Hive and Pig

Juliana Freire
New York University

Some slides from J. Lin and C. Olston
Data Warehousing

Scale

- Often not scalable enough

Prohibitively expensive at web scale
  - Up to $200K/TB

SQL

- Little control over execution method
  - Query optimization is hard
    - Parallel environment
    - Little or no statistics
      - Lots of UDFs

[Olston et al., SIGMOD 2008]
The Map-Reduce Appeal

Scalable due to simpler design
• Only parallelizable operations
• No transactions

Runs on cheap commodity hardware

Procedural Control- a processing “pipe”

Scale

$\$\$

SQL

[Olston et al., SIGMOD 2008]
Map Reduce: Disadvantages

1. Extremely rigid data flow
   - Other flows constantly hacked in
   - Join, Union
   - Split
   - Chains

2. Common operations must be coded by hand
   - Join, filter, projection, aggregates, sorting, distinct

3. Semantics hidden inside map-reduce functions
   - Difficult to maintain, extend, and optimize

[Olston et al., SIGMOD 2008]
Need for Data Management

• Hadoop is great for large-data processing!
• But writing Java programs for everything is verbose and slow
• Not everyone wants to (or can) write Java code

Solution:
• Develop data management solutions
• Develop higher-level data processing languages
Hive and Pig

- **Hive**: data warehousing application in Hadoop
  - Query language is HQL, variant of SQL
  - Tables stored on HDFS as flat files
  - Developed by Facebook, now open source
- **Pig**: large-scale data processing system
  - Scripts are written in Pig Latin, a dataflow language
  - Developed by Yahoo!, now open source
  - Roughly 1/3 of all Yahoo! internal jobs
- **Common idea:**
  - Provide higher-level language to facilitate large-data processing
  - Higher-level language “compiles down” to Hadoop jobs
Hive: Background

- Started at Facebook  [Thusoo et al., VLDB2009 and ICDE2010]
- Data was collected by nightly cron jobs into Oracle DB
- “ETL” via hand-coded python
- Grew from 10s of GBs (2006) to 1 TB/day new data (2007)
- Some daily processing jobs were taking more than a day!
- Started to use Hadoop
  - Jobs that took more than a day, ran in a few hours
- But, it was too hard for end users…
- Goal: bring the familiar concepts of tables, columns, partitions and a subset of SQL to the unstructured world of Hadoop, while still maintaining the extensibility and flexibility that Hadoop enjoyed

Source: cc-licensed slide by Cloudera

“In Facebook, the Hive warehouse contains several thousand tables with over 700 terabytes of data and is being used extensively for both reporting and ad-hoc analyses by more than 100 users”

Hive Components

- Shell: allows interactive queries
- Driver: session handles, fetch, execute
- Compiler: parse, plan, optimize
- Translates HiveQL statement into a DAG of mapreduce jobs
- Execution engine: DAG of stages (MR, HDFS, metadata)
- Metastore: schema, location in HDFS, SerDe (custom serialization/deserialization)
Metastore

- Database: namespace containing a set of tables
- Holds table definitions (column types, physical layout)
- Holds partitioning information
- Can be stored in Derby (https://db.apache.org/derby), MySQL, and many other relational databases
- Why not use HDFS?
Data Model: Tables

- **Tables**: analogous to tables in RDBMS
  - Each table has a corresponding HDFS directory – contents stored as files
  - Typed columns (int, float, string, boolean)
  - Structs: {a INT; b INT}.
  - Also, list<element type>
  - associative arrays map<key-type,value-type>
- Supports arbitrary data types: useful for external and legacy data
  - Programmer needs to provide a jar that contains implementation for SerDe (serialization/deserialization)
  - Deserializer: string or binary object → Java object
  - Serializer: Java object → object that is written to HDFS

See example in https://blog.cloudera.com/blog/2012/12/how-to-use-a-serde-in-apache-hive/
Data Model: Partitions

- Tables are organized into partitions based on values
  - E.g., range-partition tables by date, partition by country
  - HDFS for ds=20120410, ctry=US
    /wh/pvs/ds=20120410/ctry=US
  - HDFS for ds=20120410, ctry=IN
    /wh/pvs/ds=20120410/ctry=IN

- Speed up query processing: only scans the appropriate directories

Tables — dir

\[ \downarrow \]

Partitions — sub-dir

\[ \downarrow \]

Buckets — file

Partitions - Each table can have one or more partitions which determine the distribution of data within sub-directories of the table directory. Suppose data for table T is in the directory /wh/T. If T is partitioned on columns ds and ctry, then data with a particular ds value 20090101 and ctry value US, will be stored in files within the directory /wh/T/ds=20090101/ctry=US.

Data Model: Buckets

- Stored as a file in the partition directory
- E.g., partition by country, bucket by employee id
- User defines the number of buckets
- E.g., Records with the employee id will be stored in the same bucket
- Useful for sampling, join optimization

Tables \(\rightarrow\) Dir

Partitions \(\rightarrow\) Sub-dir

Buckets \(\rightarrow\) File
Physical Layout

- Warehouse directory in HDFS
  - E.g., /user/hive/warehouse
- Tables stored in subdirectories of warehouse
  - Partitions form subdirectories of tables
  - Each table has a corresponding HDFS directory
- Actual data stored in flat files
  - Users can associate a table with a serialization format
  - Control char-delimited text, or SequenceFiles
  - With custom SerDe, can use arbitrary format

Source: cc-licensed slide by Cloudera
Query Language

- Data definition language (DDL)
  - Create tables with specific serialization formats
  - Specify partitioning and buckets
- Data Manipulation Language
  - Load data from external sources
  - Insert query results into Hive tables
  - In 2009, no support for updates or deletions, or transactions!
  - Transactions added in v 0.13, but by default, transactions are turned off


https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions
CREATE TABLE test_part(ds string, hr int)
PARTITIONED BY (ds string, hr int);

SELECT * FROM test_part WHERE ds='2009-01-01';

will only scan all the files within the
/user/hive/warehouse/test_part/ds=2009-01-01 directory

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

will only scan all the files within the
/user/hive/warehouse/test_part/ds=2009-02-02/hr=11 directory
Hive: Example

• Hive looks similar to an SQL database
• Relational join on two tables:
  • Table of word counts from Shakespeare collection
  • Table of word counts from the bible
  
  SELECT s.word, s.freq, k.freq
  FROM shakespeare s JOIN bible k ON (s.word = k.word)
  WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;

  the     25848   62394
  I       23031    8854
  and     19671   38985
  to      18038   13526
  of      16700   34654
  a       14170    8057
  you     12702    2720
  my      11297    4135
  in      10797   12445
  is      8882    6884

Source: Material drawn from Cloudera training VM
Hive: Another Example

Facebook status updates logged into status_updates, then load daily into Hive

```
LOAD DATA LOCAL INPATH '/logs/status_updates'
INTO TABLE status_updates PARTITION (ds='2009-03-20')
```

Detailed user info in: profiles(userid int,school string,gender int)

```
FROM (SELECT a.status, b.school, b.gender
          FROM status_updates a JOIN profiles b
          ON (a.userid = b.userid and
              a.ds='2009-03-20')
      ) subq1
INSERT OVERWRITE TABLE gender_summary
    PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender
```

```
INSERT OVERWRITE TABLE school_summary
    PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1) GROUP BY subq1.school
```
FROM (SELECT a.status, b.school, b.gender
FROM status_updates a JOIN profiles b
ON (a.userid = b.userid and a.ds='2009-03-20') ) subq1
INSERT OVERWRITE TABLE gender_summary
PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender
INSERT OVERWRITE TABLE school_summary
PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1) GROUP BY subq1.school

A Query Plan
Hive: Another Example

- HiveQL provides MapReduce constructs

```
REDUCE subq2.school, subq2.meme, subq2.cnt
    USING 'top10.py' AS (school,meme,cnt)
FROM (SELECT subq1.school, subq1.meme, COUNT(1) AS cnt
    FROM (MAP b.school, a.status
    USING 'meme-extractor.py' AS (school,meme)
    FROM status_updates a JOIN profiles b
        ON (a.userid = b.userid)
    ) subq1
    GROUP BY subq1.school, subq1.meme
    DISTRIBUTION BY school, meme
    SORT BY school, meme, cnt desc
) subq2;
```
Example: Ad-Hoc Data Analysis

Find users who tend to visit “good” pages.

<table>
<thead>
<tr>
<th>Visits</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>url</td>
</tr>
<tr>
<td>Amy</td>
<td><a href="http://www.cnn.com">www.cnn.com</a></td>
</tr>
<tr>
<td>Amy</td>
<td><a href="http://www.crap.com">www.crap.com</a></td>
</tr>
<tr>
<td>Amy</td>
<td><a href="http://www.myblog.com">www.myblog.com</a></td>
</tr>
<tr>
<td>Amy</td>
<td><a href="http://www.flickr.com">www.flickr.com</a></td>
</tr>
<tr>
<td>Fred</td>
<td>cnn.com/index.htm</td>
</tr>
</tbody>
</table>

Pig Slides adapted from Olston et al.
Should we use Hive?
Warehouses and Ad-Hoc Analyses

- Parallel database
  - E.g., Teradata, Oracle RAC, Netezza
  - Simple SQL interface, hide complexity of physical cluster
  - Solutions are expensive
- Hive
  - Simple SQL interface, hide complexity of physical cluster
  - Free ;-)
  - Require substantial configuration
    - Too much work for a one-off task
- Map-reduce is too low level and rigid
  - Hard to maintain and re-use
Conceptual Dataflow

Find users who tend to visit “good” pages.

Load
Visits(user, url, time)

Canonical URLs

Load
Pages(url, pagerank)

Join
url = url

Group by user

Compute Average Pagerank

Filter
avgPR > 0.5

Pig Slides adapted from Olston et al.
System-Level Dataflow

Visits

Pages

load

load

join by url

group by user

compute average pagerank

filter

the answer

Map

Reduce

Pig Slides adapted from Olston et al.
Visits = load '/data/visits' as (user, url, time);
Visits = foreach Visits generate user, Canonicalize(url), time;

Pages = load '/data/pages' as (url, pagerank);

VP = join Visits by url, Pages by url;
UserVisits = group VP by user;
UserPageranks = foreach UserVisits generate user,
    AVG(VP.pagerank) as avgpr;
GoodUsers = filter UserPageranks by avgpr > '0.5';

store GoodUsers into '/data/good_users';
Visits = load '/data/visits' as (user, url, time);
Visits = foreach Visits generate user, Canonicalize(url), time;

Pages = load '/data/pages' as (url, pagerank);

VP = join Visits by url, Pages by url;
UserVisits = group VP by user;
UserPageranks = foreach UserVisits generate user,
               AVG(VP.pagerank) as avgpr;
GoodUsers = filter UserPageranks by avgpr > '0.5';

store GoodUsers into '/data/good_users';

Schema is optional; assigned dynamically
Pig Latin Script: Loading data

Visits = load '/data/visits' as (user, url, time);
Visits = foreach Visits generate user, Canonicalize(url), time;

Visits = load '/data/visits' using myLoad()
as (user, url, time);

UserVisits = group Visits by user,
UserPageranks = foreach UserVisits generate user,
           AVG(VP.pagerank) as avgpr;
GoodUsers = filter UserPageranks by avgpr > '0.5';

store GoodUsers into '/data/good_users';
Visits = load '/data/visits' as (user, url, time);

Visits = foreach Visits generate user, Canonicalize(url), time;

- Each tuple is processed independently
- Can be easily parallelized

`AVG(VP.pagerank) as avgpr;

GoodUsers = filter UserPageranks by avgpr > '0.5';

store GoodUsers into '/data/good_users';`
Visits = load '/data/visits' as (user, url, time);

Visits = foreach Visits generate user, Canonicalize(url), time;

Pages = load '/data/pages' as (url, pagerank);

VP = join Visits by url, Pages by url;

UserVisits = group VP by user;

UserPageranks = foreach UserVisits generate user,
               AVG(VP.pagerank) as avgpr;

GoodUsers = filter UserPageranks by avgpr > '0.5';

store GoodUsers into '/data/good_users';
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6

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);

Similar to specifying an execution plan!
See, it is useful to learn relational algebra
Pig Latin has a fully-nestable data model with:
- Atomic values, tuples, bags (lists), and maps
- More natural to programmers than flat tuples
- Avoids expensive joins
- User-defined functions
  - Spam_urls = FILTER urls BY isSpam(url);
  - User define functions in Java, Python, JavaScript – see https://pig.apache.org/docs/r0.9.1/udf.html
- See paper [Olston et al., SIGMOD 2008]
Implementation

automatic rewrite + optimize

Pig

Hadoop Map-Reduce

cluster

or

or

user
Compilation into Map-Reduce

Every group or join operation forms a map-reduce boundary

Other operations pipelined into map and reduce phases
Optimizations

- Pig Latin programs supply explicit sequence of operations, but they are not necessarily executed in that order.
- High-level relational-algebra-style operations enable traditional database optimization.
- Example:
  
  spam_urls = FILTER urls BY isSpam(url);
  
  culprit_urls = FILTER spam_urls BY pagerank > 0.8;

- If isSpam is an expensive function and the FILTER condition is selective, it is more efficient to execute the second statement first.
Java vs. Pig Latin

1/20 the lines of code

Performance on par with raw Hadoop!

1/16 the development time
Pig takes care of...

- Schema and type checking
- Translating into efficient physical dataflow
  - (i.e., sequence of one or more MapReduce jobs)
- Exploiting data reduction opportunities
  - (e.g., early partial aggregation via a combiner)
- Executing the system-level dataflow
  - (i.e., running the MapReduce jobs)
- Tracking progress, errors, etc.
References

• Getting started with Pig: http://pig.apache.org/docs/r0.11.1/start.html
• Pig Tutorial: http://pig.apache.org/docs/r0.7.0/tutorial.html
• Hive Tutorial: https://cwiki.apache.org/confluence/display/Hive/Tutorial
Questions