MapReduce: A Flexible Data Processing Tool

MapReduce is a programming model for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. We built a system around this programming model in 2003 to simplify construction of the inverted index for handling searches at Google.com. Since then, more than 10,000 distinct programs have been implemented using MapReduce at Google, including algorithms for large-scale graph processing, text processing, machine learning, and statistical machine translation. The Hadoop open source implementation of MapReduce has been used extensively outside of Google by a number of organizations.

To help illustrate the MapReduce programming model, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code like the following pseudocode:

```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));
```

The map function emits each word plus an associated count of occurrences (just ’1’ in this simple example). The reduce function sums together all counts emitted for a particular word.

MapReduce automatically parallelizes and executes the program on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing required inter-machine communication. MapReduce allows programmers with no experience with parallel and distributed systems to easily utilize the resources of a large distributed system. A typical MapReduce computation processes many terabytes of data on hundreds or thousands of machines. Programmers find the system easy to use, and more than 100,000 MapReduce jobs are executed on Google’s clusters every day.

Compared to Parallel Databases

The query languages built into parallel database systems are also used to
express the type of computations supported by MapReduce. A 2009 paper by Andrew Pavlo et al. (referred to here as the “comparison paper”\textsuperscript{12}) compared the performance of MapReduce and parallel databases. It evaluated the open source Hadoop implementation\textsuperscript{19} of the MapReduce programming model, DBMS-X (an unidentified commercial database system), and Vertica (a column-store database system from a company co-founded by one of the authors of the comparison paper). Earlier blog posts by some of the paper’s authors characterized MapReduce as “a major step backwards,”\textsuperscript{5,6} In this article, we address several misconceptions about MapReduce in these three publications:

- MapReduce cannot use indices and implies a full scan of all input data;  
- MapReduce input and outputs are always simple files in a file system; and  
- MapReduce requires the use of inefficient textual data formats.

We also discuss other important issues:

- MapReduce is storage-system independent and can process data without first requiring it to be loaded into a database. In many cases, it is possible to run 50 or more separate MapReduce analyses in complete passes over the data before it is possible to load the data into a database and complete a single analysis;  
- Complicated transformations are often easier to express in MapReduce than in SQL; and  
- Many conclusions in the comparison paper were based on implementation and evaluation shortcomings not fundamental to the MapReduce model; we discuss these shortcomings later in this article.

We encourage readers to read the original MapReduce paper\textsuperscript{4} and the comparison paper\textsuperscript{13} for more context.

**Heterogenous Systems**

Many production environments contain a mix of storage systems. Customer data may be stored in a relational database, and user requests may be logged to a file system. Furthermore, as such environments evolve, data may migrate to new storage systems. MapReduce provides a simple model for analyzing data in such heterogenous systems. End users can extend MapReduce to support a new storage system by defining simple reader and writer implementations that operate on the storage system. Examples of supported storage systems are files stored in distributed file systems,\textsuperscript{7} database query results,\textsuperscript{2,9} data stored in Bigtable,\textsuperscript{3} and structured input files (such as B-trees). A single MapReduce operation easily processes and combines data from a variety of storage systems.

Now consider a system in which a parallel DBMS is used to perform all data analysis. The input to such analysis must first be copied into the parallel DBMS. This loading phase is inconvenient. It may also be unacceptably slow, especially if the data will be analyzed only once or twice after being loaded. For example, consider a batch-oriented Web-crawling-and-indexing system that fetches a set of Web pages and generates an inverted index. It seems awkward and inefficient to load the set of fetched pages into a database just so they can be read through once to generate an inverted index. Even if the cost of loading the input into a parallel DBMS is acceptable, we still need an appropriate loading tool. Here is another place MapReduce can be used; instead of writing a custom loader with its own ad hoc parallelization and fault-tolerance support, a simple MapReduce program can be written to load the data into the parallel DBMS.

**Indices**

The comparison paper incorrectly said that MapReduce cannot take advantage of pregenerated indices, leading to skewed benchmark results in the paper. For example, consider a large data set partitioned into a collection of nondistributed databases, perhaps using a hash function. An index can be added to each database, and the result of running a database query using this index can be used as an input to MapReduce. If the data is stored in D database partitions, we will run D database queries that will become the D inputs to the MapReduce execution. Indeed, some of the authors of Pavlo et al. have pursued this approach in their more recent work.\textsuperscript{11}

Another example of the use of indices is a MapReduce that reads from Bigtable. If the data needed maps to a sub-range of the Bigtable row space, we would need to read only that sub-range instead of scanning the entire Bigtable. Furthermore, like Vertica and other column-store databases, we will read data only from the columns needed for this analysis, since Bigtable can store data segregated by columns.

Yet another example is the processing of log data within a certain date range; see the Join task discussion in the comparison paper, where the Hadoop benchmark reads through 155 million records to process the 134,000 records that fall within the date range of interest. Nearly every logging system we are familiar with rolls over to a new log file periodically and embeds the rollover time in the name of each log file. Therefore, we can easily run a MapReduce operation over just the log files that may potentially overlap the specified date range, instead of reading all log files.

**Complex Functions**

Map and Reduce functions are often fairly simple and have straightforward SQL equivalents. However, in many cases, especially for Map functions, the function is too complicated to be expressed easily in a SQL query, as in the following examples:

- Extracting the set of outgoing links from a collection of HTML documents and aggregating by target document;  
- Stitching together overlapping satellite images to remove seams and to select high-quality imagery for Google Earth;  
- Generating a collection of inverted index files using a compression scheme tuned for efficient support of Google search queries;  
- Processing all road segments in the world and rendering map tile images that display these segments for Google Maps; and  
- Fault-tolerant parallel execution of programs written in higher-level languages (such as Sawzall\textsuperscript{14} and Pig Latin\textsuperscript{15}) across a collection of input data.

Conceptually, such user-defined functions (UDFs) can be combined with SQL queries, but the experience reported in the comparison paper indicates that UDF support is either buggy (in DBMS-X) or missing (in Vertica). These concerns may go away over the long term, but for now, MapReduce is a better framework for doing more com-
complicated tasks (such as those listed earlier) than the selection and aggregation that are SQL’s forte.

**Structured Data and Schemas**

Pavlo et al. did raise a good point that schemas are helpful in allowing multiple applications to share the same data. For example, consider the following schema from the comparison paper:

```sql
CREATE TABLE Rankings ( 
    pageURL VARCHAR(100) PRIMARY KEY, 
    pagerank INT, 
    avgDuration INT );
```

The corresponding Hadoop benchmarks in the comparison paper used an inefficient and fragile textual format with different attributes separated by vertical bar characters:

```
137|http://www.somehost.com/index.html|602
```

In contrast to ad hoc, inefficient formats, virtually all MapReduce operations at Google read and write data in the Protocol Buffer format. A high-level language describes the input and output types, and compiler-generated code is used to hide the details of encoding/decoding from application code. The corresponding protocol buffer description for the Rankings data would be:

```protobuf
message Rankings { 
    required string pageurl = 1; 
    required int32 pagerank = 2; 
    required int32 avgduration = 3; 
}
```

The following Map function fragment processes a Rankings record:

```java
Rankings r = new Rankings();
r.parseFrom(value);
if (r.getPagerank() > 10) { ... }
```

The protocol buffer framework allows types to be upgraded (in constrained ways) without requiring existing applications to be changed (or even recompiled or rebuilt). This level of schema support has proved sufficient for allowing thousands of Google engineers to share the same evolving data types.

Furthermore, the implementation of protocol buffers uses an optimized binary representation that is more compact and much faster to encode and decode than the textual formats used by the Hadoop benchmarks in the comparison paper. For example, the automatically generated code to parse a Rankings protocol buffer record runs in 20 nanoseconds per record as compared to the 1,731 nanoseconds required per record to parse the textual input format used in the Hadoop benchmark mentioned earlier. These measurements were obtained on a JVM running on a 2.4GHz Intel Core-2 Duo. The Java code fragments used for the benchmark runs were:

```java
// Fragment 1: protocol buffer parsing
for (int i = 0; i < numIterations; i++) { 
    rankings.parseFrom(value);
    pagerank = rankings.getPagerank();
}
```

```java
// Fragment 2: text format parsing (extracted from Benchmark1.java // from the source code posted by Pavlo et al.)
for (int i = 0; i < numIterations; i++) { 
    String data[] = value.toString().split(“\|”);
    pagerank = Integer.valueOf(data[0]);
}
```

Given the factor of an 80-fold difference in this record-parsing benchmark, we suspect the absolute numbers for the Hadoop benchmarks in the comparison paper are inflated and cannot be used to reach conclusions about fundamental differences in the performance of MapReduce and parallel DBMS.

**Fault Tolerance**

The MapReduce implementation uses a pull model for moving data between mappers and reducers, as opposed to a push model where mappers write directly to reducers. Pavlo et al. correctly pointed out that the pull model can result in the creation of many small files and many disk seeks to move data between mappers and reducers. Imple-
Merging results. The measurements of Hadoop in all five benchmarks in the comparison paper included the cost of a final phase to merge the results of the initial MapReduce into one file. In practice, this merging is unnecessary, since the next consumer of MapReduce output is usually another MapReduce that can easily operate over the set of files produced by the first MapReduce, instead of requiring a single merged input. Even if the consumer is not another MapReduce, the reducer processes in the initial MapReduce can write directly to a merged destination (such as a Bigtable or parallel database table).

Data loading. The DBMS measurements in the comparison paper demonstrated the high cost of loading input data into a database before it is analyzed. For many of the benchmarks in the comparison paper, the time needed to load the input data into a parallel database is five to 50 times the time needed to analyze the data via Hadoop. Put another way, for some of
the benchmarks, starting with data in a collection of files on disk, it is possible to run 50 separate MapReduce analyses over the data before it is possible to load the data into a database and complete a single analysis. Long load times may not matter if many queries will be run on the data after loading, but this is often not the case; data sets are often generated, processed once or twice, and then discarded. For example, the Web-search index-building system described in the MapReduce paper is a sequence of MapReduce phases where the output of most phases is consumed by one or two subsequent MapReduce phases.

**Conclusion**

The conclusions about performance in the comparison paper were based on flawed assumptions about MapReduce and overstated the benefit of parallel database systems. In our experience, MapReduce is a highly effective and efficient tool for large-scale fault-tolerant data analysis. However, a few useful lessons can be drawn from this discussion:

*Startup latency.* MapReduce implementations should strive to reduce startup latency by using techniques like worker processes that are reused across different invocations;

*Data shuffling.* Careful attention must be paid to the implementation of the data-shuffling phase to avoid generating $O(M*R)$ seeks in a MapReduce with $M$ map tasks and $R$ reduce tasks;

*Textual formats.* MapReduce users should avoid using inefficient textual formats;

*Natural indices.* MapReduce users should take advantage of natural indices (such as timestamps in log file names) whenever possible; and

*Unmerged output.* Most MapReduce output should be left unmerged, since there is no benefit to merging if the next consumer is another MapReduce program.

MapReduce provides many significant advantages over parallel databases. First and foremost, it provides fine-grain fault tolerance for large jobs; failure in the middle of a multi-hour execution does not require restarting the job from scratch. Second, MapReduce is very useful for handling data processing and data loading in a heterogenous system with many different storage systems. Third, MapReduce provides a good framework for the execution of more complicated functions than are supported directly in SQL.

**References**


6. Devitt, D. and Stonebraker, M. MapReduce II blogpost; http://databasecolumn.vertica.com/database-innovation/mapreduce-ii/


Jeffrey Dean (jeff@google.com) is a Google Fellow in the Systems Infrastructure Group of Google, Mountain View, CA.

Sanjay Ghemawat (sanjay@google.com) is a Google Fellow in the Systems Infrastructure Group of Google, Mountain View, CA.

© 2010 ACM 0001-0782/10/0100 $10.00

JANUARY 2010 | VOL. 53 | NO. 1 | COMMUNICATIONS OF THE ACM 77