Abstract

Distributed join algorithms are becoming increasingly important as databases spanning multiple servers and queries running on shared-nothing systems find increasing use in order to meet rising requirements. We present several join algorithms in the easily theoretically analyzable parallel computing models MPC and BSP, then we consider how they relate to MapReduce and, more specifically, the implementation of MapReduce in Hadoop, a popular distributed computation framework. Based on these we describe and compare the major MapReduce join algorithms in theory and perform an experimental comparison on a Hadoop cluster to determine the strengths and weaknesses of the different implementations in practice. Beside input size, the handling of data skew is a major issue we address. Our results show that each algorithm is the most effective in a different situation and that all of them need to be chosen from to compute the join optimally in arbitrary situations. While Hadoop is in theory equally powerful as MPC, we come to the conclusion that the implementation specifics of the framework restricts the realization of join algorithms.

Contents

CHAPTER 1

Introduction

Join operations are an essential feature of most database systems and various algorithms have been developed to optimize their performance on single machines and more recently on single machines with multiple cores. However, with the rising popularity of decentralized and shared-nothing "big data" systems the requirements have shifted and new distributed algorithms were needed. While single-core and multi-core speed and memory utilization were the most important metrics on a single machine, communication cost and effective coordination between nodes have become the deciding factors for join performance on decentralized systems.

The MapReduce model and its open-source implementation Hadoop was originally developed for batch processing and applications similar to PageRank on large clusters of inexpensive hardware rather than to serve the same purposes as typical RDBMSs.[1] On top of Hadoop's distributed file system (*HDFS*) and MapReduce framework many widely-used systems have emerged that are again very similar to a RDBMS even implementing SQL. As a consequence multiple algorithms and optimizations of joins for the MapReduce model were implemented in software such as Hive, Impala, Presto and Pig.

Although the popularity of MapReduce and distributed shared-nothing architectures has only risen recently, methods for computing joins on more general and less restricted architectures were already known for longer. The more general parallel computation models BSP (*bulk synchronous parallel*) and MPC (*massively parallel computation*) simplify analysis of algorithms and more thought has already been put into optimizing joins in them compared to MapReduce.

Surprisingly, although MapReduce has been around since 2004 and the first version of Hadoop was released in 2006, there is still substantial room for optimization of joins in all implementations. Less common types of joins such as inequality joins tend to still be completely unoptimized. This can possibly be explained by the effective horizontal scaling of MapReduce which may lead to more machines being added instead of the algorithm being optimized.

All code written in the scope of this thesis and data generated experimentally can be accessed online. For the code see the public git repository:

https://github.com/arselzer/mapreduce_join_comparison. For the data see data/.

CHAPTER 2

Preliminaries

2.1 Traditional Single-Processor Join Algorithms

Before looking at distributed join algorithms the older commonly known algorithms employed in traditional database systems and their strengths and weaknesses are presented. They are also directly relevant to distributed join algorithms since they might process a part of the join on each node and the choice of per-node algorithm will make a difference in the total performance.

2.1.1 Nested-Loop Join

The *nested-loop join* is the most straight-forward way of computing a join.

The binary inner equi-join $R \bowtie_{R.a=S.b} S$ is computed as follows by the nested-loop join algorithm:

```
for each $r$ in $R$ do
    for each $s$ in $S$ do
        if $r.a = s.b$ then
            Add $(r, s)$ to the result set
        end
    end
end
```

This algorithm will have a time complexity of $O(|R| \cdot |S|)$. Memory consumption can be near zero if the tables are not loaded into memory and otherwise depends on buffering.

The *block nested-loop join* is a common optimization of the nested-loop join that can greatly improve the performance if one of the tables fits into memory by decreasing the amount of disk reads and seeks.[2]

The *nested-loop join* is the simplest join algorithm and it can be applied to arbitrary join conditions instead of only equi-joins while its obvious disadvantage is the performance that will not scale well at all with input size.

2.1.2 Indexed Nested Loop Join

Assuming the relation S is indexed, an indexed join can be used to compute $R \bowtie_{R.a=S.b} S$.

```
for each $r$ in $R$ do
    $s \leftarrow lookup(r.a)$
    if $s$ was found then
        Add $(r, s)$ to the result set
    end
end
```

If a B-Tree index is used the algorithm will run in $O((|R| + |S|) \cdot \log(|R| + |S|))$.

2.1.3 Hash Join

The *hash join* uses a hash map to increase the performance of the Nested-Loop Join by eliminating the outer loop. First the smaller table is loaded into the hash map then the other table is scanned and matching rows can be determined by a lookup in the hash map. The tables can be referred to as the *"build table"* (the one loaded into the hash map) and the *"probe table"* (the one being scanned each time).[3] The actual values will also need to be compared due to the possibility of collisions.

The join can be computed as follows (where $|R| < |S|$):

```
Build a hash table $HT$ of $R$ with the hash function $H(x)$
for each $s$ in $S$ do
    if $HT$ contains $H(s)$ and $HT(H(s)) = s$ then
        Add $HT(H(s)), s)$ to the result set
    end
end
```

The time cost of the algorithm is $O(|R| + |S|)$. Whenever possible, the hash table is fully loaded into memory leading to a memory consumption approximately equal to the size of the smaller table. A disadvantage of the hash join is that it can only be applied to equi-join problems.

There are further optimizations such as the *grace hash join*, which is an improvement over the simple hash join that is faster if the amount of memory available is less than the square root of the smaller table and the *hybrid hash join* which combines the simple hash join and the grace hash join.

2.1.4 Sort-Merge Join

In the *sort-merge join* both tables are first sorted and then merged. The sorting step is the most expensive (on average $O((|R| + |S|) \cdot \log(|R| + |S|))$ in the case of quicksort) while the cost of merging is low ($O(|R| + |S|)$). This effectively leads to a average-case performance of $O((|R| + |S|) \cdot \log(|R| + |S|))$.

The algorithm is very effective if a table is already sorted by the attribute being joined (usually a primary or foreign key) because the sorting step can be skipped. If the tables are already stored in sorted order, which is very common, only a merge join has to be performed and the algorithm is most likely preferable to a hash join which first has to build a hash table.

However, if the data is not already sorted, no index is available, and its size greatly exceeds the memory size of the machine, sorting can be very expensive.

A form of the sort-merge join can be expressed in pseudocode as follows:

```
Sort relation $R$ by attribute $a$ and $S$ by attribute $b$
$r \leftarrow next(R)$
$s \leftarrow next(S)$
while $\neg empty(R) \land \neg empty(S)$ do
    if $r.a < s.b$ then
        $r \leftarrow next(R)$
    end
    else if $r.a > s.b$ then
        $s \leftarrow next(S)$
    end
    else
        $T \leftarrow \{\}$
        $a \leftarrow r.a$
        while $\neg empty(S) \land s.b = a$ do
            $T \leftarrow T \cup s$
            $s \leftarrow next(S)$
        end
        while $\neg empty(R) \land r.a = a$ do
            for $each\ t\ in\ T$ do
                Add $(r, t)$ to the result set
            end
            $r \leftarrow next(R)$
        end
    end
end
```

CHAPTER 3

Distributed Computation Models

Since the complexity of distributed algorithms can quickly become overwhelming several models have been developed to simplify their design and analysis by abstracting over lower-level details and introducing restrictions.

3.1 BSP

Perhaps the most well-known such model is the BSP (Bulk Synchronous Parallel) model published 1990 by Leslie Valiant.[4] BSP algorithms run in a series of supersteps where each superstep consists of a communication phase followed by a computation phase. The nodes (frequently referred to as processors) are barrier synchronized between supersteps. Hence the slowest node determines the time needed for the computation phase of the superstep and the largest amount of data sent by one or to one node will determine the time needed for the communication phase.

To achieve a simple abstraction BSP restricts locality with barrier synchronization (waiting for all processors to complete their current task). This has the effect that it will not be an effective model in some domains where locality (computation and communication with only partial synchronization or any non-local memory accesses outside of the communication phase) is important. [5] Several extensions of BSP have been developed which allow this at the cost of increased complexity.[6]

BSP defines the following parameters:

p The number of processors

S The number of rounds

$\mathbf{h_i}$ The "h-relation" - the maximum amount of data (words) sent or received by a processor in the communication phase during superstep i

$\mathbf{w_i}$ The maximum time cost of computation of a processor in the computation phase during superstep i

g The time cost of transferring one word of data

l The "start-up latency" or cost of barrier synchronization

When $W = \sum_{i=1}^{S} w_i$ and $H = \sum_{i=1}^{S} h_i$, the total cost of the algorithm is then given by

$$\sum_{s=1}^{S} w_s + g \sum_{s=1}^{S} h_s + Sl = W + gH + Sl \tag{3.1}$$

Alternative cost models have been proposed for BSP such as the following which allows overlapping computation and communication:[5]

$$\sum_{s=1}^{S} \max\{w_s, gh_s\} + Sl \tag{3.2}$$

g and l are platform-dependent parameters. They depend on the network topology, number of nodes, clock speed and further factors.[6] g is determined by the network's bandwidth and latency (packet delivery time). It is hard to determine in practice since nodes in a network are usually not arranged in topologies that are easy to analyze and predictable such as the hypercube topology and congestion can occur in parts of the network. g can be estimated empirically for a network if necessary. The worst-case lower bound of l is given by the network diameter (the worst-case number of hops between two nodes in a network). The bisection bandwidth is the bandwidth available between any two partitions in the network.

Network Type	**l**	**g**	**Diameter**	**Bisection Bandwidth**
2D Mesh	$\Omega(\sqrt{p})$	$O(\sqrt{p})$	$2(\sqrt{p}-1)$	$\sqrt{p}$
Hypercube	$\Omega(\log p)$	$O(\log p)$	$\log_2 p$	$p/2$
Butterfly	$\Omega(\log p)$	$O(\log p)$	$\log_2 p$	$p/2$

In the original paper by Valiant et al.[4] a further parameter L (synchronization periodicity) is given which is needed for the form of synchronization described in the paper. However, other types of synchronization can also be used and we will ignore L for simplicity.

The following diagrams illustrate a BSP superstep. Here, w_i^j stands for the computation time of processor j in superstep i and h_i^j for the communication costs of the same processor.

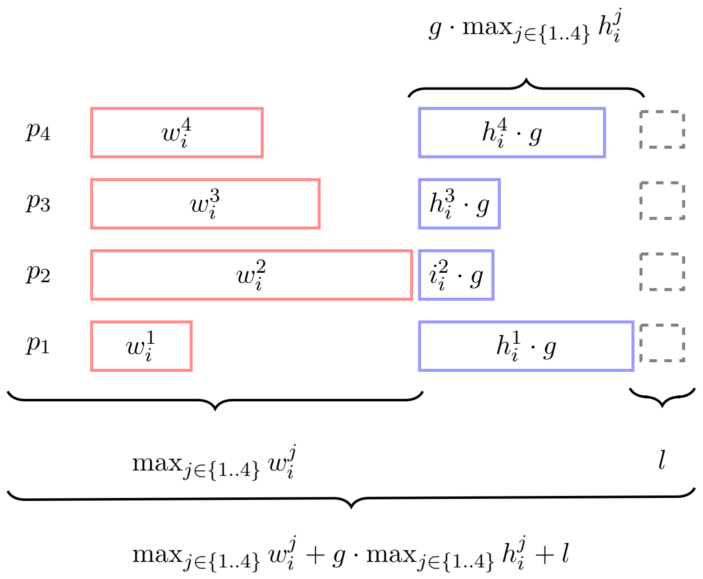

$g \cdot \max_{j\in\{1..4\}} h_i^j$
p_4
w_i^4
$h_i^4 \cdot g$
p_3
w_i^3
$h_i^3 \cdot g$
p_2
w_i^2
$i_i^2 \cdot g$
p_1
w_i^1
$h_i^1 \cdot g$
$\max_{j\in\{1..4\}} w_i^j$
l
$\max_{j\in\{1..4\}} w_i^j + g \cdot \max_{j\in\{1..4\}} h_i^j + l$

Figure 3.1: The phases of a BSP superstep

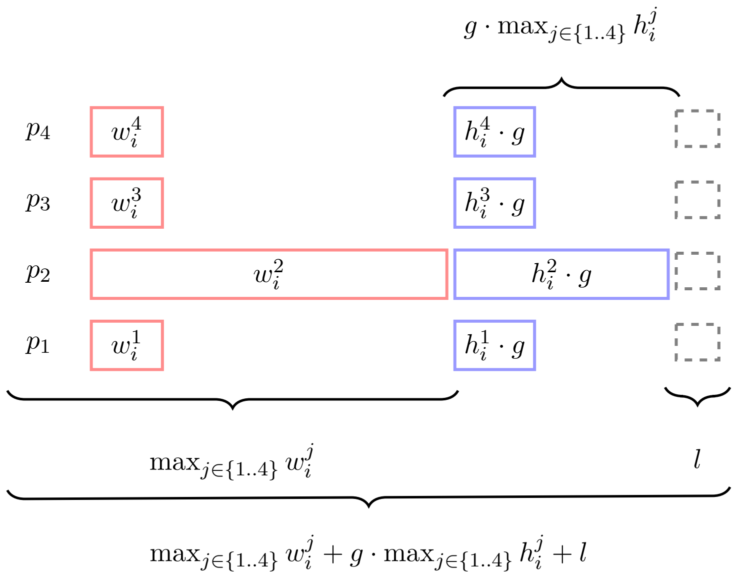

$g \cdot \max_{j\in\{1..4\}} h_i^j$
p_4
w_i^4
$h_i^4 \cdot g$
p_3
w_i^3
$h_i^3 \cdot g$
p_2
w_i^2
$h_i^2 \cdot g$
p_1
w_i^1
$h_i^1 \cdot g$
$\max_{j\in\{1..4\}} w_i^j$
l
$\max_{j\in\{1..4\}} w_i^j + g \cdot \max_{j\in\{1..4\}} h_i^j + l$

Figure 3.2: BSP with high skew

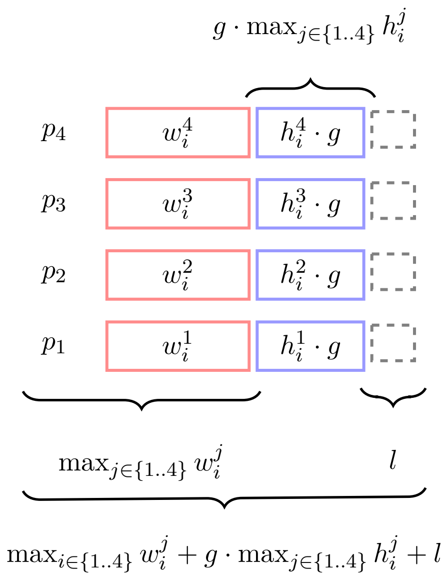

$g \cdot \max_{j\in\{1..4\}} h_i^j$
p_4
w_i^4
$h_i^4 \cdot g$
p_3
w_i^3
$h_i^3 \cdot g$
p_2
w_i^2
$h_i^2 \cdot g$
p_1
w_i^1
$h_i^1 \cdot g$
$\max_{j\in\{1..4\}} w_i^j$
l
$\max_{i\in\{1..4\}} w_i^j + g \cdot \max_{j\in\{1..4\}} h_i^j + l$

Figure 3.3: BSP with no skew

3.2 MPC

The MPC (Massively Parallel Computation) model is a simplification of the BSP model introduced by Koutris et al. It serves the purpose of providing an even further simplified but still realistic model for modern shared-nothing architectures such as MapReduce.[7] It focuses on the minimization of both the number of rounds and the load per node during each round.

An MPC algorithm is described by the following parameters:

p The number of processors

r The number of rounds

$\mathbf{L_u^k}$ The load of processor u during round k

L The maximum load: $\max_{u=1}^{p} \max_{k=1}^{r} L_u^k$

C The total communication: $\sum_{u=1}^{p} \sum_{k=1}^{r} L_u^k$

IN The size of the input data

OUT The size of the output data

The input data is initially distributed uniformly across processors such that $L_u^0 = {}^{IN}/_{p}$.

A further distinction can be made between the *stateless* and *stateful* MPC model. In the stateless model all data is erased after each round while the stateful model allows data to be kept locally on processors over multiple rounds. The stateless model can simulate the stateful model when each processor sends the data it wants to keep to itself during each round. If the number of rounds is constant with problem size ($r = O(1)$) then $L' = O(L)$ where L is the load of stateful MPC and L' is the load of a stateless simulation of the stateful algorithm.

The two goals of MPC algorithm design are (1) the minimization of the number of rounds and (2) the minimization of the load per processor. In the ideal case $L = {}^{IN}/_{p}$.

As opposed to the BSP model, the total cost of computation is not specified in the MPC model. The computation time is not featured in the MPC model and is specific to the system, algorithm and load. While the simplicity of MPC has disadvantages, MPC still tends to be accurate at predicting the performance of algorithms on shared-nothing systems.[7]

3.3 MapReduce in Hadoop

Hadoop's implementation of MapReduce is one of multiple implementations of the more general MapReduce concept originally introduced and popularized by Google together with the Google File System (GFS).[1] Another major implementation of MapReduce is Google's proprietary implementation famously used to solve the PageRank problem. Hadoop has the advantage of being open-source and freely available. The implementation specifics in Hadoop can have effects on the performance of MapReduce algorithms so we will focus on MapReduce in Hadoop.

3.3.1 The HDFS

The *Hadoop distributed filesystem* (HDFS) is a filesystem originally developed for Hadoop and MapReduce which is now even more broadly used. The HDFS follows the design of the GFS since it was designed to provide a missing open-source component for a full MapReduce system. The major design goals are support of very large files and reliability on commodity (inexpensive) hardware. The HDFS achieves this by replication and distribution across a large cluster. Files are broken up into chunks of per default 128MB and replicated three times across the cluster to prevent data loss if a server fails. The block size is intentionally significantly larger than in traditional file systems (e.g. ext4: 1-64KiB) due to the different use case of sequential read-only access on large files.

3.3.2 MapReduce Jobs

If the data to be used is not already in the HDFS the client will generally first have to copy it there (an exception being Hadoop running in standalone mode). The client application submits a request containing the job to the resource manager. The MapReduce job includes several parameters such as the amount of mappers and reducers. Map tasks are then distributed among the cluster taking into consideration data locality (machines and racks). Reduce tasks are assigned only after a percentage of map tasks have completed (with the default being 5%).

3.3.3 Mapping Stage

The map function $M : \langle k, v \rangle \rightarrow [\langle k, v \rangle, ...]$ maps one key-value pair to any number of key-value pairs.

Hadoop's *Mapper* class provides a similar interface. Outputs are generated using the *write* method of the context.

```
protected void map(KEYIN key, VALUEIN value, Context context)
```

Map tasks repeatedly call the map function on a chunk of data. They will be assigned HDFS blocks as input, but if the split size is set to a different value than the block size, the file will be partitioned into differently-sized chunks when sent to the mappers. For example, consider a 256MB HDFS file. Due to the default block size, the file will be broken up into two blocks that will likely reside on two different HDFS nodes. If no split size is defined, it will be processed by two mappers. If a split size of 256MB is given, it will be be processed by one mapper and if one of 64MB is given, it will be sent to four mappers.

3.3.4 Shuffling Stage

Mappers have a circular buffer that is filled as the map task progresses. At a certain threshold, before the buffer would be filled, the content is grouped by key, sorted (using quicksort) and spilled to disk.

If a combiner function is specified, it is also applied to the mapper's output before being spilled to disk. The combiner function is an optimization that is used on the map side to reduce the output that would otherwise have to be sent over the network, which tends to be more costly than performing a reduce task on the mapper. Often the reduce function can be reused as the combiner function.

A partitioner function will determine where the pairs of a key are sent. Per default a simple deterministic hash function is used - the *HashPartitioner*:

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
  public int getPartition(K key, V value, int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }
}
```

This partitioner will in many cases cause values to be distributed uniformly across reducers. However, it fails at achieving an even distribution when there are key values occuring frequently. For example, if the keys were English words randomly sampled from books and the hashing partitioner is used, it can be expected that "the" will make up approximately 7% of all keys. All "the" keys will be sent to the same reducer. If there are 10 reducers, the remaining 93% of words will be split up so that, on average, each reducer would receive 9.3% of them. However, one unlucky reducer will additionally receive all "the" keys, making it responsible for on average 16.3% of all keys, leading to a total load (and in the case of WordCount, runtime) of 75% more than average. As the number of reducers is increased the effect becomes even more extreme. If all keys of the same value do not need to end up at the same reducer a random partitioner can be utilized to handle the problem of skew. The random partitioner will not form a strict total order of keys among reducers, which is the case when for tuples $\langle k_i, v_i \rangle$ and a total order of keys where $i < j \Rightarrow k_i \leq k_j$ it is not guaranteed that for all reducers where R_k is a list of keys assigned to reducer k $R_i[x] < R_j[y]$ for any x and y.

The reduce tasks will receive the output of the mapper tasks, which are already sorted lists. These can then be efficiently merged in $O(n)$ time by the reducer tasks. The sort algorithm employed by Hadoop is therefore a hybrid form of quicksort and merge sort.

3.3.5 Reducing Stage

The reduce function $R : \langle k, [v, ..] \rangle \rightarrow \langle k, v \rangle$ reduces a key and several corresponding values to a single value.

```
public void reduce(Key key, Iterable<IntWritable> values,
Context context)
```

After the map and shuffle steps have completed, all the output from the mappers is available in sorted form at the reducer it was sent to. Thus, a linear scan can be performed to group values by key and pass them to the reduce function.

A MapReduce algorithm might have to run for multiple rounds like some of the BSP algorithms we have seen previously. When this is the case more than one job will be created and the output of the reducers of the first job will become the input to the mappers of the second. Chaining jobs is however relatively expensive in Hadoop.

Due to the processing of keys in sorted order in reducers the output of each reducer will also be sorted. However, whether there is a sorting order over reducers or not will depend on the partitioning before the reducing stage.[1][8]

3.3.6 The Cost of MapReduce

Goodrich et al.[9] give a lower bound for the running time of a MapReduce algorithm. The following parameters determine the running time:

L The latency of the shuffle network

B The bandwidth of the shuffle network

R The number of rounds

t_r The maximum internal running time of a mapper or reducer during round r

t The total internal running time $\sum_{r=0}^{R-1} t_r$

C_r The communication complexity of round r

C The total communication complexity $\sum_{r=0}^{R-1} C_r$

The (worst-case) lower bound for the total running time T is given by:

$$T = \Omega(\sum_{r=0}^{R-1}(t_r + L + {}^{C_r}/_{B})) = \Omega(t + RL + {}^{C}/_{B}) \tag{3.3}$$

The reason that only a lower bound can be given is that the implementation of MapReduce may introduce additional asymptotic costs. Hadoop only supports grouping through sorting adding $\Theta(\frac{C \log C}{p})$ to T:

$$T_{\text{hadoop}} = \Omega(t + RL + {}^{C}/_{B} + \frac{C \log C}{p}) \tag{3.4}$$

A more complete cost model is given in [10] and [11].

3.3.7 Skew and MapReduce

Skewed runtime tends to be one of the major weaknesses of MapReduce applications. A skew can result in a non-uniform distribution of data or work across mappers or reducers causing the slowest one to run significantly longer than the fastest one and limiting the total speed (wall clock time) of the MapReduce job while leaving nodes running idle. This frequent phenomenon was named "the curse of the last reducer" due to the whole job having to wait until the last reducer has completed.[12] A distinction can be made between several causes of skew.[13]

Map-Side: Expensive Record One or several key-value inputs take significantly longer to process than the average input. For example a node of a graph with a large degree.

Map-Side: Heterogeneous Maps Mappers process different datasets as a single input without the same computation time leading to a bimodal distribution of the running time of mappers. For example, two tables R and S might be processed by mappers. However, transforming the tuples of S takes twice as long as those of R. As a consequence, the mappers assigned solely tuples of S will also take twice as long while the other mappers are idle half of the time.

Map-Side: Non-Homomorphic Map When mappers run tasks keeping state over multiple map calls - which would usually be done by reducers, but is sometimes more efficient in the mapping phase - a skew dependent on the order and values of the inputs may occur.

Reduce-Side: Partitioning Skew Data is unevenly partitioned across reducers. If hash partitioning is used there is no guarantee that pairs are distributed equally in number among reducers.

Reduce-Side: Expensive Input Similar to the *expensive record* skew in mappers, this type of skew occurs if a $\langle k, [v, ..]\rangle$ pair is expensive to compute either due to expensive values or the number of values.

In the case of joins *partitioning skew* and *expensive input* skew of reducers are most relevant. Partitioning skew is the easier type of skew to handle. Myung et al. describe three methods of implementing skew-resistant Reduce-Side Joins by providing alternatives to the default hash partitoning.[14] Swapping the default hash partitioner for a sampling range-based partitioner is a simple and effective solution. Although Hadoop does not provide support for it, this type of skew could even be handled automatically by the MapReduce framefork. For example, Kwon et al. have described a method for handling partitioning skew automatically by identifying "stragglers" (long-running jobs) and redistributing their data.[15]

The *expensive input* skew is in general harder to prevent and solutions are specific to the algorithm. If one key occurs more than $^{IN}/_{p}$ times before the shuffle phase the skew is no longer preventable by partitioning solutions. We will encounter this problem in join algorithms.

3.4 MPC and BSP vs. MapReduce

BSP is a slightly more powerful model than MapReduce even though they are conceptually similar. The most deciding way in which BSP differs from MapReduce is that MapReduce clears the local memory of nodes after each round and transfers the data to shared space in the HDFS instead of keeping it in memory over multiple rounds. BSP supports keeping data locally over multiple rounds. The stateless MPC model is equally powerful as MapReduce. The reason that MapReduce behaves this way is that it makes achieving fault tolerance easier. Still, this is considered one of the major weaknesses of MapReduce and Hadoop, and is addressed by some alternatives to Hadoop such as Spark. Iterative algorithms often found in machine learning can benefit from a stateful model. For joins a stateless model works relatively well in general.[16]

It has been shown that all MapReduce algorithms can be simulated on the BSP model without an increase in asymptotic costs.[17] The simulation maps the map phase to a BSP superstep and the reduce phase to another.

Also, in the other direction, all BSP algorithms can also be simulated in MapReduce. When the communication costs due to the extra transfer of data caused by the clearing of memory and re-sending do not exceed the other communication costs, they can in theory be simulated in MapReduce without any loss in asymptotic efficiency[17]. In practice, however, there is a shuffle phase which usually groups by sorting and no runtime lower than log-linear time is achievable.

The simulation of BSP in MapReduce was originally described by Goodrich et al.[9] and later in more detail by Pace[17]. Mappers are too restricted to be useful to the simulation so they are set to the identity function. Each MapReduce reduce function simulates a BSP processor. It receives the input of the BSP processor it is simulating, performs the same computation and produces the same output. Any data stored locally on the BSP processor has to be stored globally in the file system and read again in the next round.

To characterize BSP algorithms that can be efficiently implemented in MapReduce an additional parameter f_i is needed which is the maximum amount of data kept locally by a processor from round i to round $i+1$ and $F = \sum_{i=1}^{S} f_i$. w_i, h_i, W, H, S and l are the familiar BSP parameters.

The lower bound for the cost of the BSP algorithm implemented in MapReduce is:

$$T = \Omega(W + (H + F)g + Sl) \tag{3.5}$$

When $H = o(F)$ (F is asymptotically dominant over H) the asymptotic cost will be higher. In the case of Hadoop's shuffle implementation which groups by sorting the cost is:

$$T = O(W + (H + F)g + (H + F)\log(H + F) + Sl) \tag{3.6}$$

3.5 Speedup and Scaleup

Speedup is a measure of parallel algorithms describing the increase in speed as more processors are added while the input size is fixed. Per the definition of speedup $S_p = \frac{T_1}{T_p}$ where p is the number of processors and T_i is the execution time on an i-processor system. The closer S_p is to p, the better parallelizable an algorithm is. In the extreme case that $S_p = 1$ it is not parallelizable. If $S_p = p$, the algorithm will run p times as fast on p

processors instead of one. Ideally speedup is linear, but in practice or when algorithms cannot be fully parallelized, it can be sub-linear.

Scaleup is the speedup with the size of the input data changing at the same rate as the number of processors. Ideally scaleup is constant, which implies that the execution time of the algorithm stays the same as the increasing input size is compensated by proportionately increasing parallelism.

A further measure that can be obtained from the speedup is efficiency. It states how efficiently processors are used: $E_p = \frac{S_p}{p}$.

CHAPTER 4

MPC Join Algorithms

4.1 The Problem

Throughout this section we will consider the problem of computing the binary inner equi-join $R \bowtie_{R.a=S.b} S$. For simplicity, we will ignore any different data types and treat all attributes as arbitrary strings.

4.2 Distributed Hash Join

A hash function h is defined that maps an attribute to $\{1, .., p\}$ and is known to all processors. In the beginning the data is distributed evenly among the processors. Each tuple is associated with the relation it comes from. The processors compute the hash function for each tuple and send it to the processor with the number $h(r.a)$ for $r \in R$ or $h(s.b)$ for $s \in S$.[7]

This algorithm runs in a single round with only one communication phase. The total communication is optimal: $C = IN$. The potentially limiting factor is the load of the processors. The load is determined by the distribution of the output of the hash function which is directly determined by the distribution of join attribute values.

As in [7], d will denote the number of occurrences of each value of the join attribute. In the extreme case of $d = 1$, all values are different and the join will be empty. In the case of $d = IN$ all values are equal and the result will be the cartesian product. If $d = {}^{IN}\!/_2$ there will be two values. The load L_u is equal to the sum of independent random variables $\sum_t X_{t,u}$ with $E[\sum_t X_{t,u}] = \frac{IN}{p}$ (for a fixed processor u) due to the random hash function.

$$P[L_u \geq (1+\delta)\frac{IN}{p}] = e^{-\frac{\delta^2 IN}{3pd}} \tag{4.1}$$

As an example, if we want to find the probability of the skew on a processor exceeding 110% of the optimal value with an input size of 1000 and 6 processors, we can calculate $P[L_u \geq (1+0.5)\frac{100000}{6}] = e^{-\frac{0.1^2 1000}{3 \cdot 6 \cdot 0.1}} = 0.003866$. A skew exceeding 110% is therefore very unlikely.

The bound for the maximum load can be derived from the load on a single processor. It is possible that a skew will occur with absolute certainty. $\frac{IN}{p}$ gives the optimal load per processor and $\frac{IN}{d}$ gives the amount of possible splits of the tuples by join attribute. When the ratio of these values to the number of processors exceeds $1+\delta$ a skew greater than $1+\delta$ will be the consequence. Therefore, under the following condition, only a skew exceeding δ is possible:

$$\frac{\frac{\frac{IN}{p}}{\frac{IN}{d}}}{p} \geq 1+\delta \implies \frac{d}{p^2} \geq 1+\delta \tag{4.2}$$

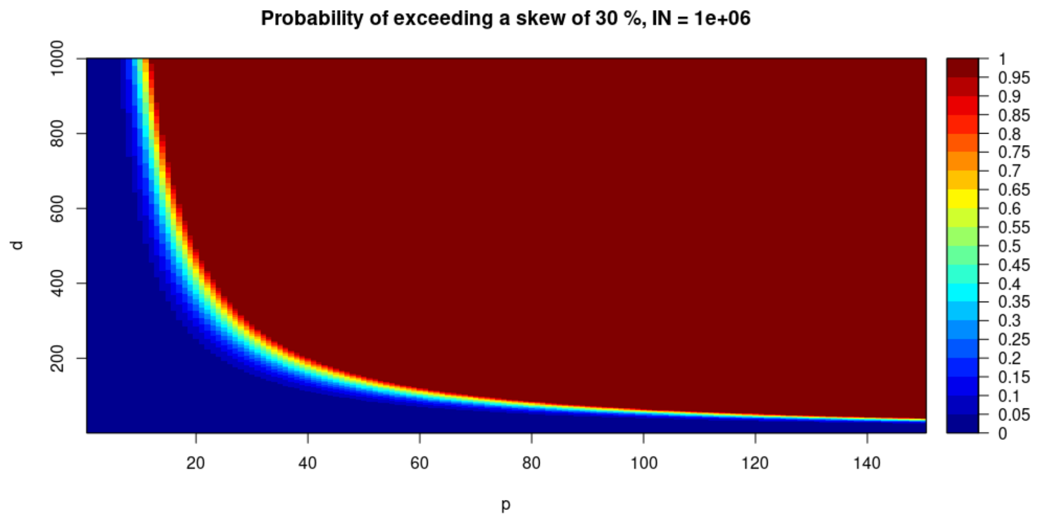
For example, if $IN = 1000000$, $d = 400$, $p = 20$ and $\delta = 0.3$, then each processor would optimally receive $\frac{1000000}{20} = 50000$ tuples and cannot receive less. The amount of unique values is $\frac{1000000}{400} = 2500$. $\frac{50000}{2500} = 20$ so if the tuples were uniformly distributed among the processors, all of the 20 processors could potentially have the same, optimal, load. There is still the possibility of a skew of less than 130%.

Assuming $IN = 1000000$, $d = 400$, $p = 30$ and $\delta = 0.3$, then each processor would optimally receive $\frac{1000000}{30} = 33333.34$ tuples. $\frac{3333.34}{2500} = 13.33$, so the optimal value could only be obtained on 13 processors. Since $\frac{30}{13.33} = 2.25$, a skew higher than 130% is inevitable.

$$P[L \geq (1+\delta)\frac{IN}{p}] = \begin{cases} 1 & \text{if } \frac{d}{p^2} \geq 1+\delta \\ pe^{-\frac{\delta^2 IN}{3pd}} & \text{else} \end{cases} \tag{4.3}$$

It follows that as the number of occurrences of a join attribute value increases or the number of processors increases, the threshold for a skew of a certain percentage increases.

Under the assumption that the computing cost is a linear function of the load, the speedup of the parallel hash join is linear.


Probability of exceeding a skew of 30 %, IN = 1e+06
d
1000
800
600
400
200
p
20
40
60
80
100
120
140
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0

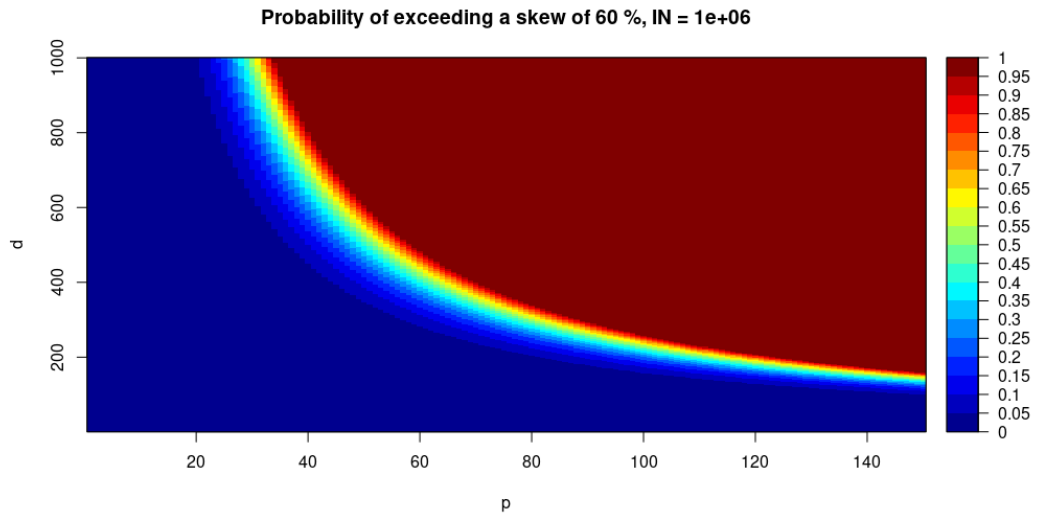

Probability of exceeding a skew of 60 %, IN = 1e+06
d
1000
800
600
400
200
p
20
40
60
80
100
120
140
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0

Figure 4.1

Algorithm 4.1: MPC Distributed Hash Join

```
1 $L \leftarrow$ a list of tuples $(t, r), t \in r, r \in R, S$
2 foreach $(t, r) \in L$ do
3     $c \leftarrow$ the join attribute of table$t$
4     Send $(t, r)$ to processor $h(t.c)$
5 end
6 Barrier synchronization
7 $M \leftarrow$ a list of received tuples$(t, r)$
8 Compute the join $R \bowtie_{R.a=S.b} S$ on the tuples in $M$
9 Barrier synchronization
```

4.3 Distributed Hash Join with Heavy Hitter Optimization

4.3.1 The Rectangle Algorithm for the Cartesian Product

In the situation of all values of the join attribute being equal, the result of the join would be the cartesian product of the both tables. The algorithm described above is however very ineffective in this case because all values are hashed to the same number and all tuples are therefore sent to the same processor leading to a load $L = IN$.

Koutras et al. have described a better distributed algorithm to compute the cartesian product, which is also a component of the skew-optimized distributed hash join.[7] As suggested by the name of the algorithm, processors are organized in a rectangle grid of size p_1 by p_2 with $p_1 \cdot p_2 = p$. Two random hash functions $h_1 : R \to \{1 \cdots p_1\}$ and $h_2 : S \to \{1 \cdots p_2\}$ are chosen. Each tuple $s \in S$ will be sent to all processors $(h_1(s), v), v \in \{1 \cdots p_2\}$ and each tuple $r \in R$ will be sent to all processors $(u, h_2(r)), u \in \{1 \cdots p_1\}$. The processors then compute the cartesian product on the incoming data. Every expected output (r, s) will be produced by the processor with coordinates $(h_1(r.a), h_2(r.b)$. The expected load of the rectangle algorithm is $L = {}^{|R|}/_{p_1} + {}^{|S|}/_{p_2}$ which is a clear improvement over the hash join algorithm.

The speedup is sub-linear and approximately equal to $\sqrt{p}$. Koutris et al. have shown that a linear speedup for the computation of the cartesian product is not possible.[7] Hence joining skewed data will unavoidably be a limiting factor to speedup and scaleup on parallel systems.

4.3.2 Detecting and Optimizing Heavy Hitters

Heavy hitters are values of the join attribute that occur very frequently. We will call any attribute occuring more than ${}^{IN}/_{p}$ times a heavy hitter like in [7] because if any attribute occurs that number of times the optimal load is no longer possible. To detect the heavy hitters, each processor counts the occurrences of the join attributes and marks any values occurring more than ${}^{IN}/_{p^2}$ times as potential heavy hitters. The counts are

Algorithm 4.2: MPC Rectangle Algorithm

```
1  $L \leftarrow$ a list of tuples $(t, r), t \in r, r \in R, S$
2  foreach $(t, R) \in L$ do
3      Send $(t, R)$ to all processors $(h(t.a), v), v \in \{1 \cdots p_2\}$
4  end
5  foreach $(t, S) \in L$ do
6      Send $(t, S)$ to all processors $(v, h(t.b)), v \in \{1 \cdots p_1\}$
7  end
8  Barrier synchronization
9  $M \leftarrow$ a list of received tuples$(t, r)$
10 Compute the cartesian product $R \times S$ on the tuples in $M$
11 Barrier synchronization
```

broadcasted and the processors determine the actual heavy hitters in the next round. Now each processor knows all heavy hitters. Next, the ordinary hash join is run on the non-heavy hitters for a round and the heavy hitters remain. Because the effort needed to compute each heavy hitter varies, each value v_i is assigned p_i processors depending on the situation. The computational effort, corresponding to the number of tuples produced, is determined by the number of occurrences in the relations R and S. When $OUT_i = |\{t \in R \mid t.a = v_i\}| \cdot |\{t \in S \mid t.b = v_i\}|$, the obviously optimal choice for p_i is

$\frac{\text{OUT}_i}{\sum_{j=1}^{p} \text{OUT}_j}$.

Algorithm 4.3: MPC Distributed Hash Join with Skew Optimization

```
1  $i \leftarrow$ the number of the current processor
2  $L \leftarrow$ a list of tuples $(t, r), t \in r, r \in R, S$
3  heavyHitters $\leftarrow$ a list
4  counts $\leftarrow \{\}$
5  foreach $(t, R) \in L$ do
6      Add $(v, 1, 0)$ to counts or update $(v, c_r, c_s)$ to $(v, c_r + 1, c_s)$
7  end
8  foreach $(t, S) \in L$ do
9      Add $(v, 0, 1)$ to counts or update $(v, c_r, c_s)$ to $(v, c_r, c_s + 1)$
10 end
11 foreach $(v, c_r, c_s)$ in counts do
12     if $c_r + c_s > {}^{IN}/_{p^2}$ then
13         Broadcast $(v, c_r, c_s)$
14     end
15 end
16 Barrier synchronization
17 otherCounts $\leftarrow$ all other potential heavy hitters identified
18 counts $\leftarrow$ counts $\cup$ otherCounts
19 Group counts by value $v$ while summing over $c_1$ and $c_2$
20 foreach $(v, c_1, c_2) \in$ counts do
21     if $c_1 + c_2 > {}^{IN}/_{p}$ then
22         $p \leftarrow c_1 \cdot c_2$ Add $(v, p)$ to heavyHitters
23     end
24 end
25 Sort heavyHitters by $p$
26 Compute the hash join on all values not in heavyHitters
27 Apply the rectangle algorithm on each value in heavyHitters
```

4.4 Distributed Sort-Merge Join

The *"Parallel Sort Join"* as described by Koutris et al. in [7] consists of a sort phase followed by a merge phase. The *parallel sort by regular sampling* (PSRS) algorithm is a good choice for the sort phase since it has low communication costs.

4.4.1 Parallel Sort By Regular Sampling

The PSRS algorithm was originally developed for parallel sorting on multiprocessors especially to address load balancing and memory/bus contention.[18] This translates to low load and communication costs in a distributed implementation. The algorithm starts with data arbitrarily partitioned across processors. Each processor sorts its list using an optimal non-parallel sorting algorithm ($O(n \log n)$). Then it determines the *regular sample* from the sorted list which is a list of $p-1$ values splitting the data into even parts. The regular sample is broadcasted and each processor merges the received samples such that all obtain the same sorted list of size $p(p-1)$. From this list $p-1$ final splitters can be obtained which all processors agree on. Next, the list of values is split at the global splitters and sent to the responsible processors The processors can merge the received lists to a sorted list and a total order is established across processors.

Algorithm 4.4: PSRS

```
1  input ← the original data
2  sample ← an empty list
3  *Sort* input *using an optimal sorting algorithm*
4  foreach $i \in \{1 \cdots p-1\}$ do
5      *Append* input[$\lfloor \frac{i \cdot \text{size(input)}}{p} \rfloor$] *to* sample
6  end
7  *Broadcast* sample
8  *Barrier synchronization*
9  *Merge received samples with* sample
10 splits ← an empty list
11 foreach $i \in \{p, 2p, \cdots (p-2)\}$ do
12     *Append* sample[$i + p/2$] *to* splits
13 end
14 partitions ← *split* input *at* splits
15 for $i$ from *1* to $p$ do
16     *Send* partitions[$i$] *to processor i*
17 end
18 *Barrier synchronization*
19 parts ← *all sorted lists received by the processor*
20 result ← *merge* parts *into a sorted list*
```

The load of PSRS can be up to IN if values might occur multiple times. If one value occurs IN times i.e. the list is only made up of that value, every value will be sent to the same processor. If the values are unique then $L \leq 2N/p$ (under the assumption that $p^3 \leq \text{IN}$ which is usually the case).[7]

The speedup is given by $\frac{L_1 \log L_1}{L_p \log L_p}$ with L_i being the load under i processors. If there are no duplicate values speedup is linear. Otherwise speedup is constant from the point onward where $\frac{\text{IN}}{p}$ is less than the maximum number of occurrences of a value.

To reduce the worst-case load of the sort-merge join values in the case of non-unique values a version with crossing values (values spanning multiple processors) can be used. PSRS can again be used with the difference that values are also tagged by the processor they initially reside on which is taken into consideration for the sort order.

Example - PSRS with duplicates and processor tags

The processors compute the regular samples:

processor	values	sorted values	selected sample
1	(1,1,1,3,3,2,2,2,2,1,1)	(1,1,**1**,1,1,**2**,2,2,**2**,3,3)	(1,2,2)
2	(1,1,1,1,3,1,1,1,2,2,2)	(1,1,**1**,1,1,**1**,1,2,**2**,2,3)	(1,1,2)
3	(3,1,1,3,3,3,2,2,3,2,2)	(1,1,**2**,2,2,**2**,3,3,**3**,3,3)	(2,2,3)
4	(1,1,1,3,3,2,2,1,1,1,1)	(1,1,**1**,1,1,**1**,1,2,**2**,3,3)	(1,1,2)

The processors broadcast the tagged regular samples

processor	broadcasts
1	$(1_1, 2_1, 2_1)$
2	$(1_2, 1_2, 2_2)$
3	$(2_3, 2_3, 3_3)$
4	$(1_4, 1_4, 2_4)$

Each processor receives $((1_1, 2_1, 2_1), (1_2, 1_2, 2_2), (2_3, 2_3, 3_3), (1_3, 1_3, 2_3))$ which is merged to $(1_1, \mathbf{1_2}, 1_2, 1_4, 1_4, \mathbf{2_1}, 2_1, 2_2, 2_3, \mathbf{2_3}, 2_4, 3_3)$. The shared regular sample is $(1_2, 2_1, 2_3)$

Repartitioning occurs:

processor	receives from p_1	receives from p_2	receives from p_3	receives from p_4
1	(1,1,1,1,1)	(1,1,1,1,1,1,1)	()	()
2	(2,2,2,2)	()	(1,1)	(1,1,1,1,1,1,1)
3	()	(2,2,2)	(2,2,2,2)	()
4	(3,3)	(3)	(3,3,3,3,3)	(2,2,3,3)

The received lists are merged:

processor	merged list
1	(1,1,1,1,1,1,1,1,1,1,1,1)
2	(1,1,1,1,1,1,1,1,1,2,2,2,2)
3	(2,2,2,2,2,2,2)
4	(2,2,3,3,3,3,3,3,3,3,3,3)

If PSRS without tagging by processor were used instead the data would have been partitioned the following way:

processor	merged list
1	(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
2	(2,2,2,2,2,2,2,2,2,2,2,2,2)
3	(3,3,3,3,3,3,3,3,3,3)
4	()

It can be seen that it is not possible to distribute duplicates across processors which leads to heavy skew.

4.4.2 Sort-Merge Join Without Crossing Values

The *distributed sort-merge join* starts, just like the hash join, with the tuples of R and S as input distributed arbitrarily among the processors. The first step is to sort all tuples by the join attribute using the regular PSRS algorithm. Now each processor has a list of tuples sorted by join attributes and can perform the joins of the tables in one scan of the list.

If all values are unique, $L = O(\frac{\text{IN}}{p})$, but if duplicates are allowed to occur, then $L = O(\text{IN})$.

4.4.3 Sort-Merge Join With Crossing Values

To address the potentially problematic performance of the previous sort-merge sort in the case of many duplicates an optimization can be added.

Firstly, the PSRS algorithm with tuples tagged by processor partitions the data evenly among them but possibly with values of the join attribute exceeding the boundaries of the processors. Next, the crossing values have to be determined. For this each processor i determines the minimum and maximum values b^i_{min} and b^i_{max} as well as the number of tuples with these values $N_{t^i_{min}}$ and $N_{t^i_{max}}$. The values are broadcasted. Each processor can determine whether a value is a crossing value or not and its number of occurrences. Non-crossing values are computed directly by the processor. In a final round, the crossing values are allocated processors based on the number of occurrences like in the hash join with heavy hitter optimization.

The worst-case load of the PSRS algorithm is reduced to $O(\frac{\text{IN}}{p})$ and the rectangle algorithm for the cartesian product of crossing values achieves a worst-case load of $O(\text{OUT}/p)$. Together, the worst-case load of the algorithm is $O(\max\{\frac{\text{IN}}{p}, \sqrt{\text{OUT}/p}\})$.

CHAPTER 5

MapReduce Join Algorithms

5.1 Reduce-Side Joins

Reduce-side joins are also referred to as repartition joins since they involve repartitioning the data. Sometimes they are referred to as hash joins, even though they also use sorting for grouping as opposed to the MPC algorithm.

Two tables, T_1 and T_2, and join attributes a_1 and a_2 are given. The goal is to compute $T_1 \bowtie_{T_1.a_1=T_2.a_2} T_2$. $T_1.a_1$ will be unique when we are dealing with normalized data and foreign keys but the algorithm can also handle non-unique values. The rows of table T_i will be given as multiple HDFS files, where T_1 and T_2 can be comprised of multiple files. For each file F_i, rows will be split up and sent to mappers in chunks of sizes $\min\{S_i, P\}$ where S_i is the file size and P is the configured split size. Due to the need to later identify which table a row came from, two types of mappers have to be used. The map function will be parameterized with the index of the table. The rows of table T_i will be sent to the mappers of type $M_{i,*}$ with the map function $M(\langle k, r\rangle) = \langle\langle i, r.a_i\rangle, \langle i, r\rangle\rangle$. Including the index of the table in the key would not be necessary in a simple implementation but it makes an optimization for reduced memory consumption possible.

In a slightly simpler implementation the map function $M(\langle k, r\rangle) = \langle r.a_i, \langle i, r\rangle\rangle$ could be used but this would require buffering all values in the reducer and generating all combinations in two loops. This is not optimal because all rows would have to be loaded into memory at once, possibly even leading to an out of memory failure. Instead, the data can be processed in a stream. To make this possible, a custom grouping comparator and partitioner have to be defined. If the data were to be partitioned with the default partitioner, the *hashCode* and *equals* of the *JoinTuple* class would be used for partitioning, possibly resulting in e.g. the keys $\langle 1, a\rangle$ and $\langle 2, a\rangle$ ending up at different reducers even though the values have to be joined. A partitioner ignoring the table index is used instead.

```
public static class JoinPartitioner
    extends Partitioner<JoinTuple, JoinTuple> {
    @Override
    public int getPartition(JoinTuple key, JoinTuple value,
        int numPartitions) {
        return (key.getTuple().hashCode() & Integer.MAX_VALUE)
            % numPartitions;
    }
}
```

Leaving the default grouping comparator, which defines equality as the bytes of both keys being equal, would lead to the rows of the same attribute but of different tables being passed to different reduce functions. To prevent this a custom comparator is defined that ignores the integer value identifying the table:

```
public static class GroupingComparator extends WritableComparator {
    protected GroupingComparator() {
        super(JoinTuple.class);
    }

    public int compare(byte[] b1, int s1, int l1,
        byte[] b2, int s2, int l2) {
        return compareBytes(b1, s1 + 4, l1 - 4,
                b2, s2 + 4, l2 - 4);
    }
}
```

The secondary sort defined in the *JoinTuple* class, which takes into consideration first the join attribute and then, when the join attributes are equal, the table index, will guarantee that the rows of the first table (usually only one) are iterated over before those of the second table. This allows buffering only the rows of the first table and then scanning those of the second without keeping them all in memory. The worst-case memory consumption of the mappers is therefore in $O(|T_1|)$ but only in the case of all attributes being equal. When they are all different it is in $O(1)$. In the naive implementation it would always be $O(|T_1| + |T_2|$ so this is a significant improvement.

Due to the shuffling stage of the MapReduce framework, a pure distributed hash join, as described in [7], cannot be implemented. The sorting of the mapper outputs cannot be disabled and is necessary for Hadoop's shuffling. The way in which reducers perform the grouping process by merging the sorted output of the mappers effectively restricts the choice of join algorithms that can be performed by the reducers to the (sort-)merge join.

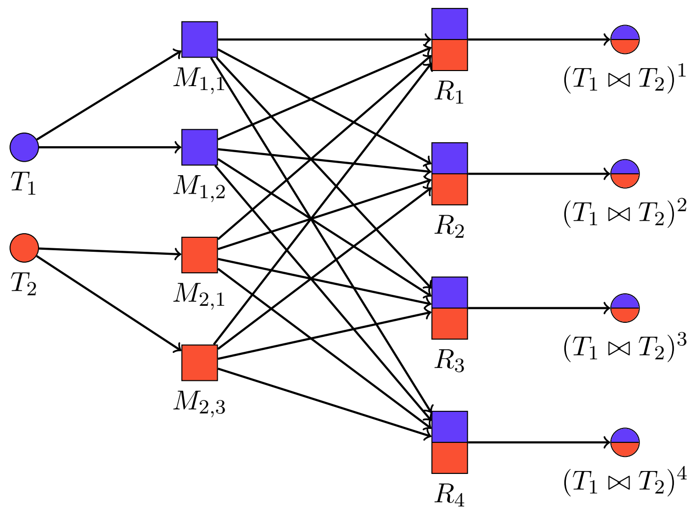


Figure 5.1: Reduce-Side Join

Hence, the reduce-side join is not entirely based on hashing and more of a mix of a hash join and a merge join.

5.2 Broadcast Joins

Broadcast Joins, also referred to as *fragment-replicate joins*, *distributed-cache joins*, *memory-backed joins*[19] and sometimes simply as *map-side joins* (even though this is a broader class) are a common method that can be applied if one side of the join is small enough that it fits into memory on all nodes.

The smaller table is "broadcasted" to all nodes, meaning replicated via the distributed cache, which then build a hash map of the table and join their assigned part of the larger table with it. Reducers are not necessary and hence the number of reducers is set to zero, disabling the shuffling process and reducing stage. The output of the reducers is directly written into the HDFS.

By disabling the shuffling stage the whole expensive process of sorting and transferring data is eliminated resulting in significant performance gains.

5.3 Map-Side Merge Join

While the distributed Sort-Merge Join, like the Hash Join, cannot be implemented exactly as described, the Hadoop framework supports a further optimization for joins that does not quite fit into the MapReduce paradigm but provides an efficient way to join already sorted data. This feature of Hadoop explicitly serves the purpose of implementing joins. The method can be applied if the data stored in the HDFS fulfills several conditions:

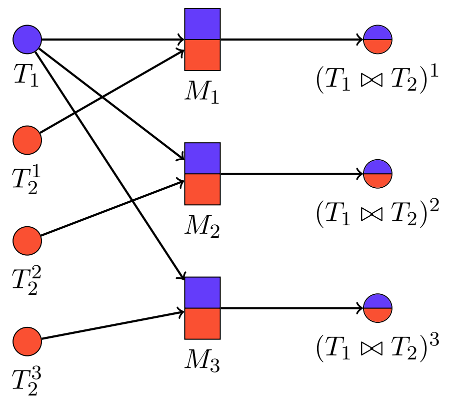


Figure 5.2: Broadcast Join

- The tables must be sorted with the same comparator (e.g. *WritableComparator* - natural ordering)
- The tables have to be partitioned with the same paritioner (e.g. *TotalOrderPartitioner*)
- The number of partitions must be the same for all tables
- The order of the partitions must be equal for all tables

This scenario is in fact very common in practice since the output of another MapReduce job will often meet these conditions. Otherwise a MapReduce job using the *TotalOrderPartitioner* and identity mappers and reducers can be used to sort the data.

Like the broadcast join, the map-side merge join does not need shuffling and reducing. The mappers, in this case, are interestingly acting more like reducers because they perform a merging operation over multiple tables. All that has to be done in the map method, since the merging is handled by Hadoop, is to combine the values of the *TupleWritable* object corresponding to the original tables to be joined which were already grouped by key by the Hadoop framework.[20]

5.4 Algorithms for Other Join Types

5.4.1 Non-Equi (Theta) Joins

Non-equi joins such as inequality joins, band joins or similarity joins are generally implemented inefficiently since the standard algorithms like hash joins or sort joins cannot be applied to the problem. Current systems like SparkSQL tend to have to compute the cartesian product to evaluate the join. As shown by Khayyat et al. distributed inequality joins can be sped up significantly with their *IEJoin* algorithm.[21]

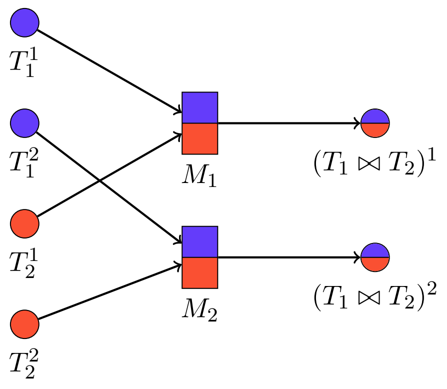


Figure 5.3: Merge Join on Sorted Data

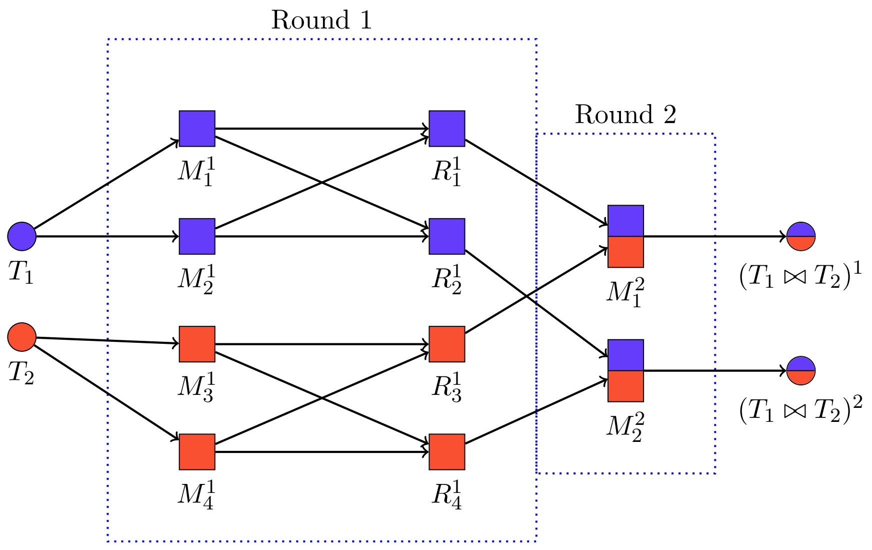


Figure 5.4: A Sort-Merge Join in Two Rounds

5.4.2 Multiway Joins

The straightforward approach towards computing multiway joins is repeated application of binary joins. This is in fact how most MapReduce systems solve the problem but it is not alwaysthe most efficient approach. One more efficient solution is the *hypercube* algorithm.[7]

CHAPTER 6

Empirical Evaluation of MapReduce Join Algorithms

Our goal is to determine how the the different join algorithms for Hadoop MapReduce compare against each other in various scenarios. For a realistic measure of real-world performance we will focus on the wall clock time of the MapReduce jobs.

6.1 Data Generation

To achieve a controlled environment a suitable data generator was implemented for the experiments. This allows quick generation of unbiased and easy-to-analyze data. For an accurate simulation of real-world scenarios the data generator takes several parameters to influence the data output. Two input CSV files or directories are written into the HDFS. The number of threads corresponding to the amount of files generated can be varied to speed up the data generation. It might however lead to fragmentation on small input sizes.

To assess performance under various levels of skew, we generate keys according to the Zipf distribution. The Zipf distribution is a discrete power-law distribution (also referred to as the *discrete pareto distribution*) frequently observed in real-world scenarios. For example, the distribution of words in the English language follows a Zipf distribution with the parameter s approximately equal to 1. We need two parameters to describe the distribution: $N \geq 0$ is the number of unique values and $s \geq 0$ is a parameter describing skew. In the case of $s = 0$ there is no skew and the distribution is uniform.

There is a distinction between what we will call *double skew* and *single skew*. In the case of single skew one table (in our experiments always the left table) contains unique and hence uniformly distributed values such as in most joins in relational databases where there is a unique key. Double skew is more problematic because both tables are skewed leading to large output sizes if the distributions are similar. We will only be investigating single skew although the data generator has an option to generate double skew.

Sometimes rows store the lines of a set of books (Word Count) but sometimes they might also store multi-megabyte images. Attributes are random alphanumeric strings and the count and length per attribute can be varied. In the following experiments the rows are random strings of a size of 200 bytes plus the keys.

6.2 Simulation Environment

Experiments were carried out on the TU Vienna Hadoop cluster (Hadoop 2.6.0 / CDH 5.15). The cluster consists of 17 nodes, most of which have 185 GB of RAM and 48 (virtual) cores available. In total 3.03 TB of memory is available. The HDFS has a capacity of 257 TB. Since there is a chance of other applications utilizing the cluster at the same time and it is not necessary to use more for a realistic comparison number of reducers is always fixed to 100 to increase robustness.

The HDFS block size is set to 64 MB and the input split size is not configured differently. While the number of reducers can be fixed, Hadoop provides no way to set the maximum number of mappers other than configuring split size. This makes it impractical to investigate speedup since the cluster provides up to 816 map tasks. To utilize all of them a starting input size of 50GB would be needed. Instead however, since Hadoop will linearly increase the number of map tasks with increasing input, the scaleup can be observed.

6.3 Speedup and Scaleup

To experimentally determine speedup and scaleup the input size is increased by the number of rows. The following figure shows the speedup of the repartition join with 100 reducers. Although there is significant noise in the data a clear linear trend can be observed.

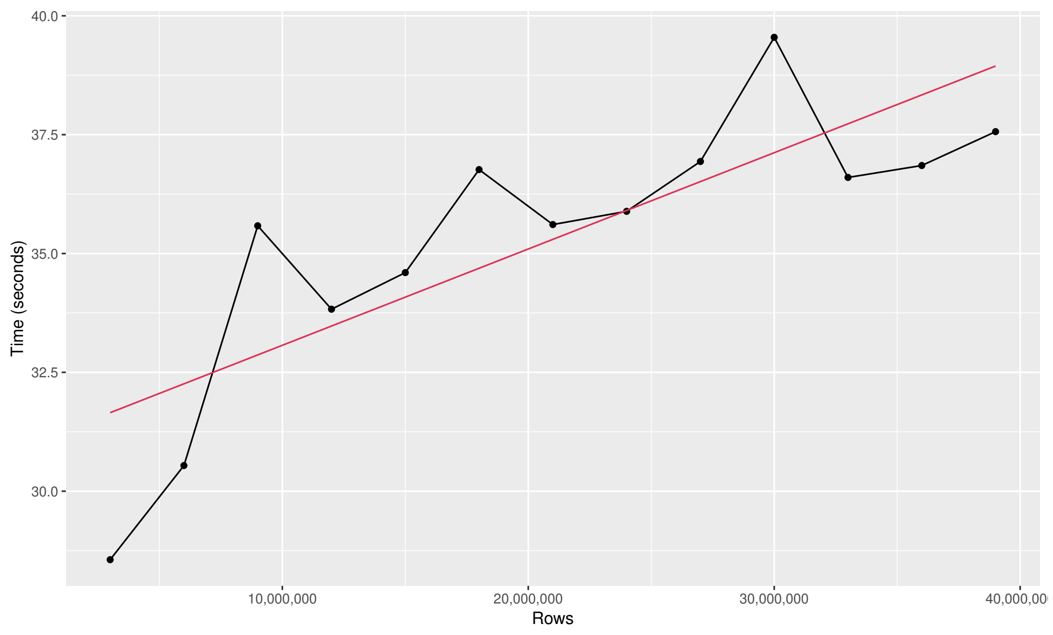


Figure 6.1: Repartition Join, skew = 0.5

In the following figure the same data is joined by the broadcast join. Since the number of reducers is automatically scaled up the diagram shows the scaleup of the algorithm, which would ideally be linear. Due to the increasing size of the smaller table (from 300,000 to 4,000,000), the performance of the broadcast join falls drastically near the end. In the range between 3M and 33M rows (of the bigger table) scaleup is still close to constant and the performance rivals that of the repartition join with a lower number of rows.

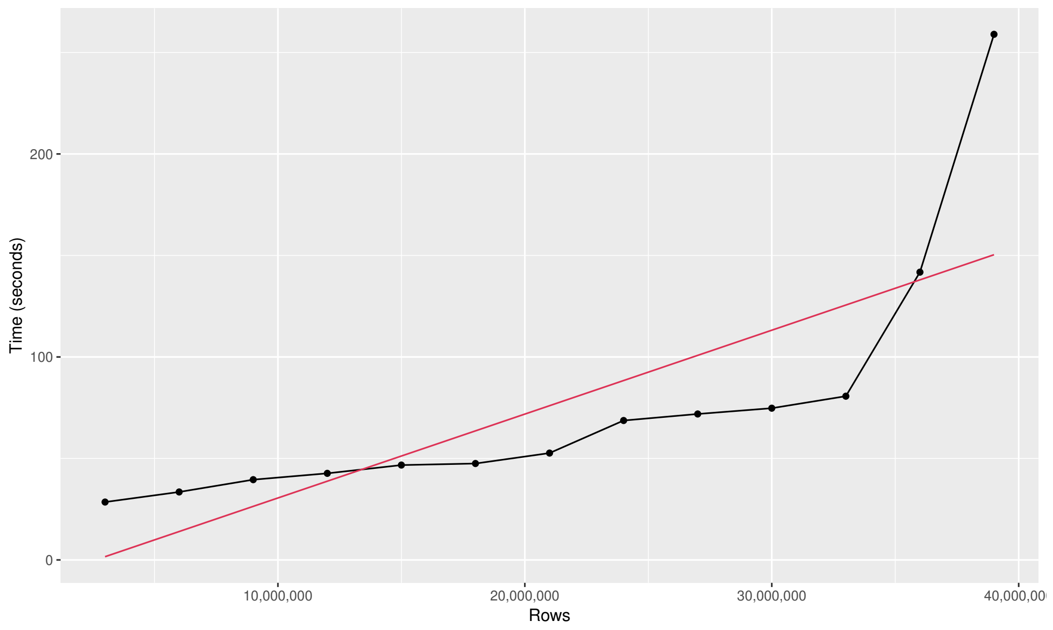


Figure 6.2: Broadcast Join, skew = 0.5

As is seen in the following figure, the merge part of the merge join has a linear scaleup. However, its performance is still superior to that of repartition and broadcast join and the increase in runtime of 30% for an increase in input size of 1000% is rather insignificant.

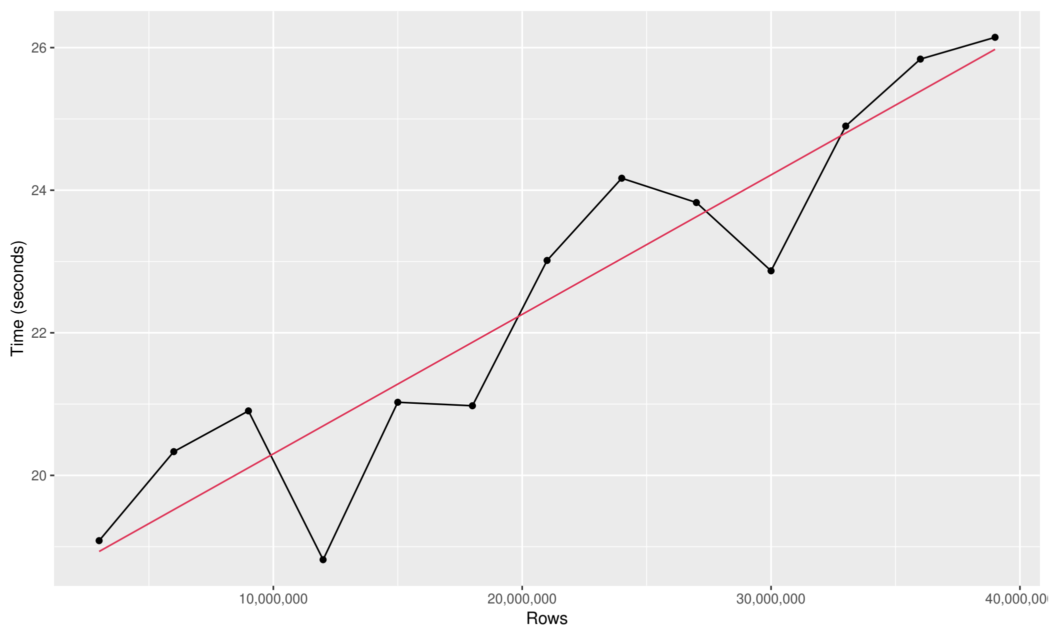


Figure 6.3: (Sort-)Merge Join, skew = 0.5

The following comparison shows the merge part (t_merge) of the merge join (t_sort_merge) separated. It follows that, if the data was already sorted and ready for the merge join, it would be the fastest choice. The repartition join provides reliable performance and scalability. Surprisingly the broadcast join does not exceed the performance of the repartition join. This appears to be attributable to the splitting of the smaller table across 40 files which likely imposes overhead due to HDFS operations. In figure 6.5 a direct comparison between the broadcast and repartition join with only one file as input for the smaller table is given supporting this hypothesis.

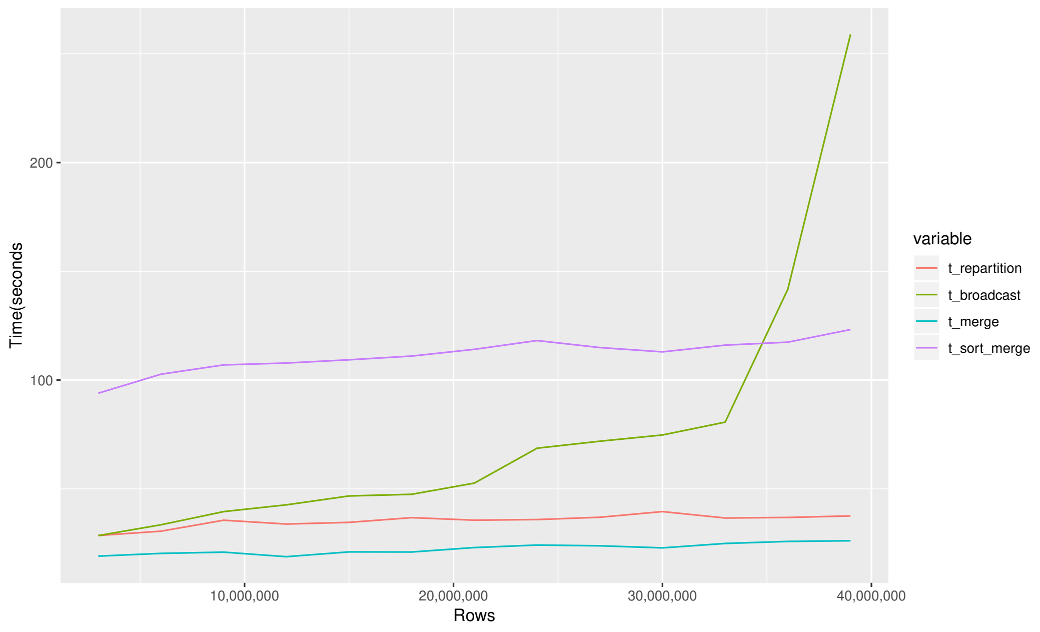


Figure 6.4

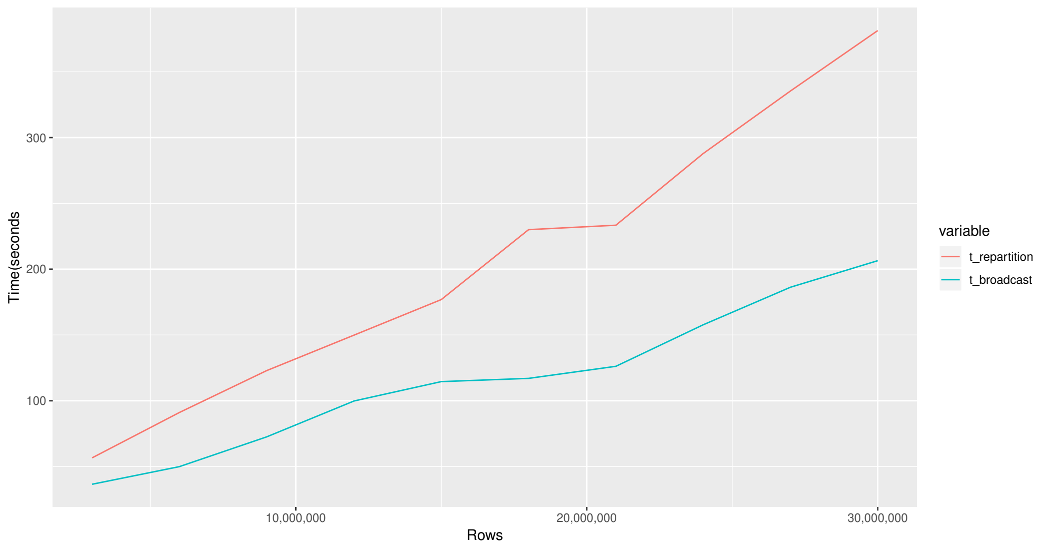


Figure 6.5: Broadcast vs. Repartition join, one input file and double skew

The broadcast join is limited by the size of the smaller table. In a simulation with about 60,000,000 rows with 10% unique keys and hence 6,000,000 keys in the smaller table, corresponding to a file size of 3795 MB, the error *GC overhead limit exceeded* occurs due to excessive memory consumption and the broadcast join is no longer a choice. With a file size of 2.5GB the broadcast join works reliably.

One issue encountered with the built-in merge join feature using *CompositeInputFormat* are random failures due to Java processes running out of memory as the number of rows and skew as well as increases. With the options *-rows 42000000 -increment 3000000 -steps 1 -unique 0.1 -reducers 100 -zipf-skew 1.2 -threads 40* and the default heap size the map tasks will most likely fail on the cluster. A failure was observed even with as few as 12000000 rows. A solution would be to increase the heap size of the Java processes. However, the random failures that increasingly occur as skew and data size increases seem to indicate that the feature might not be reliable and in the worst case there is always a non-zero chance of failure.

6.4 Performance under Skew

We have performed two experiments with a different number of rows and skew varying from 0.1 to 1.1 and and 0.1 to 1.0 (differing due to a failure of the merge join). 6 million rows, with our fixed row size, result in input file sizes of 3.75GB and 375MB while 40 million rows correspond to 25GB and 2.5GB. The reason for the relative slowness of the broadcast join is the fact that the smaller table is, like the larger table, generated by 40 threads resulting in 40 small files (using one file would be faster in this case), and the file size which is close to the limit.

From Figure 6.4 and Figure 6.5 it can be seen clearly that the performance of the merge join and to a lesser degree also of the repartition join is degrading as the skew increases while the broadcast join's runtime remains constant as expected. In the experiment of Figure 6.5 the merge join was failing at a skew of 1.1.

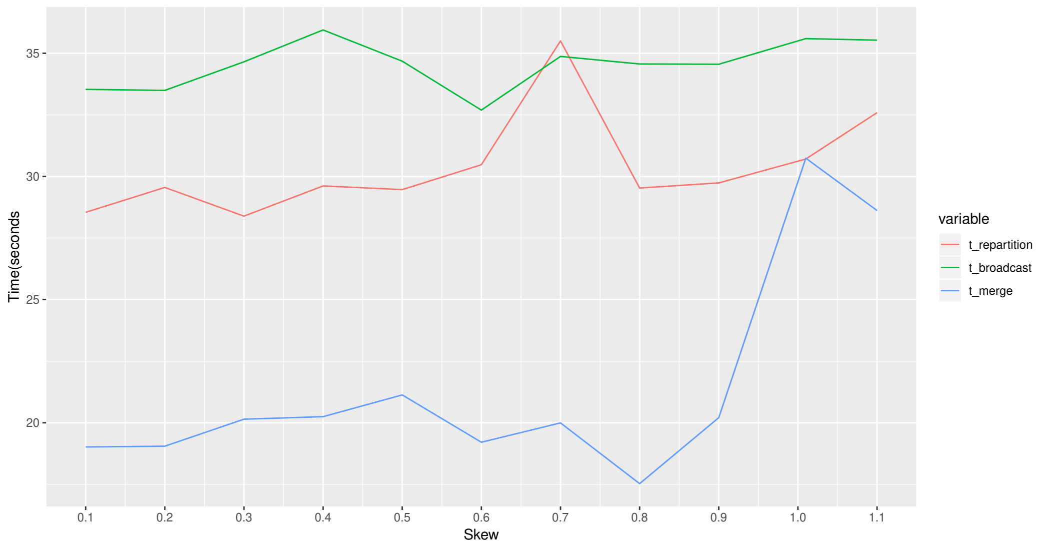


Figure 6.6: Performance as skew increases with 6,000,000 rows

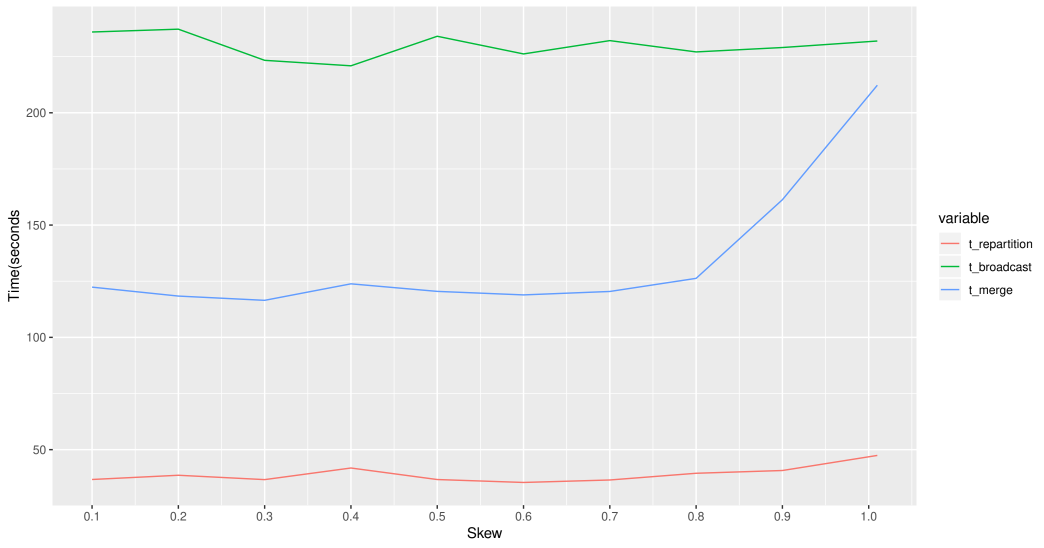


Figure 6.7: Performance as skew increases with 40,000,000 rows

The runtimes of the map and reduce tasks paint a picture of why increased skew in the input data causes the algorithm to slow down. As seen in Figure 6.8, the reduce task runtimes are significantly skewed at an input key skew of 0.9 and 1.0. In the $s = 1.0$ case the slowest reduce task takes 25 seconds to complete, which is about 50% longer than the second-slowest reduce task and more than twice as long as average.

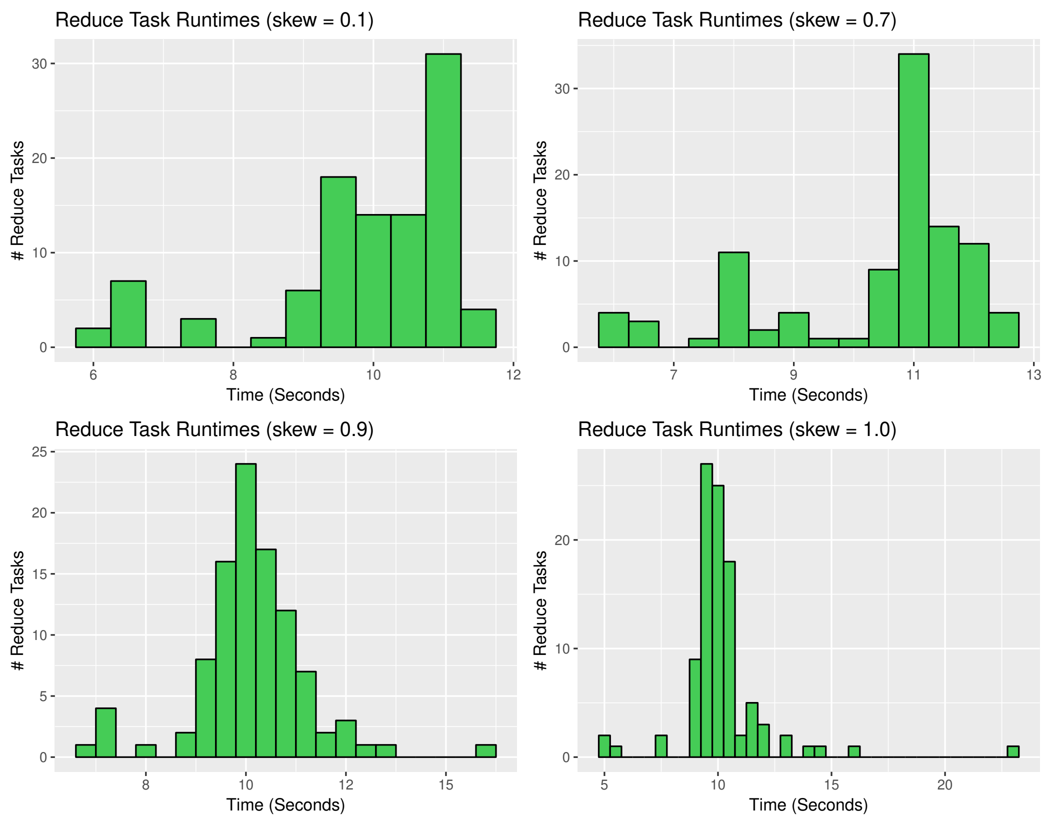


Figure 6.8: Reparition Join Reduce Task Runtimes

As can be seen in Figure 6.9 the broadcast join's reduce task runtimes have no noticeable skew.

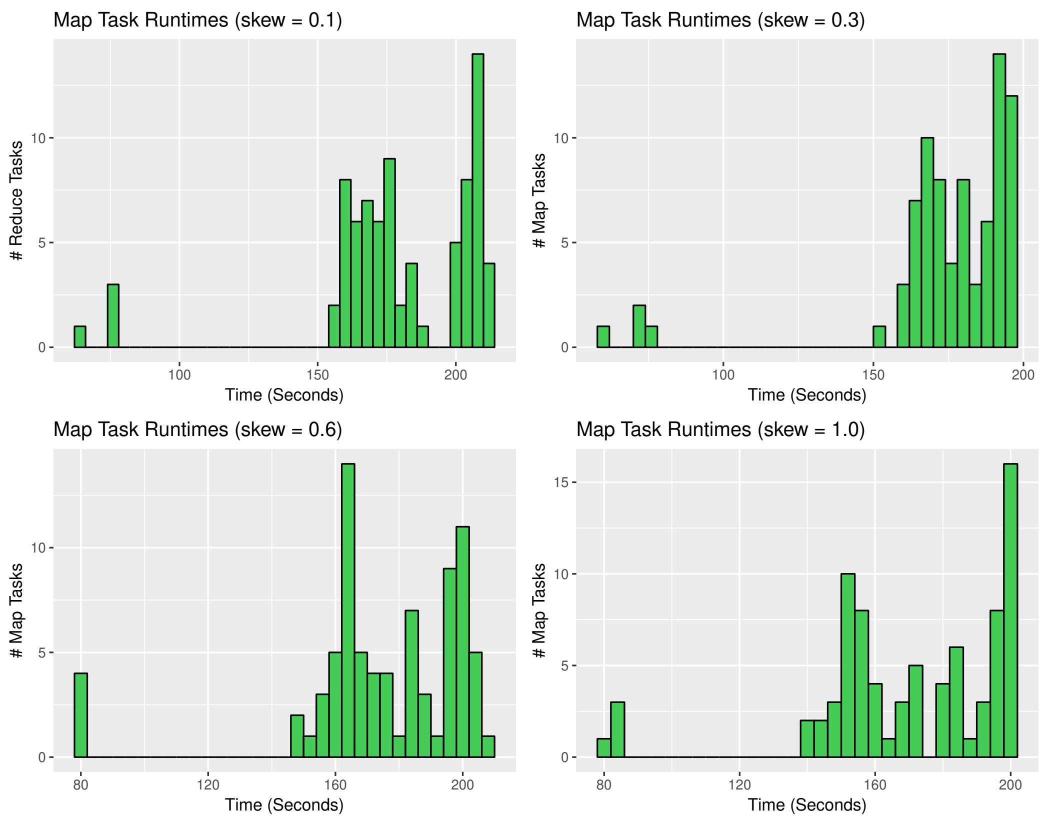


Figure 6.9: Broadcast Join Map Task Runtimes

Figure 6.10 shows how, in the map-side merge join, the amount of map tasks has to be reduced as skew increases, and that runtimes are still somewhat skewed.

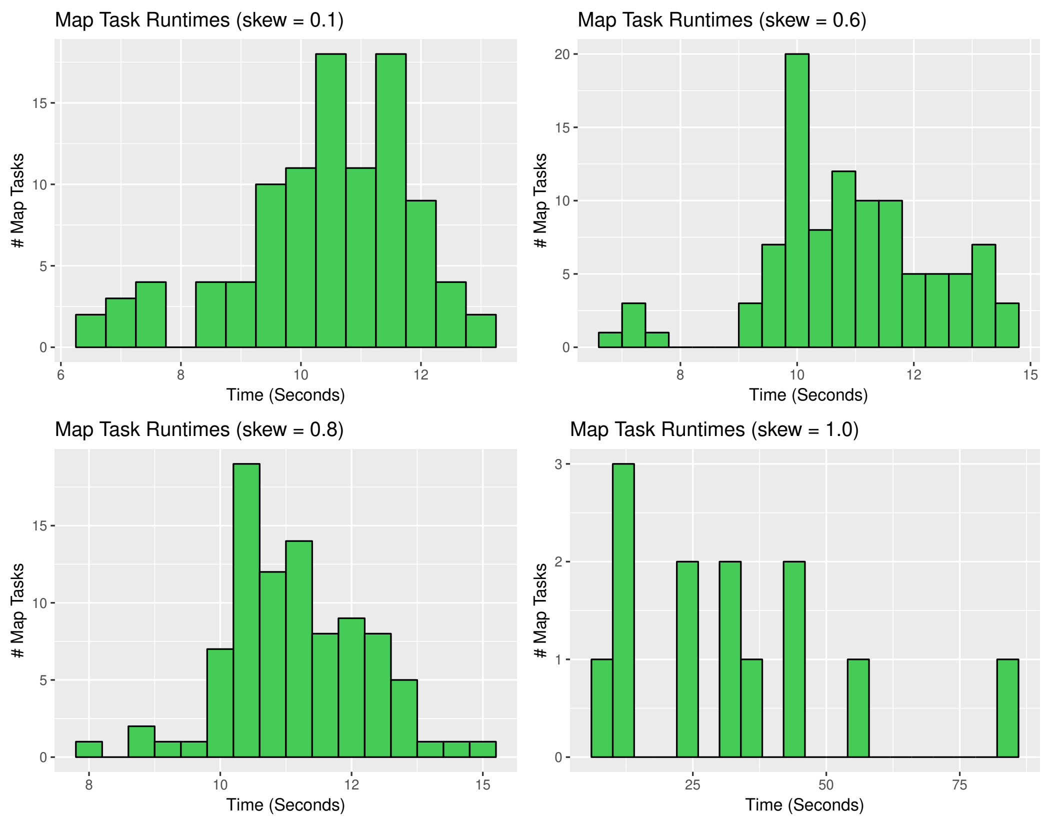


Figure 6.10: Merge Join Map Task Runtimes

Our implementation of the merge join in Hadoop using the built-in RandomSampler has to be increasingly limited in its parallelism and therefore its performance as skew increases. If the number of splits is too large Hadoop will distribute keys in a way in which they are spanning multiple splits. This later results in an error as the partition file is read. To prevent such a failure the number of reducers and split size has to be set so that when each partition receives the same number of keys the number of keys on each partition will not exceed the maximum number of occurrences of the most common value of the input data. To make the join work reliably on arbitrary data the maximum occurrences would need to be counted possibly imposing a slight overhead. In our case, since we know the distribution, we can easily determine how many of the most common value (1) are (approximately) generated by the Zipf distribution.

6.4.1 Choosing the Best Join Algorithm

When the data is already sorted and partitioned such that a merge-only join can be performed directly, as we have seen from the empirical data, a merge join is the best choice. If the smaller table fits into memory (when that is the case depends on the environment), a broadcast join can be performed very efficiently, to prevent the need for the shuffle phase. If, however, neither criterion is fulfilled the repartition join with the shuffle phase is likely necessary. When skew is high, due to the instability of the map-side merge join in Hadoop, the repartition join is most likely the best choice. Note however that this might as well be an issue with our implementation of the merge join. Since the majority of the cost of the sort-merge join is made up by the sorting stage, if data is partially sorted, the sort-merge join might well be a viable choice and more performant than the repartition join.

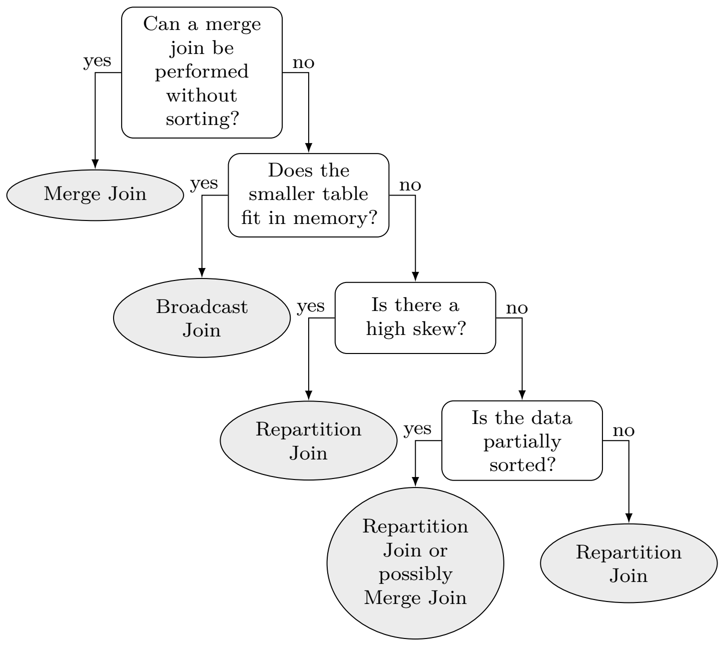

Can a merge join be performed without sorting?
yes
no
Merge Join
Does the smaller table fit in memory?
yes
no
Broadcast Join
Is there a high skew?
yes
no
Repartition Join
Is the data partially sorted?
yes
no
Repartition Join or possibly Merge Join
Repartition Join

Figure 6.11: Decision tree for deciding on a join type

CHAPTER 7

Conclusion

We have seen that the join algorithms described in BSP and MPC, while easy to analyze, are challenging to convert into MapReduce implementations without extra thought and some deviations due to restrictions of the framework and optimization needs. Although it is sometimes ignored, the shuffle stage of the Hadoop MapReduce framework and the implementation of grouping by sorting can make a significant difference in the performance of the actual implementation.

The experimental evaluation of different algorithms has shown that the choice of the optimal algorithm depends very much on the specific situation and the choice has to be made considering the data and environment.

Further Work

We have only focused on binary inner equi-joins and did not make any ambitious attempts to optimize the algorithms. There are more topics that would be a good addition but would have increased the scope too much:

- The feasibility of replacing grouping by sorting by grouping by hashing and the performance effects
- Optimizing the merge join by rewriting the sampler and partitioning
- Automated choice of the optimal algorithm on arbitrary architectures
- Optimizing semi-joins
- Implementing heavy-hitter optimizations in Hadoop
- Algorithms for inequality joins

- Algorithms for theta-joins
- The implementations of joins in existing software
- Differences in the possible implementations of joins and their efficiency in Hadoop vs. Spark
- Optimizing the broadcast join with multiple input files

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[2] "Mysql 5.7 reference manual :: 8.2.1.6 nested-loop join algorithms," accessed: 2018-08-18. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/nested-loop-joins.html

[3] "Hash joins," Jun 2014, accessed: 2018-08-20. [Online]. Available: https://jonathanlewis.wordpress.com/2013/09/07/hash-joins/

[4] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[5] D. B. Skillicorn, J. Hill, and W. F. McColl, "Questions and answers about bsp," *Scientific Programming*, vol. 6, no. 3, pp. 249–274, 1997.

[6] H. Cha and D. Lee, "H-bsp: A hierarchical bsp computation model," *The Journal of Supercomputing*, vol. 18, no. 2, pp. 179–200, 2001.

[7] S. S. Paraschos Koutris and D. Suciu, "Algorithmic aspects of parallel data processing," *Foundations and Trends® in Databases*, vol. 8, 2018.

[8] T. White, *Hadoop: The Definite Guide.* O'Reilly, 2015.

[9] M. T. Goodrich, N. Sitchinava, and Q. Zhang, "Sorting, Searching, and Simulation in the MapReduce Framework," *ArXiv e-prints*, Jan. 2011.

[10] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Mrshare: sharing across multiple queries in mapreduce," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 494–505, 2010.

[11] J. Daenen, F. Neven, T. Tan, and S. Vansummeren, "Parallel evaluation of multi-semi-joins," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 732–743, 2016.

[12] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 607–614. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963491

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A study of skew in mapreduce applications."

[14] J. Myung, J. Shim, J. Yeon, and S.-g. Lee, "Handling data skew in join algorithms using mapreduce," *Expert Syst. Appl.*, vol. 51, no. C, pp. 286–299, Jun. 2016. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2015.12.024

[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213840

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets."

[17] M. Felice Pace, "BSP vs MapReduce," *ArXiv e-prints*, Mar. 2012.

[18] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of parallel and distributed computing*, vol. 14, no. 4, pp. 361–372, 1992.

[19] F. Atta, S. D. Viglas, and S. Niazi, "Sand join—a skew handling join algorithm for google's mapreduce framework," in *Multitopic Conference (INMIC), 2011 IEEE 14th International*. IEEE, 2011, pp. 170–175.

[20] J. Venner, *Pro Hadoop*, 2009.

[21] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis, "Lightning fast and space efficient inequality joins," *Proc. VLDB Endow.*, vol. 8, no. 13, pp. 2074–2085, Sep. 2015. [Online]. Available: https://doi.org/10.14778/2831360.2831362