Using MySQL, Hadoop and Spark for Data Analysis

Alexander Rubin
Principle Architect, Percona
September 21, 2015
About Me

Alexander Rubin, Principal Consultant, Percona

• Working with MySQL for over 10 years
  – Started at MySQL AB, Sun Microsystems, Oracle (MySQL Consulting)
  – Worked at Hortonworks (Hadoop company)
  – Joined Percona in 2013
Agenda

• Why Hadoop?
• Why Spark?
• Hadoop
  – Big Data Analytics with Hadoop
  – Star Schema benchmark
  – MySQL and Hadoop Integration
• Spark examples
Why Hadoop and not MySQL?

- Petabytes of data
- Unstructured/raw data
  - No normalization in the first place
- Data is collected at a high rate

http://en.wikipedia.org/wiki/Big_data#Definition
Why Spark?

• Claimed to be faster
  • ... in memory processing
• Direct access to data sources (i.e. MySQL)

```python
>>> df = sqlContext.load(source="jdbc",
url="jdbc:mysql://localhost?user=root",
dbtable="ontime.ontime_sm")
```

• Native R (and Python) integration
Inside Hadoop
Where is my data?

- HDFS = Data is spread between MANY machines
- Also Amazon S3 is supported
The Famous Picture!

- HDFS = Data is spread between MANY machines
- Write files, “append-only” mode
Hive

- SQL level for Hadoop
- *Translates SQL to Map-Reduce jobs*
- Schema on Read – does not check the data on load
Hive Example

hive> create table lineitem ( 
  l_orderkey int, 
  l_partkey int, 
  l_suppkey int, 
  l_linenumber int, 
  l_quantity double, 
  l_extendedprice double, 
  ... 
  l_shipmode string, 
  l_comment string) 
  row format delimited fields terminated by '|';
hive> create external table lineitem ( ...)
    row format delimited fields terminated by '|
    location '/ssb/lineorder/';
Impala: Faster SQL

Not based on Map-Reduce, directly get data from HDFS

Hadoop vs MySQL
Hadoop vs. MySQL for BigData

- Indexes
- Partitioning
- “Sharding”

- Full table scan
- Partitioning
- Map/Reduce
Hadoop (vs. MySQL)

- No indexes
  - All processing is full scan
  - BUT: *distributed and parallel*
- No transactions
- High latency (usually)

MySQL:
1 query = 1 CPU core
Indexes (BTree) for Big Data challenge

- Creating an index for Petabytes of data?
- Updating an index for Petabytes of data?
- Reading a terabyte index?
- Random read of Petabyte?

Full scan in parallel is better for big data
ETL vs ELT

**ETL**

1. **Extract** data from external source
2. **Transform** before loading
3. **Load** data into MySQL

**ELT**

1. **Extract** data from external source
2. **Load** data into Hadoop (as is)
3. **Transform** data/ Analyze data/ Visualize data;

Parallelism
Example: Loading wikistat into MySQL

1. Extract data from external source *(uncompress!)*

   1. Load data into MySQL and Transform

   - Wikipedia page counts – download, >10TB
   - load data local infile '${file}' into table wikistats.wikistats_full
     CHARACTER SET latin1
     FIELDS TERMINATED BY ' '
     (project_name, title, num_requests, content_size)

     set request_date = STR_TO_DATE('${datestr}',
     '%Y%m%d %H%i%S'),
     title_md5=unhex(md5(title));

   - [http://dumps.wikimedia.org/other/pagecounts-raw/](http://dumps.wikimedia.org/other/pagecounts-raw/)
Load timing per hour of wikistat

- InnoDB: 52.34 sec
- MyISAM: 11.08 sec (+ indexes)
- 1 hour of wikistats = 1 minute
- 1 year will load in 6 days
  - (8765.81 hours in 1 year)
- 6 year = > 1 month to load
Loading wikistat into Hadoop

- Just copy files to HDFS…and create hive structure
- How fast to search?
  - Depends upon the number of nodes
- **1000 nodes** spark cluster
  - 4.5 TB, 104 Billion records
  - Exec time: 45 sec
  - Scanning 4.5TB of data
Amazon Elastic Map Reduce
Hive on Amazon S3

hive> create external table lineitem ( 
  l_orderkey int, l_partkey int, l_suppkey int, 
  l_linenumber int, l_quantity double, 
  l_extendedprice double, l_discount double, l_tax 
  double, l_returnflag string, 
  l_linestatus string, l_shipdate string, 
  l_commitdate string, l_receiptdate string, 
  l_shipinstruct string, l_shipmode string, 
  l_comment string) 
row format delimited fields terminated by '|' 
location 's3n://data.s3ndemo.hive/tpch/lineitem';
Amazon Elastic Map Reduce

- Store data on S3
- Prepare SQL file (create table, select, etc)
- Run Elastic Map Reduce
  - Will start N boxes then stop them
- Results loaded to S3
Hadoop and MySQL Together

Integrating MySQL and Hadoop
Archiving to Hadoop

OLTP / Web site

Goal:
keep 100G – 1TB

MySQL

Can store Petabytes for archiving

BI / Data Analysis

Hadoop Cluster

Can store Petabytes for archiving

ELT
Integration: Hadoop -> MySQL
MySQL -> Hadoop: Sqoop

$ sqoop import
   --connect jdbc:mysql://mysql_host/db_name
   --table ORDERS
   --hive-import

MySQL Server

Hadoop Cluster
MySQL to Hadoop: Hadoop Applier

Only inserts are supported

Replication Master

MySQL Applier (reads binlogs from Master)

Hadoop Cluster

MySQL

Regular MySQL Slaves
MySQL to Hadoop: Hadoop Applier

- Download from: http://labs.mysql.com/
- Still Alpha version right now
- We need to write code how to process data
Start Spark (no Hadoop)

root@thor:~/spark/spark-1.4.1-bin-hadoop2.6# ./sbin/start-master.sh

less ../logs/spark-root-org.apache.spark.deploy.master.Master-1-thor.out

15/08/25 11:21:21 INFO Master: Running Spark version 1.4.1
15/08/25 11:21:21 INFO Utils: Successfully started service 'MasterUI' on port 8080.

root@thor:~/spark/spark-1.4.1-bin-hadoop2.6# ./sbin/start-slave.sh spark://thor:7077
root@thor:~/spark/spark-1.4.1-bin-hadoop2.6# cat env.sh
export MASTER=spark://`hostname`:7077
export SPARK_CLASSPATH=/root/spark/mysql-connector-java-5.1.36-bin.jar
# optional
export SPARK_WORKER_MEMORY=6g
export SPARK_MEM=6g
export SPARK_DAEMON_MEMORY=6g
export SPARK_DAEMON_JAVA_OPTS="-Dspark.executor.memory=6g"
PySpark and MySQL

root@thor:~/spark/spark-1.4.1-bin-hadoop2.6# ./bin/pyspark

Welcome to

Welcome to PySpark version 1.4.1

Using Python version 2.7.3 (default, Jun 22 2015 19:33:41)
SparkContext available as sc, HiveContext available as sqlContext.
df.registerTempTable("City")
res = sqlContext.sql("select * from City order by Population desc limit 10")
for x in res.collect():
    print x.Name, x.Population
SparkSQL and MySQL

root@thor:~/spark/spark-1.4.1-bin-hadoop2.6# ./bin/spark-sql 2>error.log

CREATE TEMPORARY TABLE City
USING org.apache.spark.sql.jdbc
OPTIONS (  
  url "jdbc:mysql://localhost?user=root",  
  dbtable "world.City"
);
select * from City order by Population desc limit 10;
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a Row.
lines = sc.textFile("/home/consultant/wikistats/dumps.wikimedia.org/other/pagecounts-raw/2008/2008-01/pagecounts-20080112-120000.gz")
parts = lines.map(lambda l: l.split("")
wiki = parts.map(lambda p: Row(project=p[0], url=p[1], uniq_visitors=int(p[2]), total_visitors=int(p[3])))

# Infer the schema, and register the DataFrame as a table.
schemaWiki = sqlContext.createDataFrame(wiki)
schemaWiki.registerTempTable("wikistats")
res = sqlContext.sql("SELECT * FROM wikistats limit 100")
for x in res.collect():
    print x.project, x.url, x.uniq_visitors
# Load a text file and convert each line to a Row.
lines = sc.textFile("/home/consultant/wikistats/
dumps.wikimedia.org/other/pagecounts-raw/2008/2008-01/")
parts = lines.map(lambda l: l.split(" "))
wiki = parts.map(lambda p: Row(project=p[0], url=p[1]))

# Infer the schema, and register the DataFrame as a table.
schemaWiki = sqlContext.createDataFrame(wiki)
schemaWiki.registerTempTable("wikistats")
res = sqlContext.sql("SELECT url, count(*) as cnt FROM wikistats
where project="en" group by url order by cnt desc limit 10")
for x in res.collect():
    print x.url, x.cnt
<table>
<thead>
<tr>
<th>Cpu</th>
<th>us</th>
<th>sy</th>
<th>ni</th>
<th>id</th>
<th>wa</th>
<th>hi</th>
<th>si</th>
<th>st</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cpu0</td>
<td>94.4%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.6%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu1</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>92.4%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>1.9%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu2</td>
<td>95.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu3</td>
<td>94.9%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.1%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu4</td>
<td>0.6%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>99.4%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu5</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu6</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu7</td>
<td>95.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu8</td>
<td>94.4%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.6%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu9</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu10</td>
<td>94.4%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.6%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu11</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu12</td>
<td>1.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>98.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu13</td>
<td>95.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu14</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu15</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu16</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu17</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu18</td>
<td>94.3%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu19</td>
<td>94.9%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.1%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu20</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu21</td>
<td>94.9%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.1%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu22</td>
<td>94.9%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>5.1%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Cpu23</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Mem: 49454372k total, 40479496k used, 8974876k free, 357360k buffers