COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 9: NoSQL and Hive

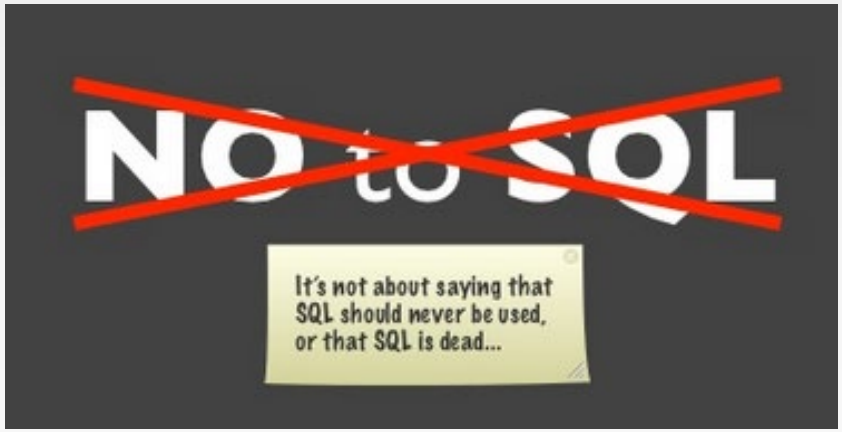
Part 1: Introduction to NoSQL

What does RDBMS provide?

- Relational model with schemas
- Powerful, flexible query language (SQL)
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support (MySQL, PostgreSQL, etc.)

What is NoSQL?

- The name stands for **N**ot **O**nly **SQL**
- Does not use SQL as querying language
- Class of non-relational data storage systems
- The term NOSQL was introduced by Eric Evans when an event was organized to discuss open-source distributed databases
- It's not a replacement for a RDBMS but compliments it
- All NoSQL offerings relax one or more of the ACID properties (will talk about the CAP theorem)



What is NoSQL?

- Key features (advantages):
  - non-relational
  - don't require strict schema
  - data are replicated to multiple nodes (so, identical & fault-tolerant) and can be partitioned:
    - down nodes easily replaced
    - no single point of failure
  - horizontal scalable
  - cheap, easy to implement (open-source)
  - massive write performance
  - fast key-value access

Why NoSQL ?

- Web apps have different needs (than the apps that RDBMS were designed for)
  - Low and predictable response time (latency)
  - Scalability & elasticity (at low cost!)
  - High availability
  - Flexible schemas / semi-structured data
  - Geographic distribution (multiple datacenters)
- Web apps can (usually) do without
  - Transactions / strong consistency / integrity
  - Complex queries

Who are Using NoSQL?

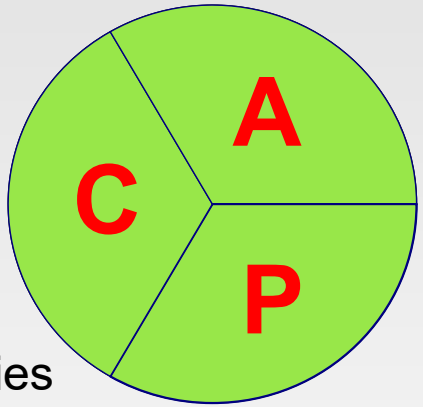
- Google (BigTable)
- LinkedIn (Voldemort)
- Facebook (Cassandra)
- Twitter (HBase, Cassandra)
- Baidu (HyperTable)

Three Major Papers for NoSQL

- Three major papers were the seeds of the NoSQL movement
  - BigTable (Google)
  - Dynamo (Amazon)
  - CAP Theorem (discuss in the next few slides)

ACID & CAP

- **ACID**
  - A DBMS is expected to support “ACID transactions,” processes that are:
  - **A**tomicity: either the whole process is done or none is
  - **C**onsistency: only valid data are written
  - **I**solation: one operation at a time
  - **D**urability: once committed, it stays that way

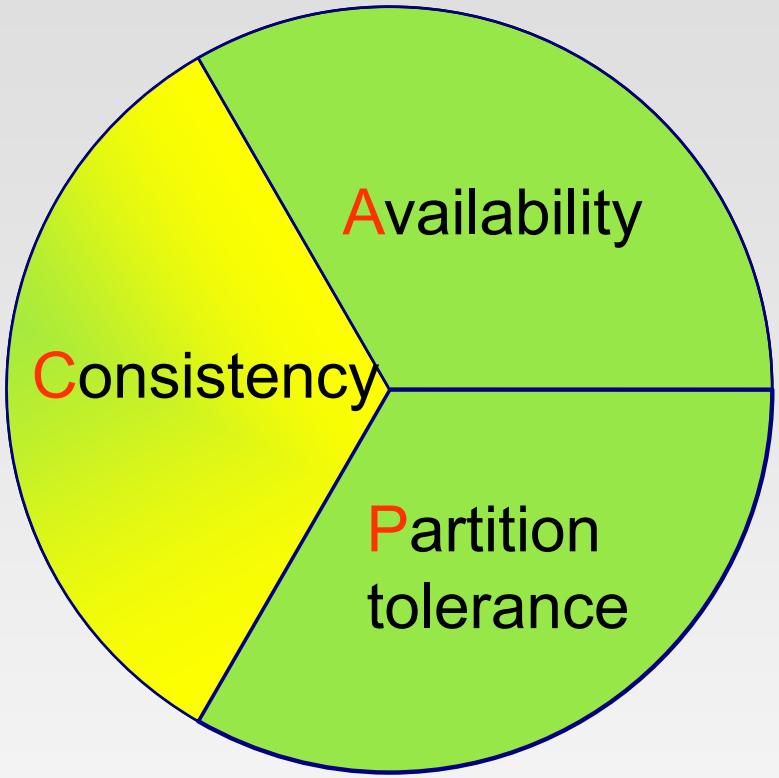


- **CAP** (suppose three properties of a distributed system)
  - **C**onsistency: all data on cluster has the same copies
  - **A**vailability: cluster always accepts reads and writes
  - **P**artition tolerance: guaranteed properties (consistency and/or availability) are maintained even when network failures prevent some machines from communicating with others

CAP Theorem

- Brewer's CAP Theorem:
  - *For any system sharing data, it is "impossible" to guarantee simultaneously all of these three properties*
  - You can have at most two of these three properties for any shared-data system
- Very large systems will "partition" at some point:
  - That leaves either **C** or **A** to choose from (traditional DBMS prefers **C** over **A** and **P**)
  - In almost all cases, you would choose **A** over **C** (except in specific applications such as order processing)

CAP Theorem: Consistency



All client always have the same view of the data

Once a writer has written, all readers will see that write

- Two kinds of consistency:
  - strong consistency – ACID (Atomicity Consistency Isolation Durability)
  - weak consistency – BASE (Basically Available Soft-state Eventual consistency)

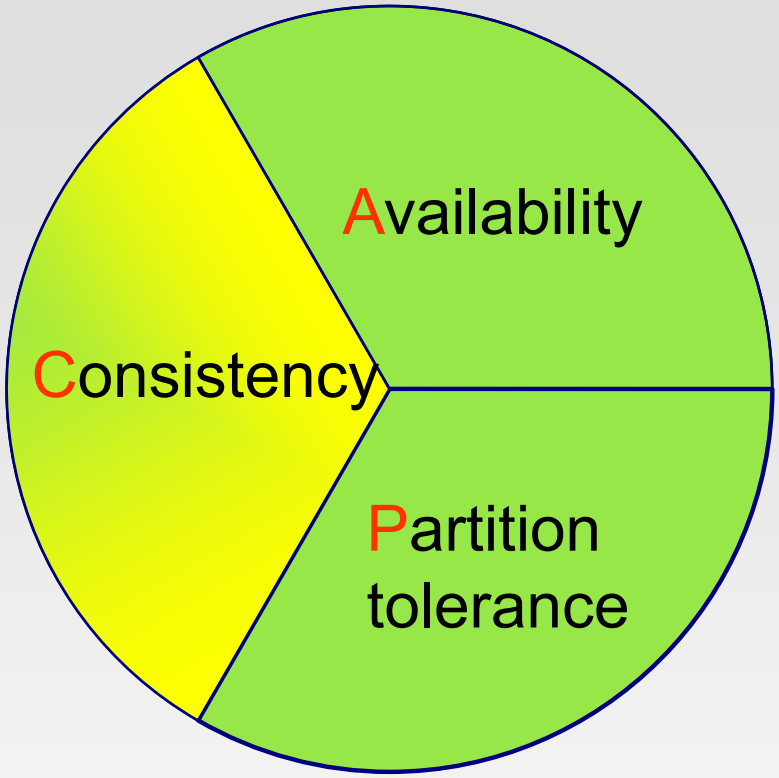
Consistency Model

- A consistency model determines rules for visibility and apparent order of updates
- Example:
  - Row X is replicated on nodes M and N
  - Client A writes row X to node N
  - Some period of time t elapses
  - Client B reads row X from node M
  - **Does client B see the write from client A?**
  - Consistency is a continuum with tradeoffs
  - **For NOSQL, the answer would be: "maybe"**
  - CAP theorem states: *"strong consistency can't be achieved at the same time as availability and partition-tolerance"*

Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
- http://en.wikipedia.org/wiki/Eventual_consistency

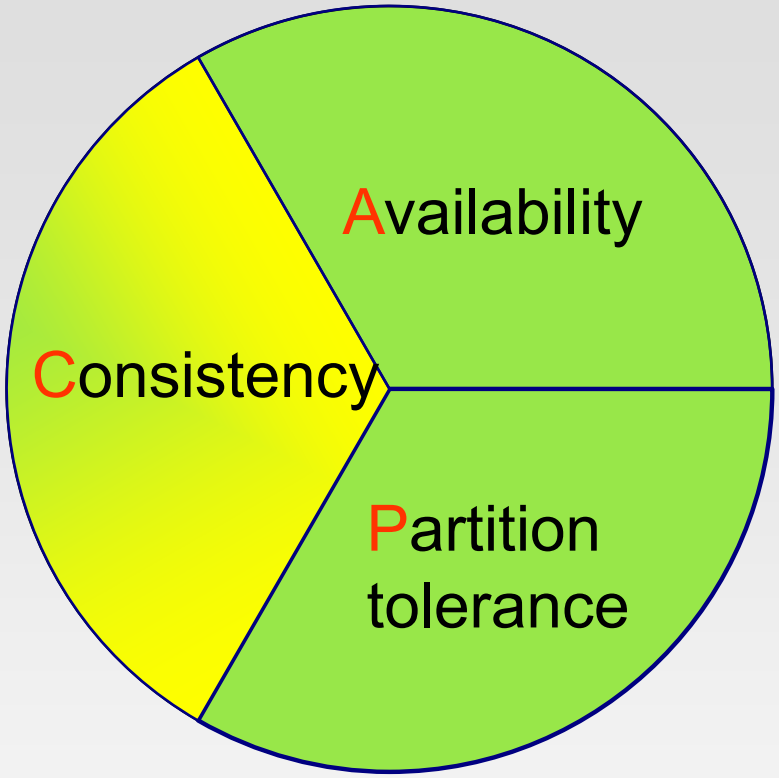
CAP Theorem: Availability



System is available during software and hardware upgrades and node failures.

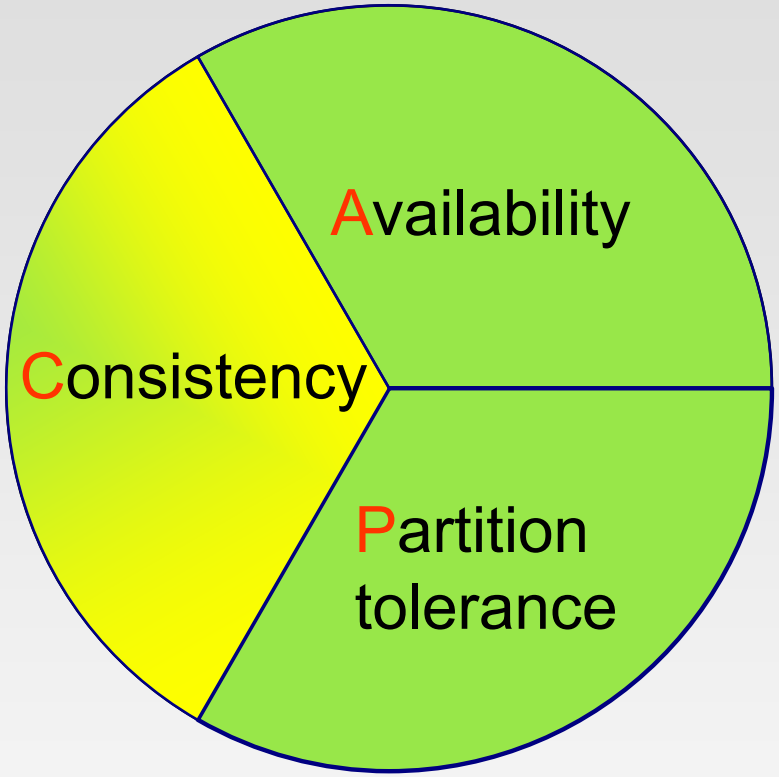
- Traditionally, thought of as the server/process available five 9's (99.999 %).
  - However, for large node system, at almost any point in time there's a good chance that a node is either down or there is a network disruption among the nodes.
  - Want a system that is resilient in the face of network disruption

CAP Theorem: Partition-Tolerance



A system can continue to operate in the presence of a network partitions.

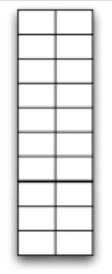
CAP Theorem

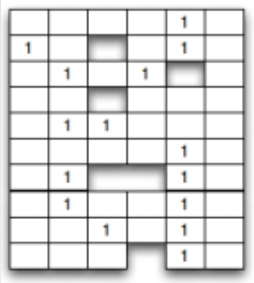


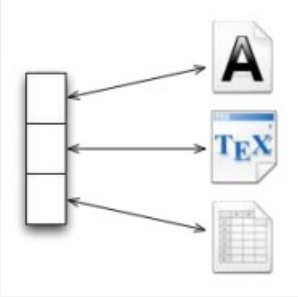
CAP Theorem: You can have at most **two** of these properties for any shared-data system

NoSQL Taxonomy

- Key-Value stores
  - Simple K/V lookups (DHT)
- Column stores
  - Each key is associated with many attributes (columns)
  - NoSQL column stores are actually hybrid row/column stores
    - Different from “pure” relational column stores!
- Document stores
  - Store semi-structured documents (JSON)
- Graph databases
  - Neo4j, etc.
  - Not exactly NoSQL
    - can’t satisfy the requirements for High Availability and Scalability/Elasticity very well







Key-value

- Focus on scaling to huge amounts of data
- Designed to handle massive load
- Based on Amazon's dynamo paper
- Data model: (global) collection of Key-value pairs
- *Dynamo ring partitioning* and *replication*
- Example: (DynamoDB)
  - *items* having one or more attributes (name, value)
  - An *attribute* can be single-valued or multi-valued like set.
  - items are combined into a *table*

Key-value

- Basic API access:
  - get(key): extract the value given a key
  - put(key, value): create or update the value given its key
  - delete(key): remove the key and its associated value
  - execute(key, operation, parameters): invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc)

Key-value

- Pros:
  - very fast
  - very scalable (horizontally distributed to nodes based on key)
  - simple data model
  - eventual consistency
  - fault-tolerance

- Cons
  - Can’t model more complex data structure such as objects

Key-value

Name	Producer	Data model	Querying
SimpleDB	Amazon	set of couples (key, {attribute}), where attribute is a couple (name, value)	restricted SQL; select, delete, GetAttributes, and PutAttributes operations
Redis	Salvatore Sanfilippo	set of couples (key, value), where value is simple typed value, list, ordered (according to ranking) or unordered set, hash value	primitive operations for each value type
Dynamo	Amazon	like SimpleDB	simple get operation and put in a context
Voldemort	LinkeIn	like SimpleDB	similar to Dynamo

Document-based

- Can model more complex objects
- Inspired by Lotus Notes
- Data model: collection of documents
- Document: JSON (**J**ava**S**cript **O**bject **N**otation is a data model, key-value pairs, which supports objects, records, structs, lists, array, maps, dates, Boolean with **nesting**), XML, other semi-structured formats.
- Example: (MongoDB) document
  - {Name:"Jaroslav",

    Address:"Malostranske nám. 25, 118 00 Praha 1”,

    Grandchildren: {Claire: "7", Barbara: "6", "Magda: "3", "Kirsten: "1", "Otis: "3", Richard: "1“}

    Phones: [“123-456-7890”, “234-567-8963”]

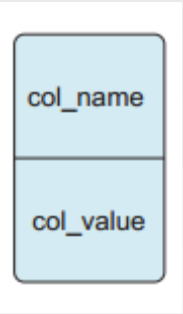
    }

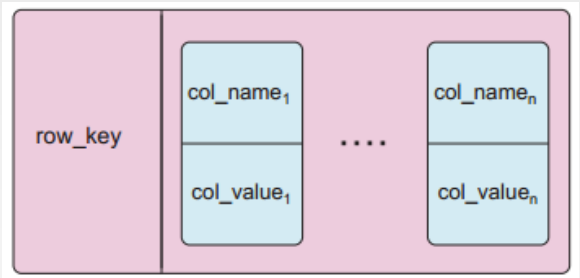
Document-based

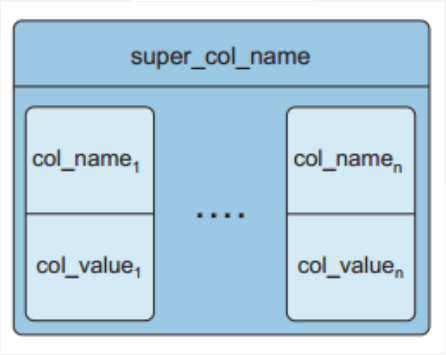
Name	Producer	Data model	Querying
MongoDB	10gen	object-structured documents stored in collections; each object has a primary key called ObjectId	manipulations with objects in collections (find object or objects via simple selections and logical expressions, delete, update,)
Couchbase	Couchbase	document as a list of named (structured) items (JSON document)	by key and key range, views via Javascript and MapReduce

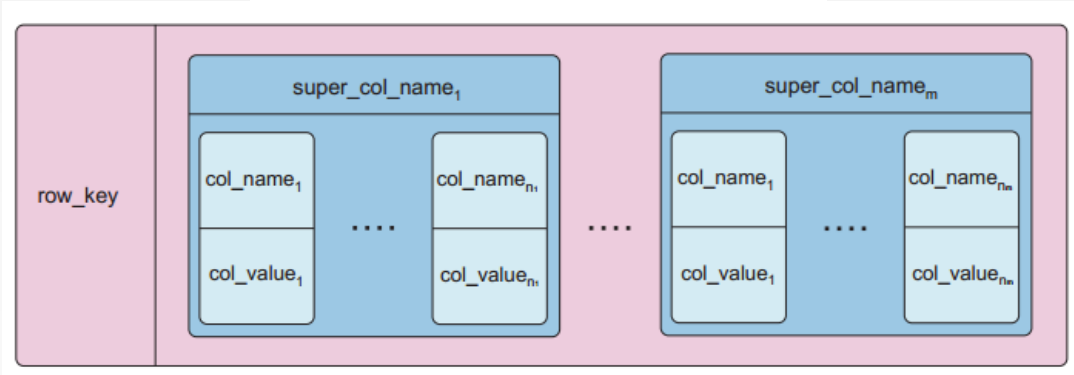
Column-based

- Based on Google's BigTable paper
- Like column oriented relational databases (store data in column order) but with a twist
- Tables similarly to RDBMS, but handle semi-structured
- Data model:
  - Collection of Column Families
  - Column family = (key, value) where value = set of **related** columns (standard, super)
  - indexed by *row key*, *column key* and *timestamp*









Column-based

- One column family can have variable numbers of columns
- Cells within a column family are sorted “physically”
- Very sparse, most cells have null values
- Comparison: RDBMS vs column-based NoSQL
  - Query on multiple tables
    - RDBMS: must fetch data from several places on disk and glue together
    - Column-based NoSQL: only fetch column families of those columns that are required by a query (all columns in a column family are stored together on the disk, so multiple rows can be retrieved in one read operation data locality)

Column-based

Name	Producer	Data model	Querying
BigTable	Google	set of couples (key, {value})	selection (by combination of row, column, and time stamp ranges)
HBase	Apache	groups of columns (a BigTable clone)	JRUBY IRB-based shell (similar to SQL)
Hypertable	Hypertable	like BigTable	HQL (Hypertext Query Language)
CASSANDRA	Apache (originally Facebook)	columns, groups of columns corresponding to a key (supercolumns)	simple selections on key, range queries, column or columns ranges
PNUTS	Yahoo	(hashed or ordered) tables, typed arrays, flexible schema	selection and projection from a single table (retrieve an arbitrary single record by primary key, range queries, complex predicates, ordering, top-k)

Graph-based

- Focus on modeling the structure of data (*interconnectivity*)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory (G=(E,V))
- Data model:
  - (Property Graph) nodes and edges
    - Nodes may have properties (including ID)
    - Edges may have labels or roles
  - Key-value pairs on both
- Interfaces and query languages vary
- *Single-step* vs *path expressions* vs *full recursion*
- Example:
  - Neo4j, FlockDB, InfoGrid …

NoSQL Pros/Cons

- Advantages
  - Massive scalability
  - High availability
  - Lower cost (than competitive solutions at that scale)
  - (usually) predictable elasticity
  - Schema flexibility, sparse & semi-structured data
- Disadvantages
  - Don't fully support relational features
    - no join, group by, order by operations (except within partitions)
    - no referential integrity constraints across partitions
  - No declarative query language (e.g., SQL) → more programming
  - Eventual consistency is not intuitive to program for
    - Makes client applications more complicated
  - No easy integration with other applications that support SQL
  - Relaxed ACID (see CAP theorem later) → fewer guarantees

Conclusion

- NOSQL database cover only a part of data-intensive cloud applications (mainly Web applications)
- Problems with cloud computing:
  - SaaS (Software as a Service or on-demand software) applications require enterprise-level functionality, including ACID transactions, security, and other features associated with commercial RDBMS technology, i.e. NOSQL should not be the only option in the cloud
  - Hybrid solutions:
    - Voldemort with MySQL as one of storage backend
    - deal with NOSQL data as semi-structured data

      ->integrating RDBMS and NOSQL via SQL/XML

Part 2: Introduction to Hive

What is Hive?

- A data warehouse system for Hadoop that
  - facilitates easy data summarization
  - supports ad-hoc queries (still batch though…)
  - created by Facebook
- A mechanism to project structure onto this data and query the data using a SQL-like language – HiveQL
  - Interactive-console –or-
  - Execute scripts
  - Kicks off one or more MapReduce jobs in the background
- An ability to use indexes, built-in user-defined functions
- Latest stable version: 3.1.3, works with Hadoop 3.x.y

Motivation of Hive

- Limitation of MR
  - Have to use M/R model
  - Not Reusable
  - Error prone
  - For complex jobs:
    - Multiple stage of Map/Reduce functions
    - Just like ask developer to write specified physical execution plan in the database
- Hive intuitive
  - Make the unstructured data looks like tables regardless how it really lays out
  - SQL based query can be directly against these tables
  - Generate specified execution plan for this query

Hive Features

- A subset of SQL covering the most common statements
- Agile data types: Array, Map, Struct, and JSON objects
- User Defined Functions and Aggregates
- Regular Expression support
- MapReduce support
- JDBC support
- Partitions and Buckets (for performance optimization)
- Views and Indexes

Word Count using MapReduce

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {

  public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
                       Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }
```

```
  public int run(String[] args) throws Exception {
    JobConf conf = new JobConf(getConf(), WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    List<String> other_args = new ArrayList<String>();
    for(int i=0; i < args.length; ++i) {
      try {
        if ("-m".equals(args[i])) {
          conf.setNumMapTasks(Integer.parseInt(args[++i]));
        } else if ("-r".equals(args[i])) {
          conf.setNumReduceTasks(Integer.parseInt(args[++i]));
        } else {
          other_args.add(args[i]);
        }
      } catch (NumberFormatException except) {
        System.out.println("ERROR: Integer expected instead of " + args[i]);
        return printUsage();
      } catch (ArrayIndexOutOfBoundsException except) {
        System.out.println("ERROR: Required parameter missing from " +
                           args[i-1]);
        return printUsage();
      }
    }
    // Make sure there are exactly 2 parameters left.
    if (other_args.size() != 2) {
      System.out.println("ERROR: Wrong number of parameters: " +
                         other_args.size() + " instead of 2.");
      return printUsage();
    }
    FileInputFormat.setInputPaths(conf, other_args.get(0));
    FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new WordCount(), args);
    System.exit(res);
  }
}
```

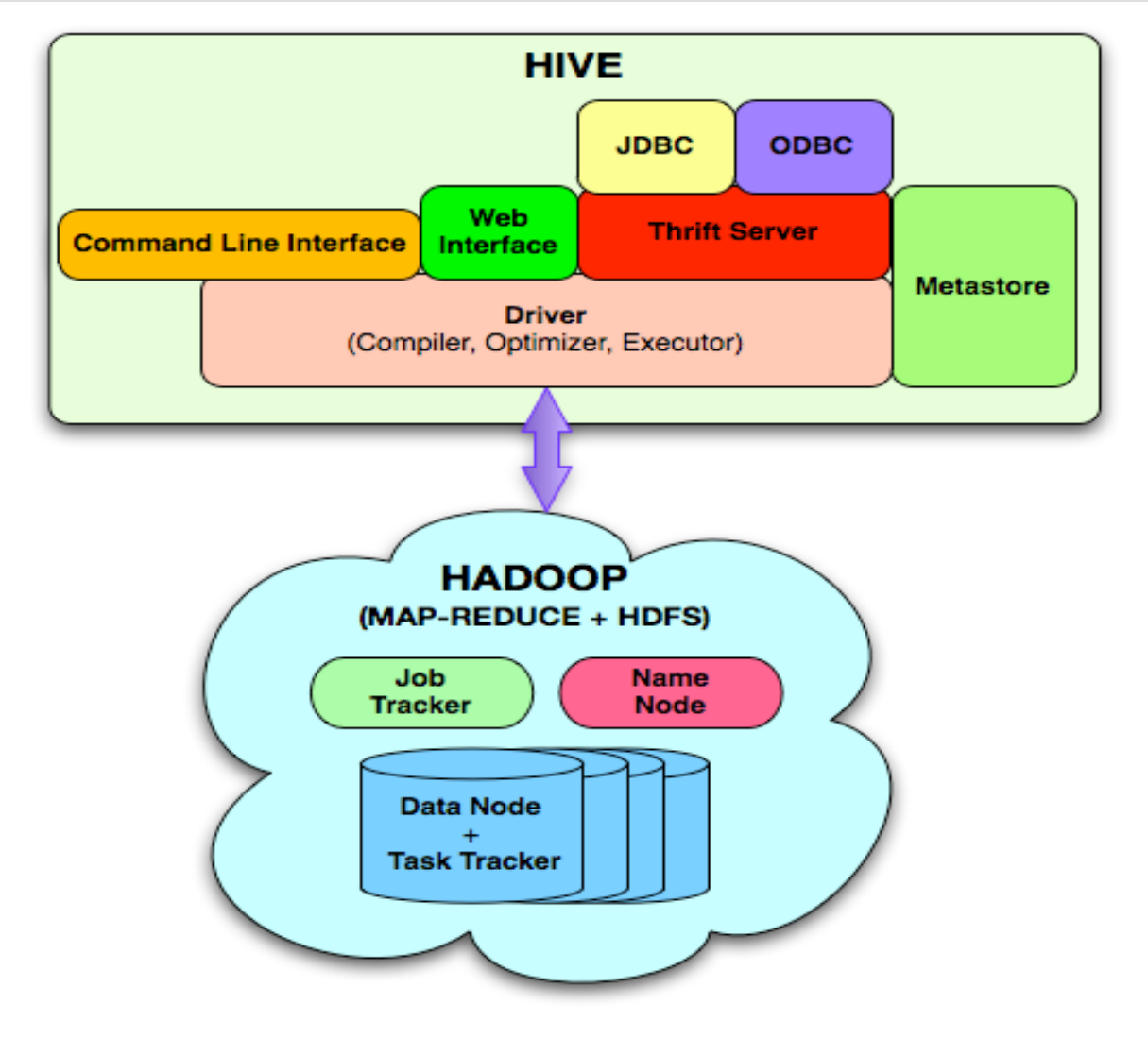
Word Count using Hive

```
create table doc(
text string
) row format delimited fields terminated by '\n' stored as textfile;

load data local inpath '/home/Words' overwrite into table doc;

SELECT word, COUNT(*) FROM (SELECT explode(split(text, ' '))
AS word FROM doc) wTable GROUP BY word;
```

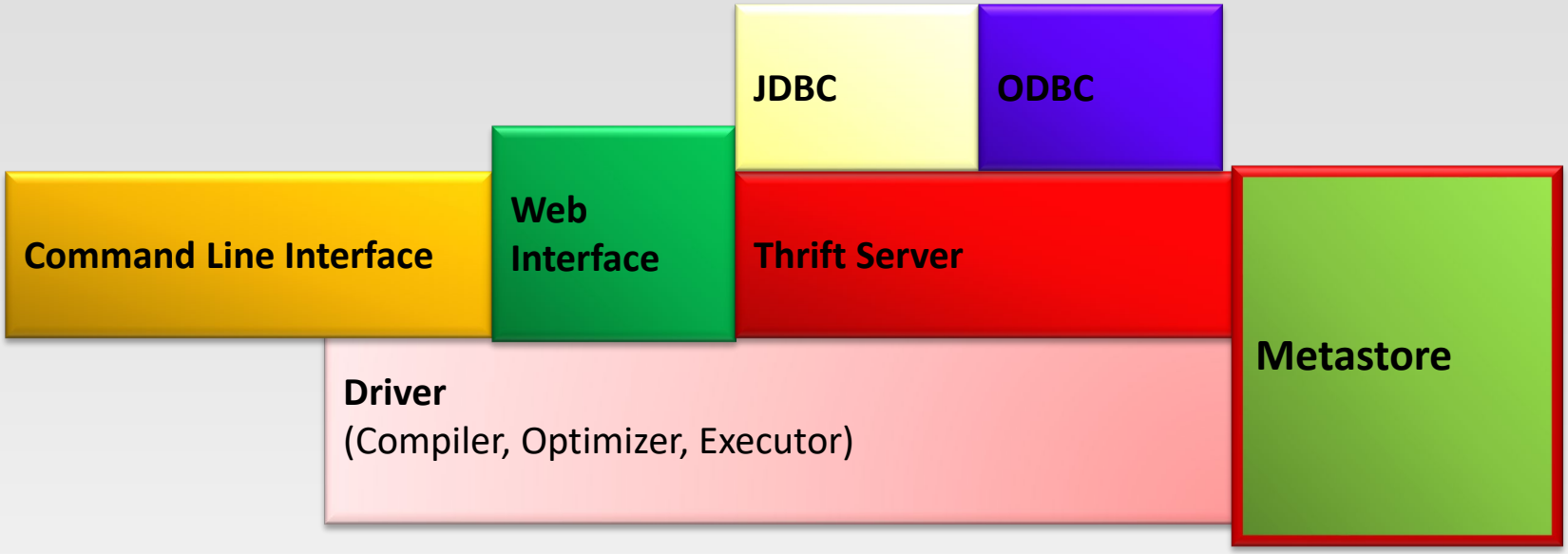
Word Count using Hive

```
create table doc(
text string
) row format delimited fields terminated by '\n' stored as textfile;

load data local inpath '/home/Words' overwrite into table doc;

SELECT word, COUNT(*) FROM doc LATERAL VIEW
explode(split(text, ' ')) wTable as word GROUP BY word;
```

Architecture of Hive

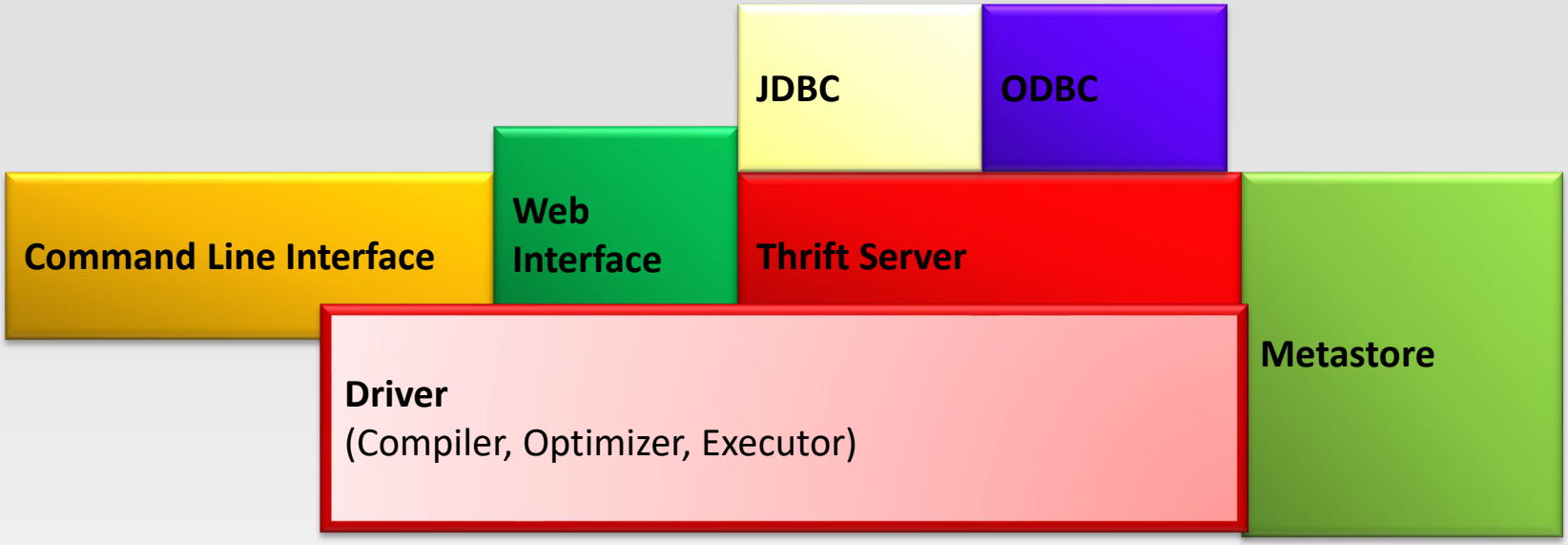


Architecture of Hive



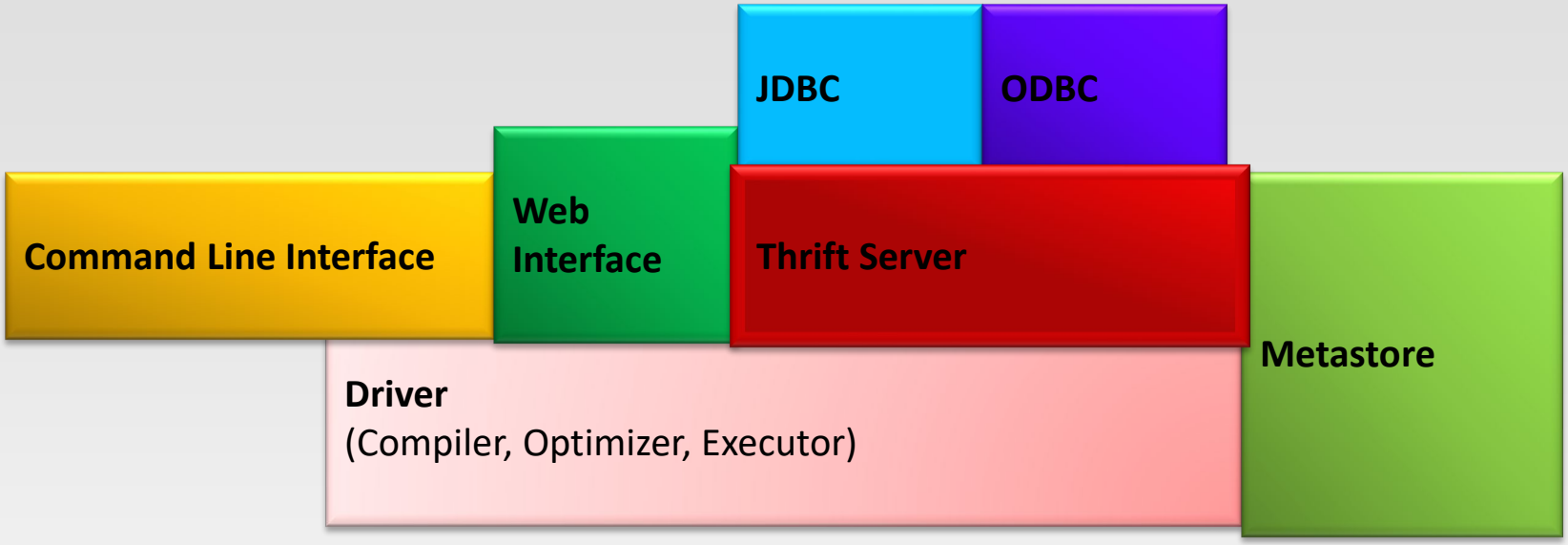
- Metastore
  - The component stores the system catalog and meta data about tables, columns, partitions etc.
  - Stored in a relational RDBMS (built-in Derby)

Architecture of Hive



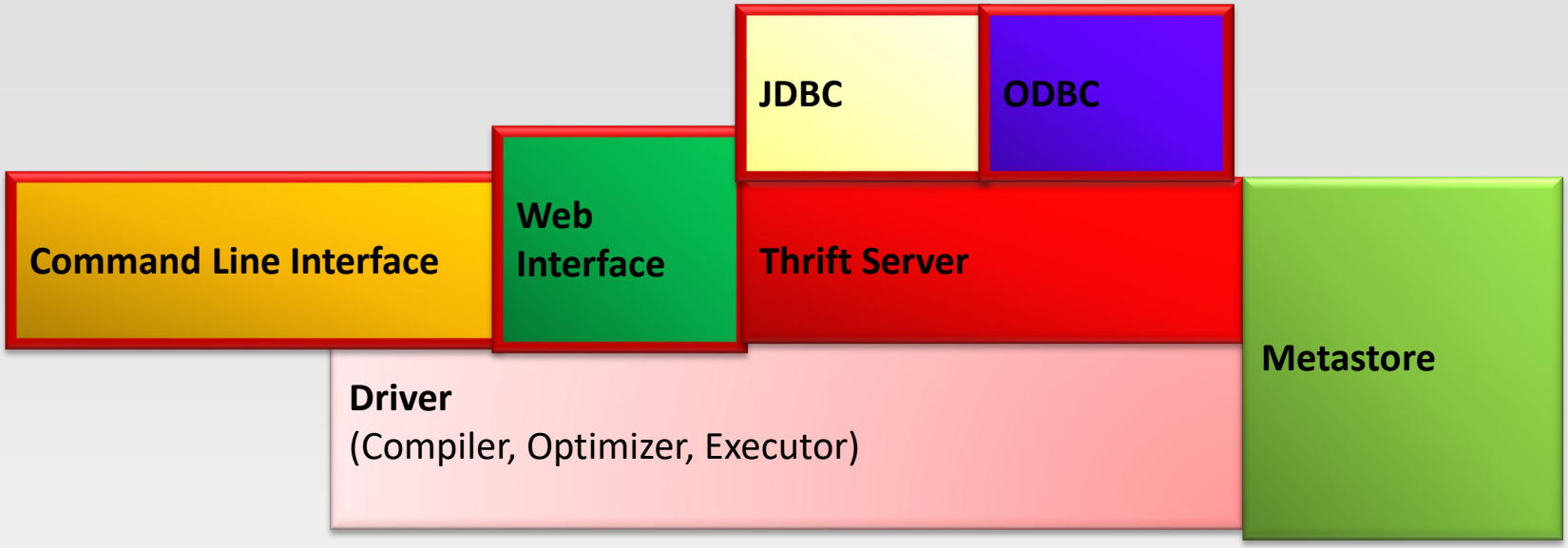
- Driver: manages the lifecycle of a HiveQL statement as it moves through Hive.
  - Query Compiler: compiles HiveQL into map/reduce tasks
  - Optimizer: generate the best execution plan
  - Execution Engine: executes the tasks produced by the compiler in proper dependency order. The execution engine interacts with the underlying Hadoop instance.

Architecture of Hive



- Thrift Server
  - Cross-language support
  - Provides a thrift interface and a JDBC/ODBC server and provides a way of integrating Hive with other applications.

Architecture of Hive



- Client Components
  - Including Command Line Interface(CLI), the web UI and JDBC/ODBC driver.

Hive Installation and Configuration

- Download at: https://hive.apache.org/downloads.html
- The latest stable version: 3.1.3
- Install:

```
$ tar xzf apache-hive-3.1.3-bin.tar.gz
$ mv apache-hive-3.1.3-bin ~/hive
```

- Environment variables in ~/.bashrc

```
export HIVE_HOME=~/hive
export PATH=$HIVE_HOME/bin:$PATH
```

- Create /tmp and /user/hive/warehouse and set them chmod g+w for more than one user usage

```
$ hdfs dfs -mkdir /tmp
$ hdfs dfs -mkdir /user/hive/warehouse
$ hdfs dfs -chmod g+w /tmp
$ hdfs dfs -chmod g+w /user/hive/warehouse
```

- Run the schematool command to initialize Hive

```
$ schematool -dbType derby -initSchema
```

- Start Hive Shell: $ hive

Hive Type System

- Primitive types
  - Integers: TINYINT, SMALLINT, INT, BIGINT.
  - Boolean: BOOLEAN.
  - Floating point numbers: FLOAT, DOUBLE.
  - Fixed point numbers: DECIMAL
  - String: STRING, CHAR, VARCHAR.
  - Date and time types: TIMESTAMP, DATE
- Complex types
  - Structs: c has type {a INT; b INT}. c.a to access the first field
  - Maps: M['group'].
  - Arrays: ['a', 'b', 'c'], A[1] returns 'b'.
- Example
  - list< map<string, struct< p1:int,p2:int > > >
  - Represents list of associative arrays that map strings to structs that contain two ints

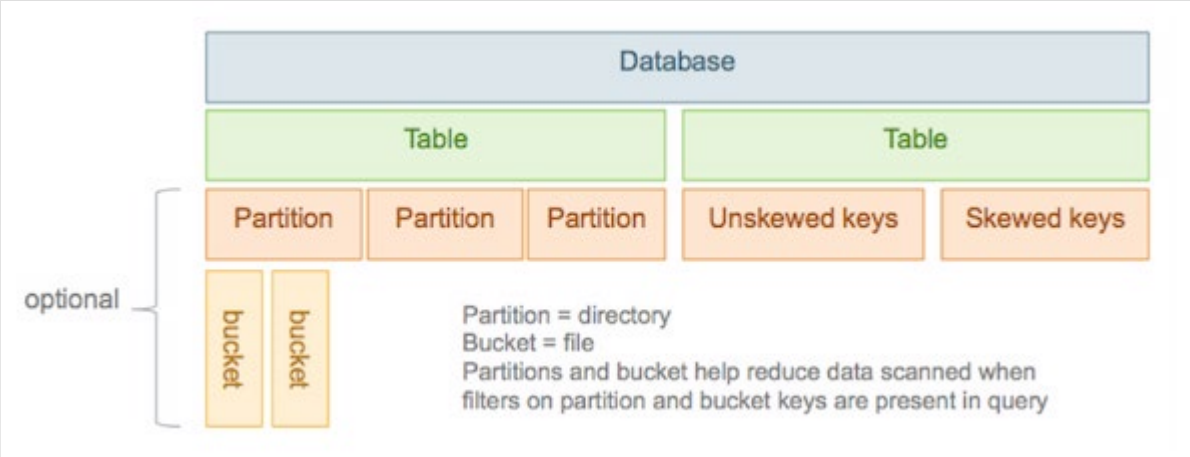
Hive Data Model

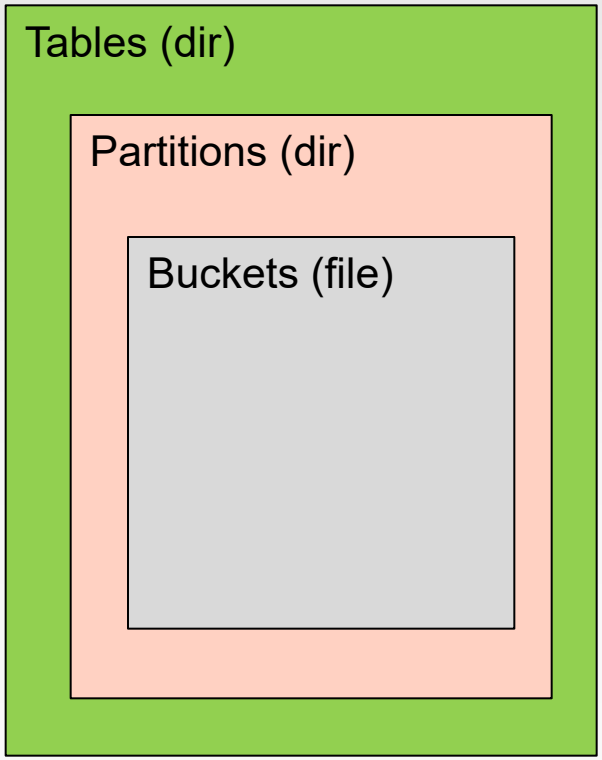
- Databases: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on.
- Tables: Homogeneous units of data which have the same schema.
  - Analogous to tables in relational DBs.
  - Each table has corresponding directory in HDFS.
  - An example table: page_views:
    - timestamp—which is of INT type that corresponds to a UNIX timestamp of when the page was viewed.
    - userid —which is of BIGINT type that identifies the user who viewed the page.
    - page_url—which is of STRING type that captures the location of the page.
    - referer_url—which is of STRING that captures the location of the page from where the user arrived at the current page.
    - IP—which is of STRING type that captures the IP address from where the page request was made.

Hive Data Model (Cont')

- Partitions:
  - Each Table can have one or more partition Keys which determines how the data is stored
  - Example:
    - Given the table page_views, we can define two partitions a date_partition of type STRING and country_partition of type STRING
    - All "US" data from "2009-12-23" is a partition of the page_views table
  - Partition columns are virtual columns, they are not part of the data itself but are derived on load
  - It is the user's job to guarantee the relationship between partition name and data content
- Buckets: 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
  - Example: the page_views table may be bucketed by userid

Data Model and Storage





/root-path
/table1
/partition1
(2011-11)
/bucket1 (1/3)
/bucket2 (2/3)
/bucket3 (3/3)
/partition2
(2011-12)
/bucket1 (1/3)
/bucket2 (2/3)
/bucket3 (3/3)
/table2
/bucket1 (1/2)
/bucket2 (2/2)

Create Table

- Syntax:

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type [COMMENT col_comment], ...)]
  [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]
   [ROW FORMAT row_format]
   [STORED AS file_format]
```

See full CREATE TABLE command at:
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL

Hive SerDe

- SerDe is a short name for "Serializer and Deserializer.“
  - Describe how to load the data from the file into a representation that make it looks like a table;
- 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
- More details see: https://cwiki.apache.org/confluence/display/Hive/DeveloperGuide#DeveloperGuide-HiveSerDe

Hive SerDe

row_format

: DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS TERMINATED BY char]

[MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]

Default values: Ctrl+A, Ctrl+B, Ctrl+C, new line, respectively

file_format:

: SEQUENCEFILE

| TEXTFILE -- (Default, depending on hive.default.fileformat configuration)

| RCFILE -- (Note: Available in Hive 0.6.0 and later)

| ORC -- (Note: Available in Hive 0.11.0 and later)

| PARQUET -- (Note: Available in Hive 0.13.0 and later)

| AVRO -- (Note: Available in Hive 0.14.0 and later)

| INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

Create Table Example

- Example:

```
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'
   LINES TERMINATED BY '\n'
 STORED AS TEXTFILE;
```

Browsing Tables and Partitions

- To list existing tables in the warehouse
  - SHOW TABLES;
- To list tables with prefix 'page'
  - SHOW TABLES 'page.*';
- To list partitions of a table
  - SHOW PARTITIONS page_view;
- To list columns and column types of table.
  - DESCRIBE page_view;

Alter Table/Partition/Column

- To rename existing table to a new name
  - ALTER TABLE old_table_name RENAME TO new_table_name;
- To rename the columns of an existing table
  - ALTER TABLE old_table_name REPLACE COLUMNS (col1 TYPE, ...);
- To add columns to an existing table
  - ALTER TABLE tab1 ADD COLUMNS (c1 INT COMMENT 'a new int column', c2 STRING DEFAULT 'def val');
- To rename a partition
  - ALTER TABLE table_name PARTITION old_partition_spec RENAME TO PARTITION new_partition_spec;
- To rename a column
  - ALTER TABLE table_name CHANGE old_col_name new_col_name column_type
- More details see: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-AlterTable

Drop Table/Partition

- To drop a table
  - DROP TABLE [IF EXISTS] table_name
  - Example:
    - DROP TABLE page_view
- To drop a paritition
  - ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec[, PARTITION partition_spec, ...]
  - Example:
    - ALTER TABLE pv_users DROP PARTITION (ds='2008-08-08')

Loading Data

- 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.
- Syntax:

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

  - Load data from a file in the local files system
    - LOAD DATA LOCAL INPATH /tmp/pv_2008-06-08_us.txt INTO TABLE page_view PARTITION(date='2008-06-08', country='US')
  - Load data from a file in HDFS
    - LOAD DATA INPATH '/user/data/pv_2008-06-08_us.txt' INTO TABLE page_view PARTITION(date='2008-06-08', country='US')
  - The input data format must be the same as the table format!

Insert Data

- Insert rows into a table:
  - Syntax

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

- Inserting data into Hive Tables from queries
  - Syntax

```
INSERT INTO TABLE tablename [PARTITION (partcol1=val1,
partcol2=val2 ...)] select_statement FROM from_statement;
```

  - Example:

```
INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active = 1;
```

Update Data

- Syntax:

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

- Synopsis
  - The referenced column must be a column of the table being updated.
  - The value assigned must be an expression that Hive supports in the select clause. Thus arithmetic operators, UDFs, casts, literals, etc. are supported. Subqueries are not supported.
  - Only rows that match the WHERE clause will be updated.
  - Partitioning columns cannot be updated.
  - Bucketing columns cannot be updated.

Query Data

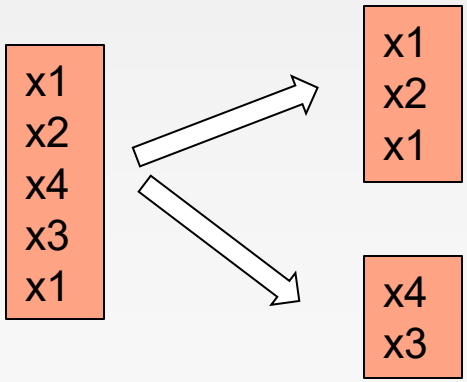
- Select Syntax:

```
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 number]
```

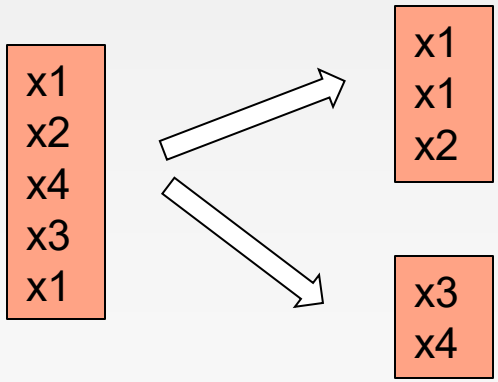
Order, Sort, Cluster, and Distribute By

- Difference between *Order By* and *Sort By*
  - The former guarantees total order in the output while the latter only guarantees ordering of the rows within a reducer

- Cluster By
  - *Cluster By* is a short-cut for both *Distribute By* and *Sort 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. However, *Distribute By* does not guarantee clustering or sorting properties on the distributed keys.



Distribute By



Cluster By

Query Examples

- Selects column 'foo' from all rows of partition ds=2008-08-15 of the invites table. The results are not stored anywhere, but are displayed on the console.

```
hive> SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';
```

- Selects all rows from partition ds=2008-08-15 of the invites table into an HDFS directory.

```
hive> INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT
a.* FROM invites a WHERE a.ds='2008-08-15';
```

- Selects all rows from pokes table into a local directory.

```
hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out'
SELECT a.* FROM pokes a;
```

Group By

- Count the number of distinct users by gender

```
INSERT OVERWRITE TABLE pv_gender_sum
SELECT pv_users.gender, count (DISTINCT pv_users.userid)
FROM pv_users
GROUP BY pv_users.gender;
```

- Multiple DISTINCT expressions in the same query is not allowed

```
INSERT OVERWRITE TABLE pv_gender_agg
SELECT pv_users.gender, count(DISTINCT pv_users.userid),
count(DISTINCT pv_users.ip)
FROM pv_users
GROUP BY pv_users.gender;
```

Joins

- Hive does not support join conditions that are not equality conditions
  - it is very difficult to express such conditions as a map/reduce job
  - SELECT a.* FROM a JOIN b ON (a.id = b.id)
  - However, the following statement is not allowed:
    - SELECT a.* FROM a JOIN b ON (a.id <> b.id)
- More than 2 tables can be joined in the same query.
  - SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
- Example:

```
SELECT s.word, s.freq, k.freq
FROM shakespeare s
  JOIN bible k ON (s.word =
k.word) WHERE s.freq >= 1
AND k.freq >= 1 ORDER BY
s.freq DESC LIMIT 10;
```

the	25848	62394
I	23031	8854
and	19671	38985
to	18038	13526
of	16700	34654
a	14170	8057
you	12702	2720
my	11297	4135
in	10797	12445
is	8882	6884

Behind the Scenes

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
  JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;
```

(Abstract Syntax Tree)

```
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF shakespeare s) (TOK_TABREF bible k) (= (. (TOK_TABLE_OR_COL s)
word) (. (TOK_TABLE_OR_COL k) word)))) (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) (TOK_SELECT
(TOK_SELEXPR (. (TOK_TABLE_OR_COL s) word)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) freq)) (TOK_SELEXPR (.
(TOK_TABLE_OR_COL k) freq))) (TOK_WHERE (AND (>= (. (TOK_TABLE_OR_COL s) freq) 1) (>= (. (TOK_TABLE_OR_COL k)
freq) 1))) (TOK_ORDERBY (TOK_TABSORTCOLNAMEDESC (. (TOK_TABLE_OR_COL s) freq))) (TOK_LIMIT 10)))
```

(one or more of MapReduce jobs)

Hive Operators and User-Defined Functions (UDFs)

- Built-in operators:
  - relational, arithmetic, logical, etc.
- Built-in functions:
  - mathematical, date function, string function, etc.
- Built-in aggregate functions:
  - max, min, count, etc.
- Built-in table-generating functions: transform a single input row to multiple output rows
  - explode(ARRAY): Returns one row for each element from the array.
  - explode(MAP): Returns one row for each key-value pair from the input map with two columns in each row
- Create Custom UDFs
- More details see: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-explode

WordCount in Hive

- Create a table in Hive

```
create table doc(
            text string
) row format delimited fields terminated by '\n' stored as textfile;
```

- Load file into table

```
load data local inpath '/home/Words' overwrite into table doc;
```

- Compute word count using select

```
SELECT word, COUNT(*) FROM doc LATERAL VIEW
explode(split(text, ' ')) wTable as word GROUP BY word;
```

  - explode() takes in an array (or a map) as an input and outputs the elements of the array (map) as separate rows.
  - Lateral view is used in conjunction with user-defined table generating functions such as explode()
  - A lateral view first applies the UDTF to each row of base table and then joins resulting output rows to form a virtual table

explode() Function

- explode() takes in an array (or a map) as an input and outputs the elements of the array (map) as separate rows.
- The following will return a table of words in doc, with a single column word

```
SELECT explode(split(text, ' ')) AS word FROM doc
```

- The following will compute the frequency of each word

```
SELECT word, COUNT(*)
FROM (SELECT explode(SPLIT(text, ' ')) AS word FROM doc) AS
words GROUP BY word;
```

Lateral View

- Lateral view is used in conjunction with user-defined table generating functions such as explode()
- A lateral view first applies the UDTF (User Defined Tabular Function) to each row of base table and then joins resulting output rows to form a virtual table.
- Lateral View Syntax
  - lateralView: LATERAL VIEW udtf(expression) tableAlias AS columnAlias (',' columnAlias)*
  - fromClause: FROM baseTable (lateralView)*
- Compare the two ways:

```
SELECT word, COUNT(*) FROM
(SELECT explode(SPLIT(text, ' ')) AS word FROM doc) AS words
GROUP BY word;
```

```
SELECT word, COUNT(*) FROM
doc LATERAL VIEW explode(split(text, ' ')) wTable as word
GROUP BY word;
```

Writing HIVE Scripts

- Rather than executing HQL statements one-by-one in a Hive shell, you can bundle them into a script and execute them all together. This is also a good way to save your statements, edit them, and run them easily whenever you like.

```
CREATE TABLE doc (
    line STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\n'
STORED AS textfile;

LOAD DATA LOCAL INPATH 'text.txt' OVERWRITE INTO TABLE
doc;

SELECT doc, COUNT(*)
FROM doc LATERAL VIEW EXPLODE(SPLIT(line, ' ')) words as
word GROUP BY word;
```

- To execute the statements in the file, just enter the following command in the terminal: **hive -f frequency.hql**

Example: Rank the Terms by Frequencies

- We use the ABC news dataset. The task is to use Hive to compute the frequency for each term within each year, and then sort the result by the year first, and then by the frequency in descending order. For the terms with the same frequency, rank them in alphabetical order.

```
20030219,council chief executive fails to secure position
20030219,council welcomes ambulance levy decision
20030219,council welcomes insurance breakthrough
20030219,fed opp to re introduce national insurance
20040501,cowboys survive eels comeback
20040501,cowboys withstand eels fightback
20040502,castro vows cuban socialism to survive bush
20200401,coronanomics things learnt about how coronavirus economy
20200401,coronavirus at home test kits selling in the chinese community
20200401,coronavirus campbell remess streams bear making classes
20201015,coronavirus pacific economy foriegn aid china
20201016,china builds pig apartment blocks to guard against swine flu
```

Solution

```
CREATE TABLE abcnews (
    NewsDate STRING,
    Headline STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH 'abcnews.txt' OVERWRITE INTO TABLE abcnews;

CREATE VIEW year_terms AS SELECT substr(NewsDate, 0, 4) as year, term FROM
abcnews LATERAL VIEW EXPLODE (SPLIT(Headline, ' ')) terms AS term;

SELECT year,term,count(*) AS num FROM year_terms GROUP BY year,term;

CREATE VIEW termcount AS SELECT year,term,count(*) AS num FROM year_terms
GROUP BY year,term;

SELECT a.year, a.term FROM (select *, rank() over (ORDER BY year, num DESC, term)
FROM termcount) as a;
```

Pros/Cons

- Pros
  - An easy way to process large scale data
  - Support SQL-based queries
  - Provide more user defined interfaces to extend
  - Programmability
  - Efficient execution plans for performance
  - Interoperability with other databases
- Cons
  - No easy way to append data
  - Files in HDFS are immutable

Applications of Hive

- Log processing
  - Daily Report
  - User Activity Measurement
- Data/Text mining
  - Machine learning (Training Data)
- Business intelligence
  - Advertising Delivery
  - Spam Detection

References

- https://cwiki.apache.org/confluence/display/Hive/Home#Home-HiveDocumentation
- http://www.tutorialspoint.com/hive/
- Hadoop the Definitive Guide. Hive Chapter

End of Chapter 9