UNIt2-Part2

HIVE

HIVE

Hive is a data warehouse system used for querying and analyzing large datasets stored in HDFS

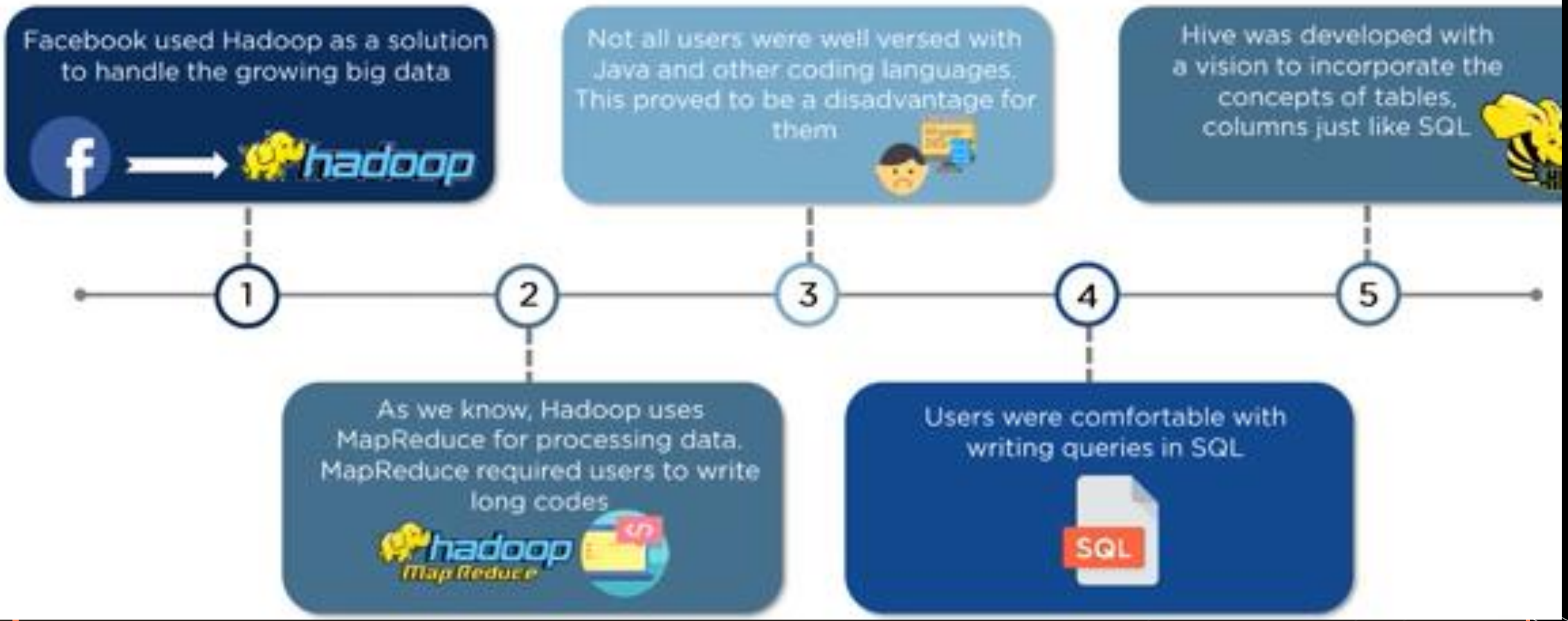
Hive runs on a workstation and converts your SQL query into a series

of MapReduce jobs for execution on a Hadoop cluster.

Hive organizes data into tables, which provide a means for attaching structure to data stored in HDFS

Metadata— such as table schemas—is stored in a database called the meta store.

History of Hive



Installing HIVE

Prerequisites

Java 6

Same version of Hadoop as the cluster

Download link

https://hdoop.apache.org/hive/releases.html

Tar the achieve

```
tar xzf hive-x.y.z-dev.tar.gz
```

Set the path

```
% export HIVE_INSTALL=/home/tom/hive-x.y.z-dev
% export PATH=$PATH:$HIVE_INSTALL/bin
```

HIVE Shell

- The shell is the primary way that we will interact with Hive, by issuing commands in HiveQL.
- HiveQL is Hive’s query language, a dialect of SQL
- Hive shell is launched by typing hive
- Like SQL, HiveQL is generally case insensitive
- The database stores its files in a directory called metastore_db, which is relative to where you ran the hive command from.
- The -f option runs the commands in non-interactive mode.

Examples

Display Database tables

```
hive> SHOW TABLES;
OK
Time taken: 10.425 seconds
```

Create a Table

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

Populating the Table

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

Examples

- Running the LOAD command tells Hive to put the specified local file in its warehouse directory.
- This is a simple filesystem operation. There is no attempt to parse the file and store it in an internal database format
- since Hive does not mandate any particular file format. Files are stored verbatim: they are not modified by Hive.
- Tables are stored as directories under Hive's warehouse directory, which is controlled by the hive.metastore.warehouse.dir, and defaults to /user/hive/warehouse.
- So

  % ls /user/hive/warehouse/records/

  Gives us the list of files stored for this table.
- The OVERWRITE keyword in the LOAD DATA statement tells Hive to delete any existing files in the directory for the table.

Hive Services

Cli

The command line interface to Hive (the shell). This is the default service.

hiveserver

Runs Hive as a server exposing a Thrift service, enabling access from a range of clients written in different languages.

hwi

The Hive Web Interface.

% export ANT_LIB=/path/to/ant/lib % hive --service hwi

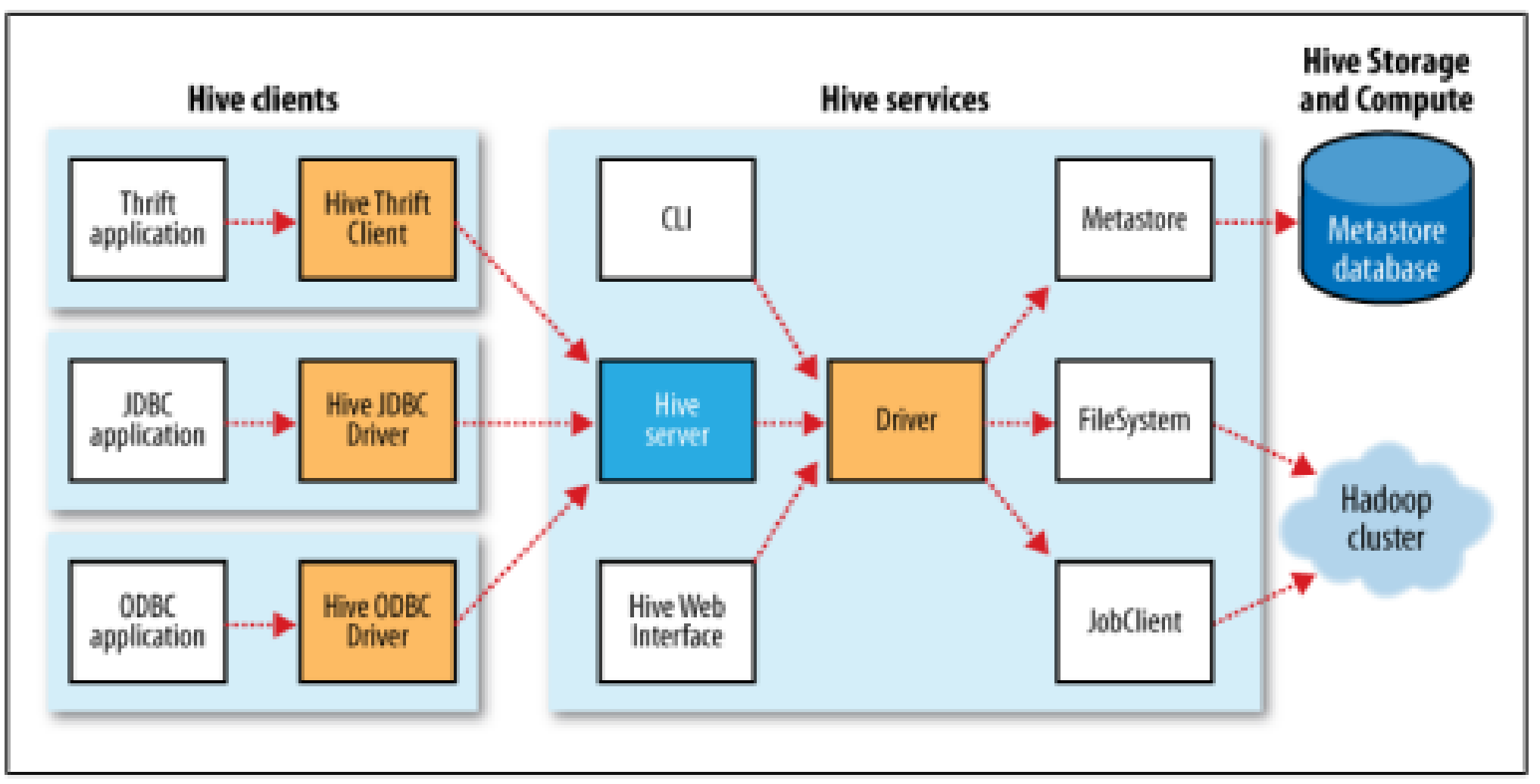
% hive --service hwi

jar

The Hive equivalent to hadoop jar, a convenient way to run Java applications that includes both Hadoop and Hive classes on the classpath.

metastore

By default, the metastore is run in the same process as the Hive service. Using this service, it is possible to run the metastore as a standalone (remote) process. Set the METASTORE_PORT environment variable to specify the port the server will listen on


Hive clients
Hive services
Hive Storage and Compute
Thrift application
Hive Thrift Client
JDBC application
Hive JDBC Driver
ODBC application
Hive ODBC Driver
CLI
Hive server
Hive Web Interface
Driver
Metastore
FileSystem
JobClient
Metastore database
Hadoop cluster

HIVE Clients

Thrift Client

The Hive Thrift Client makes it easy to run Hive commands from a wide range of programming languages.

Thrift bindings for Hive are available for C++, Java, PHP, Python, and Ruby.

They can be found in the src/service/src subdirectory in the Hive distribution.

JDBC Driver

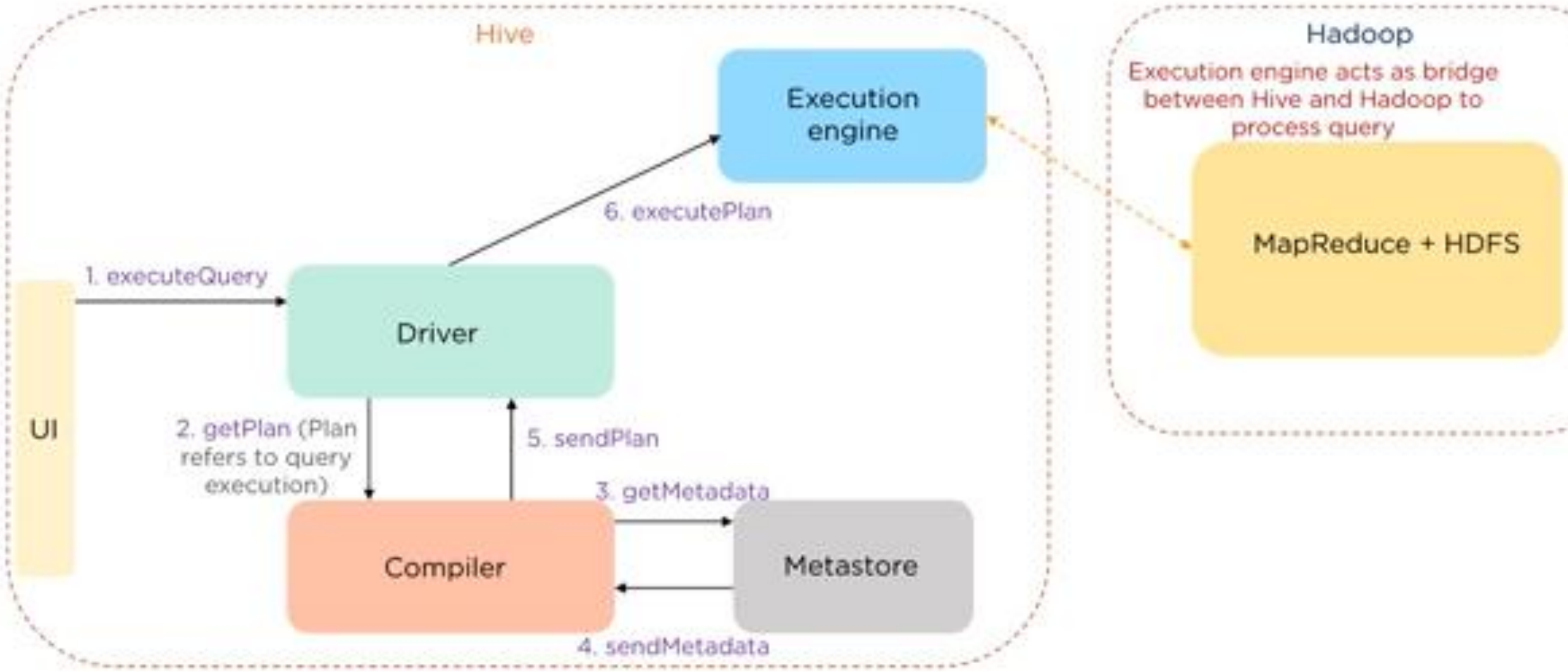
Hive provides a Type 4 (pure Java) JDBC driver, defined in the class org.apache.hadoop.hive.jdbc.HiveDriver.

When configured with a JDBC URI of the form jdbc:hive://host:port/dbname, a Java application will connect to a Hive server running in a separate process at the given host and port.

ODBC Driver

The Hive ODBC Driver allows applications that support the ODBC protocol to connect to Hive.

Data flow in Hive



The Metastore

- The metastore is the central repository of Hive metadata.
- The metastore is divided into two pieces:
  - A service
  - The backing store for the data.

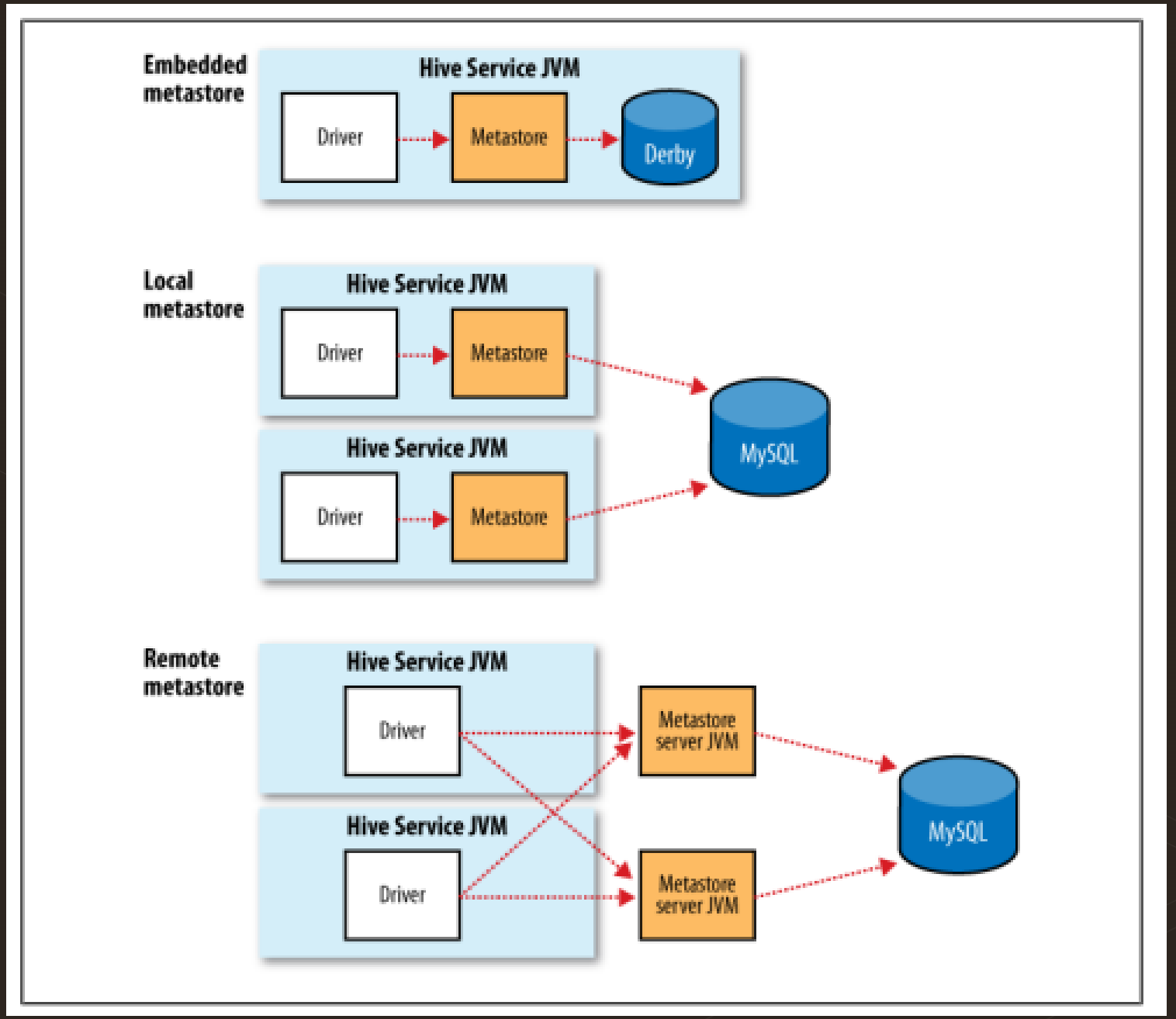
  By default, the metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance backed by the local disk. This is called the embedded meta store configuration

  With embedded meta store only one session can be launched at a time.

  Multiple sessions can be launched by using single standalone database. This configuration is referred to as a local metastore

  Remote meta store, lets one or more metastore servers run in separate processes to the Hive service.

Hive Meta Store Configurations



Comparison with Traditional Databases

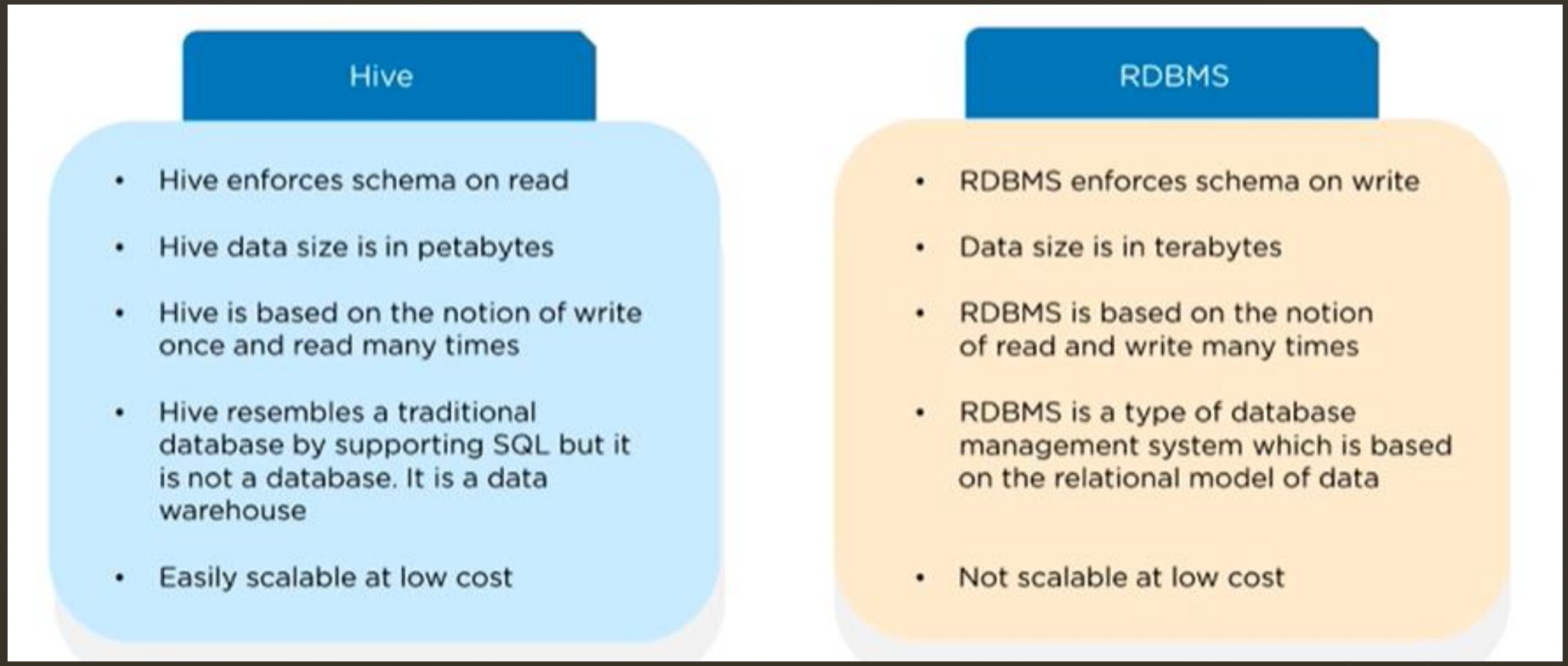
While Hive resembles a traditional database in many ways, its HDFS and MapReduce underpinnings mean that there are a number of architectural differences that directly influence the features that Hive supports

Schema:

- In a traditional database, a table's schema is enforced at data load time. If the data being loaded doesn't conform to the schema, then it is rejected. This design is sometimes called schema on write
- Hive, doesn't verify the data when it is loaded, but rather when a query is issued. This is called schema on read.

Transactions :

- Hive was built to operate over HDFS data using MapReduce, where full-table scans are the norm and a table update is achieved by transforming the data into a new table.
- Hive doesn't define clear semantics for concurrent access to tables, which means applications need to build their own application-level concurrency or locking mechanism


Hive
Hive enforces schema on read
Hive data size is in petabytes
Hive is based on the notion of write once and read many times
Hive resembles a traditional database by supporting SQL but it is not a database. It is a data warehouse
Easily scalable at low cost
RDBMS
RDBMS enforces schema on write
Data size is in terabytes
RDBMS is based on the notion of read and write many times
RDBMS is a type of database management system which is based on the relational model of data
Not scalable at low cost

HiveQL

- Hive’s SQL dialect, called HiveQL, does not support the full SQL-92 specification.
- One reason is being the open source project, it lets the features to be added by developers to meet their users’ needs.
- Hive has some extensions that are not in SQL-92, which have been inspired by syntax from other database systems, notably MySQL.
- SQL-92 constructs that are missing from HiveQL are easy to work around using other language features

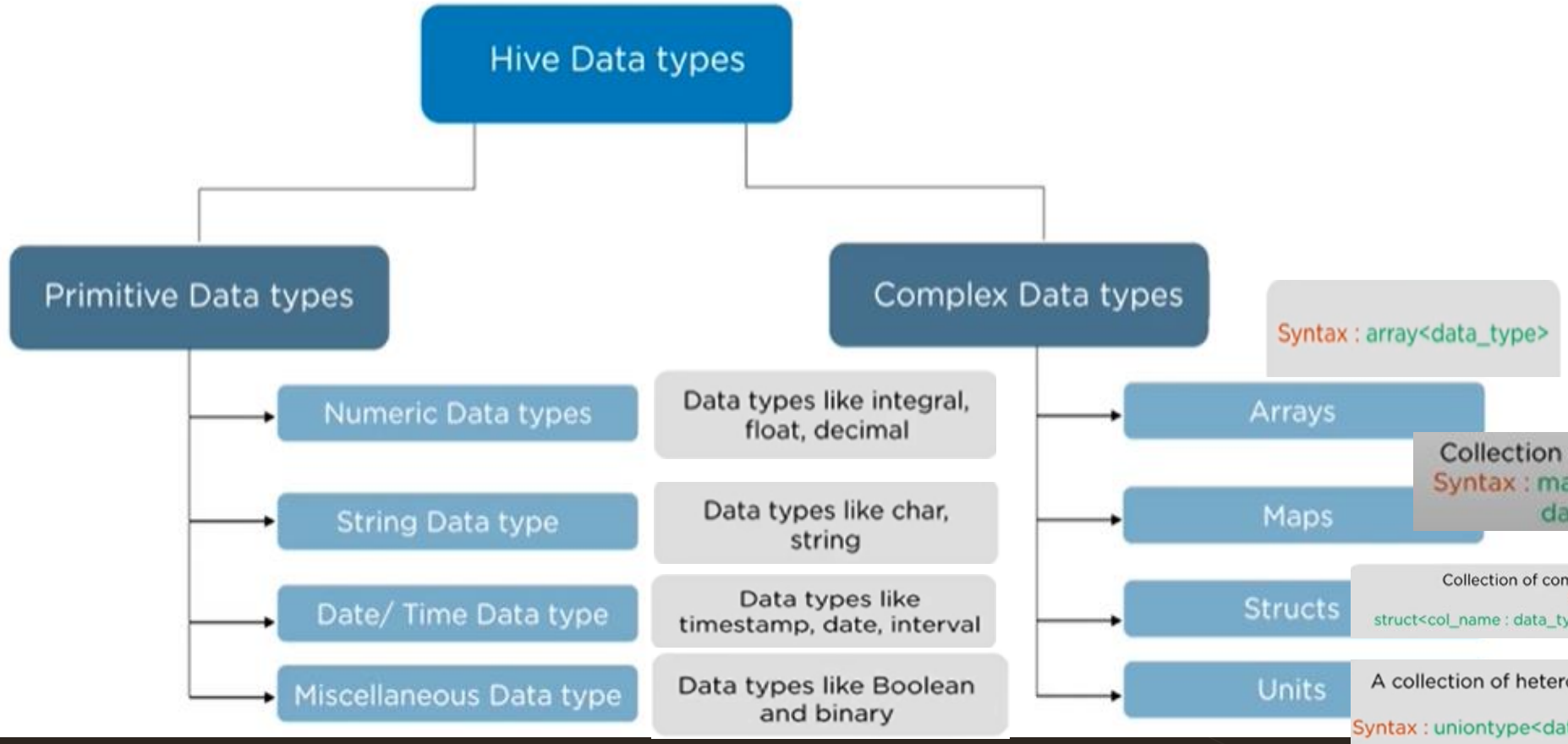
Difference between SQL and HIVE

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

Difference between SQL and HIVE

Feature	SQL	HiveQL
Select	SQL-92	Single table or view in the FROM clause. SORT BY for partial ordering. LIMIT to limit number of rows returned. HAVING not supported.
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 Data types



Primitive Data 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"

Complex Data Types of HIVE

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)` [b]

Tables

- A Hive table is logically made up of the data being stored and the associated metadata describing the layout of the data in the table.
- The data typically resides in HDFS, although it may reside in any Hadoop filesystem, including the local filesystem or S3.
- Hive stores the metadata in a relational database—and not in HDFS.

Managed Tables and External Tables

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

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

For example:

```
CREATE TABLE managed_table (dummy STRING);

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

will move the file hdfs://user/tom/data.txt into Hive's warehouse directory for the

managed_table table, which is hdfs://user/hive/warehouse/managed_table.

If the table is later dropped, using:

```
DROP TABLE managed_table;
```

then the table, including its metadata and its data, is deleted

Managed Tables and External Tables

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

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

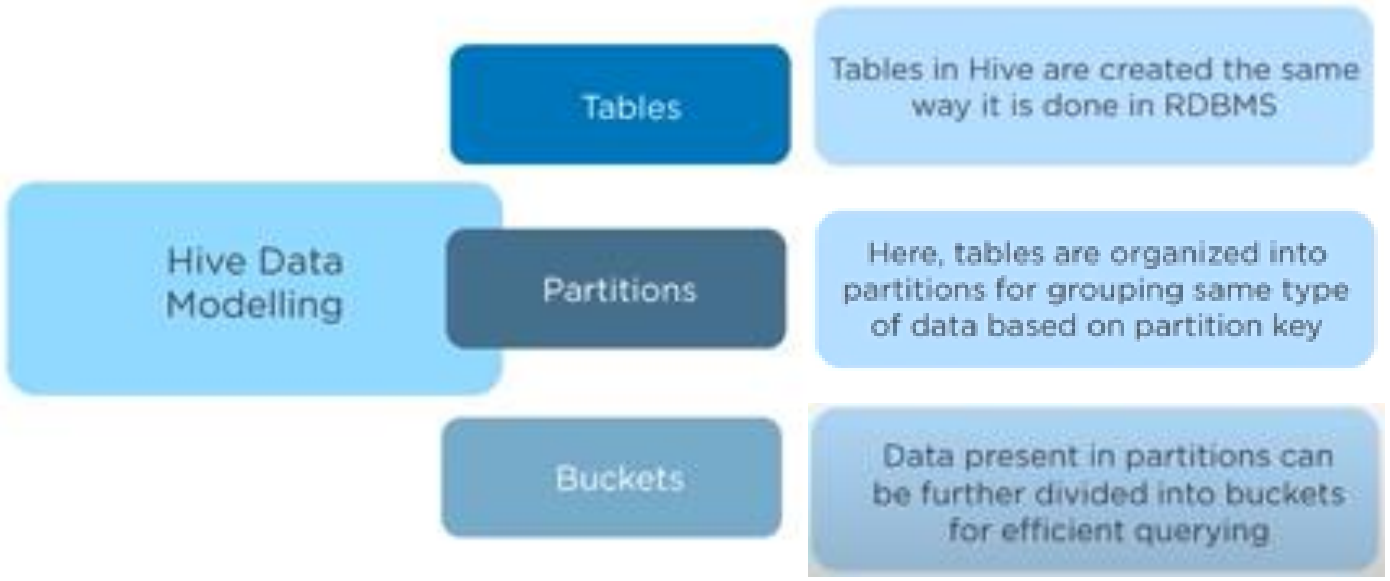
```
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

With the EXTERNAL keyword, Hive knows that it is not managing the data, so it doesn’t move it to its warehouse directory.

Indeed, it doesn’t even check if the external location exists at the time it is defined.

When you drop an external table, Hive will leave the data untouched and only delete the metadata.

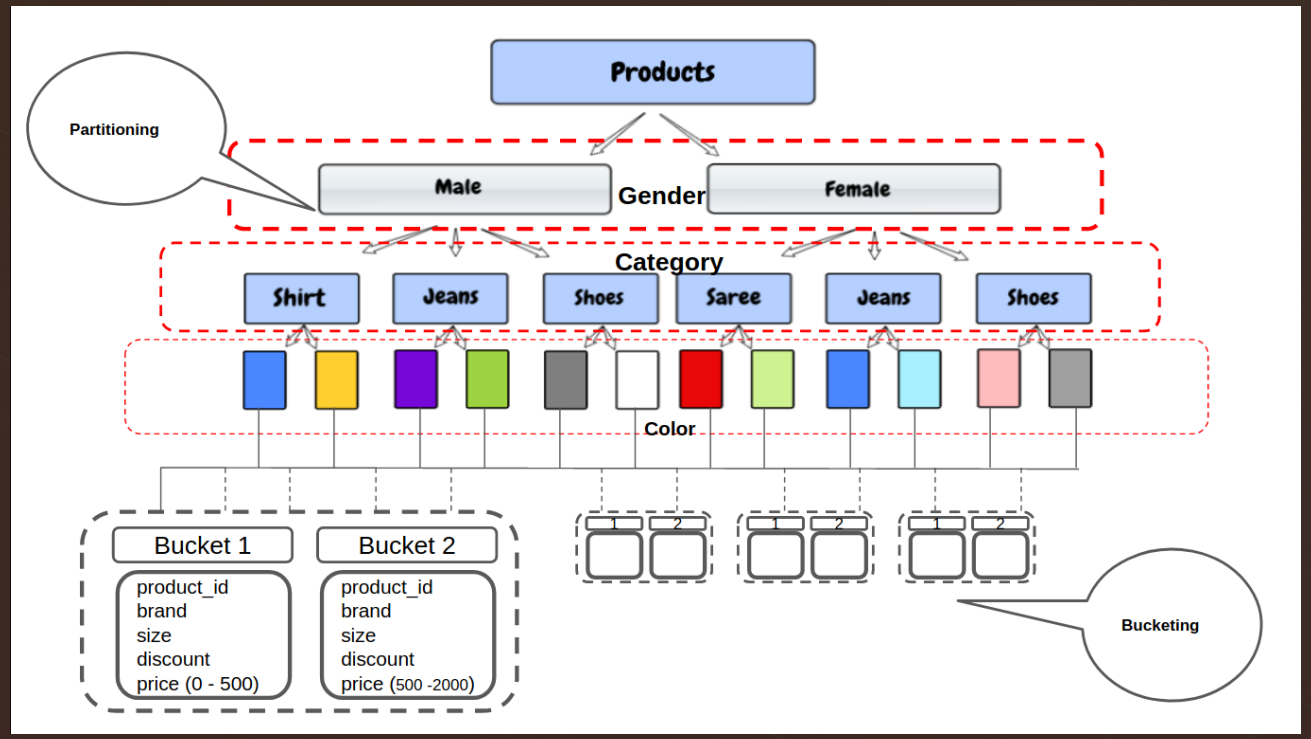
Hive Data Modelling

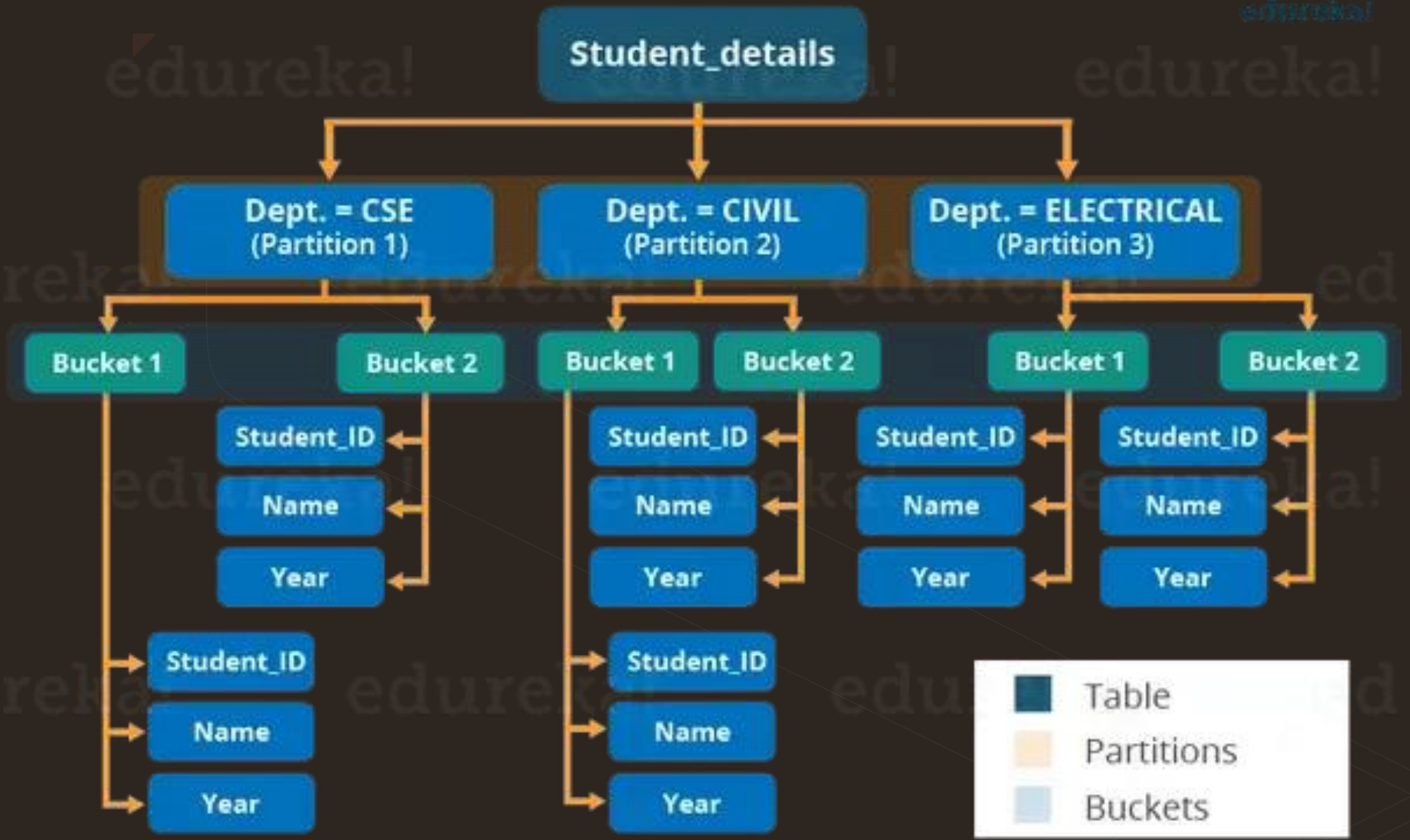


Partitions and Buckets

•Hive organizes tables into partitions, a way of dividing a table into coarse-grained parts based on the value of a partition column,

•Tables or partitions may further be subdivided into buckets, to give extra structure to the data that may be used for more efficient queries.




Student_details
Dept. = CSE (Partition 1)
Dept. = CIVIL (Partition 2)
Dept. = ELECTRICAL (Partition 3)
Bucket 1
Bucket 2
Bucket 1
Bucket 2
Bucket 1
Bucket 2
Student_ID
Name
Year
Student_ID
Name
Year
Student_ID
Name
Year
Student_ID
Name
Year
Student_ID
Name
Year
Student_ID
Name
Year
Table
Partitions
Buckets

Partitions

- Partitions are defined at table creation time‡ using the PARTITIONED BY clause, which takes a list of column definitions.

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

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

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

At the filesystem level, partitions are simply nested subdirectories of the table directory.

```
/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
```

Partitions

We can ask Hive for the partitions in a table using SHOW PARTITIONS:

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

One thing to bear in mind is that the column definitions in the PARTITIONED BY clause are full-fledged table columns, called partition columns;

however, the data files do not contain values for these columns since they are derived from the directory names.

You can use partition columns in SELECT statements in the usual way. Hive performs

input pruning to scan only the relevant partitions. For example:

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

will only scan file1, file2, and file4.

Buckets

- Bucketing imposes extra structure on the table, which Hive can take advantage of when performing certain queries.
- The second reason to bucket a table is to make sampling more efficient
- We use the CLUSTERED BY clause to specify the columns to bucket on and the number of buckets:

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

Here we are using the user ID to determine the bucket

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

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

To populate the bucketed table, we need to set the hive.enforce.bucketing property to true

Storage Formats

There are two dimensions that govern table storage in Hive:

- The row format and
- The file format

Row-format:

The row format dictates how rows, and the fields in a particular row, are stored.

In Hive parlance, the row format is defined by a SerDe for a Serializer-Deserializer.

While querying a table a SerDe will deserialize a row of data from the bytes in the file to objects used internally by Hive to operate on that row of data

When used as a serializer, which is the case when performing an INSERT or CTAS the table's SerDe will serialize Hive's internal representation of a row of data into the bytes that are written to the output file.

File Storage

The file format dictates the container format for fields in a row.

- Plaintext Format
- Row-oriented
- Column-oriented

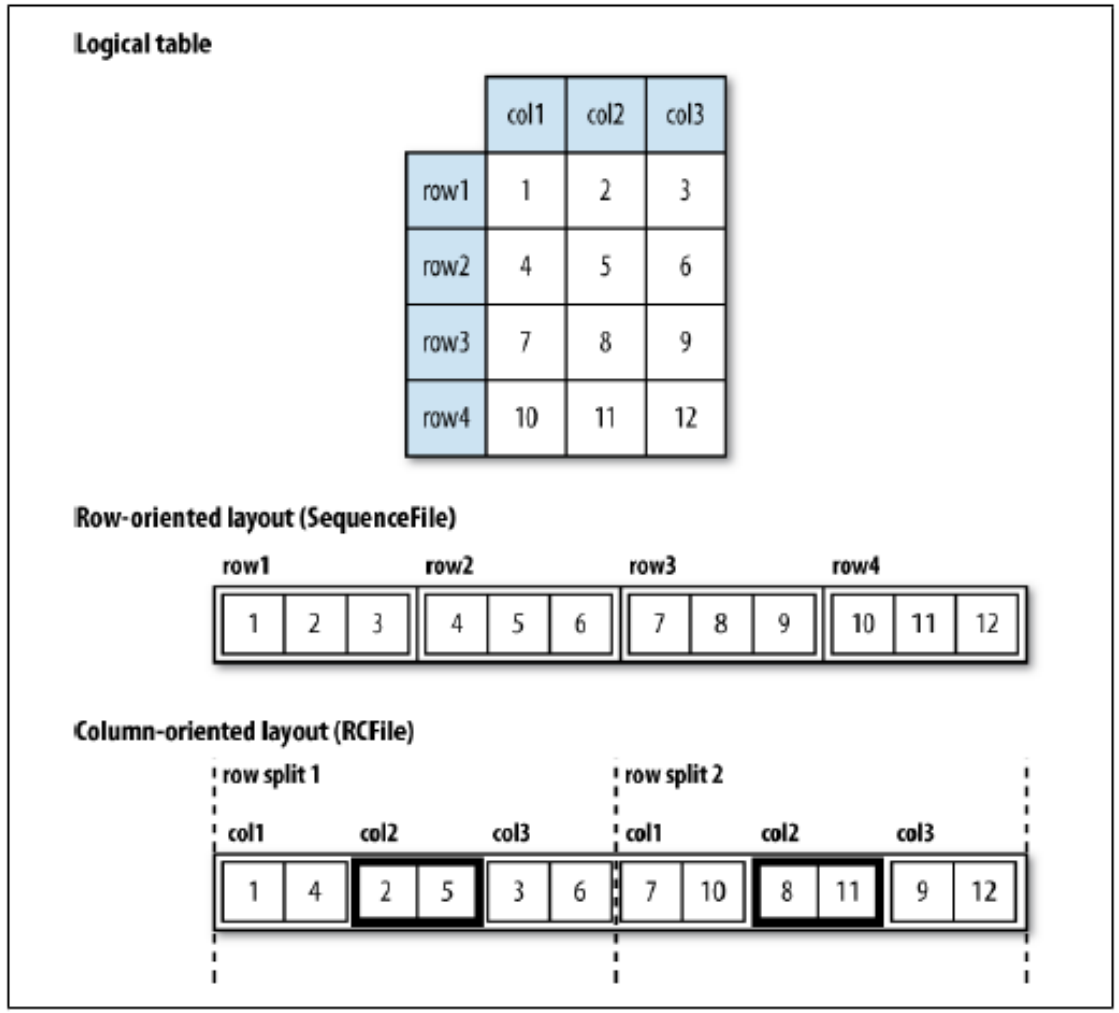


Figure 12-3. Row-oriented versus column-oriented storage

Plaintext Format

- When you create a table with no ROW FORMAT or STORED AS clauses, the default format is delimited text, with a row per line.
- The default row delimiter is not a tab character, but the Control-A character from the set of ASCII control codes since it is less likely to be a part of the field text than a tab character.
- The default collection item delimiter is a Control-B character, used to delimit items in an ARRAY or STRUCT, or key-value pairs in a MAP
- The default map key delimiter is a Control-C character, used to delimit the key and value in a MAP.
- Hive uses a SerDe called LazySimpleSerDe for this delimited format since it deserializes fields lazily—only as they are accessed.

Plaintext Format

Binary storage formats: Sequence files and RCFiles

Hadoop’s sequence file format is a general purpose binary format for sequences of records (key-value pairs).

You can use sequence files in Hive by using the declaration STORED AS SEQUENCEFILE in the CREATE TABLE statement.

one of the main benefits of using sequence files is their support for splitable compression.

Hive provides another binary storage format called RCFile, short for Record Columnar File.

In general, column-oriented formats work well when queries access only a small number of columns in the table.

Conversely, row-oriented formats are appropriate when a large number of columns of a single row are needed for processing at the same time.

Importing Data

- You can populate a table with data from another Hive table using an INSERT statement.
- At the creation time we can use the CTAS construct, which is an abbreviation used to refer to CREATE TABLE...AS SELECT

```
INSERT OVERWRITE TABLE target SELECT col1, col2 FROM source;
```

- For partitioned tables, you can specify the partition to insert into by supplying a PARTITION clause:

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2010-01-01')
SELECT col1, col2 FROM source;
```

Importing Data

- INSERT can be used to populate multiple tables when used with the FROM clause;

```
FROM records2

INSERT OVERWRITE TABLE stations_by_year

SELECT year, COUNT(DISTINCT station) GROUP BY year

 INSERT OVERWRITE TABLE records_by_year SELECT year, COUNT(1) GROUP BY year

INSERT OVERWRITE TABLE good_records_by_year SELECT year, COUNT(1) WHERE temperature != 9999 AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9) GROUP BY year
```

CREATE TABLE...AS SELECT

It’s often very convenient to store the output of a Hive query in a new table using CREATE TABLE...AS SELECT

The new table’s column definitions are derived from the columns retrieved by the SELECT clause.

```
CREATE TABLE target
 AS SELECT col1, col2
FROM source;
```

Altering Tables

You can rename a table using the ALTER TABLE statement: ALTER TABLE source RENAME TO target;

In addition to updating the table metadata, ALTER TABLE moves the underlying table directory so that it reflects the new name.

Hive allows you to change the definition for columns, add new columns, or even replace all existing columns in a table with a new set.

For example, consider adding a new column:

```
ALTER TABLE target ADD COLUMNS (col3 STRING);
```

The new column col3 is added after the existing columns.

Dropping the Tables

The DROP TABLE statement deletes the data and metadata for a table.

In the case of external tables, only the metadata is deleted—the data is left untouched.

If you want to delete all the data in a table, but keep the table definition then you can simply delete the data files.

For example: hive> dfs -rmr /user/hive/warehouse/my_table;

Querying Data

Sorting and Aggregating

Sorting data in Hive can be achieved by use of a standard ORDER BY clause

ORDER BY produces a result that is totally sorted, as expected, but to do so it sets the number of reducers to one, making it very inefficient for large datasets.

When a globally sorted result is not required we can use Hive's nonstandard extension, SORT BY.

SORT BY produces a sorted file per reducer.

Querying Data

If we want to control which reducer a particular row goes to, you can perform some subsequent aggregation using DISTRIBUTE BY clause.

```
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

Map reduce Scripts

The TRANSFORM, MAP, and REDUCE clauses make it possible to invoke an external script or program from Hive.

You can also copy archive files to your tasks, using the -archives option; these are unarchived on the task node. The -libjars option will add JAR files to the class path of the mapper and reducer tasks

Example 12-1. Python script to filter out poor quality weather records

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  (year, temp, q) = line.strip().split()
  if (temp != "9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

We can use the script as follows:

```
hive> ADD FILE /path/to/is_good_quality.py;
hive> FROM records2
    > SELECT TRANSFORM(year, temperature, quality)
    > USING 'is_good_quality.py'
    > AS year, temperature;
1949    111
1949    78
1950    0
1950    22
1950    -11
```

Map reduce Scripts

If we use a nested form for the query, we can specify a map and a reduce function.

```
FROM (
  FROM records2
  MAP year, temperature, quality
  USING 'is_good_quality.py'
  AS year, temperature) map_output
REDUCE year, temperature
USING 'max_temperature_reduce.py'
AS year, temperature;
```

Joins

- To combine records from two or more tables in the database we use JOIN clause.
- Inner joins The simplest kind of join is the inner join, where each match in the input tables results in a row in the output.
- In Hive, you can join on multiple columns in the join predicate by specifying a series of expressions, separated by AND keywords.

```
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
```

We can perform an inner join on the two tables as follows:

```
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
```

Outer Joins

- Outer joins allow you to find nonmatches in the tables being joined

```
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 supports right outer joins, which reverses the roles of the tables relative to the left join.

```
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
```

There is a full outer join, where the output has a row for each row from both tables in the join:

```
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
```

Semi Joins

- Hive doesn’t support IN subqueries (at the time of writing), but you can use a LEFT SEMI JOIN to do the same thing.

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

We can rewrite it as follows:

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

Map Joins

- Map joins If one table is small enough to fit in memory, then Hive can load the smaller table into memory to perform the join in each of the mappers

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

The job to execute this query has no reducers, so this query would not work for a RIGHT or FULL OUTER JOIN, since absence of matching can only be detected in an aggregating (reduce) step across all the inputs

Subqueries and Views

Subquery

A subquery is a SELECT statement that is embedded in another SQL statement.

Hive has limited support for subqueries, permitting a subquery in the FROM clause of a SELECT statement, or in the WHERE clause in certain cases.

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

Views

A view is a sort of “virtual table” that is defined by a SELECT statement. Views can be used to present data to users in a way that differs from the way it is actually stored on disk.

In Hive, a view is not materialized to disk when it is created; rather, the view’s SELECT statement is executed when the statement that refers to the view is run.

```
CREATE VIEW valid_records AS SELECT * FROM records2

WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9);
```

Functions

- When the query we want to create could not be compiled by its built-in functions, HIVE enables us to create and use our own User Defined Functions.

  UDFs have to be written in Java, the language that Hive itself is written in.

- For other languages, consider using a SELECT TRANSFORM query, which allows you to stream data through a user-defined script.

Three types of UDF in Hive:

- (regular) UDFs: operates on a single row and produces a single row as its output.

  Ex: Mathematical and String functions

- user-defined aggregate functions (UDAFs): works on multiple input rows and creates a single output row.

  Ex: Count, Max

- user-defined table-generating functions (UDTFs): operates on a single row and produces multiple rows—a table—as output.

  Ex:

- They differ in the number of rows that they accept as input and produce as output:

Writing a UDF

A UDF for stripping characters from the ends of strings

Example 12-2. A UDF for stripping characters from the ends of strings

```
package com.hadoopbook.hive;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class Strip extends UDF {
  private Text result = new Text();

  public Text evaluate(Text str) {
    if (str == null) {
      return null;
    }
    result.set(StringUtils.strip(str.toString()));
    return result;
  }

  public Text evaluate(Text str, String stripChars) {
    if (str == null) {
      return null;
    }
    result.set(StringUtils.strip(str.toString(), stripChars));
    return result;
  }
}
```

Executing UDF

A UDF must satisfy the following two properties:

1. A UDF must be a subclass of org.apache.hadoop.hive.ql.exec.UDF.
2. A UDF must implement at least one evaluate() method.

ADD JAR /path/to/hive-examples.jar;

CREATE TEMPORARY FUNCTION strip AS 'com.hadoopbook.hive.Strip';

hive> SELECT strip(' bee ') FROM dummy; bee hive> SELECT strip('banana', 'ab') FROM dummy; nan

Writing a UDAF

A UDAF for calculating the maximum of a collection of integers

```
package com.hadoopbook.hive;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.io.IntWritable;

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;
    }
  }
}
```

init()

The init() method initializes the evaluator and resets its internal state. In MaximumIntUDAFEvaluator, we set the IntWritable object holding the final result to null.

iterate()

The iterate() method is called every time there is a new value to be aggregated. The evaluator should update its internal state with the result of performing the aggregation.

terminatePartial()

The terminatePartial() method is called when Hive wants a result for the partial aggregation. The method must return an object that encapsulates the state of the aggregation.

merge()

The merge() method is called when Hive decides to combine one partial aggregation with another. The method takes a single object whose type must correspond to the return type of the terminatePartial() method.