DOI:10.1145/1629175.1629198

MapReduce advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs.

BY JEFFREY DEAN AND SANJAY GHEMAWAT

MapReduce: A Flexible Data Processing Tool

MAPREDUCE IS A programming model for processing and generating large data sets.[4] 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.[10,11]

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:

```
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));
```

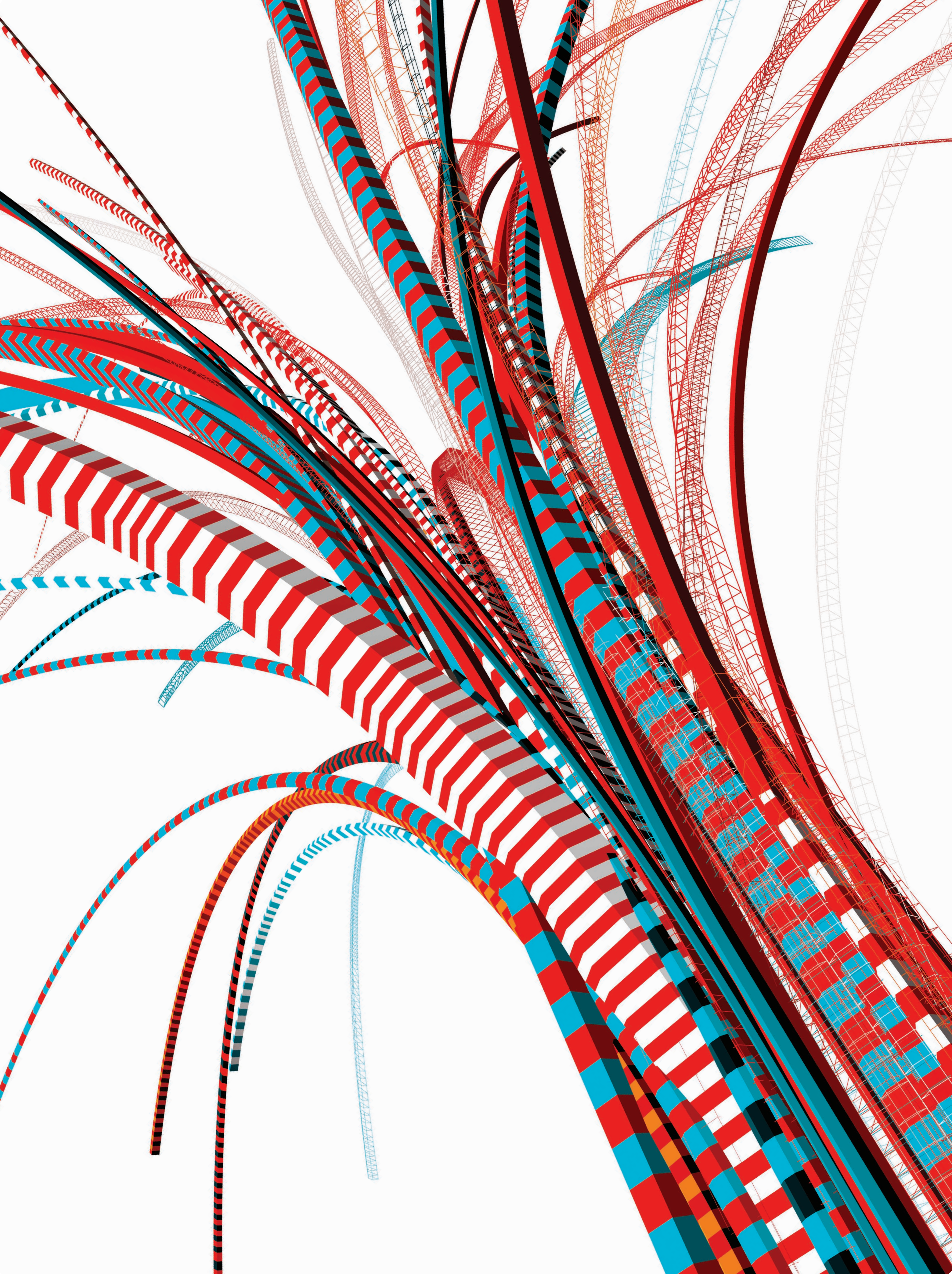
The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.

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

ILLUSTRATION BY MARIUS WATZ



express the type of computations supported by MapReduce. A 2009 paper by Andrew Pavlo et al. (referred to here as the "comparison paper"[13]) compared the performance of MapReduce and parallel databases. It evaluated the open source Hadoop implementation[10] 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."[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[4] and the comparison paper[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,[7] database query results,[2,9] data stored in Bigtable,[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.[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[14] and Pig Latin[12]) 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 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:

```
  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.[8] 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:

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

The following Map function fragment processes a Rankings record:

```
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:

```
// Fragment 1: protocol buf-
fer parsing
for (int i = 0; i < numItera-
tions; i++) {
  rankings.parseFrom(value);
  pagerank = rankings.get-
  Pagerank();
}

// Fragment 2: text for-
mat parsing (extracted from
Benchmark1.java
// from the source code
posted by Pavlo et al.)
for (int i = 0; i < numItera-
tions; i++) {
  String data[] = value.to-
  String().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.

MapReduce is a highly effective and efficient tool for large-scale fault-tolerant data analysis.

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. Implementation tricks like batching, sorting, and grouping of intermediate data and smart scheduling of reads are used by Google's MapReduce implementation to mitigate these costs.

MapReduce implementations tend not to use a push model due to the fault-tolerance properties required by Google's developers. Most MapReduce executions over large data sets encounter at least a few failures; apart from hardware and software problems, Google's cluster scheduling system can preempt MapReduce tasks by killing them to make room for higher-priority tasks. In a push model, failure of a reducer would force re-execution of all Map tasks.

We suspect that as data sets grow larger, analyses will require more computation, and fault tolerance will become more important. There are already more than a dozen distinct data sets at Google more than 1PB in size and dozens more hundreds of TBs in size that are processed daily using MapReduce. Outside of Google, many users listed on the Hadoop users list[11] are handling data sets of multiple hundreds of terabytes or more. Clearly, as data sets continue to grow, more users will need a fault-tolerant system like MapReduce that can be used to process these large data sets efficiently and effectively.

Performance

Pavlo et al. compared the performance of the Hadoop MapReduce implementation to two database implementations; here, we discuss the performance differences of the various systems:

Engineering considerations. Startup overhead and sequential scanning speed are indicators of maturity of implementation and engineering tradeoffs, not fundamental differences in programming models. These differences are certainly important but can be addressed in a variety of ways. For example, startup overhead can be addressed by keeping worker processes live, waiting for the next MapReduce invocation, an optimization added more than a year ago to Google's MapReduce implementation.

Google has also addressed sequential scanning performance with a variety of performance optimizations by, for example, using efficient binary-encoding format for structured data (protocol buffers) instead of inefficient textual formats.

Reading unnecessary data. The comparison paper says, "MR is always forced to start a query with a scan of the entire input file." MapReduce does not require a full scan over the data; it requires only an implementation of its input interface to yield a set of records that match some input specification. Examples of input specifications are:

- All records in a set of files;
- All records with a visit-date in the range [2000-01-15..2000-01-22]; and
- All data in Bigtable table T whose "language" column is "Turkish."

The input may require a full scan over a set of files, as Pavlo et al. suggested, but alternate implementations are often used. For example, the input may be a database with an index that provides efficient filtering or an indexed file structure (such as daily log files used for efficient date-based filtering of log data).

This mistaken assumption about MapReduce affects three of the five benchmarks in the comparison paper (the selection, aggregation, and join tasks) and invalidates the conclusions in the paper about the relative performance of MapReduce and parallel databases.

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[4] 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. C

MapReduce 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.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Silberschatz, A., and Rasin, A. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *Proceedings of the Conference on Very Large Databases* (Lyon, France, 2009); http://db.cs.yale.edu/hadoopdb/
2. Aster Data Systems, Inc. *In-Database MapReduce for Rich Analytics*; http://www.asterdata.com/product/mapreduce.php.
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E. Bigtable: A distributed storage system for structured data. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation* (Seattle, WA, Nov. 6–8). Usenix Association, 2006; http://labs.google.com/papers/bigtable.html
4. Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation* (San Francisco, CA, Dec. 6–8). Usenix Association, 2004; http://labs.google.com/papers/mapreduce.html
5. Dewitt, D. and Stonebraker, M. MapReduce: A Major Step Backwards blogpost; http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/
6. Dewitt, D. and Stonebraker, M. MapReduce II blogpost; http://databasecolumn.vertica.com/database-innovation/mapreduce-ii/
7. Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Lake George, NY, Oct. 19–22). ACM Press, New York, 2003; http://labs.google.com/papers/gfs.html
8. Google. Protocol Buffers: Google's Data Interchange Format. Documentation and open source release; http://code.google.com/p/protobuf/
9. Greenplum. Greenplum MapReduce: Bringing Next-Generation Analytics Technology to the Enterprise; http://www.greenplum.com/resources/mapreduce/
10. Hadoop. Documentation and open source release; http://hadoop.apache.org/core/
11. Hadoop. Users list; http://wiki.apache.org/hadoop/PoweredBy
12. Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD 2008 International Conference on Management of Data* (Auckland, New Zealand, June 2008); http://hadoop.apache.org/pig/
13. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., and Stonebraker, M. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference* (Providence, RI, June 29–July 2). ACM Press, New York, 2009; http://database.cs.brown.edu/projects/mapreduce-vs-dbms/
14. Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure 13*, 4, 227–298. http://labs.google.com/papers/sawzall.html

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.