CIT650 Introduction to Big Data

HIVE Data Management

SQL on Hadoop

- Abstractions over MapReduce
  - Hive
  - Impala
  - Drill
  - . . .
  - Spark SQL

Limitations of MapReduce

- Recall from MapReduce lecture?
- Usability
    - You would need to be a Java developer
    - You need to think MapReduce
    - You have to build several jobs
- Example?
    - Get total sales by department
        - You have your data in sales file (a huge file) and department file (relatively very small)

Joining data in different HDFS files

Input

ID	Name	No Employees
1	Life Insurance	20
2	Vehicle insurance	15
3	Real estate insurance	30

Dept ID	Date	EmpID	Amount
1	25-01-2017	102	50K
2	16-02-2017	715	100K
2	28-02-2017	80	20K
…	…	…	..

Joining data in different HDFS files

Input

ID	Name	No Employees
1	Life Insurance	20
2	Vehicle insurance	15
3	Real estate insurance	30

Dept ID	Date	EmpID	Amount
1	25-01-2017	102	50K
2	16-02-2017	715	100K
2	28-02-2017	80	20K
…	…	…	..

Output

ID	Name	Month-Year	Total
1	Life Insurance	01-2017	50K
2	Vehicle insurance	02-2017	120K
3	Real estate insurance	…	…

Join in Map Reduce

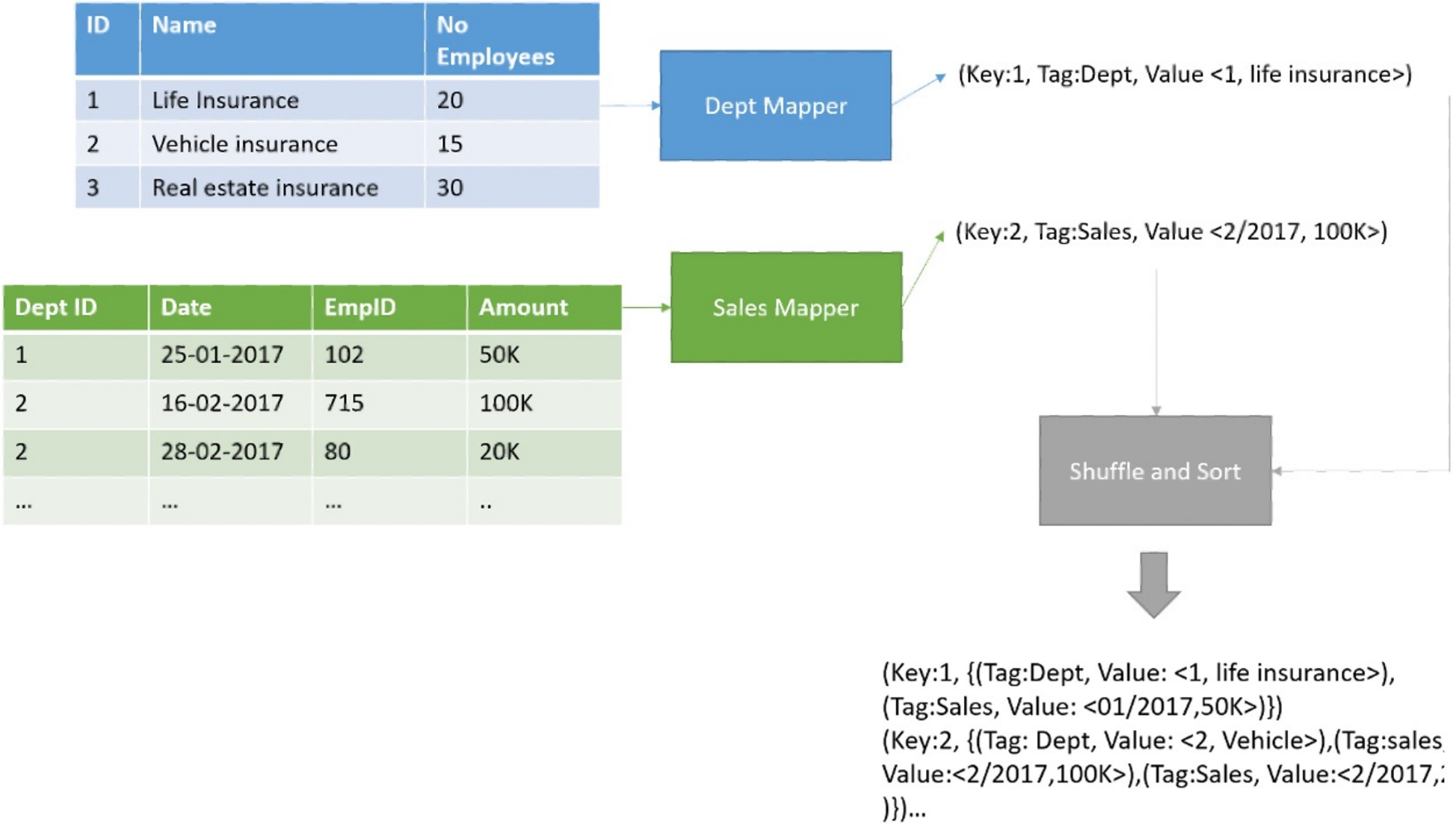
ID	Name	No Employees
1	Life Insurance	20
2	Vehicle insurance	15
3	Real estate insurance	30

Dept Mapper

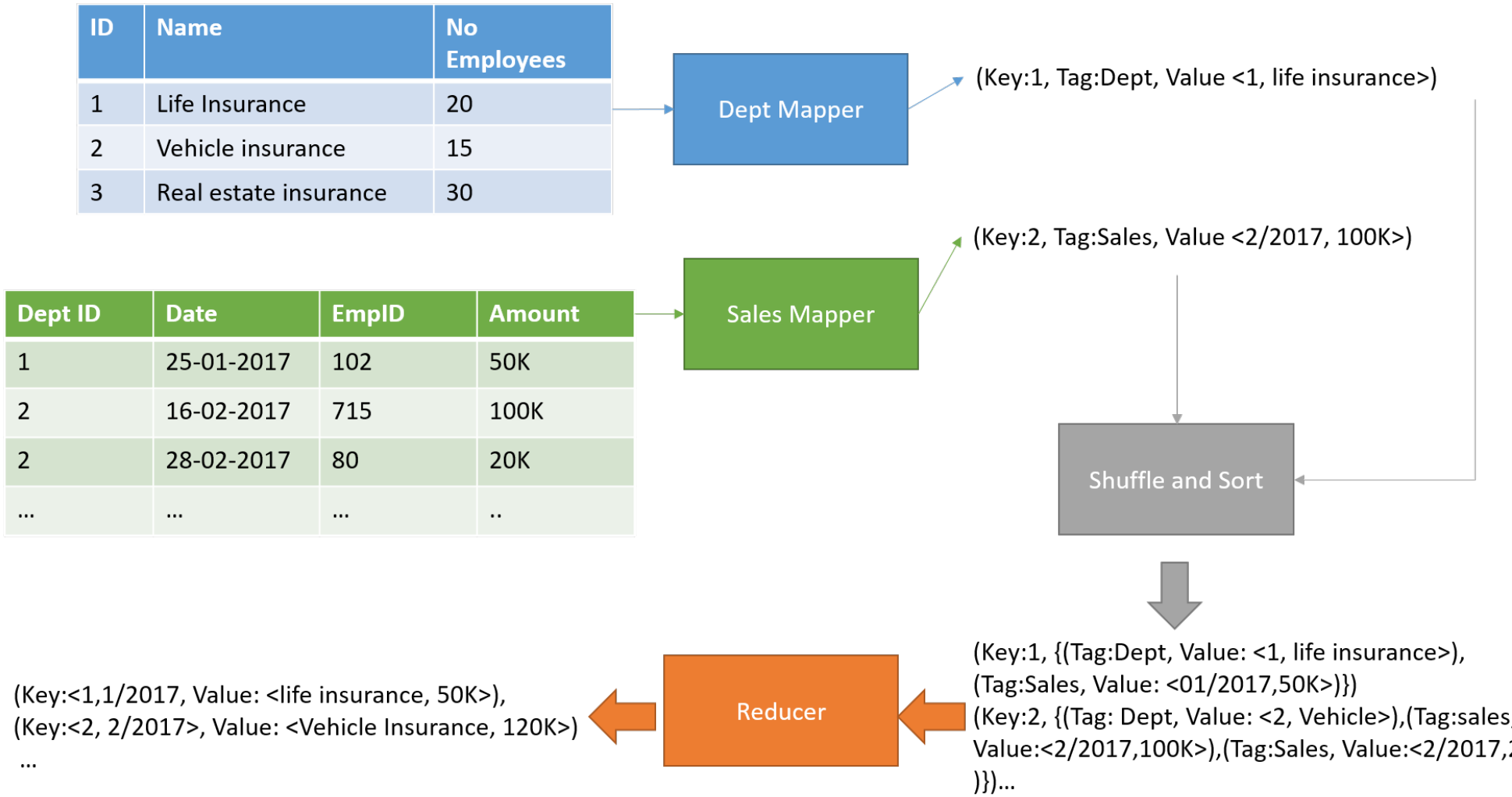
Dept ID	Date	EmpID	Amount
1	25-01-2017	102	50K
2	16-02-2017	715	100K
2	28-02-2017	80	20K
…	…	…	..

Sales Mapper

Join in Map Reduce



Join in Map Reduce



What can be done to improve?

- Hide the complexity of MapReduce jobs
  - Cannot we write SQL statement like

```
SELECT Dept.Id, Dept.Name, MonthYear(Sales.Date) AS
Month_Year,
SUM (Sales.Amount) AS TotalSales
FROM Dept
JOIN Sales ON Dept.Id = Sales.DeptId
GROUP BY Dept.Id, Dept.Name, MonthYear(Sales.Date);
```

  - that maps automatically to MapReduce job(s)?
  - Apache Pig was one step on the way. Yet, still procedural. Apache Hive was the first SQL-like abstraction

What is HIVE?

A system for managing and querying structured data built on top of Hadoop

- Uses Map-Reduce for execution
- HDFS for storage
- Extensible to other Data Repositories

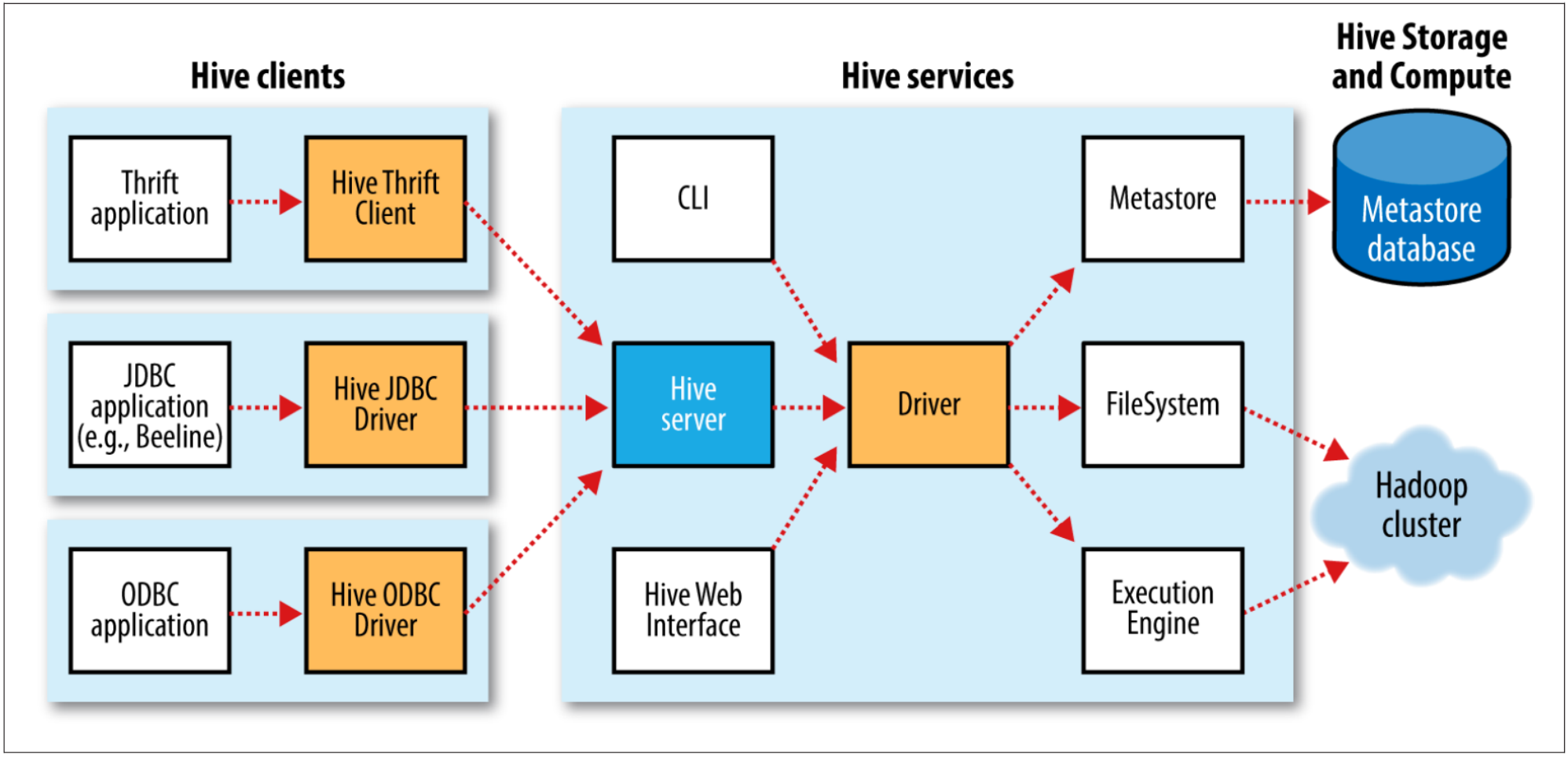
Key Building Principles:

- SQL on structured data as a familiar data warehousing tool
- Extensibility (Pluggable map/reduce scripts in the language of your choice, Rich and User Defined data types, User Defined Functions)
- Interoperability (Extensible framework to support different file and data formats)

What HIVE Is Not

- Hive doesn’t support OLTP. Hive supports Online Analytical Processing (OLAP), but not Online Transaction Processing (OLTP).
- It doesn’t support subqueries.
- It has a high latency.
- Hive tables don’t support delete or update operations

Hive Architecture



Hive Architecture

- **Hive Clients:** Interface applications for interacting with Hive, including Thrift, JDBC, and ODBC applications. They communicate with Hive services to execute queries.
- **Hive Thrift Client:** A client that enables remote procedure calls (RPC) from languages not supported natively by Hive.
- **Hive JDBC Driver:** A Java Database Connectivity (JDBC) driver that enables Java applications (like Beeline) to connect to Hive.
- **Hive ODBC Driver:** An Open Database Connectivity (ODBC) driver that allows ODBC-compliant applications to connect to Hive.

Hive Architecture

- **CLI:** The command-line interface for Hive that allows users to execute HiveQL commands directly.
- **Hive Web Interface:** A web-based UI for interacting with Hive.
- **Hive Server:** A service that provides a way for external clients to interact with Hive via Thrift, JDBC, and ODBC.
- **Driver:** The component that receives HiveQL statements, plans the execution, and sends tasks to the execution engine.

Hive Architecture

- **Metastore:** A service that stores metadata for Hive tables and partitions in a relational database.
- **Metastore Database:** The relational database where the metastore service stores its metadata.
- **FileSystem:** Represents the Hadoop Distributed File System (HDFS) where Hive data is stored.
- **Execution Engine:** The component that executes the tasks in the Hadoop cluster as per the execution plan.
- **Hadoop Cluster:** A collection of nodes managed by Hadoop for distributed storage and processing of large data sets.

Hive vs. Relational Databases

Feature	Hive	Traditional RDBMS
Function	Data warehouse for big data analysis	Structured data storage and management
Data Storage	Hadoop Distributed File System (HDFS)	Local file systems/SANs
Processing	MapReduce, Tez, or Spark	SQL processing engines
Query Language	HiveQL (translates to MapReduce jobs)	SQL
Performance	Optimized for large data sets (batch processing)	Optimized for OLTP, quick response times
Schema	Applied on read (schema-on-read)	Defined on write (schema-on-write)
Transactions	Limited support	Full ACID support
Use Cases	Large-scale data warehousing	OLTP, CRM, ERP systems
Scalability	Horizontally scalable (petabytes of data)	Vertically scalable (challenging at large scale)

Hive Applications

- Summarization
  - Daily/Weekly aggregations of impression/click counts
  - Complex measures of user engagement
- Ad hoc Analysis
  - How many group admins broken down by state/country
- Data Mining (Assembling training data)
  - User Engagement as a function of user attributes
- Spam Detection
  - Anomalous patterns for Site Integrity
  - Application API usage patterns
- Ad Optimization
- etc etc

Hive Setup

hive-env.sh

```
# Set HADOOP_HOME to point to a specific hadoop install directory
export HADOOP_HOME=/Users/tom/Downloads/hadoop-3.2.3

# Set JAVA_HOME
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_202.jdk/Contents/Home
```

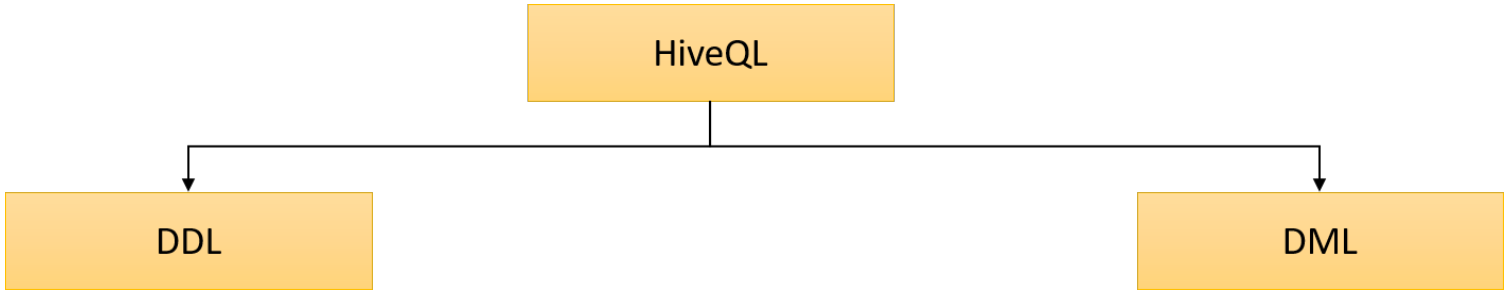
hive-site.xml

```
<!-- Properties for MySql as Metastore -->
<property>
   <name>javax.jdo.option.ConnectionURL</name>
   <value>jdbc:mysql://localhost:3306/metastore?createDatabaseIfNotExist=TRUE</value>
   <description>metadata is stored in a MySQL server</description>
</property>
<property>
   <name>javax.jdo.option.ConnectionDriverName</name>
   <value>com.mysql.cj.jdbc.Driver</value>
   <description>MySQL JDBC driver class</description>
</property>
<property>
   <name>javax.jdo.option.ConnectionUserName</name>
   <value>root</value>
   <description>user name for connecting to mysql server</description>
</property>
<property>
   <name>javax.jdo.option.ConnectionPassword</name>
   <value>password</value>
   <description>password for connecting to mysql server</description>
</property>
```

Basic HiveQL Commands

Command	Description
SHOW DATABASES;	List all databases
DESCRIBE DATABASE database_name;	Display the database's properties
USE database_name;	Switch to a specific database
SHOW TABLES;	List all tables in the current database
DESCRIBE table_name;	Describe the structure of a table
DESCRIBE EXTENDED table_name;	Extended description of a table
SHOW TBLPROPERTIES table_name;	Show the table properties
SHOW PARTITIONS table_name;	List all partitions of a partitioned table
SELECT * FROM table_name LIMIT 10;	Preview the content of a table
SELECT COUNT(*) FROM table_name;	Count the number of rows in a table

HiveQL



CREATE DATABASE [IF **NOT EXISTS**] <db_name>
DROP DATABASE [IF **EXISTS**] <db_name>
ALTER DATABASE

CREATE [EXTERNAL] TABLE [IF **NOT EXISTS**] <tbl_name>
DROP TABLE [IF **EXISTS**] <tbl_name>
ALTER TABLE

LOAD DATA [**LOCAL**] INPATH ?filepath?
[OVERWRITE] **INTO TABLE** <tb1 > ;

INSERT OVERWRITE **INTO TABLE** <tbl >
[IF **NOT EXISTS**] select statement ;

INSERT INTO TABLE tbl1 select statement1
INSERT INTO TABLE tb12 select statement2 ;

SELECT expression
FROM tbl1 [**JOIN** tbl2 **ON** condition]
[**WHERE** Boolean expression]
[**Having** aggreaate Boolean expression]
[**ORDER BY** col _list]
[**GROUP BY** col_list]
[**CLUSTER BY** col_list | [**DISTRIBUTE BY** col_list]
[**SORT BY** col_list]]
[**LIMIT** rows]

HiveQL DDL

- Hive Stores data on HDFS
- DDL Commands affects
  - Metastore (metadata or catalog)
  - Folders on HDFS
  - Files on HDFS
  - Content of the different HFDS file blocks

Hive Data Types

- Primitive
  - INT, SMALLINT, TINYINT, BIGINT
  - DOUBLE, FLOAT, DECIMAL
  - DATE, TIMESTAMP, INTERVAL
  - STRING, VARCHAR, CHAR
- Union: value can match one of the sub types
- Complex
  - Array,
  - Map,
  - Struct

More details on https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types

CREATE DATABASE

- Databases are simply name spaces

CREATE DATABASE [IF **NOT EXISTS**] <dbname>;

- Adds an entry to Hive's metastore
- Creates a new folder in HDFS under Hive's folder
  - Default /user/hive/warehouse/<dbname>.db
- Can be dropped

DROP DATABASE [IF **EXISTS**] <dbname> [**CASCADE**]

CREATE TABLE

- A table maps to a subdirectory on HDFS
  - Creates a subdirectory under the respective database directory /user/hive/warehouse/tablename
  - Adds an entry to Hive's metastore

```
CREATE TABLE tablename (colname DATATYPE, ...)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY char
STORED AS {TEXTFILE | SEQUENCEFILE | AVRO | ...}
```

- Create table based on another table

```
CREATE TABLE jobsarchived LIKE jobs;
CREATE TABLE nycustomers AS
SELECT custid, fname, lname FROM
customers WHERE state = 'NY';
```

CREATE TABLE

- You can explicitly point to the location of the table's HDFS folder

```
CREATE TABLE jobs (id INT, title STRING, salary INT, posted TIMESTAMP)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ', '
LOCATION <HDFS_FOLDER>;
```

- Unlike RDBMS, data may exist before the schema.
- Schema is just a wrapping of HDFS files to allow SQL-like access to data
- However, if you drop the table its data will be deleted from HDFS!

External Tables

- Hive allows dropping tables from its Metastore but without deleting the actual data
- Use keyword EXTERNAL when creating the table
- Location must be specified

```
CREATE EXTERNAL TABLE adclicks (
campaign_id STRING, click_time TIMESTAMP, keyword STRING, site
STRING, placement STRING, was_clicked BOOLEAN, cost SMALLINT)
LOCATION '/sales/ad_data';
```

Loading Data

- Via HDFS fs command
  hdfs fs -mv /tmp/sales.txt /user/hive/warehouse/sales/

- Load Data command
  LOAD DATA INPATH '/tmp/sales.txt' [OVERWRITE] INTO TABLE sales;

- Based on a query

```
INSERT INTO TABLE loyal_customers
SELECT * FROM accounts
WHERE YEAR( acct_create_dt ) = 2008 AND acct_close_dt IS NULL ;
```

Loading Data

- Unlike RDBMS, Hive does not apply schema on write
  - Schema is just a wrapping layer to allow SQL-like access.
  - Data is still stored in HDFS as files
- Hive applies schema on read
  - Missing columns in the underlying file will be replaced with Null

HiveQL: DML

Follows SQL

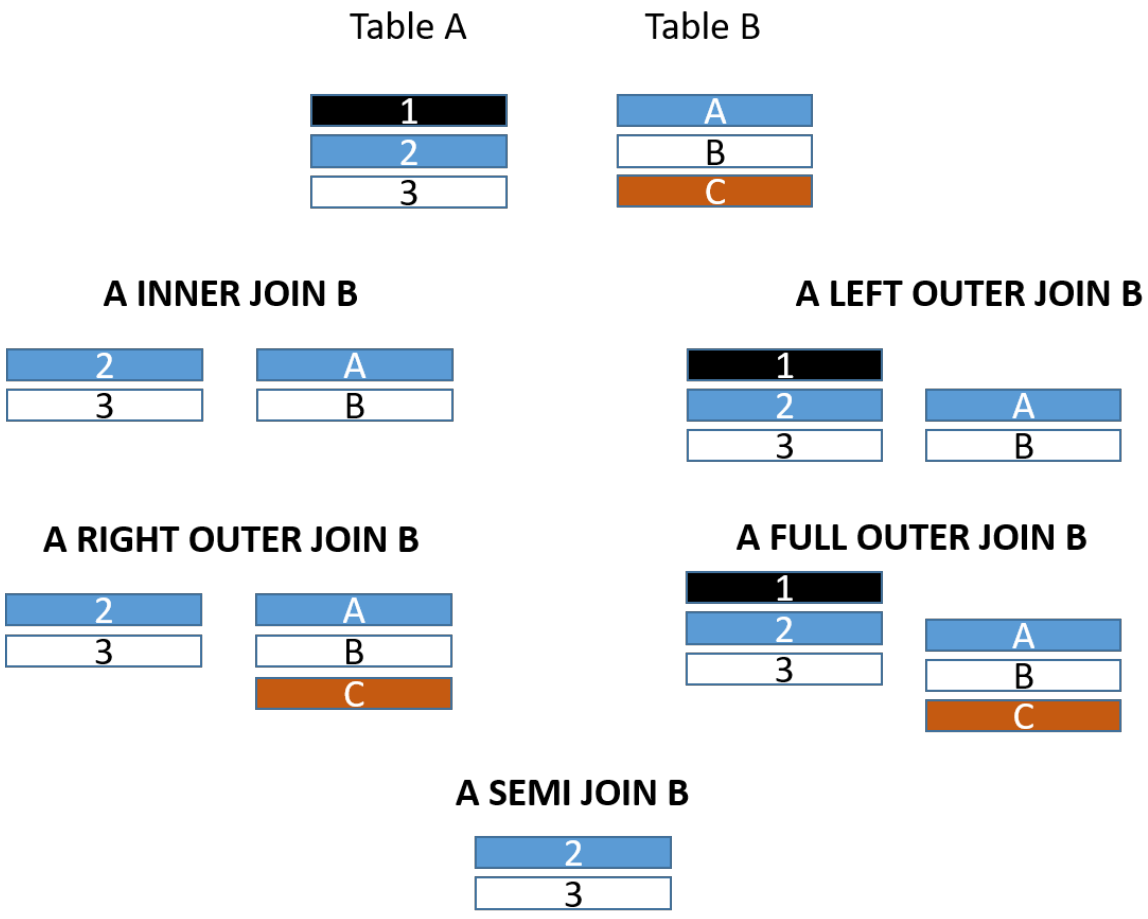
```
WITH CommonTableExpression AS ( /* define CTE */ ),
AnotherCommonTableExpression AS ( /* define another CTE */ )
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[ORDER BY col_list]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT [offset,] rows]
```

HiveQL: DML

Clause	Purpose
WITH CTE AS	Defines a Common Table Expression to simplify complex subqueries and improve readability.
SELECT [ALL | DISTINCT]	Selects data from tables; DISTINCT removes duplicate rows.
FROM table_reference	Specifies the table to select data from, can be a CTE.
WHERE condition	Filters results based on a condition.
GROUP BY col_list	Groups results by one or more columns for aggregation.
ORDER BY col_list	Orders the result set by one or more columns.
CLUSTER BY col_list	Distributes and sorts the data within each reducer by column values.
DISTRIBUTE BY col_list	Distributes the rows to reducers based on column values without sorting.
SORT BY col_list	Sorts data within each reducer, not across the entire result set.
LIMIT [offset,] rows	Limits the number of rows returned, optionally skipping a number of rows first.

Joins

- Hive Supports
  - Inner join
  - Left outer join
  - Right outer join
  - Full outer join
  - Semi join



Hive Query Performance

- Query Performance is subject to different aspects
  - File storage format: text files versus binary files, e.g., ORC, Avro, Parquet, ?
  - Data layout on the disk: How many files are there for each table?
    - Partitioning
  - Execution engines
    - Yarn
    - Tez
    - Vectorized execution
  - Query plan optimization

Hive Query Performance

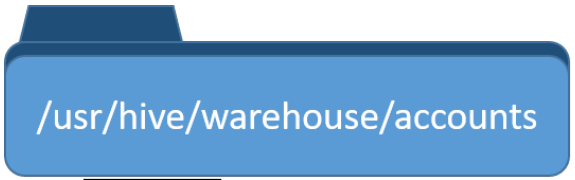
- **Select * from tbl:**
  - Hive reads the entire table to retrieve all the data.
- **Select * from tbl where col1 = '124’:**
  - Plain: hive reads all the data but only returns rows where col1 equals '124’.
  - Partitioned: hive only reads the section of the table with '124' in col1, which is more efficient.
- **Select col0, col2 from tbl where col1 = '124’:**
  - Plain: hive acts like the above but only returns the specified columns, col0 and col2.
  - Partitioned: hive reads only the relevant section.
  - ORC format: hive is even more efficient, reading only the necessary columns directly from disk.

Partitioning

- We can specify one or more columns to partition the storage of Table's data.
- Declared in Create table statement
- Maps to subdirectories under table's directory
- When partitioning column(s) are used in `Where` clause, Hive only reads files under those partitions' subdirectories, resulting in speeding up query processing

Partitioning

Table Accounts non-partitioned




```
Select cust_ID, FName from accounts
where state = 'AZ'
```

File 1

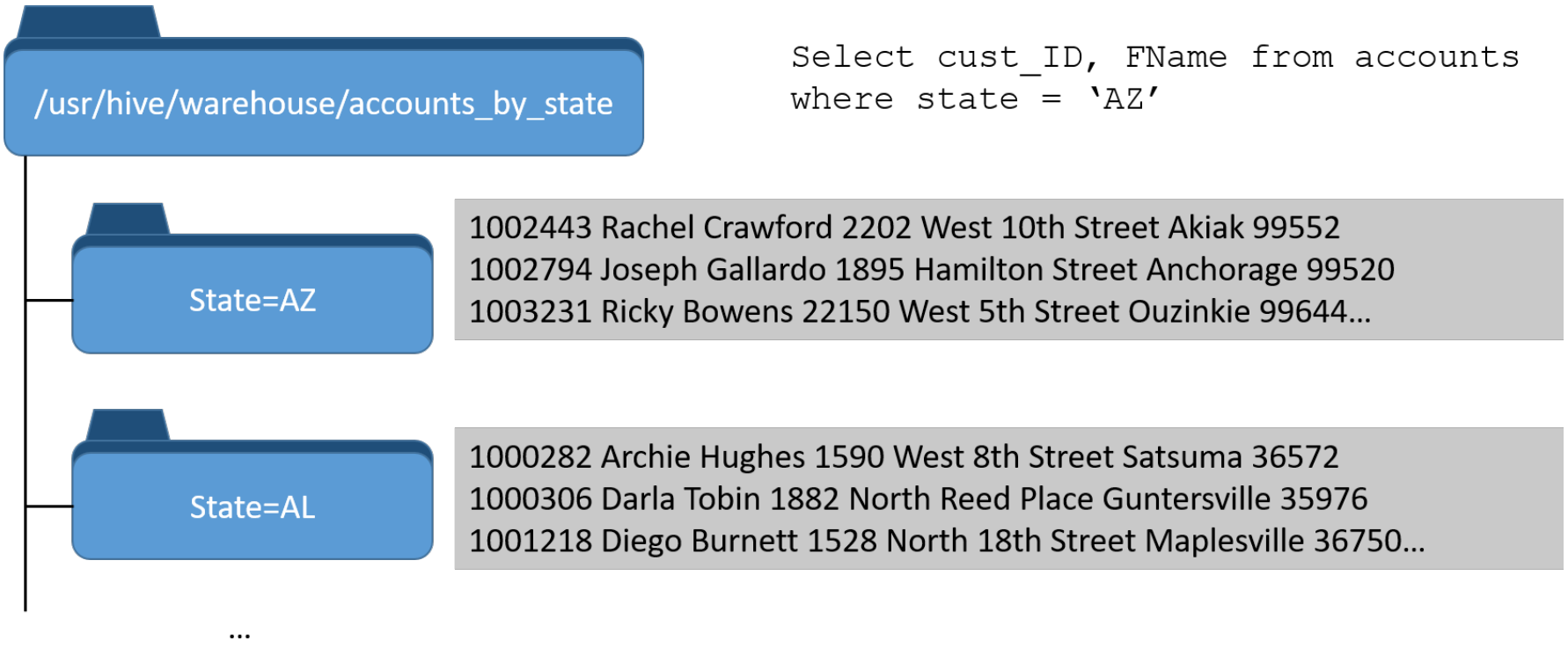
1000000 Quentin Shepard 32092 West 10th Street Prairie City SD 57649
1000001 Brandon Louis 1311 North 2nd Street Clearfield IA 50840
1000002 Marilyn Ham 25831 North 25th Street Concord CA 94522
…

File 2

1050344 Denise Carey 1819 North Willow Parkway Phoenix AZ 85042
1050345 Donna Pettigrew 1725 Patterson Street Garberville CA 95542
1050346 Hans Swann 1148 North Hornbeam Avenue Sacramento CA 94230
…

Partitioning

Table Accounts_by_state partitioned




```
Select cust_ID, FName from accounts
where state = 'AZ'
```

1002443 Rachel Crawford 2202 West 10th Street Akiak 99552
1002794 Joseph Gallardo 1895 Hamilton Street Anchorage 99520
1003231 Ricky Bowens 22150 West 5th Street Ouzinkie 99644…

1000282 Archie Hughes 1590 West 8th Street Satsuma 36572
1000306 Darla Tobin 1882 North Reed Place Guntersville 35976
1001218 Diego Burnett 1528 North 18th Street Maplesville 36750…

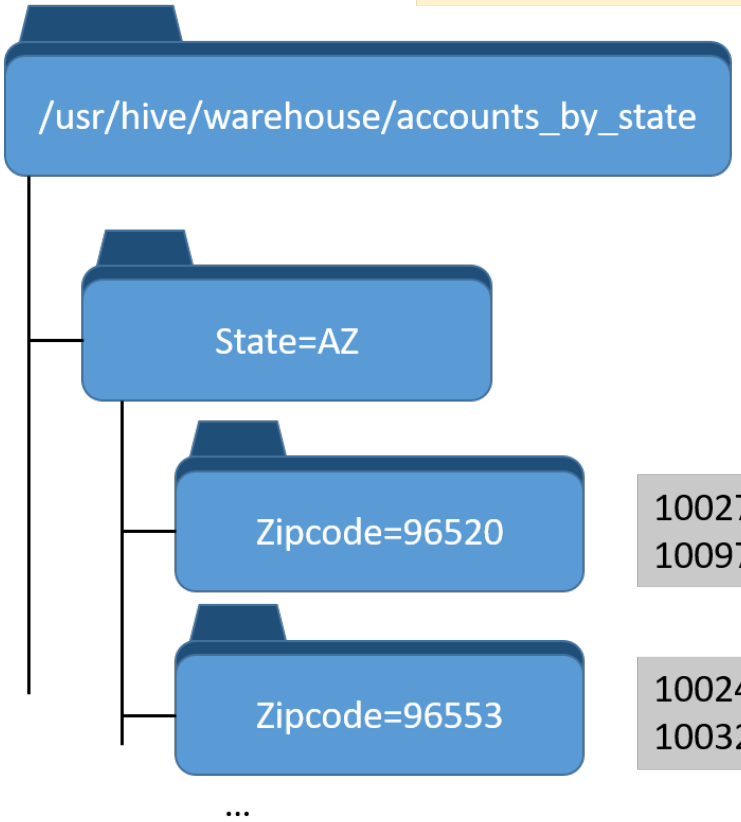
Partitioning

```
CREATE EXTERNAL TABLE accounts
(
cust_id INT, fname STRING,
lname STRING, address STRING,
city STRING,
state STRING,
zipcode STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/sales/accounts';
```

```
CREATE EXTERNAL TABLE accounts
(
cust_id INT, fname STRING,
lname STRING, address STRING,
city STRING,
zipcode STRING
)
PARTITIONED BY (state STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/sales/accounts-by-state';
```

Nested Partitioning

… PARTITIONED BY (state STRING , zipcode STRING)



1002794 Joseph Gallardo 1895 Hamilton Street Anchorage
1009777 Tree van Nilson 1331 Village Lane Anchorage …

1002443 Rachel Crawford 2202 West 10th Street Akiak
1003232 Penny Lane 233 West 5th Street Akiak …

Loading Data into Partitioned Table

- Dynamic partitioning
  - New partitions are added automatically at load time
  - Data is stored in the correct partition (subdirectory) based on column value

```
INSERT OVERWRITE TABLE accounts_by_state PARTITION (state)
SELECT custid, fname, lname, address, city, zipcode, state
FROM accounts;
```

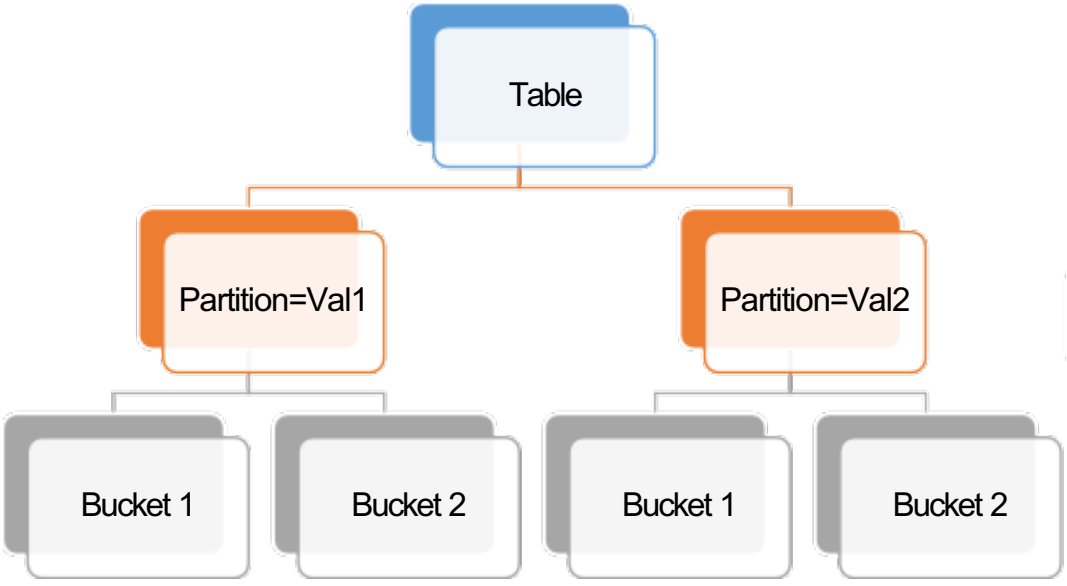
- Static partitioning
  - Invoke ADD PARTITION DDL

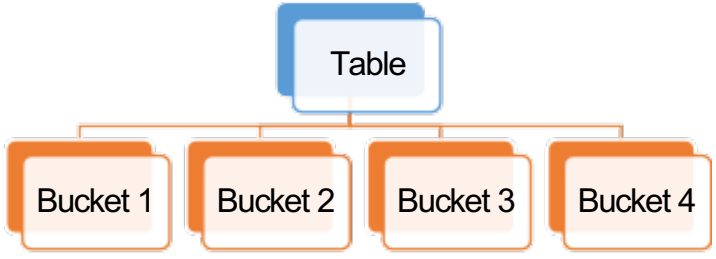
```
ALTER TABLE accounts_by_state ADD PARTITION (state='AZ');
LOAD DATA INPATH '/mystaging/AZ.txt' INTO TABLE
accounts_by_state [OVERWRITE] PARTITION (state='AZ');
```

Bucketing Data

- Partitioning is one way to organize the data on HDFS in the form of sub-folders.
- Without bucketing, each partition will have one file containing all the data from the table for the respective partition.
- Bucketing clusters data into manageable "buckets" based on the value of a hash function applied to a column(s) in a table.
- Unlike partitioning, data engineer has to explicitly define the number of buckets.
- Bucketing can be used with/without partitioning

Partitioning and Bucketing





Why bucketing might be needed?

- Sometimes, partitioning might produce unbalanced data within partitions
- Bucketing is useful for:
    - **Optimizing Joins**: It supports efficient join operations like sort-merge joins by organizing data into structured buckets.
    - **Map-side Joins**: It allows for joins to be executed at the map phase, saving on processing time by avoiding data shuffling across reducers.
    - **Faster Filters**: It speeds up query filters by confining searches to relevant buckets instead of the whole table.

Creating a bucketed table

```
CREATE TABLE bucketed_user (
  firstname VARCHAR(64) ,
  lastname VARCHAR(64) ,
  address STRING,
  city VARCHAR(64) ,
  state VARCHAR(64) ,
  post STRING ,
  phone1 VARCHAR(64) ,
  phone2 STRING ,
  email STRING ,
  web STRING)
PARTITIONED BY (country VARCHAR(64))
CLUSTERED BY (state)
SORTED BY (city)
INTO 32 BUCKETS STORED AS
SEQUENCEFILE;
```

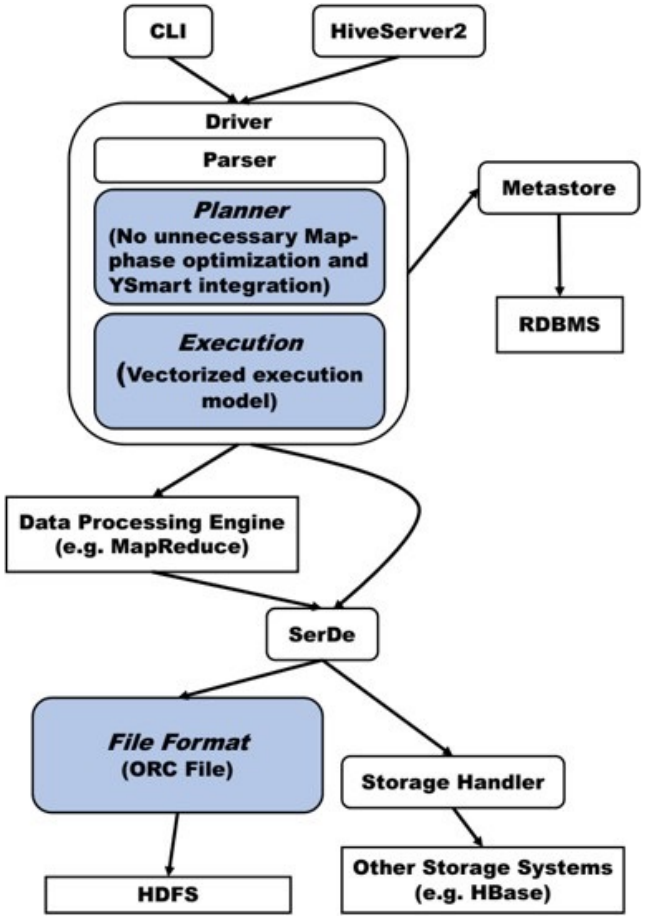
Loading data into a bucketed table

We cannot use LOAD DATA INPATH ... to populate a bucketed table.
We have to use INSERT (OVERWRITE) TABLE

```
set hive.enforce.bucketing = true ;
INSERT OVERWRITE TABLE bucketed_user
PARTITION (country)
SELECT firstname ,
lastname,
address ,
city ,
state ,
post ,
phone1 ,
phone2 ,
email ,
web ,
country
FROM temp_user ;
```

Query Processing Improvements[1,2]

- File Formats
  - ORC
- Query Planning. Avoid: Unnecessary
  - Map phases Unnecessary data
  - loading Unnecessary data
  - re-partitioning
- Query Execution
  - Vectorized execution



[1]Huai, Yin, et al. "Major technical advancements in apache hive." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014.

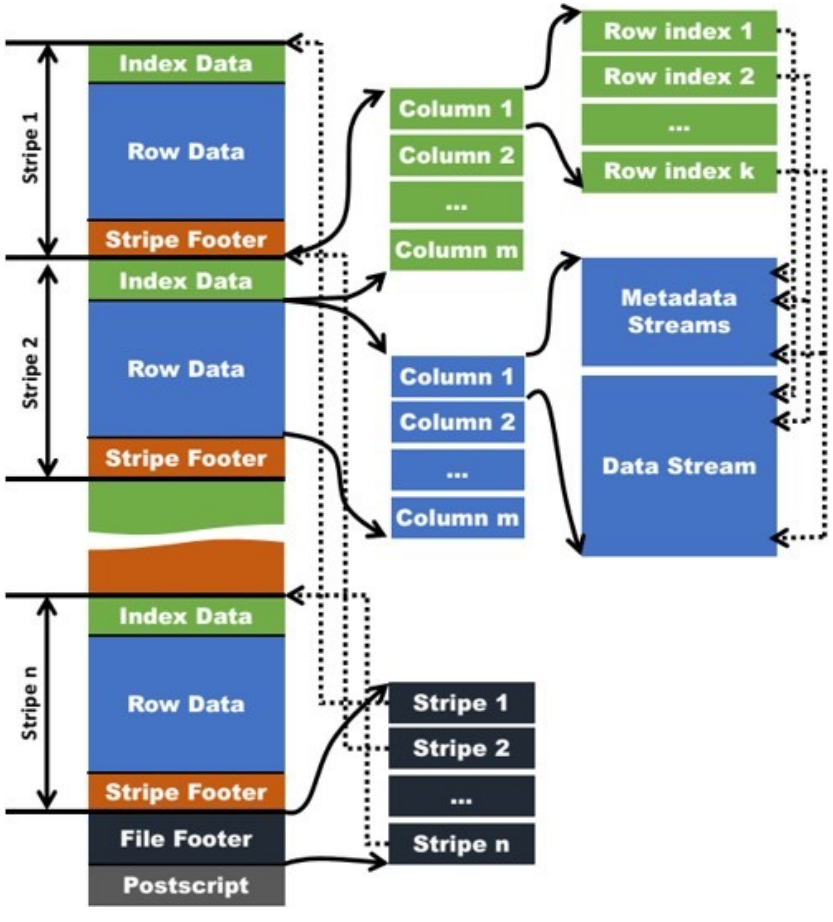
[2]Lee, Rubao, et al. "Ysmart: Yet another sql-to-mapreduce translator." 2011 31st International Conference on Distributed Computing Systems. IEEE, 2011.

Query Processing Improvements[1,2]

- **Client**: The user interface where SQL-like queries are submitted.
- **Driver**: The component that receives queries. It has three main sub-components:
    - **Parser**: Checks the syntax and parses the SQL queries.
    - **Planner**: Generates the execution plan by deciding how the query will be executed, including which algorithms to use for joins and aggregations.
    - **Optimizer**: Improves the execution plan for efficient query processing by applying various optimizations.
- **Execution Engine**: Executes the plan generated by the planner and optimized by the optimizer.
- **HDFS (Hadoop Distributed File System)**: The storage system used to store large datasets across multiple machines.
- **SerDe (Serializer/Deserializer)**: Responsible for converting the query results from the HDFS format to the format understood by the user and vice versa.
- **File Format (ORC files)**: Indicates the type of file format used, which is often ORC (Optimized Row Columnar) for efficient storage and fast reads.
- **Metadata**: Stores metadata about the tables such as schema and location.
- **Other Storage Systems**: Apart from HDFS, the architecture can also integrate with other storage systems.

Optimized Record Columnar: ORC

- Limitations of earlier formats
  - Data-type agnostic
  - SerDe handle one row at a time
  - Compression not possible (type-based compression)
  - Sequential access only
  - Unable to provide fine-grained access to properties of complex types
- ORC addresses these deficiencies
  - Indexes,
  - Table (data) placement
  - Memory manager



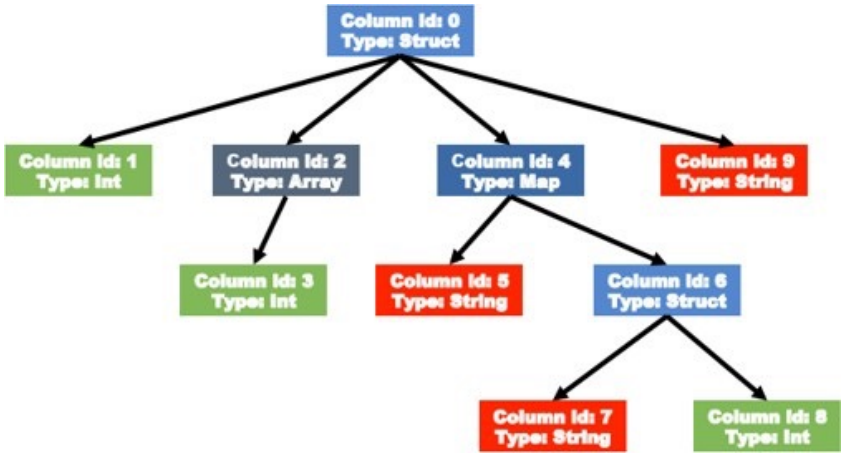
Optimized Record Columnar: ORC

ORC (Optimized Row Columnar) files, are used in Hadoop ecosystems for efficient storage of large data sets.

- **Stripes**: ORC files are divided into stripes, which are large blocks of data (typically around 64MB to 256MB). Each stripe contains a subset of the data and can be read independently.
- **Index Data**: Within each stripe, there is an index data section that contains statistics about the data in that stripe (like min and max values), to help match query criteria.
- **Row Data**: This is the actual data content, stored in a columnar format within the stripe, which is beneficial for analytical querying and compression.
- **Stripe Footer**: It contains metadata about the stripe, such as the number of rows, the encoding of each column, and the directory of streams which tells where each column's data starts in the row data section.
- **File Footer**: The file footer contains metadata about the ORC file, such as the schema of the data, the number of rows in the file, and a list of stripes with their offsets and lengths.
- **Postscript**: Holds information about the ORC file itself, such as the version of the writer, the compression used, and the size of the file footer.
- **Metadata Streams**: Contains additional metadata that describes the data in more detail, useful for readers to understand the data structure and types.

ORC: Table Placement

- Enhancements over RC file format
  - Larger block (stripe) size 256 MB
  - Decomposition of Complex columns, e.g., Maps or structs, into multiple child columns
  - Aligning stripe boundaries with HDFS blocks
    - No stripe crosses stored over multiple blocks




```
CREATE TABLE tbl (
col1 Int,
col2 Array<Int>,
col4 Map<String,
Struct<col7: String, col8: Int>>,
col9 String)
```

ORC: Indexing

- Indexes speed up data access and allow random access
- Two types of indexes:
  - Data statistics
    - Statistics are created by the ORC Writer (at data creation time),
    - Used by ORC Reader to avoid reading unnecessary data
    - Statistics: number of values, min, max, average, and length
    - File, Stripe, Index-level statistics
  - Position Pointers
    - To locate entries to metadata, e.g., data statistics and internal columns
    - To locate entries to data, i.e., leaf columns data

ORC: Memory Manager

- Set a threshold for memory allocated to writers
- If threshold is exceeded, resize stripes of newly registered writers
- When writers close, resize the stripes
- This ensures writing to ORC file does not run out of memory