Hive: A data warehouse on Hadoop

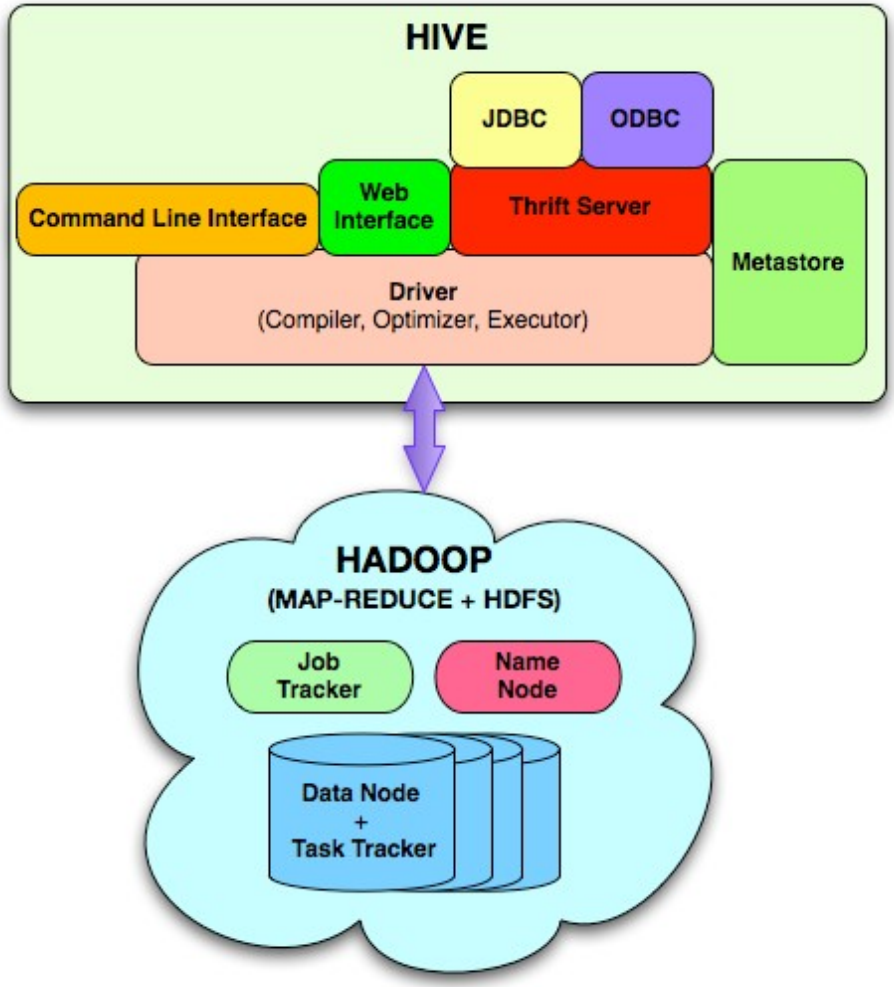
Motivation

- Yahoo worked on Pig to facilitate application deployment on Hadoop.
  - Their need mainly was focused on unstructured data
- Simultaneously Facebook started working on deploying warehouse solutions on Hadoop that resulted in Hive.
  - The size of data being collected and analyzed in industry for business intelligence (BI) is growing rapidly making traditional warehousing solution prohibitively expensive.

Hadoop MR

- MR is very low level and requires customers to write custom programs.
- HIVE supports queries expressed in SQL-like language called HiveQL which are compiled into MR jobs that are executed on Hadoop.
- Hive also allows MR scripts
- It also includes MetaStore that contains schemas and statistics that are useful for data explorations, query optimization and query compilation.
- At Facebook Hive warehouse contains tens of thousands of tables, stores over 700TB and is used for reporting and ad-hoc analyses by 200 Fb users.

Hive architecture (from the paper)



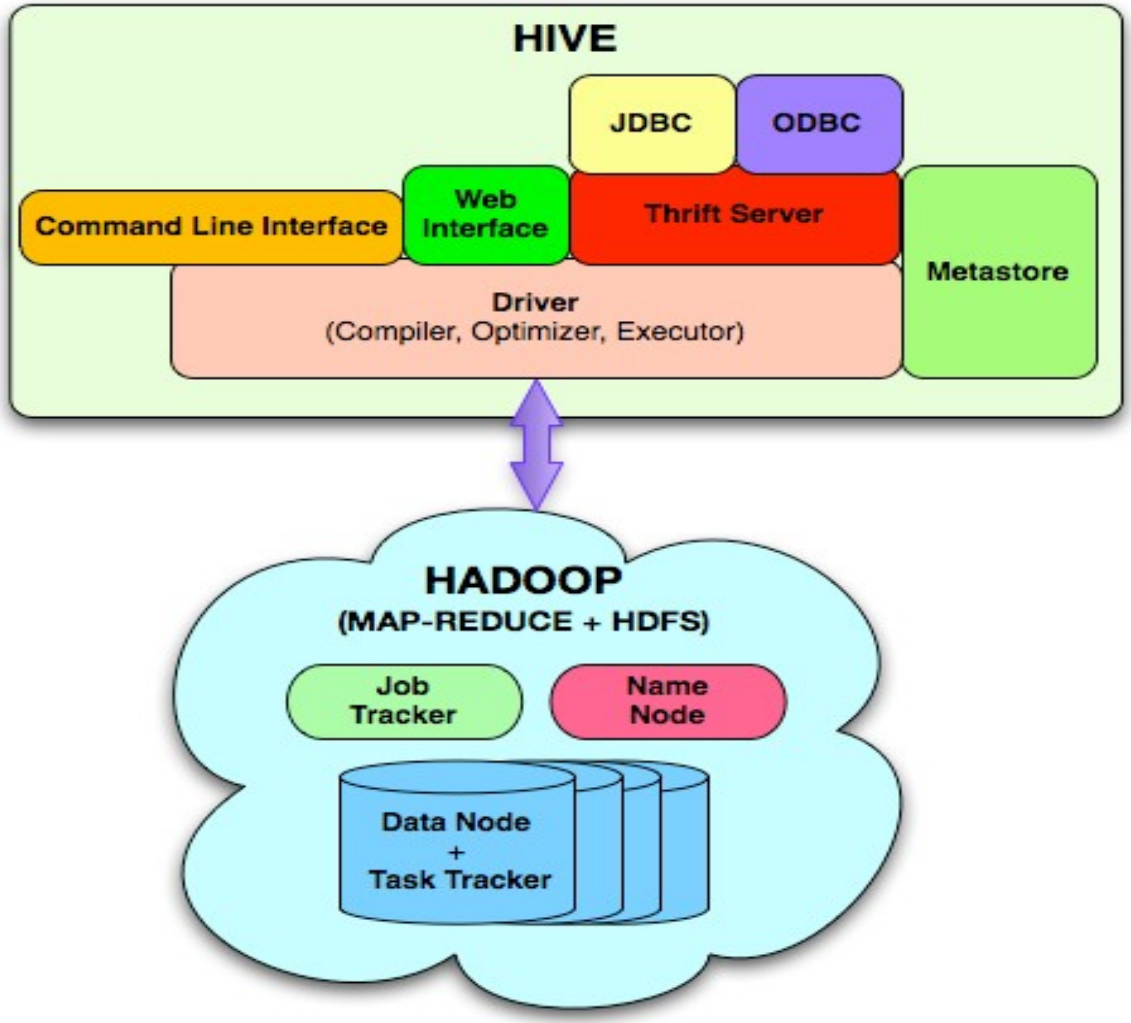
Query Language (HiveQL)

- Subset of SQL
- Meta-data queries
- Limited equality and join predicates
- No inserts on existing tables (to preserve worm property)
  - Can overwrite an entire table

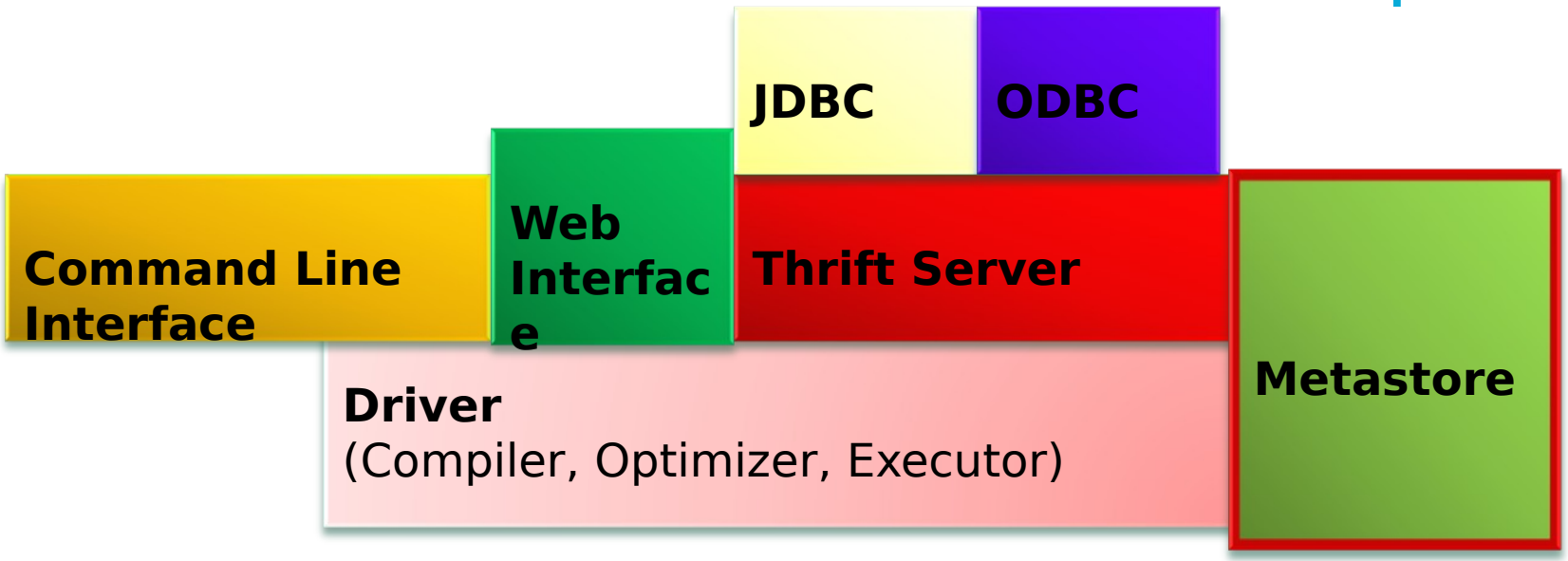
Data Storage

- Tables are logical data units; table metadata associates the data in the table to hdfs directories.
- Hdfs namespace: tables (hdfs directory), partition (hdfs subdirectory), buckets (subdirectories within partition)
- /user/hive/warehouse/test_table is a hdfs directory

System Architecture and Components



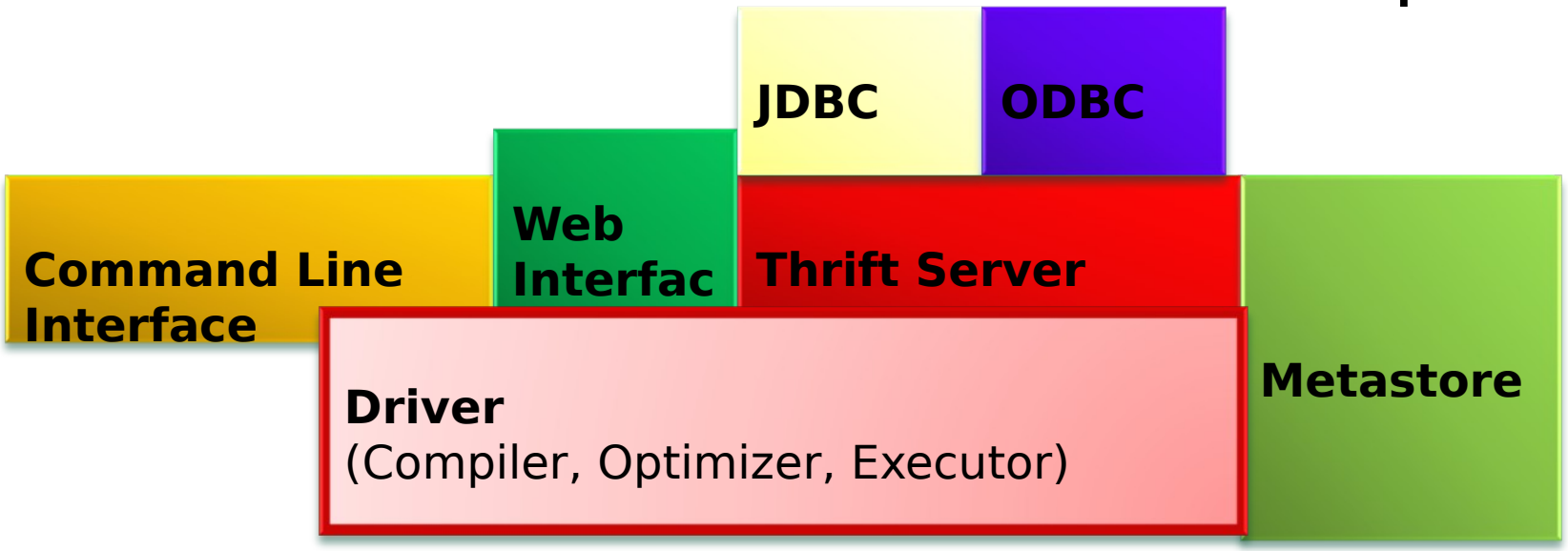
System Architecture and Components



Metastore

- The component that store the system catalog and meta data about tables, columns, partitions etc.
- Stored on a traditional RDBMS

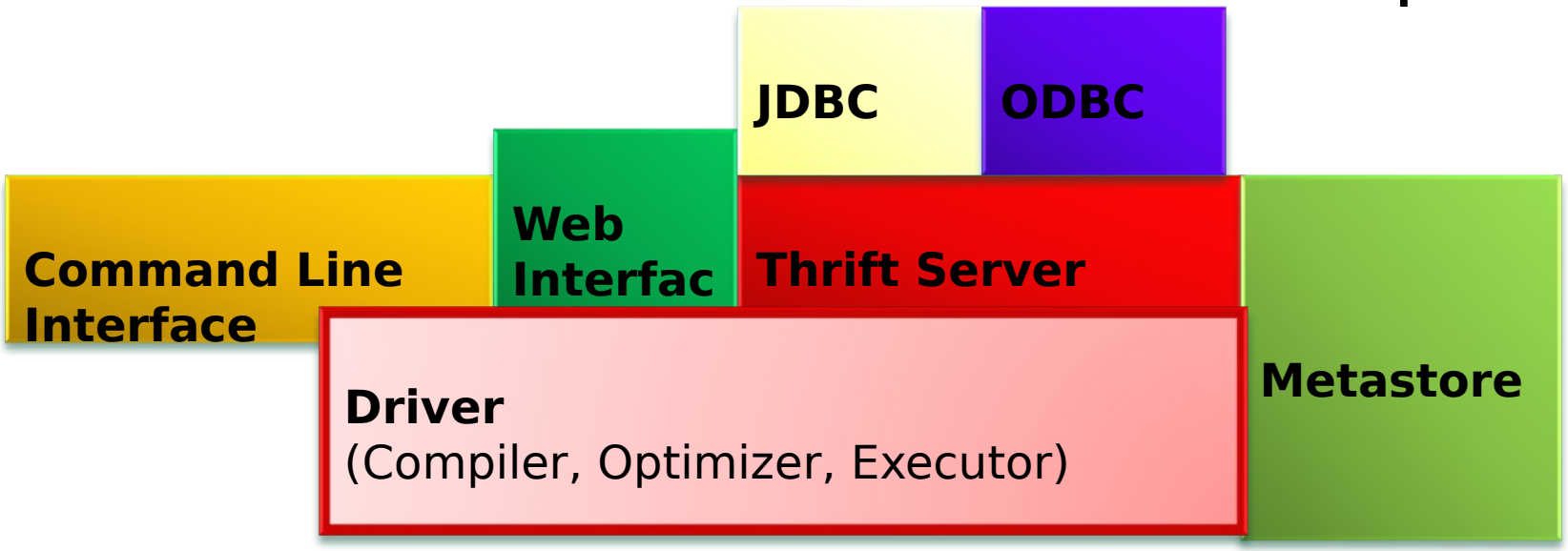
System Architecture and Components



- Driver

  The component that manages the lifecycle of a HiveQL statement as it moves through Hive. The driver also maintains a session handle and any session statistics.

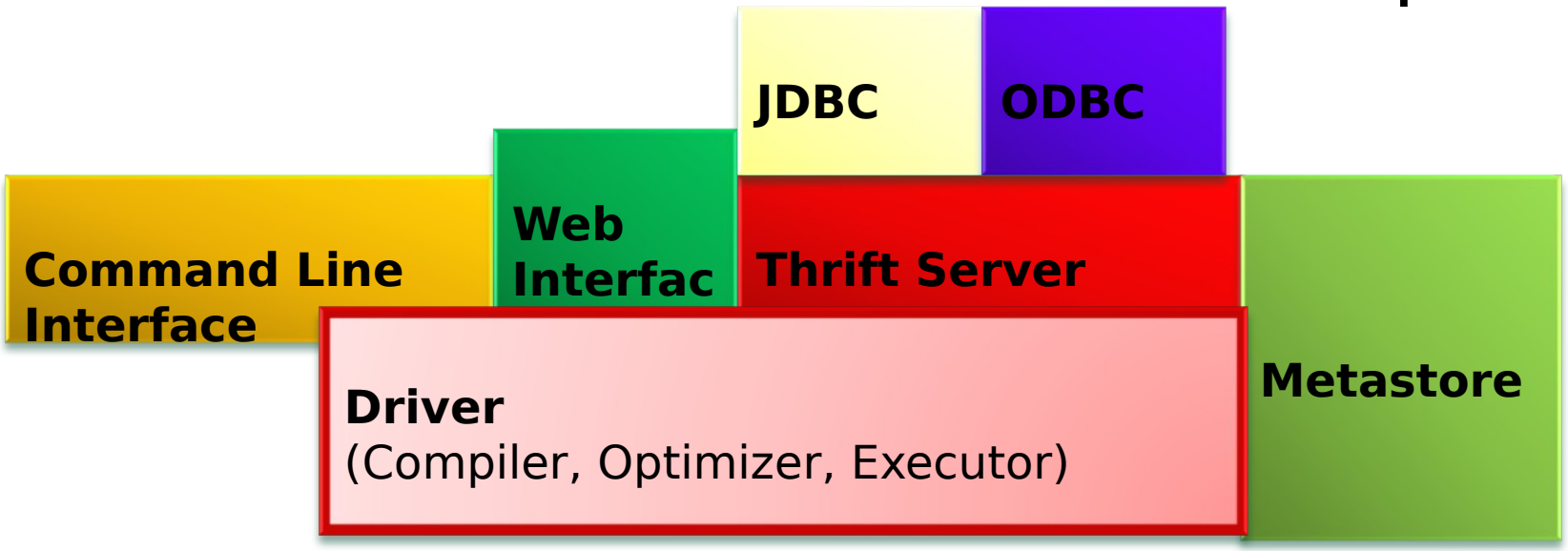
System Architecture and Components



- Query Compiler

  The component that compiles HiveQL into a directed acyclic graph of map/reduce tasks.

System Architecture and Components

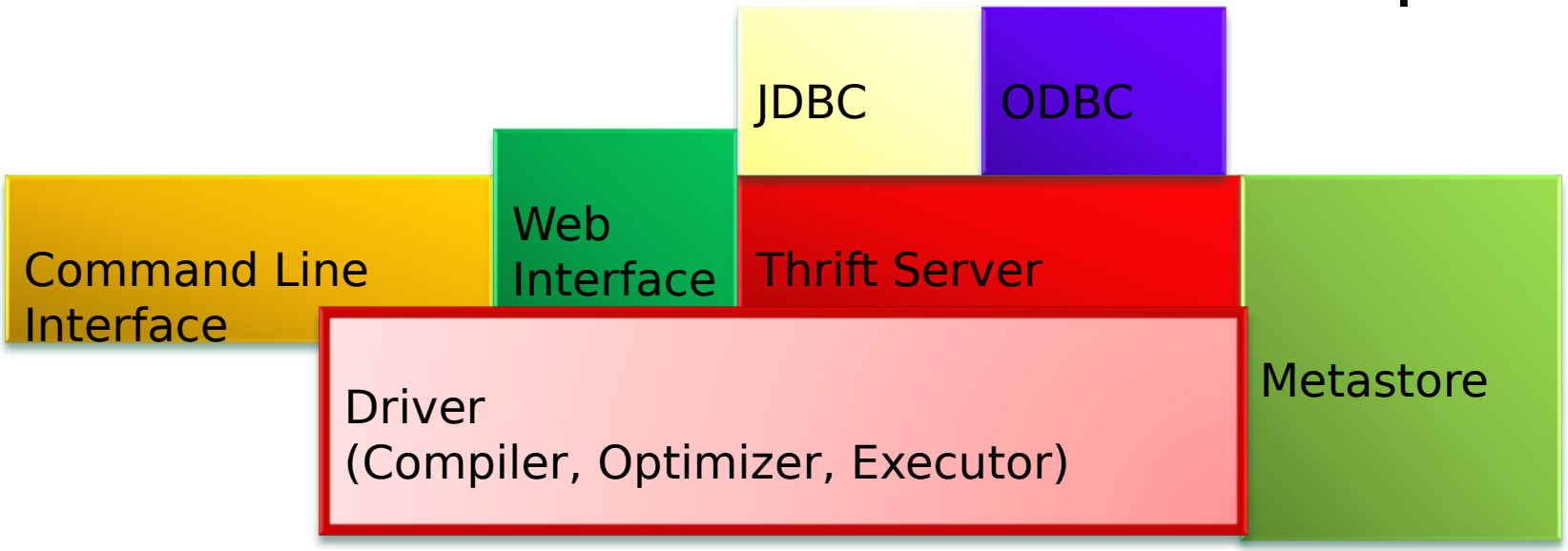


- Optimizer

consists of a chain of transformations such that the operator DAG resulting from one transformation is passed as input to the next transformation

Performs tasks like Column Pruning , Partition Pruning, Repartitioning of Data

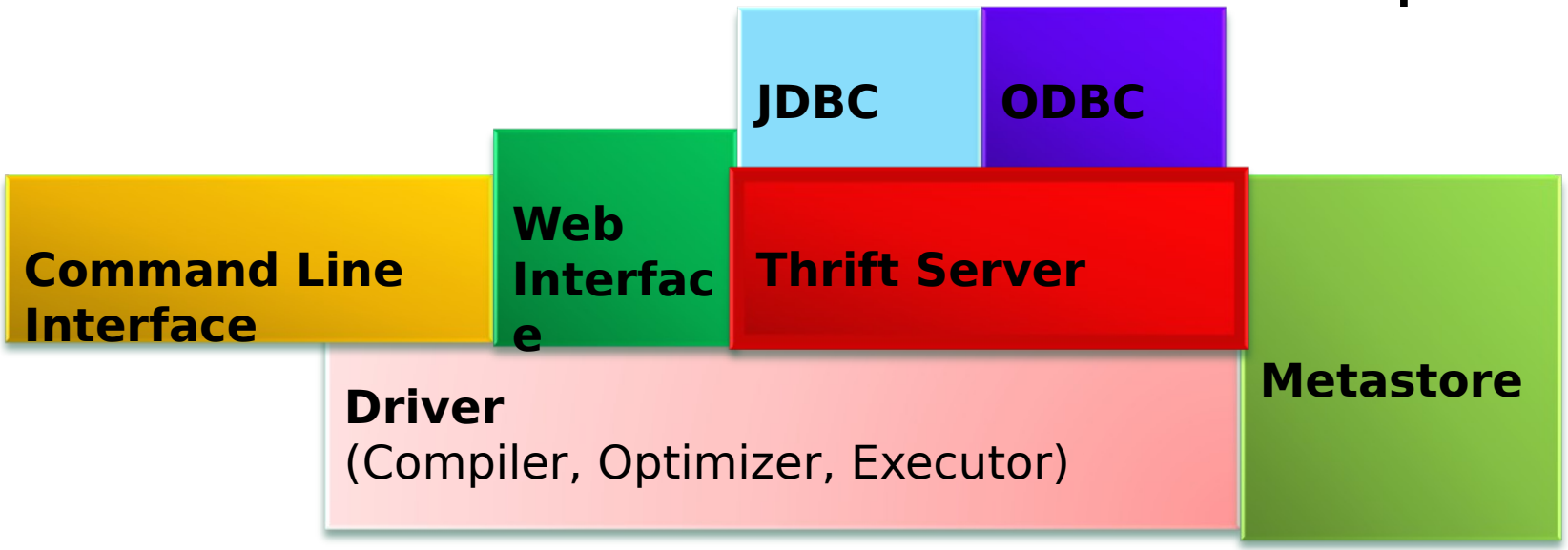
System Architecture and Components



- Execution Engine

  The component that executes the tasks produced by the compiler in proper dependency order. The execution engine interacts with the underlying Hadoop instance.

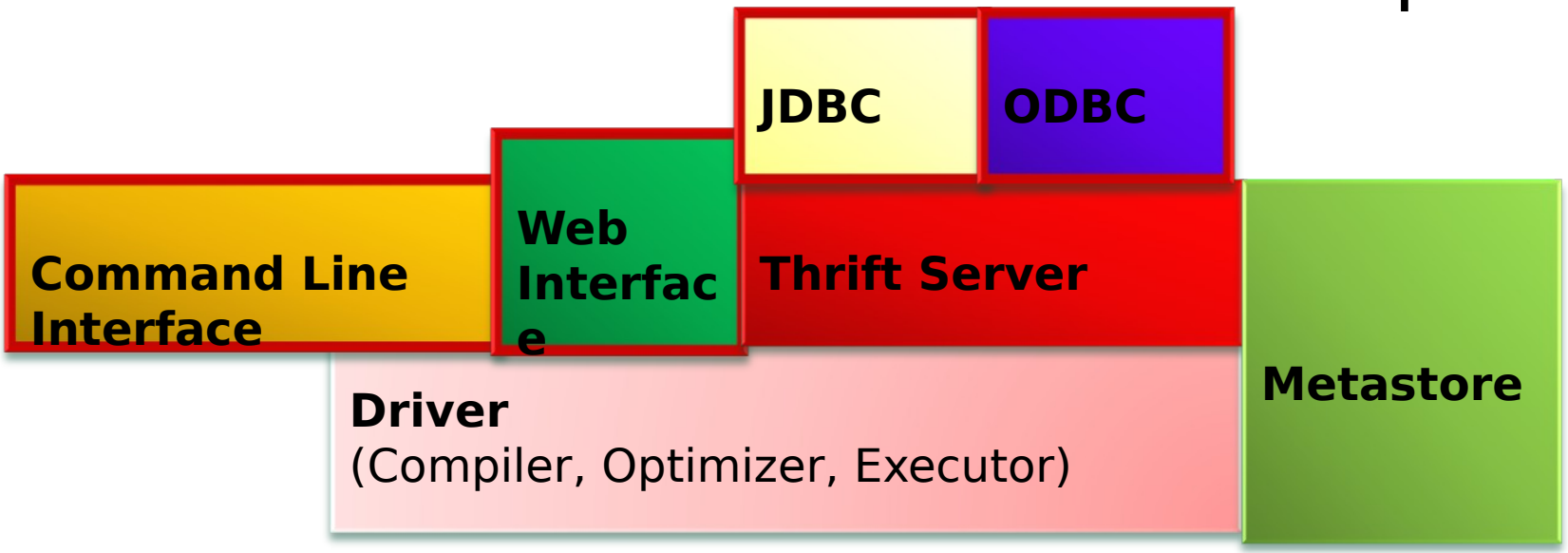
System Architecture and Components



- HiveServer

  The component that provides a trift interface and a JDBC/ODBC server and provides a way of integrating Hive with other applications.

System Architecture and Components



- Client Components

  Client component like Command Line Interface(CLI), the web UI and JDBC/ODBC driver.

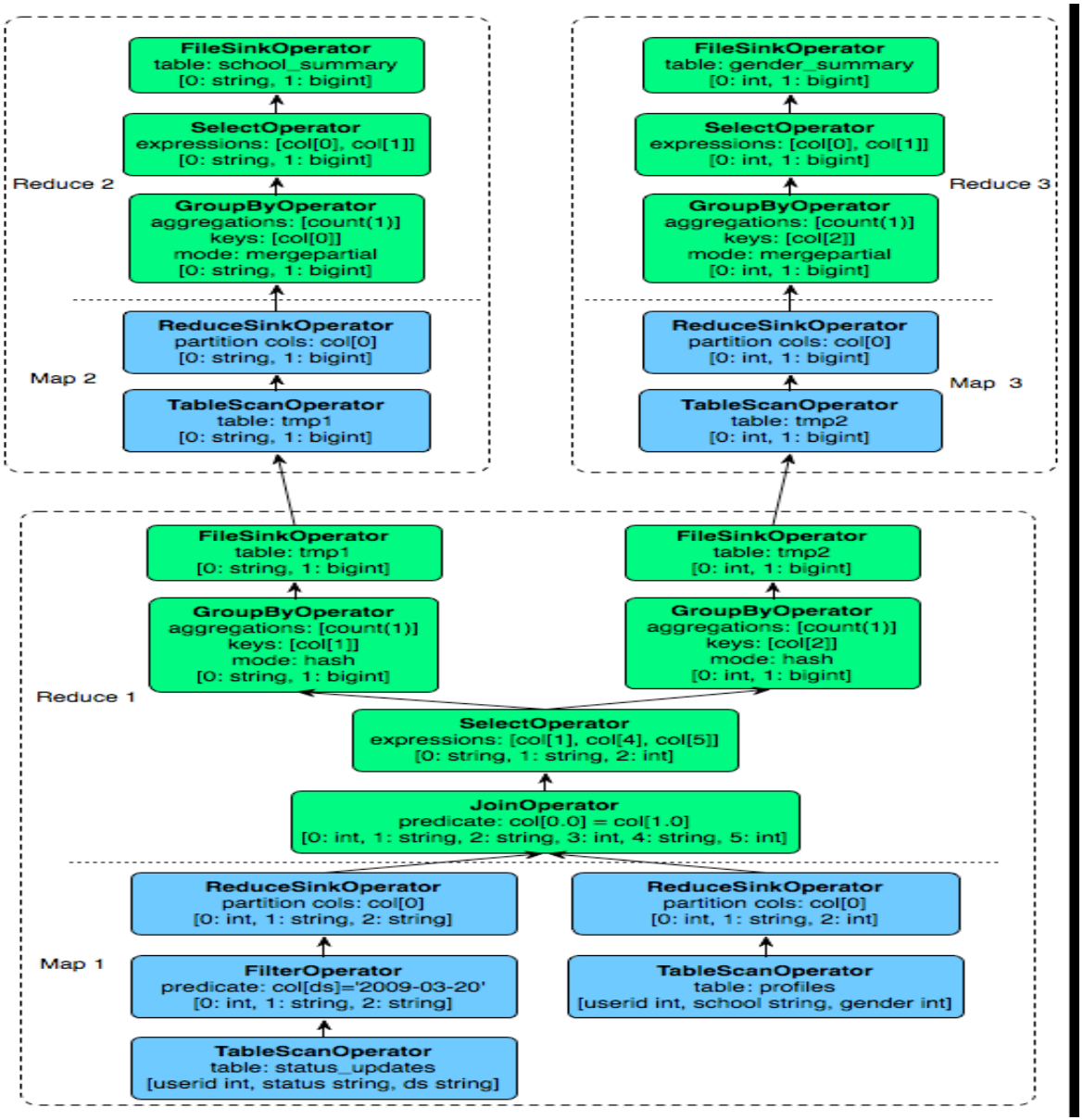
Hive Query Language

- Basic SQL
  - From clause sub-query
  - ANSI JOIN (equi-join only)
  - Multi-Table insert
  - Multi group-by
  - Sampling
  - Objects Traversal
- Extensibility
  - Pluggable Map-reduce scripts using TRANSFORM

Architecture

- Metastore: stores system catalog
- Driver: manages life cycle of HiveQL query as it moves thru' HIVE; also manages session handle and session statistics
- Query compiler: Compiles HiveQL into a directed acyclic graph of map/reduce tasks
- Execution engines: The component executes the tasks in proper dependency order; interacts with Hadoop
- HiveServer: provides Thrift interface and JDBC/ODBC for integrating other applications.
- Client components: CLI, web interface, jdbc/odbc inteface
- Extensibility interface include SerDe, User Defined Functions and User Defined Aggregate Function

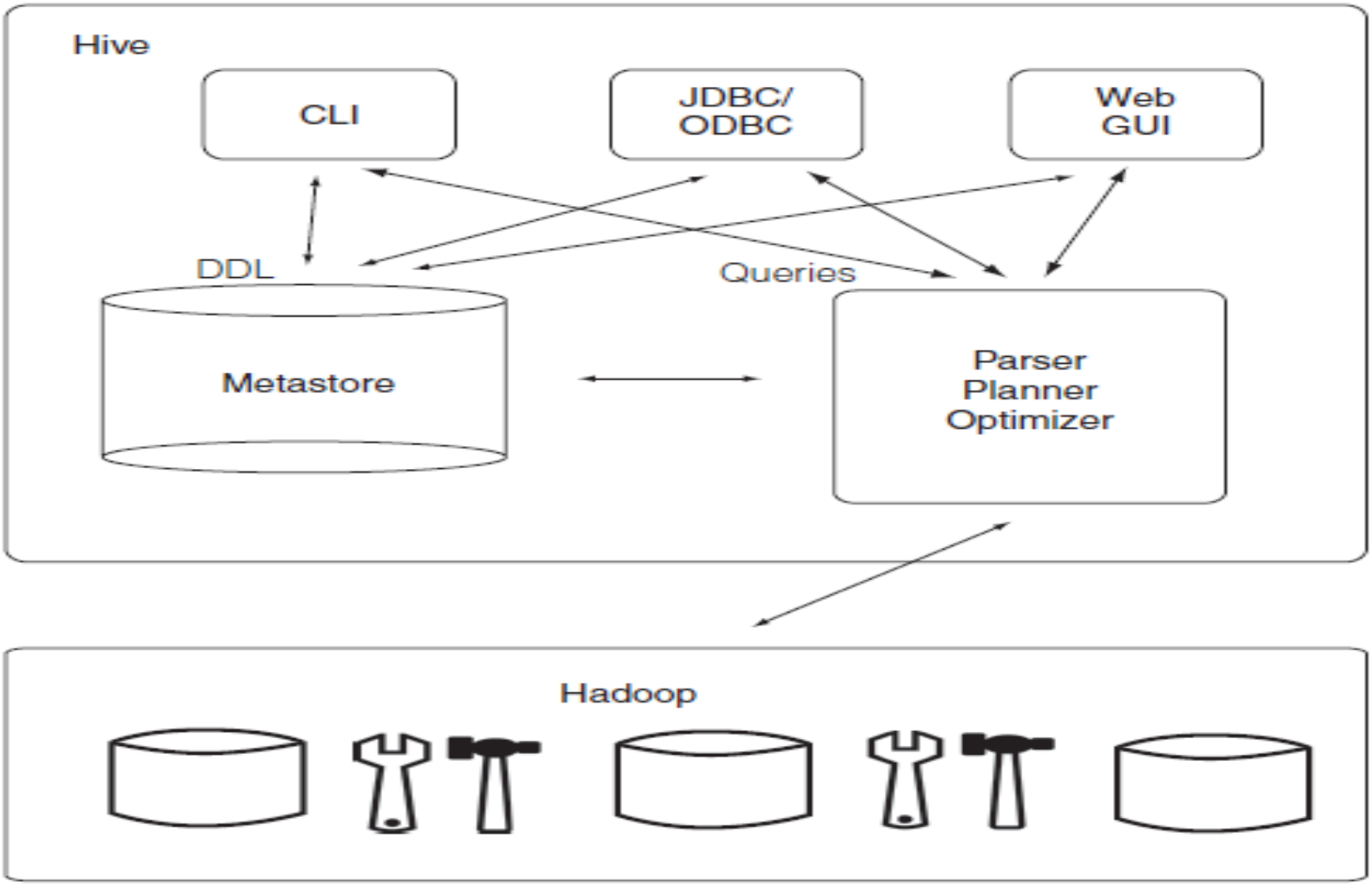
Sample Query Plan



Hive Usage in Facebook

- Hive and Hadoop are extensively used in Facbook for different kinds of operations.
- 700 TB = 2.1Petabyte after replication!
- Think of other application model that can leverage Hadoop MR.

Hive Operations



Hive Operations:tables

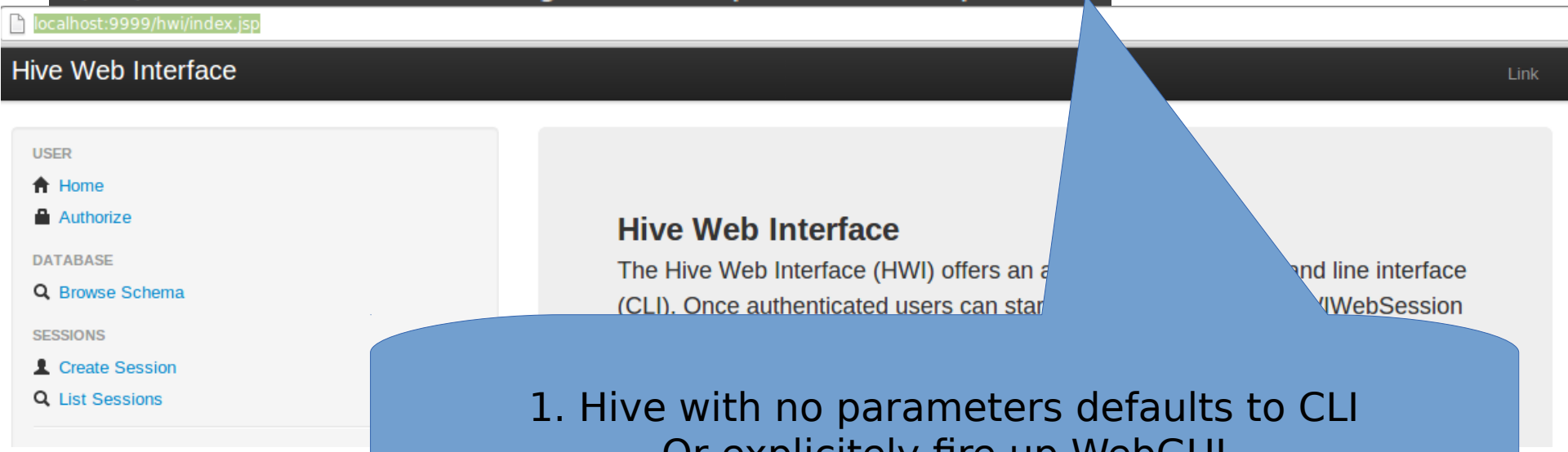
```
hadoop fs -mkdir /tmp
hadoop fs -mkdir /user/hive/warehouse
hadoop fs -chmod g+w /tmp
hadoop fs -chmod g+w /user/hive/warehouse
```

1. Hadoop needs to be up and running already. In addition, you need to set up a couple directories in HDFS for Hive to use.
2. It's good to let Hive manage your data if you plan on using Hive to query it. But if you already have your data in HDFS Hive can work with them too.
3. Hive will take your data as is and won't optimize.
4. Hive does NOT requires data to be in some special Hive format.

Hive Operations:WebGUI

```
<property>
    <name>hive.hwi.war.file</name>
    <value>lib/hive-hwi-0.12.0.2.0.6.0-76.war</value>
    <description>This sets the path to the HWI war file, relative to ${HIVE_HOME}. </description>
</property>
```

```
mohit@NoMind ~/Work/BigData $ hive --service hwi
14/01/13 13:35:18 INFO hwi.HWIServer: HWI is starting up
14/01/13 13:35:18 INFO Configuration.deprecation: mapred.inpu
```



1. Hive with no parameters defaults to CLI
Or explicitely fire up WebGUI

Hive Operations

Metastore is a traditional DB and can be configured with hive-site.xml.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"
<configuration>
    <property>
        <name>hive.metastore.local</name>
        <value>false</value>
        <description>JDBC connect string for a JDBC metastore</description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>jdbc:derby:$HIVE_HOME/../resources/hive_data;databaseName=metastore_db;create=true</value>
        <description>JDBC connect string for a JDBC metastore</description>
    </property>

    <property>
      <name>javax.jdo.option.ConnectionDriverName</name>
      <value>org.apache.derby.jdbc.EmbeddedDriver</value>
      <description>Driver class name for a JDBC metastore</description>
    </property>
```

Hive Operations:creating tables

```
hive> CREATE TABLE cite (citing INT, cited INT)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY ','
    > STORED AS TEXTFILE;
OK
Time taken: 0.246 seconds

hive> SHOW TABLES;
OK
cite
Time taken: 0.053 seconds

hive> DESCRIBE cite;
OK
citing  int
cited   int
Time taken: 0.13 seconds
```

Hive Operations:loading data

```
hive> LOAD DATA LOCAL INPATH 'cite75_99.txt'
    > OVERWRITE INTO TABLE cite;
Copying data from file:/root/cite75_99.txt
Loading data to table cite
OK
Time taken: 9.51 seconds
```

Contents of directory /user/hive/warehouse/cite

Goto : /user/hive/warehouse/cite go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
cite75_99.txt	file	251.84 MB	1	128 MB	2014-01-13 13:54	rw-r--r--	mohit	supergroup

Go back to DFS home

```
hive> SELECT * FROM cite LIMIT 10;
OK
NULL    NULL
3858241 956203
3858241 1324234
3858241 3398406
3858241 3557384
3858241 3634889
3858242 1515701
3858242 3319261
3858242 3668705
3858242 3707004
Time taken: 0.17 seconds
```

Hive Operations:querying data

```
hive> SELECT COUNT(1) FROM cite;
Total MapReduce jobs = 1
Number of reduce tasks determined at compile time: 1
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>
Starting Job = job_200908250716_0001, Tracking URL = http://ip-10-244-199-
143.ec2.internal:50030/jobdetails.jsp?jobid=job_200908250716_0001
Kill Command = /usr/lib/hadoop/bin/hadoop job  -Dmapred.job.tracker=ip-10-
244-199-143.ec2.internal:9001 -kill job_200908250716_0001
 map = 0%,  reduce =0%
 map = 12%,  reduce =0%
 map = 25%,  reduce =0%
 map = 30%,  reduce =0%
 map = 34%,  reduce =0%
 map = 43%,  reduce =0%
 map = 53%,  reduce =0%
 map = 62%,  reduce =0%
 map = 71%,  reduce =0%
 map = 75%,  reduce =0%
 map = 79%,  reduce =0%
 map = 88%,  reduce =0%
 map = 97%,  reduce =0%
 map = 99%,  reduce =0%
 map = 100%,  reduce =0%
 map = 100%,  reduce =67%
 map = 100%,  reduce =100%
Ended Job = job_200908250716_0001
OK
16522439
Time taken: 85.153 seconds
```

Hive Operations:querying data(group by)

```
CREATE TABLE records (year STRING, temperature INT, quality INT)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH 'input/ncdc/micro-tab/sample.txt'
OVERWRITE INTO TABLE records;

% ls /user/hive/warehouse/records/
sample.txt

hive> SELECT year, MAX(temperature)
    > FROM records
    > WHERE temperature != 9999
    >   AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
    > GROUP BY year;
1949    111
1950    22
```

Hive Operations:clauses

Feature	SQL	HiveQL
Updates	UPDATE, INSERT, DELETE	INSERT OVERWRITE TABLE (populates whole table or partition)
Transactions	Supported	Not supported
Indexes	Supported	Not supported
Latency	Sub-second	Minutes
Data types	Integral, floating point, fixed point, text and binary strings, temporal	Integral, floating point, boolean, string, array, map, struct
Functions	Hundreds of built-in functions	Dozens of built-in functions
Multitable inserts	Not supported	Supported
Create table as select	Not valid SQL-92, but found in some databases	Supported
Select	SQL-92	Single table or view in the FROM clause. SORT BY for partial ordering. LIMIT to limit number of rows re-turned.

Hive Operations:clauses

Feature	SQL	HiveQL
Joins	SQL-92 or variants (join tables in the FROM clause, join condition in the WHERE clause)	Inner joins, outer joins, semi joins, map joins. SQL-92 syntax, with hinting.
Subqueries	In any clause. Correlated or noncorrelated.	Only in the FROM clause. Correlated subqueries not supported
Views	Updatable. Materialized or nonmaterialized.	Read-only. Materialized views not supported
Extension points	User-defined functions. Stored procedures.	User-defined functions. MapReduce scripts.

Hive Operations:types

Category	Type	Description	Literal examples
Primitive	TINYINT	1-byte (8-bit) signed integer, from -128 to 127	1
	SMALLINT	2-byte (16-bit) signed integer, from -32,768 to 32,767	1
	INT	4-byte (32-bit) signed integer, from -2,147,483,648 to 2,147,483,647	1
	BIGINT	8-byte (64-bit) signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	1
	FLOAT	4-byte (32-bit) single-precision floating-point number	1.0
	DOUBLE	8-byte (64-bit) double-precision floating-point number	1.0
	BOOLEAN	true/false value	TRUE
	STRING	Character string	'a', "a"
	BINARY	Byte array	Not supported
	TIMESTAMP	Timestamp with nanosecond precision	1325502245000, '2012-01-02 03:04:05.123456789'

Hive Operations:types

Category	Type	Description	Literal examples
Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type.	`array(1, 2)`[a]
	MAP	An unordered collection of key-value pairs. Keys must be primitives; values may be any type. For a particular map, the keys must be the same type, and the values must be the same type.	`map('a', 1, 'b', 2)`
	STRUCT	A collection of named fields. The fields may be of different types.	`struct('a', 1, 1.0)`

Hive Operations:types(complex)

```
CREATE TABLE complex (
  col1 ARRAY<INT>,
  col2 MAP<STRING, INT>,
  col3 STRUCT<a:STRING, b:INT, c:DOUBLE>
);
```

```
hive> SELECT col1[0], col2['b'], col3.c FROM complex;
1    2    1.0
```

Hive Operations:partitions

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

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

```
/user/hive/warehouse/logs/dt=2010-01-01/country=GB/file1
                                                  /file2
                                        /country=US/file3
                              /dt=2010-01-02/country=GB/file4
                                        /country=US/file5
                                                  /file6
```

Hive Operations: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
```

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

Hive Operations:buckets

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

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

```
hive> SELECT * FROM users;
0     Nat
2     Joe
3     Kay
4     Ann
```

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

1. Bucketing imposes extra Structure on the table, Hive can exploit buckets when joining of two tables. can be efficiently implemented as a map-side join.
2. Buckets help in sampling for queries run Large data set.

Hive Operations:buckets

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

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

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

Hive Operations:format

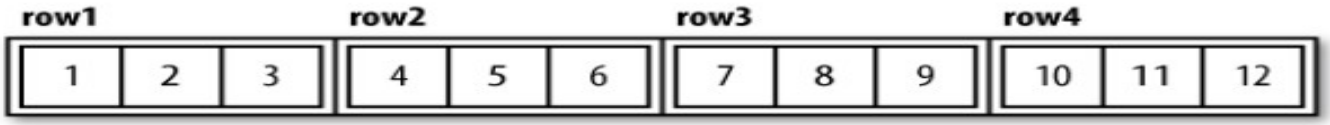
```
DROP TABLE IF EXISTS compressed_users;
CREATE TABLE compressed_users (id INT, name STRING) STORED AS SEQUENCEFILE;
SET hive.exec.compress.output=true;
SET mapred.output.compress=true;
SET mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;
INSERT OVERWRITE TABLE compressed_users SELECT * FROM users;
```

Hive Operations:format(RC)

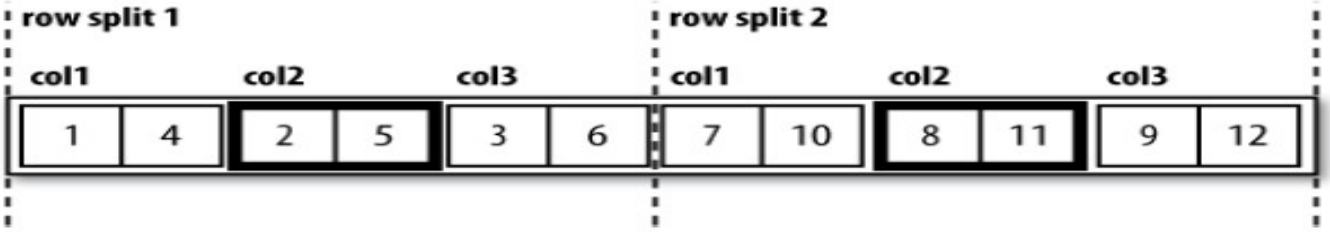
Logical table

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9
row4	10	11	12

Row-oriented layout (SequenceFile)



Column-oriented layout (RCFile)



Hive Operations:joins

```
hive> SELECT * FROM sales;
Joe     2
Hank    4
Ali     0
Eve     3
Hank    2
hive> SELECT * FROM things;
2       Tie
4       Coat
3       Hat
1       Scarf
```

Hive Operations:joins

```
hive> SELECT sales.*, things.*
    > FROM sales JOIN things ON (sales.id = things.id);
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

```
hive> SELECT sales.*, things.*
    > FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
Ali     0    NULL NULL
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Hive Operations:joins

```
hive> SELECT sales.*, things.*
    > FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);
NULL    NULL 1    Scarf
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

```
hive> SELECT sales.*, things.*
    > FROM sales FULL OUTER JOIN things ON (sales.id = things.id);
Ali     0    NULL NULL
NULL    NULL 1    Scarf
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Hive Operations:joins

```
SELECT *
FROM things
WHERE things.id IN (SELECT id from sales);
```

```
hive> SELECT *
    > FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);
2    Tie
```

```
SELECT /*+ MAPJOIN(things) */ sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

Hive Operations:subqueries(from)

```
SELECT station, year, AVG(max_temperature)
FROM (
  SELECT station, year, MAX(temperature) AS max_temperature
  FROM records2
  WHERE temperature != 9999
    AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
  GROUP BY station, year
```

Hive Operations:views

```
CREATE VIEW valid_records
AS
SELECT *
FROM records2
WHERE temperature != 9999
  AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9);
```

```
CREATE VIEW max_temperatures (station, year, max_temperature)
AS
SELECT station, year, MAX(temperature)
FROM valid_records
GROUP BY station, year;
```

Hive Operations:serde

```
CREATE TABLE stations (usaf STRING, wban STRING, name STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "(\\d{6}) (\\d{5}) (.{29}) .*"
);
```

```
LOAD DATA LOCAL INPATH "input/ncdc/metadata/stations-fixed-width.txt"
INTO TABLE stations;
```

```
hive> SELECT * FROM stations LIMIT 4;
010000    99999    BOGUS NORWAY
010003    99999    BOGUS NORWAY
010010    99999    JAN MAYEN
010013    99999    ROST
```

Hive Operations:serde(sort)

```
hive> FROM records2
    > SELECT year, temperature
    > DISTRIBUTE BY year
    > SORT BY year ASC, temperature DESC;
1949    111
1949    78
1950    22
1950    0
1950    -11
```

Hive Operations:serde

```
hive> CREATE EXTERNAL TABLE logs_20120101 (
        host STRING,
        identity STRING,
        user STRING,
        time STRING,
        request STRING,
        status STRING,
        size STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
 "input.regex" =
    "([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\])
➥  ([^ \"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)",
"output.format.string"="%1$s %2$s %3$s %4$s %5$s %6$s %7$s"
)
STORED AS TEXTFILE LOCATION '/data/logs/20120101/';
```

The regular expression used to match and extract groups that are mapped to the table columns. Also note that there's a single space separator where the regular expression is split across two lines.

Determines the order and formatting of the table when it's being written.

```
"input.regex" =
   "([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^ \"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)",
```

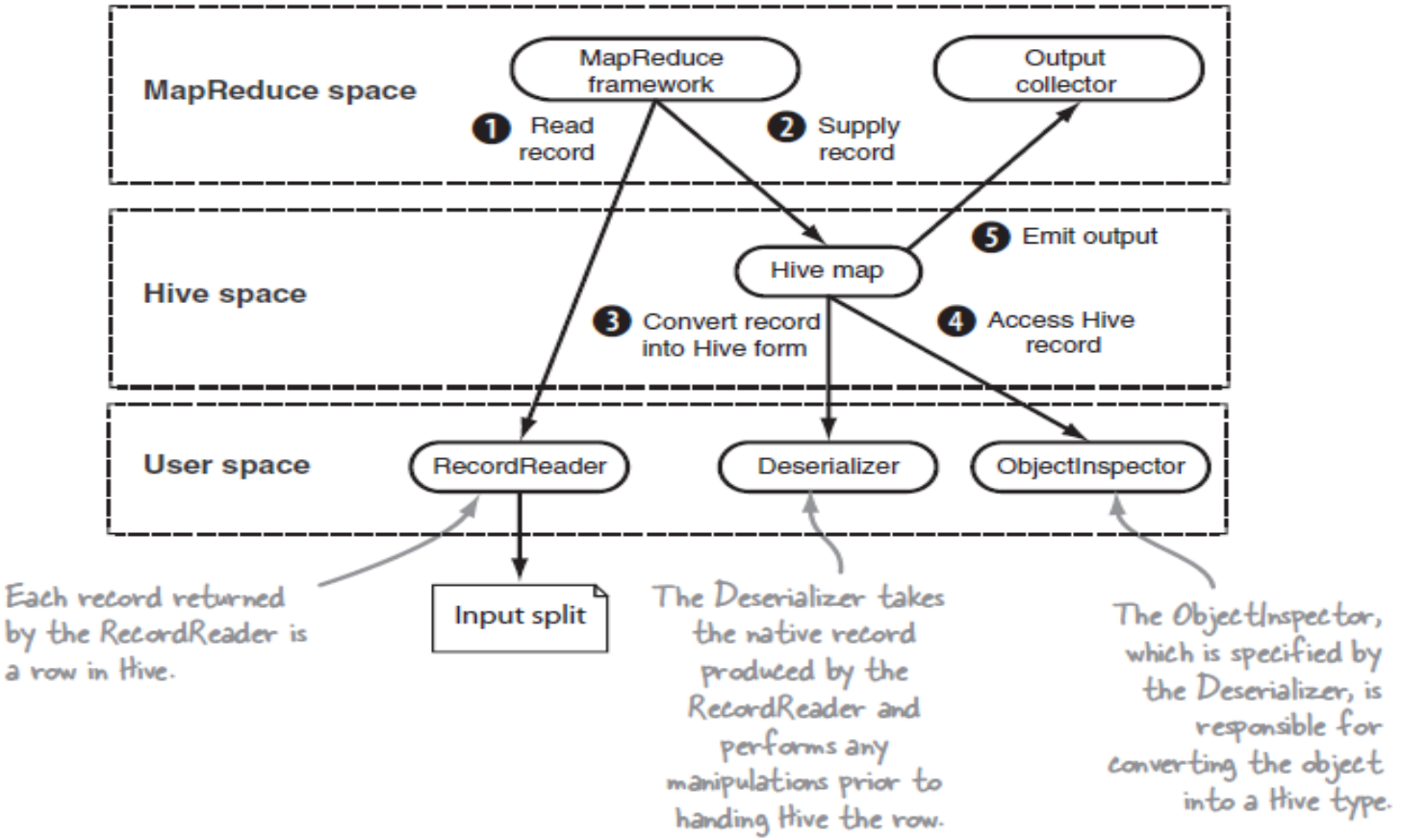
space space space space space space

Hive Operations:serde

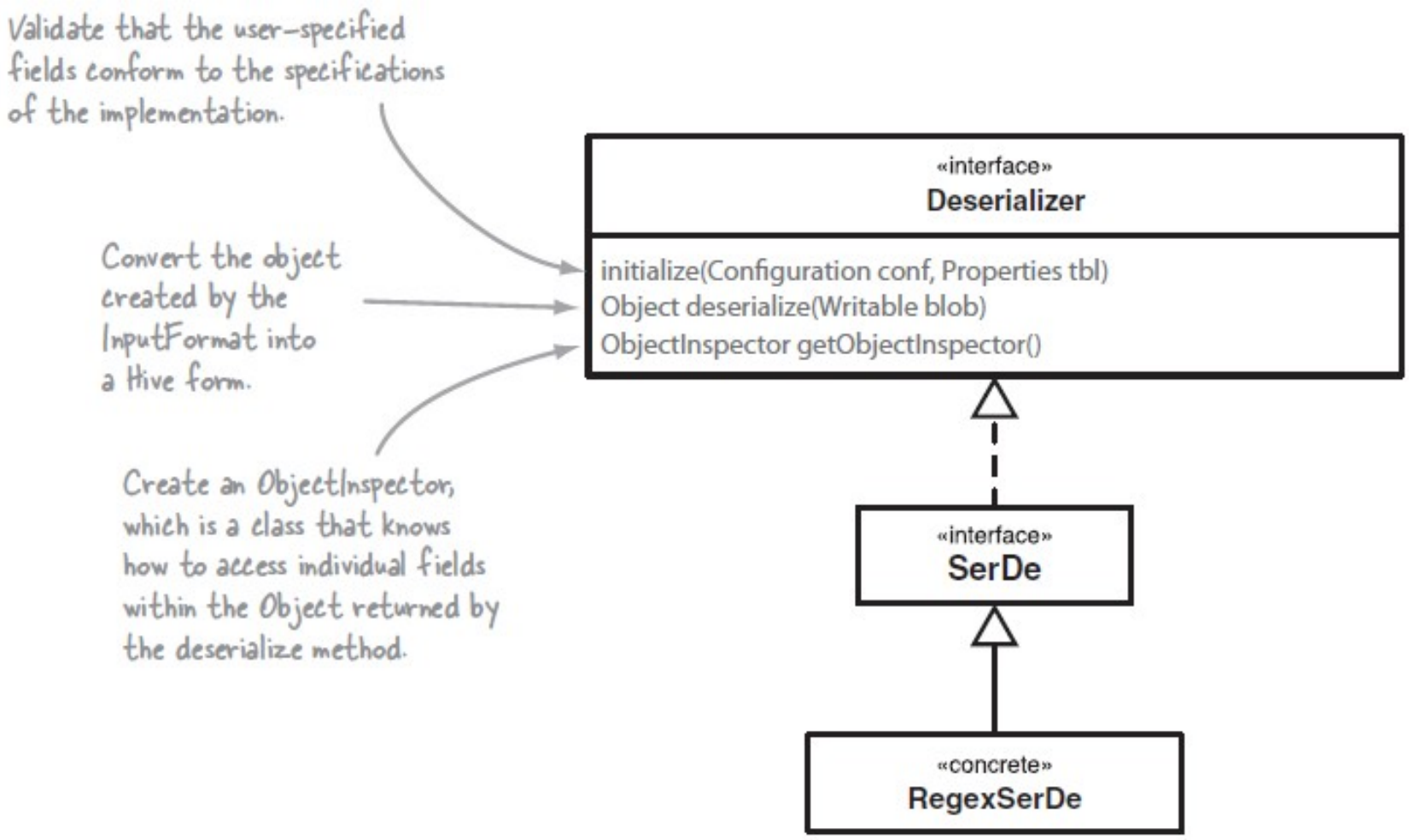
```
hive> add jar $HIVE_HOME/lib/hive-contrib-0.7.1-cdh3u2.jar;
hive> SELECT host, request FROM logs_20120101 LIMIT 10;

89.151.85.133   "GET /movie/127Hours HTTP/1.1"
212.76.137.2    "GET /movie/BlackSwan HTTP/1.1"
74.125.113.104  "GET /movie/TheFighter HTTP/1.1"
212.76.137.2    "GET /movie/Inception HTTP/1.1"
127.0.0.1       "GET /movie/TrueGrit HTTP/1.1"
10.0.12.1       "GET /movie/WintersBone HTTP/1.1"
```

CDH Hive is installed under /usr/lib/hive. You should substitute $HIVE_HOME with the location of your Hive installation because Hive doesn't expand environment variables.

Hive Operations:serde



Hive Operations:serde



```
@Override
public void initialize(Configuration conf, Properties tbl)
    throws SerDeException {

  inputRegex = tbl.getProperty("input.regex");

  String columnNameProperty = tbl.getProperty(
                              Constants.LIST_COLUMNS);
  String columnTypeProperty = tbl.getProperty(
                              Constants.LIST_COLUMN_TYPES);

  inputPattern = Pattern.compile(inputRegex, ...);

  List<String> columnNames = Arrays.asList(
    columnNameProperty.split(","));

  List<TypeInfo> columnTypes = TypeInfoUtils
      .getTypeInfosFromTypeString(columnTypeProperty);

  for (int c = 0; c < numColumns; c++) {
    if (!columnTypes.get(c).equals(TypeInfoFactory.stringTypeInfo)) {
      throw new SerDeException(...);
    }
  }

  List<ObjectInspector> columnOIs = new ArrayList<ObjectInspector>(
      columnNames.size());
```

Read the regular expression from the table definition.

Read the column names from the table definition.

Read the column types from the table definition.

Construct the Java Pattern object, which will be used in the deserialize method.

Tokenize the column names.

Tokenize the column types.

Ensure that each column type is a String.

Hive Operations:serde

```
  for (int c = 0; c < numColumns; c++) {
    columnOIs.add(
      PrimitiveObjectInspectorFactory.javaStringObjectInspector);
  }

  rowOI =
     ObjectInspectorFactory.getStandardStructObjectInspector(
      columnNames, columnOIs);
}

@Override
public ObjectInspector getObjectInspector() throws SerDeException {
  return rowOI;
}
```

Create a primitive ObjectInspector for each field.

Create a structure ObjectInspector for the row that works with List and Array-based representations of rows.

Hive Operations:serde

```
@Override
public Object deserialize(Writable blob) throws SerDeException {

  Text rowText = (Text) blob;

  Matcher m = inputPattern.matcher(rowText.toString());

  // If do not match, ignore the line, return a row with all nulls.
  if (!m.matches()) {
    return null;
  }

  // Otherwise, return the row.
  for (int c = 0; c < numColumns; c++) {
    try {
      row.set(c, m.group(c + 1));
    } catch (RuntimeException e) {
      row.set(c, null);
    }
  }
  return row;
}
```

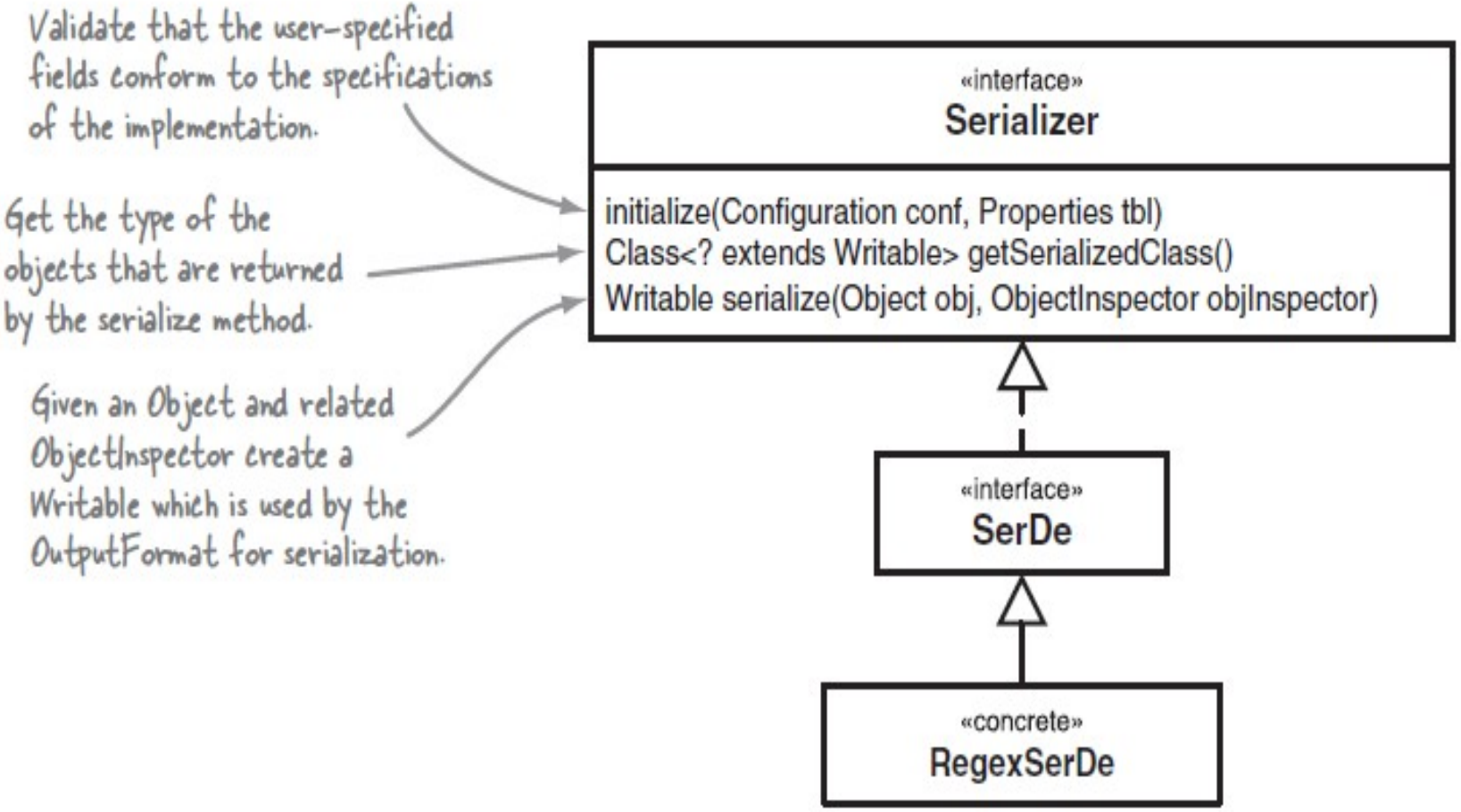
Convert the Writable into a Text object.

If the regular expression didn't match the record, return null.

For each group in the regular expression, set the appropriate column in the array—the row is a reusable ArrayList that was created in the initialize method, which was omitted for brevity.

Set a null if you ran out of groups.

Hive Operations:serde



```
@Override
public Class<? extends Writable> getSerializedClass() {
  return Text.class;
}

Object[] outputFields;
Text outputRowText;

@Override
public Writable serialize(Object obj, ObjectInspector objInspector)
    throws SerDeException {

  StructObjectInspector outputRowOI =
    (StructObjectInspector) objInspector;
  List<? extends StructField> outputFieldRefs = outputRowOI
      .getAllStructFieldRefs();

  for (int c = 0; c < numColumns; c++) {
    Object field = outputRowOI
        .getStructFieldData(obj, outputFieldRefs.get(c));
    ObjectInspector fieldOI = outputFieldRefs.get(c)
        .getFieldObjectInspector();
    StringObjectInspector fieldStringOI = (StringObjectInspector)
      fieldOI;
    outputFields[c] =
      fieldStringOI.getPrimitiveJavaObject(field);
  }

  String outputRowString = String.format(
      outputFormatString, outputFields);

  outputRowText.set(outputRowString);
  return outputRowText;
}
```

Tell Hive that your serialize method produces Text objects.

Extract the individual ObjectInspector for each field in the table.

Use the ObjectInspector to extract the column.

Create the output line using all the columns with the format defined in the output format string, which is set in the SerDe properties

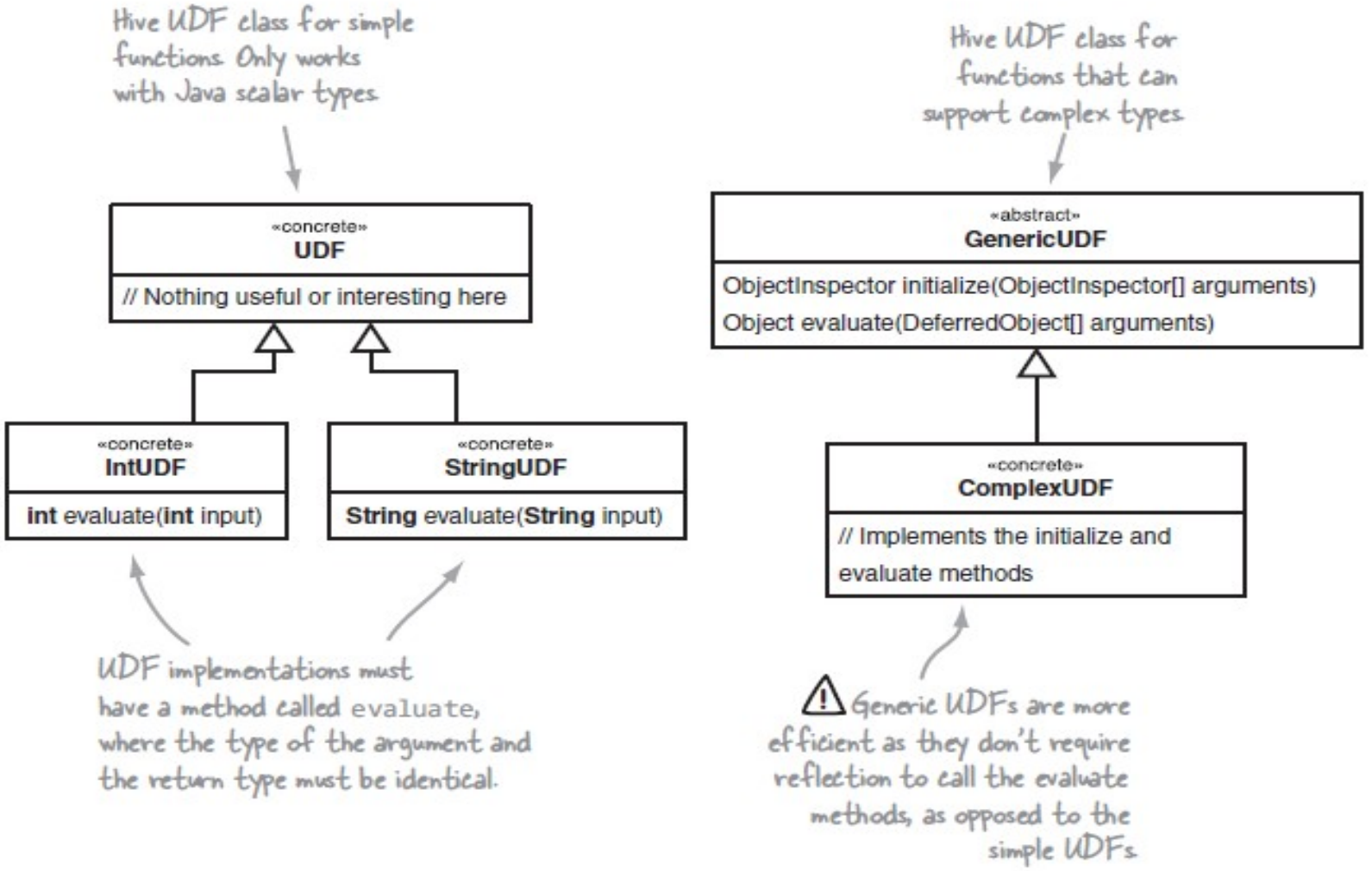
Hive:data analytics:example-1

```
    > select * from logs_20120101;
OK
89.151.85.133   -       -       [23/Jun/2009:10:39:11 +0300]    "GET /movie/127Hours HTTP/1.1"  200     766
212.76.137.2    -       -       [23/Jun/2009:10:39:11 +0300]    "GET /movie/BlackSwan HTTP/1.1" 200     766
74.125.113.104  -       -       [23/Jun/2009:10:39:11 +0300]    "GET /movie/TheFighter HTTP/1.1"        200     766
212.76.137.2    -       -       [23/Jun/2009:10:39:11 +0300]    "GET /movie/Inception HTTP/1.1" 200     766
127.0.0.1       -       -       [23/Jun/2009:10:39:11 +0300]    "GET /movie/TrueGrit HTTP/1.1"  200     766
10.0.12.1       -       -       [23/Jun/2009:10:39:11 +0300]    "GET /movie/WintersBone HTTP/1.1"       200     766
Time taken: 0.128 seconds, Fetched: 6 row(s)
```

```
hive> select * from movie categories;
FAILED: SemanticException [Error 10001]: Line 1:14 Tab
hive> select * from movie_categories;
OK
127Hours        Adventure
127Hours        Drama
BlackSwan       Drama
BlackSwan       Thriller
TheFighter      Drama
TheFighter      Drama
13Assassins     Action
Time taken: 0.088 seconds, Fetched: 7 row(s)
```

1. join table on movie to view categories
2. See geographical location the request is coming from
3. Understand UDFs and UDAF
4. See dynamic and static partitioning in progress

Hive:data analytics:example-1:UDFsand UDAFs



Hive:data analytics:example-1:UDFsand UDAFs

```
public class ExtractMovieUDF extends UDF {
  private Text result = new Text();
  public Text evaluate(final Text t) {
    if (t == null) { return null; }
    String s = t.toString();
    String[] parts = StringUtils.split(s, " ");
    if(parts.length != 3) {
      return null;
    }
    String path = parts[1];

    if(!path.startsWith("/movie/")) {
      return null;
    }
    result.set(path.substring(7));
    return result;
  }
}
```

Looking at the UDF class, there aren't actually any methods to override to implement the function. Hive actually uses reflection to find methods whose names are evaluate and matches the arguments used in the HiveQL function call. Hive can work with both the Hadoop Writables and the Java primitives, but it's recommended to work with the Writables since they can be reused.

This UDF works on the request field, which you split into three parts: the HTTP method, the resource (which is the URL path), and the protocol.

Ignore URLs that don't pertain to the movie part of your website.

Extract the text after the leading movie path, which contains your movie title.

The Description annotation is used to provide usage information in the Hive shell (you'll see how this works following this code).

```
@Description(
    name = "country",
    value = "_FUNC_(ip, geolocfile) - Returns the geolocated " +
    "country code for the IP"
)
public class GeolocUDF extends GenericUDF {
  private LookupService geoloc;
  private ObjectInspectorConverters.Converter[] converters;

  @Override
  public ObjectInspector initialize(ObjectInspector[] arguments) {
    converters =
      new ObjectInspectorConverters.Converter[arguments.length];
    for (int i = 0; i < arguments.length; i++) {
      converters[i] =
        ObjectInspectorConverters.getConverter(arguments[i],
        PrimitiveObjectInspectorFactory.javaStringObjectInspector);

  }
  return PrimitiveObjectInspectorFactory
    .getPrimitiveJavaObjectInspector(
        PrimitiveObjectInspector.PrimitiveCategory.STRING);
}
```

The geolocation lookup class

Converters, which you'll use to convert the input types to the types you want to operate with.

Create a converter that you can use in the evaluate method to convert all the arguments (which in your case are the IP address and geolocation file) from their native type into Java Strings.

Specify that the return type for the UDF (in other words the evaluate function) will be a Java String.

```
@Override
public Object evaluate(GenericUDF.DeferredObject[] arguments) {

  Text ip = (Text) converters[0].convert(arguments[0].get());
  Text filename = (Text) converters[1].convert(arguments[1].get());

  return lookup(ip, filename);
}

protected String lookup(Text ip, Text filename)
                        throws HiveException {
  try {
    if (geoloc == null) {
      URL u = getClass().getClassLoader()
      .getResource(filename.toString());
      geoloc =
          new LookupService(u.getFile(),
                            LookupService.GEOIP_MEMORY_CACHE);
    }

    String countryCode =
      geoloc.getCountry(ip.toString()).getCode();

    if ("--".equals(countryCode)) {
      return null;
    }

    return countryCode;
  } catch (IOException e) {
    throw new HiveException("Caught IO exception", e);
  }
```

After retrieving the IP address and geolocation filename from the arguments, call a function to perform the geolocation.

Load the geolocation data file from the distributed cache.

Create an instance of the MaxMind Lookup class.

Perform the geolocation and extract the country code.

Return the country code.

Dynamic partitions needs to be explicitly enabled in Hive.

Earlier you specified that the viewed_movies was a bucketed table with 64 buckets. Bucketed tables are optimized for sampling because without them extracting a sample from a table requires a full table scan. Whenever you write to a bucketed table, you need to make sure that you either set hive.enforce.bucketing to true, or set mapred.reduce.tasks to the number of buckets.

```
hive> SET hive.exec.dynamic.partition=true;

hive> SET hive.enforce.bucketing = true;

hive> add jar /home/mohit/Work/BigData/resources/hive/
        lib/hive-contrib-0.11.0.2.0.5.0-67.jar;
```

Add the JAR containing your UDF so that it can be used in MapReduce.

```
hive> ADD jar /home/mohit/Work/BigData/TOOLS/Hive/
        target/Hive-1.0-SNAPSHOT-jar-with-dependencies.jar;
```

Add the geolocation data file into the distributed cache.

```
hive> ADD file /home/mohit/Work/BigData/input/hive/
        serde/GeoIP.dat;
```

Define country_udf as the alias for your geolocation UDF and specify the class name.

```
hive> CREATE temporary function country_udf AS 'org.mk.
        training.hive.serde.udf.GeolocUDF';

hive> CREATE temporary function movie_udf AS 'org.mk.
        training.hive.serde.udf.ExtractMovieUDF';
```

Define an alias for your movie UDF, which extracts the movie name from the URL path.

Enable compression for MapReduce job outputs.

```
hive> SET hive.exec.compress.output=true;

hive> SET hive.exec.compress.intermediate = true;

hive> SET mapred.output.compression.codec =
          org.apache.hadoop.io.compress.SnappyCodec;

hive> INSERT OVERWRITE TABLE viewed_movies
      PARTITION (dt='2012-01-01', country)
      SELECT host, movie_udf(request),
        country_udf(host, "GeoIP.dat")
      FROM logs_20120101;

hive> SELECT * from viewed_movies;
OK
89.151.85.133   127Hours     2012-01-01  GB
212.76.137.2    BlackSwan    2012-01-01  RU
212.76.137.2    Inception    2012-01-01  RU
74.125.113.104  TheFighter   2012-01-01  US
127.0.0.1       TrueGrit     2012-01-01

  __HIVE_DEFAULT_PARTITION__
10.0.12.1       WintersBone  2012-01-01  __HIVE_DEFAULT_PARTITION__
```

Enable compression for intermediate map outputs.

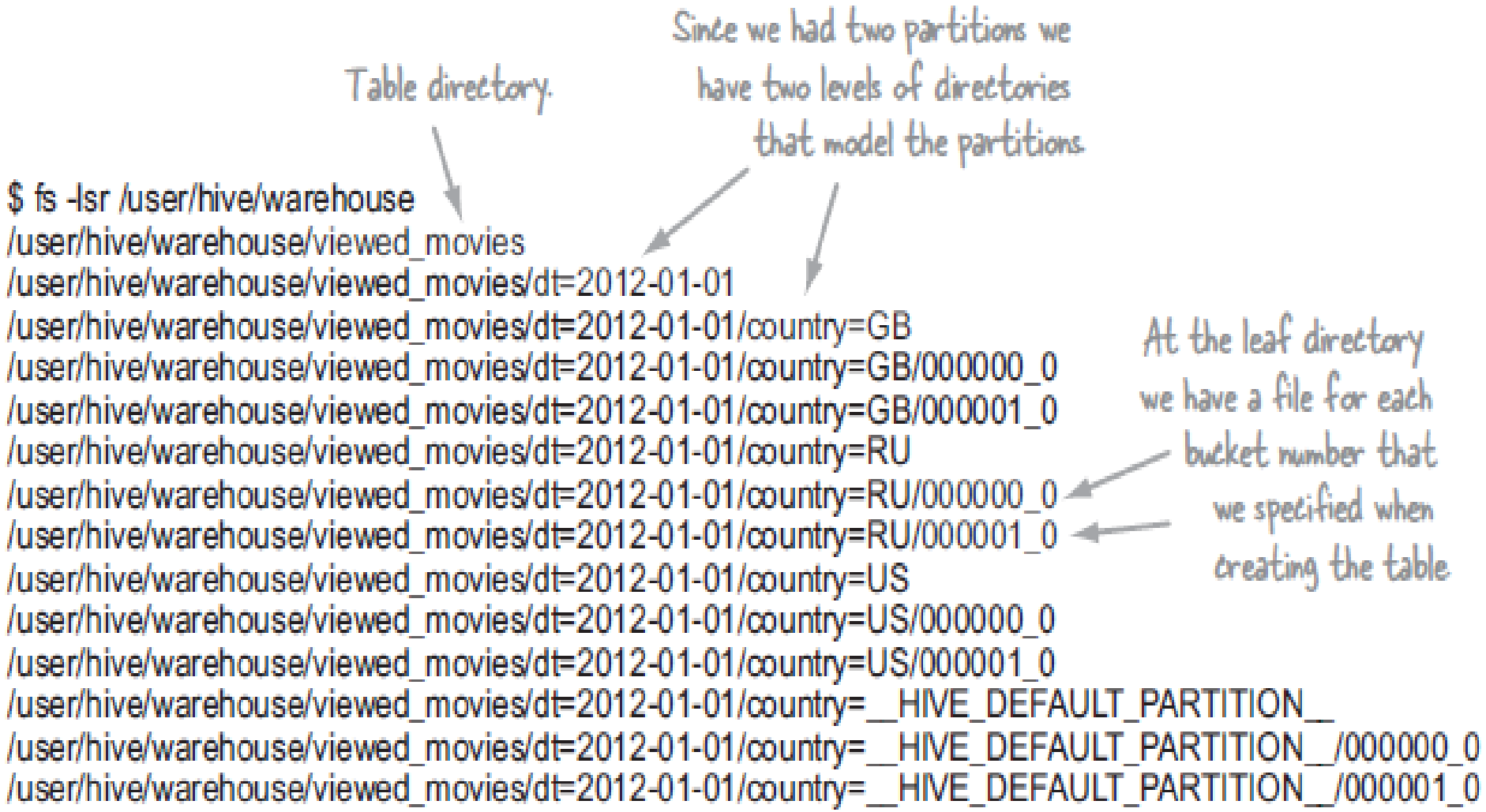
Use the Snappy compression codec.

An example of both a static (the dt column) and dynamic (the country column) partitions in action.

Call your UDF specifying the field on which it should operate (the host column from the logs table), and the filename of the geolocation data file, which is in the distributed cache.

The __HIVE_DEFAULT_PARTITION__ is used to store records whose dynamic partition column value is NULL, or the empty string.

Hive:data analytics:example-1:partitioning and bucketing



Hive:data analytics:example-1:joins

```
hive> SELECT * FROM viewed_movies;
89.151.85.133   127Hours      2012-01-01  GB
212.76.137.2    BlackSwan     2012-01-01  RU
212.76.137.2    Inception     2012-01-01  RU
74.125.113.104  TheFighter    2012-01-01  US
127.0.0.1       TrueGrit      2012-01-01  __HIVE_DEFAULT_PARTITION__
10.0.12.1       WintersBone   2012-01-01  __HIVE_DEFAULT_PARTITION__

hive> SELECT * from movie_categories;
127Hours      Adventure
127Hours      Drama
BlackSwan     Drama
BlackSwan     Thriller
TheFighter    Drama
13Assassins   Action

hive> SET mapred.reduce.tasks=2;

hive> SELECT viewed_movies.movie, movie_categories.category
      FROM viewed_movies
      JOIN movie_categories ON
           (viewed_movies.movie = movie_categories.title);
127Hours     Adventure
127Hours     Drama
BlackSwan    Drama
BlackSwan    Thriller
TheFighter   Drama
```

Set the number of reducers for the job.

Indicate that you're joining the viewed_movies table with the movie_categories table, and specify the columns that should be joined (viewed_movies.movie and movie_categories.title).

Hive:data analytics:example-1:joins

```
hive> SELECT viewed_movies.movie, movie_categories.category
      FROM viewed_movies
      LEFT OUTER JOIN movie_categories ON
           (viewed_movies.movie = movie_categories.title);
127Hours      Adventure
127Hours      Drama
BlackSwan     Drama
BlackSwan     Thriller
Inception     NULL
TheFighter    Drama
TrueGrit      NULL
WintersBone   NULL
```

A left outer join includes all rows from the viewed_movies table regardless of whether or not they have a matching row in the movie_categories table.

You have some movies that don't have categories, in which case with a left outer join they'll have a NULL value for the category field.

Hive:data analytics:example-1:joins

```
hive> SELECT viewed_movies.movie, movie_categories.category,
            movie_categories.title
      FROM viewed_movies
      RIGHT OUTER JOIN movie_categories ON
           (viewed_movies.movie = movie_categories.title);
NULL         Action     13Assassins
BlackSwan    Drama      BlackSwan
BlackSwan    Thriller   BlackSwan
TheFighter   Drama      TheFighter
TheFighter   Drama      TheFighter
127Hours     Adventure  127Hours
127Hours     Drama      127Hours
```

A right outer join includes all rows from the movie_categories tables, and only matching rows from the viewed_movies table.

Because all the rows from the movie_categories are included, rows that don't have a matching entry in the viewed_movies table will contain a *NULL* value for any columns from that table.

Hive:data analytics:example-1:joins

```
hive> SELECT viewed_movies.movie, movie_categories.category,
            movie_categories.title
      FROM viewed_movies
      FULL OUTER JOIN movie_categories ON
           (viewed_movies.movie = movie_categories.title);
NULL          Action      13Assassins
BlackSwan     Drama       BlackSwan
BlackSwan     Thriller    BlackSwan
TheFighter    Drama       TheFighter
TheFighter    Drama       TheFighter
TrueGrit      NULL        NULL
WintersBone   NULL        NULL
127Hours      Adventure   127Hours
127Hours      Drama       127Hours
Inception     NULL        NULL
```

A full outer join includes all rows from both tables regardless of whether there is a match.

Just like with the left and right outer joins, any rows that fail to match will contain *NULL* values for the table with no corresponding entry.

Hive:data analytics:example-1:joins

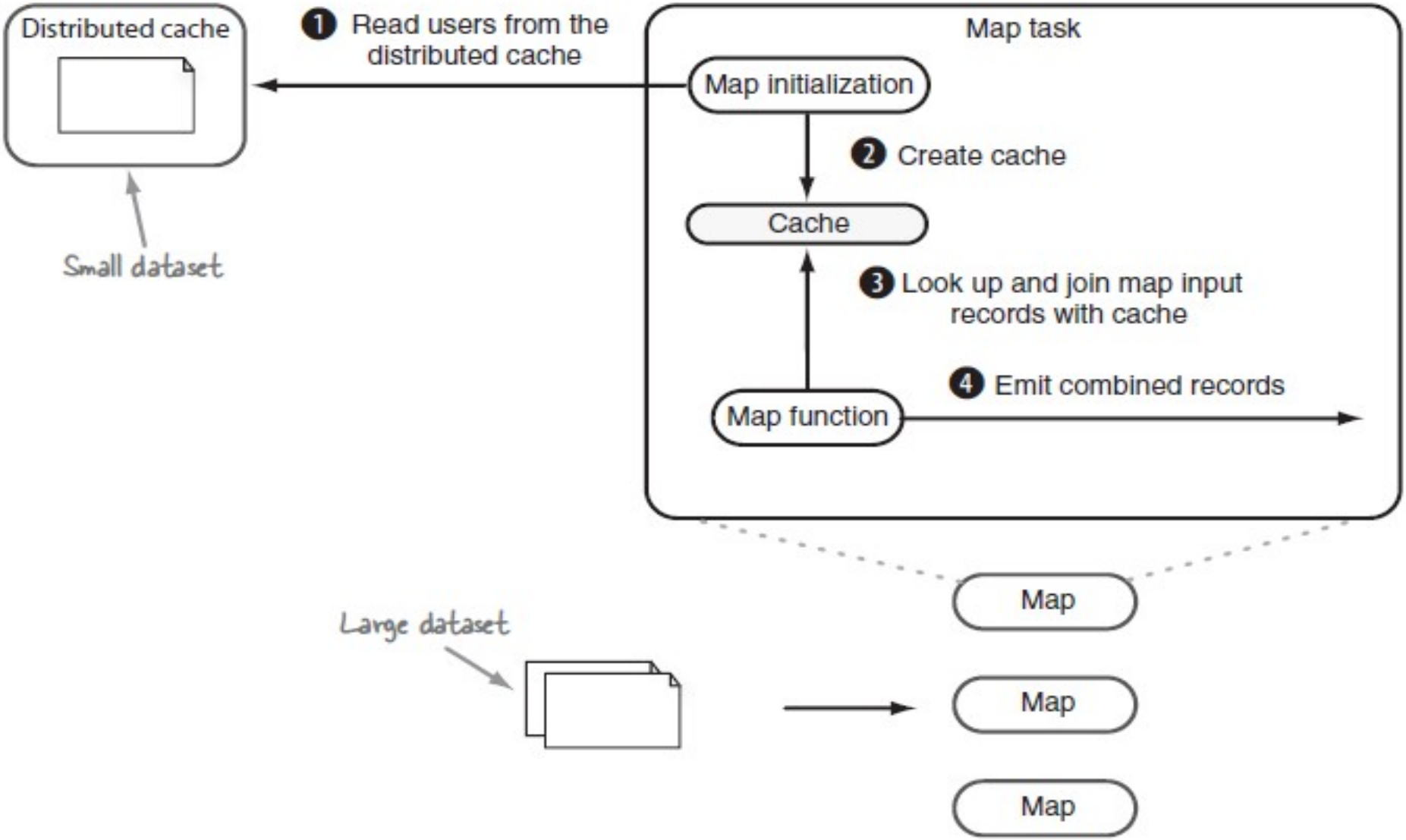
```
hive> SELECT /*+ MAPJOIN(movie_categories) */
        viewed_movies.movie, movie_categories.category
      FROM viewed_movies
      JOIN movie_categories
      ON viewed_movies.movie = movie_categories.title;
```

The hint which triggers the map join and also tells Hive which table (movie_categories) to cache.

The files in the small tables are smaller than the value specified in hive .mapjoin.smalltable.filesize, whose default value is set at 25 MB.

The memory utilization of loading the small table must be less than hive .mapjoin.localtask.max.memory.usage, which is set at 0.90 (90 percent) by default.

Hive:data analytics:example-1:joins



Hive:data analytics:example-1:joins

```
hive> SET hive.auto.convert.join = true;
```

Enable Hive's automatic join optimization to convert repartition joins to replicated joins if one of the tables is small enough.

```
hive> SELECT viewed_movies.movie, movie_categories.category
      FROM viewed_movies
      JOIN movie_categories
      ON viewed_movies.movie = movie_categories.title;
...
Mapred Local Task Succeeded. Convert the Join into MapJoin
...
```

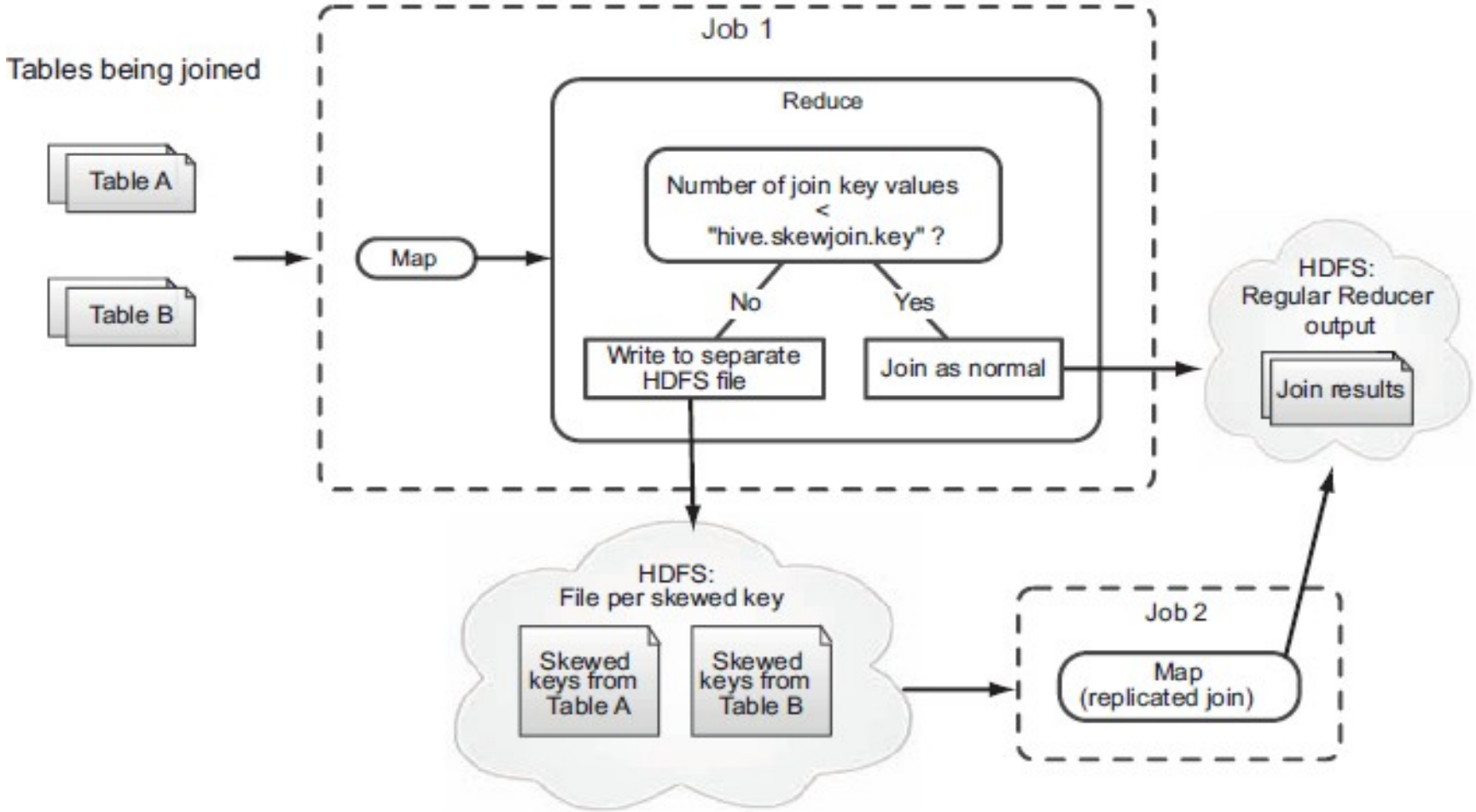
This output tells you that Hive decided to go ahead and use a replicated join.

Hive:data analytics:example-1:semi-joins

```
hive> SELECT  viewed_movies.movie
      FROM viewed_movies
      LEFT SEMI JOIN movie_categories
      ON viewed_movies.movie = movie_categories.title;
127Hours
BlackSwan
TheFighter
```

In a semi-join you can't have the result contain any fields from the right-hand side of the join, which in your case is the movie_categories table.

Hive:data analytics:example-1:skew-joins



Hive:data analytics:example-1:skew-joins

```
hive> SET hive.optimize.skewjoin = true;
```

Tell Hive to optimize joins where it sees skewed data.

```
hive> SET hive.skewjoin.key = 100000;
```

Sets the threshold beyond which a key is considered to be skewed.

Hive:UDAFs

```
public class Maximum extends UDAF {

  public static class MaximumIntUDAFEvaluator implements UDAFEvaluator {

    private IntWritable result;

    public void init() {
      result = null;
    }
    public boolean iterate(IntWritable value) {
      if (value == null) {
        return true;
      }
      if (result == null) {
        result = new IntWritable(value.get());
      } else {
        result.set(Math.max(result.get(), value.get()));
      }
      return true;
    }

    public IntWritable terminatePartial() {
      return result;
    }

    public boolean merge(IntWritable other) {
      return iterate(other);
    }

    public IntWritable terminate() {
      return result;
    }
  }
}
```

Hive:UDAFs

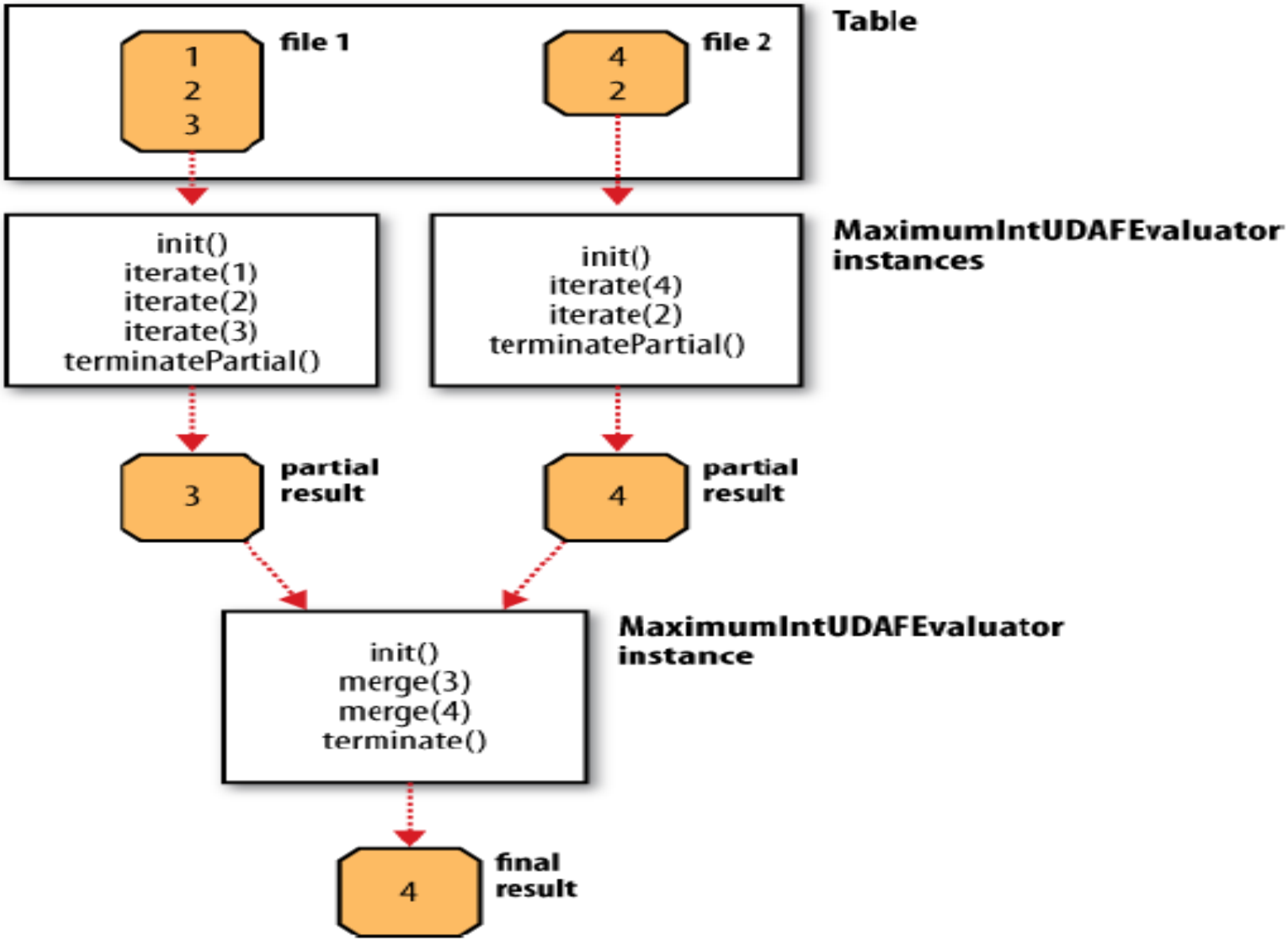


Table
file 1
1
2
3
file 2
4
2
MaximumIntUDAFEvaluator instances
init()
iterate(1)
iterate(2)
iterate(3)
terminatePartial()
init()
iterate(4)
iterate(2)
terminatePartial()
partial result
3
partial result
4
MaximumIntUDAFEvaluator instance
init()
merge(3)
merge(4)
terminate()
final result
4

Hive:OrderBy vs SortBy

```
hive> SELECT movie_categories.category, count(1) AS cnt
      FROM viewed_movies
      JOIN movie_categories ON
           (viewed_movies.movie = movie_categories.title)
      WHERE viewed_movies.country = "RU"
      GROUP BY movie_categories.category
      ORDER BY cnt DESC;
```

Search for the top categories in Russia.
Your table is partitioned by country,
which speeds up your query because
only files contained in these partitions will be loaded.

Order the results by the number of movies in each category,
so the most popular category is the first result

Hive:OrderBy vs SortBy:Explain

```
ABSTRACT SYNTAX TREE:
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF
(TOK_TABNAME viewed_movies)) (TOK_TABREF
(TOK_TABNAME movie_categories)) (= (.
...
```

```
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-2 depends on stages: Stage-1
  Stage-3 depends on stages: Stage-2
  Stage-0 is a root stage
```

```
Stage: Stage-1
  Map Reduce
    Alias -> Map Operator Tree:
      movie_categories
        TableScan
          alias: movie_categories
          Reduce Output Operator
            key expressions:
                  expr: title
                  type: string
            sort order: +
            Map-reduce partition columns:
                  expr: title
                  type: string
            tag: 1
            value expressions:
                  expr: category
                  type: string
      viewed_movies
        TableScan
          alias: viewed_movies
          Reduce Output Operator
            key expressions:
                  expr: movie
                  type: string
            sort order: +
            Map-reduce partition columns:
                  expr: movie
                  type: string
            tag: 0
            value expressions:
                  expr: country
                  type: string
```

This tells you that the movie_categories table is one of the input tables for the job.

This tells you that the map is emitting an output key/value tuple where the key is the movie title.

The title, which is the output key, is being used for partitioning.

The output value is the movie category.

This section shows details for the viewed_movies table. The output key/value tuples are the movie title and country, respectively.

```
Reduce Operator Tree:
  Join Operator
    condition map:
         Inner Join 0 to 1
    condition expressions:
      0 {VALUE._col3}

    1 {VALUE._col1}
  handleSkewJoin: false
  outputColumnNames: _col3, _col7
  Filter Operator
    predicate:
        expr: (_col3 = 'RU')
        type: boolean
    Select Operator
      expressions:
            expr: _col7
            type: string
      outputColumnNames: _col7
      Group By Operator
        aggregations:
              expr: count()
        bucketGroup: false
        keys:
              expr: _col7
              type: string
        mode: hash
        outputColumnNames: _col0, _col1
        File Output Operator
          compressed: true
          GlobalTableId: 0
          table:
            input format:  org.apache.hadoop.mapred.
                           SequenceFileInputFormat
            output format: org.apache.hadoop.hive.ql.io.
                           HiveSequenceFileOutputFormat
```

The first operation on the reduce side is a join, and by looking down a couple of lines you can see that it's an inner join.

This operator is filtering results based on your criteria that only results from Russia be included.

The select operator does nothing other than pass through data to the next operator.

This group is a localized group within all the rows for a given join key. You're grouping on the movie category, so the output of this group is the category name and a count of the number of rows with the same category.

The last step indicates that the output is being written to file and the output format used to perform the write.