquet	ORC
Schema Evolution	Most efficient, as the schema is stores as JSON with the file.	Schema evolution is expensive as it needs to be read and merged across all the parquet files	Schema evolution here is limited to adding new columns and a few cases of column type-widening.
Compression	Less efficient	Best with Snappy	Best with ZLIB
Splitability	Partially	Partially	Fully
Row/Column Oriented	Row	Column	Column
Read/Write	Write-Heavy	Read-Heavy	Read-Heavy
Optimized Processing Engines	Kafka	Spark(Cloudera)	Hive, Presto (Horton Works)

Parquet :

1. Column oriented storage format.
2. Schema is store in the footer of the file.
3. Due to merging of schema across multiple files, schema evolution is little expensive.
4. Ideal for read heavy data operations.
5. Excellent for selected column data consumption and processing.
6. Works excellent with spark as there is vectorized reader for parquet.

Avro :

1. Row oriented storage format.
2. Schema is store as JSON within file.
3. It is also a Serialization and RPC framework.
4. Excellent for schema evolution.
5. Ideal for write heavy data operations.
6. Excellent for entire row consumption and processing.

ORC :

1. Column oriented storage format.
2. Schema is store in the footer of the file.
3. Schema evolution is limited to adding new columns.
4. Ideal for read heavy data operations.
5. Excellent for selected column data consumption and processing.
6. Works excellent with Hive as there is vectorized reader for parquet.

Registration of Native SerDes

As of Hive 0.14 a registration mechanism has been introduced for native Hive SerDes. This allows dynamic binding between a "STORED AS" keyword in place of a triplet of {SerDe, InputFormat, and OutputFormat} specification, in CreateTable statements.

The following mappings have been added through this registration mechanism:

Syntax	Equivalent
--------	------------

Syntax	Equivalent
STORED AS AVRO / STORED AS AVROFILE	ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe' STORED AS INPUTFORMAT 'org.apache.hadoop.hive ql.io.avro.AvroContainerInputFormat' OUTPUTFORMAT 'org.apache.hadoop.hive ql.io.avro.AvroContainerOutputFormat'
STORED AS ORC / STORED AS ORCFILE	ROW FORMAT SERDE 'org.apache.hadoop.hive ql.io.orc.OrcSerde' STORED AS INPUTFORMAT 'org.apache.hadoop.hive ql.io.orc.OrcInputFormat' OUTPUTFORMAT 'org.apache.hadoop.hive ql.io.orc.OrcOutputFormat'
STORED AS PARQUET / STORED AS PARQUETFILE	ROW FORMAT SERDE 'org.apache.hadoop.hive ql.io.parquet.serde.ParquetHiveSerDe' STORED AS INPUTFORMAT 'org.apache.hadoop.hive ql.io.parquet.MapredParquetInputFormat' OUTPUTFORMAT 'org.apache.hadoop.hive ql.io.parquet.MapredParquetOutputFormat'
STORED AS TEXTFILE	STORED AS INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat' OUTPUTFORMAT 'org.apache.hadoop.hive ql.io.IgnoreKeyTextOutputFormat'

Performance Tuning

HiveQL is a declarative language where users issue declarative queries and Hive figures out how to translate them into MapReduce jobs. Most of the time, you don't need to understand how Hive works, freeing you to focus on the problem at hand. While the sophisticated process of query parsing, planning, optimization, and execution is the result of many years of hard engineering work by the Hive team, most of the time you can remain oblivious to it. However, as you become more experienced with Hive, learning about the theory behind Hive, and the low-level implementation details, will let you use Hive more effectively, especially where performance optimizations are concerned.

Sort by, Order by, Cluster by, Distribute by

In Apache Hive, It's always a matter of confusion over how SORT BY, ORDER BY, DISTRIBUTE BY and CLUSTER BY differs. I have compiled a set of differences between these based on attributes like how will final output look like and ordering of data in output –

SORT BY

Hive uses the columns in SORT BY to sort the rows before feeding the rows to a reducer. The sort order will be dependent on the column types. If the column is of numeric type, then the sort order is also in numeric order. If the column is of string type, then the sort order will be lexicographical order.

Ordering : It orders data at each of 'N' reducers , but each reducer can have overlapping ranges of data.

Outcome : N or more sorted files with overlapping ranges.

Let's understand with an example :-

```
SELECT key, value FROM src SORT BY key ASC, value DESC
```

The query had 2 reducers, and the output of each is:

Reducer 1 :

0 5

0 3

3 6

9 1

Reducer 2 :

0 4

0 3

1 1

2 5

As, we can see, each reducer output is ordered but total ordering is missing, since we end up with multiple outputs per reducer.

ORDER BY

This is similar to ORDER BY in SQL Language.

In Hive, ORDER BY guarantees total ordering of data, but for that it has to be passed on to a single reducer, which is normally unacceptable and therefore in strict mode, hive makes it compulsory to use LIMIT with ORDER BY so that reducer doesn't get overburdened.

Ordering : Total Ordered data.

Outcome : Single output i.e. fully ordered.

For example :

```
SELECT key, value FROM src ORDER BY key ASC, value DESC
```

1

```
SELECT key, value FROM src ORDER BY key ASC, value DESC
```

Reducer :

0 5

0 4

0 3

0 3

1 1

2 5

3 6

9 1

DISTRIBUTE BY

Hive uses the columns in Distribute By to distribute the rows among reducers. All rows with the same Distribute By columns will go to the same reducer.

It ensures each of N reducers gets non-overlapping ranges of column, but doesn't sort the output of each reducer. You end up with N or more unsorted files with non-overlapping ranges.

Example:-

We are Distributing By x on the following 5 rows to 2 reducer:

x1

x2

x4

x3

x1

Reducer 1 got

x1

x2

x1

Reducer 2 got

x4

x3

Note that all rows with the same key x1 is guaranteed to be distributed to the same reducer (reducer 1 in this case), but they are not guaranteed to be clustered in adjacent positions.

CLUSTER BY

Cluster By is a short-cut for both Distribute By and Sort By.

CLUSTER BY x ensures each of N reducers gets non-overlapping ranges, then sorts by those ranges at the reducers.

Ordering : Global ordering between multiple reducers.

Outcome : N or more sorted files with non-overlapping ranges.

For the same example as above , if we use Cluster By x, the two reducers will further sort rows on x:

Reducer 1 got

x1

x1

x2

Reducer 2 got

x3

x4

Instead of specifying Cluster By, the user can specify Distribute By and Sort By, so the partition columns and sort columns can be different.

Using EXPLAIN

The first step to learning how Hive works (after reading this book...) is to use the EXPLAIN feature to learn how Hive translates queries into MapReduce jobs.

Consider the following exam'ple:

```
hive> DESCRIBE onecol;
```

```
number int
```

```
hive> SELECT * FROM onecol;
```

```
5
```

```
5
```

4

```
hive> SELECT SUM(number) FROM onecol;
```

14

Now, put the EXPLAIN keyword in front of the last query to see the query plan and other information. The query will not be executed.

```
hive> EXPLAIN SELECT SUM(number) FROM onecol;
```

The output requires some explaining and practice to understand. First, the *abstract syntax tree* is printed. This shows how Hive parsed the query into tokens and literals, as part of the first step in turning the query into the ultimate result:

ABSTRACT SYNTAX TREE:

```
(TOK_QUERY
(TOK_FROM (TOK_TABREF (TOK_TABNAME onecol)))
(TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))
(TOK_SELECT
(TOK_SELEXPR
(TOK_FUNCTION sum (TOK_TABLE_OR_COL number))))))
```

(The indentation of the actual output was changed to fit the page.) For those not familiar with parsers and tokenizers, this can look overwhelming. However, even if you are a novice in this area, you can study the output to get a sense for what Hive is doing with the SQL statement. (As a first step, ignore the TOK_ prefixes.) Even though our query will write its output to the console, Hive will actually write the output to a temporary file first, as shown by this part of the output:

```
'(TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))'
```

EXPLAIN EXTENDED

Using EXPLAIN EXTENDED produces even more output. In an effort to “go green,” we won’t show the entire output, but we will show you the Reduce Operator Tree to demonstrate the different output:

Reduce **Operator** Tree:

Group By Operator

aggregations:

expr: **sum**(VALUE._col0)

bucketGroup: **false**

mode: mergepartial

outputColumnNames: _col0

Select Operator

expressions:

expr: _col0

type: bigint

outputColumnNames: _col0

File Output Operator

compressed: **false**

GlobalTableId: 0

directory: file:/tmp/edward/hive_2012-[long number]/-ext-10001

NumFilesPerFileSink: 1

Stats Publishing **Key Prefix**:

file:/tmp/edward/hive_2012-[long number]/-ext-10001/

table:

input format: org.apache.hadoop.mapred.TextInputFormat

output format:

org.apache.hadoop.hive ql.io.HiveIgnoreKeyTextOutputFormat

properties:

columns_col0

columns.types bigint

escape.delim \

serialization.format 1

TotalFiles: 1

GatherStats: **false**

MultiFileSpray: **false**

Optimized Joins

use the `/* streamtable(table_name) */` directive.

If all but one table is small enough, typically to fit in memory, then Hive can perform a *map-side join*, eliminating the need for reduce tasks and even some map tasks. Sometimes even tables that do not fit in memory are good candidates because removing the reduce phase outweighs the cost of bringing semi-large tables into each map tasks.

Local Mode

Many Hadoop jobs need the full scalability benefits of Hadoop to process large data sets. However, there are times when the input to Hive is very small. In these cases, the overhead of launching tasks for queries consumes a significant percentage of the overall job execution time. In many of these cases, Hive can leverage the lighter weight of the *local mode* to perform all the tasks for the job on a single machine and sometimes in the same process. The reduction in execution times can be dramatic for small data sets.

You can explicitly enable local mode temporarily, as in this example:

```
hive> set mapred.job.tracker=local;
```

```
hive> SELECT * from people WHERE firstname=bob;
```

You can also tell Hive to automatically apply this optimization by setting

hive.exec.mode.local.auto to true

To set this property permanently for all users, change the value in your `$HIVE_HOME/conf/hive-site.xml`:

```
<property>
```

```
<name>hive.exec.mode.local.auto</name>
```

```
<value>true</value>
```

```
<description>
```

```
Let hive determine whether to run in local mode automatically
```

```
</description>
```

```
</property>
```

Parallel Execution

Hive converts a query into one or more stages. Stages could be a MapReduce stage, a sampling stage, a merge stage, a limit stage, or other possible tasks Hive needs to do. By default, Hive executes these stages one at a time. However, a particular job may consist of some stages that are not dependent on each other and could be executed in parallel, possibly allowing the overall job to complete more quickly. However, if more stages are run simultaneously, the job may complete much faster.

Setting `hive.exec.parallel` to true enables parallel execution. Be careful in a shared cluster, however. If a job is running more stages in parallel, it will increase its cluster utilization:

```
<property>
<name>hive.exec.parallel</name>
<value>true</value>
<description>Whether to execute jobs in parallel</description>
</property>
```

Strict Mode

Strict mode is a setting in Hive that prevents users from issuing queries that could have unintended and undesirable effects.

Setting the property `hive.mapred.mode` to `strict` disables three types of queries.

First, queries on partitioned tables are not permitted unless they include a *partition filter* in the WHERE clause, limiting their scope. In other words, you're prevented from queries that will scan all partitions.

The rationale for this limitation is that partitioned tables often hold very large data sets that may be growing rapidly. An unrestricted partition could consume unacceptably large resources over such a large table:

```
hive> SELECT DISTINCT(planner_id) FROM fracture_ins WHERE planner_id=5;
FAILED: Error in semantic analysis: No Partition Predicate Found for
```

The following enhancement adds a partition filter—the table partitions—to the WHERE clause:

```
hive> SELECT DISTINCT(planner_id) FROM fracture_ins
> WHERE planner_id=5 AND hit_date=20120101;
... normal results ...
```

The second type of restricted query are those with ORDER BY clauses, but no LIMIT clause. Because ORDER BY sends all results to a single reducer to perform the ordering, forcing the user to specify a LIMIT clause prevents the reducer from executing for an extended period of time:

```
hive> SELECT * FROM fracture_ins WHERE hit_date>2012 ORDER BY planner_id;
FAILED: Error in semantic analysis: line 1:56 In strict mode,
limit must be specified if ORDER BY is present planner_id
```

To issue this query, add a LIMIT clause:

```
hive> SELECT * FROM fracture_ins WHERE hit_date>2012 ORDER BY planner_id
> LIMIT 100000;
... normal results ...
```

The third and final type of query prevented is a *Cartesian product*. Users coming from the relational database world may expect that queries that perform a JOIN not with an ON clause but with a WHERE clause will have the query optimized by the query planner, effectively converting the WHERE clause into an ON clause. Unfortunately, Hive does not perform this optimization, so a runaway query will occur if the tables are large:

```
hive> SELECT * FROM fracture_act JOIN fracture_ads
> WHERE fracture_act.planner_id = fracture_ads.planner_id;
FAILED: Error in semantic analysis: In strict mode, cartesian product
```

is **not** allowed. If you really want to perform the **operation**,

+set hive.mapred.**mode**=nonstrict+

Here is a properly constructed query with JOIN and ON clauses:

```
hive> SELECT * FROM fracture_act JOIN fracture_ads  
> ON (fracture_act.planner_id = fracture_ads.planner_id);  
... normal results ...
```

Tuning the Number of Mappers and Reducers

Hive is able to parallelize queries by breaking the query into one or more MapReduce jobs. Each of which might have multiple mapper and reducer tasks, at least some of which can run in parallel. Determining the optimal number of mappers and reducers depends on many variables, such as the size of the input and the operation being performed on the data.

A balance is required. Having too many mapper or reducer tasks causes excessive overhead in starting, scheduling, and running the job, while too few tasks means the inherent parallelism of the cluster is underutilized.

When running a Hive query that has a reduce phase, the CLI prints information about how the number of reducers can be tuned. Let's see an example that uses a GROUP BY query, because they always require a reduce phase. In contrast, many other queries are converted into map-only jobs:

```
hive> SELECT pixel_id, count FROM fracture_ins WHERE hit_date=20120119  
> GROUP BY pixel_id;  
Total MapReduce jobs = 1  
Launching Job 1 out of 1  
Number of reduce tasks not specified. Estimated from input data size: 3
```

In order to change the average **load for** a reducer (in bytes):

set hive.exec.reducers.bytes.per.reducer=<number>

In order to limit the maximum number of reducers:

set hive.exec.reducers.max=<number>

In order to set a constant number of reducers:

set mapred.reduce.tasks=<number>

Hive is determining the number of reducers from the input size. This can be confirmed using the dfs -count command, which works something like the Linux du -s command; it computes a total size for all the data under a given directory:

```
$ hadoop dfs -count /user/media6/fracture/ins/* | tail -4  
1 8 2614608737 hdfs://.../user/media6/fracture/ins/hit_date=20120118  
1 7 2742992546 hdfs://.../user/media6/fracture/ins/hit_date=20120119  
1 17 2656878252 hdfs://.../user/media6/fracture/ins/hit_date=20120120  
1 2 362657644 hdfs://.../user/media6/fracture/ins/hit_date=20120121
```

(We've reformatted the output and elided some details for space.)

The default value of `hive.exec.reducers.bytes.per.reducer` is 1 GB. Changing this value to 750 MB causes Hive to estimate four reducers for this job:

```
hive> set hive.exec.reducers.bytes.per.reducer=750000000;
hive> SELECT pixel_id,count(1) FROM fracture_ins WHERE hit_date=20120119
> GROUP BY pixel_id;
```

Total MapReduce jobs = 1

Launching Job 1 out of 1

Number of reduce tasks not specified. Estimated from input data size: 4

This default typically yields good results. However, there are cases where a query's map phase will create significantly more data than the input size. In the case of excessive map phase data, the input size of the default might be selecting too few reducers. Likewise the map function might filter a large portion of the data from the data set and then fewer reducers may be justified.

A quick way to experiment is by setting the number of reducers to a fixed size, rather than allowing Hive to calculate the value. If you remember, the Hive default estimate is three reducers. Set `mapred.reduce.tasks` to different numbers and determine if more or fewer reducers results in faster run times. Remember that benchmarking like this is complicated by external factors such as other users running jobs simultaneously. Hadoop has a few seconds overhead to start up and schedule map and reduce tasks. When executing performance tests, it's important to keep these factors in mind, especially if the jobs are small.

The `hive.exec.reducers.max` property is useful for controlling resource utilization on shared clusters when dealing with large jobs. A Hadoop cluster has a fixed number of map and reduce "slots" to allocate to tasks. One large job could reserve all of the slots and block other jobs from starting. Setting `hive.exec.reducers.max` can stop a query from taking too many reducer resources. It is a good idea to set this value in your

`$HIVE_HOME/conf/hive-site.xml`. A suggested formula is to set the value to the result of this calculation:

$$(\text{Total Cluster Reduce Slots} * 1.5) / (\text{avg number of queries running})$$

The 1.5 multiplier is a fudge factor to prevent underutilization of the cluster.