LANGUAGES FOR Hadoop: PIG & HIVE

Michail Michailidis & Patrick Maiden
Motivation

• Native MapReduce
  • Gives fine-grained control over how program interacts with data
  • Not very reusable
    • Can be arduous for simple tasks

• Last week – general Hadoop Framework using AWS
  • Does not allow for easy data manipulation
    • Must be handled in map() function

• Some use cases are best handled by a system that sits between these two
Why Declarative Languages?

- In most database systems, a declarative language is used (i.e. SQL)
  - Data Independence
    - User applications cannot change organization of data
  - Schema – structure of the data
    - Allows code for queries to be much more concise
    - User only cares about the part of the data he wants

SQL:
```
SELECT count(*) FROM word_frequency
WHERE word = 'the'
```

Java-ESQe:
```
int countThe = 0;
for (String word: words) {
    if (word.equals("the")) {
        countThe++;
    }
}
return countThe;
```
Native MapReduce - Wordcount

In native MapReduce, simple tasks can be a hassle to code:

```java
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true);
    }
}
```

source: http://wiki.apache.org/hadoop/WordCount
Pig (Latin)

• Developed by Yahoo! around 2006
  • Now maintained by Apache

• Pig Latin – Language

• Pig – System implemented on Hadoop

• Citation note: Many of the examples are pulled from the research paper on Pig Latin

Image source: ebiquity.umbc.edu
Pig Latin – Language Overview

• Pig Latin – language of Pig Framework
  • Data-flow language
    • Stylistically between declarative and procedural
  • Allows for easy integration of user-defined functions (UDFs)
    • “First Class Citizens”
• All primitives can be parallelized
• Debugger
Pig Latin - Wordcount

• Wordcount in Pig Latin is significantly simpler:

```java
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS (line:chararray);
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
filtered_words = FILTER words BY word MATCHES '\\w+';
word_groups = GROUP filtered_words BY word;
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

source: http://en.wikipedia.org/wiki/Pig_(programming_tool)
Pig Latin – Data Model

Atom
‘providence’

Tuple
(‘providence’, ‘apple’)

Bag
(‘providence’, ’apple’)
(‘providence’, (‘javascript’, ‘red’))

Map
‘goes to’ → { (‘school’) }
‘gender’ → ‘female’
Pig – Under the Hood

- Parsing/Validation
- Logical Planning
  - Optimizes data storage
    - Bags only made when necessary
- Complied into MapReduce Jobs
  - Current implementation of Pig uses Hadoop

\[
\begin{align*}
\text{map}_1 & \quad \rightarrow \quad \text{filter} \quad \rightarrow \quad \text{group} \quad \rightarrow \quad \text{cogroup} \quad \rightarrow \quad \text{cogroup} \\
\text{reduce}_1 & \quad \rightarrow \quad \text{reduce}_i \quad \rightarrow \quad \text{reduce}_{i+1} \quad \rightarrow \quad \text{reduce}_{i+1}
\end{align*}
\]
Pig Latin – Key Commands

• LOAD
  • Specifies input format
    • Does not actually load data into tables
  • Can specify schema or use position notation
    • $0$ for first field, and so on

  \[ \text{queries} = \text{LOAD} \ 'query\_log.txt' \]
  \[ \hspace{1cm} \text{USING} \ \text{myLoad}() \]
  \[ \hspace{1.5cm} \text{AS} \ (\text{userId}, \text{queryString}, \]
  \[ \hspace{5cm} \text{timestamp}); \]

• FOREACH
  • Allows iteration over tuples

  \[ \text{expanded\_queries} = \text{FOREACH} \ \text{queries} \ \text{GENERATE} \]
  \[ \hspace{1cm} \text{userId}, \ \text{expandQuery(queryString)}; \]

• FILTER
  • Returns only data that passes the condition

  \[ \text{real\_queries} = \text{FILTER} \ \text{queries} \ \text{BY} \ \text{userId} \ \text{neq} \ 'bot'; \]
Pig Latin – Key Commands (2)

- **COGROUP/GROUP**
  - Preserves nested structure
- **JOIN**
  - Normal equi-join
  - Flattens data

```plaintext
results: (queryString, url , rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)

revenue: (queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)

grouped_data = COGROUP results BY queryString, revenue BY queryString;

join_result = JOIN results BY queryString, revenue BY queryString;
```

- **grouped_data:**
  - (lakers, nba.com, 1)
  - (lakers, espn.com, 2)
  - (kings, nhl.com, 1)
  - (kings, nba.com, 2)

- **join_result:**
  - (lakers, nba.com, 1, top, 50)
  - (lakers, espn.com, 2, side, 20)
  - (kings, nhl.com, 1, top, 30)
  - (kings, nba.com, 2, side, 10)
  ...
Pig Latin – Key Commands (3)

- Some Other Commands
  - UNION
    - Union of 2+ bags
  - CROSS
    - Cross product of 2+ bags
  - ORDER
    - Sorts by a certain field
  - DISTINCT
    - Removes duplicates
  - STORE
    - Outputs Data

- Commands can be nested

```pig
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue{
  top_slot = FILTER revenue BY adSlot eq 'top';
  GENERATE queryString,
  SUM(top_slot.amount),
  SUM(revenue.amount);
};
```
Pig Latin – MapReduce Example

• The current implementation of Pig compiles into MapReduce jobs
• However, if the workflow itself requires a MapReduce structure, it is extremely easy to implement in Pig Latin

```pig
map_result = FOREACH input GENERATE FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
output = FOREACH key_groups GENERATE reduce(*);
```

*MapReduce in 3 lines*

• Any UDF can be used for map() and reduce() here
Pig Latin – UDF Support

• Pig Latin currently supports the following languages:
  • Java
  • Python
  • Javascript
  • Ruby

• Piggy Bank
  • Repository of Java UDFs

• OSS
  • Possibility for more support

Image sources:
http://www.jbase.com/new/products/images/java.png
http://research.yahoo.com/files/images/pig_open.gif
http://img2.nairaland.com/attachments/693947_python-logo.png
26f0333ad80ad765dabb1115dde48966
Pig Pen – Debugger

- Debugging big data can be tricky
  - Programs take a long time to run

- Pig provides debugging tools that generate a sandboxed data set
  - Generates a small dataset that is representative of the full one
Hive

- Development by Facebook started in early 2007
- Open-sourced in 2008
- Uses HiveQL as language

Why was it built?
- Hadoop lacked the expressiveness of languages like SQL

Important: It is **not** a database!
Queries take many minutes ➔
HiveQL – Language Overview

• Subset of SQL with some deviations:
  • Only equality predicates are supported for joins
  • Inserts override existing data
  • INSERT INTO, UPDATE, DELETE are not supported → simpler locking mechanisms
  • FROM can be before SELECT for multiple insertions

• Very expressive for complex logic in terms of map-reduce (MAP / REDUCE predicates)

• Arbitrary data format insertion is provided through extensions in many languages
CREATE EXTERNAL TABLE lines(line string)
LOAD DATA INPATH 'books' OVERWRITE INTO TABLE lines

SELECT word, count(*) FROM lines
    LATERAL VIEW explode(split(text, ' ' )) lTable as word
GROUP BY word;
Type System

• Basic Types:
  - Integers, Floats, String

• Complex Types:
  - Associative Arrays – \textit{map}<key-type, value-type>
  - Lists – \textit{list}<element-type>
  - Structs – \textit{struct}<field-name: field-type, … >

• Arbitrary Complex Types:

\texttt{list< map< string, struct< p1:int,p2:int > > >}

represents list of associative arrays that map strings to structs that contain two ints – access with dot operator
**Data Model and Storage**

TABLES (dirs)

PARTITIONS (dirs)

BUCKETS (files)

**Directory Structure**

/root-path

/table1

/partition1 (2011-11)
- /bucket1 (1/3)
- /bucket2 (2/3)
- /bucket3 (3/3)

/partition2 (2011-12)
- /bucket1 (1/3)
- /bucket2 (2/3)
- /bucket3 (3/3)

/table2
- /bucket1 (1/2)
- /bucket2 (2/2)

*e.g. partition by date, time

Used for Subsampling*
System Architecture

- **Metastore (mysql):** stores metadata about tables, columns, partitions etc
- **Driver:** manages the lifecycle of HiveQL through Hive
- **Query Compiler:** compiles HiveQL → DAG of map/reduce tasks
- **Optimizer:** (part of compiler) Optimizes the DAG
- **Execution Engine:** executes the tasks produced by compiler in proper dependency
- **JDBC, ODBC, Thrift:** provide cross-language support
Optimization steps

A lot of similarities with SQL optimization

- Column pruning
- Predicate pushdown
- Partition pruning
- Map Side Joins
- Join Reordering

SELECT name FROM Student JOIN Student_Course ON (Student.id = Student_Course.sid) WHERE year = 3

• JOIN will be 23000x5000 => 115000000 rows
• If we select 3rd-year Students first and then join, the operation will be just 900x5000 = 20700000 rows

CREATE TABLE Student (id, name, year)
PARTITION BY class_year
SELECT * FROM Student WHERE class_year >= 2012

• Here only the data in partitions 2012 and 2013 will be used
• Smaller tables in joins are replicated in all mappers and joined with other tables in reducers

Smaller tables in joins are replicated in all mappers and joined with other tables in reducers.

Large tables are streamed in the reducer and smaller tables are kept in memory.

Optimization steps
- Column pruning
- Predicate pushdown
- Partition pruning
- Map Side Joins
- Join Reordering

SELECT id FROM Student WHERE year = 3

Student(id, name, year) 5000 rows
Student_Course(sid, cid) 23000 rows

Student(id, name, year) 5000 rows
Student_Course(sid, cid) 23000 rows

SELECT name FROM Student JOIN
ON (Student.id = Student_Course.sid) WHERE
year = 3

SELECT * FROM Student WHERE
• class_year >= 2012

Large tables are joins
• Raw replication in all reducers and joined with smaller tables in mappers and reducers.

The resulting map/reduce jobs will need only id from Student
• Minimizes the data transmission between mappers and reducers.

Friday, September 27, 13
# Pig & Hive - Comparison

<table>
<thead>
<tr>
<th>Pig</th>
<th>Hive</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Schema-optional</td>
<td>• Schema-aware</td>
</tr>
<tr>
<td>• Great at ETL (Extract-Transform-Load) jobs</td>
<td>• Great at ad-hoc analytics</td>
</tr>
<tr>
<td>• Dataflow style language</td>
<td>• Leverages SQL expertise</td>
</tr>
<tr>
<td>• Nested data in Bags</td>
<td>• Supports sampling and partitioning of data</td>
</tr>
<tr>
<td>• Debugging tools</td>
<td>• Optimizes queries</td>
</tr>
</tbody>
</table>
Hive & Pig – Limitations

• Neither of these systems are databases in and of themselves
  • Rely on Hadoop and MapReduce
    • Can be slow, especially when compared to other systems for simple tasks
    • Neither of these systems are well-suited for heirachically-structured data
References

• Pig
  - Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig Latin: A Not-So-Foreign Language for Data Processing
  - http://pig.apache.org/docs/

• Hive
  - Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu and Raghotham Murthy
    Hive – A Petabyte Scale Data Warehouse Using Hadoop
  - Pig vs Hive vs Native Map Reduce http://stackoverflow.com/questions/17950248/pig-vs-hive-vs-native-map-reduce
  - http://hive.apache.org/docs/