A Comparison of Join Algorithms for Log Processing in MapReduce

Spyros Blanas, Jignesh M. Patel
Computer Sciences Department
University of Wisconsin-Madison
{sblanas,jignesh}@cs.wisc.edu

Vuk Ercegovac, Jun Rao,
Eugene J. Shekita, Yuanyuan Tian
IBM Almaden Research Center
{vercego,junrao,shekita,ytian}@us.ibm.com

ABSTRACT

The MapReduce framework is increasingly being used to analyze large volumes of data. One important type of data analysis done with MapReduce is log processing, in which a click-stream or an event log is filtered, aggregated, or mined for patterns. As part of this analysis, the log often needs to be joined with reference data such as information about users. Although there have been many studies examining join algorithms in parallel and distributed DBMSs, the MapReduce framework is cumbersome for joins. MapReduce programmers often use simple but inefficient algorithms to perform joins. In this paper, we describe crucial implementation details of a number of well-known join strategies in MapReduce, and present a comprehensive experimental comparison of these join techniques on a 100-node Hadoop cluster. Our results provide insights that are unique to the MapReduce platform and offer guidance on when to use a particular join algorithm on this platform.

Categories and Subject Descriptors

H.2.4. [**Systems**]: Distributed Databases; H.3.4. [**Systems and software**]: Distributed Systems

General Terms

Performance, Experimentation

Keywords

MapReduce, Hadoop, Join processing, Analytics

1. INTRODUCTION

Since its introduction just a few years ago, the MapReduce framework [16] has become extremely popular for analyzing large datasets in cluster environments. The success of MapReduce stems from hiding the details of parallelization, fault tolerance, and load balancing in a simple programming framework. Despite its popularity, some have argued that MapReduce is a step backwards and ignores many of the valuable lessons learned in parallel RDBMSs over the last 30 years [18, 25]. Among other things, the critics of MapReduce like to point out its lack of a schema, lack of a declarative query language, and lack of indexes.

We agree with all of these criticisms. At the same time, we believe that MapReduce is not a passing fad, as witnessed by the rapidly growing MapReduce community, including many of the big Web 2.0 companies like Facebook, Yahoo!, and of course Google. Even traditional enterprise customers of RDBMSs, such as JP Morgan Chase [1], VISA [2], The New York Times [3] and China Mobile [4] have started investigating and embracing MapReduce. More than 80 companies and organizations are listed as users of Hadoop – the open source version of MapReduce [5]. In fact, IBM is engaged with a number of enterprise customers to prototype novel Hadoop-based solutions on massive amount of structured and unstructured data for their business analytics applications. A few examples were provided in the talk given by IBM at the Hadoop Summit 2009 [6]. Therefore, it is very important to study analytic techniques on this new platform.

MapReduce is being used for various data analytic applications. Based on our service engagement with enterprise customers, as well as the observation from the use of MapReduce in Web 2.0 companies, log processing emerges as an important type of data analysis commonly done with MapReduce. In log processing, a log of events, such as click-stream, log of phone call records or even a sequence of transactions, is continuously collected and stored in flat files. MapReduce is then used to compute various statistics and derive business insights from the data.

There are several reasons that make MapReduce preferable over a parallel RDBMS for log processing. First, there is the sheer amount of data involved. For example, China Mobile gathers 5–8TB of phone call records per day [4]. At Facebook [7], almost 6TB of new log data is collected every day, with 1.7PB of log data accumulated over time. Just formatting and loading that much data into a parallel RDBMS in a timely manner is a challenge. Second, the log records do not always follow the same schema. Developers often want the flexibility to add and drop attributes and the interpretation of a log record may also change over time. This makes the lack of a rigid schema in MapReduce a feature rather than a shortcoming. Third, all the log records within a time period are typically analyzed together, making simple scans preferable to index scans. Fourth, log processing can be very time consuming and therefore it is important

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

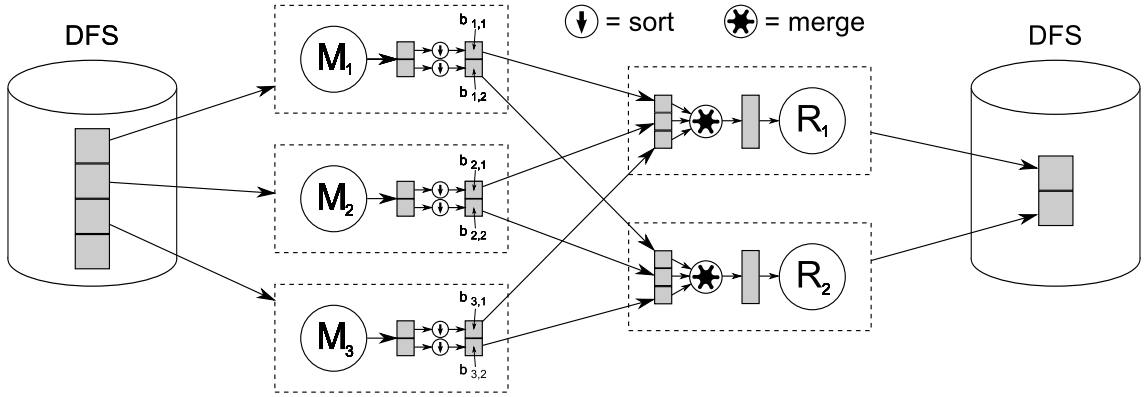


Figure 1: Overview of MapReduce.

to keep the analysis job going even in the event of failures. While most parallel RDBMSs have fault tolerance support, a query usually has to be restarted from scratch even if just one node in the cluster fails. In contrast, MapReduce deals more gracefully with failures and can redo only the part of the computation that was lost because of a failure. Last but not least, the Hadoop [8] implementation of MapReduce is freely available as open-source and runs well on inexpensive commodity hardware. For non-critical log data that is analyzed and eventually discarded, cost can be an important factor. In comparison, a commercial parallel RDBMS can cost hundreds of thousands of dollars, even on a moderate size cluster.

There are, of course, many different ways of doing log processing with MapReduce. However, at an abstract level, a few common patterns emerge (e.g., [7, 9]). The log data is typically stored in the underlying distributed file system (DFS) in timestamp order. In addition, there are usually one or more reference tables containing information about users, locations, etc. These reference tables vary in size but are usually much smaller than the log data. They are typically maintained in an RDBMS but copied to the DFS to make log processing in MapReduce more efficient. For example, at Facebook, 4TB of reference data is reloaded into Hadoop's DFS every day [7].

A simple log processing job in MapReduce might scan a subset of log files corresponding to a specified time window (e.g. a day), and compute aggregated counts grouped by age, gender or location. As part of this analysis, an equi-join is often required between the log and one or more of the reference tables. Note that the equi-join for log processing is different from typical star joins in traditional data warehouse applications. In star joins, dimension data is often first reduced based on constraints, then the product of several dimension tables is joined with the fact table often making use of indexes. In contrast, equi-join for log processing is usually performed on all the log records, and constraints on reference data are seldom specified. This is because one of the goals of log processing is to compute statistics for all the log records on various combination of reference dimensions. Also note that it is not feasible to pre-join the log with all its reference data, since the pre-joined result could be orders of magnitude larger than the log.

The equi-join between the log and the reference data can have a large impact on the performance of log processing. Unfortunately, the MapReduce framework is somewhat cumbersome for joins, since it was not originally designed to combine information from two or more data sources. Consequently, adapting well-known join algorithms to MapReduce is not as straightforward as one might hope, and MapReduce programmers often use simple but inefficient algorithms to perform equi-joins.

Leveraging the rich studies of join algorithms in parallel and distributed database systems, this paper demonstrates the implementation of several well-known join strategies in MapReduce and conducts comprehensive experiments to compare these join techniques on this new platform. To the best of our knowledge, this paper is the first to conduct a systematic experimental comparison of a broad set of join algorithms on the MapReduce platform. The contributions of this paper are as follows:

1. We provide a detailed description of several equi-join implementations for the MapReduce framework. We show that it is possible, but not always straightforward, to implement the well-known join strategies efficiently as vanilla MapReduce jobs.

2. For each algorithm, we design various practical preprocessing techniques to further improve the join performance at query time. Unlike previous work in this area [27], our join algorithms do not require any modification to the basic MapReduce framework and can be readily used by existing MapReduce platforms like Hadoop.

3. We conduct an extensive experimental evaluation to compare the various join algorithms on a 100-node Hadoop cluster. First, a detailed breakdown of a MapReduce join algorithm is provided to reveal the overheads inherent in the MapReduce framework. Then, we thoroughly demonstrate the tradeoffs among the different join algorithms and the benefit that preprocessing can provide.

4. Our results show that the tradeoffs on this new platform are quite different from those found in a parallel RDBMS, due to deliberate design choices that sacrifice performance for scalability in MapReduce. An important trend is the development of declarative query languages [10, 11, 15, 24] that sit on top of MapReduce. Our findings provide an important first step for query optimization in these languages.

The rest of the paper is organized as follows. We provide an overview of the MapReduce framework in Section 2. Then Section 3 describes our equi-join implementations for log processing in MapReduce. Next, we present our experimental results in Section 4. Finally, related work is discussed is Section 5, and we conclude in Section 6.

2. MAPREDUCE OVERVIEW

Programming in MapReduce is fairly easy. A user only needs to define a map and a reduce function. The map function takes a key-value pair (K, V) as the input and generates some other pairs of (K', V') as the output. The reduce function takes as the input a $(K', LIST\_V')$ pair, where $LIST\_V'$ is a list of all V' values that are associated with a given key K'. The reduce function produces yet another key-value pair as the output. Typically, both the input and the output of a MapReduce job are files in a distributed file system, such as the Google file system [19]. An overview of MapReduce is shown in Figure 1.

When a MapReduce job is launched on a cluster of n nodes, a *job-tracker* creates a total of m map tasks and r reduce tasks. As a rule of thumb [16], m and r are often set to be $10 \times n$ and n, respectively. Each map task works on a non-overlapping partition (called a *split*) of the input file. Typically, a split corresponds to a block in the DFS. Each map task M_i, $i = 1, \ldots, m$ does the following: first, it reads the assigned file split; second, it converts the raw bytes into a sequence of (K, V) pairs according to an "interpreter" associated with the file format; and finally it calls the user-defined map function with each (K, V) pair. The output (K', V') pairs are first partitioned into r chunks $b_{i,1}, \ldots, b_{i,r}$, one for each reduce task. The partitioning function can be customized, but has to guarantee that pairs with the same key are always allocated to the same chunk. Next, the pairs in each chunk are sorted by K'. Finally, the sorted chunks are written to (persistent) local storage.

Each reduce task R_j needs to fetch the j^{th} chunk outputted by every map task before calling the reduce function. The job-tracker keeps track of the state of every task. On the completion of a map task M_i, it informs each reduce task R_j, which then pulls over the j^{th} chunk $b_{i,j}$ remotely. This is referred to as the *shuffle* phase in MapReduce. Note that the shuffling of one map output can happen in parallel with the execution of another map task. After all map tasks have finished and all the chunks $b_{1,j}, \ldots, b_{m,j}$ have been copied over, each R_j merges its chunks and produces a single list of (K', V') pairs sorted by K'. The reduce task then generates $(K', LIST\_V')$ pairs from this list and invokes the user-defined reduce function to produce the final output.

MapReduce has several built-in features desirable in a large-scale distributed environment, as described below.

First, MapReduce exploits function-shipping to reduce the network overhead. Each map task is preferred to be scheduled on a node that stores a copy of the input data split locally.

Second, a MapReduce job is fault-tolerant. The DFS replicates every data block multiple times on different nodes. Thus, the input file can survive node failure. If a map task fails, the job-tracker will schedule a new one on a different node to redo the same work. Similarly, a new reduce task is scheduled if one fails. Note that because each map task stores its output persistently in the local storage, the failure of a reduce task does not trigger the re-execution of any completed map task, as long as the local storage is still accessible.

Finally, MapReduce supports load-balancing. The tasks for a given job are not assigned to the cluster all at once. Each node is given a certain number of initial tasks. Every time a node finishes a task, it asks for more work from the job-tracker. This way, faster nodes naturally get more tasks than slower ones. Such a design also enables better performance under failures: if a node fails, tasks assigned to this node are spread over the surviving nodes in the cluster.

Note that compared to traditional RDBMSs where path length is carefully optimized, MapReduce has more overheads because of its support for schema flexibility, fault-tolerance and load-balancing. The philosophy of MapReduce is to concern less about the per-node efficiency, but to focus more on scalability instead. As long as the framework scales, any moderate performance loss on a single node can be compensated by simply employing more hardware.

Basic strategy	Preprocessing technique
standard repartition join, improved repartition join	directed join with pre-partitioned L and R
broadcast join	pre-replicate R
semi-join	pre-filter R
per-split semi-join	pre-filter R

Table 1: MapReduce-based join algorithms studied in this paper.

3. JOIN ALGORITHMS IN MAPREDUCE

In this section, we describe how to implement several equi-join algorithms for log processing in MapReduce. We use the MapReduce framework as is, without any modification. Therefore, the support for fault tolerance and load balancing in MapReduce is preserved.

Problem Statement: We consider an equi-join between a log table L and a reference table R on a single column, $L \bowtie_{L.k=R.k} R$, with $|L| \gg |R|$. (Multi-way joins are briefly discussed in Section 3.5.) We assume that both L and R as well as the join result are stored in DFS. For simplicity, we assume that there are no local predicates or projections on either table. Extending our methods to relax these assumptions is straightforward. Finally, we assume that scans are used to access L and R. Incorporating indexes in MapReduce is an important problem, but is left for future work.

Our equi-join algorithms for MapReduce borrow from the research literature on join processing in shared-nothing parallel RDBMSs [17] and distributed RDBMSs [14]. Table 3 summarizes the equi-join strategies that we study. For each strategy, we consider further improving its performance with some preprocessing techniques. Although these join strategies are well-known in the RDBMS literature, adapting them to MapReduce is not always straightforward. We provide crucial implementation details of these join algorithms in MapReduce. The declarative query languages appearing on top of MapReduce, such as Pig [24] from Yahoo!, Hive [10] from Facebook, and Jaql [11] from IBM, already use some of these equi-join strategies but often implement them in a less efficient way. We will summarize the implementation differences as we proceed. In addition, we provide an experimental comparison between our join algorithms and those implemented in Pig in Section 4.4.

We assume that each map or reduce task can optionally implement two additional functions: *init()* and *close()*. These are called before and after each map or reduce task, respectively. We also assume that the default partitioning function in a map task assigns each output pair to a reducer according to a hash of the output key K'. Some of our join algorithms will customize this partitioning function as well as how records are grouped in the reducer. Sometimes, we

need neither the key nor the value when outputting a pair. In that case, we simply output a *null*. A MapReduce job can be configured to be map-only, i.e., no reduce function is specified. In that case, the sort and the merge phase are bypassed and the output of each map task is stored as a separate file directly in the DFS, instead of in the local file system.

3.1 Repartition Join

Repartition join is the most commonly used join strategy in the MapReduce framework. In this join strategy, L and R are dynamically partitioned on the join key and the corresponding pairs of partitions are joined.

Standard Repartition Join: This join strategy resembles a partitioned sort-merge join in the parallel RDBMS literature. It is also the join algorithm provided in the contributed join package of Hadoop (org.apache.hadoop.contrib.utils.join).

The standard repartition join can be implemented in one MapReduce job. In the map phase, each map task works on a split of either R or L. To identify which table an input record is from, each map task tags the record with its originating table, and outputs the extracted join key and the tagged record as a (*key*, *value*) pair. The outputs are then partitioned, sorted and merged by the framework. All the records for each join key are grouped together and eventually fed to a reducer. For each join key, the reduce function first separates and buffers the input records into two sets according to the table tag and then performs a cross-product between records in these sets. The pseudo code of this algorithm is provided in Appendix A.1.

One potential problem with the standard repartition join is that all the records for a given join key from both L and R have to be buffered. When the key cardinality is small or when the data is highly skewed, all the records for a given join key may not fit in memory. Variants of the standard repartition join are used in Pig [24], Hive [10], and Jaql [11] today. They all suffer from the same problem that all records from the larger table L may have to be buffered.

Improved Repartition Join: To fix the buffering problem of the standard repartition join, we introduce the improved repartition join with a few critical changes. First, in the map function, the output key is changed to a composite of the join key and the table tag. The table tags are generated in a way that ensures records from R will be sorted ahead of those from L on a given join key. Second, the partitioning function is customized so that the hashcode is computed from just the join key part of the composite key. This way records with the same join key are still assigned to the same reduce task. The grouping function in the reducer is also customized so that records are grouped on just the join key. Finally, as records from the smaller table R are guaranteed to be ahead of those from L for a given join key, only R records are buffered and L records are streamed to generate the join output. The detail implementation of this algorithm is shown in Appendix A.2.

The improved repartition join fixes the buffering problem in the standard version. However, both versions include two major sources of overhead that can hurt performance. In particular, both L and R have to be sorted and sent over the network during the shuffle phase of MapReduce.

Preprocessing for Repartition Join: The shuffle overhead in the repartition join can be decreased if both L and R have already been partitioned on the join key before the join operation. This can be accomplished by pre-partitioning L on the join key as log records are generated and by pre-partitioning R on the join key when it is loaded into the DFS. Then at query time, matching partitions from L and R can be directly joined.

In contrast to a parallel RDBMS, it is not possible to guarantee that the corresponding partitions from L and R are collocated on the same node. This is because the DFS makes independent decisions on where to store a particular data block. Therefore, at query time, we have to rely on the directed join strategy. The directed join algorithm is implemented as a map-only MapReduce job. Each map task is scheduled on a split of L_i. During the initialization phase, R_i is retrieved from the DFS, if it's not already in local storage, and a main-memory hash table is built on it. Then the map function scans each record from a split of L_i and probes the hash table to do the join. Note that the number of partitions can be chosen so that each R_i fits in memory. The pseudo code of this algorithm is provided in Appendix A.3.

Note that the map-side join provided in the Hadoop join package (org.apache.hadoop.mapred.join) is similar to the directed join algorithm. However, the map-side join requires that all records in each partition of the input tables are strictly sorted by the join key. In addition, this algorithm buffers all the records with the same join key from all input tables in memory. In the case of skewed L table, the map-side join can easily run out of memory. In contrast, the directed join algorithm only buffers the partition from the small table R, thus can handle data skew in L.

3.2 Broadcast Join

In most applications, the reference table R is much smaller than the log table L, i.e. $|R| \ll |L|$. Instead of moving both R and L across the network as in the repartition-based joins, we can simply broadcast the smaller table R, as it avoids sorting on both tables and more importantly avoids the network overhead for moving the larger table L.

Broadcast join is run as a map-only job. On each node, all of R is retrieved from the DFS to avoid sending L over the network. Each map task uses a main-memory hash table to join a split of L with R.

In the init() function of each map task, broadcast join checks if R is already stored in the local file system. If not, it retrieves R from the DFS, partitions R on the join key, and stores those partitions in the local file system. We do this in the hope that not all partitions of R have to be loaded in memory during the join.

Broadcast join dynamically decides whether to build the hash table on L or R. The smaller of R and the split of L is chosen to build the hash table, which is assumed to always fit in memory. This is a safe assumption since a typical split is less than 100MB. If R is smaller than the split of L, the init() function is used to load all the partitions of R into memory to build the hash table. Then the map function extracts the join key from each record in L and uses it to probe the hash table and generate the join output. On the other hand, if the split of L is smaller than R, the join is not done in the map function. Instead, the map function partitions L in the same way as it partitioned R. Then in the close() function, the corresponding partitions of R and L are joined. We avoid loading those partitions in R if the corresponding partition of L has no records. This optimization is useful when the

domain of the join key is large. Appendix A.4 shows the details of the broadcast join implementation.

Note that across map tasks, the partitions of R may be reloaded several times, since each map task runs as a separate process. We discuss opportunities for reducing this overhead in Section 4.5.

Pig also has an implementation of broadcast join. In Section 4.4, we compare our broadcast join with the Pig version.

Preprocessing for Broadcast Join: Although there is no way to precisely control the physical placement of replicas in the DFS, by increasing the replication factor for R we can ensure that most nodes in the cluster have a local copy of R. This can enable broadcast join to avoid retrieving R from the DFS in its init() function.

3.3 Semi-Join

Often, when R is large, many records in R may not be actually referenced by any records in table L. Consider Facebook as an example. Its user table has hundreds of millions of records. However, an hour worth of log data likely contains the activities of only a few million unique users and the majority of the users are not present in this log at all. For broadcast join, this means that a large portion of the records in R that are shipped across the network (via the DFS) and loaded in the hash table are not used by the join. We exploit semi-join to avoid sending the records in R over the network that will not join with L. To the best of our knowledge, this is the first description of a semi-join implementation on MapReduce.

The semi-join implementation has three phases, each corresponding to a separate MapReduce job (pseudo code is provided in Appendix A.5).

The first phase of semi-join is run as a full MapReduce job. In the map function, a main-memory hash table is used to determine the set of unique join keys in a split of L. A hash table is used because MapReduce does not directly support hash-based aggregation. By sending only the unique keys to the map output, the amount of data that needs to be sorted is decreased. The reduce task simply outputs each unique join key. One reducer is used to consolidate all the unique keys into a single file $L.uk$, which we assume is small enough to fit in memory.

The second phase of semi-join is similar to the broadcast join and is run as a map-only job. The init() function loads $L.uk$ into a main-memory hash table. Then the map() function iterates through each record in R and outputs it if its join key is found in $L.uk$. The output of this phase is a list of files R_i, one for each split of R.

Finally, in the third phase of semi-join, all the R_i are joined with L, using the broadcast join previously described.

Although semi-join avoids sending the records in R over the network that will not join with L, it does this at the cost of an extra scan of L. In Section 4, we evaluate whether the extra scan of L actually pays off.

Preprocessing for Semi-Join: It is possible to move the first two phases of semi-join to a preprocessing step and only execute the last phase at query time. This preprocessing can even be done incrementally. For example, as new records in L are generated, the unique join keys of these records can be accumulated and joined with R to determine the subset of R that will join with L. Note that unlike pre-partitioning, this preprocessing technique does not move the log data.

3.4 Per-Split Semi-Join

One problem with semi-join is that not every record in the filtered version of R will join with a particular split L_i of L. The per-split semi-join is designed to address this problem.

The per-split semi-join also has three phases, each corresponding to a separate MapReduce job. The first and the third phases are map-only jobs, while the second phase is a full map-reduce job. The first phase generates the set of unique join keys in a split L_i of L and stores them in the DFS file $L_i.uk$. In the second phase, the map function loads all records from a split of R into a main-memory hash table. The close function of the map task reads the unique keys from each $L_i.uk$ file and probes the hash table for matching records in R. Each matched record is outputted with a tag R_{L_i}, which is used by the reduce function to collect all the records in R that will join with L_i. In the final phase, the file for R_{L_i} and L_i are joined using the directed join. The implementation details of the per-split semi-join algorithm can be found in Appendix A.6.

Compared to the basic semi-join, the per-split semi-join makes the third phase even cheaper since it moves just the records in R that will join with each split of L. However, its first two phases are more involved.

Preprocessing for Per-split Semi-join: Like the basic semi-join, the per-split semi-join algorithm can also benefit from moving its first two phases to a preprocessing step.

3.5 Discussion

Preprocessing Considerations: The pre-partitioning technique for directed join often has limited applicability, since L can only be physically partitioned one way. This can become a problem if L needs to be joined with different reference tables on different join keys. Other preprocessing techniques do not have this limitation because they are insensitive to how L is partitioned. Also, pre-partitioning typically has a higher cost than the preprocessing used in semi-join and per-split semi-join since the larger table L has to be shuffled. Finally, the pre-replication technique described for broadcast join becomes infeasible when R or the size of the cluster is large.

Multi-Way Joins: So far, our discussion has been limited to single joins. But of course in practice, L may be joined with several reference tables on different join keys. Repartition join would need to run a separate MapReduce job to join each reference table in this case. In contrast, it is fairly easy to extend broadcast join, semi-join and per-split semi-join to process multiple reference tables at the same time.

4. EXPERIMENTAL EVALUATION

All our experiments were run on a 100-node cluster. Each node had a single 2.4GHz Intel Core 2 Duo processor with 4GB of DRAM and two SATA disks. Red Hat Enterprise Server 5.2 running Linux 2.6.18 was used as the operating system. The 100 nodes were spread across two racks, and each rack had its own gigabit Ethernet switch. The rack level bandwidth is 32Gb/s. Under full load, we observed about 35MB/s cross-rack node-to-node (full-duplex) bandwidth.

We used Hadoop version 0.19.0 and configured it to run up to two map and two reduce tasks concurrently per node. Thus, at any point in time, at most 200 map tasks and 200 reduce tasks could run concurrently in our cluster. We configured Hadoop DFS (HDFS) to stripe across both disks on

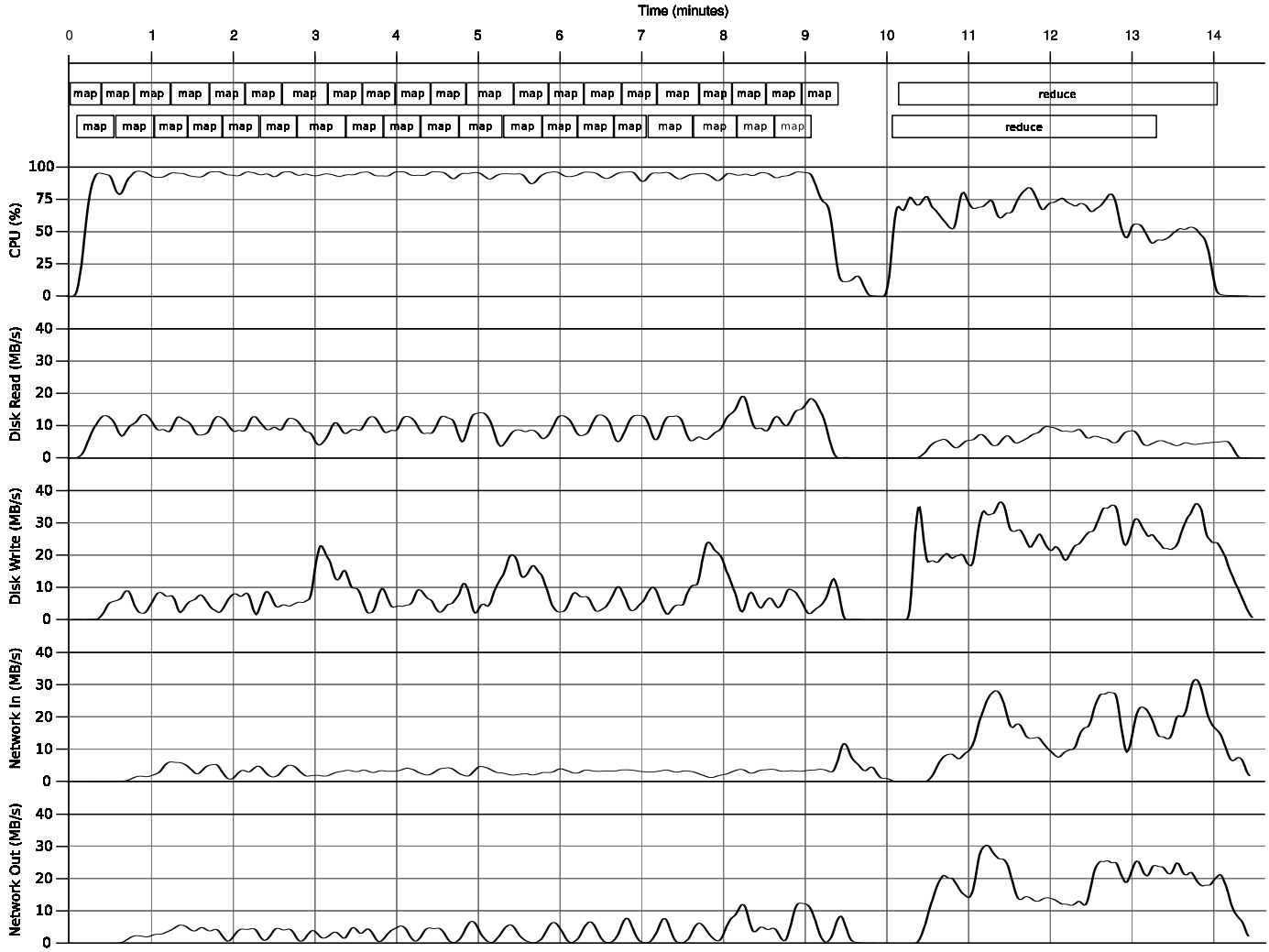


Figure 2: System behavior on a single node during the execution of the standard repartition join algorithm. Two concurrent map and two concurrent reduce tasks can run on a node at the same time.

each node and use a 128MB block size. For fault tolerance, each HDFS block was replicated three times.

We implemented all the join algorithms listed in Table 3 in Section 3. Although the tradeoffs of similar join algorithms have been studied in database systems, as we shall see, those results are not the same in the MapReduce environment. For instance, because of the overheads in MapReduce, savings in network bandwidth may not be as important.

4.1 Datasets

In our experiments, we use synthetic datasets to simulate an event log L being joined to some sort of user information R. Here, both L and R were stored as HDFS files and striped across the cluster. Each record in L had 100 bytes on average, whereas each record in R had exactly 100 bytes. The join result is a 10-byte join key, a 10-byte column from L, and a 5-byte column from R. Columns that were not needed were removed as early as possible in join processing. The size of L was fixed at 500GB, while the size of R varied from 100K records (roughly 10MB) to 1000M records (roughly 100GB).

The join between L and R is designed to be the typical n-to-1 join, with one or more records in L referencing exactly one record in R. To simulate a common case in log analysis where many users are inactive over the measurement period, we fixed the fraction of R that was referenced by L to be 0.1%, 1%, or 10%. Note that in this setting all the records in L always appeared in the join result, whereas only a fraction of R appeared in the result. To simulate some users being more active than others, a Zipf distribution was used to determine the frequency of each reference key appearing in L. We studied the cases where the Zipf skew factor was 0 (uniform) and 0.5. In both settings, the referenced R keys are assigned randomly to records in L.

4.2 MapReduce Time Breakdown

To understand the performance of the join algorithms discussed in this paper, one needs to first understand what transpires during the execution of a MapReduce job and the overhead of various execution components of MapReduce. As a representative example, we inspect the execution of the standard repartition join algorithm from the perspective of a single node in the cluster. This join is performed

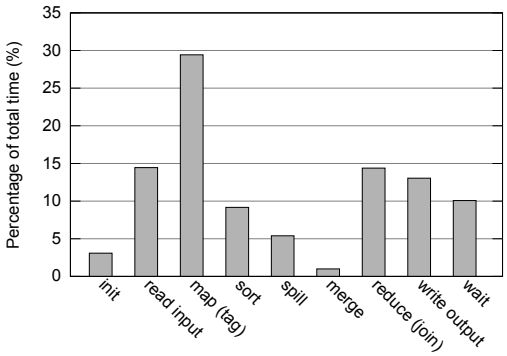


Figure 3: Time breakdown.

on the 500GB log table and a 30MB reference table with 1% actually referenced by the log records. This MapReduce job had about 4000 map tasks and 200 reduce tasks. On average, a node was assigned 40 map and 2 reduce tasks. In Figure 2, we show the CPU, disk and network activities on a single node, with respect to the current state of the job.

This breakdown leads to several interesting observations. First, the map phase was clearly CPU-bound, while the reduce phase was limited by the network bandwidth when writing the three copies of the join result to HDFS. Second, while the map tasks were running, the disk and the network activities were moderate and periodic. The peaks were related to the output generation in the map task and the shuffle phase in the reduce task. Third, notice that the node was almost idle for about 30 seconds between the 9 minute and 10 minute mark. During this time, the reduce tasks were simply waiting for the slowest map task to finish as the reduce function cannot be called until all map tasks on all the nodes have completed. Finally, Figure 2 also illustrates how the MapReduce framework, despite its simplicity, is able to use all available system resources. By enabling independent and concurrent map tasks, and by using a loose coupling between map and reduce phases, almost all CPU, disk and network activities can be overlapped. Moreover, this is done in a way that can dynamically adapt to workload skew or failures.

To get an idea of the cluster-wide behavior during the same time, Figure 3 further subdivides the map and reduce tasks into smaller operations, showing their average performance over *all* nodes in the cluster. On average, about 65% of the total time was spent in the map task and about 25% in the reduce task. Each node spent about 10% of the time waiting. Operations such as sort, spill, and merge (which can be avoided depending on the join algorithm) accounted for 17% of the time. While some of the overheads will be reduced as Hadoop matures, the following four overheads are inherent in the MapReduce framework. (1) For fault tolerance, the full output of each map task is checkpointed to local storage. (2) To accommodate schema flexibility, the framework adds the overhead of interpreting each input record at runtime. (3) By leveraging commodity storage, MapReduce has to maintain data consistency in the software layer. In HDFS, by default, a checksum is created for every 512 bytes of data and is verified on every read. (4) For better load balancing and failure performance, many relatively small tasks are created and each task always has some initialization cost. Note that checkpointing is mainly I/O overhead, whereas the latter three are all CPU overheads.

	Parameter	Our cluster	Facebook
N	# nodes per rack	50	40
C	# concurrent map task per node	2	8
B	uplink bandwidth (Gb/s) per rack	16	4
P	output rate (MB/s) per map task	1.3	N/A

Table 2: Parameters and typical values

The overheads listed above could change the conventional wisdom that network is often the bottleneck in a distributed system. Now, we analytically estimate when the network is actually saturated in terms of the rate that each map task generates its output. Suppose that we have a number of parameters given in Table 4.2. Each node generates PC mega bytes of data per second. Most of the data has to be sent to other nodes. The cross-rack network bandwidth per node is B/N giga bits per second. So, in order to saturate the network, we need:

$$PC > \frac{1024B}{8N} \Rightarrow P > \frac{128B}{NC}$$

Plugging in the parameters of our cluster listed in Table 4.2, each map task has to produce data at approximately 20MB/s to saturate the network. This rate is much higher than the rate of about 1.3MB/s that we observed in Figure 2. Table 4.2 also listed a configuration of the Facebook environment [21]. There, the network is the bottleneck as long as P is above 1.6MB/s, very close to our observed rate.

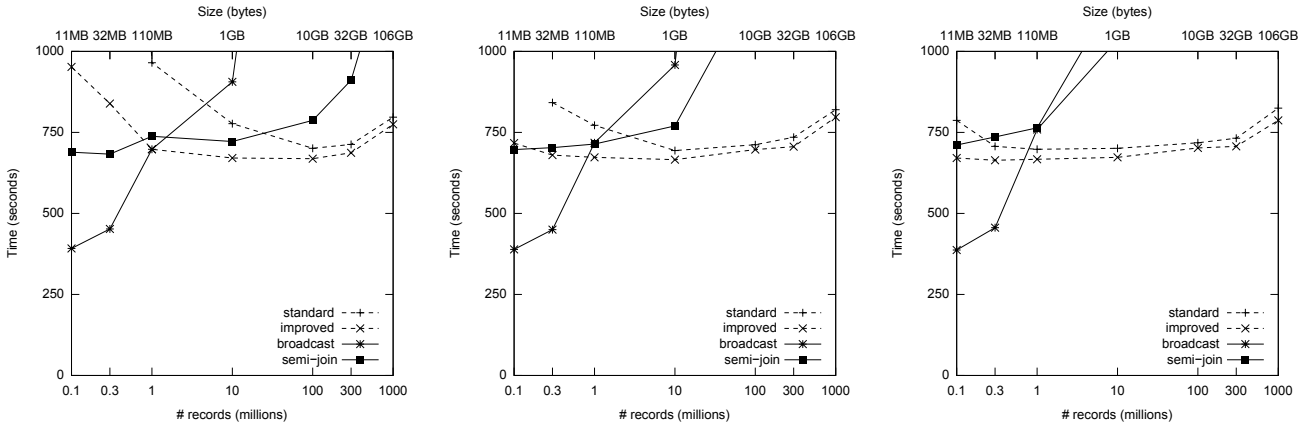
4.3 Experimental Results

We now present the performance results for the different join algorithms described in this paper. First, we consider the case where no preprocessing is done for the join between L and R. Then we consider the case where some preprocessing is done to improve the join performance. Finally, we evaluate the scalability of these join algorithms.

Note that the results in this section omit the overhead of writing the join output to HDFS. Not only does this help highlight the differences in the join algorithms, but it also simulates the common case where the join output is aggregated into a much smaller dataset immediately after the join.

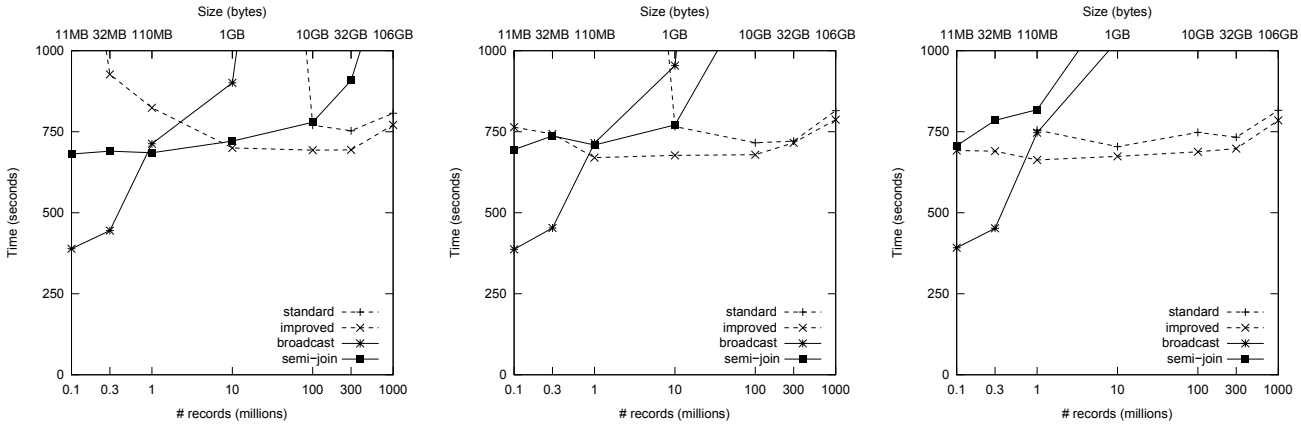
No Preprocessing: In Figure 4, we present the results for the join algorithms on the uniform dataset. Each graph corresponds to a different fraction of R being referenced. The x-axis shows the size of R in records and bytes in log scale. The y-axis shows the elapsed time for each join algorithm in seconds.

Let's start with the standard repartition join, denoted as *standard*, in Figure 4(a). Moving from right to left, we observe that as R got smaller the time for the standard repartition join decreased initially (as expected), but then surprisingly increased. This is because, as R got smaller, there were more records in L with the same join key. And since the standard join has to buffer all the records in L with the same join key, this caused a large number of objects to be allocated in memory. In some extreme cases, the standard repartition join did not finish because it ran out of memory. Similar patterns can be observed in Figure 4(b) and 4(c).



(a) Referencing 0.1% of R (b) Referencing 1% of R (c) Referencing 10% of R

Figure 4: Uniform distribution of join key in L, no preprocessing.



(a) Referencing 0.1% of R (b) Referencing 1% of R (c) Referencing 10% of R

Figure 5: Skewed distribution of join key in L, no preprocessing.

# records in R	improved	broadcast
0.3 million	145 GB	6 GB
10.0 million	145 GB	195 GB
300.0 million	151 GB	6,240 GB

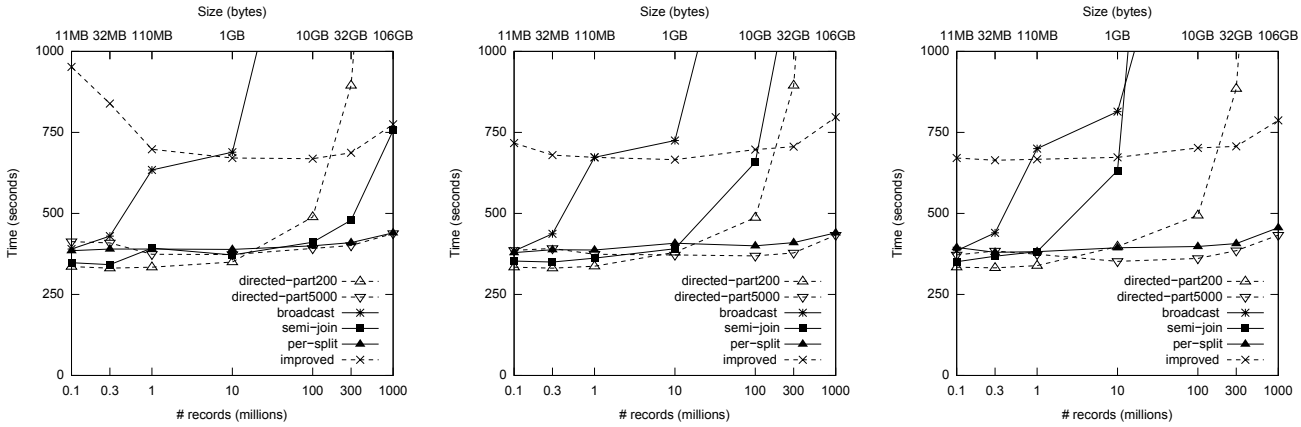
Table 3: Amount of data moved over the network for the uniform dataset and with 1% of R referenced.

As expected, the improved repartition join, denoted as *improved*, always performed better than the standard repartition join, since it did not buffer L in memory. However, the time for the improved repartition join still increased as R got smaller. This is because eventually there was not enough join keys to distribute the work evenly among the reduce tasks. For example, when R had 100K records and 0.1% of it was referenced, there were only 100 unique join keys in L. Since all the records in L with the same join key were sent to a single reduce task, about half of the 200 reduce tasks had no work to do while the other half received twice the average workload.

Next, we consider the broadcast join. Its performance improved as the size of R and the fraction that was referenced decreased. When R had about 1M records or less, it was always better than the improved repartition join algorithm. However, broadcast join's relative performance degraded rapidly as R got bigger. This is because the cost of transferring R to every node across the network and loading it in memory started to dominate. Table 3 compares the amount of data transferred in the broadcast join and the improved repartition join. In Section 4.5, we will explore ways to increase the applicable range for the broadcast join.

Turning to semi-join, it was never the best algorithm. This is mainly because of the relatively high overheads of scanning L from HDFS, as listed in the previous section. Therefore, the extra scan of L required by semi-join negated any performance advantage it might have had. Finally, we do not show the results for the per-split semi-join, since it was always worse than semi-join due to a more expensive second phase.

The results for the skewed dataset are shown in Figure 5. The graphs follow similar patterns to those with the uniform dataset. The main difference is that the performance of the standard repartition join degraded faster than that of the improved repartition join as the size and the fraction



(a) Referencing 0.1% of R (b) Referencing 1% of R (c) Referencing 10% of R

Figure 6: Uniform distribution of join key in L, with preprocessing.

of referenced R got smaller. This is because the skewed dataset amplified the problems discussed earlier with these two algorithms.

With Preprocessing: Often, as records are generated for L or R, some preprocessing can be done on those records to improve join performance. In this section, we study how much improvement can be achieved by the preprocessing versions of the join algorithms. We also include the results of the improved repartition join (without preprocessing) to serve as a baseline.

Directed join is performed on pre-partitioned L and R. It is tricky to determine the right number of partitions. We experimented with 200 and 5000 partitions, referred as *directed-part200* and *directed-part5000* respectively, the former to match the number of cores in our cluster and the latter to ensure most partitions fit in memory. The preprocessing step of broadcast join replicates R to every node in the cluster. For both semi-join and per-split semi-join, we assume that their first two phases are done during preprocessing and only the last phase is executed at query time.

The preprocessing results on the uniform dataset are shown in Figure 6. Results for the skewed dataset are omitted, since they were similar to those for the uniform dataset. Looking at the graphs in Figure 6, going from left to right, we see that as the size of R increased, broadcast join degraded the fastest, followed by directed-part200 and semi-join. Per-split semi-join was almost as efficient as directed-part5000 and both tolerated data skew and reference ratio changes well. This mainly resulted from working with smaller hash tables on R that fit in memory. In general, compared to the improved repartition join, preprocessing lowered the time by almost 60% (from about 700 to 300 seconds) over a wide range of settings.

Finally, the cost of preprocessing also varies. The average preprocessing time for semi-join, per-split and prepartition-5000 are 5, 30, and 60 minutes, respectively. The former two are cheaper since they do not move L across the network.

Scalability: In this section, we explore the performance impact of increasing the join input size linearly with the number of nodes in the cluster. For our scalability experiment, we used L with uniform distribution and referencing 1% of R. We fixed R to have 10 million records and varied the amount of data in L by maintaining 5GB per node. We then scaled the number of nodes in the cluster from 5 to 100. The results are shown in Figure 7. We see that all algorithms scaled almost linearly. These algorithms naturally spawn off more map and reduce tasks as the number of nodes increases, and they use partitioned parallelism which has robust scalability properties [17].

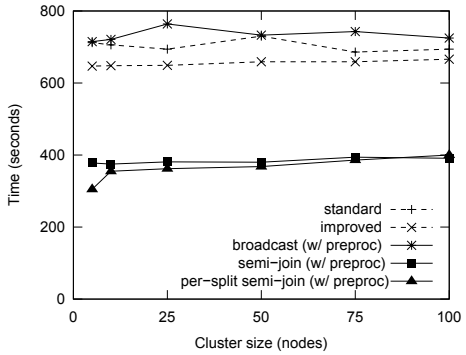


Figure 7: Results of scale-up experiment (uniform join key distribution, 1% of R referenced)

4.4 Comparison with Join Algorithms in Pig

A few join methods have been explored in the declarative frameworks on MapReduce, like Pig [24], Hive [10] and Jaql [11]. In this section, we compare our join algorithms to those in Pig. Note that the contribution of this paper is to investigate various parallel/distributed join techniques on the MapReduce platform, thus the insights from our work can be directly used by declarative frameworks like Pig.

There are two join strategies provided in Pig: repartition join and fragment replicate join. They resemble our improved repartition join and broadcast join, respectively.

Table 4.4 shows a direct comparison between the improved repartition join and the Pig repartition join (in Pig version 0.2) at a few selected points (in the interest of space we only show a few representative results). These results consistently show a more than 2.5X speedup with our improved

	# records in R	improved	pig-2.0
uniform L	0.3 million	669	1805
ref 1% R	300 million	670	1819
skewed L	0.3 million	803	2089
ref 0.1% R	300 million	706	1764

Table 4: Improved repartition join vs. Pig repartition join

repartition join over the Pig repartition join, for both uniform and skewed distribution of join keys in L and various sizes of R with different percentage being referenced by L. Besides the Pig runtime overhead, the following implementation choices in Pig may contribute to the difference in performance. First, the Pig repartition join treats both input sources the same, therefore it may have to buffer records from the larger input L (although this algorithm can spill L to disk if necessary). Second, while our improved repartition join employs a customized partitioning function, the Pig repartition join uses an extra secondary sort in the reduce phase to guarantee that records of the same key are ordered by their input tags. This secondary sort introduces extra overhead to this join algorithm.

We also compare our broadcast join against the fragment replicate join in Pig. For 0.3 million records in R, the broadcast join is consistently more than 3 times faster than the fragment replicate join, on both uniform and skewed L referencing 0.1% and 1% of R. Our broadcast join is more efficient because all map tasks on the same node share one local copy of R, whereas the fragment replicate join always re-reads R from DFS in every map task. In addition, our broadcast join dynamically selects the smaller input (R or the split of L) for the in-memory hash table, whereas Pig always loads the full R in memory.

4.5 Discussion

Performance Analysis: As shown from our experiments, the performance differences among the various join strategies are within a factor of 3 in most cases, instead of an order of magnitude that one might expect in traditional parallel or distributed RDBMSs. As listed in Section 4.2, MapReduce has more built-in computational overheads such as input record interpretation, checksum validation and task initialization. Because of these overheads and the relative high network bandwidth in our environment, the shuffling cost in repartition-based joins did not get highlighted. In other environments with much lower network bandwidth, like the cluster used in Facebook (see Table 4.2), the network costs in repartition-based joins will become more prominent. As a result, the performance gap among various join strategies would be larger.

Customized Splits: For broadcast join, every map task has the fixed overhead of loading the reference data in memory. If load balancing and failure performance are not important, one way to amortize such overhead is to make each map task bigger. Instead of each map task working on a single DFS block, we can assign multiple blocks to a map task. If we simply group several consecutive blocks into a split, the blocks are likely spread over different nodes. As a result, we lose data locality when scheduling the map task. To preserve locality, we customized the function that generates splits in Hadoop, so that multiple non-consecutive blocks co-located on the same node are grouped into one logical split, which

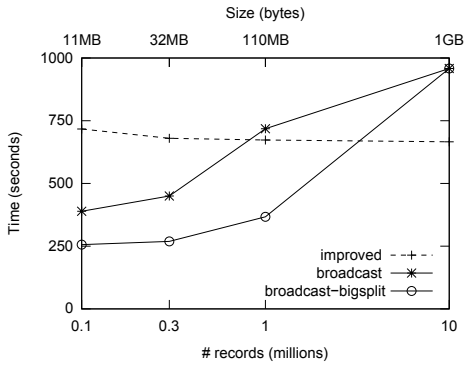


Figure 8: Minimizing initialization overhead (uniform join key distribution, 1% of R referenced)

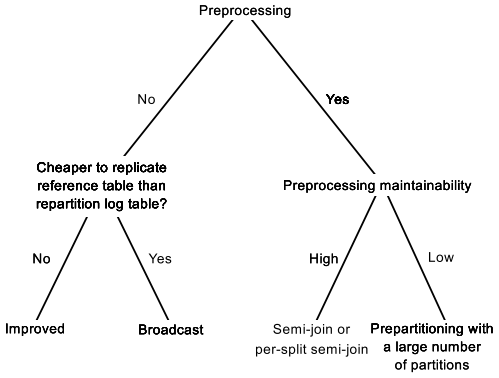


Figure 9: The tradeoffs of various join strategies summarized in a decision tree.

we call a big split. In our experiment, we grouped 5 blocks into a big split.

Figure 8 shows the effect of using big splits for some selected join algorithms. The elapsed time of broadcast join was reduced by up to 50%, when using big splits. Without big splits, each node had to load the reference data in memory about 40 times. With big splits, each node only loaded the reference data 8 times. In contrast, the performance improvement for the improved repartition join was only about 3%, because using big splits only saved some initialization cost in the join algorithm. We note that there is limitation when using big splits, since a larger split size could adversely affect load balancing.

Choosing the Right Strategy: Figure 9 summaries the tradeoffs of the studied join strategies into a decision tree. Based on statistics, such as the relative data size and the fraction of the join key referenced (these statistics could be collected on a sample of the inputs), this decision tree tries to determine what is the right join strategy for a given circumstance.

If data is not preprocessed, the right join strategy depends on the size of the data transferred via the network. If the network cost of broadcasting table R to every node is less expensive than transferring both R and projected L, then the broadcast join algorithm should be used.

When preprocessing is allowed, semi-join, per-split semi-join and directed join with enough partitions are the best choices. Semi-join and per-split semi-join offer further flex-

ibility since their preprocessing steps are insensitive to how the log table is organized, and thus suitable for any number of reference tables. In addition, the preprocessing steps of these two algorithms are cheaper since there is no shuffling of the log data.

Although crude, these guidelines shown in Figure 9 provide an important first step for query optimization in the high level query languages on MapReduce, such as Pig, Hive and Jaql.

5. RELATED WORK

There is a rich history of studying join algorithms in parallel and distributed RDBMSs [12, 13, 14, 26]. An interested reader can refer to surveys on these algorithms [20, 22].

A more recent work [27] proposes extending the current MapReduce interface with a merge function. While such an extension makes it easier to express a join operation, it also complicates the logic for fault-tolerance. In our work, we choose not to change the existing MapReduce interface for join implementation.

Lately, there have been several efforts in designing high level query languages on MapReduce. This includes Pig [24], Hive [10], and Jaql [11], all of which are open source. They differ in the underlying data model and query syntax. Our work is synergistic with such efforts by providing guidelines on the choice of different join algorithms.

In [23], the authors discussed three join strategies (also included in our study) that can be chosen by the suggested optimizer in Pig. However, this work did not provide any experimental result on the performance of these strategies. In contrast, our study considered a broader set of join algorithms and preprocessing strategies, contained crucial implementation details, and conducted a comprehensive experimental evaluation.

6. CONCLUSIONS AND FUTURE WORK

Joining log data with all kinds of reference data in MapReduce has emerged as an important part of analytic operations for enterprise customers, as well as Web 2.0 companies. This paper has leveraged over three decades of work in the database community to design a series of join algorithms on top of MapReduce, without requiring any modification to the actual framework. Our design revealed many details that make the implementation more efficient. We have evaluated the join methods on a 100-node system and shown the unique tradeoffs of these join algorithms in the context of MapReduce. We have also explored how our join algorithms can benefit from certain types of practical preprocessing techniques. The valuable insights obtained from our study can help an optimizer select the appropriate algorithm based on a few data and manageability characteristics.

There are a number of directions for future work, including evaluating our methods for multi-way joins, exploring indexing methods to speedup join queries, and designing an optimization module that can automatically select the appropriate join algorithms. In addition, this paper reveals the limitations of the MapReduce programming model for implementing joins, thus another important future direction is to design new programming models to extend the MapReduce framework for more advanced analytic techniques.

Acknowledgments

This work is supported in part by the Microsoft Jim Gray Systems Lab and the NSF award CRI-0707437.

7. REFERENCES

[1] http://www.slideshare.net/cloudera/hw09-data-processing-in-the-enterprise.
[2] http://www.slideshare.net/cloudera/hw09-large-scale-transaction-analysis.
[3] http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun.
[4] http://www.slideshare.net/cloudera/hw09-hadoop-based-data-mining-platform-for-the-telecom-industry.
[5] http://wiki.apache.org/hadoop/PoweredBy.
[6] http://developer.yahoo.net/blogs/theater/archives/2009/06/hadoop_summit_hadoop_and_the_enterprise.html.
[7] http://www.slideshare.net/prasadc/hive-percona-2009.
[8] http://hadoop.apache.org/.
[9] http://research.yahoo.com/files/facebook-hadoop-summit.pdf.
[10] http://hadoop.apache.org/hive/.
[11] http://www.jaql.org.
[12] Teradata: DBC/1012 data base computer concepts and facilities, Teradata Corp., Document No. C02-0001-00, 1984.
[13] P. A. Bernstein and N. Goodman. Full reducers for relational queries using multi-attribute semijoins. In *Symp. On Comp. Network*, 1979.
[14] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, 1981.
[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] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[17] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6), 1992.
[18] D. J. DeWitt and M. Stonebraker. MapReduce: A major step backwards. Blog post at *The Database Column*, 17 January 2008.
[19] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.
[20] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), 1993.
[21] J. Hammerbacher. Managing a large Hadoop cluster. Presentation, Facebook Inc., May 2008.
[22] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1), 1992.
[23] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.
[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.
[25] 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*, 2009.
[26] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD*, 1989.
[27] 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, 2007.

APPENDIX

A. PSEUDO CODE OF MAPREDUCE JOIN ALGORITHMS

We assume that the inputs to all the following algorithms are (key, value) pairs, where the value V is bound to a full record of R or L, and the input key K is simply null.

A.1 Standard Repartition Join

Map (K: null, V: a record from a split of either R or L)
  $join\_key \leftarrow$ extract the join column from V
  $tagged\_record \leftarrow$ add a tag of either R or L to V
  emit $(join\_key, tagged\_record)$

Reduce (K': a join key,
  $LIST\_V'$: records from R and L with join key K')
  create buffers B_R and B_L for R and L, respectively
  for each record t in $LIST\_V'$ **do**
    append t to one of the buffers according to its tag
  for each pair of records (r, l) in $B_R \times B_L$ **do**
    emit (null, $new\_record(r, l)$)

A.2 Improved Repartition Join

Map (K: null, V: a record from a split of either R or L)
  $join\_key \leftarrow$ extract the join column from V
  $tagged\_record \leftarrow$ add a tag of either R or L to V
  $composite\_key \leftarrow (join\_key, tag)$
  emit $(composite\_key, tagged\_record)$

Partition (K: input key)
  $hashcode \leftarrow hash\_func(K.join\_key)$
  return $hashcode \bmod \#reducers$

Reduce (K': a composite key with the join key and the tag
  $LIST\_V'$: records for K', first from R, then L)
  create a buffer B_R for R
  for each R record r in $LIST\_V'$ **do**
    store r in B_R
  for each L record l in $LIST\_V'$ **do**
    for each record r in B_R **do**
      emit (null, $new\_record(r, l)$)

A.3 Directed Join

Init ()
  if R_i not exist in local storage **then**
    remotely retrieve R_i and store locally
  $H_{R_i} \leftarrow$ build a hash table from R_i

Map (K: null, V: a record from a split of L_i)
  probe H_{R_i} with the join column extracted from V
  for each match r from H_{R_i} **do**
    emit (null, $new\_record(r, V)$)

A.4 Broadcast Join

Init ()
  if R not exist in local storage **then**
    remotely retrieve R
    partition R into p chunks $R_1..R_p$
    save $R_1..R_p$ to local storage

  if $R <$ a split of L **then**
    $H_R \leftarrow$ build a hash table from $R_1..R_p$
  else
    $H_{L_1}..H_{L_p} \leftarrow$ initialize p hash tables for L

Map (K: null, V: a record from an L split)
  if H_R exist **then**
    probe H_R with the join column extracted from V
    for each match r from H_R **do**
      emit (null, $new\_record(r, V)$)
  else
    add V to an H_{L_i} hashing its join column

Close ()
  if H_R not exist **then**
    for each non-empty H_{L_i} **do**
      load R_i in memory
      for each record r in R_i **do**
        probe H_{L_i} with r's join column
        for each match l from H_{L_i} **do**
          emit (null, $new\_record(r, l)$)

A.5 Semi-Join

Phase 1: Extract unique join keys in L to a single file $L.uk$
  Map (K: null, V: a record from an L split)
    $join\_key \leftarrow$ extract the join column from V
    if $join\_key$ not in $unique\_key\_table$ **then**
      add $join\_key$ to $unique\_key\_table$
      emit ($join\_key$, null)

  Reduce (K': a unique join key from table L,
    $LIST\_V'$: a list of null)
    emit (K', null)

Phase 2: Use $L.uk$ to filter referenced R records;
generate a file R_i for each R split
  Init ()
    $ref\_keys \leftarrow$ load $L.uk$ from phase 1 to a hash table

  Map (K: null, V: a record from an R split)
    $join\_col \leftarrow$ extract join column from V
    if $join\_col$ in ref_keys **then**
      emit (null, V)

Phase 3: Broadcast all R_i to each L split for the final join

A.6 Per-Split Semi-Join

Phase 1: Extract unique join keys for each L split to $L_i.uk$
  Map (K: null, V: a record from an L split L_i)
    $join\_key \leftarrow$ extract the join column from V
    if $join\_key$ not in $unique\_key\_table$ **then**
      add $join\_key$ to $unique\_key\_table$
      emit ($join\_key$, null) (output file called $L_i.uk$)

Phase 2: Use $L_i.uk$ to filter referenced R;
generate a file R_{L_i} for each L_i
  Map (K: null, V: a record from an R split)
    $join\_col \leftarrow$ extract join column from V
    $H_R \leftarrow$ add $(join\_col, V)$ to a hash table

  Close () (for map task)
    $files \leftarrow$ all $L_i.uk$ files from the first phase
    for each file $L_i.uk$ in $files$ **do**
      for each key k in $L_i.uk$ **do**
        $result \leftarrow$ probe H_R with key k
        emit (R_{L_i}, $result$)

  Reduce (K': R_{L_i}, $LIST\_V'$: all R records for R_{L_i})
    write (null, $LIST\_V'$) to file R_{L_i} in DFS

Phase 3: Directed join between each R_{L_i} and L_i pair