MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM Permalink[1] | Comments (44) | TrackBacks (1)

[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here with our views on MapReduce[2]. This is a good time to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of "jelly beans" rather than utilizing a much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to teach students how to program such clusters using a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on

First, we will briefly discuss what MapReduce is; then we will go into more detail about our five reactions listed above.

What is MapReduce?

The basic idea of MapReduce is straightforward. It consists of two programs that the user writes called *map* and *reduce* plus a framework for executing a possibly large number of instances of each program on a compute cluster.

The map program reads a set of "records" from an input file, does any desired filtering and/or transformations, and then outputs a set of records of the form (key, data). As the map program produces output records, a "split" function partitions the records into *M* disjoint buckets by applying a function to the key of each output record. This split function is typically a hash function, though any deterministic function will suffice. When a bucket fills, it is written to disk. The map program terminates with *M* output files, one for each bucket.

In general, there are multiple instances of the map program running on different nodes of a compute cluster. Each map instance is given a distinct portion of the input file by the MapReduce scheduler to process. If *N* nodes participate in the map phase, then there are *M* files on disk storage at each of *N* nodes, for a total of *N* * *M* files; $F_{i,j}$, $1 \leq i \leq N$, $1 \leq j \leq M$.

The key thing to observe is that all map instances use the same hash function. Hence, all output records with the same hash value will be in corresponding output files.

The second phase of a MapReduce job executes *M* instances of the reduce program, R_j, $1 \leq j \leq M$. The input for each reduce instance R_j consists of the files $F_{i,j}$, $1 \leq i \leq N$. Again notice that all output records from the map phase with the same hash value will be consumed by the same reduce instance -- no matter which map instance produced them. After being collected by the map-reduce framework, the input records to a reduce instance are grouped on their keys (by sorting or hashing) and feed to the reduce program. Like the map program, the reduce program is an arbitrary computation in a general-purpose language. Hence, it can do anything it wants with its records. For example, it might compute some additional function over other data fields in the record. Each reduce instance can write records to an output file, which forms part of the "answer" to a MapReduce computation.

To draw an analogy to SQL, map is like the *group-by* clause of an aggregate query. Reduce is analogous to the *aggregate* function (e.g., average) that is computed over all the rows with the same group-by attribute.

We now turn to the five concerns we have with this computing paradigm.

1. MapReduce is a step backwards in database access

As a data processing paradigm, MapReduce represents a giant step backwards. The database community has learned the following three lessons from the 40 years that have unfolded since IBM first released IMS in 1968.

- Schemas are good.
- Separation of the schema from the application is good.
- High-level access languages are good.

MapReduce has learned none of these lessons and represents a throw back to the 1960s, before modern DBMSs were invented.

The DBMS community learned the importance of schemas, whereby the fields and their data types are recorded in storage. More importantly, the run-time system of the DBMS can ensure that input records obey this schema. This is the best way to keep an application from adding "garbage" to a data set. MapReduce has no such functionality, and there are no controls to keep garbage out of its data sets. A corrupted MapReduce dataset can actually silently break all the MapReduce applications that use that dataset.

It is also crucial to separate the schema from the application program. If a programmer wants to write a new application against a data set, he or she must discover the record structure. In modern DBMSs, the schema is stored in a collection of system catalogs and can be queried (in SQL) by any user to uncover such structure. In contrast, when the schema does not exist or is buried in an application program, the programmer must discover the structure by an examination of the code. Not only is this a very tedious exercise, but also the programmer must find the source code for the application. This latter tedium is forced onto every MapReduce programmer, since there are no system catalogs recording the structure of records -- if any such structure exists.

During the 1970s the DBMS community engaged in a "great debate" between the relational advocates and the Codasyl advocates. One of the key issues was whether a DBMS access program should be written:

- By stating what you want - rather than presenting an algorithm for how to get it (relational view)
- By presenting an algorithm for data access (Codasyl view)

The result is now ancient history, but the entire world saw the value of high-level languages and relational systems prevailed. Programs in high-level languages are easier to write, easier to modify, and easier for a new person to understand. Codasyl was rightly criticized for being "the assembly language of DBMS access." A MapReduce programmer is analogous to a Codasyl programmer -- he or she is writing in a low-level language performing low-level record manipulation. Nobody advocates returning to assembly language; similarly nobody should be forced to program in MapReduce.

MapReduce advocates might counter this argument by claiming that the datasets they are targeting have no schema. We dismiss this assertion. In extracting a key from the input data set, the map function is relying on the existence of at least one data field in each input record. The same holds for a reduce function that computes some value from the records it receives to process.

Writing MapReduce applications on top of Google's BigTable (or Hadoop's HBase) does not really change the situation significantly. By using a self-describing tuple format (row key, column name, {values}) different tuples within the same table can actually have different schemas. In addition, BigTable and HBase do not provide logical independence, for example with a view mechanism. Views significantly simplify keeping applications running when the logical schema changes.

2. MapReduce is a poor implementation

All modern DBMSs use hash or B-tree indexes to accelerate access to data. If one is looking for a subset of the records (e.g., those employees with a salary of 10,000 or those in the shoe department), then one can often use an index to advantage to cut down the scope of the **search** by one to two orders of magnitude. In addition, there is a query optimizer to decide whether to use an index or perform a brute-force sequential **search**.

MapReduce has no indexes and therefore has only brute force as a processing option. It will be creamed whenever an index is the better access mechanism.

One could argue that value of MapReduce is automatically providing parallel execution on a grid of computers. This feature was explored by the DBMS research community in the 1980s, and multiple prototypes were built including Gamma [2,3], Bubba [4], and Grace [5]. Commercialization of these ideas occurred in the late 1980s with systems such as Teradata.

In summary to this first point, there have been high-performance, commercial, grid-oriented SQL engines (with schemas and indexing) for the past 20 years. MapReduce does not fare well when compared with such systems.

There are also some lower-level implementation issues with MapReduce, specifically skew and data interchange.

One factor that MapReduce advocates seem to have overlooked is the issue of skew. As described in "Parallel Database System: The Future of High Performance Database Systems," [6] skew is a huge impediment to achieving successful scale-up in parallel query systems. The problem occurs in the map phase when there is wide variance in the distribution of records with the same key. This variance, in turn, causes some reduce instances to take much longer to run than others, resulting in the execution time for the computation being the running time of the slowest reduce instance. The parallel database community has studied this problem extensively and has developed solutions that the MapReduce community might want to adopt.

There is a second serious performance problem that gets glossed over by the MapReduce proponents. Recall that each of the *N* map instances produces *M* output files -- each destined for a different reduce instance. These files are written to a disk local to the computer used to run the map instance. If *N* is 1,000 and *M* is 500, the map phase produces 500,000 local files. When the reduce phase starts, each of the 500 reduce instances needs to read its 1,000 input files and must use a protocol like FTP to "pull" each of its input files from the nodes on which the map instances were run. With 100s of reduce instances running simultaneously, it is inevitable that two or more reduce instances will attempt to read their input files from the same map node simultaneously -- inducing large numbers of disk seeks and slowing the effective disk transfer rate by more than a factor of 20. This is why parallel database systems do not materialize their split files and use push (to sockets) instead of pull. Since much of the excellent fault-tolerance that MapReduce obtains depends on materializing its split files, it is not clear whether the MapReduce framework could be successfully modified to use the push paradigm instead.

Given the experimental evaluations to date, we have serious doubts about how well MapReduce applications can scale. Moreover, the MapReduce implementers would do well to study the last 25 years of parallel DBMS research literature.

3. MapReduce is not novel

The MapReduce community seems to feel that they have discovered an entirely new paradigm for processing large data sets. In actuality, the techniques employed by MapReduce are more than 20 years old. The idea of partitioning a large data set into smaller partitions was first proposed in "Application of Hash to Data Base Machine and Its Architecture" [11] as the basis for a new type of join algorithm. In "Multiprocessor Hash-Based Join Algorithms," [7], Gerber demonstrated how Kitsuregawa's techniques could be extended to execute joins in parallel on a shared-nothing [8] cluster using a combination of partitioned tables, partitioned execution, and hash based splitting. DeWitt [2] showed how these techniques could be adopted to execute aggregates with and without group by clauses in parallel. DeWitt and Gray [6] described parallel database systems and how they process queries. Shatdal and Naughton [9] explored alternative strategies for executing aggregates in parallel.

Teradata has been selling a commercial DBMS utilizing all of these techniques for more than 20 years; exactly the techniques that the MapReduce crowd claims to have invented.

While MapReduce advocates will undoubtedly assert that being able to write MapReduce functions is what differentiates their software from a parallel SQL implementation, we would remind them that POSTGRES supported user-defined functions and user-defined aggregates in the mid 1980s. Essentially, all modern database systems have provided such functionality for quite a while, starting with the Illustra engine around 1995.

4. MapReduce is missing features

All of the following features are routinely provided by modern DBMSs, and all are missing from MapReduce:

- **Bulk loader** -- to transform input data in files into a desired format and load it into a DBMS
- **Indexing** -- as noted above
- **Updates** -- to change the data in the data base
- **Transactions** -- to support parallel update and recovery from failures during update
- **Integrity constraints** -- to help keep garbage out of the data base
- **Referential integrity** -- again, to help keep garbage out of the data base
- **Views** -- so the schema can change without having to rewrite the application program

In summary, MapReduce provides only a sliver of the functionality found in modern DBMSs.

5. MapReduce is incompatible with the DBMS tools

A modern SQL DBMS has available all of the following classes of tools:

- **Report writers** (e.g., Crystal reports) to prepare reports for human visualization
- **Business intelligence tools** (e.g., Business Objects or Cognos) to enable ad-hoc querying of large data warehouses
- **Data mining tools** (e.g., Oracle Data Mining or IBM DB2 Intelligent Miner) to allow a user to discover structure in large data sets
- **Replication tools** (e.g., Golden Gate) to allow a user to replicate data from on DBMS to another
- **Database design tools** (e.g., Embarcadero) to assist the user in constructing a data base.

MapReduce cannot use these tools and has none of its own. Until it becomes SQL-compatible or until someone writes all of these tools, MapReduce will remain very difficult to use in an end-to-end task.

In Summary

It is exciting to see a much larger community engaged in the design and implementation of scalable query processing techniques. We, however, assert that they should not overlook the lessons of more than 40 years of database technology -- in particular the many advantages that a data model, physical and logical data independence, and a declarative query language, such as SQL, bring to the design, implementation, and maintenance of application programs. Moreover, computer science communities tend to be insular and do not read the literature of other communities. We would encourage the wider community to examine the parallel DBMS literature of the last 25 years. Last, before MapReduce can measure up to modern DBMSs, there is a large collection of unmet features and required tools that must be added.

We fully understand that database systems are not without their problems. The database community recognizes that database systems are too "hard" to use and is working to solve this problem. The database community can also learn something valuable from the excellent fault-tolerance that MapReduce provides its applications.

Finally we note that some database researchers are beginning to explore using the MapReduce framework as the basis for building scalable database systems. The Pig[10] project at Yahoo! Research is one such effort.

References

[1] "MapReduce: Simplified Data Processing on Large Clusters," Jeff Dean and Sanjay Ghemawat, Proceedings of the 2004 OSDI Conference, 2004.

[2] "The Gamma Database Machine Project," DeWitt, et. al., IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990.

[4] "Gamma - A High Performance Dataflow Database Machine," DeWitt, D, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, Proceedings of the 1986 VLDB Conference, 1986.

[5] "Prototyping Bubba, A Highly Parallel Database System," Boral, et. al., IEEE Transactions on Knowledge and Data Engineering,Vol. 2, No. 1, March 1990.

[6] "Parallel Database System: The Future of High Performance Database Systems," David J. DeWitt and Jim Gray, CACM, Vol. 35, No. 6, June 1992.

[7] "Multiprocessor Hash-Based Join Algorithms," David J. DeWitt and Robert H. Gerber, Proceedings of the 1985 VLDB Conference, 1985.

[8] "The Case for Shared-Nothing," Michael Stonebraker, Data Engineering Bulletin, Vol. 9, No. 1, 1986.

[9] "Adaptive Parallel Aggregation Algorithms," Ambuj Shatdal and Jeffrey F. Naughton, Proceedings of the 1995 SIGMOD Conference, 1995.

[10] "Pig", Chris Olston, http://research.yahoo.com/project/90

[11] "Application of Hash to Data Base Machine and Its Architecture," Masaru Kitsuregawa, Hidehiko Tanaka, Tohru Moto-Oka, New Generation Comput. 1(1): 63-74 (1983)

1. http://databasecolumn.vertica.com/2008/01/mapreduce_a_major_step_back.html
2. http://en.wikipedia.org/wiki/MapReduce