A Comparison study of Different type of Data Analysis

Ravinder Singh

Research Scholar, Manav Bharti University, H.P., INDIA

ABSTRACT: There is currently considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis. Although the basic control flow of this framework has existed in parallel SQL database management systems (DBMS) for over 20 years, some have called MR a dramatically new computing model. In this paper, we describe and compare both paradigms. Furthermore, we evaluate both kinds of systems in terms of performance and development complexity. To this end, we define a benchmark consisting of a collection of tasks that we have run on an open source version of MR as well as on two parallel DBMSs. For each task, we measure each system's performance for various degrees of parallelism on a cluster of 100 nodes. Our results reveal some interesting trade-offs. Although the process to load data into and tune the execution of parallel DBMSs took much longer than the MR system, the observed performance of these DBMSs was strikingly better. We speculate about the causes of the dramatic performance difference and consider implementation concepts that future systems should take from both kinds of architectures

--♦------------------------------------

1. INTRODUCTION

Recently the trade press has been filled with news of the revolution of "cluster 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 low-end servers instead of deploying a smaller set of high-end servers. With this rise of interest in clusters has come a proliferation of tools for programming them. One of the earliest and best known such tools in MapReduce (MR) [8]. MapReduce is attractive because it provides a simple model through which users can express relatively sophisticated distributed programs, leading to significant interest in the educational community. For example, IBM and Google have announced plans to make a 1000 processor MapReduce cluster available to teach students distributed programming.

Given this interest in MapReduce, it is natural to ask "Why not use a parallel DBMS instead?" Parallel database systems (which all share a common architectural design) have been commercially available for nearly two decades, and there are now about a dozen in the marketplace, including Teradata, Aster Data, Netezza, DATAllegro (and therefore soon Microsoft SQL Server via Project Madison), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via the Database Partitioning Feature), and Oracle (via Exadata). They are robust, high performance computing platforms. Like MapReduce, they provide a high-level programming environment and parallelize readily. Though it may seem thatMR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set ofMapReduce jobs. Inspired by this question, our goal is to understand the differences between the MapReduce approach to performing large-scale data analysis and the approach taken by parallel database systems. The two classes of systems make different choices in several key areas. For example, all DBMSs require that data conform to a well-defined schema, whereas MR permits data to be in any arbitrary format. Other differences also include how each system provides indexing and compression optimizations, programming models, the way in which data is distributed, and query execution strategies.

The purpose of this paper is to consider these choices, and the trade-offs that they entail. We begin in Section 2 with a brief review of the two alternative classes of systems, followed by a discussion in Section 3 of the architectural trade-offs. Then, in Section 4 we present our benchmark consisting of a variety of tasks, one taken from the MR paper [8], and the rest a collection of more demanding tasks. In addition, we present the results of running the benchmark on a 100-node cluster to execute each task. We tested the publicly available open-source version of MapReduce, Hadoop [1], against two parallel SQL DBMSs, Vertica [3] and a second system from a major relational vendor. We also present results on the time each system took to load the test data and report informally on the procedures needed to set up and tune the software for each task.

In general, the SQL DBMSs were significantly faster and required less code to implement each task, but took longer to tune and load the data. Hence, we conclude with a discussion on the reasons for the differences between the approaches and provide suggestions on the best practices for any large-scale data analysis engine.

Some readers may feel that experiments conducted using 100 nodes are not interesting or representative of real world data processing systems. We disagree with this conjecture on two points. First, as we demonstrate in Section 4, at 100 nodes the two parallel DBMSs range from a factor of 3.1 to 6.5 faster than MapReduce on a variety of analytic tasks. While MR may indeed be capable of scaling up to 1000s of nodes, the superior efficiency of modern DBMSs alleviates the need to use such massive hardware on datasets in the range of 1–2PB (1000 nodes with 2TB of disk/node has a total disk capacity of 2PB). For example, eBay's Teradata configuration uses just 72 nodes (two quad-core CPUs, 32GB RAM, 104 300GB disks per node) to manage approximately 2.4PB of relational data. As another example, Fox Interactive Media's warehouse is implemented using a 40-node Greenplum DBMS. Each node is a Sun X4500 machine with two dual-core CPUs, 48 500GB disks, and 16 GB RAM (1PB total disk space) [7]. Since few data sets in the world even approach a petabyte in size, it is not at all clear how many MR users really need 1,000 nodes.

2. TWO APPROACHES TO LARGE SCALE DATA ANALYSIS

The two classes of systems we consider in this paper run on a "shared nothing" collection of computers [19]. That is, the system is deployed on a collection of independent machines, each with local disk and local main memory, connected together on a highspeed local area network. Both systems achieve parallelism by dividing any data set to be utilized into partitions, which are allocated to different nodes to facilitate parallel processing. In this section, we provide an overview of how both the MR model and traditional parallel DBMSs operate in this environment.

2.1 MapReduce

One of the attractive qualities about the MapReduce programming model is its simplicity: an MR program consists only of two functions, called Map and Reduce, that are written by a user to process key/value data pairs. 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 Map function reads a set of "records" from an input file, does any desired filtering and/or transformations, and then outputs a set of intermediate records in the form of new key/value pairs. As the Map function produces these output records, a "split" function partitions the records into R 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. Each map bucket is written to the processing node's local disk. The Map function terminates having produced R output files, one for each bucket. In general, there are multiple instances of the Map function running on different nodes of a compute cluster. We use the term *instance* to mean a unique running invocation of either the Map or Reduce function. Each Map instance is assigned a distinct portion of the input file by the MR scheduler to process. If there are M such distinct portions of the input file, then there are R files on disk storage for each of the M Map tasks, for a total of M × R files; $F_{ij}, 1 \leq i \leq M, 1 \leq j \leq R.$ The key observation is that all Map instances use the same hash function; thus, all output records with the same hash value are stored in the same output file.

The second phase of a MR program executes R instances of the Reduce program, where R is typically the number of nodes. The input for each Reduce instance R_j consists of the files $F_{ij}, 1 \leq i \leq M.$ These files are transferred over the network from the Map nodes' local disks. Note that again all output records from the Map phase with the same hash value are consumed by the same Reduce instance, regardless of whichMap instance produced the data. Each Reduce processes or combines the records assigned to it in some way, and then writes records to an output file (in the distributed file system), which forms part of the computation's final output.

2.2 Parallel DBMSs

Database systems capable of running on clusters of shared nothing nodes have existed since the late 1980s. These systems all support standard relational tables and SQL, and thus the fact that the data is stored on multiple machines is transparent to the end-user.

Many of these systems build on the pioneering research from the Gamma [10] and Grace [11] parallel DBMS projects. The two key aspects that enable parallel execution are that (1) most (or even all) tables are partitioned over the nodes in a cluster and that (2) the system uses an optimizer that translates SQL commands into a query plan whose execution is divided amongst multiple nodes. Because programmers only need to specify their goal in a high level language, they are not burdened by the underlying storage details, such as indexing options and join strategies.

Consider a SQL command to filter the records in a table T_1 based on a predicate, along with a join to a second table T_2 with an aggregate computed on the result of the join. A basic sketch of how this command is processed in a parallel DBMS consists of three phases.

Since the database will have already stored T_1 on some collection of the nodes partitioned on some attribute, the filter sub-query is first performed in parallel at these sites similar to the filtering performed in a Map function. Following this step, one of two common parallel join algorithms are employed based on the size of data tables.

3. ARCHITECTURAL ELEMENTS

In this section, we consider aspects of the two system architectures that are necessary for processing large amounts of data in a distributed environment. One theme in our discussion is that the nature of the MR model is well suited for development environments with a small number of programmers and a limited application domain. This lack of constraints, however, may not be appropriate for longer-term and larger-sized projects.

3.1 Schema Support

Parallel DBMSs require data to fit into the relational paradigm of rows and columns. In contrast, the MR model does not require that data files adhere to a schema defined using the relational data model. That is, the MR programmer is free to structure their data in any manner or even to have no structure at all.

One might think that the absence of a rigid schema automatically makes MR the preferable option. For example, SQL is often criticized for its requirement that the programmer must specify the “shape” of the data in a data definition facility. On the other hand, the MR programmer must often write a custom parser in order to derive the appropriate semantics for their input records, which is at least an equivalent amount of work. But there are also other potential problems with not using a schema for large data sets.

Whatever structure exists in MR input files must be built into the Map and Reduce programs. Existing MR implementations provide built-in functionality to handle simple key/value pair formats, but the programmer must explicitly write support for more complex data structures, such as compound keys. This is possibly an acceptable approach if a MR data set is not accessed by multiple applications. If such data sharing exists, however, a second programmer must decipher the code written by the first programmer to decide how to process the input file. A better approach, followed by all SQL DBMSs, is to separate the schema from the application and store it in a set of system catalogs that can be queried.

But even if the schema is separated from the application and made available to multiple MR programs through a description facility, the developers must also agree on a single schema. This obviously requires some commitment to a data model or models, and the input files must obey this commitment as it is cumbersome to modify data attributes once the files are created.

3.2 Indexing

All modern DBMSs use hash or B-tree indexes to accelerate access to data. If one is looking for a subset of records (e.g., employees with a salary greater than $100,000), then using a proper index reduces the scope of the search dramatically. Most database systems also support multiple indexes per table. Thus, the query optimizer can decide which index to use for each query or whether to simply perform a brute-force sequential search.

3.3 Programming Model

During the 1970s, the database research community engaged in a contentious debate between the relational advocates and the Codasyl advocates [18]. The salient issue of this discussion was whether a program to access data in a DBMS should be written either by:

1. Stating what you want – rather than presenting an algorithm for how to get it (Relational)

2. Presenting an algorithm for data access (Codasyl)

4. DISCUSSION

We now discuss broader issues about the benchmark results and comment on particular aspects of each system that the raw numbers may not convey. In the benchmark above, both DBMS-X and Vertica execute most of the tasks much faster than Hadoop at all scaling levels. The next subsections describe, in greater detail than the previous section, the reasons for this dramatic performance difference.

4.1 System-level Aspects

In this section, we describe how architectural decisions made at the system-level affect the relative performance of the two classes of data analysis systems. Since installation and configuration parameters can have a significant difference in the ultimate performance of the system, we begin with a discussion of the relative ease with which these parameters are set. Afterwards, we discuss some lower level implementation details. While some of these details affect performance in fundamental ways (e.g., the fact that MR does not transform data on loading precludes various I/O optimizations and necessitates runtime parsing which increases CPU costs), others are more implementation specific (e.g., the high start-up cost of MR).

4.1.1 System Installation, Configuration, and Tuning

We were able to get Hadoop installed and running jobs with little effort. Installing the system only requires setting up data directories on each node and deploying the system library and configuration files. Configuring the system for optimal performance was done through trial and error. We found that certain parameters, such as the size of the sort buffers or the number of replicas, had no affect on execution performance, whereas other parameters, such as using larger block sizes, improved performance significantly.

The DBMS-X installation process was relatively straightforward. A GUI leads the user through the initial steps on one of the cluster nodes, and then prepares a file that can be fed to an installer utility in parallel on the other nodes to complete the installation. Despite this simple process, we found that DBMS-X was complicated to configure in order to start running queries. Initially, we were frustrated by the failure of anything but the most basic of operations. We eventually discovered each node's kernel was configured to limit the total amount of allocated virtual address space. When this limit was hit, new processes could not be created and DBMS-X operations would fail. We mention this even though it was our own administrative error, as we were surprised that DBMS-X's extensive system probing and self-adjusting configuration was not able to detect this limitation.

4.1.2 Task Start-up

We found that our MR programs took some time before all nodes were running at full capacity. On a cluster of 100 nodes, it takes 10 seconds from the moment that a job is submitted to the JobTracker before the first Map task begins to execute and 25 seconds until all the nodes in the cluster are executing the job. This coincides with the results in [8], where the data processing rate does not reach its peak for nearly 60 seconds on a cluster of 1800 nodes. The “cold start” nature is symptomatic to Hadoop’s (and apparently Google’s) implementation and not inherent to the actual MR model itself. For example, we also found that prior versions of Hadoop would create a new JVM process for each Map and Reduce instance on a node, which we found increased the overhead of running jobs on large data sets; enabling the JVM reuse feature in the latest version of Hadoop improved our results for MR by 10–15%.

4.1.3 Compression

Almost every parallel DBMS (including DBMS-X and Vertica) allows for optional compression of stored data. It is not uncommon for compression to result in a factor of 6–10 space savings. Vertica’s internal data representation is highly optimized for data compression and has an execution engine that operates directly on compressed data (i.e., it avoids decompressing the data during processing whenever possible). In general, since analysis tasks on large data sets are often I/O bound, trading CPU cycles (needed to decompress input data) for I/O bandwidth (compressed data means that there is less data to read) is a good strategy and translates to faster execution. In situations where the executor can operate directly on compressed data, there is often no trade-off at all and compression is an obvious win.

Hadoop and its underlying distributed filesystem support both block-level and record-level compression on input data. We found, however, that neither technique improved Hadoop’s performance and in some cases actually slowed execution. It also required more effort on our part to either change code or prepare the input data. It should also be noted that compression was also not used in the original MR benchmark [8].

4.1.4 Loading and Data Layout

Parallel DBMSs have the opportunity to reorganize the input data file at load time. This allows for certain optimizations, such as storing each attribute of a table separately (as done in column-stores such as Vertica). For read-only queries that only touch a subset of the attributes of a table, this optimization can improve performance by allowing the attributes that are not accessed by a particular query to be left on disk and never read. Similar to the compression optimization described above, this saves critical I/O bandwidth. MR systems by default do not transform the data when it is loaded into their distributed file system, and thus are unable to change the layout

of input data, which precludes this class of optimization opportunities.

4.1.5 Execution Strategies

As noted earlier, the query planner in parallel DBMSs are careful to transfer data between nodes only if it is absolutely necessary. This allows the systems to optimize the join algorithm depending on the characteristics of the data and perform push-oriented messaging without writing intermediate data sets. Over time, MR advocates should study the techniques used in parallel DBMSs and incorporate the concepts that are germane to their model. In doing so, we believe that again the performance of MR frameworks will improve dramatically.

4.1.6 Failure Model

As discussed previously, while not providing support for transactions, MR is able to recover from faults in the middle of query execution in a way that most parallel database systems cannot. Since parallel DBMSs will be deployed on larger clusters over time, the probability of mid-query hardware failures will increase. Thus, for long running queries, it may be important to implement such a fault tolerance model. While improving the fault-tolerance of DBMSs is clearly a good idea, we are wary of devoting huge computational clusters and “brute force” approaches to computation when sophisticated software would could do the same processing with far less hardware and consume far less energy, or in less time, thereby obviating the need for a sophisticated fault tolerance model. A multithousand-node cluster of the sort Google, Microsoft, and Yahoo! run uses huge amounts of energy, and as our results show, for many data processing tasks a parallel DBMS can often achieve the same performance using far fewer nodes. As such, the desirable approach is to use high-performance algorithms with modest parallelism rather than brute force approaches on much larger clusters.

4.2 User-level Aspects

A data processing system’s performance is irrelevant to a user or an organization if the system is not usable. In this section, we discuss aspects of each system that we encountered from a userlevel perspective while conducting the benchmark study that may promote or inhibit application development and adoption.

4.2.1 Ease of Use

Once the system is on-line and the data has been loaded, the programmer then begins to write the query or the code needed to perform their task. Like other kinds of programming, this is often an

iterative process: the programmer writes a little bit of code, tests it, and then writes some more. The programmer can easily determine whether his/her code is syntactically correct in both types of systems: the MR framework can check whether the user’s code compiles and the SQL engines can determine whether the queries parse correctly. Both systems also provide runtime support to assist users in debugging their programs.

It is also worth considering the way in which the programmer writes the query. MR programs in Hadoop are primarily written in Java (though other language bindings exist). Most programmers are more familiar with object-oriented, imperative programming than with other language technologies, such as SQL. That said, SQL is taught in many undergraduate programs and is fairly portable – we were able to share the SQL commands between DBMS-X and Vertica with only minor modifications.

4.2.2 Additional Tools

Hadoop comes with a rudimentary web interface that allows the user to browse the contents of the distributed filesystemand monitor the execution of jobs. Any additional tools would most likely at this time have to be developed in house.

SQL databases, on the other hand, have tons of existing tools and applications for reporting and data analysis. Entire software industries have developed around providing DBMS users with third-party extensions. The types of software that many of these tools include (1) data visualization, (2) business intelligence, (3) data mining, (4) data replication, and (5) automatic database design. Because MR technologies are still nascent, the market for such software for MR is limited; however, as the user base grows, many of the existing SQL-based tools will likely support MR systems.

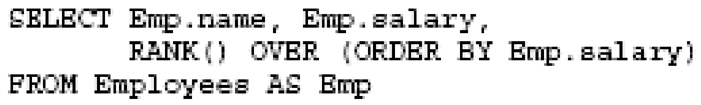
5. CONCLUSION

There are a number of interesting conclusions that can be drawn from the results presented in this paper. First, at the scale of the experiments we conducted, both parallel database systems displayed a significant performance advantage over Hadoop MR in executing a variety of data intensive analysis benchmarks. Averaged across all five tasks at 100 nodes, DBMS-X was 3.2 times faster than MR and Vertica was 2.3 times faster than DBMS-X.While we cannot verify this claim, we believe that the systems would have the same relative performance on 1,000 nodes (the largest Teradata configuration is less than 100 nodes managing over four petabytes of data). The dual of these numbers is that a parallel database system that provides the same response time with far fewer processors will certainly uses far less energy; theMapReduce model on multi-thousand node clusters is a brute force solution that wastes vast amounts of energy. While it is rumored that the Google version of MR is faster than the Hadoop version, we did not have access to this code and hence could not test it. We are doubtful again, however, that there would be a substantial difference in the performance of the two versions as MR is always forced to start a query with a scan of the entire input file.

This performance advantage that the two database systems share is the result of a number of technologies developed over the past 25 years, including (1) B-tree indices to speed the execution of selection operations, (2) novel storage mechanisms (e.g., columnorientation), (3) aggressive compression techniques with ability to operate directly on compressed data, and (4) sophisticated parallel algorithms for querying large amounts of relational data. In the case of a column-store database like Vertica, only those columns that are needed to execute a query are actually read from disk. Furthermore, the column-wise storage of data results in better compression factors (approximately a factor of 2.0 for Vertica, versus a factor of 1.8 for DBMS-X and 1.25 for Hadoop); this also further reduces the amount of disk I/O that is performed to execute a query.

Extensibility was another area where we found the database systems we tested lacking. Extending a DBMS with user-defined types and functions is an idea that is now 25 years old [16]. Neither of the parallel systems we tested did a good job on the UDF aggregation tasks, forcing us to find workarounds when we encountered limitations (e.g., Vertica) and bugs (e.g., DBMS-X).

While all DB systems are tolerant of a wide variety of software failures, there is no question that MR does a superior job of minimizing the amount of work that is lost when a hardware failure occurs. This capability, however, comes with a potentially large performance penalty, due to the cost of materializing the intermediate files between the map and reduce phases. Left unanswered is how significant this performance penalty is. Unfortunately, to investigate this question properly requires implementing both the materialization and no-materialization strategies in a common framework, which is an effort beyond the scope of this paper. Despite a clear advantage in this domain, it is not completely clear how significant a factor Hadoop's ability to tolerate failures during execution really is in practice. In addition, if a MR system needs 1,000 nodes to match the performance of a 100 node parallel database system, it is ten times more likely that a node will fail while a query is executing.



```
SELECT Emp.name, Emp.salary,
       RANK() OVER (ORDER BY Emp.salary)
FROM Employees AS Emp
```

Computing this in parallel requires producing a total order of all employees followed by a second phase in which each node adjusts the rank values of its records with the counts of the number of records on each node to its “left” (i.e., those nodes with salary values that are strictly smaller). Although aMR program could perform this sort in parallel, it is not easy to fit this query into the MR paradigm of group by aggregation. RANK is just one of the many powerful analytic functions provided by modern parallel database systems. For example, both Teradata and Oracle support a rich set of functions, such as functions over windows of ordered records.

Two architectural differences are likely to remain in the long run. MR makes a commitment to a “schema later” or even “schema never” paradigm. But this lack of a schema has a number of important consequences. Foremost, it means that parsing records at run time is inevitable, in contrast to DBMSs, which perform parsing at load time. This difference makes compression less valuable in MR and causes a portion of the performance difference between the two classes of systems. Without a schema, each user must write a custom parser, complicating sharing data among multiple applications. Second, a schema is needed for maintaining information that is critical for optimizing declarative queries, including what indices exist, how tables are partitioned, table cardinalities, and histograms that capture the distribution of values within a column.

In our opinion there is a lot to learn from both kinds of systems. Most importantly is that higher level interfaces, such as Pig [15], Hive [2], are being put on top of the MR foundation, and a number of tools similar in spirit but more expressive than MR are being developed, such as Dryad [13] and Scope [5]. This will make complex tasks easier to code in MR-style systems and remove one of the big advantages of SQL engines, namely that they take much less code on the tasks in our benchmark. For parallel databases, we believe that both commercial and open-source systems will dramatically improve the parallelization of user-defined functions. Hence, the APIs of the two classes of systems are clearly moving toward each other. Early evidence of this is seen in the solutions for integrating SQL with MR offered by Greenplum and Asterdata.

6. REFERENCES

[1] Hadoop. http://hadoop.apache.org/.

[2] Hive. http://hadoop.apache.org/hive/.

[3] Vertica. http://www.vertica.com/.

[4] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical report, 1998.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[6] Cisco Systems. *Cisco Catalyst 3750-E Series Switches Data Sheet*, June 2008.

[7] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *Under Submission*, March 2009.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI ’04*, pages 10–10, 2004.

[9] D. J. DeWitt and R. H. Gerber. Multiprocessor Hash-based Join Algorithms. In *VLDB ’85*, pages 151–164, 1985.

[10] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB '86*, pages 228–237, 1986.

[11] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine. In *VLDB '86*, pages 209–219, 1986.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys '07*, pages 59–72, 2007.

[14] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06*, pages 706–706, 2006.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD '08*, pages 1099–1110, 2008.

[16] J. Ong, D. Fogg, and M. Stonebraker. Implementation of data abstraction in the relational database system ingres. *SIGMOD Rec.*, 14(1):1–14, 1983.

[17] D. A. Patterson. Technical Perspective: The Data Center is the Computer. *Commun. ACM*, 51(1):105–105, 2008.

[18] R. Rustin, editor. *ACM-SIGMOD Workshop on Data Description, Access and Control*, May 1974.

[19] M. Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[20] M. Stonebraker and J. Hellerstein. What Goes Around Comes Around. In *Readings in Database Systems*, pages 2–41. The MIT Press, 4th edition, 2005.

[21] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.