Outline

1. Map Reduce
2. Friends or Foes?
3. Pig Latin

Amol Deshpande  CMSC724: MapReduce
Discussion/Thoughts

- From Curt Monash’s Blog, especially Mapreduce part
- New key industry players in large-scale data analysis/data warehousing
  - Netezza, Aster, Greenplum, Vertica (Stonebraker) etc...
  - Along with Oracle (Exadata), DB2, Teradata, many more
  - Many have a few customers each
Discussion/Thoughts

- From Curt Monash’s Blog, especially Mapreduce part
- New key industry players in large-scale data analysis/data warehousing
  - Netezza, Aster, Greenplum, Vertica (Stonebraker) etc...
  - Along with Oracle (Exadata), DB2, Teradata, many more
  - Many have a few customers each
- Netezza, Aster, Greenplum offer Mapreduce functionality by now
  - Aster: Highly parallel data warehousing solution – very nice whitepaper on Mapreduce
  - We will see some syntax later
- SIGMOD 2009 paper: A Comparison of Large-scale Data Analysis
Goal: efficient parallelization of various tasks across 1000’s of machines without the user having to worry about the details such as:

- How to parallelize
- How to distribute the data
- How to handle failures
MapReduce

- Goal: efficient parallelization of various tasks across 1000’s of machines without the user having to worry about the details such as:
  - How to parallelize
  - How to distribute the data
  - How to handle failures

- Basic Idea:
  - If you force programs to be written using two primitives (*map* and *reduce*), parallelism can be gotten for free
    - Replace: map-reduce with SQL, parallelism with speed/ease-of-use
  - More programs than you might think can be written this way
MapReduce: Applications

- From Nice Overview by Curt Monash
- Three major classes:
  - Text tokenization, indexing, and search
  - Creation of other kinds of data structures (e.g., graphs)
  - Data mining and machine learning
- See this blog post for a long list of applications
- Or See Hadoop List
- For Machine Learning algorithms, see MAHOUT
Mapreduce

- Users needs to write two key functions:
  - Map: generate a set of (key, value) pairs
  - Reduce: group the pairs by key’s and combine them (GROUP BY)
- Borrowed from Lisp

```java
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names specified by the user). Typically, users do not need to combine these R output files into one file; they often pass these files as input to another MapReduce call or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures
The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed) and the identity of the worker machine (for nonidle tasks). The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

3.3 Fault Tolerance
Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Handling Worker Failures
The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling. Completed map tasks are reexecuted on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be reexecuted since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the reexecution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply reexecuted the work done by the unreachable worker machines and continued to make forward progress, eventually completing the MapReduce operation.

Semantics in the Presence of Failures
When the user-supplied map and reduce operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a nonfaulting sequential execution of the entire program.

Fig. 1. Execution overview.
MapReduce: Implementation

- A master for each task, assigns tasks to workers
- Data transfers using the file system (by passing file-names)
- Master pings the workers to make sure they are alive
  - If not, reassign the task to some other worker
- Work is divided into a large number of small chunks
  - Similar ideas used in parallel database for handling data skew
- Atomic commits using the file system
Map reduce: Implementation

- Google File System
  - A distributed, fault-tolerant file system
  - Data divided into blocks of 64MB
  - Each block stored on several machines (typically 3)
- Mapreduce uses the location information to assign work
Mapreduce: Implementation

- Google File System
  - A distributed, fault-tolerant file system
  - Data divided into blocks of 64MB
  - Each block stored on several machines (typically 3)
- Mapreduce uses the location information to assign work
- Many other optimizations
  - Backup tasks to handle “straggler”
  - Control over partitioning functions
  - Ability to skip “bad” records
Mapreduce

- Has been used within Google for:
  - Large-scale machine learning problems
  - Clustering problems for Google News etc.
  - Generating summary reports
  - Large-scale graph computations
- Also replaced the original tools for large-scale indexing
  - i.e., generating the inverted indexes etc.
  - runs as a sequence of 5 to 10 Mapreduce operations
Mapreduce: Thoughts

- **Hadoop**
  - Open-source implementation of Mapreduce
  - Has support for both the distributed file system and Mapreduce

- **Cloud Computing**
  - Somewhat vague term, but quite related
Mapreduce + Databases: Thoughts

- Abstract ideas have been known before
  - See [Mapreduce: A Major Step Backwards](#); DeWitt and Stonebraker
  - Can be implemented using user-defined aggregates in PostgreSQL quite easily
  - Top-down, declarative design
    - The user specifies what is to be done, not how many machines to use etc...
Abstract ideas have been known before
See [Mapreduce: A Major Step Backwards](#); DeWitt and Stonebraker
Can be implemented using user-defined aggregates in PostgreSQL quite easily
Top-down, declarative design
The user specifies what is to be done, not how many machines to use etc...

The strength comes from simplicity and ease of use
No database system can come close to the performance of Mapreduce infrastructure
RDBMSs can’t scale to that degree, are not as fault-tolerant etc...

Again: this is mainly because of ACID
Databases were designed to support it
Most of the Google tasks don’t worry about that
MapReduce is very good at what it was designed for
  But may not be ideal for more complex tasks
    E.g. no notion of “Query Optimization” (in particular, operator order optimization)
    The sequence of MapReduce tasks makes it procedural within a single machine

Joins are tricky to do
  MapReduce assumes a single input
Mapreduce + Databases: Thoughts

- Mapreduce is very good at what it was designed for
  - But may not be ideal for more complex tasks
    - E.g. no notion of “Query Optimization” (in particular, operator order optimization)
    - The sequence of Mapreduce tasks makes it procedural within a single machine
  - Joins are tricky to do
    - Mapreduce assumes a single input

- Trying to force use of Mapreduce may not be the best option
- However, much work in recent years on extending the functionality
  - See [Pig project at Yahoo](http://pig.apache.org), [Map-reduce-merge](https://github.com/mapreduce/map-reduce-merge) etc.
Map Reduce + Databases: Aster

- From the Aster White Paper
- Write two functions using your favorite language
  - *Map* and *Reduce*
- Use them directly in SQL
- Aster will take care of pipelining, parallel execution etc..

```sql
select token, sum(occurrences) as globalOccurrence
from map ( ON
  select word, count(*) as occurrences
  from WordOccurrences
  group by word )
group by token;
```
MR: A Major Step Backwards?

- An (in)famous blog post by DeWitt and Stonebraker
  - Discussed why MR wasn’t a new idea, and how most of the concepts were developed in parallel databases a long time ago
    - Still an interesting read
  - Later changed their position quite a bit
    - Result: the comparison paper (SIGMOD 2009)
- Key points
  - MR very good at extract-transform-load tasks
    - Experiments indicate loading data is much slower in databases
  - But not good at tasks that are best suited for DBMSes
  - UDF functionality in databases can cover many of other intended MR uses
Possible applications of MR

- (According to the authors)
- ETL and "read once" data sets
  - ETL has typically been distinct from databases
- Complex analytics
- Semi-structured data
- Quick and dirty analyses
  - MR has much shorter latency with such tasks
In the representative implementations (Hadoop vs Parallel Database like Vertica)

- Repetitive record parsing
  - Databases convert data into an internal format
- Compression
- Pipelining vs Materialization
  - Addressed by "Mapreduce Online" line of work

- Another important issue
  - Parallel Databases are very very expensive
Google Rebuttal to the Comparison Paper

- Conclusions based on implementation and evaluation shortcomings not fundamental to MR
  - In many cases, addressed in the Google implementation
- 1. MapReduce can exploit indices
  - Hadoop has the notion of "Input Connectors"
  - HadoopDB (Abadi et al.): Put Hadoop on top of relational databases
- 2. Map functions are often complex, and not easy to represent as UDFs
- 3. Protocol buffers for optimizing read/writes (no repetitive parsing)
- 4. Startup overhead in MR can be addressed by keeping worker processes alive
A lot of the differences are really about the implementation, not frameworks.

Somewhat orthogonal frameworks, with their pros and cons.
Many dataflow systems since the early Hadoop work

- Pig, Hive, DryadLINQ, Spark, F1, Impala, Tez, Naiad, Flink/Stratosphere, AsterixDB, Drill, ...
- Often hard to differentiate
- Typically have:
  - higher-level query languages like SQL (e.g., Spark DataFrames)
  - advanced execution strategies
  - ability to use indexes (typically through “connectors”)

Legacy?

- Schema flexibility – can process arbitrarily structured data
- Interface flexibility – use SQL, or imperative code
- Architecture flexibility – modular tools, that play nice with each other
Discussion/thoughts (From Redbook Chapter 5)

What’s next?

- Recent arguments that fault tolerance is not that essential except for very large clusters
- Increasing memory sizes mean many datasets fit in memory (see Frank McSherry’s blog posts on this topic)
- Much ongoing work on in-memory structured storage formats to get around issues of trying to read from HDFS (e.g., Parquet)
- Many supposedly MR products don’t really use MR
  - e.g., Cloudera Impala
Outline

1. Map Reduce
2. Friends or Foes?
3. Pig Latin
Overview

- Something that fits in between SQL and MapReduce
  - To make it easy for programmers to write procedural, non-SQL code
- Open source, on top of Hadoop
- No transactions – read-only analysis queries
- Supports nested data model (i.e., not in 1NF)
  - Allows sets/maps as fields
  - Interestingly: need this for GROUP operator
- UDFs written in Java
Example 1. Suppose we have a table urls: (url, category, pagerank). The following is a simple SQL query that finds, for each sufficiently large category, the average pagerank of high-pagerank urls in that category.

```
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
```

An equivalent Pig Latin program is the following. (Pig Latin is described in detail in Section 3; a detailed understanding of the language is not required to follow this example.)

```
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);
```
Basic Idea: Show the results of the operations on a small sample of the data

- Technical challenges: how to make sure that these are actually useful?
- e.g., Joins: if you take random samples of the relations, the result may contain nothing
  - Need to take biased samples

- Pig Pen: a visual debugging environment
Nevertheless, there still remain cases where `$GROUP` is followed by something other than an algebraic UDF, e.g., the program in Example 3.5, where `distributeRevenue` is not algebraic. To cope with these cases, our implementation allows for nested bags to spill to disk. Our disk-resident bag implementation comes with database-style external sort algorithms to do operations such as sorting and duplicate elimination of the nested bags (recall Section 3.7).

5. DEBUGGING ENVIRONMENT

The process of constructing a Pig Latin program is typically an iterative one: The user makes an initial stab at writing a program, submits it to the system for execution, and inspects the output to determine whether the program had the intended effect. If not, the user revises the program and repeats this process. If programs take a long time to execute (e.g., because the data is large), this process can be inefficient.

To avoid this inefficiency, users often create a side data set consisting of a small sample of the original one, for experimentation. Unfortunately this method does not always work well. As a simple example, suppose the program performs an equijoin of tables $A(x, y)$ and $B(x, z)$ on attribute $x$. If the original data contains many distinct values for $x$, then it is unlikely that a small sample of $A$ and a small sample of $B$ will contain any matching $x$ values [3]. Hence the join over the sample data set may well produce an empty result, even if the program is correct. Similarly, a program with a selective filter executed on a sample data set may produce an empty result. In general it can be difficult to test the semantics of a program over a sample data set.

Pig comes with a debugging environment called Pig Pen, which creates a side data set automatically, and in a manner that avoids the problems outlined in the previous paragraph. To avoid these problems successfully, the side data set must be tailored to the particular user program at hand. We refer to this dynamically-constructed side data set as a sandbox data set; we briefly describe how it is created in Section 5.1.

Pig Pen's user interface consists of a two-panel window as shown in Figure 4. The left-hand panel is where the user enters her Pig Latin commands. The right-hand panel is populated automatically, and shows the effect of the user's program on the sandbox data set. In particular, the intermediate bag produced by each Pig Latin command is displayed.

Suppose we have two data sets: a log of page visits, `visits: (user, url, time)` and a catalog of pages and their pageranks, `pages: (url, pagerank)`. The program shown in Figure 4 finds web surfers who tend to visit high-pagerank pages. The program joins the two data sets after first running the log entries through a UDF that converts urls to a canonical form. After the join, the program groups tuples by user, computes the average pagerank for each user, and then filters users by average pagerank.

The right-hand panel of Figure 4 shows a sandbox data set, and how it is transformed by each successive command. The main semantics of each command are illustrated via the sandbox data set: We see that the `JOIN` command matches `visits` tuples with `pages` tuples on `url`. We also see that grouping by `user` creates one tuple per group, possibly containing multiple nested tuples as in the case of Amy. Lastly we see that the `FOREACH` command eliminates the nesting via aggregation, and that the `FILTER` command eliminates Fred, whose average pagerank is too low.

If one or more commands had been written incorrectly, e.g., if the user had forgotten to include `GROUP` following `FOREACH`, the problem would be apparent in the right-hand panel. Similarly, if the program contains UDFs (as is common among real Pig users), the right-hand panel indicates whether the correct UDF is being applied, and whether it...