



DOI:10.1145/1629175.1629197

MapReduce complements DBMSs since databases are not designed for extract-transform-load tasks, a MapReduce specialty.

BY MICHAEL STONEBRAKER, DANIEL ABADI,
DAVID J. DEWITT, SAM MADDEN, ERIK PAULSON,
ANDREW PAVLO, AND ALEXANDER RASIN

MapReduce and Parallel DBMSs: Friends or Foes?

THE MAPREDUCE⁷ (MR) PARADIGM has been hailed as a revolutionary new platform for large-scale, massively parallel data access.¹⁶ Some proponents claim the extreme scalability of MR will relegate relational database management systems (DBMS) to the status of legacy technology. At least one enterprise, Facebook, has implemented a large data warehouse system using MR technology rather than a DBMS.¹⁴

Here, we argue that using MR systems to perform tasks that are best suited for DBMSs yields less than satisfactory results,¹⁷ concluding that MR is more like an extract-transform-load (ETL) system than a

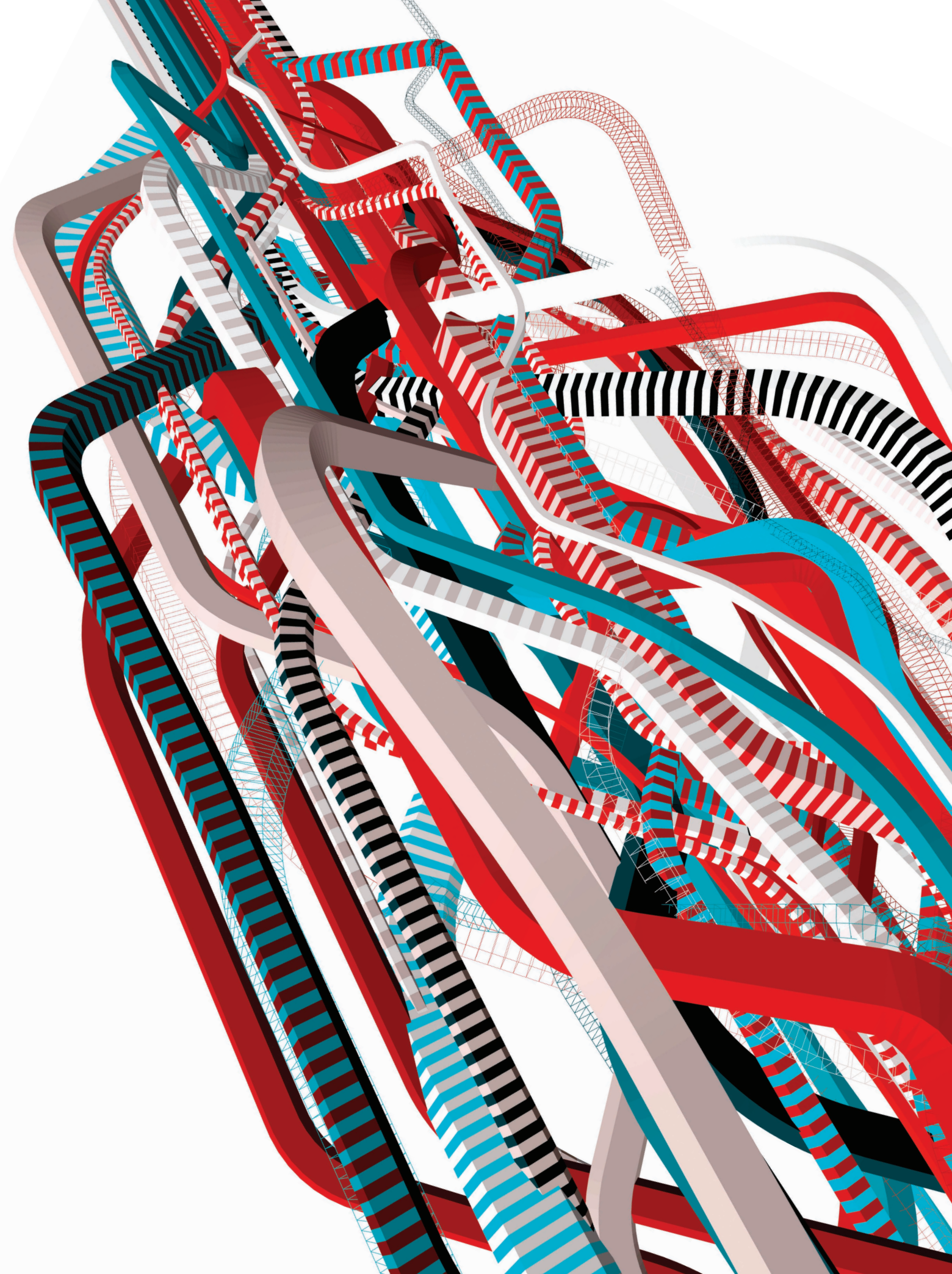
DBMS, as it quickly loads and processes large amounts of data in an ad hoc manner. As such, it complements DBMS technology rather than competes with it. We also discuss the differences in the architectural decisions of MR systems and database systems and provide insight into how the systems should complement one another.

The technology press has been focusing on the revolution of “cloud computing,” a paradigm that entails the harnessing of large numbers of processors working in parallel to solve computing problems. In effect, this suggests constructing a data center by lining up a large number of low-end servers, rather than deploying a smaller set of high-end servers. Along with this interest in clusters has come a proliferation of tools for programming them. MR is one such tool, an attractive option to many because it provides a simple model through which users are able to express relatively sophisticated distributed programs.

Given the interest in the MR model both commercially and academically, it is natural to ask whether MR systems should replace parallel database systems. Parallel DBMSs were first available commercially nearly two decades ago, and, today, systems (from about a dozen vendors) are available. As robust, high-performance computing platforms, they provide a high-level programming environment that is inherently parallelizable. Although it might seem that MR and parallel DBMSs are different, it is possible to write almost any parallel-processing task as either a set of database queries or a set of MR jobs.

Our discussions with MR users lead us to conclude that the most common use case for MR is more like an ETL system. As such, it is complementary to DBMSs, not a competing technology, since databases are not designed to be good at ETL tasks. Here, we describe what we believe is the ideal use of MR technology and highlight the different MR and parallel DBMS markets.

ILLUSTRATION BY MARIUS WATZ



We recently conducted a benchmark study using a popular open-source MR implementation and two parallel DBMSs.¹⁷ The results show that the DBMSs are substantially faster than the MR system once the data is loaded, but that loading the data takes considerably longer in the database systems. Here, we discuss the source of these performance differences, including the limiting architectural factors we perceive in the two classes of system, and conclude with lessons the MR and DBMS communities can learn from each other, along with future trends in large-scale data analysis.

Parallel Database Systems

In the mid-1980s the Teradata²⁰ and Gamma projects⁹ pioneered a new architectural paradigm for parallel database systems based on a cluster of commodity computers called “shared-nothing nodes” (or separate CPU, memory, and disks) connected through a high-speed interconnect.¹⁹ Every parallel database system built since then essentially uses the techniques first pioneered by these two projects: horizontal partitioning of relational tables, along with the partitioned execution of SQL queries.

The idea behind horizontal partitioning is to distribute the rows of a relational table across the nodes of the cluster so they can be processed in parallel. For example, partitioning

a 10-million-row table across a cluster of 50 nodes, each with four disks, would place 50,000 rows on each of the 200 disks. Most parallel database systems offer a variety of partitioning strategies, including hash, range, and round-robin partitioning.⁸ Under a hash-partitioning physical layout, as each row is loaded, a hash function is applied to one or more attributes of each row to determine the target node and disk where the row should be stored.

The use of horizontal partitioning of tables across the nodes of a cluster is critical to obtaining scalable performance of SQL queries⁸ and leads naturally to the concept of partitioned execution of the SQL operators: selection, aggregation, join, projection, and update. As an example how data partitioning is used in a parallel DBMS, consider the following SQL query:

```
SELECT custId, amount FROM Sales
WHERE date BETWEEN
    "12/1/2009" AND "12/25/2009";
```

With the Sales table horizontally partitioned across the nodes of the cluster, this query can be trivially executed in parallel by executing a SELECT operator against the Sales records with the specified date predicate on each node of the cluster. The intermediate results from each node are then sent to a single node that performs a MERGE operation in order to

return the final result to the application program that issued the query.

Suppose we would like to know the total sales amount for each custId within the same date range. This is done through the following query:

```
SELECT custId, SUM(amount)
FROM Sales
WHERE date BETWEEN
    "12/1/2009" AND "12/25/2009"
GROUP BY custId;
```

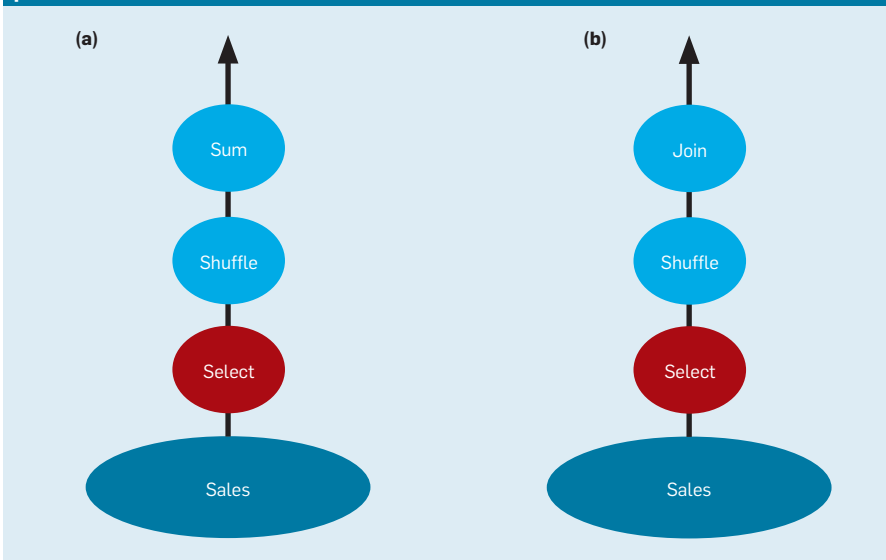
If the Sales table is round-robin partitioned across the nodes in the cluster, then the rows corresponding to any single customer will be spread across multiple nodes. The DBMS compiles this query into the three-operator pipeline in Figure(a), then executes the query plan on all the nodes in the cluster in parallel. Each SELECT operator scans the fragment of the Sales table stored at that node. Any rows satisfying the date predicate are passed to a SHUFFLE operator that dynamically repartitions the rows; this is typically done by applying a hash function on the value of the custId attribute of each row to map them to a particular node. Since the same hash function is used for the SHUFFLE operation on all nodes, rows for the same customer are routed to the single node where they are aggregated to compute the final total for each customer.

As a final example of how SQL is parallelized using data partitioning, consider the following query for finding the names and email addresses of customers who purchased an item costing more than \$1,000 during the holiday shopping period:

```
SELECT C.name, C.email FROM
Customers C, Sales S
WHERE C.custId = S.custId
AND S.amount > 1000
AND S.date BETWEEN
    "12/1/2009" AND
    "12/25/2009";
```

Assume again that the Sales table is round-robin partitioned, but we now hash-partition the Customers table on the Customer.custId attribute. The DBMS compiles this query into the operator pipeline in Figure(b) that is executed in parallel at all nodes in the cluster. Each SELECT operator scans

Parallel database query execution plans. (a) Example operator pipeline for calculating a single-table aggregate. (b) Example operator pipeline for performing a joining on two partitioned tables.



its fragment of the Sales table looking for rows that satisfy the predicate

```
S.amount > 1000 and S.date
BETWEEN "12/1/2009" and
"12/25/2009."
```

Qualifying rows are pipelined into a shuffle operator that repartitions its input rows by hashing on the Sales.custId attribute. By using the same hash function that was used when loading rows of the Customer table (hash partitioned on the Customer.custId attribute), the shuffle operators route each qualifying Sales row to the node where the matching Customer tuple is stored, allowing the join operator ($C.custId = S.custId$) to execute in parallel on all the nodes.

Another key benefit of parallel DBMSs is that the system automatically manages the various alternative partitioning strategies for the tables involved in the query. For example, if Sales and Customers are each hash-partitioned on their custId attribute, the query optimizer will recognize that the two tables are both hash-partitioned on the joining attributes and omit the shuffle operator from the compiled query plan. Likewise, if both tables are round-robin partitioned, then the optimizer will insert shuffle operators for both tables so tuples that join with one another end up on the same node. All this happens transparently to the user and to application programs.

Many commercial implementations are available, including Teradata, Netezza, DataAllegro (Microsoft), ParAccel, Greenplum, Aster, Vertica, and DB2. All run on shared-nothing clusters of nodes, with tables horizontally partitioned over them.

Mapping Parallel DBMSs onto MapReduce

An attractive quality of the MR programming model is simplicity; an MR program consists of only two functions—Map and Reduce—written by a user to process key/value data pairs.⁷ The input data set is stored in a collection of partitions in a distributed file system deployed on each node in the cluster. The program is then injected into a distributed-processing framework and executed in a manner to be described. The MR model was first popularized by Google

in 2004, and, today, numerous open source and commercial implementations are available. The most popular MR system is Hadoop, an open-source project under development by Yahoo! and the Apache Software Foundation (<http://hadoop.apache.org/>).

The semantics of the MR model are not unique, as the filtering and transformation of individual data items (tuples in tables) can be executed by a modern parallel DBMS using SQL. For Map operations not easily expressed in SQL, many DBMSs support user-defined functions¹⁸; UDF extensibility provides the equivalent functionality of a Map operation. SQL aggregates augmented with UDFs and user-defined aggregates provide DBMS users the same MR-style reduce functionality. Lastly, the reshuffle that occurs between the Map and Reduce tasks in MR is equivalent to a GROUP BY operation in SQL. Given this, parallel DBMSs provide the same computing model as MR, with the added benefit of using a declarative language (SQL).

The linear scalability of parallel DBMSs has been widely touted for two decades¹⁰; that is, as nodes are added to an installation, the database size can be increased proportionally while maintaining constant response times. Several production databases in the multi-petabyte range are run by very large customers operating on clusters of order 100 nodes.¹³ The people who manage these systems do not report the need for additional parallelism. Thus, parallel DBMSs offer great scalability over the range of nodes that customers desire. There is no reason why scalability cannot be increased dramatically to the levels reported by Jeffrey Dean and Sanjay Ghemawat,⁷ assuming there is customer demand.

Possible Applications

Even though parallel DBMSs are able to execute the same semantic workload as MR, several application classes are routinely mentioned as possible use cases in which the MR model might be a better choice than a DBMS. We now explore five of these scenarios, discussing the ramifications of using one class of system over another:

ETL and “read once” data sets. The canonical use of MR is characterized

by the following template of five operations:

- ▶ Read logs of information from several different sources;
- ▶ Parse and clean the log data;
- ▶ Perform complex transformations (such as “sessionalization”);
- ▶ Decide what attribute data to store; and
- ▶ Load the information into a DBMS or other storage engine.

These steps are analogous to the extract, transform, and load phases in ETL systems; the MR system is essentially “cooking” raw data into useful information that is consumed by another storage system. Hence, an MR system can be considered a general-purpose parallel ETL system.

For parallel DBMSs, many products perform ETL, including Ascential, Informatica, Jaspersoft, and Talend. The market is large, as almost all major enterprises use ETL systems to load large quantities of data into data warehouses. One reason for this symbiotic relationship is the clear distinction as to what each class of system provides to users: DBMSs do not try to do ETL, and ETL systems do not try to do DBMS services. An ETL system is typically upstream from a DBMS, as the load phase usually feeds data directly into a DBMS.

Complex analytics. In many data-mining and data-clustering applications, the program must make multiple passes over the data. Such applications cannot be structured as single SQL aggregate queries, requiring instead a complex dataflow program where the output of one part of the application is the input of another. MR is a good candidate for such applications.

Semi-structured data. Unlike a DBMS, MR systems do not require users to define a schema for their data. Thus, MR-style systems easily store and process what is known as “semi-structured” data. In our experience, such data often looks like key-value pairs, where the number of attributes present in any given record varies; this style of data is typical of Web traffic logs derived from disparate sources.

With a relational DBMS, one way to model such data is to use a wide table with many attributes to accommodate multiple record types. Each unrequired attribute uses NULLs for the

values that are not present for a given record. Row-based DBMSs generally have trouble with the tables, often suffering poor performance. On the other hand, column-based DBMSs (such as Vertica) mitigate the problem by reading only the relevant attributes for any query and automatically suppressing the NULL values.³ These techniques have been shown to provide good performance on RDF data sets,² and we expect the same would be true for simple key-value data.

To the extent that semistructured data fits the “cooking” paradigm discussed earlier (that is, the data is prepared for loading into a back-end data-processing system), then MR-style systems are a good fit. If the semistructured data set is primarily for analytical queries, we expect a parallel column store to be a better solution.

Quick-and-dirty analyses. One disappointing aspect of many current parallel DBMSs is that they are difficult to install and configure properly, as users are often faced with a myriad of tuning parameters that must be set correctly for the system to operate effectively. From our experiences with installing two commercial parallel systems, an open-source MR implementation provides the best “out-of-the-box” experience¹⁷; that is, we were able to get the MR system up and running queries significantly faster than either of the DBMSs. In fact, it was not until we received expert support from one of the vendors that we were able to get one particular DBMS to run queries that completed in minutes, rather than hours or days.

Once a DBMS is up and running properly, programmers must still write a schema for their data (if one does not already exist), then load the data set into the system. This process takes considerably longer in a DBMS than in an MR system, because the DBMS must parse and verify each datum in the tuples. In contrast, the default (therefore most common) way for MR programmers to load their data is to just copy it into the MR system’s underlying distributed block-based storage system.

If a programmer must perform some one-off analysis on transient data, then the MR model’s quick startup time is clearly preferable. On the other hand,

professional DBMS programmers and administrators are more willing to pay in terms of longer learning curves and startup times, because the performance gains from faster queries offset the upfront costs.

Limited-budget operations. Another strength of MR systems is that most are open source projects available for free. DBMSs, and in particular parallel DBMSs, are expensive; though there are good single-node open source solutions, to the best of our knowledge, there are no robust, community-supported parallel DBMSs. Though enterprise users with heavy demand and big budgets might be willing to pay for a commercial system and all the tools, support, and service agreements those systems provide, users with more modest budgets or requirements find open source systems more attractive. The database community has missed an opportunity by not providing a more complete parallel, open source solution.

Powerful tools. MR systems are fundamentally powerful tools for ETL-style applications and for complex analytics. Additionally, they are popular for “quick and dirty” analyses and for users with limited budgets. On the other hand, if the application is query-intensive, whether semistructured or rigidly structured, then a DBMS is probably the better choice. In the next section, we discuss results from use cases that demonstrate this performance superiority; the processing tasks range from those MR systems ought to be good at to those that are quite complex queries.

DBMS “Sweet Spot”

To demonstrate the performance trade-offs between parallel DBMSs and MR systems, we published a benchmark comparing two parallel DBMSs to the Hadoop MR framework on a variety of tasks.¹⁷ We wished to discover the performance envelope of each approach when applied to areas inside and outside their target application space. We used two database systems: Vertica, a commercial column-store relational database, and DBMS-X, a row-based database from a large commercial vendor. Our benchmark study included a simple benchmark presented in the original MR paper from Google,⁷ as well as four other analyti-

cal tasks of increasing complexity we think are common processing tasks that could be done using either class of systems. We ran all experiments on a 100-node shared-nothing cluster at the University of Wisconsin-Madison. The full paper¹⁷ includes the complete results and discussion from all our experiments, including load times; here, we provide a summary of the most interesting results. (The source code for the benchmark study is available at <http://database.cs.brown.edu/projects/mapreduce-vs-dbms/>.)

Hadoop is by far the most popular publicly available version of the MR framework (the Google version might be faster but is not available to us), and DBMS-X and Vertica are popular row- and column-store parallel database systems, respectively.

In the time since publication of Pavlo et al.¹⁷ we have continued to tune all three systems. Moreover, we have received many suggestions from the Hadoop community on ways to improve performance. We have tried them all, and the results here (as of August 2009) represent the best we can do with a substantial amount of expert help on all three systems. In fact, the time we’ve spent tuning Hadoop has now exceeded the time we spent on either of the other systems. Though Hadoop offers a good out-of-the-box experience, tuning it to obtain maximum performance was an arduous task. Obviously, performance is a moving target, as new releases of all three products occur regularly.

Original MR Grep task. Our first benchmark experiment is the “Grep task” from the original MR paper, which described it as “representative of a large subset of the real programs written by users of MapReduce.”⁷ For the task, each system must scan through a data set of 100B records looking for a three-character pattern. Each record consists of a unique key in the first 10B, followed by a 90B random value. The search pattern is found only in the last 90B once in every 10,000 records. We use a 1TB data set spread over the 100 nodes (10GB/node). The data set consists of 10 billion records, each 100B. Since this is essentially a sequential search of the data set looking for the pattern, it provides a simple measurement of how

quickly a software system can scan through a large collection of records. The task cannot take advantage of any sorting or indexing and is easy to specify in both MR and SQL. Therefore, one would expect a lower-level interface (such as Hadoop) running directly on top of the file system (HDFS) to execute faster than the more heavyweight DBMSs.

However, the execution times in the table here show a surprising result: The database systems are about two times faster than Hadoop. We explain some of the reasons for this conclusion in the section on architectural differences.

Web log task. The second task is a conventional SQL aggregation with a GROUP BY clause on a table of user visits in a Web server log. Such data is fairly typical of Web logs, and the query is commonly used in traffic analytics. For this experiment, we used a 2TB data set consisting of 155 million records spread over the 100 nodes (20GB/node). Each system must calculate the total ad revenue generated for each visited IP address from the logs. Like the previous task, the records must all be read, and thus there is no indexing opportunity for the DBMSs. One might think that Hadoop would excel at this task since it is a straightforward calculation, but the results in the table show that Hadoop is beaten by the databases by a larger margin than in the Grep task.

Join task. The final task we discuss here is a fairly complex join operation over two tables requiring an additional aggregation and filtering operation. The user-visit data set from the previous task is joined with an additional 100GB table of PageRank values for 18 million URLs (1GB/node). The join task consists of two subtasks that perform a complex calculation on the two data sets. In the first part of the task, each system must find the IP address that generated the most revenue within a particular date range in the user visits. Once these intermediate records are generated, the system must then calculate the average PageRank of all pages visited during this interval.

DBMSs ought to be good at analytical queries involving complex join operations (see the table). The DBMSs are a factor of 36 and 21 respectively faster than Hadoop. In general, query times

Benchmark performance on a 100-node cluster.

	Hadoop	DBMS-X	Vertica	Hadoop/DBMS-X	Hadoop/Vertica
Grep	284s	194s	108x	1.5x	2.6x
Web Log	1,146s	740s	268s	1.6x	4.3x
Join	1,158s	32s	55s	36.3x	21.0x

for a typical user task fall somewhere in between these extremes. In the next section, we explore the reasons for these results.

Architectural Differences

The performance differences between Hadoop and the DBMSs can be explained by a variety of factors. Before delving into the details, we should say these differences result from implementation choices made by the two classes of system, not from any fundamental difference in the two models. For example, the MR processing model is independent of the underlying storage system, so data could theoretically be massaged, indexed, compressed, and carefully laid out on storage during a load phase, just like a DBMS. Hence, the goal of our study was to compare the real-life differences in performance of representative realizations of the two models.

Repetitive record parsing. One contributing factor for Hadoop's slower performance is that the default configuration of Hadoop stores data in the accompanying distributed file system (HDFS), in the same textual format in which the data was generated. Consequently, this default storage method places the burden of parsing the fields of each record on user code. This parsing task requires each Map and Reduce task repeatedly parse and convert string fields into the appropriate type. Hadoop provides the ability to store data as key/value pairs as serialized tuples called SequenceFiles, but despite this ability it still requires user code to parse the value portion of the record if it contains multiple attributes. Thus, we found that using SequenceFiles without compression consistently yielded slower performance on our benchmark. Note that using SequenceFiles without compression was but one of the tactics for possibly improving Ha-

doop's performance suggested by the MR community.

In contrast to repetitive parsing in MR, records are parsed by DBMSs when the data is initially loaded. This initial parsing step allows the DBMSs storage manager to carefully lay out records in storage such that attributes can be directly addressed at runtime in their most efficient storage representation. As such, there is no record interpretation performed during query execution in parallel DBMSs.

There is nothing fundamental about the MR model that says data cannot be parsed in advance and stored in optimized data structures (that is, trading off some load time for increased runtime performance). For example, data could be stored in the underlying file system using Protocol Buffers (<http://code.google.com/p/protobuf/>), Google's platform-neutral, extensible mechanism for serializing structured data; this option is not available in Hadoop. Alternatively, one could move the data outside the MR framework into a relational DBMS at each node, thereby replacing the HDFS storage layer with DBMS-style optimized storage for structured data.⁴

There may be ways to improve the Hadoop system by taking advantage of these ideas. Hence, parsing overhead is a problem, and SequenceFiles are not an effective solution. The problem should be viewed as a signpost for guiding future development.


Compression. We found that enabling data compression in the DBMSs delivered a significant performance gain. The benchmark results show that using compression in Vertica and DBMS-X on these workloads improves performance by a factor of two to four. On the other hand, Hadoop often executed slower when we used compression on its input files; at most, compression improved performance by 15%; the benchmark results in Dean and Ghemawat⁷ also

did not use compression.


It is unclear to us why this improvement was insignificant, as essentially all commercial SQL data warehouses use compression to improve performance. We postulate that commercial DBMSs use carefully tuned compression algorithms to ensure that the cost of decompressing tuples does not offset the performance gains from the reduced I/O cost of reading compressed data. For example, we have found that on modern processors standard Unix implementations of *gzip* and *bzip* are often too slow to provide any benefit.

Pipelining. All parallel DBMSs operate by creating a query plan that is distributed to the appropriate nodes at execution time. When one operator in this plan must send data to the next operator, regardless of whether that operator is running on the same or a different node, the qualifying data is “pushed” by the first operator to the second operator. Hence, data is streamed from producer to consumer; the intermediate data is never written to disk; the resulting “back-pressure” in the runtime system will stall the producer before it has a chance to overrun the consumer. This streaming technique differs from the approach taken in MR systems, where the producer writes the intermediate results to local data structures, and the consumer subsequently “pulls” the data. These data structures are often quite large, so the system must write them out to disk, introducing a potential bottleneck. Though writing data structures to disk gives Hadoop a convenient way to checkpoint the output of intermediate map jobs, thereby improving fault tolerance, we found from our investigation that it adds significant performance overhead.

Scheduling. In a parallel DBMS, each node knows exactly what it must do and when it must do it according to the distributed query plan. Because the operations are known in advance, the system is able to optimize the execution plan to minimize data transmission between nodes. In contrast, each task in an MR system is scheduled on processing nodes one storage block at a time. Such runtime work scheduling at a granularity of storage blocks is much more expensive than the DBMS



The commercial DBMS products must move toward one-button installs, automatic tuning that works correctly, better Web sites with example code, better query generators, and better documentation.



compile-time scheduling. The former has the advantage, as some have argued,⁴ of allowing the MR scheduler to adapt to workload skew and performance differences between nodes.

Column-oriented storage. In a column store-based database (such as Vertica), the system reads only the attributes necessary for solving the user query. This limited need for reading data represents a considerable performance advantage over traditional, row-stored databases, where the system reads all attributes off the disk. DBMS-X and Hadoop/HDFS are both essentially row stores, while Vertica is a column store, giving Vertica a significant advantage over the other two systems in our Web log benchmark task.

Discussion. The Hadoop community will presumably fix the compression problem in a future release. Furthermore, some of the other performance advantages of parallel databases (such as column-storage and operating directly on compressed data) can be implemented in an MR system with user code. Also, other implementations of the MR framework (such as Google’s proprietary implementation) may well have a different performance envelope. The scheduling mechanism and pull model of data transmission are fundamental to the MR block-level fault-tolerance model and thus unlikely to be changed.

Meanwhile, DBMSs offer transaction-level fault tolerance. DBMS researchers often point out that as databases get bigger and the number of nodes increases, the need for finer-granularity fault tolerance increases as well. DBMSs readily adapt to this need by marking one or more operators in a query plan as “restart operators.” The runtime system saves the result of these operators to disk, facilitating “operator level” restart. Any number of operators can be so marked, allowing the granularity of restart to be tuned. Such a mechanism is easily integrated into the efficient query execution framework of DBMSs while allowing variable granularity restart. We know of at least two separate research groups, one at the University of Washington, the other at the University of California, Berkeley, that are exploring the trade-off between runtime overhead and the amount of work lost when a failure occurs.

We generally expect ETL and complex analytics to be amenable to MR systems and query-intensive workloads to be run by DBMSs. Hence, we expect the best solution is to interface an MR framework to a DBMS so MR can do complex analytics, and interface to a DBMS to do embedded queries. HadoopDB,⁴ Hive,²¹ Aster, Greenplum, Cloudera, and Vertica all have commercially available products or prototypes in this “hybrid” category.

Learning from Each Other

What can MR learn from DBMSs? MR advocates should learn from parallel DBMS the technologies and techniques for efficient query parallel execution. Engineers should stand on the shoulders of those who went before, rather than on their toes. There are many good ideas in parallel DBMS executors that MR system developers would be wise to adopt.

We also feel that higher-level languages are invariably a good idea for any data-processing system. Relational DBMSs have been fabulously successful in pushing programmers to a higher, more-productive level of abstraction, where they simply state what they want from the system, rather than writing an algorithm for how to get what they want from the system. In our benchmark study, we found that writing the SQL code for each task was substantially easier than writing MR code.

Efforts to build higher-level interfaces on top of MR/Hadoop should be accelerated; we applaud Hive,²¹ Pig,¹⁵ Scope,⁶ Dryad/Linq,¹² and other projects that point the way in this area.

What can DBMSs learn from MR? The out-of-the-box experience for most DBMSs is less than ideal for being able to quickly set up and begin running queries. The commercial DBMS products must move toward one-button installs, automatic tuning that works correctly, better Web sites with example code, better query generators, and better documentation.

Most database systems cannot deal with tables stored in the file system (in situ data). Consider the case where a DBMS is used to store a very large data set on which a user wishes to perform analysis in conjunction with a smaller, private data set. In order to access the


larger data set, the user must first load the data into the DBMS. Unless the user plans to run many analyses, it is preferable to simply point the DBMS at data on the local disk without a load phase. There is no good reason DBMSs cannot deal with in situ data. Though some database systems (such as PostgreSQL, DB2, and SQL Server) have capabilities in this area, further flexibility is needed.

Conclusion

Most of the architectural differences discussed here are the result of the different focuses of the two classes of system. Parallel DBMSs excel at efficient querying of large data sets; MR-style systems excel at complex analytics and ETL tasks. Neither is good at what the other does well. Hence, the two technologies are complementary, and we expect MR-style systems performing ETL to live directly upstream from DBMSs.

Many complex analytical problems require the capabilities provided by both systems. This requirement motivates the need for interfaces between MR systems and DBMSs that allow each system to do what it is good at. The result is a much more efficient overall system than if one tries to do the entire application in either system. That is, “smart software” is always a good idea.

Acknowledgment

This work is supported in part by National Science Foundation grants CRI-0707437, CluE-0844013, and CluE-0844480. 

References

1. Abadi, D.J., Madden, S.R., and Hachem, N. Column-stores vs. row-stores: How different are they really? In *Proceedings of the SIGMOD Conference on Management of Data*. ACM Press, New York, 2008.
2. Abadi, D.J., Marcus, A., Madden, S.R., and Hollenbach, K. Scalable semantic Web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Databases*, 2007.
3. Abadi, D.J. Column-stores for wide and sparse data. In *Proceedings of the Conference on Innovative Data Systems Research*, 2007.
4. Abouzaid, 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*, 2009.
5. Boral, H. et al. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (Mar. 1990), 4–24.
6. Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of the Conference on Very Large Databases*, 2008.
7. Dean, J. and Ghemawat, S. MapReduce: Simplified

data processing on large clusters. In *Proceedings of the Sixth Conference on Operating System Design and Implementation* (Berkeley, CA, 2004).

8. DeWitt, D.J. and Gray, J. Parallel database systems: The future of high-performance database systems. *Commun. ACM* 35, 6 (June 1992), 85–98.
9. DeWitt, D.J., Gerber, R.H., Graefe, G., Heytens, M.L., Kumar, K.B., and Muralikrishna, M. GAMMA: A high-performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Databases*. Morgan Kaufmann Publishers, Inc., 1986, 228–237.
10. Englert, S., Gray, J., Kocher, T., and Shah, P. A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. *Sigmetrics Performance Evaluation Review* 18, 1 (1990), 245–246.
11. Fushimi, S., Kitsuregawa, M., and Tanaka, H. An overview of the system software of a parallel relational database machine. In *Proceedings of the 12th International Conference on Very Large Databases*. Morgan Kaufmann Publishers, Inc., 1986, 209–219.
12. Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Operating System Review* 41, 3 (2007), 59–72.
13. Monash, C. Some very, very, very large data warehouses. In *NetworkWorld.com community blog*, May 12, 2009; <http://www.networkworld.com/community/node/41777>.
14. Monash, C. Cloudera presents the MapReduce bull case. In *DBMS2.com blog*, Apr. 15, 2009; <http://www.dbms2.com/2009/04/15/cloudera-presents-the-mapreduce-bull-case/>.
15. 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 SIGMOD Conference*. ACM Press, New York, 2008, 1099–1110.
16. Patterson, D.A. Technical perspective: The data center is the computer. *Commun. ACM* 51, 1 (Jan. 2008), 105.
17. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S.R., and Stonebraker, M. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*. ACM Press, New York, 2009, 165–178.
18. Stonebraker, M. and Rowe, L. The design of Postgres. In *Proceedings of the SIGMOD Conference*, 1986, 340–355.
19. Stonebraker, M. The case for shared nothing. *Data Engineering* 9 (Mar. 1986), 4–9.
20. Teradata Corp. *Database Computer System Manual, Release 1.3*. Los Angeles, CA, Feb. 1985.
21. Thusoo, A. et al. Hive: A warehousing solution over a Map-Reduce framework. In *Proceedings of the Conference on Very Large Databases*, 2009, 1626–1629.

Michael Stonebraker (stonebraker@csail.mit.edu) is an adjunct professor in the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology, Cambridge, MA.

Daniel J. Abadi (dna@cs.yale.edu) is an assistant professor in the Department of Computer Science at Yale University, New Haven, CT.

David J. DeWitt (dewitt@microsoft.com) is a technical fellow in the Jim Gray Systems Lab at Microsoft Inc., Madison, WI.

Samuel Madden (madden@csail.mit.edu) is a professor in the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology, Cambridge, MA.

Erik Paulson (epaulson@cs.wisc.edu) is a Ph.D. candidate in the Department of Computer Sciences at the University of Wisconsin-Madison, Madison, WI.

Andrew Pavlo (pavlo@cs.brown.edu) is a Ph.D. candidate in the Department of Computer Science at Brown University, Providence, RI.

Alexander Rasin (alex@cs.brown.edu) is a Ph.D. candidate in the Department of Computer Science at Brown University, Providence, RI.