Clydesdale: Structured Data Processing on MapReduce

Tim Kaldewey[†], Eugene J. Shekita[♯*], Sandeep Tata[†]

[†]*IBM Almaden Research Center*

[♯]*Google*

tkaldew@us.ibm.com, shekita@google.com, stata@us.ibm.com

ABSTRACT

MapReduce has emerged as a promising architecture for large scale data analytics on commodity clusters. The rapid adoption of Hive, a SQL-like data processing language on Hadoop (an open source implementation of MapReduce), shows the increasing importance of processing structured data on MapReduce platforms. MapReduce offers several attractive properties such as the use of low-cost hardware, fault-tolerance, scalability, and elasticity. However, these advantages have required a substantial performance sacrifice.

In this paper we introduce Clydesdale, a novel system for structured data processing on Hadoop – a popular implementation of MapReduce. We show that Clydesdale provides more than an order of magnitude in performance improvements compared to existing approaches without requiring *any* changes to the underlying platform. Clydesdale is aimed at workloads where the data fits a star schema. It draws on column oriented storage, tailored join-plans, and multi-core execution strategies and carefully fits them into the constraints of a typical MapReduce platform. Using the star schema benchmark, we show that Clydesdale is on average 38x faster than Hive. This demonstrates that MapReduce in general, and Hadoop in particular, is a far more compelling platform for structured data processing than previous results suggest.

1. INTRODUCTION

In recent years, there has been tremendous interest in using MapReduce as a large scale data processing platform. While parallel database management systems (DBMSs) were traditionally the platform of choice for large scale data processing, MapReduce has gained substantial momentum because of several attractive properties – the use of commodity low-cost hardware, fault-tolerance, elasticity, scalability, and a flexible programming model.

The initial interest in MapReduce was mainly for large scale analysis of text data such as web crawls and for constructing text indexes in parallel [19]. Enterprises typically start off using a MapReduce cluster for unstructured data such as application logs and document collections. The data often goes through an ETL-phase and relatively structured data is produced for reporting and analysis. We have learned from conversations with several enterprise customers that while they use a MapReduce cluster for several different workloads (including data mining and machine learning), structured data processing is becoming increasingly important. The emergence and increasing adoption of systems like Hive [3] is further indication of the growing need for structured data processing on this platform.

Recent work [34] points to several performance problems for structured data processing on MapReduce. Critics of the MapReduce paradigm have argued that many of these problems can be avoided by simply using a parallel DBMS for structured data processing tasks. However, given the flexibility of MapReduce, i.e. its ability to cater to a larger variety of applications, as well as growing investments in hardware and administration for this platform in many enterprises, replacing it with a parallel DBMS may neither be desirable nor practical.

Previous efforts have tried to combine the flexibility and scalability of MapReduce with the performance of relational DBMSs. However, they either involved radical changes to the MapReduce platform, i.e. discarding the distributed filesystem and running a DBMS instance on each node [7], or adding MapReduce-like fault-tolerance to an existing parallel DBMS [37]. Other approaches [32, 20] have advocated using indexing and storage organization techniques with limited performance gains.

This paper describes Clydesdale [1], a research prototype built on top of Hadoop. Hadoop is a popular open-source implementation of MapReduce. With Clydesdale, we show that dramatic performance improvements can be achieved for structured data processing without any changes to the underlying implementation. This is of significant practical value since it allows us to run Clydesdale on future versions of Hadoop without having to re-compile and re-test Hadoop with a set of custom changes. Such as design also allows Clydesdale to inherit the fault-tolerance, elasticity, and scalability properties of MapReduce. Using the star schema benchmark [33], we show that Clydesdale is 5x – 83x faster than Hive.

Clydesdale's design draws on several existing techniques from parallel DBMSs such as columnar storage, tailored join plans, and block iteration. However, adapting these techniques for the MapReduce environment is not straightforward, in particular when trying to preserve all the properties that make the platform attrac-

*Work done while author was at IBM Almaden Research Center

[1]Clydesdale is a robust and flexible breed of work horse in contrast to a racing thoroughbred, which is fast, but fragile and inflexible.

EDBT 2012, March 26–30, 2012, Berlin, Germany.

tive. The challenges arise from two important differences between a parallel DBMS and MapReduce: the presence of a distributed filesystem and the constraints of the task-scheduling infrastructure. Clydesdale uses the Hadoop distributed filesystem (HDFS) to store its data. To our knowledge, no commercial shared-nothing relational DBMS uses a distributed filesystem. Most commercial databases either rely on the storage tier (RAID controllers, SAN storage) or use hot standbys to manage replication and mask disk failures. Using a distributed filesystem that works on commodity nodes to manage replication requires a careful design of the storage organization layer. Clydesdale uses a novel column-oriented layout [21] that interacts with the replication strategy of HDFS to ensure that columns from a given row are co-located on a given node.

The task-scheduling infrastructure in Hadoop was designed to provide locality-aware scheduling [19] at the granularity of map and reduce *tasks*. Map and reduce tasks are expected to last a short while (few minutes) and a typical job may consist of several map and reduce tasks. In contrast, DBMS runtimes have traditionally used iterator-based operators [23] that are scheduled to run for the entire duration of the query on a given server. Clydesdale uses carefully designed map tasks so that data structures used in query processing can be shared across multiple threads and even multiple tasks consecutively executed on any node. This allows Clydesdale to ameliorate the per-task overheads that can adversely impact the performance of MapReduce jobs. Clydesdale also exploits current hardware trends such as servers with large memories and multiple cores and uses an appropriately tailored n-way join algorithm instead of repeated use of a generic two-way join.

Clydesdale is aimed at workloads where the data fits a star schema. Leaders of the database community have argued [36, 18] that an overwhelming majority of structured data repositories are either star or snowflake schemas. Star-schemas have stood the test of time as a good way to model large datasets in many industries, and we expect this will continue to hold true for modeling structured data on new platforms like Hadoop. While we focus on star schemas in this paper, Clydesdale can also be used for more general processing like Hive.

Clydesdale's central contribution is a demonstration that an unmodified instance of Hadoop can provide more than an order of magnitude in performance improvements for structured data processing. We describe the design and implementation of Clydesdale focusing on the join strategy and its execution in the context of Hadoop (Section 4). We describe the specific challenges posed by HDFS and MapReduce and how Clydesdale overcomes them to leverage columnar storage, multi-core execution, and block-iteration (Section 5). In Section 6, we demonstrate Clydesdale's performance on the star schema benchmark [33] and compare it with Hive. We show that Clydesdale is 5x – 83x faster than Hive, averaging an advantage of 38x. We describe several opportunities for future research on managing updates, multi-workload scheduling, and advanced storage organization (Section 8)

2. RELATED WORK

An initial comparison [34] of MapReduce and parallel DBMSs pointed out a substantial performance difference between the two platforms. Subsequent studies have demonstrated that indexing [20, 30], columnar storage [21], and other organization techniques [31] can be adapted to MapReduce. to reduce this performance gap. Other approaches [13, 8, 38] have discussed join processing on MapReduce. Yet others [25] have examined how to tune the various parameters in a MapReduce platform like Hadoop to provide the best performance for a given job. However, these efforts address general workloads and have not particularly focused on structured data or on star schemas. Consequently, the performance improvements they offer are not in the range that can be achieved by targeting specific workloads. Clydesdale's design can be viewed as an adaptation of existing techniques to Hadoop. Clydesdale draws from columnar storage [35] for I/O performance, join techniques [6] suitable for multi-core servers and Hadoop, and block-iteration [39] for CPU performance.

Recent efforts examining the suitability of MapReduce as a platform for structured data processing include Hive [3], Cheetah [17], HadoopDB [7], Llama [32], and Tenzing [16]. Hive is a popular open-source project that provides SQL-like processing for structured data on Hadoop. It is designed to be general purpose and works with a variety of workloads. Cheetah targets warehouse workloads on MapReduce, but does not provide details of the implementation or a comparison with other systems like Hive.

Llama is a recent system that combines columnar storage and tailored join algorithms. It demonstrates a speedup of at most 5x compared to Hive, while Clydesdale's speedup ranges from 5.2x to 82.7x. Llama also proposes joining more than two tables at a time using a *concurrent join* algorithm. The algorithm relies on storing column-group projections of the fact table sorted by the foreign key. The dimension tables are also stored sorted by foreign key, such that the join can be executed using a sort-merge plan. Storing multiple projections of the fact table sorted by each foreign key imposes substantial overhead for rolling in additional fact data. New data being rolled in needs to be split into column groups, and each column group needs to be merged with the corresponding column group of the fact table. Rolling in additional fact data is a common occurrence. Frequently requiring the entire fact table, which is typically very large, to be merged and rewritten to the filesystem is a prohibitive overhead. Clydesdale does not require that the fact table be stored in any sorted order. Roll-in and roll-out of fact table data is straightforward.

HadoopDB [7] takes an alternate approach of constructing a parallel DBMS by combining a single node relational database with Hadoop to get better performance. The HadoopDB architecture assumes that data is stored in the storage subsystem of individual database nodes instead of a distributed filesystem like HDFS. Fault-tolerance for the data is achieved either through reliable storage or replication. In this paper, we focus on supporting structured data processing workloads without discarding the distributed filesystem – a critical component of the MapReduce platform that provides reliable data access across a cluster of low-cost commodity disks. In fact, one can view Clydesdale as a demonstration that large performance improvements can be obtained for structured data processing on Hadoop without requiring a complex hybrid architecture like HadoopDB.

A recent paper describes Tenzing [16], a SQL implementation on Google's MapReduce platform. The paper describes several modifications to MapReduce including long-surviving worker processes to reduce latency, streaming data between successive MapReduce jobs, memory chaining to co-locate the reduce function of one MapReduce job and the map function of the next job in the same process, avoiding unnecessary sorting for MapReduce jobs that require only shuffling, and several other optimizations. All of these

techniques can be implemented in Hadoop, and are complementary to the ideas in Clydesdale. Likewise, the techniques used by Clydesdale (exploiting multiple cores by sharing dimension hash tables, the tailored join plan, and block iteration) can be used to improve star join performance in Tenzing.

Pig [22] and Jaql [12] are high-level languages for composing and executing complex dataflows on Hadoop. Pig and Jaql focus on supporting ETL-like workflows that require complex data transformations. Like Hive, they also support sort-merge and hash join strategies, and are also constrained to joining two tables at a time. Since the join strategies in Pig, Jaql, and Hive are essentially the same, we only compare with Hive and not with Pig and Jaql.

ASTERIX [11], DryadLinq [28], Nephele/PACT [10] propose ways to generalize the MapReduce paradigm. ASTERIX focuses on supporting semistructured data processing especially with evolving schema information. DryadLinq includes an execution engine that supports a general DAG, provides fault-tolerance, and operators that embed cleanly in .NET host languages. Nephele also generalizes the MapReduce paradigm. Scope [15] provides a simple SQL-like interface to processing large datasets on Dryad, but does not specifically address join processing workloads. In contrast to ASTERIX, DryadLinq, Nephele, and Scope, Clydesdale's focus is not to extend or generalize MapReduce, but to work within the constraints of Hadoop to provide efficient structured data processing.

3. BACKGROUND

We describe the extensibility points in Hadoop that were used to implement Clydesdale. These include `InputFormats`, `OutputFormats`, `MapRunners`, and schedulers.

A Hadoop job is typically configured with several parameters before it is launched. The `InputFormat` , `OutputFormat`, the map function, the reduce function, and the data types expected as inputs to the map function are all specified in the job configuration.

An `InputFormat` is an important extensibility point in Hadoop and is responsible for two functions. The first function is to generate `splits` of the data that can each be assigned to a map task. A job consists of several map and reduce tasks, each of which may run for several seconds to minutes. A `split` is the unit of scheduling and is a non-overlapping partition of the input data that is assigned to a map task. The second function is to transform data on disk to the typed key and value pairs that are required by the map function. An `InputFormat` implements the following two methods:

- **getSplits()** is used by the Hadoop scheduler to get a list of `splits` for the job.
- **getRecordReader()** is invoked by Hadoop to obtain an implementation of a `RecordReader`, which is used to read the key and value pairs from a given `split`.

Hadoop provides different `InputFormats` to consume data from text files, comma separated files, etc. The dual of an `InputFormat` in Hadoop is an `OutputFormat`, which is responsible for transforming the key-value pairs output by a MapReduce job to an on-disk format.

Hadoop also allows the user to specify a `MapRunner` along with the map and reduce functions. This is optional, and a default

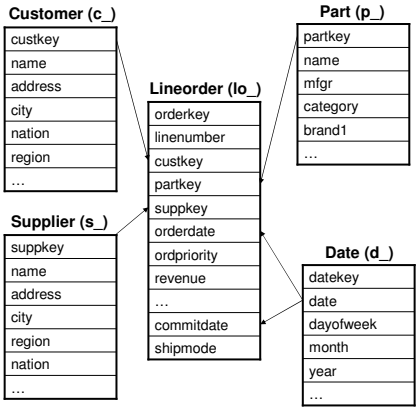

Figure 1: Example star schema.

`MapRunner` is used when this is not specified. The `MapRunner` encapsulates the logic for *how* to apply the map function on the input key-value pairs. The default `MapRunner` simply does the following: first, it takes the `split` that the task is supposed to process and opens a `RecordReader`. It then repeatedly reads a key-value pair from the `RecordReader` and applies the map function on it until all the records in the split have been processed. Alternate `MapRunner` implementations can be provided without recompiling Hadoop. This is useful for cases where different behavior is required, such as processing the keys and values in multiple threads. Section 5 describes how Clydesdale exploits this feature of Hadoop for improved performance.

Hadoop also has a feature called *JVM reuse*. By default, Hadoop runs each task in a separate Java Virtual Machine (JVM) . Enabling JVM reuse lets a single JVM run multiple consecutive map tasks from the same job. This is particularly useful for jobs where each map task may need to construct a large amount of state in memory before it begins processing. With JVM reuse, only the first map task on the node needs to construct this state. If this is stored as a static object, subsequent tasks running in the JVM can simply reuse this state. Section 5.2 describes Clydesdale's use of JVM reuse to eliminate redundant computation.

4. CLYDESDALE ARCHITECTURE

Clydesdale targets workloads where the data fits a star schema. A star schema consists of one or more fact tables referencing any number of dimension tables. An example from the star schema benchmark [33] is shown in Figure 1. In such datasets, the fact tables are usually much larger than the dimension tables – often by several orders of magnitude. A typical query joins the fact table with one or more dimension tables to aggregate measure columns in the fact table. Queries are currently written as Java programs that execute as MapReduce jobs on Hadoop.

Figure 2 shows the general architecture of Clydesdale. The fact table is stored in the distributed filesystem (HDFS) so that it is spread across all the nodes. A master copy of the dimension tables is available in HDFS. Dimension tables are also cached on the local storage of each node. New nodes, or nodes that have lost their local copy of the dimension data because of disk failures may copy the dimension data from HDFS.

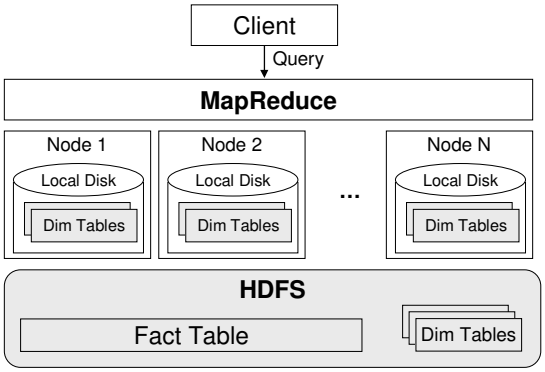

Figure 2: Architecture of Clydesdale.

4.1 Columnar Storage

The fact table is stored using `ColumnInputFormat` [21] (CIF). The benefits of being able to avoid I/O for the columns that are not used in a query are well known [14, 35]. CIF stores each column of a table in a separate HDFS file and only reads the files corresponding to the columns that are needed for the query. However, implementing columnar storage on a replicated distributed filesystem poses a unique challenge: storing each column in a different file makes it difficult to ensure that the corresponding HDFS blocks for different columns in a single row are co-located at every replica in HDFS. This is important to ensure that a task that processes a set of rows from the fact table can be scheduled at a node where the data for *all* the columns is available locally.

CIF is a column-oriented `InputFormat` that solves the above problem. and lets Hadoop continue using locality aware scheduling for map tasks. CIF leverages the support for pluggable placement policies in HDFS 21.0 to accomplish this. For details of how this is implemented and the interface presented to MapReduce programs that use CIF, see [21].

4.2 Join Strategy

Star-join queries in Clydesdale are executed as a MapReduce job. The map phase is responsible for joining the fact table with the dimension tables. The reduce phase is responsible for the grouping and aggregation. The flow for a typical join job is depicted in Figure 3. We describe the join plan using an example. Consider query 3.1 from the star schema benchmark:

```
SELECT c_nation, s_nation, d_year,
  sum(lo_revenue) as revenue
FROM  lineorder, supplier, date, customer
WHERE lo_custkey = c_custkey
  and lo_orderdate = d_datekey
  and lo_suppkey = s_suppkey
  and c_region = 'ASIA' and s_region='ASIA'
  and d_year >=1992 and d_year <= 1997
GROUP BY c_nation, s_nation, d_year
ORDER BY d_year asc, revenue desc;
```

The query computes the total revenue grouped by Customer's nation, Supplier's nation, and the year for orders between 1992 and 1997 where the Supplier and Customer were from Asia. In this query, the fact table is joined with three dimension tables, Supplier, Date, and Customer.

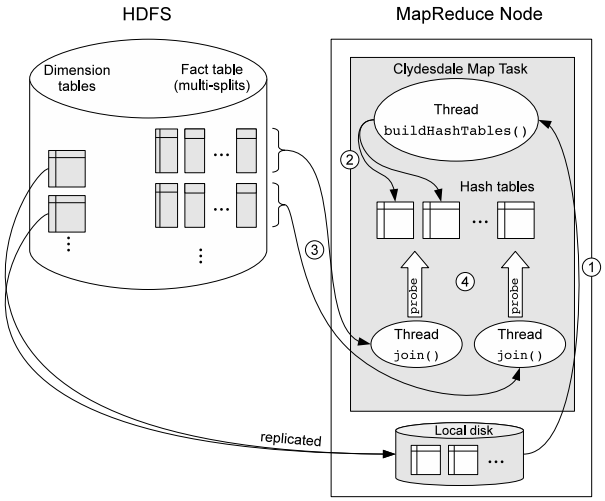

Figure 3: Clydesdale's join implementation.

Figure 4 shows the pseudocode for this job. The plan proceeds as follows: In the initialization phase of the map tasks, the predicates on the dimension tables are evaluated and a hash table is built for each dimension table with the qualifying rows. This happens in the `buildHashTables()` method in Figure 4 (line 5). The body of this method is omitted since the implementation is straightforward. In this example, `c_region` = 'ASIA' is the predicate on the Customer table, `s_region` = 'ASIA' is the predicate on the Supplier table, and $1992 \leq$ `d_year` ≤ 1997 is the predicate on the Date table. The key of the hash table is the primary key of the dimension table, and the value is the set of auxiliary columns referenced in the query for that dimension table. In the above example, `c_nation` , `s_nation`, and `d_year` are the auxiliary columns from the Customer, Supplier, and Date dimensions respectively. In general, a dimension hash table may contain zero or more auxiliary columns. Clydesdale currently uses a single-threaded algorithm to build each hash table. The degree of parallelism during the build phase is limited to the number of dimension tables joined to the fact table.

Once the hash tables are built, each map task begins to scan its `split` of the fact table. The list of columns that need to be scanned – the relevant foreign key columns and the measures used in the query – is pushed into CIF so that unnecessary disk I/O is avoided. For each row in the fact table, the hash tables are probed to check if the row satisfies the join conditions. This logic is implemented in the `probeHashTables()` method in line 8 of the figure. In this example, foreign key columns `lo_custkey`, `lo_suppkey`, and `lo_orderdate` are used to probe the Customer, Supplier, and Date hash tables respectively. After each successful probe, the fact table row is augmented with the auxiliary columns from the dimension table. The probing uses "early-out" in the sense that as soon as one of the probes fails, it is no longer necessary to probe the remaining dimension tables for the current row. The logic for `probeHashTables()` is relatively straightforward and the details are not shown in Figure 4.

A row is emitted as the output for the map function by constructing a key from the subset of columns needed for grouping. In this example, `c_nation` , `s_nation`, and `d_year` are included as part of the key (line 9). `lo_revenue` is emitted as the value (line 10). A simple `sum()` function is used to aggregate the value in the reducer (line 19). While not shown in Figure 4, combiners can

```
1.  class Query {
2.    class QMapper implements Mapper {
3.      List<HashTables> hlist;
4.      initialize(JobConf conf) {
5.        hlist = buildHashTables(conf);
6.      }

7.      void map (Record key, Record value, OutputCollector c) {
8.        if (probeHashTables(hlist, value)){
9.          Record groupKey = value.project(
                "c_nation,s_nation, d_year");
10.         Record rest = value.project("lo_revenue");
11.       }
12.       c.collect(groupKey,rest);
13.     }
14.   }

15.   class QReducer implements Reducer {
16.     void reduce (Record key, Iterator<Record> values,
          OutputCollector c) {
17.       Float sum = 0;
18.       while (values.hasNext()){
19.         sum += values.next().get("lo_revenue");
20.       }
21.     c.collect(key, sum);
22.     }
23.   }

24.   main(){
25.     job = new JobConf();
26.     ColumnInputFormat.addInputPath(job,"/data/facttable");
27.     job.setOutput("/tmp/output/");
28.     job.set("dimtables.directory", "/local/tmp/dimensions");
29.     ...
30.     ...
31.     job.set("queryparams", queryParams);
32.     JobClient.runJob(job);
33.     sortResult("/tmp/output", "/output/result", conf);
34.   }
35.}
```

Figure 4: MapReduce job pseudocode for the example query.

be used for partial aggregation before sending the results over to the reducers. The final order by clause is simply evaluated using a single process sort (line 33). All the relevant information for the job is passed in through the `JobConf` in the main function (lines 24 – 34).

5. CHALLENGES

While the join algorithm described above seems straightforward, naive implementations are not able to achieve good performance. For an efficient implementation we found that it is necessary to reconsider how multi-core parallelism is used in MapReduce, how MapReduce Tasks are scheduled, and how individual rows or key-value pairs are processed.

5.1 Exploiting Multi-Core Parallelism

While contemporary servers typically have 8 – 16 cores, core counts are rapidly increasing with 32-core and 64-core servers likely to be commonplace very soon. A typical Hadoop cluster takes advantage of multiple cores by configuring the nodes to have multiple *slots* for map and reduce tasks. As a result, as many (map and reduce) tasks are scheduled on a node as there are slots available. Each task is run in a separate JVM for fault isolation.

```
1.  class MTMapRunner{
2.    void run(RecordReader input, OutputCollector output) {
3.      buildHashTables(conf);
4.      RecordReader[] inputs = input.getMultipleReaders();
5.      List<Thread> tList = new List<Thread>();
6.      for (r in inputs) {
7.        tList.add(new JoinThread(r, output));
8.      }
9.      waitForAllThreadsToComplete(tList);
10.   }
11.   class JoinThread extends Thread {
12.   ...
13.     void run() {
14.       while ( input.next(key, value)) {
15.         //do join processing
16.       }
17.     }
18.   }
19.}
```

Figure 5: MapRunner class used in Clydesdale for the probe phase.

For the join algorithm described in Section 4.2, if each map task independently computes a set of hash tables, then each server will have as many copies of the hash tables in memory as there are map slots. This approach becomes impractical for servers with a large number of map slots, especially when dimension hash tables have non-trivial sizes. The problem at hand is to find a solution without changing the underlying implementation of Hadoop.

Clydesdale tackles this problem in two parts. First, a Clydesdale job, when submitted, requests the Hadoop scheduler to only schedule a *single* map task per node irrespective of the number of map slots available on the node. Section 5.2 discusses how this is done. Second, it uses a custom `MapRunner` class to run a multi-threaded map task that can occupy all the slots on the node. Recall that a custom `MapRunner` can be used without any changes to the Hadoop codebase itself. This class, called `MTMapRunner` is shown in Figure 5. The `MTMapRunner` replaces the default `MapRunner` and incorporates the join processing logic in Figure 4 (lines 2 – 14). This map task can now use a single copy of the dimension hash tables in memory that can be shared by all the threads. Once they are built, the hash tables are read-only data structures and therefore do not require synchronization for access by multiple threads.

Using a multi-threaded `MapRunner` reduces the number of copies of the hash tables to one per node, but introduces another problem. Since the scheduler only assigns a single `split` per map task, all the threads on a single node read from a single `split`. The `next()` method on the `RecordReader` for a `split` is synchronized. As a result, all the threads in the `MapRunner` get bottlenecked on deserializing data from the `split`. Recall that a `split` provides a `RecordReader()` to iterate through the key-value pairs in it. The `next()` method is responsible for performing I/O from HDFS if necessary and deserializing the key-value pair.

We overcome this bottleneck by wrapping CIF into a class called `MultiColumnInputFormat` (*MultiCIF*) that packs multiple input `splits` into a single `multi-split`. The number of `splits` to pack into a single `multi-split` can be configured. `MTMapRunner` can unpack the `multi-split` and create a `RecordReader` for each constituent `split` (line 4 in Fig-

ure 5). Now each thread can independently read and deserialize key-value pairs (lines 6 – 8) and the input `split` is no longer a deserialization bottleneck.

The hash tables are also shared across consecutive map tasks that run on the same node. Clydesdale exploits Hadoop's JVM reuse feature to ensure that the next map task that is scheduled on this node runs in the same JVM. By storing the hash tables as static data structures, the next map task can reuse them for processing its `multi-split`. As a result of these optimizations, dimension hash tables are computed exactly once per node for a given query in Clydesdale.

Discussion. The total size of the hash tables is limited by the memory on each node. We note that in a majority of data repositories, dimension tables are small – the largest dimension tables encountered in practice are often a few gigabytes. These are several orders of magnitude smaller than the fact table, which can be hundreds of terabytes or larger. The hash table often needs to be built on only a subset of the rows and columns in the dimension table, and is usually even smaller. Given that system memory sizes are increasing rapidly – 64GB per node are common for modern servers – we believe that it is reasonable to assume that dimension hash tables will fit in the memory of a single node. For the rare case where the cluster nodes have little memory or for unusual datasets with extremely large dimension tables, one could reduce the memory footprint by joining with a single hash table at a time. A subsequent pass over the intermediate joined result can be made to join with the remaining dimension tables, either all at once or a single table at a time. This strategy works when the aggregate size of the dimension hash tables exceeds available memory, but each one fits in memory by itself. For the case of a single large dimension table, we expect to resort a repartition join strategy [13, 38].

5.2 Task Scheduling

Scheduling in MapReduce is designed to work with large heterogeneous clusters where each job can be broken into small schedulable units – map and reduce tasks. The actual scheduling implementation is pluggable, and different schedulers can be used for different resource management objectives. Hadoop scheduling assumes that each map task can be executed by a single core in a relatively short amount of time (a few minutes). Clydesdale's join tasks break with the assumption that map tasks are short single-core tasks. Each map task runs multiple threads so the hash tables are shared across them. Given these constraints, the scheduler needs to:

1. Schedule only one map task from the join job on a given node. This will prevent wasteful re-computation of dimension hash tables in multiple slots on a node.
2. Schedule subsequent map tasks from the join job on the nodes where the dimension hash table has already been built so that they can be reused. This may need to be balanced with respect to the locality requirements of other jobs running in the cluster.
3. Communicate to the map task the number of slots, or processor cores it can use on the node. This is important to ensure that the other tasks from other jobs executing on this node get their share of CPU time.

Clydesdale accomplishes (1) by marking each task as requiring a large amount of memory. The capacity scheduler [1] on Hadoop can use this information to schedule only one task from the job on each node. Subsequent map tasks from the same job on this node do not have to build the hash tables again, thanks to JVM reuse.

In evaluating our current prototype, since the entire cluster was used to run the join queries we did not encounter the problems in (2) and (3) above. However, using Clydesdale *efficiently* on a large shared cluster will require some improvements to the scheduling support in Hadoop. In fact, we believe that the changes being proposed as part of the next generation of Hadoop [5] will be sufficient to support Clydesdale and other workloads concurrently on a large cluster. A more detailed discussion of tradeoffs in scheduling complex workload mixes is beyond the scope of this paper. We refer the reader to recent work on multi-framework scheduling [26, 5].

5.3 Block Iteration

The overheads of a Volcano-style [23] row-at-a-time iterator pipelining model for relational databases has previously been described [39]. Not surprisingly, query processing on Hadoop can also suffer from high per-row overheads. For workloads like text analytics which require substantial CPU work per document, overheads of a few function calls per key-value pair do not matter. However, for a structured data processing, the number of instructions performing useful work may be small compared to the overhead of moving a single key-value pair through the framework, one at a time.

We solve this problem with a block iteration technique that amortizes the overheads associated with retrieving a single key-value pair from a `split` (or `multi-split`) over a block of key-value pairs. We layer a new `BlockColumn- InputFormat` (*B-CIF*) over the same input data to return an array of rows instead of a single row at a time. The array is populated by filling up a block of values from each column at a time. As a result, the cost of `RecordReader.next()` is incurred only once per block of rows. Furthermore, this allows the underlying deserializers for the different columns to be called in a tighter loop leading to better cache performance. Columnar execution techniques such as late tuple reconstruction [6, 27], while not currently implemented, may be used to further improve cache performance.

6. EXPERIMENTS

We compare the performance of Clydesdale with Hive [3] using the star schema benchmark [33]. We examine the execution plans for both systems, and analyze how each of the features discussed in Section 5 contributes to Clydesdale's performance.

6.1 Hive Background

Hive is a popular open-source project that brings large-scale structured data processing using SQL to Hadoop. Hive is fairly general-purpose, and supports two join plans: the re-partition join (also referred to as *common join*) and the broadcast join (also referred to as *mapjoin*). A re-partition is essentially a sort-merge join [13, 38] where the mappers tag each input record with the table they come from and output the record with the join column as the key. The records from both tables with a given join key end up at the same reducer, which actually joins the tuples. This is a robust technique that works with any combinations of sizes of the tables being joined. The primary disadvantage of this technique is that it requires both the tables to be sent over the network during the shuffle phase, which often becomes the bottleneck.

A mapjoin in Hive resembles a hash join and is designed for the case where one table is significantly smaller than the other. Figure 6 shows how this plan works. First the Hive master node builds a hash table on the smaller table. The hash table is then serialized, compressed, and disseminated to the map tasks using Hadoop's distributed cache mechanism. The distributed cache broadcasts data by copying it to HDFS. Each node can then read this data off HDFS and write it to local storage. The distributed cache ensures that the data is copied to each node only once per job irrespective of the number of map slots on the node. The map tasks then read and deserialize the hash table into memory, where they probe it against the local `splits` of the larger table, emitting the join results.

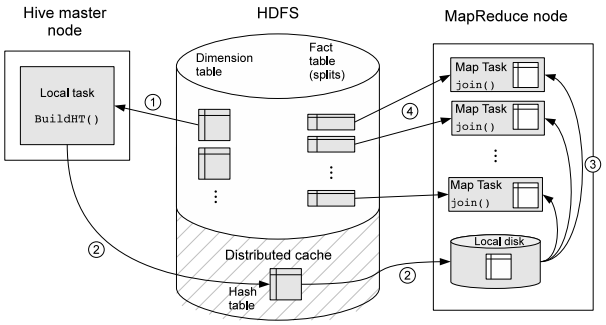

Figure 6: Hive's Mapjoin plan.

6.2 Experimental Setup

Hardware. For our experiments we report results from two clusters, A and B. **Cluster A** comprises 9 nodes connected by a 1Gbit ethernet switch. One node was assigned to run the Hadoop *jobtracker* (the MapReduce master node responsible for assigning tasks to different worker nodes) and the *namenode* (the master node for HDFS). Eight worker nodes were used to store data and to run MapReduce jobs. Each node had two quad-core AMD Opteron processors, 16GB of main memory, and eight 250GB SATA disks.

Cluster B is a larger cluster with 42 nodes, also connected by a 1Gbit ethernet switch. Two nodes were reserved to run jobtracker and namenode. The remaining 40 nodes were used for HDFS and MapReduce. Each node had two quad-core Intel Xeon processors with 32GB of main memory and five 500GB SATA disks.

These clusters represent two common tradeoffs made when configuring servers for Hadoop. A portion of the budget can be spent either on additional memory or additional disks. Cluster A is more memory constrained, i.e. 2GB per core as opposed to 4GB per core on cluster B.

Software Configuration. Both clusters were running 64-bit Linux with Kernel 2.6. Hadoop was configured to run six map *slots* (mappers) and one reduce slot (reducer) per node. Both clusters have Hadoop version 0.20.2 and 0.21.0 installed, running on the Sun JVM 1.6.0_16 14.2-b01. Clydesdale runs on top of Hadoop-0.21.0, while Hive-0.7.0 uses Hadoop-0.20.2. Clydesdale requires the pluggable block placement policy [2] feature currently only available in Hadoop-0.21.0. On the other hand, Hive 0.7.0 does not support Hadoop-0.21.0 at the moment. Our single node tests running Clydesdale confirmed that there was no significant performance difference between using either version of Hadoop.

Workload. We used the star schema benchmark [33], a modified TPC-H benchmark. It uses a single central fact table that references several dimension tables as shown in Figure 1. In the experiments below we compare the execution times of its queries at scale factor (SF) 1000. At SF1000 the size of the uncompressed fact table in text format is approximately 600GB, while the dimension tables are significantly smaller – Customer (2.8GB), Supplier (828MB), Part (166MB), and Date (225KB), under 4GB in total.

The workload consists of four query flights, with three or four queries each, with varying predicates and selectivities. The first flight is based on joining the fact table with the smallest dimension table – Date, to produce the aggregate earnings for a specific year. The second flight consists of a join with three dimension tables – Date, Part, and Supplier and adding a group by operator. The third flight involves a join with Customer, Supplier, and Date. Finally, the fourth flight involves joining all four dimension tables. Each query computes an aggregate on the measure columns in the fact table and groups by one or more columns from the dimension tables. See [33] for more details on these queries.

Storage Format. For Clydesdale, the fact table was stored in Multi-CIF format [21], whose binary encoding reduced the size to approximately 334GB in HDFS. The dimension tables were stored in HDFS in binary format and all data nodes kept a local copy as well.

For the Hive experiments, all tables were stored in RCFile [24] format, which required approximately 558GB of disk space. RCFile [24] is a recently introduced hybrid columnar format for Hadoop that uses a PAX [9]-like layout of records within each HDFS block to eliminate unnecessary I/O. As we configured HDFS with a replication factor of three, the above numbers have to be multiplied by three to obtain the total amount of disk space required. Neither Clydesdale nor Hive used any indexes.

6.3 Comparison with Hive

Figures 7 and 8 show the execution time of all star schema benchmark queries using Clydesdale vs. Hive on clusters A and B respectively. The time reported is the average execution time from three runs of the query. In each case, the filesystem cache was flushed before running the query to ensure that the data was being read from the disks. For Hive, the execution times are shown for both the repartition join and the mapjoin plan. Hive results in the figure are also annotated with the speedup Clydesdale offers compared to Hive. This was computed by simply dividing the execution time of Hive by that of Clydesdale for the same query.

Clydesdale was 17.4x to 82.7x faster than Hive, averaging a 38x speedup on cluster A. On cluster B, the speedup ranged from 5.2x to 21.4x, averaging 11.1x. In order to better understand the performance differences between Clydesdale and Hive, we examine the query plan produced by each system for a representative query. Consider query 2.1, which joins the fact table with three dimension tables: Date, Part, and Supplier.

```
SELECT sum(lo_revenue), d_year, p_brand1
FROM lineorder, date, part, supplier
WHERE lo_orderdate = d_datekey
  and lo_partkey = p_partkey
  and lo_suppkey = s_suppkey
  and p_category = 'MFGR#12'
  and s_region = 'AMERICA'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1;
```

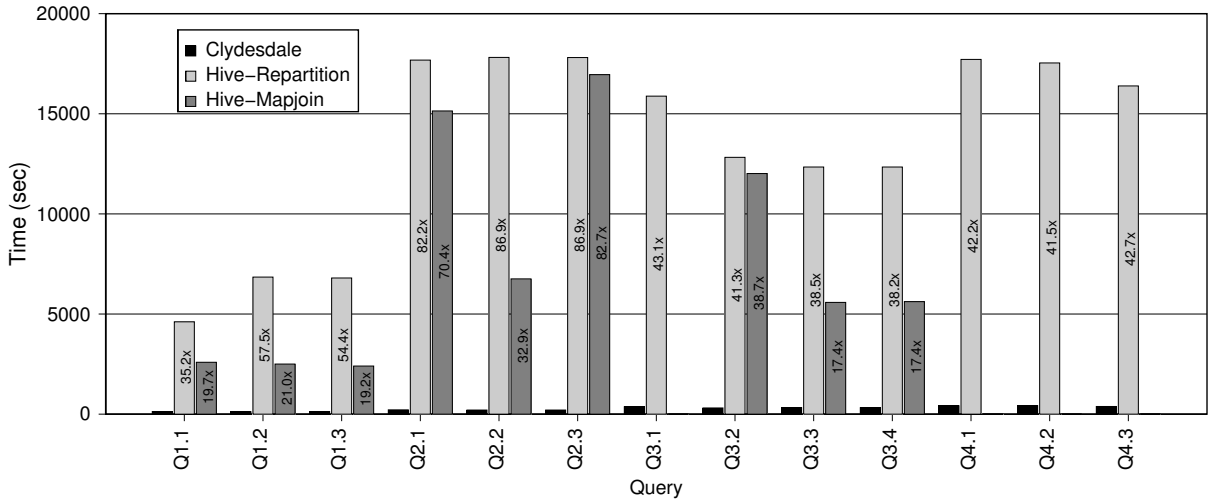

Figure 7: Clydesdale vs. Hive at SF1000 on Cluster A (9 nodes).

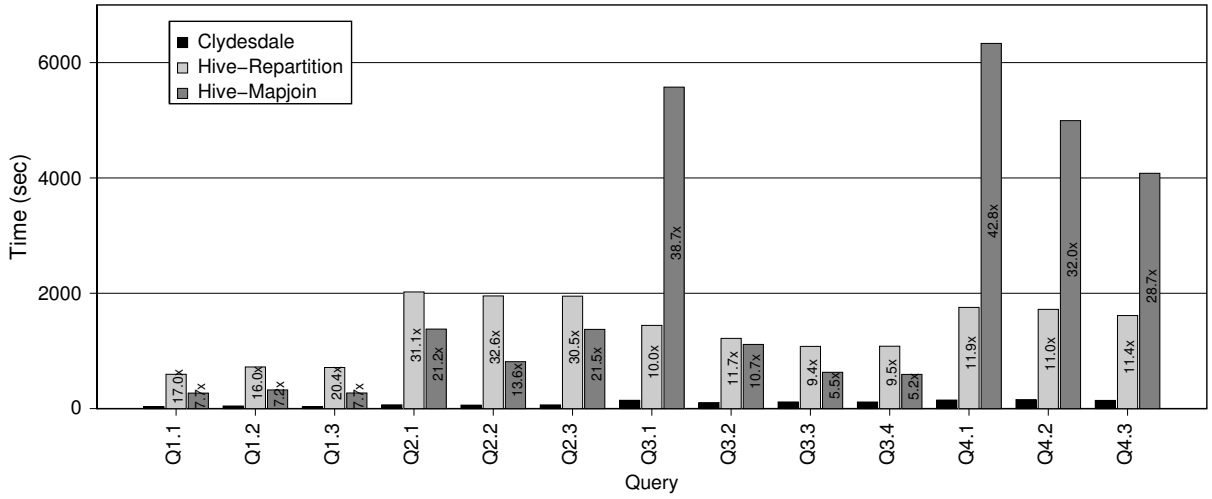

Figure 8: Clydesdale vs. Hive at SF1000 on Cluster B (42 nodes).

We breakdown the execution times of this query on cluster A. Clydesdale took 215 seconds for query 2.1 compared to 15,142 seconds for Hive. On Clydesdale, eight map tasks were executed, one on each worker node, each taking on average about 200 seconds. Recall that Clydesdale runs only one multi-threaded map task per node at a time, and the hash tables on the dimensions need to be built exactly once per node.

Examining the logs of a representative map task we noted that it took 27 seconds to build the three hash tables and 164 seconds to process the actual join. The task processed 10.8GB worth of fact table data, the count of actual bytes read from HDFS, at a rate of 67 MB/s (over the 164 second period). Interestingly, this is substantially lower than the raw I/O bandwidth available on the node, which is over 560MB/s. This is in part because of performance problems that affect HDFS (see Section 6.6). The final sort for the order by clause took under 10 seconds.

For query 2.1, Hive generates a five stage mapjoin plan. Each stage is a MapReduce job. The first three stages perform joins with the three dimension tables – Date, Part, and Supplier – one at a time. Building and distributing the hash table on Date took under 2 seconds. During the join with the Date dimension (the first MapReduce job) each map takes approximately 25 seconds to process a `split`. There are no reduce tasks. Overall, stage 1 took 2,640 seconds. It processed 4,887 map tasks averaging 25 seconds on the 48 map slots across cluster A. Note that each map task had to reload the hash table from disk. Across the cluster, this was done 4,887 times. This is clearly redundant work compared to the approach in Clydesdale that needs to do this once per node. The RCFile `InputFormat` did not allow us to decrease the number of `splits`. When we re-ran this query using the `TextInputFormat`, decreasing the number of `splits` increased the runtime by 10% to 20%.

Stages 2 and 3 of the query joined the intermediate result with the Part and Supplier dimensions taking 2,040 seconds and 9,180 seconds respectively. Unlike Clydesdale, since the joins in Hive are performed one after another, the intermediate results have to be written to HDFS and read back. This additional I/O also adds over-

head. Stage 2 and stage 3 read approximately 200GB and 188GB, eventually producing a join output of size 11GB. Stage 3 took the longest to execute since the size of the dimension hash table for Supplier was relatively large, occupying 100MB compressed on disk and about 500MB decompressed in memory.

After the joins completed, Hive launched 2 more MapReduce jobs, one for the group by (720 seconds) and one for order by operation (19 seconds). Overall, the mapjoin plan for query 2.1 took 15,142 seconds.

Hive's repartition join plan for query 2.1 also comprises five stages of which the first three essentially perform a sort-merge join [13] with the dimension tables – Date, Part and Supplier – one at a time. Stage 1 took 9,720 seconds, stage 2 took 7,140 seconds, and stage 3 took 420 seconds. Stages 4 and 5 were comparable to the mapjoin plan. The overall time for the repartition join plan was 17,700 seconds.

6.4 Discussion

A possible criticism of this comparison is that while Clydesdale replicates the dimension tables on to local storage on each node, Hive's current implementation does not support such a strategy. However, merely fixing this will not improve Hive's performance to be comparable to Clydesdale. In fact, in each of the queries, disseminating the hash table is a relatively small fraction of the query cost compared to all the other overheads described above. For instance, in query 2.1, where the overall running time was 15,142 seconds on cluster A, the time spent in building and distributing the three hash tables was around 192 seconds. Even subtracting this out of the runtime for Hive, Clydesdale is still 70x faster.

The speedup Clydesdale provides over Hive is greater for queries that include more dimension tables and those that produce larger dimension hash tables. Given the discussion of the query plans above, the reasons are fairly obvious. First, Hive joins one dimension table at a time with the fact table. This requires several MapReduce jobs compared to Clydesdale's approach of joining with all the required dimension tables in a single MapReduce job. Second, Hive maintains as many copies of the hash tables in memory as there are map tasks on a machine. As a result of this problem, queries 3.1, 4.1, 4.2, and 4.3 did not complete using the Hive mapjoin plan on cluster A because of out-of-memory errors. Cluster B had more memory per node and was able to complete the mapjoin plan. Clydesdale's use of `MTMapRunners` allows it to share a single copy across multiple join threads on a node. Third, Hive creates the hash table on a single node and pays the cost of disseminating it to the entire cluster. Clydesdale exploits the fact that dimension tables are smaller and replicates them on local storage. Finally, each map task in Hive has to load and deserialize the hash table when it starts. Hive does not currently reuse JVMs to retain the hash table in memory throughout a given job.

Clydesdale's performance speedup for the SF1000 workload on cluster A was larger than that on cluster B. This is because the amount of work done per node on cluster B, the larger cluster, was significantly lower given the fixed amount of data. Obviously, the number of `splits` processed per node on the larger cluster (B) was five times smaller as it had five times the number of nodes. As a result, the fraction of time each node spent building the dimension hash tables was significant compared to the time spent actually processing the join. For instance, in query 2.1 above the typical map task spent 16 seconds building the dimension tables, and 29 seconds in the probe phase with an overall run time of around 65 seconds. Furthermore, since the overall runtimes are relatively short, under 180 seconds in all cases, the Hadoop scheduling overheads become a non-trivial part of the runtime. We expect that Clydesdale's advantages over Hive will continue to hold on Cluster B with larger scale factors, e.g. 10,000. Unfortunately the data generator for the star schema benchmark [33] does not currently support scale factor 10,000. Verifying performance at SF 10,000 is left as future work.

6.5 Analysis of Clydesdale

We now present experiments that analyze the impact of the individual techniques used in Clydesdale and their contribution to the overall speedup. We turn off each of its features – block iteration, columnar storage, and the use of multi-threaded map tasks – one at a time, to measure their impact on the runtime. Figure 9 shows the results of this experiment on Cluster A, again at SF 1,000.

The average slowdown from turning off block iteration was approximately 1.2x. Turning off columnar storage, i.e., reading all the columns of the fact-table stored in CIF, resulted in a slowdown of 3.4x and made the scan of the fact table the bottleneck. As expected, queries that originally touched fewer bytes from the fact table, i.e. only scanned a few columns were more adversely affected by this change than the queries scanned more columns. For instance, query flight 2, which only scanned 4 columns in the fact table slowed down by 3.8x. On the other hand, query flight 4, which scanned 6 columns was slower by 2.0x.

Finally, turning off the use of multi threaded tasks slowed down performance by 2.4x. Each task was executed single threaded and built its own copy of the dimension hash tables. Again, queries that joined the fact table with more or larger dimension tables were slowed down substantially more than those requiring fewer or smaller dimension tables. For instance, query flight 1 was slowed down by just 1.2x as it only joins the fact table with Date, the smallest dimension table. On the other hand, query flight 4, that touched all the dimension tables was 4.5x slower.

In summary, no single technique was responsible for all of Clydesdale's improved performance. Each of these techniques were complementary, and in combination, resulted in the overall performance improvement.

6.6 Limitations

For structured data processing on Hadoop, one of the main bottlenecks appears to be HDFS bandwidth. The bandwidth at which map tasks could read from HDFS was only a fraction of the raw disk bandwidth observed using `dd` . For instance, we measured that each disk was able to supply between 70MB/s and 100MB/s. Conservatively assuming 70MB/s per disk would result in 560MB/s for cluster A's eight disks and 280MB/s for cluster B's four disks. To check if HDFS could deliver this bandwidth, we ran the TestDFSIO benchmark, included in the Hadoop distribution. The benchmark consists of a write and a read job. Each map task of the write job writes a file of pre-specified size to HDFS, while the mappers in the read job retrieve the previously written files from HDFS. Locality is respected, such that data is being read from the disks local to the node. The filesystem cache on each node in the cluster was flushed before running the read test to ensure that the data was actually being read from disk. The results of this experiment are summarized in Table 1.

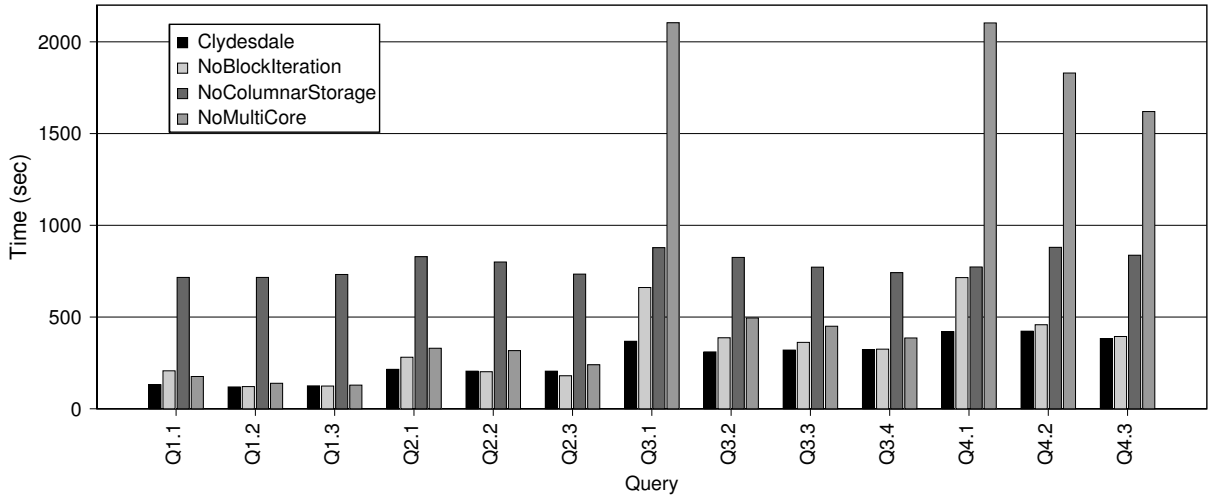

Figure 9: Clydesdale's performance as different enhancements are turned off one at a time.

Cluster	Disk B/W (MB/sec)	HDFS B/W (MB/sec)
Cluster A	560	126
Cluster B	280	105

Table 1: HDFS read bandwidth available per node.

We were surprised by the poor results observed in these experiments, with HDFS providing only a fraction of the raw bandwidth available on the nodes. We traced the reason for the poor bandwidth from HDFS to a bug in the Sun JVM that gets triggered by the NIO socket implementation in Hadoop. This is a known issue [4] that leads to excessive CPU usage ultimately resulting in reduced I/O bandwidth from HDFS. As a result of this bug, the additional disks on cluster A (8 disks) did not really provide significantly better I/O bandwidth. We believe that Clydesdale's performance will be substantially improved once this bug is fixed and HDFS is able to deliver better I/O bandwidth.

7. CONCLUSIONS

In this paper, we described the design and architecture of Clydesdale, a system for structured data processing on Hadoop. Clydesdale draws on many recent techniques from DBMSs including columnar storage, tailored plans for star schemas, block iteration for efficiency, and multi-core aware execution plans. We showed that with a careful combination of these techniques Clydesdale is able to out-perform Hive by 38x on the star schema benchmark. Clydesdale achieves these results without requiring *any* changes to the implementation of Hadoop. This allows Clydesdale to inherit all the attractive properties of Hadoop including the ability to run on low-cost hardware, fault-tolerance, elasticity, and scalability. We view this result as evidence that MapReduce in general, and Hadoop in particular is significantly more compelling as a platform for structured data processing than previously assumed.

8. FUTURE WORK

While Clydesdale's performance advantage over Hive is often more than an order of magnitude, and sometimes nearly two orders of magnitude, we would like to point out that Clydesdale is a research prototype and not a fully functional system like Hive. Clydesdale does not currently have a SQL parser. Queries are written as MapReduce programs in Java. We are working on building a parser and compiler for a simple subset of SQL.

Currently, neither Hive nor Clydesdale support updates to the dimension tables. In Clydesdale, we plan to manage updates by keeping the dimension tables in a separate transactional database and periodically exporting a copy of the dimension tables to HDFS. From there, each node copies over the data from HDFS to the local storage. We plan to use a simple form of snapshot isolation along with the batch update of the dimension tables. Clydesdale can take advantage of several advanced techniques from column store databases such as storage organization [29] and late tuple reconstruction [27]. Finally, efficiently scheduling Clydesdale workloads alongside traditional MapReduce workloads poses new workload management challenges.

9. REFERENCES

[1] Capacity Scheduler. *http://hadoop.apache.org/common/docs/r0.20.2/capacity-scheduler.html*.

[2] HDFS Issue 385. *https://issues.apache.org/jira/browse/HDFS-385*.

[3] Hive. *http://hive.apache.org/*.

[4] MAPREDUCE Issue 2386. *https://issues.apache.org/jira/browse/ MAPREDUCE-2386*.

[5] Next Generation of Apache Hadoop MapReduce – The Scheduler. *http://developer.yahoo.com/blogs/hadoop/posts/2011/03/mapreduce-nextgen-scheduler*.

[6] D. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, pages 967–980, 2008.

[7] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[8] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, pages 99–110, 2010.

[9] A. Ailamaki, D. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[10] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC*, pages 119–130, 2010.

[11] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-World Models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[12] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 2011.

[13] 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 *SIGMOD Conference*, pages 975–986, 2010.

[14] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, pages 54–65, 1999.

[15] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.

[16] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL Implementation on the MapReduce Framework. *PVLDB*, 4(12):1318–1327, 2011.

[17] S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(2):1459–1468, 2010.

[18] X. Chen, P. E. O'Neil, and E. J. O'Neil. Adjoined Dimension Column Clustering to Improve Data Warehouse Query Performance. In *ICDE*, pages 1409–1411, 2008.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.

[20] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3(1):518–529, 2010.

[21] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-Oriented Storage Techniques for MapReduce. *PVLDB*, 4(7), 2011.

[22] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience. *PVLDB*, 2(2):1414–1425, 2009.

[23] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6:120–135, February 1994.

[24] Y. He, R. Lee, S. Zheng, N. Jain, Z. Xu, and X. Zhang. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *ICDE*, 2011.

[25] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, pages 261–272, 2011.

[26] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Networked Systems Design and Implementation*, pages 22–22, 2011.

[27] S. Idreos, M. L. Kersten, and S. Manegold. Self-Organizing Tuple Reconstruction in Column-Stores. In *SIGMOD*, pages 297–308, 2009.

[28] M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *SIGMOD Conference*, pages 987–994, 2009.

[29] M. Ivanova, M. L. Kersten, and N. Nes. Self-Organizing Strategies for a Column-Store Database. In *EDBT*, pages 157–168, 2008.

[30] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6), 2011.

[31] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1):472–483, 2010.

[32] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In *SIGMOD Conference*, 2011.

[33] P. E. O'Neil, E. J. O'Neil, and X. Chen. The Star Schema Benchmark (SSB). *http://www.cs.umb.edu/ËIJponeil/StarSchemaB.PDF*.

[34] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, pages 165–178, 2009.

[35] 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*, pages 553–564, 2005.

[36] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[37] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE*, pages 657–668, 2010.

[38] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD*, pages 1029–1040. ACM, 2007.

[39] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.