CSE6331: Cloud Computing

Leonidas Fegaras

University of Texas at Arlington

Query Languages for Big Data

Based on:
Pig Latin and Hive, by M.Grossniklaus
http://datalab.cs.pdx.edu/education/clouddbms-win2014/notes/CloudDb-2014-Lect10-full.pdf
Beyond SQL: Speeding up Spark with DataFrame, by R. Xin
https://www.slideshare.net/databricks/spark-sqlsse2015public

Hive and Pig

- Need for High-Level Languages
  - Hadoop is great for large-data processing
  - But writing Java programs is verbose and slow
  - Not everyone likes to write Java code
- Pig: `pig.apache.org`
  - Large-scale data processing system
  - Scripts are written in Pig Latin, a dataflow language
  - Developed by Yahoo!, now open source at Apache
  - Roughly 1/3 of all Yahoo! Map-Reduce jobs
- Hive: `hive.apache.org`
  - Data warehousing application in Hadoop
  - The query language is a variant of SQL
  - Tables stored on HDFS as flat files
  - Developed by Facebook, now open source at Apache
  - Used for over 90% of Facebook Map-Reduce jobs

A Pig Latin Example

- SQL:

select e.ename, d.dname, e.address
from Employee **as** e, Department **as** d
where e.dno = d.dno

- Pig Latin script:

E = **load** *'e.txt'* **using PigStorage**(',') **as** (ename, dno, address);
D = **load** *'d.txt'* **using PigStorage**(',') **as** (dname, dno);
J = **join** E **by** dno, D **by** dno;
O = **foreach** J **generate** ename, dname, address;
store O **into** *'output'* **using PigStorage** (',');

Another Example

- SQL:

```
select category, avg(pagerank)
from urls
where pagerank > 0.2
group by category
having count(*) > 10 6
```

- Pig Latin script:

```
good_urls = filter urls by pagerank > 0.2;
groups = group good_urls by category;
big_groups = filter groups by count(good_urls) > 10 6 ;
output = foreach big_groups generate category, avg(good_urls.pagerank);
```

Pig Latin is a Dataflow Language

- Embodies “best of both worlds” approach
  - dataflow: a sequence of steps
  - similar to an imperative language
  - each step carries out a single data transformation
  - appealing to many developers
- High-level transformations
  - similar to specifying a query execution plan
  - bags in $\longrightarrow$ bag out
  - high-level operations render low-level manipulations unnecessary
  - offer a potential for optimization
- Data model
  - flexible, fully nested data model
  - extensive support for user-defined functions
  - ability to operate over plain input files without any schema
  - debugging environment to deal with enormous data sets

Pig Latin is a Dataflow Language

- Extensive support for user-defined functions (UDFs)
- Powerful execution framework
  - Pig Latin programs are executed using Pig
  - compiled into a workflow of Map-Reduce jobs
  - executed using Hadoop
- Pig only supports read-only data analysis of data sets
  - stored schemas are strictly optional
  - no need for time-consuming data import
  - user-provided function converts input into tuples
- Pig Latin has a flexible, fully nested data model
  - closer to how programmers think
  - many data already stored in nested fashion in source files
  - expressing processing tasks as sequences of steps, where each step performs a single transformation, requires a nested data model
  - eg, **group** returns a non-atomic result
- Pig Latin is geared towards Web-scale data
  - consists of a small set of operations that can easily be parallelized
  - inefficient evaluations have been deliberately excluded: non-equi-joins and correlated sub-queries

Implementation

- Pig Latin programs supply explicit sequence of operations, but are not necessarily executed in that order
- High-level relational-algebra-style operations enable traditional database optimization
- Lazy execution
  - processing is only triggered when **store** command is invoked
  - enables in-memory pipelining and filter reordering across multiple Pig Latin commands
- Logical query plan builder
  - checks that input files and bags being referred to are valid
  - builds a plan for every bag the user defines
  - is independent of data processing backend
- Physical query plan compiler
  - compiles a Pig Latin program into Map-Reduce jobs
- Two map-reduce jobs are required for the **order** command
  - first job samples input to determine statistics of sort key
  - map of second job range partitions input according to statistics
  - reduce of second job performs the sort

The Map-Reduce Barrier

- A Map-Reduce barrier is a part of a script that forces a reduce stage
- Similar to the Spark stage
- Some scripts can be done with just mappers

```
students = load 'students.txt'
           as (first:chararray, last:chararray, age:int, dept:chararray);
students_filtered = filter students by age >= 20;
students_proj = foreach students_filtered generate last, dept;
```

- But most will need the full Map-Reduce cycle

```
students = load 'students.txt'
           as (first:chararray, last:chararray, age:int, dept:chararray);
students_grouped = group students by dept;
students_proj = foreach students_grouped generate group, count(students);
```

- The **group** is the map-Reduce barrier which requires a reduce step
- Operators that cause a reduce stage: **group, cogroup, join, cross, order, distinct**

Data Model

- Atomic: a simple atomic value (chararray, int, float, ...)
- Tuple: sequence of fields
  - each field can be any of the data types
  - fields are referred to by positional notation or by name
  - positional notation is indicated with $: $0, $1, $2, ...

(**chararray** , (**chararray** , **int**))
(name: **chararray** , address : (street : **chararray** , number: **int**))
(*'Smith'*,(*'Main'*,35))
(name: *'Smith'*, address : (street : *'Main'*, number: 35))

- Bag: an inner collection of tuples with possible duplicates
  - does not require that every tuple contain the same number of fields or that the fields in the same position have the same type

(A: **int** , B: { (t1 : **int** , t2 : **int** , t3 : **int**) })
(3, { (1,2,3), (2,3,4) })

- Relation: A distributed dataset (an outer Bag)
- Map: a unique mapping from keys to values

(A: **int** , B: [charray : **int**])
(1, [*'M'*#2,*'N'*#3])

Data Model

$$t = \left(\text{'alice'}, \left\{ \begin{array}{c} (\text{'lakers'},\ 1) \\ (\text{'iPod'},\ 2) \end{array} \right\}, [\text{'age'} \rightarrow 20] \right)$$

Let fields of tuple `t` be called `f1, f2, f3`

Expression Type	Example	Value for `t`
Constant	'bob'	Independent of `t`
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Data Loading and Storing

- Syntax for loading from HDFS:

  alias = **load** '*file*' **using** function **as** schema;

- The file contents are converted into tuples using the deserializer function
- the schema is a Pig Latin type

  A = **load** '*e.txt*' **using** **PigStorage**(',')
  as (name: **chararray**, dno: **int** , adress : **chararray** , salary : **int**);

- To print the type of A:

  describe A;

- To print the tuples of A in the output:

  dump A;

- Syntax for storing data into HDFS:

  store alias **into** '*directory*' **using** function ;

- Example:

  store A **into** '*myoutput*' **using** **PigStorage**(',');

Pig Latin Transformations

- **group** groups together tuples that have the same group key
- Syntax:

  alias = **group** alias **by** expr, expr, ...;

- Example:

  B = **group** A **by** dno;

- It returns tuples of type:

(group: **int**, A: {(name:**chararray**,dno:**int**, adress:**chararray**, salary:**int**)})

Pig Latin Transformations (cont.)

- **foreach** applies some processing to every tuple of a data set
- Syntax:

  alias = **foreach** alias **generate** expr, expr, ...;

- Examples:

  C = **foreach** B **generate** group, **SUM**(A.salary);
  D = **foreach** B **generate** group, **flatten**(A);

- You may also process inner bags using the syntax:

  alias = **foreach** alias { alias = op; ... **generate** expr, expr, ... };

- Example:

  E = **foreach** B {
  F = **filter** A **by** salary > 10000;
  generate group, **SUM**(F.salary);
  };

Pig Latin Transformations (cont.)

- **join** performs an equijoin between two or more relations

  A = **load** *'data1'* **as** (owner: **chararray**, pet: **chararray**);
  B = **load** *'data2'* **as** (friend1 : **chararray**, friend2 : **chararray**);
  X = **join** A **by** owner, B **by** friend2 ;

- X has the type:

  X: {(A::owner: **chararray**, A::pet: **chararray**,
  B:: friend1 : **chararray**, B:: friend2 : **chararray**)}

- **cogroup** is a generalized join:

  Y = **cogroup** A **by** owner, B **by** friend2;

- Y has the type:

  Y: {(group: **chararray**,
  A: {(owner: **chararray**, pet: **chararray**)},
  B: {(friend1 : **chararray**, friend2 : **chararray**)})}

Pig Latin Transformations (cont.)

- **filter** selects tuples from a relation based on some condition

  X = **filter** A **by** (f1 == 8) **or** (**not** (f2+f3 > f1));

- **order** sorts a relation based on one or more fields

  X = **order** A **by** a3 **desc**;

- **distinct** removes duplicate tuples in a relation

  X = **distinct** A;

Hive

- Hive provides:
  - tools to enable easy data extract/transform/load (ETL)
  - a mechanism to impose structure on a variety of data formats
  - access to files stored either directly in HDFS or in other data storage systems such as Hbase, Cassandra, MongoDB, and Google Spreadsheets
  - a simple SQL-like query language (not a dataflow language like Pig Latin)
  - query execution via MapReduce
- Uses Metastore to store system catalogue and metadata about tables, columns, partitions etc.

Hive Example

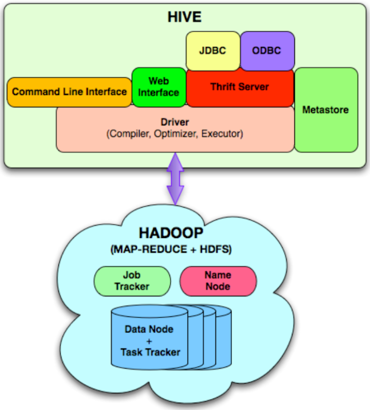
```
create table Employee (
  name string,
  dno int,
  address string)
row format delimited fields terminated by ',' stored as textfile;

create table Department (
  name string,
  dno int)
row format delimited fields terminated by ',' stored as textfile;

load data local inpath 'e.txt' overwrite into table Employee;
load data local inpath 'd.txt' overwrite into table Department;

select e.name, d.name
from Employee as e join Department as d on e.dno = d.dno;
```

Hive Architecture



- **Clients** use command line interface (CLI), Web UI, or JDBC/ODBC driver
- **HiveServer** provides Thrift and JDBC/ODBC interfaces
- **Metastore** stores system catalogue and metadata about tables, columns, partitions etc.
- **Driver** manages lifecycle of HiveQL statement as it moves through Hive

Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Data Model

- Unlike Pig Latin, schemas are **not** optional in Hive
- Hive structures data into well-understood database concepts like tables, columns, rows, and partitions
- Primitive types
  - **Integers**: bigint (8 bytes), int (4 bytes), smallint (2 bytes), tinyint (1 byte)
  - **Floating point**: float (single precision), double (double precision)
  - **String**

Complex Types

- Complex types
  - **Associative arrays**: map<key-type, value-type>
  - **Lists**: list<element-type>
  - **Structs**: struct<field-name: field-type, …>
- Complex types are templated and can be composed to create types of arbitrary complexity
  - `li list<map<string, struct<p1:int, p2:int>>`

Complex Types

- Complex types
  - **Associative arrays**: map<key-type, value-type>
  - **Lists**: list<element-type>
  - **Structs**: struct<field-name: field-type, …>
- Accessors
  - **Associative arrays:** m['key']
  - **Lists:** li[0]
  - **Structs:** s.field-name
- Example:
  - `li list<map<string, struct<p1:int, p2:int>>`
  - `t1.li[0]['key'].p1` gives the **`p1`** field of the struct associated with the **`key`** of the first array of the list **`li`**

Query Language

- HiveQL is a subset of SQL plus some extensions
  - from clause sub-queries
  - various types of joins: inner, left outer, right outer and outer joins
  - Cartesian products
  - group by and aggregation
  - union all
  - create table as select
- Limitations
  - only equality joins
  - joins need to be written using ANSI join syntax (not in **`WHERE`** clause)
  - no support for inserts in existing table or data partition
  - all inserts overwrite existing data

Query Language

- Hive supports user defined functions written in java
- Three types of UDFs
  - UDF: user defined function
    - Input: single row
    - Output: single row
  - UDAF: user defined aggregate function
    - Input: multiple rows
    - Output: single row
  - UDTF: user defined table function
    - Input: single row
    - Output: multiple rows (table)

Creating Tables

- Tables are created using the **CREATE TABLE** DDL statement
- Example:

```
CREATE TABLE t1(
    st string,
    fl float,
    li list<map<string, struct<p1:int, p2:int>>
);
```

- Tables may be partitioned or non-partitioned (we'll see more about this later)
- Partitioned tables are created using the **PARTITIONED BY** statement

```
CREATE TABLE test_part(c1 string, c2 string)
PARTITIONED BY (ds string, hr int);
```

Inserting Data

- Example

```
INSERT OVERWRITE TABLE t2
SELECT t3.c2, COUNT(1)
FROM t3
WHERE t3.c1 <= 20
GROUP BY t3.c2;
```

  - **OVERWRITE** (instead of **INTO**) keyword to make semantics of insert statement explicit
- Lack of **INSERT INTO**, **UPDATE**, and **DELETE** enable simple mechanisms to deal with reader and writer concurrency
- At Facebook, these restrictions have not been a problem
  - data is loaded into warehouse daily or hourly
  - each batch is loaded into a new partition of the table that corresponds to that day or hour

Inserting Data

- Hive supports inserting data into HDFS, local directories, or directly into partitions (more on that later)
- Inserting into HDFS

```
INSERT OVERWRITE DIRECTORY '/output_dir'
SELECT t3.c2, AVG(t3.c1)
FROM t3
WHERE t3.c1 > 20 AND t3.c1 <= 30
GROUP BY t3.c2;
```

- Inserting into local directory

```
INSERT OVERWRITE LOCAL DIRECTORY '/home/dir'
SELECT t3.c2, SUM(t3.c1)
FROM t3
WHERE t3.c1 > 30
GROUP BY t3.c2;
```

Inserting Data

- Hive supports inserting data into multiple tables/files from a single source given multiple transformations
- Example (corrected from paper):

```
FROM t1
    INSERT OVERWRITE TABLE t2
    SELECT t1.c2, count(1)
    WHERE t1.c1 <= 20
    GROUP BY t1.c2;

    INSERT OVERWRITE DIRECTORY '/output_dir'
    SELECT t1.c2, AVG(t1.c1)
    WHERE t1.c1 > 20 AND t1.c1 <= 30
    GROUP BY t1.c2;

    INSERT OVERWRITE LOCAL DIRECTORY '/home/dir'
    SELECT t1.c2, SUM(t1.c1)
    WHERE t1.c1 > 30
    GROUP BY t1.c2;
```

Loading Data

- Hive also supports syntax that can load the data from a file in the local files system directly into a Hive table where the input data format is the same as the table format
- Example:
  - Assume we have previously issued a **CREATE TABLE** statement for **page_view**

```
LOAD DATA INPATH '/user/data/pv_2008-06-08_us.txt'
INTO TABLE page_view
```

- Alternatively we can create a table directly from the file (as we will see a little bit later)

We Gotta Have Map/Reduce!

- HiveQL has extensions to express map-reduce programs
- Example

```
FROM (
  MAP doctext USING ‘python wc_mapper.py’
               AS (word, cnt)
  FROM docs CLUSTER BY word
) a
REDUCE word, cnt USING ‘python wc_reduce.py’;
```

- **MAP** clause indicates how the input columns are transformed by the mapper UDF (and supplies schema)
- **CLUSTER BY** clause specifies output columns that are hashed and distributed to reducers
- **REDUCE** clause specifies the UDF to be used by the reducers

We Gotta Have Map/Reduce!

- Distribution criteria between mappers and reducers can be fine tuned using **`DISTRIBUTE BY`** and **`SORT BY`**
- Example

```
FROM (
  FROM session_table
  SELECT sessionid,tstamp,data
  DISTRIBUTE BY sessionid SORT BY tstamp
) a
REDUCE sessionid, tstamp, data USING
'session_reducer.sh';
```

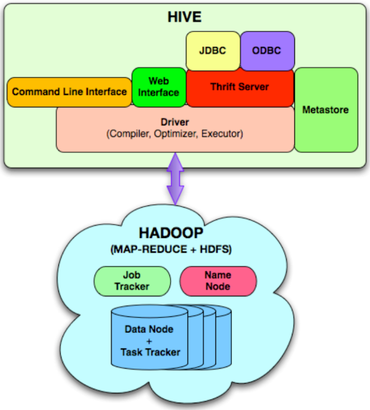
- If no transformation is necessary in the mapper or reducer the UDF can be omitted

We Gotta Have Map/Reduce!

```
FROM (
  FROM session_table
  SELECT sessionid,tstamp,data
  DISTRIBUTE BY sessionid SORT BY tstamp
) a
REDUCE sessionid, tstamp, data USING
'session_reducer.sh';
```

- Users can interchange the order of the **FROM** and **SELECT/MAP/REDUCE** clauses within a given subquery
- Mappers and reducers can be written in numerous languages

Hive Architecture



- **Clients** use command line interface, Web UI, or JDBC/ODBC driver
- **HiveServer** provides Thrift and JDBC/ODBC interfaces
- **Metastore** stores system catalogue and metadata about tables, columns, partitions etc.
- **Driver** manages lifecycle of HiveQL statement as it moves through Hive

Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Metastore

- Stores system catalog and metadata about tables, columns, partitions, etc.
- Uses a traditional RDBMS "as this information needs to be served fast"
- Backed up regularly (since everything depends on this)
- Needs to be able to scale with the number of submitted queries (we don't won't thousands of Hadoop workers hitting this DB for every task)
- Only Query Compiler talks to Metastore (metadata is then sent to Hadoop workers in XML plans at runtime)

Data Storage

- Table metadata associates data in a table to HDFS directories
  - **tables**: represented by a top-level directory in HDFS
  - **table partitions**: stored as a sub-directory of the table directory
  - **buckets**: stores the actual data and resides in the sub-directory that corresponds to the bucket's partition, or in the top-level directory if there are no partitions

- Tables are stored under the Hive root directory

```
CREATE TABLE test_table (…);
```

  - Creates a directory like

    <warehouse_root_directory>/test_table

    where <warehouse_root_directory> is determined by the Hive configuration

Partitions

- Partitioned tables are created using the **`PARTITIONED BY`** clause in the **`CREATE TABLE`** statement

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITIONED BY (ds string, hr int);
```

- Note that partitioning columns are not part of the table data
- New partitions can be created through an **`INSERT`** statement or an **`ALTER`** statement that adds a partition to a table

Partition Example

```
INSERT OVERWRITE TABLE test_part
PARTITION(ds='2009-01-01', hr=12)
SELECT * FROM t;
ALTER TABLE test_part
ADD PARTITION(ds='2009-02-02', hr=11);
```

- Each of these statements creates a new directory
  - /…/test_part/ds=2009-01-01/hr=12
  - /…/test_part/ds=2009-02-02/hr=11
- HiveQL compiler uses this information to prune directories that need to be scanned to evaluate a query

```
SELECT * FROM test_part WHERE ds='2009-01-01';
SELECT * FROM test_part
WHERE ds='2009-02-02' AND hr=11;
```

Buckets

- A bucket is a file in the leaf level directory of a table or partition
- Users specify number of buckets and column on which to bucket data using the **CLUSTERED BY** clause

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITIONED BY (ds string, hr int)
CLUSTERED BY (c1) INTO 32 BUCKETS;
```

Buckets

- Bucket information is then used to prune data in the case the user runs queries on a sample of data
- Example:

```
SELECT * FROM test_part TABLESAMPLE (2 OUT OF 32);
```

  – This query will only use 1/32 of the data as a sample from the second bucket in each partition

Serialization/Deserialization (SerDe)

- Tables are serialized and deserialized using serializers and deserializers provided by Hive or as user defined functions
- Default Hive SerDe is called the LazySerDe
  - Data stored in files
  - Rows delimited by newlines
  - Columns delimited by ctrl-A (ascii code 13)
  - Deserializes columns lazily only when a column is used in a query expression
  - Alternate delimiters can be used

```
CREATE TABLE test_delimited(c1 string, c2 int)
   ROW FORMAT DELIMITED
      FIELDS TERMINATED BY '\002'
      LINES TERMINATED BY '\012';
```

Additional SerDes

- Facebook maintains additional SerDes including the RegexSerDe for regular expressions
- RegexSerDe can be used to interpret apache logs

```
add jar 'hive_contrib.jar';
CREATE TABLE apachelog(host string,
  identity string,user string,time string,
  request string,status string,size string,
  referer string,agent 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$s %8$s %9$s‘
);
```

Custom SerDes

- Legacy data or data from other applications is supported through custom serializers and deserializers
  - SerDe framework
  - **`ObjectInspector`** interface
- Example

```
ADD JAR /jars/myformat.jar
CREATE TABLE t2
ROW FORMAT SERDE 'com.myformat.MySerDe';
```

File Formats

- Hadoop can store files in different formats (text, binary, column-oriented, …)
- Different formats can provide performance improvements
- Users can specify file formats in Hive using the **`STORED AS`** clause
  - Example:

```
CREATE TABLE dest1(key INT, value STRING)
 STORED AS
  INPUTFORMAT
   'org.apache.hadoop.mapred.SequenceFileInputFormat'
  OUTPUTFORMAT
   'org.apache.hadoop.mapred.SequenceFileOutputFormat`
```

- File format classes can be added as jar files in the same fashion as custom SerDes

External Tables

- Hive also supports using data that does not reside in the HDFS directories of the warehouse using the **EXTERNAL** statement
  - Example:

```
CREATE EXTERNAL TABLE test_extern(c1 string, c2 int)
LOCATION '/user/mytables/mydata';
```

- If no custom SerDes the data in the 'mydata' file is assumed to be Hive's internal format
- Difference between external and normal tables occurs when **DROP** commands are performed
  - Normal table: metadata is dropped from Hive catalogue and data is dropped as well
  - External table: only metadata is dropped from Hive catalogue, no data is deleted

Custom Storage Handlers

- Hive supports using storage handlers besides HDFS
  - e.g. HBase, Cassandra, MongoDB, …
- A storage handler builds on existing features
  - Input formats
  - Output formats
  - SerDe libraries
- Additionally storage handlers must implement a metadata interface so that the Hive metastore and the custom storage catalogs are maintained simultaneously and consistently

Custom Storage Handlers

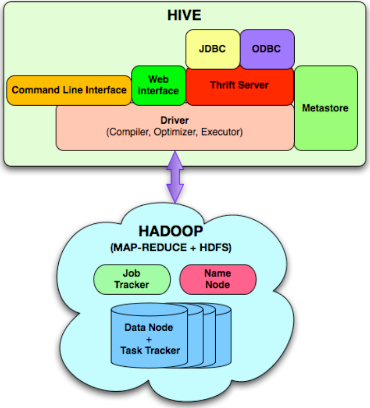
- Hive supports using custom storage and HDFS storage simultaneously
- Tables stored in custom storage are created using the **STORED BY** statement
  - Example:

```
CREATE TABLE hbase_table_1(key int, value string)
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler‘;
```

Custom Storage Handlers

- As we saw earlier Hive has *normal (managed)* and *external* tables
- Now we have *native* (stored in HDFS) and *non-native* (stored in custom storage) tables
- non-native may also use external tables
- Four possibilities for base tables
  - managed native: **`CREATE TABLE …`**
  - external native: **`CREATE EXTERNAL TABLE …`**
  - managed non-native: **`CREATE TABLE … STORED BY …`**
  - external non-native: **`CREATE EXTERNAL TABLE … STORED BY …`**

Hive Architecture



- **Clients** use command line interface, Web UI, or JDBC/ODBC driver
- **HiveServer** provides Thrift and JDBC/ODBC interfaces
- **Metastore** stores system catalogue and metadata about tables, columns, partitions etc.
- **Driver** manages lifecycle of HiveQL statement as it moves through Hive

Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Query Compiler

- Parses HiveQL using Antlr to generate an abstract syntax tree
- Type checks and performs semantic analysis based on Metastore information
- Naïve rule-based optimizations
- Compiles HiveQL into a directed acyclic graph of MapReduce tasks

Optimizations

- Column Pruning
  - Ensures that only columns needed in query expressions are deserialized and used by the execution plan
- Predicate Pushdown
  - Filters out rows in the first scan if possible
- Partition Pruning
  - Ensures that only partitions needed by the query plan are used

Optimizations

- Map side joins
  - If one table in a join is very small it can be replicated in all of the mappers and joined with other tables
  - User must know ahead of time which are the small tables and provide hints to Hive

```
SELECT /*+ MAPJOIN(t2) */ t1.c1, t2.c1
FROM t1 JOIN t2 ON(t1.c2 = t2.c2);
```

- Join reordering
  - Smaller tables are kept in memory and larger tables are streamed in reducers ensuring that the join does not exceed memory limits

Optimizations

- GROUP BY repartitioning
  - If data is skewed in GROUP BY columns the user can specify hints like MAPJOIN

```
set hive.groupby.skewindata=true;
SELECT t1.c1, sum(t1.c2)
FROM t1
GROUP BY t1;
```

- Hashed based partial aggregations in mappers
  - Hive enables users to control the amount of memory used on mappers to hold rows in a hash table
  - As soon as that amount of memory is used, partial aggregates are sent to reducers.

Spark Query Languages

- The Spark optimizer does not understand
  - the functional parameters of the RDD methods
  - the structure of the data in RDDs
  - the semantics of user functions
- Limited optimizations
- Example: Spark cannot move a filter before a join because it cannot look at the code of the condition function to see if this is allowed
- SHARK: Hive on Spark
  - It can only be used to query external data in Hive catalog – limited data sources
  - It can only pass the SQL query as a string to Spark – error prone
  - The Hive optimizer is tailored for MapReduce – difficult to extend

Spark SQL

- Part of the core distribution since 2014
- Runs SQL/HiveQL queries, optionally alongside or replacing existing Hive deployments
  - Allows creating and running Spark programs faster
  - Write less code
  - Read less data
  - Let the optimizer do the hard work

select count(*)
from hiveTable
where udf(data) > 100

Programming Interface

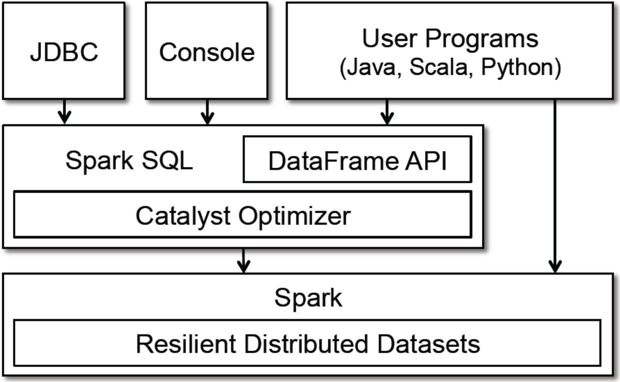


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

DataFrame

- A distributed collection of rows organized into named columns
- An abstraction for selecting, filtering, aggregating and plotting structured data
- Write less code:
  - The Spark SQL's DataSource API can read and write DataFrames using a variety of formats
  - JSON, built-in external JDBC, etc
- High-level operations
- Common operations can be expressed concisely as calls to the DataFrame API:
  - select required columns
  - join different data sources
  - aggregation (count, sum, average, etc)
  - filtering
- Read less data:
  - The fastest way to process big data is to never read it

DataFrames Data Model and Operations

- Nested data model
- Supports:
  - primitive SQL types (boolean, integer, double, decimal, string, data, timestamp)
  - complex types (structs, arrays, maps, and unions)
  - user defined types
- First class support for complex data types
- Relational operations (select, where, join, groupBy) via a special syntax
- Operators take expression objects (abstract syntax trees, not values)
- Operators build up an abstract syntax tree, which is then optimized by Catalyst
- Schema Inference:
  - Spark SQL can automatically infer the schema of these objects using reflection

Example

- Compute averages using SQL:

```
select  name, avg(age)
from people
group by name
```

- Using Spark RDDs:

```
data = sc. textFile  (...).  split (",")
data.map( x => (p(0),(p(1),1)) )
     .reduceByKey{ case (x,y) => (x(0)+y(0),x(1)+y(1)) }
     .map{ case (x,(s,c))  => (x,s/c) }
     . collect ()
```

- Using DataFrames:

```
ctx. table ("people")
    .groupBy("name")
    .agg("name", avg("age"))
    . collect ()
```

Another Example

- SQL:

```
select departments.id, departments.name, count(name)
from employees, departments
where employees.deptId = departments.id
group by departments.id, departments.name
```

- DataFrames:

```
employees.join(departments, employees("deptId") === departments("id"))
         .where(employees("gender") === "female")
         .groupBy(departments("id"), departments("name"))
         .agg(count("name"))
```

- Alternatively, register a DataFrame as temp SQL table and write a traditional SQL query as a string

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("select count(*), avg(age) from young")
```

User-Defined Functions (UDFs)

- DataFrames support few operations, but it's easy to add new ones
- Allows inline registration of UDFs
- Compare with Pig, which requires the UDF to be written in a Java package thats loaded into the Pig script
- Can be defined on simple data types or entire tables
- UDFs are available after registration

```
ctx.udf.register("predict",
                 (x: Float, y: Float) => model.predict(Vector(x,y)))
```

Nested Query

```
case class X ( A: Int , D: Int )
case class Y ( B: Int , C: Int )
```

main program:

```
val sc = new SparkContext(sparkConf)
val spark = SparkSession.builder().config(sparkConf).getOrCreate()

val XC = spark.sparkContext.textFile( xfile )
              .map(_.split(",")).map(n => X(n(0).toInt,n(1).toInt))
              .toDF()
val YC = spark.sparkContext.textFile( yfile )
              .map(_.split(",")).map(n => Y(n(0).toInt,n(1).toInt))
              .toDF()
```

Nested Query (cont.)

```
XC.createOrReplaceTempView("X")
YC.createOrReplaceTempView("Y")

spark.sql("""
          SELECT x.A
          FROM X x
          WHERE x.D IN (SELECT y.C FROM Y y WHERE x.A=y.B)
          """)
```

KMeans Clustering

```
case class Point ( X: Double, Y: Double )

var centroids
  = spark.sparkContext.textFile( centroid_file )
         .map(_.split(",")).map(n => Point(n(0).toDouble,n(1).toDouble))
         .collect()

sqlContext.udf.register("get_closest_centroid",
     ( x: Double, y: Double ) => { val p = new Point(x,y)
                                   centroids.map(c => (distance(p,c),c))
                                            .sortBy(_._1).head._2 })

val points = spark.sparkContext.textFile( point_file )
                  .map(_.split(","))
                  .map(n => Point(n(0).toDouble,n(1).toDouble))
                  .toDF()
```

KMeans Clustering (cont.)

```
points.createOrReplaceTempView("points")

for (i <- 1 to iterations) {
    centroids = spark.sql("""
                    SELECT AVG(p.X), AVG(p.Y)
                    FROM points p
                    GROUP BY get_closest_centroid(p.X,p.Y)
                """).rdd.map { case Row(x:Double,y:Double) => Point(x,y) }
                    .collect()
}
```

PageRank

```
case class Edge ( src: Int, dest: Int )
case class Rank ( id: Int, degree: Int, rank: Double )

def rank ( sums: Long, counts: Long ): Double
        = (1-alpha)+alpha*sums/counts
val edges = spark.sparkContext.textFile ( input_file )
              .map(_.split (",")).map(n => Edge(n(0).toInt,n(1).toInt))
              .toDF()
var nodes = edges.groupBy("src").agg(count("dest").as("degree"))
                 .withColumn("rank",lit(1-alpha))
                 .withColumnRenamed("src","id")
for ( i <- 1 to 10 ) {
  val newranks = nodes.join(edges,nodes("id")===edges("src"))
                      .groupBy("dest").agg(udf(rank _)
                      .apply(sum("rank"),sum("degree")).as("newrank"))
  nodes = newranks.join(nodes,nodes("id")===newranks("dest"))
                  .drop("rank","dest")
                  .withColumnRenamed("newrank","rank")
}
```