Efficient Processing of Data Warehousing Queries in a Split Execution Environment

Kamil Bajda-Pawlikowski[1+2], Daniel J. Abadi[1+2], Avi Silberschatz[2], Erik Paulson[3]
[1]Hadapt Inc., [2]Yale University, [3]University of Wisconsin-Madison
{kbajda,dna}@hadapt.com; avi@cs.yale.edu; epaulson@cs.wisc.edu

ABSTRACT

Hadapt is a start-up company currently commercializing the Yale University research project called HadoopDB. The company focuses on building a platform for Big Data analytics in the cloud by introducing a storage layer optimized for structured data and by providing a framework for executing SQL queries efficiently.

This work considers processing data warehousing queries over very large datasets. Our goal is to maximize performance while, at the same time, not giving up fault tolerance and scalability. We analyze the complexity of this problem in the split execution environment of HadoopDB. Here, incoming queries are examined; parts of the query are pushed down and executed inside the higher performing database layer; and the rest of the query is processed in a more generic MapReduce framework.

In this paper, we discuss in detail performance-oriented query execution strategies for data warehouse queries in split execution environments, with particular focus on join and aggregation operations. The efficiency of our techniques is demonstrated by running experiments using the TPC-H benchmark with 3TB of data. In these experiments we compare our results with a standard commercial parallel database and an open-source MapReduce implementation featuring a SQL interface (Hive). We show that HadoopDB successfully competes with other systems.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query processing

General Terms

Performance, Algorithms, Experimentation

Keywords

Query Execution, MapReduce, Hadoop

SIGMOD'11, June 12–16, 2011, Athens, Greece.

1. INTRODUCTION

MapReduce [19] is emerging as a leading framework for performing scalable parallel analytics and data mining. Some of the reasons for the popularity of MapReduce include the availability of a free and open source implementation (Hadoop) [2], impressive ease-of-use experience [30], as well as Google's, Yahoo!'s, and Facebook's wide usage [19, 25] and evangelization of this technology. Moreover, MapReduce has been shown to deliver stellar performance on extreme-scale benchmarks [17, 3]. All these factors have resulted in the rapid adoption of MapReduce for many different kinds of data analysis and processing [15, 18, 32, 29, 25, 11].

Historically, the main applications of the MapReduce framework included Web indexing, text analytics, and graph data mining.

Now, however, as MapReduce is steadily developing into the de facto data analysis standard, it repeatedly becomes employed for querying structured data — an area traditionally dominated by relational databases in data warehouse deployments. Even though many argue that MapReduce is not optimal for analyzing structured data [21, 30], it is nonetheless used increasingly frequently for that purpose because of a growing tendency to unify the data management platform. Thus, the standard structured data analysis can proceed side-by-side with the complex analytics that MapReduce is well-suited for. Moreover, data warehousing in this new platform enjoys the superior scalability of MapReduce [9] at a lower price. For example, Facebook famously ran a proof of concept comparing several parallel relational database vendors before deciding to run their 2.5 petabyte clickstream data warehouse using Hadoop [27] instead.

Consequently, in recent years a significant amount of research and commercial activity has focused on integrating MapReduce and relational database technology [31, 9, 24, 16, 34, 33, 22, 14]. There are two approaches to this problem: (1) Starting with a parallel database system and adding some MapReduce features [24, 16, 33], and (2) Starting with MapReduce and adding database system technology [31, 34, 9, 22, 14]. While both options are valid routes towards the integration, we expect that the second approach will ultimately prevail. This is because while there exists no widely available open source parallel database system, MapReduce is offered as an open source project. Furthermore, it is accompanied by a plethora of free tools, as well as cluster availability and support.

HadoopDB [9] follows the second of the approaches men-

tioned above. The technology developed at Yale University is commercialized by Hadapt [1]. The research project revealed that many of Hadoop's problems with performance on structured data can be attributed to a suboptimal storage layer. The default Hadoop storage layer, HDFS, is the distributed file system. When HDFS was replaced with multiple instances of a relational database system (one instance per node in a shared-nothing cluster), HadoopDB outperformed Hadoop's default configuration by up to an order of magnitude. The reason for the performance improvement can be attributed to leveraging decades' worth of research in the database systems community. Some optimizations developed during this period include the careful layout of data on disk, indexing, sorting, shared I/O, buffer management, compression, and query optimization. By combining the job scheduler, task coordination, and parallelization layer of Hadoop, with the storage layer of the DBMS, we were able to retain the best features of both systems. While achieving performance on structured data analysis comparable with commercial parallel database systems, we maitained Hadoop's fault tolerance, scalability, ability to handle heterogeneous node performance, and query interface flexibility.

In this paper, we describe several query execution and storage layer strategies that we developed to improve performance by yet another order of magnitude in comparison to the original research project. As a result, HadoopDB performs up to two orders of magnitude better than standard Hadoop. Furthermore, these modifications enabled HadoopDB to efficiently process significantly more complicated SQL queries. These include queries from the TPC-H benchmark — the most commonly used benchmark for comparing modern parallel database systems. The techniques we employ range from integrating with a column-store database system (in particular, one based on the MonetDB/X100 project), introducing referential partitioning to maximize the number of single-node joins, integrating semi-joins into the Hadoop Map phase, preparing aggregated data before performing joins, and combining joins and aggregation in a single Reduce phase.

Some of the strategies we discuss have been previously used or are currently available in commercial parallel database systems. What is interesting about these strategies in the context of HadoopDB, however, is the relative importance of the different techniques in a split query execution environment where both relational database systems and MapReduce are responsible for query processing. Futhermore, many commercial parallel DBMS vendors do not publish their query execution techniques in the research community. Therefore, while not necessarily new to implementation, some of the techniques presented in this paper are nevertheless new to publication.

In general, there are two heuristics that guide our optimizations:

1. Database systems can process data at a faster rate than Hadoop.

2. Each MapReduce job typically involves many I/O operations and network transfers. Thus, it is important to minimize the number of MapReduce jobs in a series into which a SQL query is translated.

Consequently, HadoopDB attempts to push as much processing as possible into single-node database systems and to perform as many relational query operators as possible in each "Map" and "Reduce" task. Our focus in this paper is on the processing of SQL queries by splitting their execution across Hadoop and DBMS. HadoopDB, however, also retains its ability to accept queries written directly in MapReduce.

In order to measure the relative effectiveness of our different query execution techniques, we selectively turn them on and off and measure the effect on the performance of HadoopDB for the TPC-H benchmark. Our primary comparison points are the first version of HadoopDB (without these techniques), and Hive, the currently dominant SQL interface to Hadoop. For continuity of comparison, we also benchmark against the same commercial parallel database system used in the original HadoopDB paper. HadoopDB shows consistently impressive performance that positions it as a legitimate player in the rapidly emerging market of "Big Data" analytics.

In addition to bringing high performance SQL to Hadoop, Hadapt adjusts on the fly to changing conditions in cloud environments. Hadapt is the only analytical database platform designed from scratch for cloud deployments. This paper does not discuss the cloud-based innovations of Hadapt. Rather, the sole focus is on the recent performance-oriented innovations developed in the Yale HadoopDB project.

2. BACKGROUND AND RELATED WORK

2.1 Hive and Hadoop

Hive [4] is an open-source data warehousing infrastructure built on top of Hadoop [2]. Hive accepts queries expressed in a SQL-like language called HiveQL and executes them against data stored in the Hadoop Distributed File System (HDFS).

A big limitation of the current implementation of Hive is its data storage layer. Because it is typically deployed on top of a distributed file system, Hive is unable to use hash-partitioning on a join key for the colocation of related tables — a typical strategy that parallel databases exploit to minimize data movement across nodes. Moreover, Hive workloads are very I/O heavy due to lack of native indexing. Furthermore, because the system catalog lacks statistics on data distribution, cost-based algorithms cannot be implemented in Hive's optimizer. We expect that Hive's developers will resolve these shortcomings in the future[1].

The original HadoopDB research project replaced HDFS with many single-node database systems. Besides yielding short-term performance benefits, this design made it easier to implement some standard parallel database techniques. Having achieved this, we can now focus on the more advanced split query execution techniques presented in this paper. We describe the original HadoopDB research in more detail in the following subsection.

2.2 HadoopDB

In this section we overview the architecture and relevant query execution strategies implemented in the HadoopDB [9, 10] project.

[1]In fact, the most recent version (0.7.0) introduced some of the missing features. Unfortunaly, it was released after we completed our experiments.

2.2.1 HadoopDB Architecture

The central idea behind HadoopDB is to create a single system by connecting multiple independent single-node databases deployed across a cluster (see our previous work [9] for more details). Figure 1 presents the architecture of the system. Queries are parallelized using Hadoop, which serves as a coordination layer. To achieve high efficiency, performance sensitive parts of query processing are pushed into underlying database systems. HadoopDB thus resembles a shared-nothing parallel database where Hadoop provides runtime scheduling and job management that ensures scalability up to thousands of nodes.

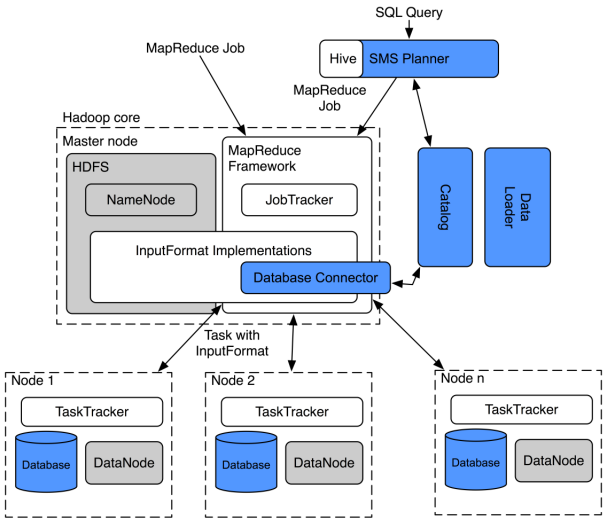

Figure 1: The HadoopDB Architecture

The main components of HadoopDB include:

1. *Database Connector* that allows Hadoop jobs to access multiple database systems by executing SQL queries via a JDBC interface.

2. *Data Loader* that hash-partitions and splits data into smaller chunks and coordinates their parallel load into the database systems.

3. *Catalog* which contains both metadata about the location of database chunks stored in the cluster and statistics about the data.

4. *Query Interface* which allows queries to be submitted via a MapReduce API or SQL.

In the original HadoopDB paper [9], the prototype was built using PostgreSQL as the underlying DBMS layer. By design, HadoopDB may leverage any JDBC-compliant database system. Our solution is able to transform a single-node DBMS into a highly scalable parallel data analytics platform that can handle very large datasets and provide automatic fault tolerance and load balancing. In this paper, we demonstrate our flexibility by integrating with a new columnar database engine described in the following section.

2.2.2 VectorWise/X100 Database

We used an early version of the VectorWise (VW) engine [7], a single-node DBMS based on the MonetDB/X100 research project [13, 35]. VW provides high performance in analytical queries due to vectorized operations on in-cache data and efficient I/O.

The unique feature of the VW/X100 database engine is its ability to take advantage of modern CPU capabilities such as SIMD instructions. This allows a data processing operation such as a predicate evaluation to be applied to several values from a column simultaneously on a single processor. Furthermore, in contrast to the tuple-at-a-time iterators traditionally employed by database systems, X100 processes multiple values (typically vectors of length 1024) at once. Moreover, VW makes an effort to keep the processed vectors in cache to reduce unnecessary RAM access.

In the storage layer, VectorWise is a flexible column-store that allows for finer-grained I/O, enabling the system to spend time reading only those attributes which are relevant to a particular query. To further reduce I/O, automatic lightweight compression is applied. Finally, clustering indices and the exploitation of data correlations through sparse MinMax indices allow even more savings in disk access.

2.2.3 HadoopDB Query Execution

The basic strategy of implementing queries in HadoopDB involves pushing those parts of query processing that can be performed independently into single-node database systems by issuing SQL statements. This approach is effective for selection, projection, and partial aggregation — processing that Hadoop typically performs during the Map and Combine phases. Employing a database system for these operations generally results in higher performance because a DBMS provides more efficient operator implementation, better I/O handling, and clustering/indexing.

Moreover, when tables are co-partitioned (e.g., hash partitioned on the join attribute), join operations can also be processed inside the database system. The benefit here is twofold. First, joins become local operarations which eliminates the necessity of sending data over the network. Second, joins are performed inside the DBMS which typically implements these operations very efficiently.

The initial release of HadoopDB included the implementation of Hadoop's InputFormat interface, which allowed, in a given job, accessing either a single table or a group of co-partitioned tables. In other words, HadoopDB's Database Connector supported only streams of tuples with an identical schema. In this paper, however, we discuss more advanced execution plans where some joins require data redistribution before computing and therefore cannot be performed entirely within single-node database systems. To accomodate such plans, we extended the Database Connector to give Hadoop access to multiple database tables within the Map phase of a single job. After repartitioning on the join key, related records are sent to the Reduce phase in which the actual join is computed.

Furthermore, in order to handle even more complicated queries that include multi-stage jobs, we enabled HadoopDB to consume records from a combined input consisting of data from both database tables and HDFS files. In addition, we enhanced HadoopDB so that, at any point during process-

ing, jobs can issue additional SQL queries via an extension we call SideDB (a "database task done on the side").

Apart from the SideDB extention, all query execution in HadoopDB beyond the Map phase is carried out inside the Hadoop framework. To achieve high performance along the entire execution path, further optimizations are necessary. These are described in detail in the next section.

3. SPLIT QUERY EXECUTION

In this section we discuss four techniques that optimize the execution of data warehouse queries across Hadoop and single-node database systems installed on every node in a shared-nothing network. We further discuss implementation details within HadoopDB.

3.1 Referential Partitioning

Distributed joins are expensive, especially in Hadoop, because they require one extra MR job [30, 34, 9] to repartition data on a join key. In general, database system developers spend a lot of time optimizing the performance of joins which are very common and costly operations. Typically, joins computed within a database system will involve far fewer reads and writes to disk than joins computed across multiple MapReduce jobs inside Hadoop. Hence, for performance reasons, HadoopDB strongly prefers to compute joins completely inside the database engine deployed on each node.

To be performed completely inside the database layer in HadoopDB, a join must be *local* i.e. each node must join data from tables stored locally without shipping any data over the network. When data needs to be sent across a cluster, Hadoop takes over query processing, which means that the join is not done inside the database engines. If two tables are hash partitioned on the join attribute (e.g., both employee and department tables on department_id), then a local join is possible since each single-node database system can compute a join on its partition of data without considering partitions stored on other nodes.

As a rule, traditional parallel database systems prefer local joins over repartitioned joins since the former are less expensive. This discrepancy in cost between local and repartitioned joins is even greater in HadoopDB due to the performance difference in join implementation between DBMS and Hadoop. For this reason, HadoopDB is willing to sacrifice certain performance benefits, such as quick load time, in exchange for local joins.

In order to push as many joins as possible into single node database systems inside HadoopDB, we perform "aggressive" hash-partitioning. Typically, database tables are hash-partitioned on an attribute selected from a given table. This method, however, limits the degree of co-partitioning, since tables can be related to each other via many steps of foreign-key/primary-key references. For example, in TPC-H, the lineitem table contains a foreign-key to the orders table via the order key attribute, while the orders table contains a foreign-key to the customer table via the customer key attribute. If the lineitem table could be partitioned by the customer who made the order, then any of the straightforward join combinations of the customer, orders, and lineitem tables would be local to each node.

Yet, since the lineitem table does not contain the customer key attribute, direct partitioning is impossible. HadoopDB was, therefore, extended to support referential partitioning. Although a similarly named technique was recently made available in Oracle 11g [23], it served a different purpose than in our project where this partitioning scheme facilitates joins across a shared-nothing network.

Obviously, this method can be extended to an arbitrary number of tables referenced in a cascading way. During data load, referential partitioning involves the additional step of joining with a parent table to retrieve its foreign key. This, however, is a one time cost that gets amortized quickly by superior performance on join queries. This technique benefits TPC-H queries 3, 5, 7, 8, 10, and 18, all of which need joins between the customer, orders, and lineitem tables.

3.2 Split MR/DB Joins

For tables that are not co-partitioned the join is generally performed using the MapReduce framework. This usually takes place in the Reduce phase of a job. The Map phase reads each table partition and, for each tuple, outputs the join attribute intended to automatically repartition the tables between the Map and Reduce phases.

Therefore, the same Reduce task is responsible for processing all tuples with the same join key. Natural joins and equi-joins require no further network communication — the Reduce tasks simply perform the join on their partition of data.

The above algorithm works similarly to a partitioned parallel join described in parallel database literature [28, 20]. In general this method requires repartitioning *both* tables across nodes. In several specific cases, however, the latter operation is unnecessary — a situation that parallel DBMS implementations take advantage of whenever possible. Two common join optimizations are the directed join and the broadcast join. The former is applicable when one of the tables is already partitioned by the join key. In this case only the other table has to be distributed using the same partitioning function. The join can proceed locally on each node. The broadcast join is used when one table is much larger than the other. The large table should be left in its original location while the *entire* small table ought to be shipped to every node in the cluster. Each partition of the larger table can then be joined locally with the smaller table.

Unfortunately, implementing directed and broadcast joins in Hadoop requires computing the join in the Map phase. This is not a trivial task[2] since reading multiple data sets with an algorithm that might require multiple passes does not fit well into the Map sequential scan model. Furthermore, HDFS does not promise to keep different datasets co-partitioned between jobs. Therefore, a Map task cannot assume that two different datasets partitioned using the same hash function are actually stored on the same node.

For this reason, previous work on adding specialized joins to the MapReduce framework typically focused on the relatively simple broadcast join. This algorithm is implemented in Hive, Pig, and a recent research paper [12][3]. Since none of the abovementioned systems implement cost-based query optimizers, a hint must be included in the query to let the system know that a broadcast join algorithm should be used.

[2]Unless both tables are already sorted by the join key, in which case one can use Hadoop's merge join operator.

[3]This work goes quite a bit farther than Hive and Pig, implementing several optimizations on top of the basic broadcast join, though each optimization maintains the single-pass sequential scan requirement of the larger table during the Map phase.

The implementation of the broadcast join in these systems is as follows. Each Map worker reads the smaller table from HDFS and stores it in an in-memory hash table. This has the effect of replicating the small table to each local node. A sequential scan of the larger table follows. As in a standard simple hash-join, the in-memory hash map is probed with each tuple of this larger table to check for a matching key value. The reading of both tables helps avoid the difficulties of implementing a multi-pass algorithm. Since the join is computed in the Map phase, it is called a *Map-side join*.

Split execution environments enable the implementation of a variety of joins in the Map phase and reveal some interesting new tradeoffs. First, take the case of the broadcast join. There are two ways that the latter can be implemented in a split execution framework. The first way is to use the standard Map-side join discussed above. The second way, possible only in HadoopDB, involves writing the smaller table to a temporary table in the database system on each node. Then the join is computed completely inside the DBMS and the resulting tuples are read by the Map tasks for further processing.

The significance of the tradeoff between these two approaches depends on the DBMS software used. A particularly important factor is the cost of writing to a temporary table and sharing this table across multiple partitions on the same node. In general, as long as this cost is not too high, computing the join inside the DBMS will yield better performance than computing it in the Java code of the Map task. This is explored further in Section 4.

Another type of join enabled by split execution environments is the directed join. Here, HadoopDB runs a standard MapReduce job to repartition the second table. First we look up in the HadoopDB catalog how the first table was distributed and use this function to repartition the second table. Any selection operations on the second table are performed in the Map phase of this job. The OutputFormat feature of Hadoop is then used to circumvent HDFS and write the output of this repartitioning directly into the database systems located on each node. HadoopDB provides native support for keeping data co-partitioned between jobs. Therefore, once both tables are partitioned on the same attribute inside the HadoopDB storage layer, the next MapReduce job can compute the join by pushing it entirely into the database systems. The resulting tuples get fed to the Map phase as a single stream.

In the experimental results presented later in this paper, we will further explore the performance of split MR/DB joins. This technique proved to be particularly beneficial in TPC-H queries 11, 16, and 17.

3.2.1 Split MR/DB Semijoin

A semijoin is one more type of join that can be split into two MapReduce jobs, the second of which computes the join in the Map phase. Here, not only does the first MapReduce job perform selection operations on the table, but it also projects the join attribute. The resulting column is then replicated as in a Map-side join. If the projected column is very small (for example, the key from a dictionary table or a table after applying a very selective predicate), the Map-side join is replaced with a selection predicate using the SQL clause 'foreignKey IN (listOfValues)' and pushed into the DBMS. This allows the join to be performed inside the database system without first loading the data into a temporary table inside the DBMS.

Furthermore, in some cases, HadoopDB's SideDB extension can be used to entirely eliminate the first MapReduce job for a split semijoin. At job setup a SideDB query extractes the projected join key column instead of running a seperate MapReduce job.

The SideDB extension is also helpful for looking up and extracting attributes from small tables such as dictionary tables. Such a situation typically occurs at the very end of the query plan, right before outputting the results. For example, integer identifiers that were carried through the query execution, are replaced by actual text values (e.g., names of the nations replacing the nation identifier in TPC-H). A similar concept in column-store databases is known as late materialization [8, 26].

The query rewrite version of the map-side split semijoin technique is commonly used in HadoopDB's implementation of TPC-H to satisfy the benchmark rules forbidding the replication of tables. All queries that include joins with region and nation tables are implemented using the selection-predicate-rewriting and SideDB optimizations.

3.3 Post-join Aggregation

In HadoopDB, since aggregation operations can be executed in database engines, there is usually no need for a MapReduce Combiner.

Still there exists no standard way of performing post-Reduce aggregation. While Reduce is meant for aggregation by design, it can only be applied if the repartitioning between the Map and Reduce phases is performed on the grouping attribute(s) specified in the query. If, however, the partitioning is done on a join key (in order to join two different tables), then another partitioning is needed to compute the aggregation, since, in general, the grouping attribute is different from the join key. The new partitioning therefore requires another MapReduce job and all its associated overhead.

In such situations, hash-based partial aggregation is done at the end of each Reduce task. The grouping attribute extracted from each result of the Reduce task is used to probe a hash table in order to update the appropriate running aggregation. This procedure can save significant I/O, since the output of Reduce tasks is written redundantly to HDFS whereas the output of Map tasks is written only locally. Hence, by outputting partially aggregated data instead of raw values, we reduce the amount of data to be written to HDFS. TPC-H queries that benefit from this technique include 5, 7, 8, and 9.

A similar technique is applied to TOP N selections, where the list of the top N entries is maintained in an in-memory tree map throughout the Reduce phase and outputted at the end. In-memory data structures are also used for combining an ORDER BY clause with another operator inside the same Reduce task, again saving an extra MapReduce job. Examples where this technique is beneficial are TPC-H queries 2, 3, 10, 13, and 18.

3.4 Pre-join Aggregation

Whereas in most database systems aggregations are typically performed after a join, in HadoopDB they sometimes get transformed into partial aggregation operators and computed before a join. This happens when the join cannot be

pushed into the database system and therefore must be performed by Hadoop which is much slower than DBMS. When the product of the cardinalities of the group-by and join-key columns is smaller than the cardinality of the entire table, it becomes beneficial to push the aggregation past the join so that it can be performed inside the database layer. Later, there might be a need to drop some of the computed aggregates for which the join condition is not satisfied. This extra work is rewarded, however, with savings in I/O and network traffic.

3.5 Constructing a query plan in HadoopDB

Figure 2 illustrates the split execution plan HadoopDB executes for TPC-H Query 20. The original SQL statement

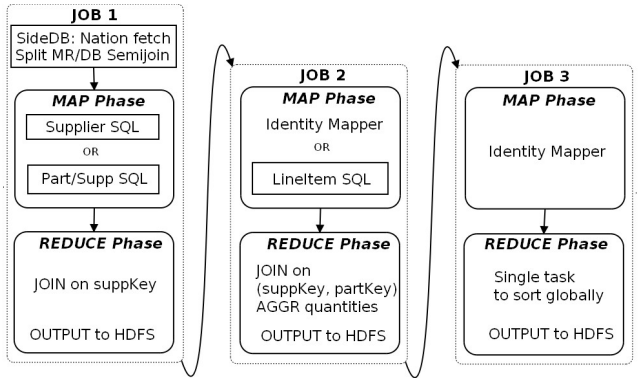

Figure 2: Query 20 Execution Plan

(see Appendix) is decomposed into four simpler independent SQL statements based on the partitioning information.

In the first stage (Job 1), HadoopDB begins by fetching the nation dictionary using the SideDB extension. The value of the nation key for Canada is used to apply the split semijoin (via query rewrite) to the supplier table processed by the first group of map tasks. The second group executes the SQL statement against the part and partsupp tables which are co-partitioned on the partkey. To compute a join in Reduce both types of map tasks output the resulting records with supplier key as the key and the remaining attributes as the value.

In the second stage (Job 2), one group of Map tasks reads the result of Job 1 and applies a simple identity function while the other group reads lineitem records filtered by a predicate on ship-date. Thanks to repartitioning on a composite key (suppkey and partkey) between Map and Reduce, the join between two incoming streams of tuples is achieved in the same job as the post-join aggregation and the application of a predicate on the quantity threshold. The last stage (Job 3) sorts the final result by supplier name within a single reducer.

4. EXPERIMENTS

Using a TPC-H benchmark with a scaling factor 3000 (3TB of data) on a 45-node cluster, we evaluated three data processing systems: HadoopDB with PostgreSQL (HDB-PSQL) and VectorWise (HDB-VW), DBMS-X (a commercial parallel row-oriented database system), and Hive with HDFS. The details of the configuration and data load process are presented in subsequent sections.

4.1 Cluster configuration

Each node in the cluster has a single 2.40 GHz Intel Core 2 Duo processor running 64-bit Red Hat Enterprise Linux 5 (kernel version 2.6.18) with 4GB RAM and two 250GB SATA-I hard disks. According to hdparm, the hard disks deliver 74MB/sec for buffered reads. All nodes are on the same rack, connected via 1Gbps network to a Cisco Catalyst 3750E-48TD switch.

4.2 Benchmarked Systems

4.2.1 DBMS-X

We installed a recent release of DBMS-X, a parallel row-oriented SQL DBMS from a major relational database company. The official TPC-H benchmark conducted by the DBMS-X vendor used the same version of the system. Consequently, in our installation of DBMS-X we followed, as far as possible, the parameters specified in the report published at the TPC website. Since the vendor ran the benchmark with considerably more RAM and hard drives per node than in our cluster, to reflect our resources we had to scale down the values of some parameters. The system is installed on each node and configured to use 4GB shared memory segments for the buffer pool and other temporary space. Furthermore, because our entire benchmark is read-only, we did not enable the replication features in DBMS-X, since rather than improving performance this would have complicated the installation process.

4.2.2 Hive and Hadoop

For experiments in this paper, we used Hive version 0.4.1 and Hadoop version 0.19.2, running on Java 1.6.0. We configured both systems according to the suggestions offered by members of Hive's development team in their report on running TPC-H on Hive [5]. To reflect our hardware capacity, we adjusted the number of map and reduce slots to 2. In addition, the HDFS block size was set to 256MB. We also enabled compression of query intermediate data with the LZO native library version 2.03.

4.2.3 HadoopDB

The Hadoop part of HadoopDB was configured similarly to Hadoop for Hive. The only difference is the number of task slots, which we set to one. Thus, on each worker node, Hadoop processes were able to use up to 2GB of RAM. The other half of memory was designated to the DBMS, which was installed on each machine independently. We used PostgreSQL version 8.4.4 and increased its memory settings: the shared buffers to 512 MB and the working memory size to 1GB. The remaining 512MB served as disk cache managed by OS. In the case of VectorWise, the buffer pool was set to 400MB and the rest was available for query processing. All other parameters of database servers remained unchanged.

4.3 TPC-H Benchmark

TPC-H [6] is a decision support benchmark that consists of a set of complex business analysis queries. The dataset models a global distribution company and includes the following tables: nation, region, supplier, part, partsupp, customer, orders, and lineitem. We ran the benchmark at scaling factor SF = 3000 (about 3TB).

4.4 Data Preparation and Loading

The benchmark data were generated using the dbgen program provided by TPC, running in parallel on every node. We used the appropriate parameters to produce a consistent dataset across the cluster. Each of the 45 nodes in our cluster received about 76GB of raw data.

4.4.1 DBMS-X

We followed the DBMS-X vendor suggestions and used the DDL scripts from their TPC-H report to create the tables and indices, and to define data distribution. All tables were globally hash-partitioned across the nodes on their primary key, except for the partsupp and lineitem relations, which were hash-partitioned on only the first of the two columns that make up their primary key. The supplier and customer relations are indexed on their respective nation keys, and the nation table was indexed on its region column. Finally, on each node of the cluster, DBMS-X organized the lineitem and orders relations by the month of their date columns for a partial ordering by date. The optimizer of DBMS-X is aware of the partial ordering and with the appropriate predicates can eliminate portions of the table from consideration.

The loading process consists of two steps. First, data are repartitioned and shuffled; second, the repartitioned data are bulk-loaded on each node. The DBMS-X loading utility, which we invoked on each node, can directly consume and transform data produced by the TPC-H data generator. The partitioning phase can proceed in parallel, but DBMS-X serializes each load phase and does not make full use of the available disk bandwidth. DBMS-X does not reliably indicate the time spent in the two phases, so we report only total load time which was 33h3min.

4.4.2 Hive and Hadoop

Hadoop's filesystem utility was run in parallel on all nodes and copied unaltered data files into HDFS under a separate directory for each table. Each file was automatically broken into 256MB blocks and stored on a local DataNode. In addition, we executed Hive DDL scripts to put relational mapping on the files. Thanks to its simplicity, the entire process took only 49 minutes.

4.4.3 HadoopDB

In the first step, HadoopDB also loaded raw data into HDFS. Then, HadoopDB Data Loader utilities, implemented as MR jobs, performed global hash-partitioning of each data file across the cluster. In the case of the lineitem table, this two-step process involved a join with the orders table to retrieve the customer key attribute needed for referential partitioning. Next, each node downloaded its partitions into a local filesystem. Finally, each group of co-partitioned tables was broken into smaller chunks, which observe referential integrity constraints with the maximum size of 3.5GB. The entire partitioning process took 11h4min. Referential hash-partitioning was the most expensive part (6h42min).

The chunked files were bulk-loaded in parallel into each instance of the VectorWise server using the standard SQL COPY command. During this process data were also sorted according to the clustering index and VW's internal indices were created. In the last step, the VW optimizedb tool was run to generate statistics and histograms to be used by the optimizer during query execution. Loading data into the databases took 3h47min.

The data layout for HadoopDB with VectorWise (HDB-VW) is as follows. The customer, orders, and lineitem tables were partitioned by the customer key and clustered by the nation key, order date, and order key, respectively. The part key attribute was used both to hash-partition and to cluster the part and partsupp tables. The supplier table was partitioned by its primary key and clustered on the nation key. Small dictionary tables, region and nation, were not partitioned and located on a single node. Despite their small size, they were not replicated since this violates TPC-H benchmarking rules. Their clustering indices were created using the region key attribute. We followed the advice of the VectorWise team on the most beneficial indices for their database system.

In short, the HadoopDB data layout was identical to the DBMS-X setup, except for the use of referential partitioning, slightly different indices, and chunking data into smaller partitions per node[4].

In HadoopDB combined with PostgreSQL (HDB-PSQL) data are partitioned in the same way as in the setup with VW. We clustered the lineitem table on ship date, however, because we found this index more beneficial for PostgreSQL. Loading and indexing the databases took 7h13min.

4.5 Benchmark Execution

We ran TPC-H queries from 1 to 20. For DBMS-X and Hive we executed the statements as suggested by the vendors. We noted that since the HiveQL syntax is a subset of SQL, in many cases the original TPC-H queries were rewritten by the Hive team into a series of simpler statements that produce the desired output in the last step. HadoopDB implemented the queries using its API to ensure the employment of the execution strategies we discuss in this paper. All queries were parametrized using substitution values specified by TPC-H for result validation.

Despite trying multiple configuration settings we could not get Q10 running on Hive because it repeatedly crashed during the join operation due to an "out of memory" error. We managed, however, to run every other query in that system. When adjusted to the larger dataset and weaker hardware, our results were in line with the numbers published by the Hive team.

4.6 General Comparison

The results of benchmarking all three systems are shown in Figure 3 while the table below presents the numbers (all times are in seconds).

First, it is worth noting that DBMS-X significantly outperforms Hive in all queries. This is not surprising, since the Hive development team found a similar difference when comparing their system with DBMS-X [5]. The main reason for Hive's inferior performance is the lack of partitioning and indexing. As a result of this limitation, every selection becomes a full data scan and most of the joins involve repartitioning and shuffling all records across the cluster.

Previous work showed that HadoopDB combined with PostgreSQL was able to approach but not quite reach the performance level of parallel databases [9]. As a result of

[4]Chunking slows down the performance slightly, but is necessary to maintain HadoopDB's fault tolerance guarantees [9].

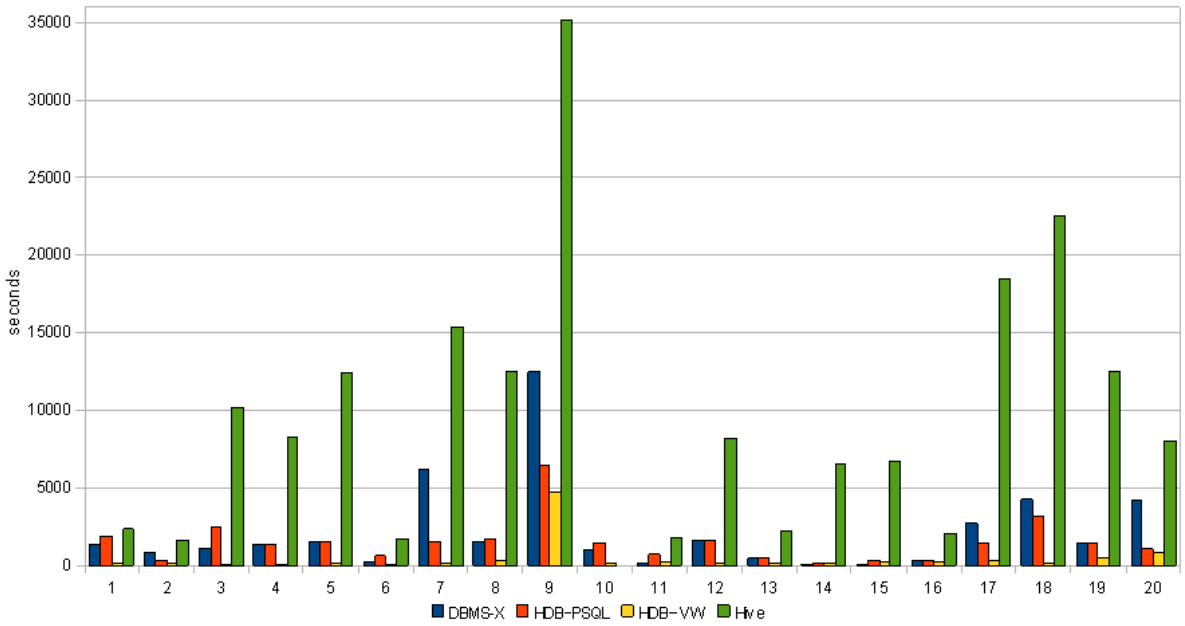

Figure 3: TPC-H Query Performance (SF = 3000)

the techniques we described in Section 3, the new version of HadoopDB (also with PostgreSQL) is able not only to match, but in some cases to significantly outperform the parallel database. The query execution enhancements that led to this improvement will be examined in detail in the subsequent sections.

Q	DBMS-X	HDB-PSQL	HDB-VW	HIVE
1	1367	1921	171	2323
2	822	358	118	1599
3	1116	2443	106	10219
4	1387	1383	92	8240
5	1515	1520	209	12352
6	224	601	102	1702
7	6133	1504	207	15398
8	1564	1739	357	12451
9	12463	6436	4685	35145
10	1022	1424	134	—
11	205	715	263	1780
12	1613	1606	139	8200
13	453	428	199	2147
14	73	198	178	6550
15	98	337	235	6743
16	364	385	220	2092
17	2746	1426	327	18493
18	4288	3130	149	22530
19	1423	1397	482	12491
20	4154	1100	841	7972

It is interesting to note that the biggest bottleneck in HadoopDB is the underlying database. Switching from a previous-generation row-store to a highly optimized column-store resulted in a considerable performance improvement for HadoopDB (approximately a factor of seven on average). This achievement highlights the benefits of the plug-and-play design which allows the use of different database systems. As a result, HadoopDB can improve at the same rate as the research on the performance of analytical database systems.

For the class of low-latency queries (such as Q11, Q14 and Q15) HadoopDB is not bottlenecked by the underlying database system. The real problem for these queries is the block-level scheduling overhead since Hadoop-based systems are optimized for batch-oriented processing rather than realtime analytics. The Hadoop community is working on eliminating some of these limitations for low latency queries, and we expect improvements in this area in the near future.

Overall, HadoopDB (with VW) outperforms DBMS-X by a factor of 7.8 on average and Hive by a factor of 42 on average. When PostgreSQL is used, HadoopDB matches the performance of the commercial DBMS.

4.7 Split Execution Techniques Breakdown

In this section, we choose several representative queries to explore in more detail the performance of the split query execution techniques described in Section 3. The SQL statements for those queries are listed in the Appendix.

4.7.1 Query 5

Query 5 requires joining six tables: customer, orders, lineitem, supplier, nation, and region. The fully optimized version makes use of most of the split query execution techniques discussed in this paper, including referential partitioning, split semijoin, and post-join aggregation. It involves only one repartitioned join (the join with the supplier table). The implementation of this query consists of two MR jobs. The first one performs the join and partial aggregation, while the second one computes the global sum of revenue per nation.

Figure 4 shows the query running time with each optimization turned off one by one. The first optimization we turn off is the post-join aggregation on the nation key. We observe that the amount of data that needs to be written

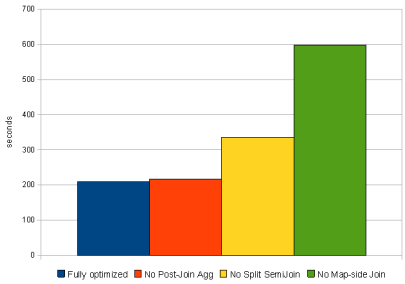

Figure 4: Q5 Breakdown for VW

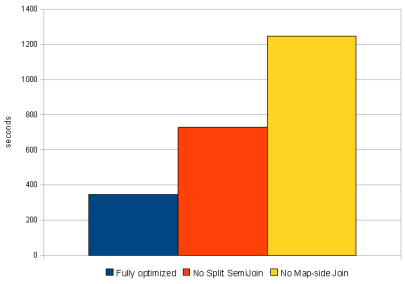

Figure 5: Q8 Breakdown for VW

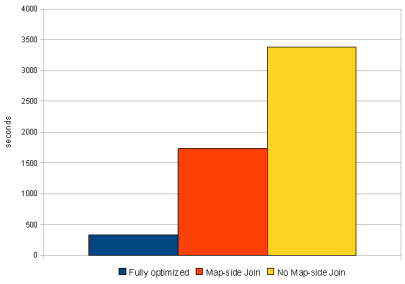

Figure 6: Q17 Breakdown for VW

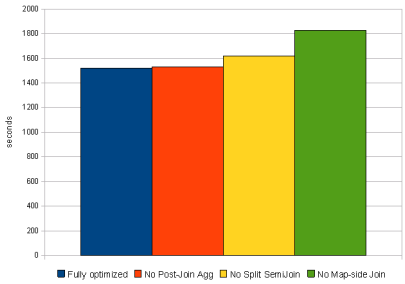

Figure 7: Q5 Breakdown for PSQL

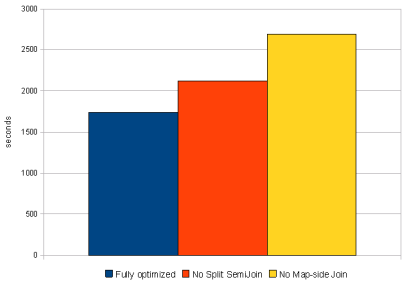

Figure 8: Q8 Breakdown for PSQL

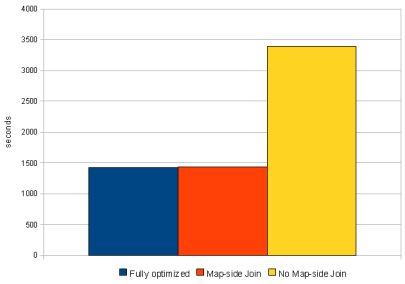

Figure 9: Q17 Breakdown for PSQL

to HDFS between the two jobs increases from 5.2KB to 83.8MB. In an isolated experiment like ours, the effect on query running time is quite insignificant. The extra I/O becomes more of a problem in production settings where there is far more competition for I/O from many concurrent jobs.

The next optimization we turn off is the split semijoin technique (used for the joins of both the customer and supplier tables with the nation and region tables). HadoopDB replaces the split semijoin with a regular Map-side join. This results in a slowdown of about 50% . The reason is that now the joins are performed outside of the database system and therefore cannot take advantage of clustering indices on the nation key.

Finally, we turn off the Map-side join and replace it with the join done in the Reduce phase. This causes query running time to double. Overall, we reach almost a factor of three slowdown versus the fully optimized version. It is worth noting that the entire operation of joining all the tables is still achieved within one MR job, thanks to the fact that the dictionary tables can be brought into memory using the SideDB extension.

Turning off optimization techniques has a relatively smaller effect on HDB-PSQL (Fig. 7) since the total running time is dominated by the slower performing PostgreSQL.

To explore the impact of referential partitioning, we implemented in HadoopDB an alternative query execution plan which assumes only standard partitioning as used in DBMS-X. In this plan, before the join with the supplier table is performed, one extra MR job is required to compute the join between the customer and orders/lineitem tables. This extra join causes the total query performance to become slower by a factor of four versus the layout with referential partitioning. The reason is that, internally, that extra job needs to send about 47GB of intermediate data over the network and write an additional 12GB to HDFS.

4.7.2 Query 8

We observe that, like in Query 5, moving the join operation to a later stage within a MapReduce job decreases performance. The results are illustrated in Figure 5. Here, employing split semijoin (to restrict nations to the region of America) gives a speedup by a factor of 2 over the Map-side join and by a factor of 3.6 over the Reduce join. When the less efficient PostgreSQL is used underneath HadoopDB, the contribution of optimization techniques (Fig. 8) is again smaller.

More specifically, switching to the Map-side join resulted in a total of 5.5M of rows returned by all the databases combined (5 times more than in the split semijoin version). In both the split semijoin and the regular Map-side join cases, the same amount of intermediate data (around 315GB) is written to disk by the Hadoop framework between the Map and Reduce phases. The amount increases to 1.7TB when we perform a repartitioned join in the Reduce phase.

4.7.3 Query 17

Query 17 involves a join between the lineitem and part tables which are not copartitioned. This query would normally involve repartitioning both tables and performing the join in Reduce. In Q17, however, very selective predicates are applied to the part table (69GB of raw data), resulting in only about 6MB of data (around 600 thousand integer identifiers). This small size gives HadoopDB the opportunity to employ the split MR/DB join technique. Note that this is not the semijoin version as in the previous two queries, since there are too many values to perform the "foreignKey IN (listOfValues)" rewrite. Instead, the result of the selection query on the part table is broadcasted to all nodes, and loaded into the database servers where the join is computed. In this way, HadoopDB performs the join in the Map phase, thereby avoiding the repartitioning of the lineitem table. The gain is about a factor of 2.5 in total running time for HDB-PSQL and an order of magnitude for HDB-VW.

In order to compare the split MR/DB join technique with a standard Map-side join, we implemented the latter as an alternative execution plan. The results are shown in Figures 6 and 9. Surprisingly, the speedup resulting from pushing the join into the DBMS is greater for VectorWise than for PostgreSQL. We believe that this is due to the vectorized processing and cache-conscious query executor in VW/X100. PostgreSQL follows the tuple-at-a-time iterator model, which results in similar performance as that of the Java hashtable lookups in the Map-side join implementation. Hence, even if the broadcasted table can fit in memory, it may prove better to write the table to an optimized DBMS and perform the join there, than to perform it in the standard way in the Map phase.

4.8 Analysis of Other Interesting Queries

In this section we analyze several additional queries that are notable in some way, usually because they deviate from our expectations. This helps to further understand the different tradeoffs and performance characteristics of the systems.

4.8.1 Query 3

Query 3 is similar to Query 5 in that referential partitioning eliminates a significant amount of disk and network I/O. We expected both HadoopDB combined with VectorWise and HadoopDB combined with PostgreSQL to significantly outperform DBMS-X which does not have the benefit of referential partitioning. This is indeed the case for VW, but the results are quite different for PostgreSQL, as HDB-PSQL is 2 times slower than DBMS-X. A close investigation revealed that the date predicates are not very selective, returning half of the records from both the lineitem and orders tables. Those records need to be joined with about a fifth of the rows from the customer table. The PostgreSQL's EXPLAIN command indicates that the hash join algorithm is applied. Given the large number of records, PostgreSQL is not able to keep all the intermediate data in memory and therefore needs to swap to disk. In contrast, VectorWise employs an efficient merge join algorithm and is able to processes each chunk without spilling intermediate results to disk.

4.8.2 Query 9

This query poses difficulties for each system we benchmarked. Six tables need to be joined and there is only one selection predicate (on the part table) to reduce the size of the data read from disk. Thus, the query requires shuffling most of the data over the network to compute joins. Both HadoopDB setups benefit from pushing three out of five joins into the local databases thanks to the partitioning of data. VectorWise outperforms PostgreSQL because the former is a column-store and in total only 15 out of 46 columns need to be read off disk. The column-store advantages are somewhat dimished here, however, due to the high cost of tuple reconstruction caused by the large number of returned rows.

4.8.3 Query 18

In this query HadoopDB with PostgreSQL is about 37% faster than the parallel database and 7 times more efficient than Hive. HDB-VW outperforms DBMS-X by a factor of 28.8 and Hive by a factor of 151. HadoopDB's highly efficient execution plan for this query again benefits greatly from referential partitioning, requiring only one MR job to produce the desired output. DBMS-X needs to perform one non-local join (with customer table) but is still able to compute the complex subquery in the WHERE clause thanks to partitioning on the order key. Hive first runs an extra job for that subquery and then joins all tables using the repartitioned join algorithm.

The relatively high difference in performance between the VW/X100 and PostgreSQL setups deserves closer investigation. It turns out that PostgreSQL spends 75% of the time computing the subquery responsible for the aggregation of the lineitem table via sorting by order key. Due to insufficient memory an external disk merge sort is needed. In contrast, VW, besides being far more computationally efficient in general, also takes advantage of the fact that the lineitem table is already clustered on the order key attribute.

5. CONCLUSION

When the query execution environment is split across MapReduce and database systems, there appear some interesting tradeoffs not present in each system separately. By carefully designing HadoopDB's query engine to operate in this split execution environment, we achieved a substantial performance improvement. After equipping HadoopDB with an efficient single-node columnar DBMS, along with several new query execution strategies, we outperformed a popular commercial parallel database system by almost an order of magnitude, and the state of the art Hadoop-based data warehouse by significantly more than an order of magnitude. By integrating with Hadoop for job scheduling and network communication, we transformed a single-node DBMS into a scalable parallel data analytics platform. Our system successfully competes in the most commonly used data warehouse benchmark against established commercial solutions.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant IIS-0844480.

7. REFERENCES

[1] Hadapt Inc. Web page. http://www.hadapt.com .
[2] Hadoop. Web page. http://hadoop.apache.org .
[3] Hadoop TeraSort. http://developer.yahoo.com/blogs/hadoop/Yahoo2009.pdf .
[4] Hive. Web page. http://hadoop.apache.org/hive .
[5] Running TPC-H queries on Hive. Web page. http://issues.apache.org/jira/browse/HIVE-600 .
[6] TPC-H. Web page. http://www.tpc.org/tpch .
[7] VectorWise. Web page. http://www.vectorwise.com .
[8] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475, Istanbul, Turkey, 2007.
[9] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB*, 2009.
[10] A. Abouzied, K. Bajda-Pawlikowski, J. Huang, D. J. Abadi, and A. Silberschatz. Hadoopdb in action: Building real world applications. Demonstration. SIGMOD, 2010.
[11] E. Albanese. Why Europe's Largest Ad Targeting Platform Uses Hadoop. http://www.cloudera.com/blog/2010/03/why-europes-largest-ad-targeting-platform-uses-hadoop .
[12] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log

processing in MapReduce. In *Proc. of SIGMOD*, pages 975–986, New York, NY, USA, 2010. ACM.

[13] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[14] S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. In *Proc. of VLDB*, 2010.

[15] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[16] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: new analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[17] G. Czajkowski. Sorting 1PB with MapReduce. googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html .

[18] G. Czajkowski, G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, and N. Leiser. Pregel: A system for large-scale graph processing. In *Proc. of SIGMOD*, 2010.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[20] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[21] D. DeWitt and M. Stonebraker. MapReduce: A major step backwards. DatabaseColumn Blog. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards .

[22] J. Dittrich, J.-A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). In *Proc. of VLDB*, 2010.

[23] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in Oracle. In *Proc. of SIGMOD*, pages 1111–1122, 2008.

[24] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[25] Hadoop. Poweredby. http://wiki.apache.org/hadoop/PoweredBy .

[26] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proc. of SIGMOD*, pages 297–308, 2009.

[27] C. Monash. Cloudera presents the MapReduce bull case. DBMS2 Blog. dbms2.com/2009/04/15/cloudera-presents-the-mapreduce-bull-case .

[28] T. Nakayama, M. Hirakawa, and T. Ichikawa. Architecture and Algorithm for Parallel Execution of a Join Operation. In *Proc. of ICDE*, pages 160–166, 1984.

[29] B. Panda, J. Herbach, S. Basu, and R. Bayard. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. *PVLDB*, 2(2):1426–1437, 2009.

[30] 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.

[31] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murth. Hive Ű A Petabyte Scale Data Warehouse Using Hadoop. In *Proc. of ICDE*, 2010.

[32] R. Vernica, M. Carey, and C. Li. Efficient Parallel Set-Similarity Joins Using MapReduce. In *Proc. of SIGMOD*, 2010.

[33] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE '10*, 2010.

[34] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proc. of SIGMOD*, pages 1029–1040, 2007.

[35] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, 2009.

APPENDIX

TPC-H Query 3

```
select first 10
  l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate, o_shippriority
from
  customer, orders, lineitem
where
  c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '1995-03-15'
  and l_shipdate > date '1995-03-15'
group by
  l_orderkey, o_orderdate, o_shippriority
order by
  revenue desc, o_orderdate
```

TPC-H Query 5

```
select
  n_name, sum(l_extendedprice * (1 - l_discount)) as revenue
from
  customer, orders, lineitem, supplier, nation, region
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'ASIA'
  and o_orderdate >= date '1994-01-01'
  and o_orderdate < date '1994-01-01'
    + interval '1' year
group by
  n_name
order by
  revenue desc
```

TPC-H Query 7

```
select
  supp_nation, cust_nation, l_year, sum(volume) as revenue
from
  (
    select
      n1.n_name as supp_nation, n2.n_name as cust_nation,
      extract(year from l_shipdate) as l_year,
      l_extendedprice * (1 - l_discount) as volume
    from
      supplier, lineitem, orders, customer, nation n1, nation n2
    where
      s_suppkey = l_suppkey
      and o_orderkey = l_orderkey
      and c_custkey = o_custkey
      and s_nationkey = n1.n_nationkey
      and c_nationkey = n2.n_nationkey
      and (
        (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
        or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE')
      )
      and l_shipdate between date '1995-01-01'
        and date '1996-12-31'
  ) as shipping
group by
  supp_nation, cust_nation, l_year
order by
  supp_nation, cust_nation, l_year
```

TPC-H Query 8

```
select
  o_year, sum(case when nation = 'BRAZIL' then volume
    else 0 end) / sum(volume) as mkt_share
from
  (
    select
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) as volume,
      n2.n_name as nation
    from
      part, supplier, lineitem, orders,
      customer, nation n1, nation n2, region
    where
      p_partkey = l_partkey
      and s_suppkey = l_suppkey
      and l_orderkey = o_orderkey
      and o_custkey = c_custkey
      and c_nationkey = n1.n_nationkey
      and n1.n_regionkey = r_regionkey
      and r_name = 'AMERICA'
      and s_nationkey = n2.n_nationkey
      and o_orderdate between date '1995-01-01'
        and date '1996-12-31'
      and p_type = 'ECONOMY ANODIZED STEEL'
  ) as all_nations
group by
  o_year
order by
  o_year
```

TPC-H Query 9

```
select
  nation, o_year, sum(amount) as sum_profit
from
  (
    select
```

```
      n_name as nation,
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount)
        - ps_supplycost * l_quantity as amount
    from
      part, supplier, lineitem, partsupp, orders, nation
    where
      s_suppkey = l_suppkey
      and ps_suppkey = l_suppkey
      and ps_partkey = l_partkey
      and p_partkey = l_partkey
      and o_orderkey = l_orderkey
      and s_nationkey = n_nationkey
      and p_name like '%green%'
  ) as profit
group by
  nation, o_year
order by
  nation, o_year desc
```

TPC-H Query 17

```
select
  sum(l_extendedprice) / 7.0 as avg_yearly
from
  lineitem, part
where
  p_partkey = l_partkey
  and p_brand = 'Brand#23'
  and p_container = 'MED BOX'
  and l_quantity < (
    select
      0.2 * avg(l_quantity)
    from
      lineitem
    where
      l_partkey = p_partkey
  )
```

TPC-H Query 18

```
select first 100
  c_name, c_custkey, o_orderkey,
  o_orderdate, o_totalprice, sum(l_quantity)
from
  customer, orders, lineitem
where
  o_orderkey in (
    select
      l_orderkey
    from
      lineitem
    group by
      l_orderkey
    having
      sum(l_quantity) > 300
  )
  and c_custkey = o_custkey
  and o_orderkey = l_orderkey
group by
  c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
order by
  o_totalprice desc, o_orderdate
```

TPC-H Query 20

```
select
  s_name, s_address
from
  supplier, nation
where
  s_suppkey in (
    select
      ps_suppkey
    from
      partsupp
    where
      ps_partkey in (
        select
          p_partkey
        from
          part
        where
          p_name like 'forest%'
      )
      and ps_availqty > (
        select
          0.5 * sum(l_quantity)
        from
          lineitem
        where
          l_partkey = ps_partkey
          and l_suppkey = ps_suppkey
          and l_shipdate >= date '1994-01-01'
          and l_shipdate < date '1994-01-01'
            + interval '1' year
      )
  )
  and s_nationkey = n_nationkey
  and n_name = 'CANADA'
order by
  s_name
```