Mitigating Skew in Large-Scale Data Processing Systems

Michael Conley
mconley@cs.ucsd.edu
University of California, San Diego

ABSTRACT

The need to process huge amounts of data has necessitated the development of large-scale data processing systems. Such systems employ parallel hardware configurations to achieve acceptable throughput and job execution times. However, since the performance of a parallel execution is dominated by the speed of its slowest parallel task, it is desirable that all tasks have approximately the same execution time. *Skew*, i.e. non-uniformity or variation in the execution time of the parallel tasks, can greatly reduce the performance of parallel execution.

Skew is an old problem that parallel database researchers have been wrestling with for decades. Many skew-resistant techniques have been proposed to mitigate both data and parallel partitioning skew. In particular, there is a large body of work on skew-resistant parallel join algorithms.

Parallel execution frameworks in use today, such as MapReduce, are also susceptible to skew. Such systems represent a new domain for the problem of skew mitigation. MapReduce architectures suffer from many types of skew ranging from hardware-related skew to data skew.

We evaluate skew both historically, in the context of parallel databases, and recently, in the context of MapReduce architectures. We also include a brief analysis of our MapReduce implementation, ThemisMR, which uses a sample-driven preprocessing step to solve data partitioning skew.

1. INTRODUCTION

Large-scale data processing is an increasingly important area of focus in the research community. The rise of 'big data' is largely due to the enormous amounts of information collected by warehouse-scale computing companies such as Google, Microsoft and FaceBook. These companies routinely need to process petabytes of data [4]. A cost-effective strategy is building large clusters of failure-prone commodity machines and relying on the software for data durability and reliability.

Researchers in the database community noticed this problem, albeit at smaller scale, decades ago. Database systems were originally written for shared-memory systems using specialized hardware but have since migrated to a shared-nothing architecture that benefits from commodity hardware [6]. These systems are termed *parallel databases* due to their parallel execution structures. Much work has been done on building efficient parallel databases. These works have been commercialized by companies such as Oracle and Teradata, typically at a high price to the end-user. It is not surprising then that these systems are generally used by large enterprises.

Large information companies such as Google and Microsoft found it more cost-effective to develop their own software. Google developed the MapReduce [5] framework that has since become popularized by its open-source implementation, Hadoop [34]. Microsoft Research designed a parallel dataflow execution engine called Dryad [12] that can be used to run MapReduce jobs among other applications.

While these data processing systems are fundamentally different from parallel databases, they share many of the same efficiency concerns. In particular, the completion time of any large computing job is dominated by the speed of its slowest parallel task. For parallel databases, this means that the query planner, scheduler, and optimizer must ensure that any individual parallel operator is not overloaded by an imbalance in the data it produces or consumes. Similarly, MapReduce architectures must guarantee that any individual `map` or `reduce` task is not significantly slower than the average task.

Generally, when the data to be processed is uniformly distributed across its value space, simple techniques such as hash or range partitioning are sufficient to guarantee that any individual operator or task is not slower than the others. However, many real world applications operate on *skewed data*, i.e. non-uniform data [21]. For example, when processing page links on the Internet, some websites are significantly more popular than others. Partitioning by domain name will cause some partitions to be skewed and be much larger than others. It will therefore take significantly longer to process these partitions, which can greatly impact the job's completion time.

We will discuss both the historical and contemporary work on *skew mitigation* techniques. The subsections

that follow will each focus primarily on one piece of work, although it will clear if we reference one work when discussing another. The rest of this paper is structured as follows: In Section 2 we will discuss join algorithms used in parallel databases and how they handle skewed relations. In Section 3 we will look at MapReduce systems and how they solve the problem of skew. In Section 4 we will briefly discuss the author's work and how it handles skew. Finally Section 5 concludes.

2. SKEW IN PARALLEL DATABASES

The database community has been operating at the heart of large data processing systems for many decades. Relational databases are responsible for finding the optimal way to translate a user's query into a sequence of operators that read and transform data stored in persistent relations to produce the desired result. Efficient software design gives rise to upward scalability, but such architectures quickly become expensive. A more practical solution is to use a scale-out architecture that parallelizes the relational database software [6].

One of the most complex operators in a database query is the join operator. A join takes tuples from several relations and combines them together, typically by some filtering predicate such as attribute equality. Since a join must in the worst case look at every pair of tuples, it is an obvious target for optimization. Parallel databases compound this problem by requiring an even partitioning of the join workload in order to get parallel speedup. If join partitions are incorrectly computed, the join will be slow, which impacts the overall query response time. The rest of this section will describe the impact of skew on the space of parallel join algorithms.

2.1 Parallel Join Algorithms

The simplest join algorithm is a *nested loop join*. The nested loop join operates by comparing every tuple in one relation, the *outer relation*, with every tuple of the other relation, the *inner relation*. This is analogous to a pair of nested `for` loops, where the *inner* loop executes inside the *outer* loop. As in the case of the `for` loops, the nested loop join compares tuples one by one. It therefore makes a quadratic number of comparisons. More importantly, it requires multiple scans if the inner relation is too large to fit in memory.

Researchers quickly determined that nested loop join was not terribly efficient and began to look at other types of joins. Schneider and DeWitt [26] compared the performance of four popular join algorithms in a shared-nothing multiprocessor environment, namely the Gamma database system. There are essentially two popular categories of join algorithms beyond nested loop. The first type, termed *sort-merge join*, relies on merging sorted runs, and the second type, *hash join*, uses hash functions to speed up comparisons. It should be noted that this use of a hash function is distinct from *hash partitioning*, in which relations are partitioned by hashing the join attribute. Partitioning is orthogonal to the join type. For example, a sort-merge join could conceivably use hash partitioning to divide the relations. This partitioning can be avoided if the relations are already partitioned on the join attributes, but this cannot be assumed in the general case.

The sort-merge join algorithm begins with an initial partitioning of the relations across the cluster by the join attributes. Tuples are written to a temporary file as they are received across the network. After all tuples have been redistributed, each file is sorted. Finally, the join result is efficiently computed via a merge of the sorted partition files for each relation. Joins can be computed locally because the relations have been redistributed on the join attributes.

In the same work, Schneider and DeWitt survey three types of hash join algorithms. The first is a *simple hash join* algorithm that begins by redistributing relations on the join attributes. As tuples are received at their destination sites, a hash table is constructed from the inner relation using a second hash function. The tuples from the outer relation probe the hash table using this second hash function to compute the join result locally. Note that a hash join algorithm such as simple hash join will only work if the join is an *equijoin*, i.e. a join with an equality condition on join attributes. While there has been work on non-equijoins [7], most of the literature focuses on equijoins because they are common in practice.

The second type of hash algorithm is the *GRACE hash join* algorithm. We will discuss this algorithm in further detail in Sections 2.3.1 and 2.3.2, but a quick description of the algorithm is as follows. The relations are partitioned into *buckets* where the number of buckets is much greater than the number of nodes in the cluster. Buckets are partitioned across the cluster in the initial *bucket-forming* stage. Next, in the *bucket-joining* stage, corresponding buckets from each relation are joined locally using a hash method similar to the simple hash join.

The *hybrid hash join* is a combination of simple hash join and GRACE hash join. Hybrid hash join operates like GRACE hash join, except the first bucket is treated separately. Instead of writing the first bucket back to stable storage, an in-memory hash table is constructed from the inner relation and probed by the outer relation as in the simple hash join. Thus the joining of the first bucket is overlapped with the bucket-forming stage for slightly increased performance. The hybrid hash join outperforms the other joins in most situations

2.2 Skew in Parallel Joins

While Schneider and DeWitt [26] characterized sev-

Skew Type		Description	Point of Manifestation
TPS	Tuple Placement Skew	Initial relation partitioning imbalanced	Before the query starts
SS	Selectivity Skew	Query only selects certain tuples	After Select operator is applied
RS	Redistribution Skew	Relation partitioned unevenly on join attribute	After tuples are redistributed
JPS	Join Product Skew	Join output volume differs between partitions	After local joins are computed

Table 1: Summary of skew in parallel join algorithms.

eral parallel join types, their analysis of the effects of skew was limited. Two years later, Walton, Dale, and Jenevein [30] constructed a taxonomy of the various types of join skew. They note that there are really two categories of skew with which to be concerned. The first, *attribute value skew*, or *AVS*, is intrinsic to the relations. AVS means that the relations themselves are skewed, for example, with some values occurring more frequently than others. The second broad category of skew is *partition skew*. Partition skew is caused by a join algorithm that poorly partitions the join workload and therefore loses some parallel speedup. Partition skew is possible even in the absence of AVS, so it is not enough to simply know the AVS properties of the joining relations.

Partition skew can be mitigated by using the proper algorithm, so Walton, Dale, and Jenevein focus on this category of skew. They further subdivide it into four separate types of skew shown in Table 1. *Tuple Placement Skew*, or *TPS*, occurs when the initial partitioning of a relation across the cluster is skewed. In this case, some nodes have more tuples than others, which causes an imbalance in scan times.

The second type of partition skew is *Selectivity Skew*, or *SS*. SS occurs when the number of selected tuples differs across nodes. An example of SS is a join involving an additional range predicate where the relations are range-partitioned across the cluster. The partitions that cover the selection predicate have many more candidate tuples than those that either partially cover or do not cover the selection predicate.

Redistribution Skew, or *RS*, is the improper redistribution of tuples across the cluster. When RS occurs, different nodes in the cluster hold different amounts of tuples to be joined. A bad hash function, or one that is not properly tuned for the join attribute distribution, can cause RS. Researchers tend to focus on solving RS since it is a direct property of the join algorithm.

The last type of partition skew is *Join Product Skew*, or *JPS*. JPS occurs when the number of matching tuples in the join differs at each node. JPS can be present even in the absence of RS, when all nodes have the same number of tuples before the join.

2.3 Solutions

Now that we have introduced several categories of parallel joins and join skew types, we can begin to discuss skew-resistant join algorithms. These algorithms can be generally classified by two broad categories [11]. *Skew resolution* techniques recognize and react to skewed parallel execution. On the other hand, *skew avoidance* techniques take proactive measures to prevent skew from occurring at all. As we will see, each algorithm focuses on solving skew in a particular situation.

2.3.1 Bucket Tuning

Kitsuregawa, Nakayama and Takagi [14] identified a potential skew issue in the single-node versions of GRACE hash join and hybrid hash join. Both of these join algorithms partition the joining relations up into buckets such that each bucket should fit entirely in memory. This avoids bucket overflow, and thus extra I/Os, in the case where all buckets are evenly sized. The problem with this approach is that buckets are statically chosen based on the properties of the joining relations. It can be the case that an upstream operator or a selection predicate applied to the join causes some buckets to be significantly larger than others. If this bucket size skew is severe enough, some buckets may overflow to disk which greatly reduces join performance.

Their solution, called *bucket tuning*, partitions the relations into a very large number of very small buckets, with the goal being that every bucket fits in memory regardless of bucket size skew. Since database systems perform I/O operations at the page level, this method can be inefficient if a bucket is smaller than a page. The bucket tuning strategy addresses this by combining very small buckets into larger buckets so that every bucket is at least one page large. The trick here is that buckets are combined *after* relations are partitioned and selection predicates are applied. The application of bucket tuning to hybrid hash join is called *dynamic hybrid GRACE hash join*, which chooses buckets dynamically rather than statically.

Kitsuregawa, Nakayama and Takagi compare dynamic hybrid GRACE hash join to hybrid hash join under three distributions of join attributes: triangular, Zipf, and uniform. They use the triangular and Zipf distributions to demonstrate the performance of the algorithm under skew. The uniform distribution represents a baseline comparison for the ideal case of no skew. They compute the number of I/Os analytically as a performance metric. Under the skewed dis-

tributions, dynamic hybrid GRACE hash join performs better than hybrid hash join. Under the uniform distribution, dynamic hybrid GRACE hash join and hybrid hash join are nearly identical in performance. The dynamic hybrid GRACE hash join algorithm represents an improvement over hybrid hash join, and all future work considers bucket tuning as an essential component of a bucketing hash join algorithm.

2.3.2 Bucket Spreading

While the bucket tuning solution discussed in Section 2.3.1 was originally proposed as a single-node algorithm, it can easily be extended to the parallel versions of GRACE hash join and hybrid hash join. Kitsuregawa and Ogawa [15] describe a parallel version of GRACE hash join with bucket tuning. In this algorithm, each node in the cluster holds a subset of buckets after partitioning and performs bucket tuning independently of the other nodes in the cluster.

The parallel GRACE hash join above has the property that tuples destined for a given bucket are read in parallel from the relation partitions and then converge on a single node. Kitsuregawa and Ogawa call this style of parallel join *bucket converging*. In contrast, a *bucket spreading* algorithm will scatter buckets across the cluster by further repartitioning the buckets into *subbuckets*, which are simply bucket fragments.

A bucket converging algorithm suffers from RS. Even though GRACE hash join with bucket tuning will create buckets that all fit in memory, there might be a significant difference in the total volume of buckets on each node. The bucket spreading technique combats RS by assigning buckets to nodes *after* bucket sizes are known. Buckets are evenly spread across nodes as subbuckets in the bucket forming stage. A coordinator node then performs bucket tuning and computes an optimal assignment of whole buckets to nodes. Finally, the subbuckets are gathered and joined in the bucket joining stage. Figure 1 illustrates the differences between bucket converging and bucket spreading on two nodes.

Bucket spreading can be tricky to implement because it requires subbuckets to be evenly spread, or *flattened*, across the cluster without prior knowledge of how many tuples for each bucket a given node will produce. Kitsuregawa and Ogawa solve this by utilizing an intelligent omega network, which is a network topology consisting of many 2x2 crossbar switches that can either be crossed or straight. Details are given in [15]. The switches in this network maintain counters that can be used to reassign ports in response to skewed traffic. The network effectively takes skewed input and produces evenly partitioned output.

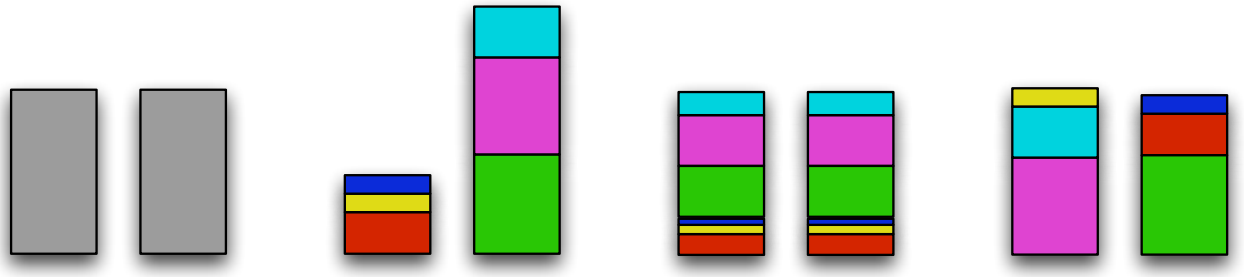
In the same work, Kitsuregawa and Ogawa evaluate both the bucket flattening mechanism of the omega network and the ability of bucket spreading to cope with skewed tuples. To evaluate bucket flattening, tuples are randomly assigned to buckets using a uniform distribution. Even though the bucket assignment is uniform, there is still small variation in bucket sizes. Using the mean standard deviation of bucket sizes as a metric, they find that without flattening this deviation increases linearly as the number of tuples increases. However, with the bucket flattening omega network, the deviation remains constant as the number of tuples increases, indicating that buckets can be effectively spread evenly across the cluster.

They compare bucket spreading to bucket converging by considering GRACE hash join on Zipf distributions of varying degrees of skew. They use the maximum number of I/Os on any individual node as the performance metric. In a low-skew environment, bucket spreading and bucket converging require roughly the same number of I/Os. However, as skew increases, the bucket converging strategy's I/O count greatly increases while bucket spreading strategy's I/O count increases more slowly. The reason for this increase is that a bucket converging algorithm will yield one node with a significantly higher volume of tuples than the others. This node will require many more I/Os to process all of its tuples. Bucket spreading, on the other hand, evenly distributes the bucket volume across the cluster, and therefore is a highly effective strategy for reducing the performance penalties caused by a Zipf distribution.

Schneider and DeWitt [26] also describe a parallel GRACE hash join algorithm using a technique related to bucket spreading. In their algorithm, a hash function, as opposed to an intelligent omega network, is used to spread buckets across the cluster. Their goal, however, is not to ensure even bucket spreading, but to gain maximum I/O throughput by spreading the bucket across all disks in the cluster. Indeed it is impossible to guarantee even bucket spreading using a hash function. Consider, for example, the case where all values are the same. In this case, a hash function will assign all tuples to the same subbucket. Nevertheless, the hashing technique has the advantage that buckets no longer need to be collected on a single node for joining. Each node can perform a local join on its subbuckets because matching tuples will map to the same subbucket in both relations.

2.3.3 Partition Tuning

While the bucket tuning technique can be thought of as a way to tune bucket sizes on a given node, the bucket spreading technique can be viewed as a way to tune partition sizes across the cluster. Hua and Lee [11] describe three algorithms using this general idea of *partition tuning*. Two of their algorithms are *skew avoidance* algorithms, meaning they prevent skew from occurring at all. They also include a *skew resolution* algorithm that initially permits skew but then later corrects it.



(a) Initial partitioning (b) Bucket converging (c) Bucket spreading (d) Subbuckets gathered

Figure 1: Tuple placements at various points during the execution of a parallel join with and without bucket spreading on two nodes. Relations are initially partitioned in some predefined way (a). A bucket converging algorithm (b) statically assigns buckets and may create uneven bucket volumes on each node. A bucket spreading algorithm evenly divides buckets into subbuckets (c) which can then be gathered into whole buckets more evenly (d).

The first skew avoidance algorithm with partition tuning is the *tuple interleaving parallel hash join*. Tuple interleaving parallel hash join is effectively identical to bucket spreading [15] except that the bucket flattening step is done in software rather than in an omega network. Buckets can be flattened by sending tuples to nodes in round robin order, thereby interleaving them across the nodes in the cluster. Hua and Lee state that processors are fast enough to do this interleaving in software, although it can also be done in hardware components such as specialized CPUs.

The second algorithm is a skew resolution algorithm called *adaptive load balancing parallel hash join*. As the name indicates, this algorithm adapts to skewed data by redistributing tuples across the cluster after partitioning. Whole buckets are initially hash-distributed to nodes without using any kind of spreading algorithm. After all buckets have been distributed, each node selects a subset of buckets that is close to its fair share of the data and then reports bucket information to a coordinator node. The coordinator node uses global information to compute an optimal reshuffling of excess buckets to equalize data across the cluster. After excess buckets have been reshuffled, each node independently performs bucket tuning and computes joins locally.

Adaptive load balancing parallel hash join has the property that its overhead, i.e. the amount of data is reshuffled, is proportional to the amount of skew in the joining relations. Under mild skew, adaptive load balancing parallel hash join will redistribute only a small number of buckets. Tuple interleaving parallel hash join, on the other hand, requires an all-to-all shuffle of data while buckets are gathered. It therefore pays the full shuffle overhead even under mild skew.

The last algorithm proposed by Hua and Lee is a skew avoidance algorithm called *extended adaptive load balancing parallel hash join*. This algorithm takes the adaptive load balancing parallel hash join a step further by computing the optimal bucket assignment earlier. Relations are initially partitioned into buckets that are written back to local disk without any network transfer. Bucket information is then sent to the coordinator, which computes an optimal bucket assignment. Finally, buckets are distributed according to this assignment and bucket tuning and local joins are performed. This algorithm effectively has no network overhead because buckets are transferred exactly once, so it works well under high skew. However, extended adaptive load balancing parallel hash join still has significant disk I/O overhead since an extra round of reads and writes is required for all tuples before any network transfer can begin.

Hua and Lee model the three algorithms analytically and evaluate them on a reasonable assignment of parameters for a parallel database system. Because the most skewed node is always the bottleneck, they assume a skew model where all nodes except for one have the same amount of data after partitioning, and the one node has more data after partitioning. As the data distribution varies from no skew, i.e. uniform, to full skew, where one node has all the data, they find that no single algorithm always wins. In the absence of skew, vanilla GRACE hash join performs the best, although it only slightly beats the next best algorithm. This is likely due to the overheads associated with the other algorithms. Under mild skew, adaptive load balancing parallel hash join beats GRACE hash join slightly. As expected, adaptive load balancing parallel hash join redistributes only a small amount of data under mild skew and still manages to equalize partitions, prevent-

ing bucket overflow. Under heavy skew, tuple interleaving parallel hash join and extended adaptive load balancing parallel hash join greatly outperform the other algorithms and are both equally good under the analytical model. The model assumes that different parts of the computation within each stage can be overlapped and that disk I/O dominates the running time. While extended adaptive load balancing parallel hash join has a much smaller network footprint, this has no impact on query response time since the network is not the bottleneck given the assumed system parameters.

In the context of the skew taxonomy provided in Section 2.2, these three algorithms all focus on solving RS. Any hash-based algorithm that splits relations into buckets has the potential for RS. Tuple interleaving parallel hash join handles RS by breaking the partitioning into two steps. The first step creates small sub-buckets on each node with interleaving, and the second step redistributes buckets evenly across the cluster. Adaptive load balancing parallel hash join mitigates RS by fixing skewed partitions after the fact using bucket redistribution. Extended adaptive load balancing parallel hash join uses a preprocessing bucketing step to compute optimal partitions, so RS never has a chance to occur.

2.3.4 Practical Skew Handling

In 1992, DeWitt et al. [8] proposed a new set of skew-resistant join algorithms. Unlike the previously discussed join algorithms, which were evaluated using analytical models and simulations, these new algorithms were actually implemented on the Gamma parallel database. Additionally, the algorithms are a marked departure from the previously discussed hash join algorithms. Instead of hash partitioning, these algorithms use sampling and range partitioning to avoid skew. They sample whole pages of tuples randomly from the inner relation. The samples are used to construct an approximate distribution of join values from which an even range partitioning can be computed.

The first algorithmic technique discussed by DeWitt et al. is *range partitioning*. The algorithm samples random tuples from the inner relation and constructs an approximate distribution of join values. Next it builds disjoint ranges from this distribution that evenly partition the sampled join values. Tuples are partitioned according to which range they fall in, and then nodes join their assigned tuples locally using standard hash join techniques.

Range partitioning by itself cannot create even partitions in the case when a common value is repeated many times. Consider, for example, the case where all tuples in one relation have the same value on the join attribute. Here, range partitioning will create a single partition containing all tuples. To combat this scenario, DeWitt et al. use a technique called *subset replicate*.

The key insight behind subset replicate is that it is possible to split a set of repeated values up into subsets, as long as all corresponding tuples in the other relation are copied to all partitions. Consider the example above, where all tuples in one relation have the same value on the join attribute. Suppose the other relation contains only a small number of tuples with this join attribute. The large number of repeated values in the first relation can be split into separate partitions, and the small number of corresponding tuples in the second relation can be copied to all of these partitions. Local joins can be computed on each partition, and their union will be the correct join output.

The simplest implementation of subset replicate splits a large set of repeated values into subsets of equal size. Since the first and last of the resulting partitions may also contain other values, uniform subsets may not actually create uniform partitions. Instead, a technique called *weighted range partitioning* is used. With weighted range partitioning, subset sizes are weighted by how large the resulting partitions will be, so a non-uniform collection of subsets will be used to create a uniform set of resulting partitions.

DeWitt et al. also provide a technique for achieving finer granularity partitions called *virtual processor partitioning*. This technique creates many more partitions than processors and statically assigns them to physical processors in round-robin fashion. This is very similar to GRACE hash join with bucket tuning [14], which creates many more buckets than nodes in order to spread tuples more evenly.

Virtual processor partitioning can alternatively use a dynamic assignment of partitions to processors. DeWitt et al. use the LPT scheduling algorithm to dynamically compute the assignment as an alternative to round robin.

They evaluate four algorithms: range partitioning, weighted range partitioning, virtual processors with round robin, and virtual processors with processor scheduling on the Gamma database system. Each algorithm implements subset replicate in its internal data structures. They compare to hybrid hash join as a good baseline. The hybrid hash join has the best performance on unskewed data due to low overhead, but its performance quickly degrades. Weighted range partitioning works quite well for mild skew, and virtual processing with round robin works well for moderate to heavy skew.

Because they only sample the inner relation, the proposed algorithms do not work well when the outer relation is skewed. In this case, hybrid hash join outperforms the other algorithms due to lower overheads. The other algorithms effectively sample the wrong relation and do not get a chance to learn about the skewed data.

An interesting result is that the performance of the

algorithm with the widest range of applicability, virtual processing with round robin, is roughly independent of the number of tuples sampled. DeWitt et al. state that the performance gain due to more accurate sampling is offset by the performance loss of actually sampling the data.

In the context of the skew taxonomy, range partitioning with sampling is a technique that mitigates RS. An algorithm that samples can more accurately create even partitions. Weighted range partitioning is an improvement that further reduces RS. Virtual processor processing can also help to create even partitions. However, DeWitt et al. motivate it as a solution to Join Product Skew, or JPS, as discussed in Section 2.2. If there is a mild to moderate amount of skew in both relations, even weighted range partitioning might not be able to separate the repeated values into enough partitions. In this case, there aren't enough repeated values to cause the subset replicate mechanism to spread the values across all nodes. However, a small number of repeated values in both relations can in the worst case cause a quadratic blowup in the magnitude of the join result. This is, by definition, JPS, and it can be solved by forcing the small number of repeated values to be spread across all nodes in the cluster with virtual processing partitioning. Even if there are only a few virtual processors per physical processor, there will still be enough partitions to evenly spread the data across the cluster.

2.3.5 Skew Handling in Sort-Merge Join

All of the algorithms discussed so far have been hash join algorithms. As mentioned in Section 2.1, there are other types of joins. Li, Gao, and Snodgrass [20] present several refinements to the sort-merge join style algorithm that improve skew resistance. Unlike many of the works discussed so far, this work does not focus on parallel algorithms, but rather focuses on the skew resistance techniques themselves.

Li, Gao, and Snodgrass are quick to mention that typical hash join algorithms suffer from bucket overflow in the presence of skew. While vanilla sort-merge join also suffers from skew, it has desirable performance properties when more than two relations are involved because the intermediate result relations are already sorted and can therefore skip the sort step. They are primarily interested in making sort-merge join skew-resistant so they can take advantage of these performance properties.

An implementation of vanilla sort-merge join may require tuple rereads in the presence of skew. This is essentially the same problem faced by nested-loop joins. If a value is repeated in both relations, its corresponding tuples in the inner relation will have to be reread once for each matching tuple in the outer relation. Depending on the implementation, the I/O cost can be enormous.

A typical optimization to the above problem is block orientation. Li, Gao, and Snodgrass present a block oriented algorithm called *R-1*. R-1 reads a block of tuples at a time from disk. If the inner relation is skewed and the skew is contained within a block, no extra I/Os are required since the repeated values already exist in memory. If the skew crosses block boundaries, all inner relation blocks containing tuples for a particular value may need to be *reread* for each matching tuple in the outer relation if. These rereads will be required in the case where the older blocks have been evicted due to memory pressure.

Here we note that in a modern system with large amounts of memory, the operating system may be able to keep old blocks in a buffer cache and prevent the rereads from touching disk. However, Li, Gao, and Snodgrass are primarily interested in special purpose database systems that tend to manage memory from within the application. The database system can use application-specific knowledge to make better use of memory than a traditional operating system using LRU replacement. Thus it is useful to consider application-level memory management techniques such as block rereading.

An alternate implementation of R-1 operates on multiple sorted runs per relation instead of a single sorted run. This algorithm is called *R-n*. R-n overlaps the join operation with the last phase of the merge-sort for extra efficiency. This efficiency comes at the cost of a trickier implementation. R-n also has the possibility of incurring more random reads since skewed tuples may be spread across multiple sorted runs.

R-1 and R-n require block rereads for every matching tuple in the outer relation. An improvement over R-1 is *BR-1*, which joins every tuple in a block in the inner relation to the entire block in the outer relation. This strategy is analogous to loop tiling optimizations used to improve cache performance [31]. BR-1 will only incur inner relation rereads every time a new *block* in the outer relation is read, rather than once per *tuple*. The *BR-n* algorithm similarly extends BR-1 to handle multiple sorted runs.

An improvement on BR-n is *BR-S-n*, which does block rereads but makes smarter use of memory. When skew in both relations is detected, the previously joined values in the memory-resident blocks are discarded and the tuples corresponding to the current join value are shifted to the top of memory. This has the effect of fitting more of the joining tuples in memory, which reduces the number of reread I/Os.

In the same work, Li, Gao, and Snodgrass discuss a different strategy called *spooled caching* that makes good use of memory. In the *SC-1* algorithm, tuples that

satisfy selection predicates, but may or may not actually be joined, are stored in an in-memory cache. This possibly prevents rereads since a tuple is only placed in the cache if it has the possibility of satisfying the join condition. If the number of such tuples is small, the cache may be able to hold all of them. On the other hand, if the cache overflows it is spooled to disk, so rereads are still required under heavy skew. In this case, only those tuples that can satisfy the join condition will be reread, so this is still an improvement over rereading everything. As an optimization, tuples from the inner relation are immediately joined with the current block of the outer relation, and if skew is detected in both relations, those tuples are added to the cache. As one might expect, the *SC-n* variant implements spooled caching for multiple sorted runs. It is more complicated and requires caching if skew is detected in any one of the sorted runs in the outer relation.

The last algorithm is *BR-NC-n* which is similar to BR-n but uses a cache that is *not* spooled to disk. Instead, when the cache fills up, blocks are simply reread as in BR-n. This algorithm has the slight advantage over BR-n in that if the number of skewed tuples is small enough to fit in the cache, but larger than a single block, no rereads will be required. Compared to SC-n, BR-NC-n must reread more tuples since it rereads from the actual relations. However, BR-NC-n does not need to flush cache blocks to disk.

Li, Gao, and Snodgrass evaluate their algorithms on the TimeIt [16] database prototyping system. They measure performance under two types of skew which represent extremes over of the spectrum of distributions. The first type, *smooth skew*, is a lightly skewed distribution where some join values have two tuples and the rest have one. The second type, *chunky skew*, has a single join value with a large number of tuples, and the rest of the values have a single tuple. They use the same relation for both sides of the join, so skew is present in both the inner and outer relations.

The experiment for smooth skew shows that SC-n, BR-NC-n and BR-S-n are all good algorithms, with SC-n being slightly better than the others. R-n and BR-n are both a little worse than the others, although the slowest algorithm is only 11% slower than the fastest. This difference is likely due to the fact that R-n and BR-n will have to perform rereads if one tuple is in one block and the other tuple is in the next block. The other algorithms use memory tricks to avoid paying this I/O cost. The *-1* algorithms that operate on a single sorted run from each relation are uniformly worse than their multiple-run *-N* counterparts due to the overlapping of the merge and join steps, so they are not evaluated.

The results are more startling for chunky skew. In this case, all of the multiple-run algorithms are roughly the same except for R-n, which is several times worse than the others. When a single join value has many tuples, R-n requires multiple rereads for each of the tuples in the corresponding tuples in the outer relation. In the presence of chunky skew, these rereads dominate the running time of the algorithm.

In the absence of skew, all of the algorithms are roughly the same. This indicates that the bookkeeping overhead of the more sophisticated algorithms is negligible. While it would be interesting to see a comparison of the memory usages of each algorithm, the authors fix the memory size at 16MB and each algorithm uses all of the available memory. Differences in algorithmic performance might manifest on systems with larger memories, but this is outside the scope of [20].

While the above algorithms are not parallel join algorithms, we can still glean some key insights. The algorithms essentially focus on solving a problem related to JPS. When the join product is very large, additional I/Os may be required in systems without adequate memory. These I/Os can dominate the computational time of the join if care is not taken when selecting the join algorithm. Li, Gao, and Snodgrass show that it is feasible to design a sort-merge join algorithm that avoids most of these I/Os in the presence of skew in both relations.

2.3.6 Partial Redistribution Partial Duplication

While many skew-resistant join algorithms have been discussed so far, Xu et al. [33] from Teradata published a paper in 2008 that states that these algorithms are too difficult to implement correctly in practice. As a result, parallel database software simply does not handle skewed data well. They discuss a simple algorithmic modification that mitigates skew and is practical to implement in real software.

Xu et al. characterize two types of parallel hash join strategies. The first type, called *redistribution*, involves redistributing tuples in the joining relations based on a hash of the join attribute. This is very similar to the simple hash join algorithm presented by Schneider and DeWitt [26] and discussed in Section 2.1. As mentioned before, redistribution suffers a performance penalty when there is intrinsic skew in the join attribute of one or both relations. In this case, some nodes will receive more data than others. Adding more nodes does not help much, and in fact increases the degree to which a *hot node* is overloaded relative to the other nodes.

The second strategy, *duplication*, works well when one relation is much smaller than the other. With duplication, the smaller relation is copied to all processing nodes, which then can perform local joins between this copy and their portion of the larger relation. Duplication does not suffer from hot spots in the same way redistribution does, since the relations are never partitioned on the join attribute. However, significant net-

work and storage I/O costs are required if neither relation is very small.

Redistribution and duplication can be combined into a hybrid algorithm called *partial redistribution partial duplication* . This algorithm attempts to gain the benefits of both techniques by redistributing some tuples and duplicating others. Skewed tuples, i.e. a set of tuples with the same join attribute, are stored locally at partition time. The tuples in the other relation with the corresponding join attribute are duplicated to all nodes. Non-skewed tuples are redistributed using a standard hash function. After this partitioning, each node computes three joins and unions them together: redistributed tuples with redistributed tuples, locally skewed tuples with duplicated tuples, and duplicated tuples with locally skewed tuples.

Xu et al. evaluate partial redistribution partial duplication on a cluster of 10 nodes, each hosting 8 virtual processing units. They compute a join on two relations and artificially set the join attribute in one of the relations so that a large fraction of the tuples contain the same join attribute. As this fraction varies from 0% to 40%, the traditional redistribution algorithms suffers linear slowdown, while partial redistribution partial duplication's performance remains constant.

The baseline redistribution algorithm is related to the simple hash join and GRACE hash join algorithms. All three of these algorithms suffer from RS. In order to avoid redistributing too many tuples to a given node, Xu et al. apply fine-grained duplication to only those problematic tuples. Because the skewed tuples are not redistributed, the initial tuple placement determines which nodes must join these tuples, so a vanilla implementation of partial redistribution partial duplication can suffer from TPS. To avoid TPS, the algorithm is modified to randomly redistribute the skewed tuples, thus spreading them evenly across the cluster at the cost of slightly higher network and disk I/O. Because the corresponding tuples in the other relation are duplicated everywhere, this modified algorithm is correct and avoids hot spots caused by TPS.

2.3.7 Outer Join Skew Optimization

All of the previously discussed algorithms focus on inner joins. Xu and Kostamaa [32] solve the issue of skew in outer joins, which are prevalent in business intelligence tools.

Outer join skew manifests itself in the computation of multiple joins. Even if the first join does not suffer from any skew problems and its output is evenly distributed across the cluster, outer join skew can be a problem. If the subsequent join attribute happens to be the same as the first join attribute, any dangling rows will contain `NULL`s on this join attribute. Since the subsequent join is partitioned on its join attribute, these `NULL`s will be redistributed to the same node. This node will then contain many more tuples than the other nodes and will bottleneck the parallel join.

The algorithm given by Xu and Kostamaa in this work, called *Outer Join Skew Optimization*, or *OJSO*, effectively handles skew caused by outer joins. OJSO treats tuples with `NULL`s as a special case. A tuple containing a `NULL` on the join attribute is saved locally, while all other tuples are redistributed as normal. Local joins are computed only on the redistributed tuples. The final result is the union of the joined output with the locally saved tuples containing `NULL`s. OJSO's output is correct because tuples containing `NULL`s cannot join with other tuples by definition, so they need not be redistributed.

Xu and Kostamaa evaluate OJSO on a cluster of 8 nodes where each node hosts 2 virtual processing units. They measure the execution time of a three-way join and vary the fraction of dangling rows, i.e. tuples containing `NULL`s, from 0% to 70%. They find that OJSO's performance is constant as outer join skew increases, whereas a conventional outer join algorithm slows down linearly with the amount of outer join skew.

Since all previously described algorithms have been inner join algorithms, it is a little challenging to compare OJSO with the others. The technique of treating some tuples as a special case is related to Xu et al.'s treatment of skewed tuples in partial redistribution partial duplication [33]. In the partial redistribution partial duplication algorithm, skewed tuples are saved locally and their corresponding tuples in the other relation are duplicated, so a join can be computed. While OJSO does not require duplication, it still uses the technique of saving some tuples locally rather than redistributing them.

At its heart, the problem of outer join skew is a form of RS that manifests in joins computed after the first. In this case, the `NULL`s are unevenly redistributed across the cluster, causing hot spots. However, one can also view this as a variant of JPS, since `NULL`s effectively do not join with anything so any partition receiving `NULL`s will have a different join product size than a partition that does not receive `NULL`s.

3. MAPREDUCE SYSTEMS

While parallel databases are suitable solutions for enterprise operations, they are very expensive at warehouse-scale. Large web companies such as Google and Microsoft opted instead to build their own systems. In particular, the MapReduce [5] framework has become popular in the past few years. Developed by Google, this framework allows programmers to write single-threaded `map` and `reduce` functions which the framework then automatically parallelizes across the cluster. A few years after the announcement of MapRe-

duce, Apache Hadoop [34], an open source alternative, was developed. Given the choice between expensive parallel databases, or free open-source parallel computation frameworks like Hadoop, other companies like Yahoo! and FaceBook adopted the MapReduce framework. Around the same time, Microsoft Research developed its own parallel dataflow execution engine called Dryad [12], which can, among other things, run MapReduce jobs. Since then, many companies have embraced the MapReduce paradigm, and higher level tools have been built on top of it, such as Pig [9] and Hive [29].

In addition to providing a parallel computation framework, MapReduce integrates with a distributed file system such as Google File System [10], or the Hadoop Distributed File System, HDFS [34]. These storage solutions differ from a traditional relational database in that they offer access to unstructured files, rather than adhering to schemas. Relational databases also offer access to indexes that are absent in MapReduce systems. Finally, since the `map` and `reduce` functions are *User Defined Functions*, or *UDF*s, automatic optimizations are nontrivial. These differences necessitate skew-resistant solutions that are slightly different from the parallel join solutions presented earlier. This section will survey skew mitigation techniques in MapReduce systems and how they relate to each other and the previously discussed parallel join techniques

3.1 Skew in MapReduce

MapReduce clusters are typically built from large numbers of unreliable commodity components. The degree of hardware unreliability in itself is a type of skew that MapReduce tackled from its inception. For example, if a node in the cluster has a faulty disk drive, it may write `map` output to disk at a much slower rate than its healthy counterparts. If the job is partitioned evenly across nodes, this slow node will take much longer to accomplish its task and may become a bottleneck for the entire job. Dean and Ghemawat identified these slow nodes as *stragglers* [5].

A typical strategy for handling skew caused by stragglers is *speculative execution* [5, 34]. A *backup copy* of a long-running task is speculatively executed on another node before the original task finishes. If the backup copy finishes first, the original task is killed, and vice versa. The hope with this strategy is that the task was slow because of faulty hardware, and so the backup on the newly selected node will finish before the original task because it will *not* have faulty hardware with high probability.

In addition to hardware-related skew, MapReduce can also exhibit skew in the data or computation. Skew manifests itself differently depending on the phase of the MapReduce job. Kwon et al. [18] identified three types of *map-skew*. The first is a type of computational skew called *expensive record skew*. When expensive record skew is present, some records take significantly longer to process than others. A good example of this is a MapReduce implementation of the PageRank [22] graph analysis algorithm. In PageRank, there are two kinds of records: small contribution records and huge graph structure records. The presence or absence of these structural records in an input partition can greatly skew the processing time for an individual `map` task.

Another type of map skew is *heterogeneous map skew*. A heterogeneous map reads multiple input sources in a single task. An example of a job with heterogeneous maps is CloudBurst [25], a DNA alignment algorithm modeled on RMAP [27] that attempts to align a set of reads to a reference genome. The `map` logic treats reads and reference records differently, yielding bimodal performance characteristics. In this case, it can be difficult to create evenly partitioned `map` tasks without application-specific knowledge.

A third type of map skew is *non-homomorphic map skew*, which occurs when the `map` function must operate on a group of records rather than processing them one-by-one in a streaming fashion. These `map` functions performs `reduce`-like logic. Some clustering algorithms used in scientific computing fall into this category.

Kwon et al. also identified two types of reduce skew. The first is *partitioning skew*, which is akin to RS in parallel databases. Under partitioning skew, the intermediate files that result from the `map` output are unevenly partitioned across the `reduce` tasks in the cluster. This can be caused by a bad hash function that unevenly partitions the `map` output. Even with a good hash function, however, duplicate keys can cause some partitions to be larger than others since the semantics of `reduce` dictate that all records of a given key must be present in the same intermediate file. Unfortunately it is generally impossible to predict the intermediate key distribution without at least partially running the `map` function because `map` is a UDF and is entirely application-specific.

The second type of reduce skew is *expensive record skew*. This skew is more severe than its `map` counterpart due to the fact that an invocation of `reduce` operates on collections of records rather than individual records. If the `reduce` function must perform any comparisons between the values associated with a key, the processing time will be super-linear in the size of the record, creating a significant source of computational skew. The degree to which this and other types of skew affect a MapReduce job depends of course on the particular application and input data.

3.2 Solutions

In Section 3.1 we discussed several types of skew that occur in MapReduce systems. Now we will present solutions proposed by the MapReduce community that

tackle these various types of skew.

3.2.1 LATE

While speculative execution as discussed above addresses the problem of stragglers, it does have some shortcomings. Speculative execution, as implemented in Hadoop [34], causes a task to be duplicated toward the end of the job if its measured progress is a below some threshold. There are several problems with this approach. The first is that a task might be intrinsically slow, so a backup copy may not actually help reduce the job's completion time. A second problem is that outlier tasks are not identified until they have already run for a long period of time. By the time a task is identified as slow, it may have already wasted significant cluster resources.

Zaharia et al. [36] present an improved scheduler for Hadoop called *Longest Approximate Time to End*, or *LATE*. LATE uses a more intelligent speculative backup mechanism than the stock Hadoop scheduler. In particular, it is designed to handle the case of clusters containing heterogeneous hardware configurations. Hadoop's default backup mechanism schedules backup copies of tasks that are some amount slower than the average currently running task. In a heterogeneous environment, any task running on older hardware will be considered too far below the average and will be speculatively executed, leading to significant resource waste. Zaharia et al. state that heterogeneous environments are actually the common case as older hardware is incrementally replaced, or in a virtualized environment where customers compete for shared resources.

LATE, like Hadoop, uses heuristics to determine when tasks should be speculatively executed. However, Hadoop's mechanism is based on a `progress score`, which varies from 0 to 1 and roughly corresponds to the fraction of the task that has been completed. Since the progress score increases over time, Hadoop can only schedule backups if a task fails to make progress long enough for it to be noticed as slower than average. More concretely, Hadoop will consider a task as slow if its `progress score` is less than 0.2 below the average. LATE, on the other hand, uses a `progress rate` metric, which is defined as `progress score / T` where `T` is the elapsed time of the task. Using the progress rate metric allows LATE to notice immediately when a task is progressing slower than it should be. In particular, LATE estimates the task's time to completion as `(1 - progress score) / progress rate`, and then uses this metric to decide which tasks to backup first. The intuition is that the task that will finish furthest in the future has the most potential for the speculative backup to overtake the original task and improve the job's overall completion time.

Zaharia et al. evaluate LATE both on a large cluster of Amazon EC2 nodes and on a small, local testbed. They compare LATE with Hadoop with and without speculative execution. They test their configuration using sort, grep and word count as application benchmarks. They observe cluster heterogeneity, and therefore hardware-related skew, by measuring the number of virtual machines per physical machine. LATE is up to a factor of two better than Hadoop with speculative execution. In some cases, speculative execution actually decreases the performance of Hadoop.

3.2.2 Mantri

While LATE offers a significant improvement over plain speculative execution, it does not go far enough to solve the problem. If an outlier task occurs early on in the job, LATE will not be able to detect it because speculative execution does not occur until the end of the job. Furthermore, the only course of action LATE can take is speculative backup, which may not be the most efficient response, especially if the problem is not directly related to the slow node.

Ananthanarayanan et al. [3] solve the problems of speculative execution with a system called *Mantri*. Mantri is an intelligent outlier-response system that actively monitors tasks in a cluster and takes action dependent on the identified cause of slowdowns. Rather than using a one-size-fits-all approach, Mantri performs a cost-benefit analysis at the task level to determine whether to take action or not. It also acts early, which prevents early outliers from slipping through the cracks.

Mantri uses two methods of task-level restarts. The first, called *kill and restart*, kills a task that is identified as an outlier and restarts it on a different node. Kill and restart has the benefit of not requiring an extra task slot, but requires that the restart must actually save time with high probability. Another method is *duplicate*, which schedules a backup copy much like speculative execution. The duplicate method uses the original task as a safety net in case the backup copy does not actually save time. In this case, Mantri will notice that the backup is also slow and will kill it off. Any progress made by the original task is maintained because it was duplicated and not killed.

When scheduling tasks, Mantri uses network-aware placement to try to prevent hot spots in the network. The main network-bottleneck that needs to be avoided is the shuffle phase that occurs before a `reduce` task can begin. Since Mantri knows about `map` outputs, it can intelligently assign `reduce` tasks to racks in order to prevent downlink congestion on racks hosting `reduce` tasks.

Another problem caused by hardware failures is recomputation. When a task's output is used by a later task, such as a `reduce` task using `map` output, it can be the case that the node storing this output fails af-

ter the first task completes but before the second tasks starts. In this case, the intermediate data files are lost and must be recomputed. Mantri reduces the penalty of recomputation by selectively replicating intermediate data if the probability of machine failure causes the cost of recomputation to exceed the cost replication. Mantri uses failure history over a long period of time to compute this probability. Additionally, if a machine does fail, all of the tasks that had output stored on the machine are proactively recomputed to reduce delay when these files are eventually requested later in the job.

The final component of Mantri is a scheduler for long-running tasks. Unlike MapReduce with speculative execution, which will schedule backups of long running tasks, Mantri will leave long tasks alone as long as they are making progress at a sufficient rate based on their input size. Mantri also schedules long tasks early, which bounds the overall completion time of the job.

Ananthanarayanan et al. evaluate Mantri on a Bing cluster consisting of thousands of servers. They compare jobs running with Mantri to prior versions of the jobs running before Mantri was enabled using a simulator. They also evaluate Mantri on some benchmark applications. They evaluate Mantri's outlier mitigation strategies and compare them to Hadoop, Dryad, and LATE. They find that Mantri significantly improves the completion time of actual jobs and is noticeably better than the other outlier mitigation algorithms.

Compared to previous works, Mantri is a very sophisticated solution to the problem of hardware-skew in MapReduce systems. Speculative execution is about the simplest possible solution, and its effectiveness is limited. LATE is more sophisticated, but can still only take one possible action. Mantri, on the other hand, uses cost-benefit analysis to determine which response will likely be the most effective. As a result, it outperforms the other systems.

While Mantri is pitched as a solution to the problem of outliers caused by hardware, it actually is intelligent enough to cope with partition skew. Mantri's ability to take task input size into account when determining outliers allows it to tolerate skewed partition sizes. Consider, for example, a `reduce` task with an abnormally large partition size. Basic speculative execution will identify this task as a straggler and will schedule a backup. While the task technically is a straggler, the backup has no hope of overtaking the original since the task is inherently slow due to data partitioning skew. Mantri will leave the task alone, which is the correct course of action. While Mantri does not proactively fix partitioning skew, it does not perform poorly in the face of such skew.

3.2.3 SkewReduce

While the above systems focus on hardware-related skew, there are several other types of skew that impact MapReduce systems. One such type of skew is *computational skew*. An application exhibits computational skew if some records take longer to process than others. Such skew can manifest even the data is evenly partitioned.

Kwon et al. [17] solve a particular type of computational skew that arises in scientific computing using a system called *SkewReduce*. Many scientific computing algorithms perform some kind of feature extraction using clustering on multidimensional data. Kwon et al. cite the example of searching astronomical imaging data for galaxies. The amount of work required to recognize a galaxy depends on how close the data points are to each other. In this sense, two partitions that contain the same number of points may have vastly different processing times if one is sparse and the other is dense.

SkewReduce is a feature-extraction framework built on top of MapReduce. Like MapReduce, SkewReduce allows users to write application logic as if it were to be executed on a single processor. SkewReduce then automatically executes this logic in parallel to eliminate the burden of writing parallel code. The SkewReduce API consists of three functions: `process`, `merge`, and `finalize`. The `process` function takes as input a set of data points and outputs a set of features, along with any data points that are no longer needed. Next, `merge` combines feature sets from previous `process` or `merge` executions to create a new one and possibly throws out more data points that are no longer needed. Lastly, the `finalize` function takes the fully merged set of features and applies it to any data points if needed, for example by labeling points with their cluster information. These functions are implemented as MapReduce jobs in Hadoop.

In order to efficiently handle computational skew, SkewReduce applies two user defined cost functions to a small sample of the input data. The user is required to supply functions that estimate the cost of `process` and `merge`. The optimizer then takes the sample data and creates an even partitioning in a greedy fashion by repeatedly splitting the most expensive partition. Each partition initially starts as a `process` job but becomes a `merge` job when split. The optimizer uses the cost functions to find the optimal splitting point, and also to determine if splitting a partition improves execution time. Partitions are split until every partition fits in memory and splitting does not further reduce execution time.

Kwon et al. evaluate SkewReduce on an 8 node Hadoop cluster where each node also serves HDFS. They use the LPT scheduling algorithm and give cost functions that compute the sum of squared frequencies over a histogram of the sampled data. They compare SkewReduce's optimizer to a query plan using uniform

partitioning with varying partition granularity. For reference, they also manually tune a query plan by hand. SkewReduce's optimizer beats all other query plans, and beats the uniform query plans by more than a factor of 2. They also find that a sample rate of 1% is sufficient to reduce computational skew and increase performance.

SkewReduce is unlike other skew-resistant solutions in that it focuses on computational skew, whereas most other solutions focus on hardware-related skew or data skew. SkewReduce's ability to cope with skew stems from its use of input data sampling, a popular technique used by DeWitt et al. [8] that was discussed in Section 2.3.4. SkewReduce is somewhat unique in that it does not apply the actual `process` and `merge` functions to the samples, but rather uses the user defined cost functions to more efficiently probe the space of partitions.

The scientific computing problems that SkewReduce tackles have unusual MapReduce implementations that exhibit non-homomorphic skew as classified by Kwon et al. [18]. Essentially, a `map` function that does clustering does not stream records, but rather computes a result based on large collections of records. The properties of these records relative to others in the collection is the central cause of computational skew. In particular, how close or far one data point is from another in a multidimensional space determines how long such a `map` function will take to execute.

3.2.4 Scarlett

Another type of skew that is somewhere between hardware-related skew and data skew is *content popularity skew*. Ananthanarayanan et al. [2] address popularity skew with *Scarlett*, which is an augmentation to existing distributed file systems such as Google File System or HDFS. Content popularity skew occurs when certain files on distributed storage are more popular than other files and are accessed more frequently. It is not quite data skew, since any individual MapReduce job may not be skewed. However, the collection of all jobs executing simultaneously on a cluster may access some pieces of data more than others. Similarly, it is not quite hardware-related skew since all machines may be equally fast, but some machines will become hot spots simply due to increased data demand. Such skew arises in practice in log processing, for example, when some logs are interesting and the rest are not.

Scarlett uses the temporal locality of data to predict popularity skew. Every 12 hours, the prior 24 hours of concurrent accesses to a given file is used to update its predicted popularity. Scarlett then adjusts the replication factor on a file-by-file basis to reduce contention from future concurrent accesses. In order to prevent too many tasks from being scheduled a single rack, Scarlett departs from existing replication schemes and spreads new replicas evenly across racks in the cluster. In particular, the new replica is created on the least loaded machine on the least loaded rack. This heuristic prevents popular files from reducing the effective availability of other files stored on other machines in the same rack.

Increasing the replication factor for a file causes extra network traffic that can interfere with running jobs and cause even more contention. To prevent such contention when a large number of replicas need to be added, Scarlett begins by adding a small number of replicas and exponentially increases the number of new replicas as new source racks become available. In addition, Scarlett uses compression to trade spare computational resources for extra network bandwidth.

Ananthanarayanan et al. evaluate Scarlett in two environments. They implement Scarlett as an extension of HDFS in Hadoop and run some benchmark Hadoop jobs driven by trace data from a Dryad cluster. Additionally, they run the actual trace data through a Dryad simulator to confirm Scarlett's ability to improve real workloads. They find that Scarlett speeds up the median Hadoop job by more than 20%, and it speeds up the median Dryad job by more than 12%. Additionally, they measure the network overhead imposed by Scarlett to be less than 1% of the total traffic.

Since Scarlett solves popularity skew, which is rather unique in nature, it is difficult to compare to other systems. Scarlett is most similar to techniques that improve read locality in MapReduce, such as delay scheduling [35] and Quincy [13]. These systems both focus on the locality and fairness. While Scarlett is primarily concerned with skew mitigation, it does so by increasing data locality. In terms of the skew already identified, popularity skew is related to the straggler problem caused by faulty hardware. A storage server in a distributed file system that contains a hot block will effectively be overloaded and will offer lower per-task throughput than a server that only serves cold blocks.

When combined with systems such as speculative execution or the kill-and-restart semantics of Mantri [3], tasks that read data from one of these hot storage serves will be considered slow and the system may try to schedule a backup. In this case, backups will actually reduce overall job performance since new tasks must compete for limited read bandwidth. The correct course of action when a server is overloaded due to content popularity skew is not to start new tasks, but to fix the root problem by increasing replication in the distributed storage system. Increasing replication across the board is far too expensive to be useful in practice, so systems like Scarlett represent a good compromise. Hot files will benefit from increased replication, whereas cold files will maintain the minimum number of replicas to satisfy a

given availability guarantee.

3.2.5 SkewTune

Handling data skew and computational skew in vanilla MapReduce is a challenging task since the system is driven entirely by user defined functions. The user must have expert knowledge of the application workload and then must design an ad-hoc partitioning method to mitigate skew. Kwon et al. [19] solve this with an alternative MapReduce implementation called *SkewTune*. SkewTune effectively solves the issues of data and computational skew in many circumstances and does so without altering the MapReduce API. It therefore enables users to run unmodified MapReduce programs and achieve the performance benefits of ad-hoc partitioning without all of the costs.

SkewTune solves data skew and computational skew by repartitioning straggler tasks into smaller tasks that can be spread across the cluster. If the `map` and `reduce` functions operate independently on individual records and key groups respectively, repartitioning a task's input will guarantee the output stays the same. Put another way, tasks can be safely repartitioned as long as `map` and `reduce` are *pure functions* without side effects. Furthermore, by using range partitioning as the repartitioning mechanism, SkewTune leaves the ordering unchanged as well, so the final output is identical to execution without SkewTune.

After all tasks have been scheduled, SkewTune begins monitoring for tasks to repartition. When a slot frees up, SkewTune determines the task with the longest remaining time. It carves up the task's remaining input into many small disjoint intervals. It then assigns some of these intervals to a task to run on the free slot. SkewTune uses estimates of remaining time to guess when other tasks will finish and prepares intervals for them to process as soon as they free up. In this way, a task is effectively repartitioned across the entire cluster, which allows for maximal slot utilization.

Kwon et al. evaluate an implementation of SkewTune in Hadoop on a 20 node cluster where each node also functions as an HDFS storage server. They evaluate SkewTune on three applications: inverted index, PageRank, and CloudBurst. They find that SkewTune achieves high performance even in the presence of poor configuration settings. If, for example, the user asks for too few reduce tasks, SkewTune can compensate by repartitioning tasks. SkewTune also reacts to heterogeneous map tasks in CloudBurst by splitting tasks that process the more expensive reference data set.

Unlike the systems examined so far, SkewTune does not solve stragglers by scheduling backups [5, 36, 3], or by killing and restarting tasks [3]. Rather, SkewTune splits long-running tasks into multiple pieces while saving the partially computed output. SkewTune is therefore able to avoid wasted computation entirely while spreading skewed tasks evenly across the cluster.

It is unsurprising, given the authors, that SkewTune solves the types of skew mentioned above. Kwon et al. [18] earlier categorized the types of skew that occur in MapReduce systems. SkewTune directly attacks the problems of expensive record map skew and heterogeneous map skew, as well as partitioning reduce skew and expensive record reduce skew. The fifth type of skew identified in [18], non-homomorphic map skew, is not solved by SkewTune. In fact, SkewTune will not operate correctly if the `map` function is non-homomorphic. To see why, consider a `map` function that operates on a collection of records rather than on individual records. In this case there is no obvious way to split a `map` task's input and still retain the correct output since some output may be dependent upon the interaction of one record in the first split and another record in the second split. Any attempt to repartition a `map` task could therefore change the job's output, so SkewTune cannot be used as a drop-in replacement for MapReduce in this case.

The techniques employed by Kwon et al. are similar in nature to the parallel join techniques proposed by Hua and Lee [11]. Specifically, the adaptive load balancing parallel hash join algorithm reacts to skewed partitions by redistributing buckets across the cluster to create an even partitioning. Kwon et al. take this a step further by not only redistributing tasks, but also by splitting them into smaller pieces for load balancing.

SkewTune, like speculative execution [5] and LATE [36], takes action towards the end of a job. Kwon et al. are primarily focused on making sure every slot in the cluster has a task to process, and acting towards the end of the job satisfies this condition. However, as discussed by Ananthanarayanan et al. [3], such a system will miss outlier tasks that are scheduled towards the beginning of the job. It might be possible to adapt SkewTune to act earlier and possibly reduce job completion time using the lessons learned from Mantri, although this is not discussed in [19].

3.2.6 A Study of Skew in MapReduce

In Section 3.1 we listed several types of skew in MapReduce identified by Kwon et al. [18]. In their paper, the authors discuss five "best practices" for building MapReduce applications that are resistant to skew. Here we will discuss these best practices and how they relate to the solutions presented so far.

The first piece of advice given by Kwon et al. is to avoid the default hash-partitioning map output scheme in Hadoop. Instead, users are directed to use range partitioning or some application-specific ad-hoc partitioning scheme. It is incredibly difficult to create a skew-resistant algorithm using hash partitioning [14, 15, 11, 8, 33, 32, 19]. SkewTune [19] uses range partitioning to

achieve a more even data split and to facilitate repartitioning. Kwon et al. mention that ad-hoc partitioning schemes may be required in the case of a *holistic* `reduce` function, which is a `reduce` function that buffers records in-memory and makes comparisons across key groups and is therefore dependent on the data distribution. An example is CloudBurst's `reduce` function, which compares the values from key groups in the reference and read data sets. An ad-hoc partitioning function might be able to use application-specific logic to limit the degree of skew, although this requires significant domain expertise from the user.

A second suggestion is to experiment with different partitioning schemes either between runs or dynamically during a run. In the first case, a production job might have smaller debug runs, or even large production runs with accompanying log data. The logs from these previous runs can inform the framework how it ought to adjust the partitioning function to create more even partitions. In the second case, previous log data may not be available, but if the partitions can be adjusted dynamically then it may still be feasible to evenly distribute the data. From the partition tuning work discussed earlier [11], the adaptive load balancing parallel hash join algorithm can react to a bad partitioning by redistributing buckets from overloaded nodes to underloaded nodes. In the context of MapReduce, SkewTune [19] effectively implements this solution by repartitioning large partitions dynamically without any previous log data.

The third solution proposed by Kwon et al. is the use of a `combiner` function. A `combiner` implements `reduce` logic on the map output file before data is shuffled across the network. This has the effect of reducing network traffic and consequently shrinks the sizes of all reduce partitions. By reducing the size of the data, a `combiner` reduces the severity of skew and improves performance. Kwon et al. are quick to mention that a traditional `combiner` task will use extra CPU and disk resources, so it is best to implement `combiner` logic inside the `map` function. It should be noted, however, that it may not be feasible to use a `combiner` if the `reduce` function is not commutative and associative since the `combiner` will be applied to a collection of unrelated partition fragments as opposed to one whole partition.

While there is no direct analogue of a `combiner` for parallel join algorithms, most database optimizers apply selection and projection operations as early as possible in order to reduce the amount of data passed to the join operator. The rationale for this optimization is essentially the same as the reason for using a `combiner`. The SC-1 and SC-n algorithms proposed by Li, Gao, and Snodgrass in their work on sort-merge joins [20] make use of a cache of tuples that satisfy additional join predicates. The cache effectively allows the algorithm to skip over uninteresting tuples when joining new tuples with old tuples. This optimization has the same spirit as a `combiner` in that the size of the data is reduced early on in order to avoid additional overhead and reduce the impact of skew later in the algorithm.

The fourth recommended practice is a preprocessing step that is applied before a long-running job actually executes in production. This is especially useful if it is known in advance that a job will be executed multiple times. The data can be repartitioned offline to eliminate skew in the actual job. If offline processing is not feasible, it is still possible to implement preprocessing just before the actual job begins. SkewReduce [17], effectively implements this preprocessing step by searching the partition space for an even partitioning and query plan. It uses cost functions estimate the actual running time of the job and creates an even partitioning plan based on these estimates.

While not strictly a MapReduce system, Scarlett [2] implements this preprocessing in the distributed file system itself. Scarlett uses predicted file popularity metrics to dynamically adjust the replication factors of popular files. When a MapReduce job eventually runs on the data, its partitioning will not only be even, but each partition will have greater data locality than the same job running without Scarlett. Put another way, the loss of data locality experienced by a system that does *not* run Scarlett effectively causes skew by requiring that some tasks fetch data over congested network links or from overloaded file servers.

In the context of parallel joins, the extended adaptive load balancing parallel hash join algorithm proposed by Hua and Lee [11] implements a form of preprocessing by having all nodes send locally computed bucket information to a coordinator node. The coordinator uses this global information to construct an even assignment of buckets to nodes, which eliminates skew.

It should be noted that if a parallel database can know ahead of time which attribute a relation will most likely be joined on, the parallel database can partition the relation according to this join attribute when the data is loaded into the database. This partitioning occurs well in advance of any join computations and allows the query planner to simply use local relation partitions as join partitions. Since the relation is already partitioned on the join attribute, the partitions are guaranteed to be evenly spread across the cluster thereby reducing skew.

The final "best practice" given by Kwon et al. is to rewrite the application to be independent of the data distribution. This advice is relevant in the cases of non-homomorphic `map` functions or holistic `reduce` functions. If an application has either of these functions, it will be incredibly difficult to eliminate skew. Unfortunately, such applications are usually quite complicated,

and designing a skew-resistant variant places an enormous burden on the user. SkewReduce [17] and SkewTune [19] get part of the way there by attempting to tackle these problems from within the framework. The heart of the problem however is in the application implementation, so it may not be feasible to completely eliminate skew.

4. THEMISMR

We have developed a MapReduce implementation on the *Themis* framework called *ThemisMR* [23]. Themis is a pipelined parallel data processing system designed with hardware and software balance in mind. Themis allows users to write applications that maximize per-machine throughput, thereby increasing cluster efficiency. We implemented an I/O efficient sorting system, TritonSort [24], on top of Themis. TritonSort competed in the SortBenchmark [28] competition in 2010 and 2011 and achieved a more than 83-fold increase in per-node efficiency over the previous record holding Hadoop system.

Briefly, the ThemisMR MapReduce implementation operates by reading input files from local disk and applying a `map` function to the data in-memory in the first phase. Unlike traditional MapReduce systems that materialize `map` output to disk, ThemisMR immediately streams `map` output over the network to its corresponding `reduce` partition, called a *logical disk*. Records bound for logical disks are written immediately to local disk on the receiving node. In the second phase, logical disks are read into memory, sorted, passed through a `reduce` function, and written back to local disk. This implementation reads and writes each record exactly twice, which is the theoretical minimum number of I/Os when the amount of data greatly exceeds the amount of available memory [1].

Under some workloads and data distributions, ThemisMR suffers performance penalties from skew. A particularly bad type of skew that occurs is reduce partitioning skew, which is related to RS. If the map output data is skewed and the correct partitioning function is not used, some logical disks will be much larger than others. In this case, ThemisMR loses intra-node parallelism since each logical disk must reside in memory in the second phase, so fewer logical disks can be reduced concurrently on a given node. In the worst case, a logical disk might not fit in memory at all, in which case some external reduce operation is required.

ThemisMR handles partitioning skew by using a custom partitioning function derived from a sample of the input data. ThemisMR runs a fast preprocessing phase, called *Phase Zero*, which applies the `map` function to a small sample of the input data. The `map` output is sorted and then range partitioned to create even `reduce` partitions. This range partitioning is then used in the first phase of the actual job.

The way ThemisMR handles skew is quite similar to existing algorithms surveyed so far. For example, DeWitt et al. [8] propose range partitioning derived from a small sample of the data for evenly partitioning data in hash-joins. Additionally, ThemisMR's use of many small logical disks that can be processed in parallel is related to the concept of virtual processor partitioning discussed in the same paper. ThemisMR essentially uses one virtual processor per output disk in order to keep all disks active. Finally, the Phase Zero preprocessing step employed by ThemisMR is directly in line with the advice of Kwon et al. [18], who suggest the use of a preprocessing step to ensure even partitioning.

In terms of the types of skew presented by Kwon et al. [18], ThemisMR successfully handles the data skew problem caused by heterogeneous maps and the reduce-side partitioning skew problem. Note that the other types of skew are primarily computational skew. Themis currently has no support for computational skew, although Phase Zero could be extended to handle this type of skew. More concretely, if Phase Zero were to take into account the execution time of `map`, it might be possible to repartition input files before starting the actual job. In this case, all `map` input partitions would take approximately the same amount of time to process. Phase Zero currently does not run the `reduce` function, so it would require substantially more effort to handle reduce-side computational skew.

5. CONCLUSION

Large-scale data processing systems tend to exhibit skew on many real world workloads. In this paper, we have discussed two types of systems, parallel databases and MapReduce architectures, and how they handle the problem of skew that arises in practice. The research community has made significant progress in reducing skew and improving the performance of such systems.

We have looked at several implementations of parallel joins in parallel databases. In particular, we have discussed both intrinsic skew and partitioning skew and how these parallel join algorithms handle each case. We have surveyed various types of joins, from hash-joins to sort-merge joins and have compared the skew-resistivity of each type.

Next we discussed how skew can arise in MapReduce applications. Many solutions exist that cover the spectrum of skew, from hardware-related skew to data skew. Some solutions apply to the framework itself, while others modify the underlying distributed storage system.

Lastly, we discussed the author's own work on ThemisMR, an I/O efficient MapReduce implementation. ThemisMR encounters partitioning skew on several popular applications and uses a sample-driven preprocessing step to mitigate the effects of such skew.

The works surveyed in this paper span several decades of research. It is interesting to note that the MapReduce community is essentially rediscovering many of the problems faced by the database community. However, the nature of the problems is slightly different due to subtle differences between the two types of systems. These differences are expressed in the similar, but subtly different solutions, that each community has developed.

Acknowledgments

I would like to thank Stefan Savage and my chair, Alin Deutsch, for their presence on my committee. I would also like to extend a special thanks to George Porter, who, in addition to serving on my committee, mentored me through the process of completing this exam. Finally I would like to thank my research adviser, Amin Vahdat, for guiding me over the years.

6. REFERENCES

[1] A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, Sept. 1988.

[2] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *EuroSys*, 2011.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, 2010.

[4] J. Dean. Designs, Lessons and Advice from Building Large Distributed Systems. LADIS keynote, 2009. `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`.

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

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

[7] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An Evaluation of Non-Equijoin Algorithms. In *VLDB*, 1991.

[8] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, 1992.

[9] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce : The Pig Experience. In *VLDB*, 2009.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.

[11] K. A. Hua and C. Lee. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. In *VLDB*, 1991.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.

[14] M. Kitsuregawa, M. Nakayama, and M. Takagi. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *VLDB*, 1989.

[15] M. Kitsuregawa and Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). In *VLDB*, 1990.

[16] R. N. Kline and M. D. Soo. The TIMEIT Temporal Database Testbed, 1998. `www.cs.auc.dk/TimeCenter/software.htm`.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *SoCC*, 2010.

[18] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A Study of Skew in MapReduce Applications. The 5th Open Cirrus Summit, 2011.

[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In *SIGMOD*, 2012.

[20] W. Li, D. Gao, and R. T. Snodgrass. Skew Handling Techniques in Sort-Merge Join. In *SIGMOD*, 2002.

[21] J. Lin. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce. In *LSDS-IR*, 2009.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report SIDL-WP-1999-0120. Stanford InfoLab, 1999.

[23] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat. ThemisMR: An I/O Efficient MapReduce. UCSD Tech Report #CS2012-0983, 2012.

[24] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI*, 2011.

[25] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–9, 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] A. D. Smith and W. Chung. The RMAP software for short-read mapping. http://rulai.cshl.edu/rmap/.
[28] Sort Benchmark. `http://sortbenchmark.org/`.
[29] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*, 2010.
[30] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB*, 1991.
[31] M. Wolfe. More Iteration Space Tiling. In *Supercomputing*, 1989.
[32] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel DBMS. In *VLDB*, 2009.
[33] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling Data Skew in Parallel Joins in Shared-Nothing Systems. In *SIGMOD*, 2008.
[34] Apache Hadoop. `http://hadoop.apache.org/`.
[35] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.
[36] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.