HadoopDB: An Architectural Hybrid of MapReduce and DBMS technologies

[1]Venkata Naga Akash Ungarala, [2]Atif Farid Mohammed, [3]Ravali Vemuganti, [4]Snigdha Sree Paladugu, [5]Sneha Suresh
University of North Carolina, Charlotte.
[1]vungaral@uncc.edu, [2]amoham19@uncc.edu, [3]rvemugan@uncc.edu, [4]spaladu4@uncc.edu, [5]ssuresh7@uncc.edu

ABSTRACT

The production environment for analytical data management applications is rapidly changing. Many enterprises are shifting away from deploying their analytical databases on high-end proprietary machines, and moving towards cheaper, lower-end, commodity hardware, typically arranged in a shared-nothing MPP architecture, often in a virtualized environment inside public or private "clouds". Programs written in MapReduce functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time 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 the required inter-machine communication. At the same time, the amount of data that needs to be analyzed is exploding, requiring hundreds to thousands of machines to work in parallel to perform the analysis. There tend to be two schools of thought regarding what technology to use for data analysis in such an environment. The implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines In this paper, we explore the feasibility of building a hybrid system that takes the best features from both technologies; the prototype we built approaches parallel databases in performance and efficiency, yet still yields the scalability, fault tolerance, and flexibility of Map-Reduce-based systems.

1. INTRODUCTION

The analytical database market currently consists of $3.98 billion of the $14.6 billion database software market (27%) and is growing at a rate of 10.3% annually. As business "bestpractices" trend increasingly towards basing decisions off data and hard facts rather than instinct and theory, the corporate thirst for systems that can manage, process, and granularly analyze data is becoming insatiable. Venture capitalists are very much aware of this trend, and have funded no fewer than a dozen new companies in recent years that build specialized analytical data management software (e.g., Netezza, Vertica, DATAllegro, Greenplum, Aster Data, Infobright, Kickfire, Dataupia, ParAccel, and Exasol), and continue to fund them, even in pressing economic. At the same time, the amount of data that needs to be stored and processed by analytical database systems is exploding. This is partly due to the increased automation with which data can be produced (more business processes are becoming digitized), the proliferation of sensors and data-producing devices, Web-scale interactions with customers, and government compliance demands along with strategic corporate initiatives requiring more historical data to be kept online for analysis. It is no longer uncommon to hear of companies claiming to load more than a terabyte of structured data per day into their analytical database system and claiming data warehouses of size more than a petabyte. Given the exploding data problem, all but three of the above mentioned analytical database start-ups deploy their DBMS on a shared-nothing architecture (a collection of independent, possibly virtual, machines, each with local disk and local main memory, connected together on a high-speed network). This architecture is widely believed to scale the best, especially if one takes hardware cost into account. Furthermore, data analysis workloads tend to consist of many large scan operations, multidimensional aggregations, and star schema joins, all of which are fairly easy to parallelize across nodes in a shared-nothing network. Analytical DBMS vendor leader, Teradata, uses a shared-nothing architecture. Oracle and Microsoft have recently announced shared-nothing analytical DBMS products in their Exadata1 and Madison projects, respectively. For the purposes of this paper, we will call analytical DBMS systems that deploy on a shared-nothing architecture parallel databases. MapReduce is a new framework specifically designed for processing huge datasets on distributed sources. Apache's Hadoop is an implementation of MapReduce. Currently Hadoop has been applied successfully for file based datasets. The execution engine that is developed on top of Hadoop applies Map and Reduce techniques to break down the parsing and execution stages for parallel and distributed processing. Moreover MapReduce will provide the fault tolerance, scalability and reliability because its library is designed to help process very large amount of data using hundred and thousands of machine, which must tolerate the machine failure. This paper proposes to utilize the parallel and distributed processing capability of Hadoop MapReduce for handling heterogeneous query execution on large datasets. So Virtual Database Engine built on top of this will result in effective high performance distributed data integration. Parallel databases have been proven to scale really well into the tens of nodes (near linear scalability is not uncommon). However, there are very few known parallel databases deployments consisting of more than one hundred nodes, and to the best of our knowledge, there exists no published deployment of a parallel database with nodes numbering into the thousands. There are a variety of reasons why parallel databases generally do not scale well into the hundreds of nodes. First, failures become increasingly common as one adds more nodes to a system, yet parallel databases tend to be designed with the assumption that failures are a rare event. Second, parallel databases generally assume a homogeneous array of machines, yet it is nearly impossible to achieve pure homogeneity at scale. Third, until recently, there have only been a handful of applications that required deployment on more than a few dozen nodes for reasonable performance, so parallel databases have not been tested at larger scales, and unforeseen engineering hurdles await. As the data that needs to be analyzed continues to grow, the number of applications that require more than one hundred nodes is beginning to multiply. Some argue that MapReduce-based systems are best suited for performing analysis at this scale since they were designed from the beginning to scale to thousands of nodes in a shared-nothing architecture, and have had proven success in Google's internal operations and on the TeraSort benchmark. Despite being originally designed for a largely different application (unstructured text data processing), MapReduce can nonetheless be used to process structured data, and can do so at tremendous scale. For example, Hadoop is being used to manage Facebook's 2.5 petabyte data warehouse Unfortunately, as pointed out by DeWitt and Stonebraker, MapReduce lacks many of the features that have proven invaluable for structured data analysis workloads (largely due to the fact that MapReduce was not originally designed to perform structured data analysis), and its immediate gratification paradigm precludes some of the long term benefits of first modeling and loading data before processing. These shortcomings can cause an order of magnitude slower performance than parallel databases. Ideally, the scalability advantages of MapReduce could be combined with the performance and efficiency advantages of parallel databases to achieve a hybrid system that is well suited for the analytical DBMS market and can handle the future demands of data intensive applications. In this paper, we describe our implementation of and experience with HadoopDB, whose goal is to serve as exactly such a hybrid system. The basic idea behind HadoopDB is to use MapReduce as the communication layer above multiple nodes running single-node DBMS instances. As MapReduce clusters get popular, their scheduling becomes increasingly important. In a MapReduce cluster, data are distributed to individual nodes and stored in their disks. To execute a map task on a node, we need to first have its input data available on that node. Since transferring data from one node to another takes time and delays task execution, an efficient MapReduce scheduler must avoid unnecessary data transmission. Queries are expressed in SQL, translated into MapReduce by extending existing tools, and as much work as possible is pushed into the higher performing single node databases. One of the advantages of MapReduce relative to parallel databases not mentioned above is cost. There exists an open source version of MapReduce (Hadoop) that can be obtained and used without cost. Yet all of the parallel databases mentioned above have a nontrivial cost, often coming with seven figure price tags for large installations. Since it is our goal to combine all of the advantages of both data analysis approaches in our hybrid system, we decided to build our prototype completely out of open source components in order to achieve the cost advantage as well. Hence, we use PostgreSQL as the database layer and Hadoop as the communication layer, Hive as the translation layer, and all code we add we release as open source. One side effect of such a design is a shared-nothing version of PostgreSQL. Existing MapReduce algorithms provide some mechanisms to improve the data locality. For instance, to assign map tasks to a node, the Hadoop default FIFO (firstin-first-out) scheduler always picks the first job in the waiting queue and schedules its local map tasks (i.e., tasks with input data stored in the node). If the job does not have any map task local to the node, only one of its non-local map tasks will be assigned to the node at a time. However, due to FIFO scheduler's inherent deficiencies (like following the strict FIFO job order for assigning tasks), this mechanism can only slightly improve the data locality. We are optimistic that our approach has the potential to help transform any single-node DBMS into a shared-nothing parallel database. Given our focus on cheap, large scale data analysis, our target platform is virtualized public or private "cloud computing" deployments, such as Amazon's Elastic Compute Cloud (EC2) or VMware's private VDC-OS offering. Such deployments significantly reduce up-front capital costs, in addition to lowering operational, facilities, and hardware costs (through maximizing current hardware utilization). Public cloud offerings such as EC2 also yield tremendous economies of scale, and pass on some of these savings to the customer. All experiments we run in this paper are on Amazon's EC2 cloud offering; however our techniques are applicable to non-virtualized cluster computing grid deployments as well. In summary, the primary contributions of our work include: We extend previous work that showed the superior performance of parallel databases relative to Hadoop. While this previous work focused only on performance in an ideal setting, we add fault tolerance and heterogeneous node experiments to demonstrate some of the issues with scaling parallel databases. We describe the design of a hybrid system that is designed to yield the advantages of both parallel databases and MapReduce. This system can also be used to allow single-node databases to run in a shared-nothing environment. We evaluate this hybrid system on a previously published benchmark to determine how close it comes to parallel DBMSs in performance and Hadoop in scalability.

2. BACKGROUND AND SHORTFALLS OF AVAILABLE APPROACHES

In this section, we give an overview of the parallel database and Map Reduce approaches to performing data analysis.

2.1 Parallel DBMSs

Parallel database systems support standard relational tables and SQL, and implement many of the performance enhancing techniques developed by the research community over the past few decades, including indexing, compression (and direct operation on compressed data), materialized views, result caching, and I/O sharing. Parallel processing divides a large task into many smaller tasks, and executes the smaller tasks concurrently on several nodes. As a result, the larger task completes more quickly.

Some tasks can be effectively divided, and thus are good candidates for parallel processing. Other tasks, however, do not lend themselves to this approach. For example, in a bank with only one teller, all customers must form a single queue to be served. With two tellers, the task can be effectively split so that customers form two queues and are served twice as fast-or they can form a single queue to provide fairness. This is an instance in which parallel processing is an effective solution. By contrast, if the bank manager must approve all loan requests, parallel processing will not necessarily speed up the flow of loans. No matter how many tellers are available to process loans, all the requests must form a single queue for bank manager approval. No amount of parallel processing can overcome this built-in bottleneck to the system.

Parallel databases best meet the “performance property” due to the performance push required to compete on the open market, and the ability to incorporate decade’s worth of performance tricks published in the database research community. Parallel databases can achieve especially high performance when administered by a highly skilled DBA who can carefully design, deploy, tune, and maintain the system, but recent advances in automating these tasks and bundling the software into appliance (pre-tuned and pre-configured) offerings have given many parallel databases high performance out of the box. Parallel databases also score well on the flexible query interface property. Although particular details of parallel database implementations vary, their historical assumptions that failures are rare events and “large” clusters mean dozens of nodes (instead of hundreds or thousands) have resulted in engineering decisions that make it difficult to achieve these properties. Furthermore, in some cases, there is a clear tradeoff between fault tolerance and performance, and parallel databases tend to choose the performance extreme of these tradeoffs. For example, frequent check-pointing of completed sub-tasks increase the fault tolerance of long-running read queries, yet this check-pointing reduces performance. In addition, pipelining intermediate results between query operators can improve performance, but can result in a large amount of work being lost upon a failure.

2.2 MapReduce

The key aspect of the MapReduce algorithm is that if every Map and Reduce is independent of all other ongoing Maps and Reduces, then the operation can be run in parallel on different keys and lists of data. On a large cluster of machines, you can go one step further, and run the Map operations on servers where the data lives. Rather than copy the data over the network to the program, you push out the program to the machines. The output list can then be saved to the distributed file system, and the reducers run to merge the results. Again, it may be possible to run these in parallel, each reducing different keys.

- A distributed file system spreads multiple copies of the data across different machines. If a machine with one copy of the data is busy or offline, another machine can be used.

- A job scheduler (in Hadoop, the JobTracker), keeps track of which MR jobs are executing, schedules individual Maps, Reduces or intermediate merging operations to specific machines, monitors the success and failures of these individual Tasks, and works to complete the entire batch job.

- The filesystem and Job scheduler can somehow be accessed by the people and programs that wish to read and write data, and to submit and monitor MR jobs.

Apache Hadoop is such a MapReduce engine. It provides its own distributed filesystem and runs [HadoopMapReduce] jobs on servers near the data stored on the filesystem -or any other supported filesystem, of which there is more than one.

MapReduce does not create a detailed query execution plan that specifies which nodes will run which tasks in advance; instead, this is determined at runtime. This allows MapReduce to adjust to node failures and slow nodes on the fly by assigning more tasks to faster nodes and reassigning tasks from failed nodes. MapReduce also checkpoints the output of each Map task to local disk in order to minimize the amount of work that has to be redone upon a failure.
Map and Reduce functions are just arbitrary computations written in a general-purpose language. Therefore, it is possible for each task to do anything on its input, just as long as its output follows the conventions defined by the model. In general, most MapReduce-based systems (such as Hadoop, which directly implements the systems-level details of the MapReduce paper) do not accept declarative SQL.

2.3 Work with Hewlett Packard

The work around of comparing Parallel db and mapreduce was done using HP’s patent data. The main motive is to find out the efficiency and time in the domain they performed. Wrangling of patent data and processing it was done using Hadoop Map reduce. The same was done using parallel database concept. Time and efficiency were noted down. We concluded that large data was handled easily using mapreduce and also the data which we had o proper structure. Mapreduce worked good with unstructured data and it divided and worked on it without much input from the user. There were occasional node failures which it didn’t consider and moved on working on others. The same was implemented using parallel db. This task required the data to be in a format in which SQL worked.

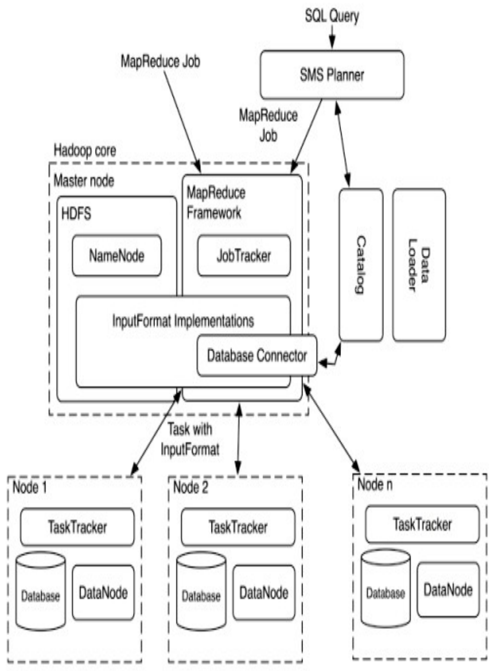
We had to take additional efforts in structuring the data and feeding it to the database systems. It was fast and efficient. There were no node failures which was expected.

3. HADOOPDB

In this section, we explain the design of HadoopDB.. In HadoopDB, the idea is to connect multiple single-node database systems using Hadoop as the network communication layer and task coordinator. The queries are parallelized across nodes using the MapReduce framework. HadoopDB accomplishes adaptation to non-critical failure and the capacity to work in heterogeneous situations by acquiring the planning and employment from Hadoop, yet it accomplishes the execution of parallel databases by doing most of the processing of queries in the database engine.

3.1 Hadoop Implementation Background

The important part of HadoopDB is the Hadoop framework. Hadoop consits of two layers: (i) a data storage layer or the Hadoop Distributed File System (HDFS) and (ii) a data processing layer or the MapReduce Framework. A **HDFS** cluster primarily consists of a NameNode that manages the file system metadata and DataNodes that store the actual data. Individual files are broken into blocks of a fixed size and distributed across multiple DataNodes in the cluster. The NameNode maintains metadata about the size and location of blocks along with their replicas. The MapReduce Framework follows a simple master-slave architecture. The master is a single JobTracker and the slaves are TaskTrackers. The JobTracker handles the runtime scheduling of MapReduce jobs and maintains information on each TaskTracker's load and available resources. The JobTracker assigns tasks to TaskTrackers based on locality and load balancing. It achieves



area by coordinating a TaskTracker to Map undertakings that procedure information nearby to it.
TaskTrackers updates regularly the JobTracker with their status through messages.
The InputFormat library represents the interface between the storage and processing layers. Its implementations generally parse text/binary files and also transform the data which can be processed by Map tasks.

3.2 HadoopDB's Components

HadoopDB extends the Hadoop framework by providing the following four components:

3.2.1 Database Connector

The Database Connector is the interface between database systems that rely on nodes in the cluster and TaskTrackers. It is a part of the InputFormat Implementations library. Each MapReduce job supplies the Connector with an SQL query and connection parameters such as: which JDBC driver to use, query fetch size and other query tuning parameters. The Connector connects to the database, executes the SQL query and returns results as key-value pairs. The Connector could theoretically connect to any JDBC-compliant database that resides in the cluster. However, different databases require different read query optimizations. We implemented connectors for MySQL and PostgreSQL. In the future we plan to integrate other databases including open-source column-store databases such as MonetDB and InfoBright. By extending Hadoop's InputFormat, we integrate seamlessly with Hadoop's MapReduce Framework. To the framework, the databases are data sources similar to data blocks in HDFS.

3.2.2 Catalog

The catalog maintains meta information about the databases such as, connection parameters that include database location, driver class and credentials and metadata such as data sets contained in the cluster and data partitioning properties. The current implementation of the HadoopDB catalog stores its metainformation as an XML file in HDFS. This file is accessed by the JobTracker and TaskTrackers to retrieve information necessary to schedule tasks and process data needed by a query.

3.2.3 Data Loader

The Data Loader is responsible for tasks such as, repartitioning data globally on a given partition key upon loading, breaking apart single node data into multiple smaller partitions or chunks and then finally, bulk-loading the single-node databases with the chunks. The Data Loader consists of two main components: Global Hasher and Local Hasher. The Global Hasher executes MapReduce job over Hadoop that reads in raw data files stored in HDFS and divides them into many parts of nodes in the cluster. The dividing job does not incur the sorting overhead of typical MapReduce jobs. The Local Hasher then replicates a partition from HDFS into the local file system of each node and then partitions the file into smaller sized chunks based on the maximum chunk size setting. The Global Hasher and the Local Hasher use different hashing functions to make sure the chunks used are of a uniform size.

3.2.4 SQL to MapReduce to SQL (SMS) Planner

HadoopDB provides a parallel database front-end to data analysts enabling them to process SQL queries. The SMS planner extends Hive [11]. SQL to MapReduce to SQL (SMS) Planner consists of a slightly modified Hive build and SMS specific classes. Each table is stored as a separate file in HDFS, Hive assumes no ordering of tables on nodes. Hence, operations that involve several tables generally require high processing to occur in the Reduce phase of a MapReduce job. This assumption does not completely hold in HadoopDB as some tables are ordered and if partitioned on the same attribute, the join operation can be pushed entirely into the database layer.

3.3 Summary

Hadoop is not replaced by HadoopDB. But both systems coexist and enables the analyst to select the relevant tools for any given dataset and task. In the next sections using the benchmarks, we show that using an efficient database storage layer reduces data processing time especially on tasks that require complex query processing over structured data such as joins. We also show that HadoopDB is advantageous in terms of fault-tolerance and the also has the ability to run on heterogeneous environments which generally comes with Hadoop-style systems. The combination of the two methods would give an output that would change the data processing and take it to next generation. With high efficiency and minimum or nil node failure and the ability to deal with all kind of data sets.

4. REFERENCES

1. MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat jeff@google.com, sanjay@google.com Google, Inc. http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
2. Hadoop. Web Page. hadoop.apache.org/core/ .
3. HadoopDB Project. Web page. db.cs.yale.edu/hadoopdb/hadoopdb.html .
4. Vertica. www.vertica.com/.
5. D. Abadi. What is the right way to measure scale? DBMS Musings Blog. dbmsmusings.blogspot.com/2009/06/what-is-right-way-to-measure-scale.html .
6. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In Proc. of SOSP, 2003.
7. 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. In Proc. of VLDB, 2008.
8. G. Czajkowski. Sorting 1pb with mapreduce. googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html .
9. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, 2004.