HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads

Azza Abouzeid[1], Kamil Bajda-Pawlikowski[1],
Daniel Abadi[1], Avi Silberschatz[1], Alexander Rasin[2]
[1]Yale University, [2]Brown University

{azza,kbajda,dna,avi}@cs.yale.edu; alexr@cs.brown.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". 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. Proponents of parallel databases argue that the strong emphasis on performance and efficiency of parallel databases makes them well-suited to perform such analysis. On the other hand, others argue that MapReduce-based systems are better suited due to their superior scalability, fault tolerance, and flexibility to handle unstructured data. 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 MapReduce-based systems.

1. INTRODUCTION

The analytical database market currently consists of $3.98 billion [25] of the $14.6 billion database software market [21] (27%) and is growing at a rate of 10.3% annually [25]. As business "best-practices" 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 times [18].

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 [19].

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 [17], 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 Exadata[1] 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**[2].

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

---

[1]To be precise, Exadata is only shared-nothing in the storage layer.
[2]This is slightly different than textbook definitions of parallel databases which sometimes include shared-memory and shared-disk architectures as well.

[8] 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 [7]. Despite being originally designed for a largely different application (unstructured text data processing), MapReduce (or one of its publicly available incarnations such as open source Hadoop [1]) 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 [20].

Unfortunately, as pointed out by DeWitt and Stonebraker [9], 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 [23].

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. 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 [2].

One side effect of such a design is a shared-nothing version of PostgreSQL. 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 [14], 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 [23] 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 experi-

ments 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. RELATED WORK

There has been some recent work on bringing together ideas from MapReduce and database systems; however, this work focuses mainly on language and interface issues. The Pig project at Yahoo [22], the SCOPE project at Microsoft [6], and the open source Hive project [11] aim to integrate declarative query constructs from the database community into MapReduce-like software to allow greater data independence, code reusability, and automatic query optimization. Greenplum and Aster Data have added the ability to write MapReduce functions (instead of, or in addition to, SQL) over data stored in their parallel database products [16].

Although these five projects are without question an important step in the hybrid direction, there remains a need for a hybrid solution at the systems level in addition to at the language and interface levels. This paper focuses on such a systems-level hybrid.

3. DESIRED PROPERTIES

In this section we describe the desired properties of a system designed for performing data analysis at the (soon to be more common) petabyte scale. In the following section, we discuss how parallel database systems and MapReduce-based systems do not meet some subset of these desired properties.

Performance. Performance is the primary characteristic that commercial database systems use to distinguish themselves from other solutions, with marketing literature often filled with claims that a particular solution is many times faster than the competition. A factor of ten can make a big difference in the amount, quality, and depth of analysis a system can do.

High performance systems can also sometimes result in cost savings. Upgrading to a faster software product can allow a corporation to delay a costly hardware upgrade, or avoid buying additional compute nodes as an application continues to scale. On public cloud computing platforms, pricing is structured in a way such that one pays only for what one uses, so the vendor price increases linearly with the requisite storage, network bandwidth, and compute power. Hence, if data analysis software product A requires an order of magnitude more compute units than data analysis software product B to perform the same task, then product A will cost (approximately) an order of magnitude more than B. Efficient software has a direct effect on the bottom line.

Fault Tolerance. Fault tolerance in the context of analytical data workloads is measured differently than fault tolerance in the context of transactional workloads. For transactional workloads, a fault tolerant DBMS can recover from a failure without losing any data or updates from recently committed transactions, and in the context of distributed databases, can successfully commit transactions and make progress on a workload even in the face of worker node failures. For read-only queries in analytical workloads, there are neither write transactions to commit, nor updates to lose upon node failure. Hence, a fault tolerant analytical DBMS is simply one that

does not have to restart a query if one of the nodes involved in query processing fails.

Given the proven operational benefits and resource consumption savings of using cheap, unreliable commodity hardware to build a shared-nothing cluster of machines, and the trend towards extremely low-end hardware in data centers [14], the probability of a node failure occurring during query processing is increasing rapidly. This problem only gets worse at scale: the larger the amount of data that needs to be accessed for analytical queries, the more nodes are required to participate in query processing. This further increases the probability of at least one node failing during query execution. Google, for example, reports an average of 1.2 failures per analysis job [8]. If a query must restart each time a node fails, then long, complex queries are difficult to complete.

Ability to run in a heterogeneous environment. As described above, there is a strong trend towards increasing the number of nodes that participate in query execution. It is nearly impossible to get homogeneous performance across hundreds or thousands of compute nodes, even if each node runs on identical hardware or on an identical virtual machine. Part failures that do not cause complete node failure, but result in degraded hardware performance become more common at scale. Individual node disk fragmentation and software configuration errors can also cause degraded performance on some nodes. Concurrent queries (or, in some cases, concurrent processes) further reduce the homogeneity of cluster performance. On virtualized machines, concurrent activities performed by different virtual machines located on the same physical machine can cause 2-4% variation in performance [5].

If the amount of work needed to execute a query is equally divided among the nodes in a shared-nothing cluster, then there is a danger that the time to complete the query will be approximately equal to time for the slowest compute node to complete its assigned task. A node with degraded performance would thus have a disproportionate effect on total query time. A system designed to run in a heterogeneous environment must take appropriate measures to prevent this from occurring.

Flexible query interface. There are a variety of customer-facing business intelligence tools that work with database software and aid in the visualization, query generation, result dash-boarding, and advanced data analysis. These tools are an important part of the analytical data management picture since business analysts are often not technically advanced and do not feel comfortable interfacing with the database software directly. Business Intelligence tools typically connect to databases using ODBC or JDBC, so databases that want to work with these tools must accept SQL queries through these interfaces.

Ideally, the data analysis system should also have a robust mechanism for allowing the user to write user defined functions (UDFs) and queries that utilize UDFs should automatically be parallelized across the processing nodes in the shared-nothing cluster. Thus, both SQL and non-SQL interface languages are desirable.

4. BACKGROUND AND SHORTFALLS OF AVAILABLE APPROACHES

In this section, we give an overview of the parallel database and MapReduce approaches to performing data analysis, and list the properties described in Section 3 that each approach meets.

4.1 Parallel DBMSs

Parallel database systems stem from research performed in the late 1980s and most current systems are designed similarly to the early Gamma [10] and Grace [12] parallel DBMS research projects.

These systems all 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. Most (or even all) tables are partitioned over multiple nodes in a shared-nothing cluster; however, the mechanism by which data is partitioned is transparent to the end-user. Parallel databases use an optimizer tailored for distributed workloads that turn SQL commands into a query plan whose execution is divided equally among multiple nodes.

Of the desired properties of large scale data analysis workloads described in Section 3, parallel databases best meet the "performance property" due to the performance push required to compete on the open market, and the ability to incorporate decades 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. Implementation of SQL and ODBC is generally a given, and many parallel databases allow UDFs (although the ability for the query planner and optimizer to parallelize UDFs well over a shared-nothing cluster varies across different implementations).

However, parallel databases generally do not score well on the fault tolerance and ability to operate in a heterogeneous environment properties. 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.

4.2 MapReduce

MapReduce was introduced by Dean et. al. in 2004 [8]. Understanding the complete details of how MapReduce works is not a necessary prerequisite for understanding this paper. In short, MapReduce processes data distributed (and replicated) across many nodes in a shared-nothing cluster via three basic operations. First, a set of Map tasks are processed in parallel by each node in the cluster without communicating with other nodes. Next, data is repartitioned across all nodes of the cluster. Finally, a set of Reduce tasks are executed in parallel by each node on the partition it receives. This can be followed by an arbitrary number of additional Map-repartition-Reduce cycles as necessary. 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.

Of the desired properties of large scale data analysis workloads,

MapReduce best meets the fault tolerance and ability to operate in heterogeneous environment properties. It achieves fault tolerance by detecting and reassigning Map tasks of failed nodes to other nodes in the cluster (preferably nodes with replicas of the input Map data). It achieves the ability to operate in a heterogeneous environment via redundant task execution. Tasks that are taking a long time to complete on slow nodes get redundantly executed on other nodes that have completed their assigned tasks. The time to complete the task becomes equal to the time for the fastest node to complete the redundantly executed task. By breaking tasks into small, granular tasks, the effect of faults and "straggler" nodes can be minimized.

MapReduce has a flexible query interface; 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. However, there are some exceptions (such as Hive).

As shown in previous work, the biggest issue with MapReduce is performance [23]. By not requiring the user to first model and load data before processing, many of the performance enhancing tools listed above that are used by database systems are not possible. Traditional business data analytical processing, that have standard reports and many repeated queries, is particularly, poorly suited for the one-time query processing model of MapReduce.

Ideally, the fault tolerance and ability to operate in heterogeneous environment properties of MapReduce could be combined with the performance of parallel databases systems. In the following sections, we will describe our attempt to build such a hybrid system.

5. HADOOPDB

In this section, we describe the design of HadoopDB. The goal of this design is to achieve all of the properties described in Section 3.

The basic idea behind HadoopDB is to connect multiple single-node database systems using Hadoop as the task coordinator and network communication layer. Queries are parallelized across nodes using the MapReduce framework; however, as much of the single node query work as possible is pushed inside of the corresponding node databases. HadoopDB achieves fault tolerance and the ability to operate in heterogeneous environments by inheriting the scheduling and job tracking implementation from Hadoop, yet it achieves the performance of parallel databases by doing much of the query processing inside of the database engine.

5.1 Hadoop Implementation Background

At the heart 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.

HDFS is a block-structured file system managed by a central NameNode. 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 and their replicas.

The MapReduce Framework follows a simple master-slave architecture. The master is a single JobTracker and the slaves or worker nodes are TaskTrackers. The JobTracker handles the run-time scheduling of MapReduce jobs and maintains information on each TaskTracker's load and available resources. Each job is broken down into Map tasks based on the number of data blocks that require processing, and Reduce tasks. The JobTracker assigns tasks to TaskTrackers based on locality and load balancing. It achieves
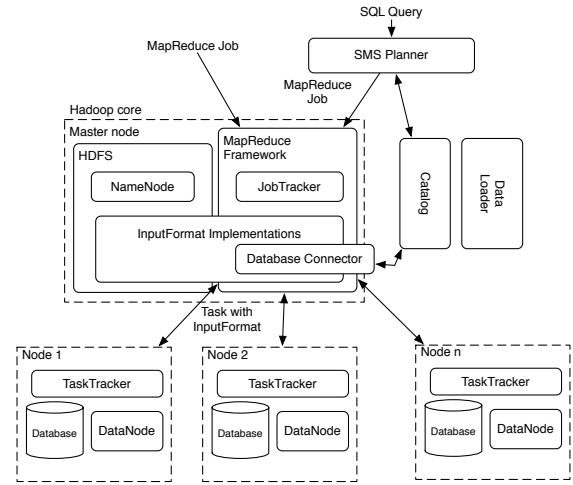

Figure 1: The Architecture of HadoopDB

locality by matching a TaskTracker to Map tasks that process data local to it. It load-balances by ensuring all available TaskTrackers are assigned tasks. TaskTrackers regularly update the JobTracker with their status through heartbeat messages.

The InputFormat library represents the interface between the storage and processing layers. InputFormat implementations parse text/binary files (or connect to arbitrary data sources) and transform the data into key-value pairs that Map tasks can process. Hadoop provides several InputFormat implementations including one that allows a single JDBC-compliant database to be accessed by all tasks in one job in a given cluster.

5.2 HadoopDB's Components

HadoopDB extends the Hadoop framework (see Fig. 1) by providing the following four components:

5.2.1 Database Connector

The Database Connector is the interface between independent database systems residing on nodes in the cluster and TaskTrackers. It extends Hadoop's InputFormat class and is part of the Input-Format 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.

5.2.2 Catalog

The catalog maintains metainformation about the databases. This includes the following: (i) connection parameters such as database location, driver class and credentials, (ii) metadata such as data sets contained in the cluster, replica locations, 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. In the future, we plan to deploy the catalog as a separate service that would work in a way similar to Hadoop's NameNode.

5.2.3 Data Loader

The Data Loader is responsible for (i) globally repartitioning data on a given partition key upon loading, (ii) breaking apart single node data into multiple smaller partitions or *chunks* and (iii) 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 a custom-made MapReduce job over Hadoop that reads in raw data files stored in HDFS and repartitions them into as many parts as the number of nodes in the cluster. The repartitioning job does not incur the sorting overhead of typical MapReduce jobs.

The Local Hasher then copies a partition from HDFS into the local file system of each node and secondarily partitions the file into smaller sized chunks based on the maximum chunk size setting.

The hashing functions used by both the Global Hasher and Local Hasher differ to ensure chunks are of a uniform size. They also differ from Hadoop's default hash-partitioning function to ensure better load balancing when executing MapReduce jobs over the data.

5.2.4 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]. Hive transforms HiveQL, a variant of SQL, into MapReduce jobs that connect to tables stored as files in HDFS. The MapReduce jobs consist of DAGs of relational operators (such as filter, select (project), join, aggregation) that operate as iterators: each operator forwards a data tuple to the next operator after processing it. Since each table is stored as a separate file in HDFS, Hive assumes no collocation of tables on nodes. Therefore, operations that involve multiple tables usually require most of the processing to occur in the Reduce phase of a MapReduce job. This assumption does not completely hold in HadoopDB as some tables are collocated and if partitioned on the same attribute, the join operation can be pushed entirely into the database layer.

To understand how we extended Hive for SMS as well as the differences between Hive and SMS, we first describe how Hive creates an executable MapReduce job for a simple GroupBy-Aggregation query. Then, we describe how we modify the execution plan for HadoopDB by pushing most of the query processing logic into the database layer.

Consider the following query:

```
SELECT YEAR(saleDate), SUM(revenue)
FROM sales GROUP BY YEAR(saleDate);
```

Hive processes the above SQL query in a series of phases:

(1) The parser transforms the query into an Abstract Syntax Tree.

(2) The Semantic Analyzer connects to Hive's internal catalog, the *MetaStore*, to retrieve the schema of the sales table. It also populates different data structures with meta information such as the Deserializer and InputFormat classes required to scan the table and extract the necessary fields.

(3) The logical plan generator then creates a DAG of relational operators, the query plan.

(4) The optimizer restructures the query plan to create a more optimized plan. For example, it pushes filter operators closer to the table scan operators. A key function of the optimizer is to break up the plan into Map or Reduce phases. In particular, it adds a Repartition operator, also known as a Reduce Sink operator, before Join or GroupBy operators. These operators mark the Map and Reduce
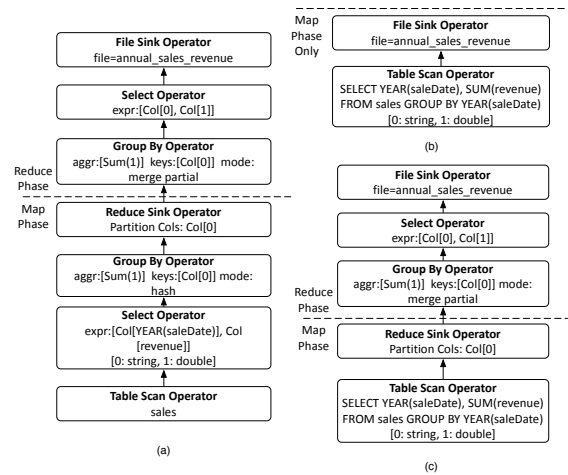
Figure 2: (a) MapReduce job generated by Hive (b) MapReduce job generated by SMS assuming sales is partitioned by YEAR(saleDate). This feature is still unsupported (c) MapReduce job generated by SMS assuming no partitioning of sales

phases of a query plan. The Hive optimizer is a simple, naïve, rule-based optimizer. It does not use cost-based optimization techniques. Therefore, it does not always generate efficient query plans. This is another advantage of pushing as much as possible of the query processing logic into DBMSs that have more sophisticated, adaptive or cost-based optimizers.

(5) Finally, the physical plan generator converts the logical query plan into a physical plan executable by one or more MapReduce jobs. The first and every other Reduce Sink operator marks a transition from a Map phase to a Reduce phase of a MapReduce job and the remaining Reduce Sink operators mark the start of new MapReduce jobs. The above SQL query results in a single MapReduce job with the physical query plan illustrated in Fig. 2(a). The boxes stand for the operators and the arrows represent the flow of data.

(6) Each DAG enclosed within a MapReduce job is serialized into an XML plan. The Hive driver then executes a Hadoop job. The job reads the XML plan and creates all the necessary operator objects that scan data from a table in HDFS, and parse and process one tuple at a time.

The SMS planner modifies Hive. In particular we intercept the normal Hive flow in two main areas:

(i) Before any query execution, we update the MetaStore with references to our database tables. Hive allows tables to exist externally, outside HDFS. The HadoopDB catalog, Section 5.2.2, provides information about the table schemas and required Deserializer and InputFormat classes to the MetaStore. We implemented these specialized classes.

(ii) After the physical query plan generation and before the execution of the MapReduce jobs, we perform two passes over the physical plan. In the first pass, we retrieve data fields that are actually processed by the plan and we determine the partitioning keys used by the Reduce Sink (Repartition) operators. In the second pass, we traverse the DAG bottom-up from table scan operators to the output or File Sink operator. All operators until the first repartition operator with a partitioning key different from the database's key are converted into one or more SQL queries and pushed into the database layer. SMS uses a rule-based SQL generator to recreate SQL from the relational operators. The query processing logic that could be pushed into the database layer ranges from none (each

table is scanned independently and tuples are pushed one at a time into the DAG of operators) to all (only a Map task is required to output the results into an HDFS file).

Given the above GroupBy query, SMS produces one of two different plans. If the sales table is partitioned by YEAR(saleDate), it produces the query plan in Fig. 2(b): this plan pushes the entire query processing logic into the database layer. Only a Map task is required to output results into an HDFS file. Otherwise, SMS produces the query plan in Fig. 2(c) in which the database layer partially aggregates data and eliminates the selection and group-by operator used in the Map phase of the Hive generated query plan (Fig. 2(a)). The final aggregation step in the Reduce phase of the MapReduce job, however, is still required in order to merge partial results from each node.

For join queries, Hive assumes that tables are not collocated. Therefore, the Hive generated plan scans each table independently and computes the join after repartitioning data by the join key. In contrast, if the join key matches the database partitioning key, SMS pushes the entire join sub-tree into the database layer.

So far, we only support filter, select (project) and aggregation operators. Currently, the partitioning features supported by Hive are extremely naïve and do not support expression-based partitioning. Therefore, we cannot detect if the sales table is partitioned by YEAR(saleDate) or not, therefore we have to make the pessimistic assumption that the data is not partitioned by this attribute. The Hive build [15] we extended is a little buggy; as explained in Section 6.2.5, it fails to execute the join task used in our benchmark, even when running over HDFS tables[3]. However, we use the SMS planner to automatically push SQL queries into HadoopDB's DBMS layer for all other benchmark queries presented in our experiments for this paper.

5.3 Summary

HadoopDB does not replace Hadoop. Both systems coexist enabling the analyst to choose the appropriate tools for a given dataset and task. Through the performance benchmarks in the following sections, we show that using an efficient database storage layer cuts down on data processing time especially on tasks that require complex query processing over structured data such as joins. We also show that HadoopDB is able to take advantage of the fault-tolerance and the ability to run on heterogeneous environments that comes naturally with Hadoop-style systems.

6. BENCHMARKS

In this section we evaluate HadoopDB, comparing it with a MapReduce implementation and two parallel database implementations, using a benchmark first presented in [23][4]. This benchmark consists of five tasks. The first task is taken directly from the original MapReduce paper [8] whose authors claim is representative of common MR tasks. The next four tasks are analytical queries designed to be representative of traditional structured data analysis workloads that HadoopDB targets.

We ran our experiments on Amazon EC2 "large" instances (zone: us-east-1b). Each instance has 7.5 GB memory, 4 EC2 Compute Units (2 virtual cores), 850 GB instance storage (2 x 420 GB plus 10 GB root partition) and runs 64-bit platform Linux Fedora 8 OS.

We observed that disk I/O performance on EC2 nodes were initially quite slow (25MB/s). Consequently, we initialized some additional space on each node so that intermediate files and output of the tasks did not suffer from this initial write slow-down. Once disk space is initialized, subsequent writes are much faster (86MB/s). Network speed is approximately 100-110MB/s. We execute each task three times and report the average of the trials. The final results from all parallel databases queries are piped from the shell command into a file. Hadoop and HadoopDB store results in Hadoop's distributed file system (HDFS). In this section, we only report results using trials where all nodes are available, operating correctly, and have no concurrent tasks during benchmark execution (we drop these requirements in Section 7). For each task, we benchmark performance on cluster sizes of 10, 50, and 100 nodes.

6.1 Benchmarked Systems

Our experiments compare performance of Hadoop, HadoopDB (with PostgreSQL[5] as the underlying database) and two commercial parallel DBMSs.

6.1.1 Hadoop

Hadoop is an open-source version of the MapReduce framework, implemented by directly following the ideas described in the original MapReduce paper, and is used today by dozens of businesses to perform data analysis [1]. For our experiments in this paper, we use Hadoop version 0.19.1 running on Java 1.6.0. We deployed the system with several changes to the default configuration settings. Data in HDFS is stored using 256MB data blocks instead of the default 64MB. Each MR executor ran with a maximum heap size of 1024MB. We allowed two Map instances and a single Reduce instance to execute concurrently on each node. We also allowed more buffer space for file read/write operations (132MB) and increased the sort buffer to 200MB with 100 concurrent streams for merging. Additionally, we modified the number of parallel transfers run by Reduce during the shuffle phase and the number of worker threads for each TaskTracker's http server to be 50. These adjustments follow the guidelines on high-performance Hadoop clusters [13]. Moreover, we enabled task JVMs to be reused.

For each benchmark trial, we stored all input and output data in HDFS with no replication (we add replication in Section 7). After benchmarking a particular cluster size, we deleted the data directories on each node, reformatted and reloaded HDFS to ensure uniform data distribution across all nodes.

We present results of both hand-coded Hadoop and Hive-coded Hadoop (i.e. Hadoop plans generated automatically via Hive's SQL interface). These separate results for Hadoop are displayed as split bars in the graphs. The bottom, colored segment of the bars represent the time taken by Hadoop when hand-coded and the rest of the bar indicates the additional overhead as a result of the automatic plan-generation by Hive, and operator function-call and dynamic data type resolution through Java's Reflection API for each tuple processed in Hive-coded jobs.

6.1.2 HadoopDB

The Hadoop part of HadoopDB was configured identically to the description above except for the number of concurrent Map tasks, which we set to one. Additionally, on each worker node, PostgreSQL version 8.2.5 was installed. We increased memory used by the PostgreSQL shared buffers to 512 MB and the working memory

---

[3]The Hive team resolved these issues in June after we completed the experiments. We plan to integrate the latest Hive with the SMS Planner.

[4]We are aware of the writing law that references shouldn't be used as nouns. However, to save space, we use [23] not as a reference, but as a shorthand for "the SIGMOD 2009 paper by Pavlo et. al."

[5]Initially, we experimented with MySQL (MyISAM storage layer). However, we found that while simple table scans are up to 30% faster, more complicated SQL queries are much slower due to the lack of clustered indices and poor join algorithms.

size to 1GB. We did not compress data in PostgreSQL.

Analogous to what we did for Hadoop, we present results of both hand-coded HadoopDB and SMS-coded HadoopDB (i.e. entire query plans created by HadoopDB's SMS planner). These separate results for HadoopDB are displayed as split bars in the graphs. The bottom, colored segment of the bars represents the time taken by HadoopDB when hand-coded and the rest of the bar indicates the additional overhead as a result of the SMS planner (e.g., SMS jobs need to serialize tuples retrieved from the underlying database and deserialize them before further processing in Hadoop).

6.1.3 Vertica

Vertica is a relatively new parallel database system (founded in 2005) [3] based on the C-Store research project [24]. Vertica is a column-store, which means that each attribute of each table is stored (and accessed) separately, a technique that has proven to improve performance for read-mostly workloads.

Vertica offers a "cloud" edition, which we used for the experiments in this paper. Vertica was also used in the performance study of previous work [23] on the same benchmark, so we configured Vertica identically to the previous experiments[6]. The Vertica configuration is therefore as follows: All data is compressed. Vertica operates on compressed data directly. Vertica implements primary indexes by sorting the table by the indexed attribute. None of Vertica's default configuration parameters were changed.

6.1.4 DBMS-X

DBMS-X is the same commercial parallel row-oriented database as was used for the benchmark in [23]. Since at the time of our VLDB submission this DBMS did not offer a cloud edition, we did not run experiments for it on EC2. However, since our Vertica numbers were consistently 10-15% slower on EC2 than on the Wisconsin cluster presented in [23][7] (this result is expected since the virtualization layer is known to introduce a performance overhead), we reproduce the DBMS-X numbers from [23] on our figures as a best case performance estimate for DBMS-X if it were to be run on EC2.

6.2 Performance and Scalability Benchmarks

The first benchmark task (the "Grep task") requires each system to scan through a data set of 100-byte records looking for a three character pattern. This is the only task that requires processing largely unstructured data, and was originally included in the benchmark by the authors of [23] since the same task was included in the original MapReduce paper [8].

To explore more complex uses of the benchmarked systems, the benchmark includes four more analytical tasks related to log-file analysis and HTML document processing. Three of these tasks operate on structured data; the final task operates on both structured and unstructured data.

The datasets used by these four tasks include a UserVisits table meant to model log files of HTTP server traffic, a Documents table containing 600,000 randomly generated HTML documents, and a Rankings table that contains some metadata calculated over the data in the Documents table. The schema of the tables in the benchmark data set is described in detail in [23]. In summary, the UserVisits table contains 9 attributes, the largest of which is destinationURL which is of type VARCHAR(100). Each tuple is on the order of 150 bytes wide. The Documents table contains two attributes: a URL

(VARCHAR(100)) and contents (arbitrary text). Finally, the Rankings table contains three attributes: pageURL (VARCHAR(100)), pageRank (INT), and avgDuration(INT).

The data generator yields 155 million UserVisits records (20GB) and 18 million Rankings records (1GB) per node. Since the data generator does not ensure that Rankings and UserVisits tuples with the same value for the URL attribute are stored on the same node, a repartitioning is done during the data load, as described later.

Records for both the UserVisits and Rankings data sets are stored in HDFS as plain text, one record per line with fields separated by a delimiting character. In order to access the different attributes at run time, the Map and Reduce functions split the record by the delimiter into an array of strings.

6.2.1 Data Loading

We report load times for two data sets, Grep and UserVisits in Fig. 3 and Fig. 4. While grep data is randomly generated and requires no preprocessing, UserVisits needs to be repartitioned by destinationURL and indexed by visitDate for all databases during the load in order to achieve better performance on analytical queries (Hadoop would not benefit from such repartitioning). We describe, briefly, the loading procedures for all systems:

Hadoop: We loaded each node with an unaltered UserVisits data file. HDFS automatically breaks the file into 256MB blocks and stores the blocks on a local DataNode. Since all nodes load their data in parallel, we report the maximum node load time from each cluster. Load time is greatly affected by stragglers. This effect is especially visible when loading UserVisits, where a single slow node in the 100-node cluster pushed the overall load time to 4355 seconds and to 2600 seconds on the 10-node cluster, despite the average load time of only 1100 seconds per node.

HadoopDB: We set the maximum chunk size to 1GB. Each chunk is located in a separate PostgreSQL database within a node, and processes SQL queries independently of other chunks. We report the maximum node load time as the entire load time for both Grep and UserVisits.

Since the Grep dataset does not require any preprocessing and is only 535MB of data per node, the entire data was loaded using the standard SQL COPY command into a single chunk on each node.

The Global Hasher partitions the entire UserVisits dataset across all nodes in the cluster. Next, the Local Hasher on each node retrieves a 20GB partition from HDFS and hash-partitions it into 20 smaller chunks, 1GB each. Each chunk is then bulk-loaded using COPY. Finally, a clustered index on visitDate is created for each chunk.

The load time for UserVisits is broken down into several phases. The first repartition carried out by Global Hasher is the most expensive step in the process. It takes nearly half the total load time, 14,000 s. Of the remaining 16,000 s, locally partitioning the data into 20 chunks takes 2500 s (15.6%), the bulk copy into tables takes 5200 s (32.5%), creating clustered indices, which includes sorting, takes 7100 s (44.4%), finally vacuuming the databases takes 1200 s (7.5%). All the steps after global repartitioning are executed in parallel on all nodes. We observed individual variance in load times. Some nodes required as little as 10,000 s to completely load UserVisits after global repartitioning was completed.

Vertica: The loading procedure for Vertica is analogous to the one described in [23]. The loading time improved since then because a newer version of Vertica (3.0) was used for these experiments. The key difference is that now bulk load COPY command runs on all nodes in the cluster completely in parallel.

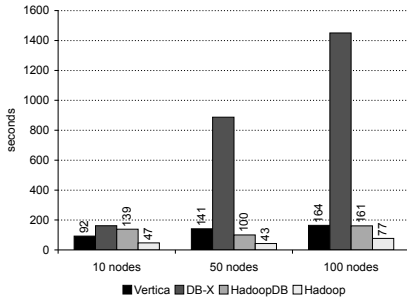**DBMS-X:** We report the total load time including data compression and indexing from [23].

---

[6]In fact, we asked the same person who ran the queries for this previous work to run the same queries on EC2 for our paper

[7]We used a later version of Vertica in these experiments than [23]. On using the identical version, slowdown was 10-15% on EC2.
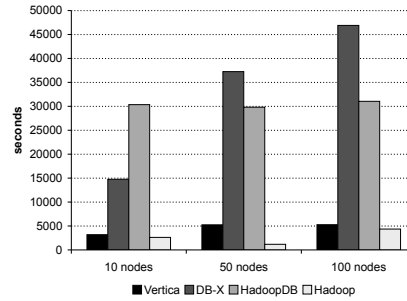
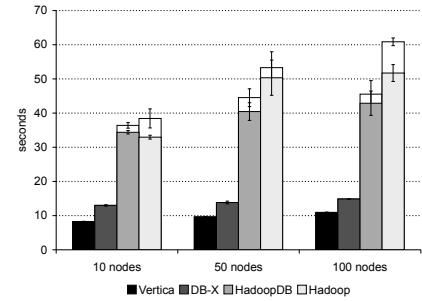**Figure 3: Load Grep (0.5GB/node)**   **Figure 4: Load UserVisits (20GB/node)**   **Figure 5: Grep Task**

In contrast to DBMS-X, the parallel load features of Hadoop, HadoopDB and Vertica ensure all systems scale as the number of nodes increases. Since the speed of loading is limited by the slowest disk-write speed in the cluster, loading is the only process that cannot benefit from Hadoop's and HadoopDB's inherent tolerance of heterogeneous environments (see section 7)[8].

6.2.2 Grep Task

Each record consists of a unique key in the first 10 bytes, followed by a 90-byte character string. The pattern "XYZ" is searched for in the 90 byte field, and is found once in every 10,000 records. Each node contains 5.6 million such 100-byte records, or roughly 535MB of data. The total number of records processed for each cluster size is 5.6 million times the number of nodes.

Vertica, DBMS-X, HadoopDB, and Hadoop(Hive) all executed the identical SQL:

```
SELECT * FROM Data WHERE field LIKE '%XYZ%';
```

None of the benchmarked systems contained an index on the field attribute. Hence, for all systems, this query requires a full table scan and is mostly limited by disk speed.

Hadoop (hand-coded) was executed identically to [23] (a simple Map function that performs a sub-string match on "XYZ"). No Reduce function is needed for this task, so the output of the Map function is written directly to HDFS.

HadoopDB's SMS planner pushes the WHERE clause into the PostgreSQL instances.

Fig. 5 displays the results (note, the split bars were explained in Section 6.1). HadoopDB slightly outperforms Hadoop as it handles I/O more efficiently than Hadoop due to the lack of runtime parsing of data. However, both systems are outperformed by the parallel databases systems. This difference is due to the fact that both Vertica and DBMS-X compress their data, which significantly reduces I/O cost ([23] note that compression speeds up DBMS-X by about 50% on all experiments).

6.2.3 Selection Task

The first structured data task evaluates a simple selection predicate on the pageRank attribute from the Rankings table. There are approximately 36,000 tuples on each node that pass this predicate.

Vertica, DBMS-X, HadoopDB, and Hadoop(Hive) all executed the identical SQL:

```
SELECT pageURL, pageRank FROM Rankings WHERE pageRank > 10;
```

Hadoop (hand-coded) was executed identically to [23]: a Map function parses Rankings tuples using the field delimiter, applies the predicate on pageRank, and outputs the tuple's pageURL and

---

[8]EC2 disks are slow on initial writes. Since performance benchmarks are not write-limited, they are not affected by disk-write speeds. Also, we initialized disks before experiments (see Section 6).

pageRank as a new key/value pair if the predicate succeeds. This task does not require a Reduce function.

HadoopDB's SMS planner pushes the selection and projection clauses into the PostgreSQL instances.

The performance of each system is presented in Fig. 6. Hadoop (with and without Hive) performs a brute-force, complete scan of all data in a file. The other systems, however, benefit from using clustered indices on the pageRank column. Hence, in general HadoopDB and the parallel DBMSs are able to outperform Hadoop.

Since data is partitioned by UserVisits destinationURL, the foreign key relationship between Rankings pageURL and UserVisits destinationURL causes the Global and Local Hasher to repartition Rankings by pageURL. Each Rankings chunk is only 50 MB (collocated with the corresponding 1GB UserVisits chunk). The overhead of scheduling twenty Map tasks to process only 1GB of data per node significantly decreases HadoopDB's performance.

We, therefore, maintain an additional, non-chunked copy of the Rankings table containing the entire 1GB. HadoopDB on this data set outperforms Hadoop because the use of a clustered index on pageRank eliminates the need to sequentially scan the entire data set. HadoopDB scales better relative to DBMS-X and Vertica mainly due to increased network costs of these systems which dominate when query time is otherwise very low.

6.2.4 Aggregation Task

The next task involves computing the total adRevenue generated from each sourceIP in the UserVisits table, grouped by either the seven-character prefix of the sourceIP column or the entire sourceIP column. Unlike the previous tasks, this task requires intermediate results to be exchanged between different nodes in the cluster (so that the final aggregate can be calculated). When grouping on the seven-character prefix, there are 2000 unique groups. When grouping on the entire sourceIP, there are 2,500,000 unique groups.

Vertica, DBMS-X, HadoopDB, and Hadoop(Hive) all executed the identical SQL:

```
Smaller query:
SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
FROM UserVisits GROUP BY SUBSTR(sourceIP, 1, 7);
Larger query:
SELECT sourceIP, SUM(adRevenue) FROM UserVisits
GROUP BY sourceIP;
```

Hadoop (hand-coded) was executed identically to [23]: a Map function outputs the adRevenue and the first seven characters of the sourceIP field (or the whole field in the larger query) which gets sent to a Reduce function which performs the sum aggregation for each prefix (or sourceIP).

The SMS planner for HadoopDB pushes the entire SQL query into the PostgreSQL instances. The output is then sent to Reduce jobs inside of Hadoop that perform the final aggregation (after collecting all pre-aggregated sums from each PostgreSQL instance).
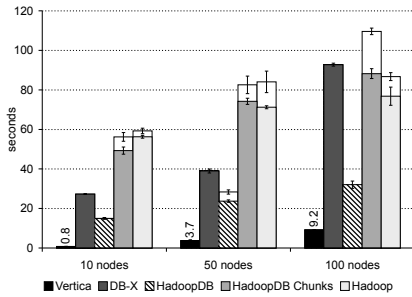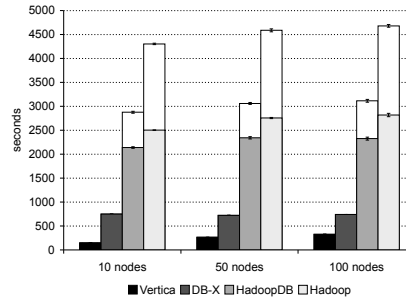
Figure 6: Selection Task
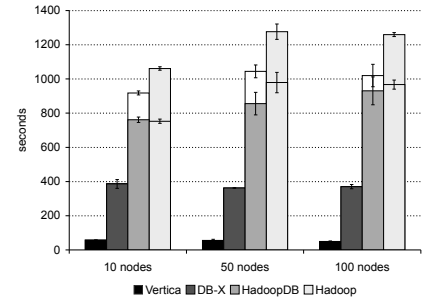

Figure 7: Large Aggregation Task

Figure 8: Small Aggregation Task

The performance numbers for each benchmarked system is displayed in Fig. 7 and 8. Similar to the Grep task, this query is limited by reading data off disk. Thus, both commercial systems benefit from compression and outperform HadoopDB and Hadoop.

We observe a reversal of the general rule that Hive adds an overhead cost to hand-coded Hadoop in the "small" (substring) aggregation task (the time taken by Hive is represented by the lower part of the Hadoop bar in Fig. 8). Hive performs much better than Hadoop because it uses a hash aggregation execution strategy (it maintains an internal hash-aggregate map in the Map phase of the job), which proves to be optimal when there is a small number of groups. In the large aggregation task, Hive switches to sort-based aggregation upon detecting that the number of groups is more than half the number of input rows per block. In contrast, in our hand-coded Hadoop plan we (and the authors of [23]) failed to take advantage of hash aggregation for the smaller query because sort-based aggregation (using Combiners) is a MapReduce standard practice.

These results illustrate the benefit of exploiting optimizers present in database systems and relational query systems like Hive, which can use statistics from the system catalog or simple optimization rules to choose between hash aggregation and sort aggregation.

Unlike Hadoop's Combiner, Hive serializes partial aggregates into strings instead of maintaining them in their natural binary representation. Hence, Hive performs much worse than Hadoop on the larger query.

PostgreSQL chooses to use hash aggregation for both tasks as it can easily fit the entire hash aggregate table for each 1GB chunk in memory. Hence, HadoopDB outperforms Hadoop on both tasks due to its efficient aggregation implementation.

This query is well-suited for systems that use column-oriented storage, since the two attributes accessed in this query (sourceIP and adRevenue) consist of only 20 out of the more than 200 bytes in each UserVisits tuple. Vertica is thus able to significantly outperform the other systems due to the commensurate I/O savings.

6.2.5 Join Task

The join task involves finding the average pageRank of the set of pages visited from the sourceIP that generated the most revenue during the week of January 15-22, 2000. The key difference between this task and the previous tasks is that it must read in two different data sets and join them together (pageRank information is found in the Rankings table and revenue information is found in the UserVisits table). There are approximately 134,000 records in the UserVisits table that have a visitDate value inside the requisite date range.

Unlike the previous three tasks, we were unable to use the same SQL for the parallel databases and for Hadoop-based systems. This is because the Hive build we extended was unable to execute this query. Although this build accepts a SQL query that joins, filters and aggregates tuples from two tables, such a query fails during execution. Additionally, we noticed that the query plan for joins of this type uses a highly inefficient execution strategy. In particular, the filtering operation is planned after joining the tables. Hence, we are only able to present hand-coded results for HadoopDB and Hadoop for this query.

In HadoopDB, we push the selection, join, and partial aggregation into the PostgreSQL instances with the following SQL:

```
SELECT sourceIP, COUNT(pageRank), SUM(pageRank),
SUM(adRevenue) FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL AND
UV.visitDate BETWEEN '2000-01-15' AND '2000-01-22'
GROUP BY UV.sourceIP;
```

We then use a single Reduce task in Hadoop that gathers all of the partial aggregates from each PostgreSQL instance to perform the final aggregation.

The parallel databases execute the SQL query specified in [23].

Although Hadoop has support for a join operator, this operator requires that both input datasets be sorted on the join key. Such a requirement limits the utility of the join operator since in many cases, including the query above, the data is not already sorted and performing a sort before the join adds significant overhead. We found that even if we sorted the input data (and did not include the sort time in total query time), query performance using the Hadoop join was lower than query performance using the three phase MR program used in [23] that used standard 'Map' and 'Reduce' operators. Hence, for the numbers we report below, we use an identical MR program as was used (and described in detail) in [23].

Fig. 9 summarizes the results of this benchmark task. For Hadoop, we observed similar results as found in [23]: its performance is limited by completely scanning the UserVisits dataset on each node in order to evaluate the selection predicate.

HadoopDB, DBMS-X, and Vertica all achieve higher performance by using an index to accelerate the selection predicate and having native support for joins. These systems see slight performance degradation with a larger number of nodes due to the final single node aggregation of and sorting by adRevenue.

6.2.6 UDF Aggregation Task

The final task computes, for each document, the number of inward links from other documents in the Documents table. URL links that appear in every document are extracted and aggregated.

HTML documents are concatenated into large files for Hadoop (256MB each) and Vertica (56MB each) at load time. HadoopDB was able to store each document separately in the Documents table using the TEXT data type. DBMS-X processed each HTML document file separately, as described below.

The parallel databases should theoretically be able to use a user-defined function, F, to parse the contents of each document and
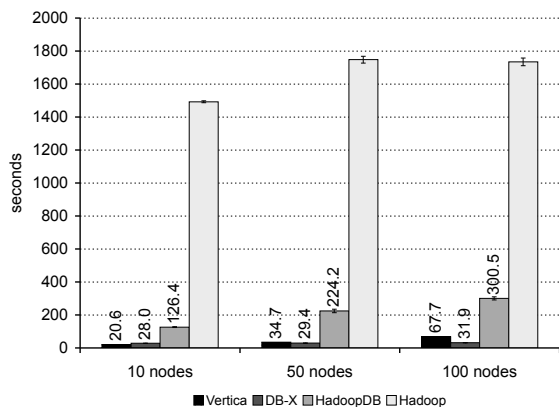
Figure 9: Join Task

Figure 10: UDF Aggregation task

emit a list of all URLs found in the document. A temporary table would then be populated with this list of URLs and then a simple count/group-by query would be executed that finds the number of instances of each unique URL.

Unfortunately, [23] found that in practice, it was difficult to implement such a UDF inside the parallel databases. In DBMS-X, it was impossible to store each document as a character BLOB inside the DBMS and have the UDF operate on it directly, due to "a known bug in [the] version of the system". Hence, the UDF was implemented inside the DBMS, but the data was stored in separate HTML documents on the raw file system and the UDF made external calls accordingly.

Vertica does not currently support UDFs, so a simple document parser had to be written in Java externally to the DBMS. This parser is executed on each node in parallel, parsing the concatenated documents file and writing the found URLs into a file on the local disk. This file is then loaded into a temporary table using Vertica's bulk-loading tools and a second query is executed that counts, for each URL, the number of inward links.

In Hadoop, we employed standard TextInputFormat and parsed each document inside a Map task, outputting a list of URLs found in each document. Both a Combine and a Reduce function sum the number of instances of each unique URL.

In HadoopDB, since text processing is more easily expressed in MapReduce, we decided to take advantage of HadoopDB's ability to accept queries in either SQL or MapReduce and we used the latter option in this case. The complete contents of the Documents table on each PostgreSQL node is passed into Hadoop with the following SQL:

```
SELECT url, contents FROM Documents;
```

Next, we process the data using a MR job. In fact, we used identical MR code for both Hadoop and HadoopDB.

Fig. 10 illustrates the power of using a hybrid system like HadoopDB. The database layer provides an efficient storage layer for HTML text documents and the MapReduce framework provides arbitrary processing expression power.

Hadoop outperforms HadoopDB as it processes merged files of multiple HTML documents. HadoopDB, however, does not lose the original structure of the data by merging many small files into larger ones. Note that the total merge time was about 6000 seconds per node. This overhead is not included in Fig. 10.

DBMS-X and Vertica perform worse than Hadoop-based systems since the input files are stored outside of the database. Moreover, for this task both commercial databases do not scale linearly with the size of the cluster.
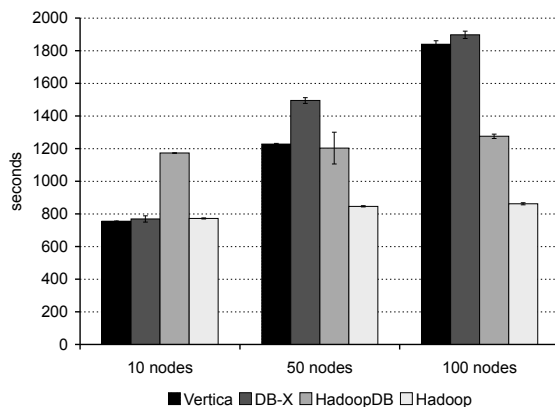
6.3 Summary of Results Thus Far

In the absence of failures or background processes, HadoopDB is able to approach the performance of the parallel database systems. The reason the performance is not equal is due to the following facts: (1) PostgreSQL is not a column-store (2) DBMS-X results are overly optimistic by approximately a factor of 15%, (3) we did not use data compression in PostgreSQL, and (4) there is some overhead in the interaction between Hadoop and PostgreSQL which gets proportionally larger as the number of chunks increases. We believe some of this overhead can be removed with the increase of engineering time.

HadoopDB consistently outperforms Hadoop (except for the UDF aggregation task since we did not count the data merging time against Hadoop).
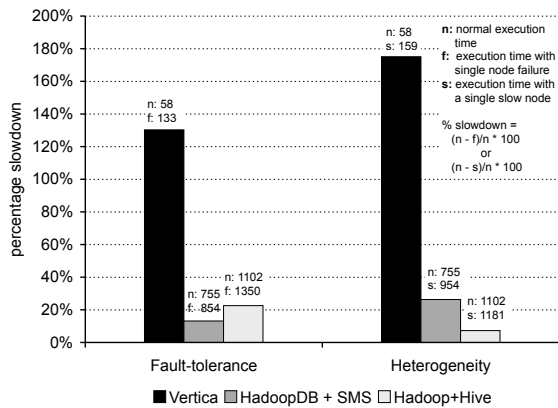
While HadoopDB's load time is about 10 times longer than Hadoop's, this cost is amortized across the higher performance of all queries that process this data. For certain tasks, such as the Join task, the factor of 10 load cost is immediately translated into a factor of 10 performance benefit.

7. FAULT TOLERANCE AND HETEROGE-NEOUS ENVIRONMENT

As described in Section 3, in large deployments of shared-nothing machines, individual nodes may experience high rates of failure or slowdown. While running our experiments for this research paper on EC2, we frequently experienced both node failure and node slowdown (e.g., some notifications we received: "4:12 PM PDT: We are investigating a localized issue in a single US-EAST Availability Zone. As a result, a small number of instances are unreachable. We are working to restore the instances.", and "Starting at 11:30PM PDT today, we will be performing maintenance on parts of the Amazon EC2 network. This maintenance has been planned to minimize the probability of impact to Amazon EC2 instances, but it is possible that some customers may experience a short period of elevated packet loss as the change takes effect.")

For parallel databases, query processing time is usually determined by the the time it takes for the slowest node to complete its task. In contrast, in MapReduce, each task can be scheduled on any node as long as input data is transferred to or already exists on a free node. Also, Hadoop speculatively redundantly executes tasks that are being performed on a straggler node to reduce the slow node's effect on query time.

Hadoop achieves fault tolerance by restarting tasks of failed

Figure with axis "percentage slowdown" from 0% to 200%. Legend:

n: normal execution time
f: execution time with single node failure
s: execution time with a single slow node

% slowdown =
(n - f)/n * 100
or
(n - s)/n * 100

Data labels:
- Fault-tolerance, Vertica: n: 58, f: 133
- Fault-tolerance, HadoopDB + SMS: n: 755, f: 854
- Fault-tolerance, Hadoop+Hive: n: 1102, f: 1350
- Heterogeneity, Vertica: n: 58, s: 159
- Heterogeneity, HadoopDB + SMS: n: 755, s: 954
- Heterogeneity, Hadoop+Hive: n: 1102, s: 1181

X-axis categories: Fault-tolerance, Heterogeneity

Legend: ■ Vertica ▨ HadoopDB + SMS □ Hadoop+Hive

Figure 11: Fault tolerance and heterogeneity experiments on 10 nodes

nodes on other nodes. The JobTracker receives heartbeats from TaskTrackers. If a TaskTracker fails to communicate with the JobTracker for a preset period of time, *TaskTracker expiry interval*, the JobTracker assumes failure and schedules all map/reduce tasks of the failed node on other TaskTrackers. This approach is different from most parallel databases which abort unfinished queries upon a node failure and restart the entire query processing (using a replica node instead of the failed node).

By inheriting the scheduling and job tracking features of Hadoop, HadoopDB yields similar fault-tolerance and straggler handling properties as Hadoop.

To test the effectiveness of HadoopDB in failure-prone and heterogeneous environments in comparison to Hadoop and Vertica, we executed the aggregation query with 2000 groups (see Section 6.2.4) on a 10-node cluster and set the replication factor to two for all systems. For Hadoop and HadoopDB we set the TaskTracker expiry interval to 60 seconds. The following lists system-specific settings for the experiments.

Hadoop (Hive): HDFS managed the replication of data. HDFS replicated each block of data on a different node selected uniformly at random.

HadoopDB (SMS): As described in Section 6, each node contains twenty 1GB-chunks of the UserVisits table. Each of these 20 chunks was replicated on a different node selected at random.

Vertica: In Vertica, replication is achieved by keeping an extra copy of every table segment. Each table is hash partitioned across the nodes and a backup copy is assigned to another node based on a replication rule. On node failure, this backup copy is used until the lost segment is rebuilt.

For fault-tolerance tests, we terminated a node at 50% query completion. For Hadoop and HadoopDB, this is equivalent to failing a node when 50% of the scheduled Map tasks are done. For Vertica, this is equivalent to failing a node after 50% of the average query completion time for the given query.

To measure percentage increase in query time in heterogeneous environments, we slow down a node by running an I/O-intensive background job that randomly seeks values from a large file and frequently clears OS caches. This file is located on the same disk where data for each system is stored.

We observed no differences in percentage slowdown between HadoopDB with or without SMS and between Hadoop with or without Hive. Therefore, we only report results of HadoopDB with SMS and Hadoop with Hive and refer to both systems as HadoopDB and Hadoop from now on.

The results of the experiments are shown in Fig. 11. Node failure caused HadoopDB and Hadoop to have smaller slowdowns than Vertica. Vertica's increase in total query execution time is due to the overhead associated with query abortion and complete restart.

In both HadoopDB and Hadoop, the tasks of the failed node are distributed over the remaining available nodes that contain replicas of the data. HadoopDB slightly outperforms Hadoop. In Hadoop TaskTrackers assigned blocks not local to them will copy the data first (from a replica) before processing. In HadoopDB, however, processing is pushed into the (replica) database. Since the number of records returned after query processing is less than the raw size of data, HadoopDB does not experience Hadoop's network overhead on node failure.

In an environment where one node is extremely slow, HadoopDB and Hadoop experience less than 30% increase in total query execution time, while Vertica experiences more than a 170% increase in query running time. Vertica waits for the straggler node to complete processing. HadoopDB and Hadoop run speculative tasks on TaskTrackers that completed their tasks. Since the data is chunked (HadoopDB has 1GB chunks, Hadoop has 256MB blocks), multiple TaskTrackers concurrently process different replicas of unprocessed blocks assigned to the straggler. Thus, the delay due to processing those blocks is distributed across the cluster.

In our experiments, we discovered an assumption made by Hadoop's task scheduler that contradicts the HadoopDB model. In Hadoop, TaskTrackers will copy data not local to them from the straggler or the replica. HadoopDB, however, does not move PostgreSQL chunks to new nodes. Instead, the TaskTracker of the redundant task connects to either the straggler's database or the replica's database. If the TaskTracker connects to the straggler's database, the straggler needs to concurrently process an additional query leading to further slowdown. Therefore, the same feature that causes HadoopDB to have slightly better fault tolerance than Hadoop, causes a slightly higher percentage slow down in heterogeneous environments for HadoopDB. We plan to modify the current task scheduler implementation to provide hints to speculative TaskTrackers to avoid connecting to a straggler node and to connect to replicas instead.

7.1 Discussion

It should be pointed out that although Vertica's percentage slowdown was larger than Hadoop and HadoopDB, its total query time (even with the failure or the slow node) was still lower than Hadoop or HadoopDB. Furthermore, Vertica's performance in the absence of failures is an order of magnitude faster than Hadoop and HadoopDB (mostly because its column-oriented layout of data is a big win for the small aggregation query). This order of magnitude of performance could be translated to the same performance as Hadoop and HadoopDB, but using an order of magnitude fewer nodes. Hence, failures and slow nodes become less likely for Vertica than for Hadoop and HadoopDB. Furthermore, eBay's 6.5 petabyte database (perhaps the largest known data warehouse worldwide as of June 2009) [4] uses only 96 nodes in a shared-nothing cluster. Failures are still reasonably rare at fewer than 100 nodes.

We argue that in the future, 1000-node clusters will be commonplace for production database deployments, and 10,000-node clusters will not be unusual. There are three trends that support this prediction. First, data production continues to grow faster than Moore's law (see Section 1). Second, it is becoming clear that from both a price/performance and (an increasingly important) power/performance perspective, many low-cost, low-power servers are far better than fewer heavy-weight servers [14]. Third, there

is now, more than ever, a requirement to perform data analysis inside of the DBMS, rather than pushing data to external systems for analysis. Disk-heavy architectures such as the eBay 96-node DBMS do not have the necessary CPU horsepower for analytical workloads [4].

Hence, awaiting us in the future are heavy-weight analytic database jobs, requiring more time and more nodes. The probability of failure in these next generation applications will be far larger than it is today, and restarting entire jobs upon a failure will be unacceptable (failures might be common enough that long-running jobs never finish!) Thus, although Hadoop and HadoopDB pay a performance penalty for runtime scheduling, block-level restart, and frequent checkpointing, such an overhead to achieve robust fault tolerance will become necessary in the future. One feature of HadoopDB is that it can elegantly transition between both ends of the spectrum. Since one chunk is the basic unit of work, it can play in the high-performance/low-fault-tolerance space of today's workloads (like Vertica) by setting a chunk size to be infinite, or in high fault tolerance by using more granular chunks (like Hadoop). In future work, we plan to explore the fault-tolerance/performance tradeoff in more detail.

8. CONCLUSION

Our experiments show that HadoopDB is able to approach the performance of parallel database systems while achieving similar scores on fault tolerance, an ability to operate in heterogeneous environments, and software license cost as Hadoop. Although the performance of HadoopDB does not in general match the performance of parallel database systems, much of this was due to the fact that PostgreSQL is not a column-store and we did not use data compression in PostgreSQL. Moreover, Hadoop and Hive are relatively young open-source projects. We expect future releases to enhance performance. As a result, HadoopDB will automatically benefit from these improvements.

HadoopDB is therefore a hybrid of the parallel DBMS and Hadoop approaches to data analysis, achieving the performance and efficiency of parallel databases, yet still yielding the scalability, fault tolerance, and flexibility of MapReduce-based systems. The ability of HadoopDB to directly incorporate Hadoop and open source DBMS software (without code modification) makes HadoopDB particularly flexible and extensible for performing data analysis at the large scales expected of future workloads.

9. ACKNOWLEDGMENTS

10. REFERENCES

[1] Hadoop. Web Page. hadoop.apache.org/core/.
[2] HadoopDB Project. Web page. db.cs.yale.edu/hadoopdb/hadoopdb.html.
[3] Vertica. www.vertica.com/.
[4] 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.

[5] 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.
[6] 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.
[7] G. Czajkowski. Sorting 1pb with mapreduce. googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.
[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
[9] D. DeWitt and M. Stonebraker. MapReduce: A major step backwards. DatabaseColumn Blog. www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html.
[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*, 1986.
[11] Facebook. Hive. Web page. issues.apache.org/jira/browse/HADOOP-3601.
[12] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine. In *VLDB '86*, 1986.
[13] Hadoop Project. Hadoop Cluster Setup. Web Page. hadoop.apache.org/core/docs/current/cluster_setup.html.
[14] J. Hamilton. Cooperative expendable micro-slice servers (cems): Low cost, low power servers for internet-scale services. In *Proc. of CIDR*, 2009.
[15] Hive Project. Hive SVN Repository. Accessed May 19th 2009. svn.apache.org/viewvc/hadoop/hive/.
[16] J. N. Hoover. Start-Ups Bring Google's Parallel Processing To Data Warehousing. InformationWeek, August 29th, 2008.
[17] S. Madden, D. DeWitt, and M. Stonebraker. Database parallelism choices greatly impact scalability. DatabaseColumn Blog. www.databasecolumn.com/2007/10/database-parallelism-choices.html.
[18] Mayank Bawa. A $5.1M Addendum to our Series B. www.asterdata.com/blog/index.php/2009/02/25/a-51m-addendum-to-our-series-b/.
[19] C. Monash. The 1-petabyte barrier is crumbling. www.networkworld.com/community/node/31439.
[20] C. Monash. Cloudera presents the MapReduce bull case. DBMS2 Blog. www.dbms2.com/2009/04/15/cloudera-presents-the-mapreduce-bull-case/.
[21] C. Olofson. Worldwide RDBMS 2005 vendor shares. Technical Report 201692, IDC, May 2006.
[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of SIGMOD*, 2008.
[23] A. Pavlo, A. Rasin, S. Madden, M. Stonebraker, D. DeWitt, E. Paulson, L. Shrinivas, and D. J. Abadi. A Comparison of Approaches to Large Scale Data Analysis. In *Proc. of SIGMOD*, 2009.
[24] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, 2005.
[25] D. Vesset. Worldwide data warehousing tools 2005 vendor shares. Technical Report 203229, IDC, August 2006.