Hive

Riccardo Torlone

Università Roma Tre

Credits: Dean Wampler (thinkbiganalytics.com)

Motivation

- Analysis of data made by both engineering and non-engineering people.
- The data are growing fast.
- Current RDBMS can NOT handle it.
- Traditional solutions are often not scalable, expensive and proprietary.

Motivation

- Hadoop supports data-intensive distributed applications.
- But…
  - You have to use MapReduce model
    - Hard to program
    - Not Reusable
    - Error prone
  - For complex jobs: multiple stage of MapReduce jobs
  - Alternative and more efficient tools exist today (e.g., Spark) but they are not easy to use
  - Most users know Java/SQL/Bash

Possible solution

- Make the unstructured data looks like tables regardless how it really lay out
- SQL (standard!) based query can be directly against these tables
- Generate specify execution plan for this query

- Hive!

- A big data management system storing structured data on Hadoop file system
- Provide an easy query these data by executing Hadoop-based plans
- Today just a part of a large category of solutions called "SQL over Hadoop"

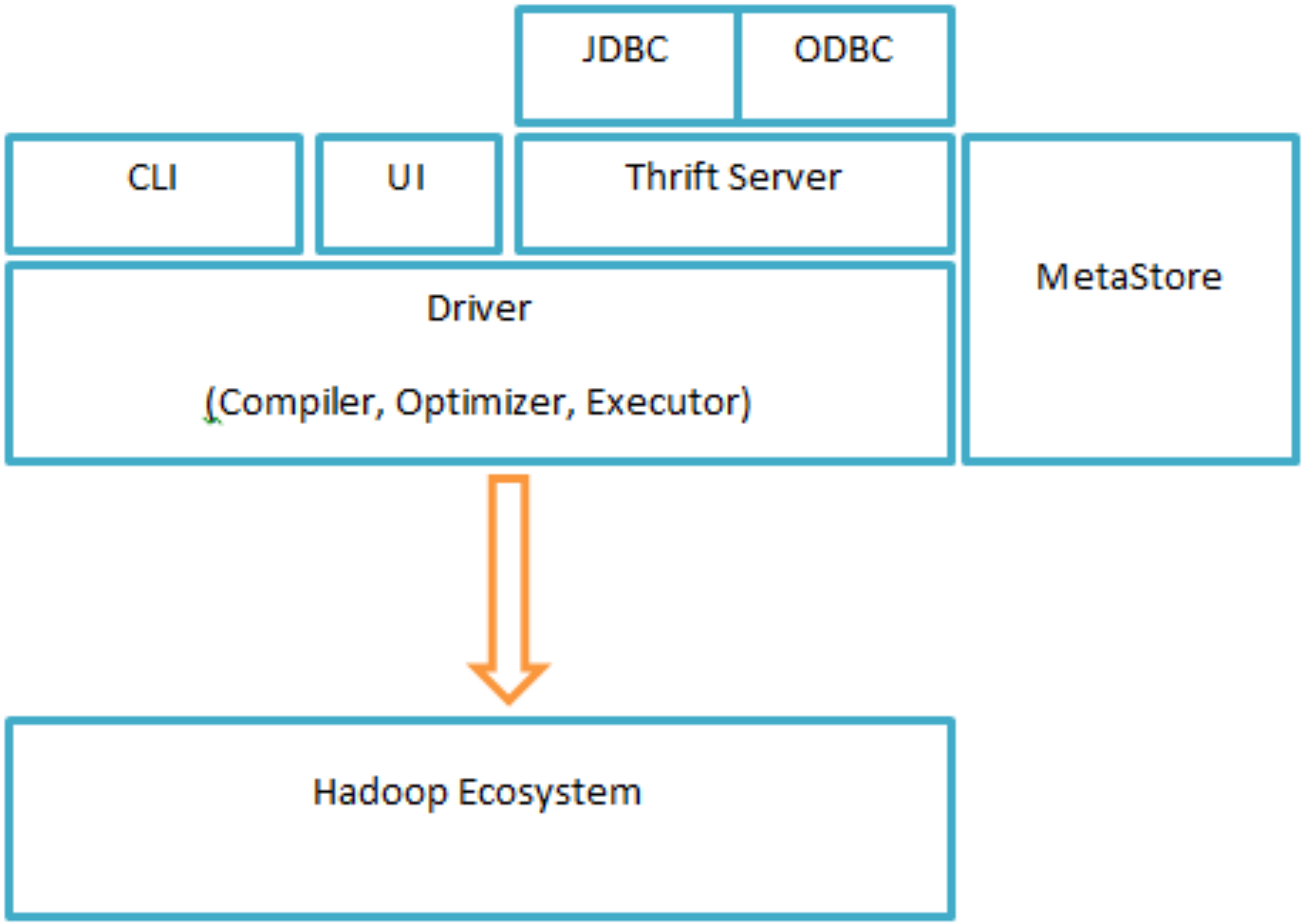
What is Hive?

- An infrastructure built on top of Hadoop for providing data summarization, query, and analysis.
  - Structure
  - Access to different storage
  - HiveQL (very close to a subset of SQL)
  - Query execution via MapReduce, Tez, and Spark
- Key Building Principles:
  - SQL is a familiar language
  - Extensibility – Types, Functions, Formats, Scripts
  - Performance

Application scenario

- No real-time queries (high latency)!
- No row level updates!
- Not designed for online transaction processing!
- Best use: batch jobs over large sets of append-only data
  - Log processing
  - Data/Text mining
  - Business intelligence
  - …
- However: current version allows a form of ACID transaction at the row level (one application can add rows while another reads from the same partition without interfering with each other).

Architecture



Data Units

- Databases
  - Containers of tables and other data units
- Tables
  - Homogeneous units of data which have the same schema.
  - Basic type columns (Int, Float, Boolean)
  - Complex type: Lists / Maps / Arrays
- Partitions
  - Each Table can have one or more partition columns (or partition keys).
  - Each unique value of the partition keys defines a partition of the Table.
  - Queries can run on the relevant partition thereby speeding up the analysis significantly.
  - Partition columns are virtual columns, they are not part of the data itself
- Buckets (or Clusters)
  - Data in each partition may be divided into Buckets based on the value of a hash function of some column of the Table.
  - These can be used to efficiently sample the data

Type System

- Primitive types
  - Integers: TINYINT, SMALLINT, INT, BIGINT
  - Boolean: BOOLEAN
  - Floating point numbers: FLOAT, DOUBLE
  - String: STRING
  - Date string: TIMESTAMP
- Complex types
  - Structs: {a INT; b INT}
  - Maps: M['group']
  - Arrays: ['a', 'b', 'c'], A[1] returns 'b'

Examples

```
CREATE TABLE demo1(
        id INT,
        name STRING);

CREATE TABLE employees (
        name STRING,
        salary FLOAT,
        subordinates ARRAY<STRING>,
        deductions MAP<STRING, FLOAT>,
        address STRUCT<street:STRING, city:STRING,
                       state:STRING, zip:INT>
);
```

Terminators

`'\n'`	Between rows (records)
`^A ('\001')`	Between fields (columns)
`^B ('\002')`	Between ARRAY and STRUCT elements and MAP key-value pairs
`^C ('\003')`	Between each MAP key and value

The actual file format

```
CREATE TABLE employees (
        name STRING,
        salary FLOAT,
        subordinates ARRAY<STRING>,
        deductions MAP<STRING, FLOAT>,
        address STRUCT<street:STRING, city:STRING,
                       state:STRING, zip:INT> )
```

John Doe^A100000.0^AMary Smith^BTodd Jones^AFederal Taxes^C.2^BState Taxes^C.05^BInsurance^C.1^A1 Michigan Ave.^BChicago^BIL^B60600\n

Partitioning

```
CREATE TABLE message_log (
        status STRING, msg STRING, hms STRING )
        PARTITIONED BY ( year INT, month INT, day INT );
```

- The partition column is virtual
- Separate directories for each partition column
- On disk:

```
message_log/year=2015/month=01/day=01/
...
message_log/year=2015/month=01/day=31/
message_log/year=2015/month=02/day=01/
...
message_log/year=2015/month=12/day=31/
```

Advantages of partitioning

- Speed queries by limiting scans to the correct partitions specified in the WHERE clause:

```
SELECT * FROM message_log
WHERE year = 2015 AND
        month = 01 AND
        day = 31;
```

Query execution with partitioning

SELECT * FROM message_log;

- ALL these directories are read.

```
message_log/year=2015/month=01/day=01/
…
message_log/year=2015/month=01/day=31/
message_log/year=2015/month=02/day=01/
…
message_log/year=2015/month=12/day=31/
```

SELECT * FROM message_log
WHERE year = 2015 AND month = 01;

- Just 31 directories are read:

```
message_log/year=2015/month=01/day=01/
…
message_log/year=2015/month=01/day=31/
message_log/year=2015/month=02/day=01/
…
message_log/year=2015/month=12/day=31/
```

Other DDL Operations

```
CREATE TABLE sample (foo INT, bar STRING)
                    PARTITIONED BY (ds STRING);

SHOW TABLES 's*';

DESCRIBE sample;

ALTER TABLE sample ADD COLUMNS (new_col INT);

DROP TABLE sample;
```

Clustering

```
CREATE TABLE sales (
        id INT, items ARRAY<STRUCT<id:INT, name:STRING>> )
PARITIONED BY (ds STRING)
CLUSTERED BY (id) INTO 32 BUCKETS;

SELECT id FROM sales TABLESAMPLE (BUCKET 1 OUT OF 32)
```

External tables

- Normally tables are in HDFS
- When you want to manage the data by yourself: external table (Hive does not use a default location for this table)

```
CREATE EXTERNAL TABLE employees (
        name STRING,
        ...)
LOCATION '/data/employees/input';
```

- We own and manage that directory (this comes in handy if you already have data generated).
- LOCATION is a directory: Hive will read all the files it contains.
- The table data are not deleted when you drop the table.
- The table metadata are deleted from the Metastore.

Locations

- The locations can be local, in HDFS, or in S3.
- Joins can join table data from any such source!

```
LOCATION ‘file://path/to/data';... ...
LOCATION 'hdfs://server:port/path/to/data'; ...
LOCATION 's3n://mybucket/path/to/data';
```

Loading data

- Loading a file that contains two columns separated by ctrl-a into sample table:

```
LOAD DATA LOCAL INPATH './sample.txt'
OVERWRITE INTO TABLE sample PARTITION (ds='2015-02-24');
```

- Loading from HDFS:

```
LOAD DATA INPATH '/user/hive/sample.txt'
OVERWRITE INTO TABLE sample PARTITION (ds='2015-02-24');
```

- Loading from CSV:

```
LOAD DATA LOCAL INPATH './sample.txt'
OVERWRITE INTO TABLE sample PARTITION (ds='2015-02-24')
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

Create and import

```
CREATE LOCAL TABLE sample (foo INT, bar STRING)
                PARTITIONED BY (ds STRING)
        ROW FORMAT DELIMITED FIELDS
        TERMINATED BY ',' STORED AS TEXTFILE
        location './sample.txt';
```

Select statements

```
SELECT ymd, symbol FROM stocks
WHERE exchange = 'NASDAQ' AND symbol = 'AAPL' ;
```

- Queries involving projection require a MR job

```
SELECT * FROM stocks
WHERE exchange = 'NASDAQ' AND symbol = 'AAPL' ;
```

- If a * query is over partitions: no MR job is required!
- A * query without the WHERE clause does not require MR as well

Storing the results

- select column 'foo' from all rows of partition ds=2015-02-24:

```
SELECT foo FROM sample  WHERE ds='2015-02-24';
```

- store the result into a local directory:

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/hdfs_out'
SELECT * FROM sample WHERE ds='2015-02-24';
```

- store the result in HDFS:

```
INSERT OVERWRITE DIRECTORY '/tmp/hive-sample-out'
SELECT * FROM sample;
```

Aggregations and groups

```
SELECT count(*) FROM stocks
WHERE exchange = 'NASDAQ' AND symbol = 'AAPL' ;

SELECT avg(price_close) FROM stocks
WHERE exchange = 'NASDAQ' AND symbol = 'AAPL' ;

SELECT year(ymd), avg(price_close)
FROM stocks
WHERE exchange = 'NASDAQ' AND symbol = 'AAPL' ;
GROUP BY year(ymd);
```

Aggregations and Groups

- get the max value of foo.

```
SELECT MAX(foo) FROM sample;
```

- groups the ds, sums the foo values for a given ds and count the amount of row for the given ds.

```
SELECT ds, COUNT(*), SUM(foo) FROM sample  GROUP BY ds;
```

- insert the output into a table.

```
INSERT OVERWRITE TABLE bar
SELECT s.bar, COUNT(*)
FROM sample s
WHERE s.foo > 0 GROUP BY s.bar;
```

Joins

```
SELECT s.ymd, s.symbol, s.price_close, d.dividend
FROM stocks s JOIN dividends d
ON s.ymd = d.ymd AND s.symbol = d.symbol
WHERE s.ymd > '2015-01-01';
```

- Only equality (x = y) conditions allowed
- Put the biggest table last!
- Reducer will stream the last table and buffer the others.

Join examples

```
CREATE TABLE customer (id INT, name STRING, address STRING);
CREATE TABLE order_cust (id INT, cus_id INT, prod_id INT, price INT);

SELECT * FROM customer c JOIN order_cust o ON (c.id=o.cus_id);

SELECT c.id, c.name, c.address, ce.exp
FROM customer c
JOIN ( SELECT cus_id, sum(price) AS exp
        FROM order_cust GROUP BY cus_id ) ce
ON (c.id=ce.cus_id);
```

Types of Join

- Four kinds supported:
  - Inner Joins
  - Outer Joins (left, right, full)
  - Semi Joins (not discussed here)
  - Map-side Joins (an optimization of others).

An example of outer join

```
SELECT s.ymd, s.symbol, s.price_close, d.dividend
FROM ( SELECT ymd, symbol, price_close
        FROM stocks
        WHERE exchange = 'NASDAQ' AND symbol = 'AAPL‘ ) s LEFT
OUTER JOIN (
        SELECT ymd, symbol, dividend
        FROM dividends
        WHERE exchange = 'NASDAQ' AND symbol = 'AAPL‘ ) d
ON s.ymd = d.ymd AND s.symbol = d.symbol;
```

Map-side Joins

- Join tables in the mapper.
- Optimization that eliminates the reduce step.
- Useful if all but one table is small.

```
SELECT s.ymd, s.symbol, s.price_close, d.dividend
FROM dividends d JOIN stocks s
ON s.ymd = d.ymd AND s.symbol = d.symbol;
```

- If all but one table is small enough, the mapper can load the small tables in memory and do the joins there, rather than invoking an expensive reduce step.
- The optimization is automatic if: `set hive.auto.convert.join = true;`
- Can't be used with RIGHT/FULL OUTER joins.

Built-in Functions

- Works on a single row.
- Mathematical: round, floor, ceil, rand, exp...
- Collection: size, map_keys, map_values, array_contains
- Type Conversion: cast
- Date: from_unixtime, to_date, year, datediff…
- Conditional: if, case, coalesce
- String: length, reverse, upper, trim…

```
hive> SHOW FUNCTIONS;
!
!=
...
abs
acos
...
```

Built-in Functions

hive> DESCRIBE FUNCTION year;
year(date) - Returns the year of date
hive> DESCRIBE FUNCTION EXTENDED year;
year(date) - Returns the year of date
date is a string in the format of 'yyyy-MM-dd HH:mm:ss'
or 'yyyy-MM-dd'.
Example:

> SELECT year('2009-03-07') FROM src LIMIT 1;
2009

Examples

```
SELECT year(ymd) FROM stocks;

SELECT year(ymd), avg(price_close) FROM stocks
WHERE symbol = 'AAPL'
GROUP BY year(ymd);
```

Table Generating Function

```
SELECT explode(subordinates) AS
subs FROM employees;
```

- Generates zero or more output rows for each input row.
- Takes an array (or a map) as an input and outputs the elements of the array (map) as separate rows.
- Effectively a new table.

- More flexible way to use TGFs:

```
SELECT name, sub
FROM employees
LATERAL VIEW explode(subordinates) subView AS sub;
```

Example

pageAds

pageid (string)	adid_list (Array<int>)
"front_page"	[1, 2, 3]
"contact_page"	[3, 4, 5]

```
SELECT pageid, adid
FROM pageAds LATERAL VIEW explode(adid_list) subA AS adid;
```

subA

pageid (string)	adid (int)
"front_page"	1
"front_page"	2
...	...

User-defined function (UDF)

```
// Java
import org.apache.hadoop.hive.ql.exec.UDF;
        public class NowUDF extends UDF {
                public long evaluate() {
                        return System.currentTimeMillis();
                }
        }
```

- You compile this Java code and build a jar file…

UDF usage

- … then:
  - include the jar in the HIVE_CLASSPATH using ADD JAR
  - create a TEMPORARY FUNCTION,
  - profit!

```
-- HQL
ADD JAR path_to_jar;
...
CREATE TEMPORARY FUNCTION now AS 'com...NowUDF';
SELECT epoch_millis FROM ...
WHERE epoch_millis < now() ...;
```

Another example

- Java code

```
package com.example.hive.udf;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;
public class Lower extends UDF {
  public Text evaluate(final Text s) {
    if (s == null) { return null; }
    return new Text(s.toString().toLowerCase());
  }
}
```

- Registering the class

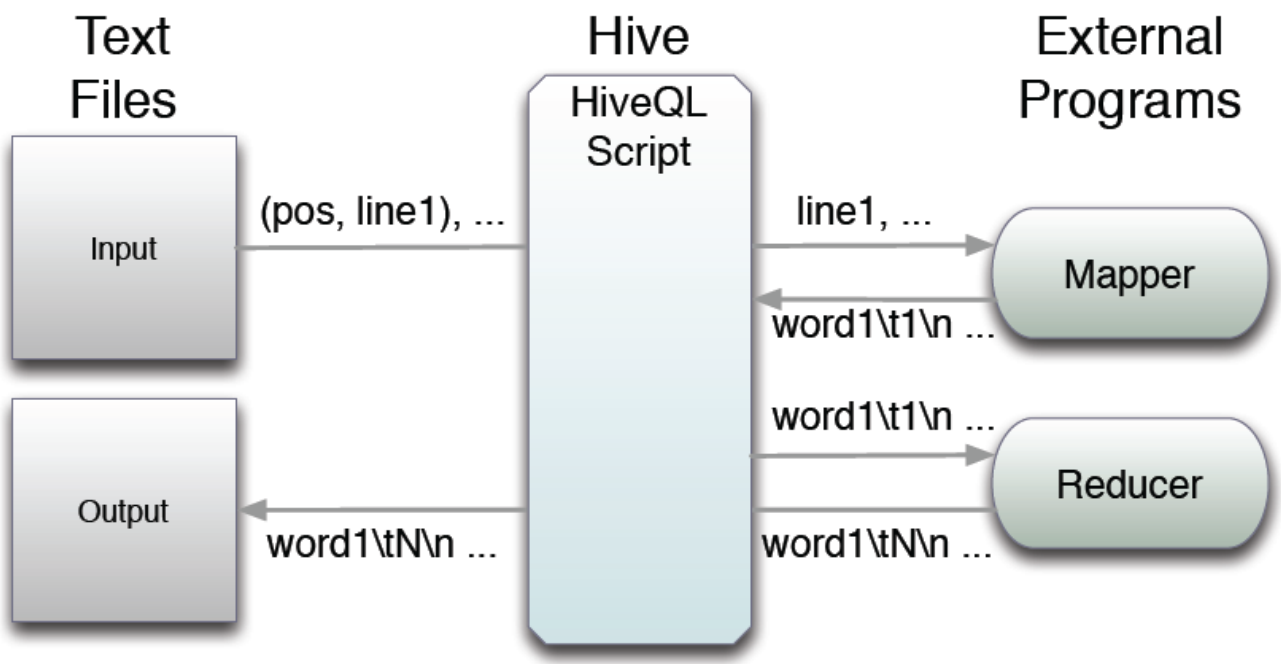
```
CREATE TEMPORARY FUNCTION my_lower AS 'com.example.hive.udf.Lower';
```

- Using the function

```
SELECT my_lower(title), sum(freq)
FROM titles GROUP BY my_lower(title);
```

Calling external Map/Reduce Scripts

- A technique for calling out to external programs to perform map and reduce operations.



- We'll use mapper and reducer scripts written in Python to compute the Word Count for Shakespeare's plays.

Mapper

```
# mapper.py
import sys

for line in sys.stdin:
        words = line.strip().split()
        for word in words:
                print "%s\t1" % (word.lower())
```

- For non-Python users:
  - We read input from “stdin” (“standard in”), a line at a time.
  - For each line, we “strip” whitespace off the beginning and end, then “split” the string on whitespace to create a list of words.
  - Finally, we iterate through the words and print the word in lower case, followed by a tab character, followed by the number 1.

Reducer

```
# reducer.py
import sys
(last_key, last_count) = (None, 0)
for line in sys.stdin:
        (key, count) = line.strip().split("\t")
        if last_key and last_key != key:
                print "%s\t%d" % (last_key, last_count)
                (last_key, count) = (key, int(count))
        else:
                last_key = key
                last_count += int(count)

if last_key:
        print "%s\t%d" % (last_key, last_count)
```

Calling external Map/Reduce Scripts

```
CREATE EXTERNAL TABLE shakespeares_plays (line STRING)
LOCATION '/data/shakespeare/input';

CREATE TABLE shakespeares_plays_wc ( word STRING, count INT)
ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t';

ADD FILE /.../mapper.py;
ADD FILE /.../reducer.py;
FROM ( FROM shakespeares_plays
       MAP line USING 'mapper.py'
       AS word, count
       CLUSTER BY word) wc
INSERT OVERWRITE TABLE shakespeares_plays_wc
REDUCE wc.word, wc.count USING 'reducer.py'
AS word, count;
```

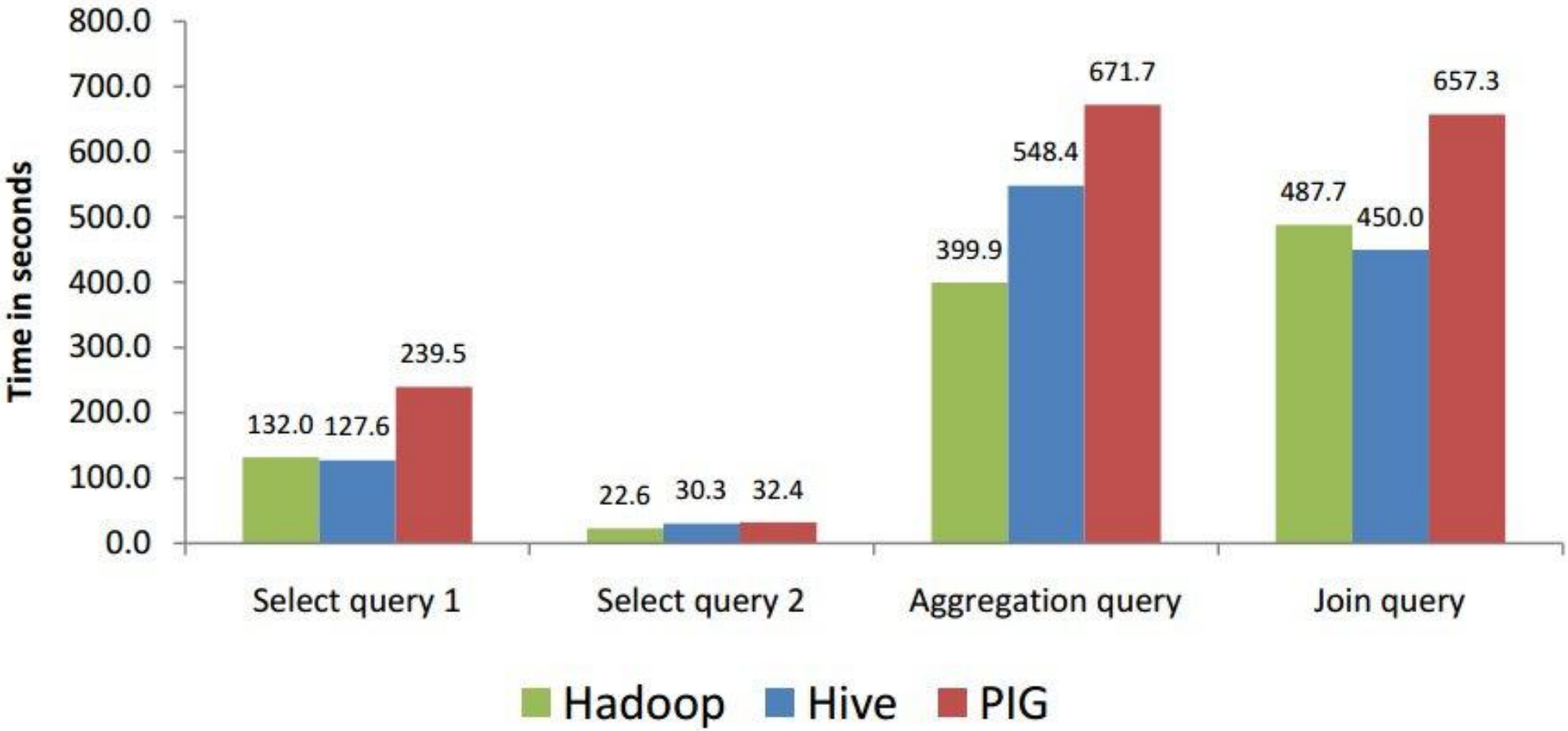
Performance - Dataset structure

grep(key VARCHAR(10), field VARCHAR(90))	2 columns, 500 million rows, 50GB
rankings(pageRank INT, pageURL VARCHAR(100), avgDuration INT)	3 columns, 56.3 million rows, 3.3GB.
uservisits(sourceIP VARCHAR(16), destURL VARCHAR(100), visitDate DATE, adRevenue FLOAT, userAgent VARCHAR(64), countryCode VARCHAR(3), languageCode VARCHAR(6), searchWord VARCHAR(32), duration INT).	9 columns, 465 million rows, 60GB (scaled down from 200GB).

Performance - Test query

Select query 1	SELECT * FROM grep WHERE field like '%XYZ%';
Select query 2	SELECT pageRank, pageURL FROM rankings WHERE pageRank > 10;
Aggregation query	SELECT sourceIP, SUM(adRevenue) FROM uservisits GROUP BY sourceIP;
Join query	SELECT INTO Temp sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue FROM rankings AS R, userVisits AS UV WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN Date(`1999-01-01') AND Date(`2000-01-01') GROUP BY UV.sourceIP;

Performance - Result



Hive – Performance

SVN Revision	Major Changes	Query A	Query B	Query C
746906	Before Lazy Deserialization	83 sec	98 sec	183 sec
747293	Lazy Deserialization	40 sec	66 sec	185 sec
751166	Map-side Aggregation	22 sec	67 sec	182 sec
770074	Object Reuse	21 sec	49 sec	130 sec
781633	Map-side Join	21 sec	48 sec	132 sec
801497	Lazy Binary Format	21 sec	48 sec	132 sec

- QueryA: SELECT count(1) FROM t;
- QueryB: SELECT concat(concat(concat(a,b),c),d) FROM t;
- QueryC: SELECT * FROM t;

Pros

- Pros
  - A easy way to process large scale data
  - Support SQL-based queries
  - Provide more user defined interfaces to extend
  - Programmability
  - Efficient execution plans for performance
  - Interoperability with other database tools

Cons

- Cons
  - Potential inefficiency
  - No easy way to append data
    - Updates are available starting in Hive 0.14
  - Files in HDFS are rarely updated
- Future work
  - Views / Variables
  - More operator
    - In/Exists semantic

Applications

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

Hive Usage @ Facebook (2010)

- Statistics per day:
  - 4TB of compressed new data added per day
  - 135TB of compressed data scanned per day
  - 7500+ Hive jobs on per day
- Hive simplifies Hadoop:
  - ~200 people/month run jobs on Hadoop/Hive
  - Analysts (non-engineers) use Hadoop through Hive
  - 95% of jobs are Hive Jobs

Competitors/Related Work

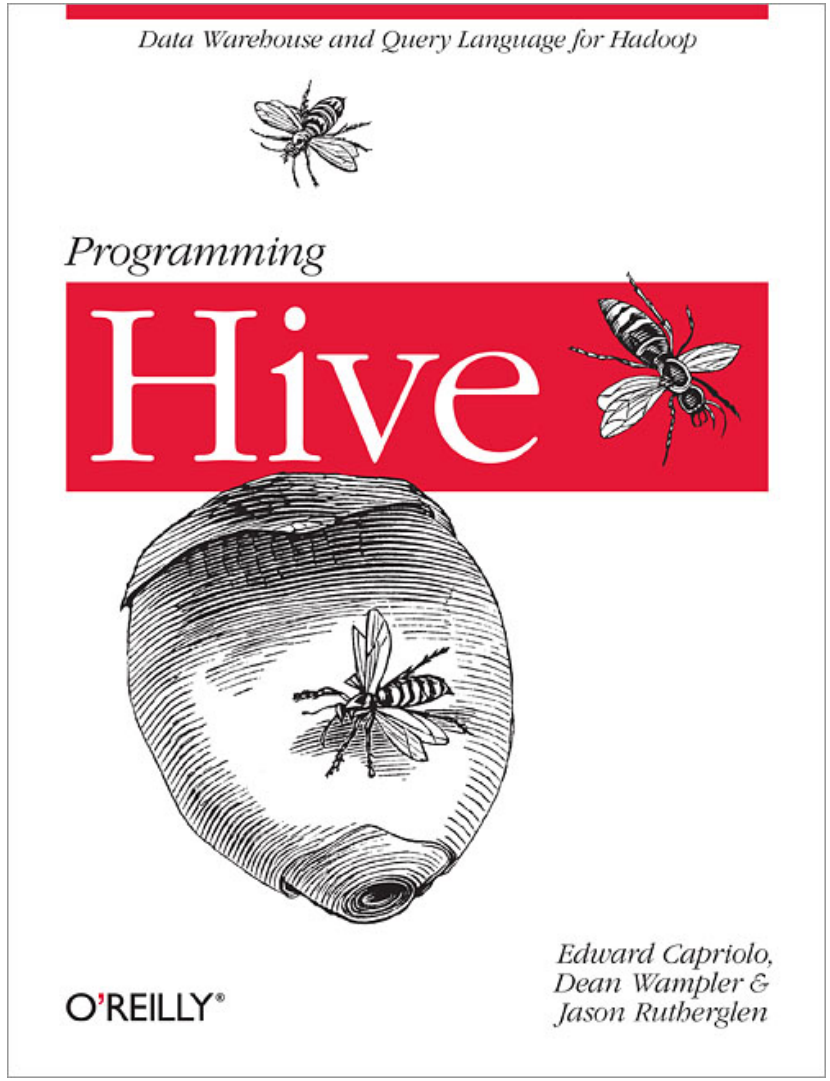
A non-exhaustive list:

- Spark SQL
- Google: (Apache) Drill, BigQuery
- IBM: BigSQL
- Oracle: Big Data SQL
- Microsoft: Cosmos
- Hortonworks: Stinger (fast Hive)
- Pivotal HD: HAWQ
- Cloudera: Impala
- Facebook: (Apache) Presto

Conclusion

- A easy way to process large scale data.
- Support SQL-based queries.
- Provide more user defined interfaces to extend
- Programmability.
- Typical applications:
  - Log processing: Daily Report, User Activity Measurement
  - Data/Text mining: Machine learning (Training Data)
  - Business intelligence: Advertising Delivery, Spam Detection

References


Data Warehouse and Query Language for Hadoop
Programming
Hive
O'REILLY®
Edward Capriolo,
Dean Wampler &
Jason Rutherglen