SkewTune: Mitigating Skew in MapReduce Applications

YongChul Kwon[1], Magdalena Balazinska[1], Bill Howe[1], Jerome Rolia[2]

[1] University of Washington, [2] HP Labs

{yongchul,magda,billhowe}@cs.washington.edu, jerry.rolia@hp.com

ABSTRACT

We present an automatic skew mitigation approach for user-defined MapReduce programs and present SkewTune, a system that implements this approach as a drop-in replacement for an existing MapReduce implementation. There are three key challenges: (a) require no extra input from the user yet work for all MapReduce applications, (b) be completely transparent, and (c) impose minimal overhead if there is no skew. The SkewTune approach addresses these challenges and works as follows: When a node in the cluster becomes idle, SkewTune identifies the task with the greatest expected remaining processing time. The unprocessed input data of this straggling task is then proactively repartitioned in a way that fully utilizes the nodes in the cluster and preserves the ordering of the input data so that the original output can be reconstructed by concatenation. We implement SkewTune as an extension to Hadoop and evaluate its effectiveness using several real applications. The results show that SkewTune can significantly reduce job runtime in the presence of skew and adds little to no overhead in the absence of skew.

1. INTRODUCTION

Today, companies, researchers, and governments accumulate increasingly large amounts of data that they process using advanced analytics. We observe that the increased demand for complex analytics support has translated into an increased demand for user-defined operations (UDOs) — relational algebra and its close derivatives are not enough [23, 33]. But UDOs complicate the algebraic reasoning and other simplifying assumptions relied on by the database community to optimize execution. Instead developers rely on "tricks" to achieve high performance: ordering properties of intermediate results, custom partitioning functions, extensions to support pipelining [34] and iteration [5], and assumptions about the number of partitions. For example, the Hadoop-based sort algorithm that won the tera-
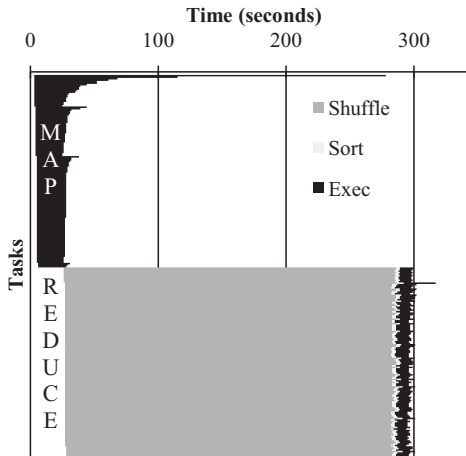
Figure 1: A timing chart of a MapReduce job running the PageRank algorithm from Cloud 9 [18]. Exec represents the actual map and reduce operations. The slowest map task (first one from the top) takes more than twice as long to complete as the second slowest map task, which is still five times slower than the average. If all tasks took approximately the same amount of time, the job would have completed in less than half the time.

sort benchmark in 2008 required a custom partition function to prescribe a global order on the data [27]. Moreover, when these UDOs are assembled into complex workflows, the overall correctness and performance of the application becomes sensitive to the characteristics of individual operations. Transparent optimization in the context of realistic UDO programming practices is a key goal in this work. In particular, we tackle the challenge of effective UDO parallelization.

MapReduce [6] has proven itself as a powerful and cost-effective approach for writing UDOs and applying them to massive-scale datasets [2]. MapReduce provides a simple API for writing UDOs: a user only needs to specify a serial map function and a serial reduce function. The implementation takes care of applying these functions in parallel to a large dataset in a shared-nothing cluster. In this paper, we therefore focus on UDOs in the form of MapReduce applications.

While MapReduce is a popular data processing tool [2], it still has several important limitations. In particular, skew is a significant challenge in many applications executed on this

platform [16, 21, 25]. When skew arises, some partitions of an operation take significantly longer to process their input data than others, slowing down the entire computation.

Figure 1 illustrates the problem. We use PageRank [4] as an example of a UDO. As the figure shows, this UDO is expressed as a MapReduce *job*, which runs in two main phases: the map phase and the reduce phase. In each phase, a subset of the input data is processed by distributed *tasks* in a cluster of computers. Each task corresponds to a partition of the UDO. When a map task completes, the reduce tasks are notified to pull newly available data. This transfer process is referred to as a shuffle. All map tasks must complete before the shuffle part of the reduce phase can complete, allowing the reduce phase to begin. Load imbalance can occur either during the map or reduce phases. We refer to such an imbalanced situation as *map-skew* and *reduce-skew* respectively. Skew can lead to significantly longer job execution times and significantly lower cluster throughput. In the figure, each line represents one task. Time increases from left to right. This job exhibits map-skew: a few map tasks take 5 to 10 times as long to complete as the average, causing the job to take twice as long as an execution without outliers.

There are several reasons why skew can occur in a UDO [16, 21, 25]. In this paper, we consider two very common types of skew: (1) skew caused by an uneven distribution of input data to operator partitions (or tasks) and (2) skew caused by some portions of the input data taking longer to process than others. For these sources of skew, speculative execution, a popular strategy in MapReduce-like systems [6, 13, 17] to mitigate skew stemming from a non-uniform performance of physical machines, is ineffective because the speculative tasks execute the same code on the same data and therefore do not complete in any less time than the original tasks.

Skew is a well-known problem that has been extensively studied in the context of parallel database management systems and adaptive or stream processing systems (See Section 6). One solution for handling skew involves the implementation of special skew-resistant operators. While this approach has successfully been applied to user-defined operators [30], it imposes an extra burden on the operator writer and only applies to operations that satisfy certain properties. An alternate common strategy involves dividing work into extremely fine-grained partitions and re-allocating these partitions to machines as needed [31]. Such a strategy is transparent to the operator writer, but it imposes significant overhead due to either state migration [31] or extra task scheduling [21]. A final strategy consists in materializing the output of an operator completely, sample that output, and plan how to re-partition it before executing the next operator. Such a strategy can yield efficient operator execution, but requires a synchronization barrier between operators, preventing pipelining and online query processing [14]. While MapReduce has limited pipelining today, significant efforts are underway to remove this constraint [34, 35].

In this paper, we propose SkewTune, a new technique for handling skew in parallel user-defined operations (UDOs). SkewTune is designed for MapReduce-type engines, characterized by disk-based processing and a record-oriented data model. We implemented the SkewTune technique by extending the Hadoop parallel data processing system [13]. SkewTune relies on two properties of the MapReduce model: (1) MapReduce's ability to buffer the output of an operator before transmitting it to the next operator; and (2) operator de-coupling, where each operator processes data as fast as possible without back-pressure from downstream operators. SkewTune's optimizations mitigate skew while preserving the fault-tolerance and scalability of vanilla MapReduce. The key features of SkewTune are:

- SkewTune mitigates two very common types of skew: Skew due to an uneven distribution of data to operator partitions and skew due to some subsets of the data taking longer to process than others.
- SkewTune can optimize unmodified MapReduce programs; programmers need not change a single line of code.
- SkewTune preserves interoperability with other UDOs. It guarantees that the output of an operator consists of the same number of partitions with data sorted in the same order within each partition as an execution without SkewTune.
- SkewTune is compatible with pipelining optimizations proposed in the literature (*c.f.*, [35]); it does not require any synchronization barrier between consecutive operators[1].

We evaluate SkewTune through experiments with real data and real applications including PageRank [4], Cloud-Burst [30], and an application that builds an inverted index over Wikipedia. We show that SkewTune can reduce processing times by up to factor of 4 when skew arises and adds only minimal overhead in the absence of skew. Most importantly, SkewTune delivers *consistent* performance independent of the initial configuration of a MapReduce job.

The rest of this paper is organized as follows. We discuss the problem description in more detail in Section 2. We present the SkewTune approach in Section 3 and the Hadoop implementation in Section 4. We show results from experiments with real application in Section 5. We finally discuss related work in Section 6.

2. PROBLEM DESCRIPTION

In this section, we review the MapReduce programming model and discuss the types of skew that can arise in this environment and that SkewTune is designed to mitigate.

### 2.1 MapReduce Programming Model

The MapReduce programming model calls for two functions *map* and *reduce* with the following types.

$$map :: (K1, V1) \rightarrow [(K2, V2)]$$
$$reduce :: (K2, [V2]) \rightarrow [(K3, V3)]$$

The *map* function takes a key and value of arbitrary types $K1$ and $V1$, and returns a sequence of (key, value) pairs of possibly different types, $K2$ and $V2$. All values associated with the same key $K2$ are grouped into a sequence and passed to the reduce function, which emits arbitrary key-value pairs of a final type $K3$ and $V3$.

Optionally, in the Hadoop implementation of MapReduce, users can also specify a custom partition function that redistributes the output of map tasks to reduce tasks.

---

[1]However, SkewTune, like MapReduce, does not allow downstream operators to throttle the flow of upstream operators, as is typically the case in parallel pipelined query plans.

(a) Map: Heterogeneous   (b) Reduce: Expensive Key
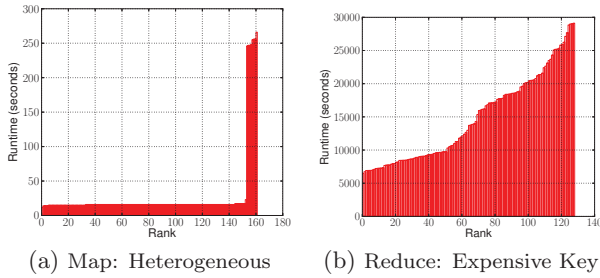
Figure 2: **(a) Distribution of map task runtime for CloudBurst with 162 mappers. The bimodal distribution corresponds to the two different types of input datasets being processed. (b) Distribution of reduce task runtime for CloudBurst with 128 reducers. The reduce is computationally expensive and has a smooth runtime distribution, but there is a factor of five difference in runtime between the fastest and the slowest reduce tasks.**

In this paper, we focus on the common class of MapReduce applications that consist of pure map and reduce functions, which operate on individual input keys without keeping any state between consecutive keys.

2.2 Types of Skew

In previous work, we analyzed the types of skew that arise in a variety of existing MapReduce applications [22]. Here, we briefly review four common types of skew that SkewTune is designed to address.

Map phase: Expensive Record. Map tasks process a collection of records in the form of key-value pairs, one-by-one. Ideally, the processing time does not vary significantly from record to record. However, depending on the application, some records may require more CPU and memory to process than others. These expensive records may simply be larger than other records, or the map algorithm's runtime may depend on the record value. PageRank [4] is an application that can experience this type of skew (Figure 1). PageRank is a link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively aggregating the weights of its inbound neighbors. Vertexes with a large outdegree take disproportionately longer to process because the map generates an output tuple per outgoing edge.

Map phase: Heterogeneous Map. MapReduce is a unary operator, but can be used to emulate an n-ary operation by logically concatenating multiple datasets as a single input. Each dataset may require different processing, leading to a multi-modal distribution of task runtimes. Figure 2(a) illustrates an example using the Cloudburst application [30]. CloudBurst is a MapReduce implementation of the RMAP algorithm for short-read gene alignment[2], which aligns a set of genome sequence reads against a reference sequence. CloudBurst distributes the approximate alignment computation across reduce tasks by partitioning n-grams of both sequences and reads. As a skew-mitigation strategy, the sequences bearing frequent n-grams are replicated across reduce tasks, while other sequences are hash-partitioned. These two algorithms exhibit different runtimes.

Reduce phase: Partitioning skew. In MapReduce, the outputs of map tasks are distributed among reduce tasks via hash partitioning (by default) or some user-defined par-

titioning logic. The default hash partitioning is usually adequate to evenly distribute the data. However, hash partitioning does not guarantee an even distribution. For example, in the inverted index building application, if the hash function partitions the data based on the first letter of a word, reducers processing more popular letters are assigned a disproportional amount of data.

Reduce phase: Expensive Key Group. In MapReduce, reduce tasks process a sequence of (key, set of values) pairs, called key groups. As in the case of expensive records processed by map, expensive key groups can skew the runtime of reduce tasks. Figure 2(b) illustrates an example.

2.3 SkewTune Design Requirements

Before presenting the SkewTune approach, we first discuss the rationale behind its design. When designing SkewTune, we had the following goals in mind:

Developer Transparency. The first goal behind SkewTune is to make it easier for MapReduce developers to achieve high performance. For this reason, we do not want these developers to even be aware that skew problems can arise. We want SkewTune to simply be an improved version of Hadoop that executes their jobs faster. As a result, we reject all design alternatives that require operator writers to either implement their jobs following special templates [3] or provide special inputs such as cost functions for their operators [21]. Instead, SkewTune should operate on unchanged MapReduce jobs.

Mitigation Transparency. Today, MapReduce makes certain guarantees to users: The output of a MapReduce job is a series of files, with one file per reducer. The user can configure the number of reducers. Additionally, the input of each reducer is sorted on the reduce key by the user-provided comparator function thus the output is produced in a specific order. To facilitate adoption and to ensure the correctness and efficiency of the overall application, we want SkewTune to preserve these guarantees. The output of a job executed with SkewTune should be the same as the output of a job executed without SkewTune: it should include the same number of files with the same data order inside these files. Indeed, users often create data analysis workflows and the application consuming the output of a MapReduce job may rely on there being a specific number of files and on the data being sorted within these files. By preserving these properties, SkewTune also helps ensure predictability: the same job executed on the same input data will produce the same output files in the same order.

Maximal Applicability. In MapReduce (and in other parallel data processing systems), many factors can cause skew in a UDO. Section 2 presented an overview of several such factors. We designed SkewTune to handle these different types of skew rather than specializing SkewTune for only one type of skew [6, 16]. In general, SkewTune strives to make the least number of assumptions about the cause of skew. Instead, it monitors execution, notices when some tasks run slower than others, and reacts accordingly independent of the reason why the tasks are slower.

No Synchronization Barriers. Finally, parallel data processing systems try to minimize global synchronization barriers to ensure high performance [20] and produce incremental results when possible. Even in MapReduce, reducers are allowed to start copying data before the previous mappers finish execution. Additionally, new MapReduce extensions

---

[2]`http://rulai.cshl.edu/rmap/`

strive to further facilitate pipelining during execution [24, 34, 35]. For those reasons, we avoided any design options that required blocking while an operator finishes processing before letting the next operator begin shuffling (and possibly processing) the data.

To achieve the above goals, SkewTune only assumes that a MapReduce job follows the API contract: each `map()` and `reduce()` invocation is independent. This assumption enables SkewTune to automate skew mitigation because it can be sure that re-partitioning input data at the boundary of map and reduce function invocations is safe. Such re-partitioning will not break the application logic.

3. SKEWTUNE APPROACH

SkewTune is designed to be API-compatible with Hadoop, providing the same parallel job execution environment while adding capabilities for detecting and mitigating skew. This section presents SkewTune's approach and core algorithms; Section 4 describes the implementation on top of Hadoop.

3.1 Overview

SkewTune takes a Hadoop job as input. For the purpose of skew mitigation, SkewTune considers the map and reduce phases of the job as separate UDOs. In SkewTune, as in Hadoop, a UDO pulls its input from the output of the previous UDO, where it is buffered locally. A UDO is assumed to take a *record* as input. A key-value pair (*i.e.*, mapper input) and a key group (*i.e.*, reducer input) are each considered a special case of a record. Each UDO is parallelized into tasks, and each task is assigned a *slot* in the cluster. There is typically one slot per CPU core per node. When a task completes, the slot becomes available.

SkewTune's skew mitigation technique is designed for MapReduce-type data processing engines. The three important characteristics of these engines with respect to skew handling are the following: (1) A coordinator-worker architecture where the coordinator node makes scheduling decisions and worker nodes run their assigned tasks. On completion of a task, the worker node requests a new task from the coordinator. This architecture is commonly used today [6, 8, 13, 17]. (2) De-coupled execution: Operators do not impose back-pressure on upstream operators. Instead, they execute independently of each other. (3) Independent record processing: The tasks are executing a UDO that processes each input record (possibly nested) independently of each other. Additionally, SkewTune requires (4) Per-task progress estimation, t_{remain}, which estimates the time remaining [26, 38] for each task. Each worker periodically reports this estimate to the coordinator. (5) Per-task statistics: each task keeps track of a few basic statistics such as the total number of (un)processed bytes and records.

Figure 3 illustrates the conceptual skew mitigation strategy of SkewTune. Without SkewTune, the operator completion time is dominated by the slowest task (*e.g.*, T2 in Figure 3(a)). With SkewTune, as shown in Figure 3(b), the system detects that T2 is experiencing skew at t_1 when T1 completes. SkewTune labels T2 as *the straggler* and mitigates the skew by repartitioning T2's *remaining* unprocessed input data. Indeed, T2 is not killed but rather terminates early as if all the data that it already processed was the only data it was allocated to process. Instead of repartitioning T2's remaining input data across only slots 1 and 2, Skew-
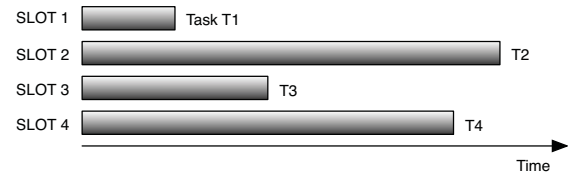
(a) Without SkewTune, operator runtime is that of the slowest task.
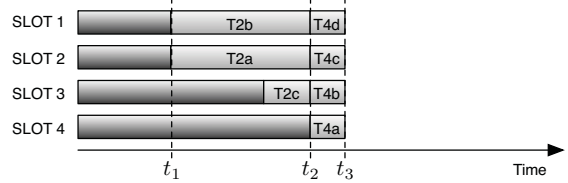
(b) With SkewTune, the system detects available resources as task T1 completes at t_1. SkewTune identifies task T2 as the straggler and re-partitions its unprocessed input data. SkewTune repeats the process until all tasks complete.

Figure 3: Conceptual skew mitigation in SkewTune

Table 1: Notations in Section 3.2,3.3

$\mathcal{N}$	Set of nodes in the cluster
$\mathcal{S}$	Set of slots in the cluster (multiple slots per node)
$\mathcal{O}$	Set of output files
$\mathcal{R}$	Set of running tasks
$\mathcal{W}$	Set of unscheduled tasks
Δ	Straggler's unprocessed data (bytes)
β	Disk bandwidth (bytes/seconds)
ρ	Task scheduling overhead (seconds)
ω	Repartitioning overhead (seconds)
T, t_{remain}	A task and its time-remaining (seconds)

Tune *proactively* repartitions the data to also exploit slot 3, which is expected to become available when T3 completes. SkewTune re-partitions the data such that all new partitions complete at the same time. The resulting subtasks T2a, T2b, and T2c are called *mitigators* and are scheduled in the longest processing-time first manner. SkewTune repeats the detection-mitigation cycle until all tasks complete. In particular, at time t_2, SkewTune identifies T4 as the next straggler and mitigates the skew by repartitioning T4's remaining input data.

In terms of our requirements from Section 2.3, SkewTune achieves *developer transparency* by detecting and mitigating skew at runtime without requiring any input from the developers. We further discuss SkewTune's skew detection approach in Section 3.2. To achieve *mitigation transparency*, SkewTune re-partitions the straggler's data using range-partitioning as we discuss further in Section 3.3. To be *maximally applicable*, SkewTune makes no assumptions about the cause of the skew. It also respects the input record boundary when repartitioning data. Thus, as long as a UDO follows the MapReduce API contract, SkewTune is applicable without breaking the application semantics. Finally, SkewTune's skew mitigation approach does not require any synchronization barriers.

3.2 Skew Detection

Skew detection determines *when* to mitigate skew experienced by *which* task. If the detection is too eager, SkewTune may split a task and pay unnecessary overhead (*i.e.*, false-positive). If the detection is too conservative, SkewTune

may miss the right mitigation timing thus diminishing the skew-mitigation gains (*i.e.*, false-negative).

Late Skew Detection: SkewTune's skew detection approach relies on the fact that tasks in consecutive phases are decoupled from each other. That is, map tasks can process their input and produce their output as fast as possible. They never block waiting for reduce tasks to consume that data. Similarly, reduce tasks can never be blocked by map tasks in a subsequent job.

This decoupling has important implications for skew handling. Because tasks can independently process their input as fast as possible, the cluster has high utilization as long as each slot is running some task. For this reason, SkewTune delays any skew mitigation decisions until a slot becomes available. We call this approach *late skew detection*. Late skew detection is analogous to MapReduce's current speculative execution mechanism [6, 13], where slow remaining tasks are replicated when slots become available. Similarly, SkewTune's repartitioning overhead is only incurred when there are idle resources. Late skew detection thus reduces opportunities for false positives. At the same time, it avoids false negatives by immediately allocating resources when they become available.

Identifying Stragglers: The next key question is to decide which task to label as the straggler. Here, we observe that it is never beneficial to re-partition more than one task at a time, since re-partitioning one task can suffice to fully occupy the cluster again. Given that only one task should be labeled as a straggler, SkewTune selects the task with the greatest t_{remain} estimate at the time of detection.

SkewTune flags skew when half of the time remaining is greater than the repartitioning overhead:

$$\frac{t_{remain}}{2} > \omega$$

The intuition is as follows. If SkewTune decides to repartition task T, at least two slots become available: the slot running T and the slot that recently became idle and triggered skew detection. After paying repartition overhead ω, the expected remaining time would be half of the remaining time of T (Table 1 summarizes the notation). The repartition thus only makes sense if the original runtime of T is greater than the new runtime plus the overhead. In our prototype implementation, ω is on the order of 30 seconds (see Section 5). Hence, our prototype only re-partitions tasks if at least 1 minute worth of processing remains. For long-running tasks where skew is particularly damaging, overhead of a few minutes is typically negligible.

Algorithm 1 summarizes SkewTune's skew detection strategy. As long as there exist unscheduled tasks, SkewTune invokes the ordinary task scheduler `chooseNextTask()`. If the coordinator runs out of tasks to schedule, SkewTune starts to consider repartitioning one of the running tasks based on the t_{remain} estimates. `stopAndMitigate()` asynchronously notifies the chosen task to stop and to commit the output produced so far. We describe the mitigation process next.

3.3 Skew Mitigation

There are three challenges related to mitigating skew through repartitioning. First, we want to minimize the number of times that we repartition any task to reduce repartitioning overhead. Second, when we repartition a straggler, we want to minimize any visible side-effects of the reparti-

---

Algorithm 1 GetNextTask()

Input: $\mathcal{R}$: set of running tasks
    $\mathcal{W}$: set of unscheduled waiting tasks
    inProgress: global flag indicating mitigation in progress
Output: a task to schedule
1: $task \leftarrow \textbf{null}$
2: **if** $\mathcal{W} \neq \emptyset$ **then**
3:    $task \leftarrow \texttt{chooseNextTask}(\mathcal{W})$
4: **else if** $\neg inProgress$ **then**
5:    $task \leftarrow \text{argmax}_{task \in \mathcal{R}} \text{ time\_remain}(task)$
6:    **if** $task \neq \textbf{null} \wedge \text{time\_remain}(task) > 2 \cdot \omega$ **then**
7:      $\texttt{stopAndMitigate}(task)$ /* asynchronous */
8:      $task \leftarrow \textbf{null}$
9:      $inProgress \leftarrow \textbf{true}$
10:   **end if**
11: **end if**
12: **return** $task$

---

tioning to achieve mitigation transparency (see Section 2.3). Finally, we want to minimize the total overhead of skew mitigation, including any unnecessary recomputations.

SkewTune strives to minimize the number of repartition operations by identifying one straggler at a time and proactively partitioning its data in a manner that accounts for slots that are likely to become available in the near future. To eliminate side-effects of skew mitigation, SkewTune uses range partitioning to ensure that the original output order of the UDO result is preserved. To minimize the mitigation overhead, SkewTune saves a straggler's output and repartitions only its unprocessed input data. It also uses an inexpensive, linear-time heuristic algorithm to plan mitigators. To drive this planning, SkewTune needs to collect information about the value distribution in the repartitioned data. To minimize overhead, SkewTune makes a cost-based decision to scan the remaining data locally at the straggler or to spawn new tasks that scan the distributed input in parallel.

Skew mitigation occurs in three steps. First, the straggler stops its computation. Second, depending on the size of the data that remains to be processed, either the straggler or the operators upstream from the straggler collect statistics about the straggler's remaining input data. Finally, the coordinator plans how to re-partition the straggler's remaining work and schedules the mitigators. We now present these steps in more detail.

3.3.1 Stopping a Straggler

When the coordinator asks a straggler to stop, the straggler captures the position of its last processed input record, allowing mitigators to skip previously processed input. If the straggler is in a state that is impossible or difficult to stop (*e.g.*, processing the last input record or performing the local sort at the end of the map phase), the request fails and the coordinator either selects another straggler or repartitions and reprocesses the entire straggler's input if this straggler is the last task in the job. Reprocessing a straggler's entire input is analogous to MapReduce's speculative execution [6, 13] except that SkewTune repartitions the input before reprocessing it.

3.3.2 Scanning Remaining Input Data

In order to ensure skew mitigation transparency, SkewTune uses range-partitioning to allocate work to mitigators. With this approach, the data order remains unchanged between the original MapReduce job and the altered job.

The output of the mitigators only needs to be concatenated to produce an output identical to the one obtained without SkewTune. An alternate design would be to use hash-partitioning and add an extra MapReduce job to sort-merge the output of the mitigators. Such an extra job would add overhead. Additionally, a hash function is not guaranteed to evenly balance load between mitigators, especially if the number of keys happens to be small. Range partitioning avoids both problems.

When range-partitioning data, a data range for a map task takes the form of an input file fragment (*i.e.*, file name, offset, and length). A range for a reduce task is an interval of reduce keys. In the rest of this section, we focus on the case of repartitioning the reduce task's input. The techniques are equally applicable to map tasks.

Range-partitioning a straggler's remaining input data requires information about the content of that data: The coordinator needs to know the key values that occur at various points in the data. SkewTune collects that information before planning the mitigator tasks.

A naïve approach is to scan the data and extract all keys together with the associated record sizes. The problem with this approach is that it may produce a large amount of data if there exists a large number of distinct keys. Such large data imposes a significant network overhead and also slows-down the mitigator planning step.

Instead, SkewTune collects a compressed summary of the input data. The summary takes the form of a series of key intervals. Each interval is approximately the same size in bytes, respecting the input boundaries (*e.g.*, a single record for map, values sharing a common reduce key for reduce). These intervals become the units of range-partitioning. Consecutive intervals can be merged to create the actual data range assigned to a mitigator.

Choosing the Interval Size: Given $|\mathcal{S}|$, the total number of slots in the cluster, and Δ, the number of unprocessed bytes, SkewTune needs to generate at least $|\mathcal{S}|$ intervals since it is possible that all cluster slots will be available for mitigators. However, because SkewTune may want to allocate an uneven amount of work to the different mitigators (*e.g.*, Figure 3), SkewTune generates $k|\mathcal{S}|$ intervals. Larger values of k enable finer-grained data allocation to mitigators but they also increase overhead by increasing the number of intervals and thus the size of the data summary. In our prototype implementation, k is set to 10. Hence, the size s of the intervals is given by $s = \lfloor \frac{\Delta}{k \cdot |\mathcal{S}|} \rfloor$.

Local Scan: If the size of the remaining straggler data is small, the worker running the straggler scans that data and generates the intervals. Algorithm 2 summarizes the interval generation process. The algorithm expects a stream of intervals I as input. This is the stream of *singleton intervals*, with one interval per key in the reducer's input. For the local scan, b is set to s and k is ignored. The algorithm iterates over these singleton intervals. To generate the output intervals, it opens an interval with the first seen key. It then merges the subsequent keys and their statistics (*e.g.*, size of all values in bytes) until the aggregated byte size reaches the threshold s. If a key has a byte size larger than s, the key remains in its own singleton interval. The process continues until the end of the data.

Choosing between a Local and a Parallel Scan: To choose between a local and a parallel scan, SkewTune com-

---

Algorithm 2 GenerateIntervals()

Input: I: Sorted stream of intervals
    b: Initial bytes-per-interval. Set to s for local scan.
    s: Target bytes-per-interval.
    k: Minimum number of intervals.
Output: list of intervals
1: $result \leftarrow []$ /* resulting intervals */
2: $cur \leftarrow \texttt{new\_interval}()$ /* current interval */
3: **for all** $i \in I$ **do**
4:     **if** $i.bytes > b \vee cur.bytes \geq b$ **then**
5:         **if** $b < s$ **then**
6:             $result.\texttt{appendIfNotEmpty}(cur)$
7:             **if** $|result| \geq 2 \times k$ **then**
8:                 /* accumulated enough intervals. increase b. */
9:                 $b \leftarrow \min\{2 \times b, s\}$
10:                /* recursively recompute buffered intervals */
11:                $result \leftarrow \texttt{GenerateIntervals}(result, b, b, k)$
12:             **end if**
13:         **else**
14:             $result.\texttt{appendIfNotEmpty}(cur)$
15:         **end if**
16:         $cur \leftarrow i$ /* open a new interval */
17:     **else**
18:         $cur.\texttt{updateStat}(i)$ /* aggregate statistics */
19:         $cur.end \leftarrow i.end$
20:     **end if**
21: **end for**
22: $result.\texttt{appendIfNotEmpty}(cur)$
23: **return** $result$

---

pares the estimated cost (in terms of total time) for each approach. The time for the local scan is given by $\frac{\Delta}{\beta}$, where Δ is the remaining input data in bytes and β is the local disk bandwidth. The time for the parallel scan is the time to schedule an extra MapReduce job to perform the scan, and the time for that job to complete. The latter is equal to the time that the slowest task in the job, say n, will take to scan its input data: $\frac{\sum_{o \in O_n} o.bytes}{\beta}$, where O_n is the set of all map outputs at node n (recall that multiple map tasks can run on a node). The decision is thus made by testing the following inequality:

$$\frac{\Delta}{\beta} > \frac{\max\{\sum_{o \in O_n} o.bytes \mid n \in \mathcal{N}\}}{\beta} + \rho$$

where $\mathcal{N}$ is the set of nodes in the cluster and ρ is the task scheduling delay. The stopping straggler tests the inequality since it knows where its input data came from. If a parallel scan is expected to be more cost-effective, the straggler immediately replies to the coordinator and the latter schedules the parallel scan.

Parallel Scan: During a parallel scan, Algorithm 2 runs in parallel over the distributed input data (*i.e.*, map outputs). The intervals generated for each map output file are then put together to estimate the intervals that would have been generated by a local scan (illustrated in Figure 4).

The s value for the Local Scan may be too large for a parallel scan because there are usually more map outputs than the total number of slots in the cluster. Thus, we set a smaller s value for the parallel scan to properly generate intervals for each map output:

$$s = \lfloor \frac{\Delta}{k \cdot \max\{|\mathcal{S}|, |\mathcal{O}|\}} \rfloor$$

where $\mathcal{O}$ is the union of all the O_n sets. Additionally, because the size of the map output files can be skewed and

Interval ID	Begin key	# values	End key
i_1	$k_3 : 4$	9	$k_7 : 3$
i_2	$k_7 : 1$	10	$k_{100} : 2$
i_3	$k_{50} : 2$	14	$k_{95} : 5$

Input: Intervals from Parallel Local Scans.

$\Rightarrow$

Key Range	Est. # values	Intervals
$[k_3, k_3]$	4	i_1
(k_3, k_7)	9	i_1
$[k_7, k_7]$	4	i_1, i_2
(k_7, k_{50})	10/5	i_2
$[k_{50}, k_{50}]$	$2 + 10/5$	i_2, i_3
(k_{50}, k_{95})	$14 + 10/5$	i_2, i_3
$[k_{95}, k_{95}]$	$5 + 10/5$	i_2, i_3
(k_{95}, k_{100})	10/5	i_2
$[k_{100}, k_{100}]$	2	i_2

Merge intervals and estimate # of values. The # values of i_2 (10) is evenly distributed over (k_7, k_{100}) range.

$\Rightarrow$

Key Range	Est. # values
$[k_3, k_3]$	4
(k_3, k_7)	9
$[k_7, k_7]$	4
(k_7, k_{50})	2
$[k_{50}, k_{50}]$	4
(k_{50}, k_{95})	16
$[k_{95}, k_{95}]$	7
(k_{95}, k_{100})	2
$[k_{100}, k_{100}]$	2

Output: Aligned key ranges and estimated # of values.

Figure 4: Merging Result of Parallel Scan. The table on the left shows the output of the parallel scan. The middle column #*values* represents the number of values that fall between *begin* and *end* keys. Each key is also associated with its number of values (the number followed by ':'). The table on the right shows the output from merging the input intervals and the estimated number of values for each range. The values of wide interval (k_7, k_{100}) introduce uncertainty. The middle table shows that how the 10 values of i_2 are evenly redistributed across the five key ranges included in (k_7, k_{100}).

because SkewTune does not know how much data in each of these files will have to be re-processed, SkewTune dynamically adjusts the interval size (variable b in Algorithm 2) starting from a small value (*e.g.*, 4 KB in prototype) and adaptively increasing it as it sees more unprocessed data. Whenever the b value is doubled, the collected intervals so far are merged using the new b value (line 7-12). Once the b value becomes s, the algorithm reaches a steady state and produces intervals every s bytes. Without this approach, a single wide key-interval may be generated for small data files and such wide key-intervals yield errors during the interval merge process at the coordinator.

Merging Intervals from a Parallel Scan: The intervals generated by a parallel scan are put together to approximate the result of a local scan. We present a two-pass algorithm illustrated in Figure 4, which estimates the number of records that would have been output by a local scan. The algorithm can be extended to handle other measures (*e.g.*, the number of bytes and the number of keys) in a straightforward manner.

The leftmost table in Figure 4 shows the input of the algorithm, which is a list of possibly overlapping intervals. Each interval is represented with a triple (begin key, # values, end key). The # *values* field represents the number of values that fall between *begin* and *end* keys. Each key is also associated with its number of values. The input is sorted by the *begin* key of each interval.

Intervals are a lossy representation of the data distribution. As a simplification, our approach assumes that values are uniformly distributed in each interval. A small complication arises when intervals overlap. For example, i_3 is completely included in i_2 and it is uncertain how the 10 values of i_2 are actually distributed in the (k_7, k_{100}) range due to i_3. The middle table of Figure 4 shows addressing such uncertainty by evenly distributing the values over (k_7, k_{100}) range given the input. In the input, we can be sure that k_{50} and k_{95} exist between k_7 and k_{100}. The two keys segment the uncertain range (k_7, k_{100}) into five intermediate ranges: $(k_7, k_{50}), [k_{50}, k_{50}], (k_{50}, k_{95}), [k_{95}, k_{95}]$, and (k_{95}, k_{100}). Then we evenly distribute the 10 values of i_2 over the five ranges. For this approximation, thus we need to know how many keys fall within (or overlap) each of input interval.

The algorithm proceeds in two passes over the input data. During the first pass, the algorithm collects statistics about existing intervals and how they overlap. It then generates the smaller key ranges and estimates the number of values in these smaller ranges during the second pass. Figure 4 shows an example of input and output for this algorithm. We omit the algorithm pseudocode since it is straightforward.

The time and space complexities of the algorithm are $O(|I| \log |I|)$ and $O(|I|)$ respectively where I is the list of input intervals. The size of I is controlled by value k in Algorithm 2 during the parallel scan. If $|I|$ is expected to be too large to fit in memory, the k value needs to be adjusted to a smaller value. The output of the algorithm can also be post-processed by Algorithm 2 (*i.e.*, merge small adjacent key ranges to make the final intervals roughly the size of s).

3.3.3 Planning Mitigators

Finally, we present SkewTune's approach to planning mitigators. The goal is to find a contiguous order-preserving assignment of intervals to mitigators, meaning that the intervals assigned to a mitigator should be totally ordered on the key and should be contiguous: *i.e.*, no intervals between the first and the last keys should be assigned to other mitigators. The assignment should also minimize the completion time of all re-allocated data.

The planning algorithm should be fast because it is on the critical path of the mitigation process. A longer execution time means a longer idle time for the available slot in the cluster. We now describe a heuristic algorithm with linear time complexity with respect to the number of intervals.

Algorithm 3 takes as input the time remaining estimates for all active tasks in the cluster, the intervals collected by the data scan, a time remaining estimator θ, which serves to estimate processing times for intervals from their statistics (*e.g.*, sizes in bytes), and overhead parameters. The algorithm proceeds in two phases. The first phase (line 1-10) computes the optimal completion time *opt* assuming a perfect split of the remaining work (*i.e.*, record boundaries are not honored). The phase stops when a slot is assigned less than 2ω work to avoid generating arbitrarily small mitigators (line 6-7). 2ω is the largest amount of work such that

Algorithm 3 LinearGreedyPlan()

Input: I: a sorted array of intervals
   T: a sorted array of t_{remain} for all slots in the cluster
   θ: time remaining estimator
   ω: repartitioning overhead
   ρ: task scheduling overhead
Output: list of intervals
   /* Phase 1: find optimal completion time opt. */
1: $opt \leftarrow 0$; $n \leftarrow 0$ /* n: # of slots that yield optimal time */
2: $W \leftarrow \theta(R)$ /* remaining work+work running in n nodes */
3: /* use increasingly many slots to do the remaining work */
4: **while** $n < |T| \wedge opt \geq T[n]$ **do**
5:    $opt' \leftarrow \frac{W+T[n]+\rho}{n+1}$ /* optimal time using $n+1$ slots */
6:    **if** $opt' - T[n] < 2 \cdot \omega$ **then**
7:       **break** /* assigned too little work to the last slot */
8:    **end if**
9:    $opt \leftarrow opt'$; $W \leftarrow W + T[n] + \rho$; $n \leftarrow n + 1$
10: **end while**
   /* Phase 2: greedily assign intervals to the slots. */
11: $P \leftarrow []$ /* intervals assigned to slots */
12: $end \leftarrow 0$ /* index of interval to consider */
13: **while** $end < |I|$ **do**
14:    $begin \leftarrow end$; $remain \leftarrow opt - T[|P|] - \rho$
15:    **while** $remain > 0$ **do**
16:       $t_{est} \leftarrow \theta(I[end])$ /* estimated proc. time of interval */
17:       **if** $remain < 0.5 \cdot t_{est}$ **then**
18:          **break** /* assign to the next slot */
19:       **end if**
20:       $end \leftarrow end + 1$; $remain \leftarrow remain - t_{est}$
21:    **end while**
22:    **if** $begin = end$ **then**
23:       $end \leftarrow end + 1$ /* assign a single interval */
24:    **end if**
25:    $P.\texttt{append}(\texttt{new\_interval}(I[begin], I[end-1]))$
26: **end while**
27: **return** P

further repartitioning is not beneficial. In the second phase, the algorithm sequentially packs the intervals for the earliest available mitigator as close as possible to the opt value. The algorithm then repeats the process for the next available mitigator until it assigns all the intervals to mitigators. The time complexity of this algorithm is $O(|I| + |\mathcal{S}| \log |\mathcal{S}|)$ where $|I|$ is the number of intervals and $\mathcal{S}$ is the number of slots in the cluster.

3.4 Discussion

SkewTune in a Shared Cluster: SkewTune currently assumes that a single user has access to all the resources in a cluster. There are two ways to incorporate SkewTune in a shared cluster setup: (1) by using a task scheduler that carves out a pre-defined set of resources for each user or (2) by implementing a SkewTune-aware scheduler that prioritizes mitigators (and preempts other tasks if necessary) if mitigating a straggler improves overall cluster utilization and latency.

Very expensive map() or reduce(): SkewTune is designed to repartition load around record boundaries. SkewTune is not designed to mitigate skew in the case where single invocations of the user-defined map() or reduce() functions take an extremely long time. To handle such cases, SkewTune would need to be extended with techniques such as those in the SkewReduce [21] system.

4. SKEWTUNE FOR HADOOP
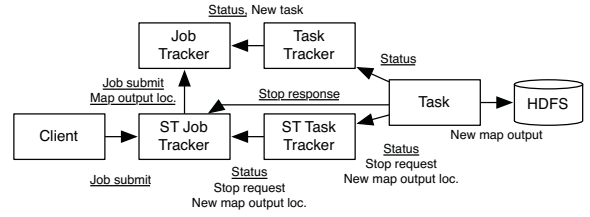
Figure 5: SkewTune Architecture. Each arrow is from sender to receiver. Messages related to mitigation are shown. Requests are underlined. Mitigator jobs are created and submitted to the job tracker by the SkewTune job tracker. Status is the progress report.

Overview: We implemented SkewTune on top of Hadoop 0.21.1. We modified core Hadoop classes related to (1) the child process, which runs the user supplied MapReduce application and (2) the Shuffle phase, which also runs in the child process. The only class we modified that runs in the Hadoop trackers is the JobInProgress class, which holds all information associated with a job. We added fields to track dependent jobs (*i.e.*, mitigator jobs) such that the map output is only cleaned up when there is no dependent job running.

The prototype consists of a job tracker and a task tracker analogous to those used in Hadoop. The child processes running with SkewTune report to both Hadoop and SkewTune trackers as shown in Figure 5. The SkewTune job tracker serves as the coordinator and is responsible for detecting and mitigating skew in the jobs submitted through its interface. The SkewTune task tracker serves as a middle tier that aggregates and delivers messages between the SkewTune job tracker and the Hadoop MapReduce tasks. When mitigating skew, the SkewTune job tracker executes a separate MapReduce job for each parallel data scan and for each mitigation.

Stopping a Straggler Task: When a straggler task has been chosen, SkewTune tries to stop it by flagging a field in the heartbeat response message from the SkewReduce job tracker to its task tracker. Upon receiving the stop request, the task immediately reports back the currently processed record (*e.g.*, current offset in the input file or the current reduce key). If only a small amount of data remains to be processed, the task also runs the local data scan and returns the summary intervals. The stopped task then completes processing the current record and terminates. If a task is in the map-side sort, shuffle, or reduce-side sort phases, stopping the task is difficult because the outputs from disk spills, different tasks are intermingled at any point of those stages and it is hard to precisely define what are the remaining work in those stages in a compact manner. Tasks re-partitioned during those phases reports there is nothing to repartition and the coordinator makes decision as described in Section 3.2.

The current prototype only supports file-based inputs. Also, the RecordReader must implement an interface, StoppableRecordReader, to support stopping the process. When stop is requested, the record reader must return the current position in the record stream, and the remaining bytes if possible. If the stop was successful, the record reader immediately returns an end of stream on the next record request so that the map can start the SORT phase.

Repartitioning a Map Task: When SkewTune de-

cides to repartition a map task, the map task runs the local scan (because map tasks are typically assigned with small amounts of data. It is possible to use the parallel scan if the size of remaining data is large and the input is replicated) and reports the summary intervals to the coordinator. The mitigators for a map task execute as map tasks within a new MapReduce job. They have the same map and, optionally combiner, functions.

We modify the original Map task implementation to sort and write the map output to HDFS when the task is a mitigator. Without this change, a map without reduce would skip the SORT phase. The map output index, *i.e.*, the information that reports which portion of the file is designated to which reduce task, is also written to HDFS for fault tolerance and sent to the SkewTune job tracker via a heartbeat message. The job tracker broadcasts the information about the mitigated map output to all reducers in the job.

Repartitioning a Reduce Task: To repartition a reduce task, the parallel scan job (if it exists) and the mitigator job read map outputs from the Hadoop task tracker[3]. Thus, we implemented `InputSplit`, `TaskTrackerInputFormat` and `MapOutputRecordReader` to directly fetch the map output from task trackers. Our implementation uses the HDFS API to read the mitigated map outputs. `MapOutputRecordReader` skips over the previously processed reduce keys to ensure that only unprocessed data is scanned and repartitioned. For both jobs, we create one map task per node, per storage type (*i.e.*, task tracker and HDFS) so that each map task reads local data if the schedule permits it.

The map task in the mitigator job runs an identity function since all the data has already been processed. The partition function is replaced with a range partitioner provided by the SkewTune framework. The bucket information generated by the planner is compressed and encoded in the job specification. If a combiner exists in the original job, the map task also runs the same combiner to reduce the amount of data. Since the map is running the identity function, SkewTune knows that it can use more memory for the combiner and sort. Thus, it adjusts the corresponding configuration values appropriately. The reduce task runs unchanged.

Merging Mitigated Output: A mitigation job creates its output directory under the original output directory so that the mitigated output can be merged or disposed with the original output. The name of directory consists of the name of original task output and a suffix that identifies the mitigated job. The output can be merged in two ways. First, the following MapReduce job can read the output using an extended `CombinedFileInputFormat` which logically concatenates the mitigated output files with `CombinedInputSplit`. Or, on completion of the job, concatenates the files using HDFS concatenate operation [4].

Reducing Launch Overhead: Launching a new MapReduce job is relatively expensive compared to launching a job in a database management system (DBMS) because every task and job starts from scratch [28]. This overhead is critical for SkewTune since eliminating such overheads enables SkewTune to be used with relatively short jobs. An optimization we made to reduce the startup overhead of mitigator jobs is that SkewTune does not copy the binaries and data per launch. Instead, SkewTune simply reuses the existing binaries and data copied to HDFS for the original job. This way, SkewTune can avoid the overhead of copying redundant files when launching a new parallel scan job as well as a new mitigator job.

Progress Monitoring and Estimation: The prototype implements Parallax to estimate time remaining [26]. We extend Parallax to handle multiple spills at the end of the Map phase. The extension is a simple analytical model of sort and spill as proposed in Li [24]. The estimate is calculated in the heartbeat thread of the child process and transmitted to the SkewTune task tracker with every report, roughly every 3 seconds. The SkewTune task tracker collects all reports from all local tasks, and submits them to SkewTune job tracker via a heartbeat message. Thus, there is an end-to-end delay from the map/reduce process to the job tracker that is double of the heartbeat interval.

Fault Tolerance: Fault-tolerance of SkewTune is mostly identical to that of Hadoop. The worst failure scenario in SkewTune is the same as in Hadoop: losing a map output due to a node failure. In this case, the lost map output has to be recomputed if there exists any reduce tasks that have not read it yet. Any map task failure or reduce task failure could be handled as if the failed tasks were experiencing skew. SkewTune may parallelize the re-execution but the current prototype does not implement this.

The failure of the coordinator could be handled similarly as in Hadoop except the coordinator has to persist repartitioning decisions so that the coordinator can make consistent decisions after recovery.

5. EVALUATION

We evaluate the benefits of SkewTune when skew arises, SkewTune's robustness to initial job configuration parameters, and SkewTune's overhead in the absence of skew. We find that SkewTune delivers up to a factor of 4X improvement on real datasets and real UDOs. It also significantly reduces runtime variability. Further, the overhead of SkewTune in the absence of skew is shown to be minimal.

All experiments are performed on a twenty-node cluster running Hadoop 0.21.1 with a separate master node. Each node uses two 2 GHz quad-core CPUs, 16 GB of RAM, and two 750 GB SATA disk drives. All nodes are used as both compute and storage nodes. The HDFS block size is set to 128 MB and each node is configured to run at most four map tasks and four reduce tasks concurrently.

We evaluate SkewTune using the following applications.

Inverted Index (II): An inverted index is a popular data structure used for Web search. We implemented a MapReduce job that builds an inverted index from the full English Wikipedia archive and generates a compressed bit vector for each word. The Potter word stemming algorithm is used to post-process the text during the map phase [5]. The RADIX partitioner is used to map letters of the alphabet to reducers and to produce a lexicographically ordered output. The total data size is 13 GB.

PageRank (PR): PageRank [4] is a popular link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively aggregating the weights of its in-

---

[3]Map output is served via HTTP by an embedded web server in the task tracker

[4]Implemented by HDFS-222 patch but requires that all intermediate blocks be full.

[5]We use a bit vector implementation and a stemming algorithm from the Apache Lucene open source search engine.

bound neighbors. We take the PageRank implementation from Cloud 9 [18] and apply it to the freebase dataset [11]. The total input data size is 2.1 GB.

CloudBurst (CB): CloudBurst [30] is a MapReduce implementation of the RMAP algorithm for short-read gene alignment[6]. CloudBurst aligns a set of genome sequence reads with a reference sequence. We take the CloudBurst application and use it to process a methylotroph dataset [19]. The total input data size is 1.1 GB.

5.1 Skew Mitigation Performance

The first question that we ask is how well SkewTune mitigates skew.

Figure 6(a) shows the runtime for the reduce phase of the Inverted Index application. When using vanilla Hadoop, the reduce phase runs across 27 reducers (one per letter of the alphabet and one for special characters) and completes in 1 hour and 52 minutes. With SkewTune, as soon as the reduce phase starts, SkewTune notices that resources are available (there are a total of 80 reduce slots). It thus partitions the 27 tasks across the available slots until the cluster becomes fully occupied. The runtime drops to only 25 minutes, a factor of 4.5 faster. This experiment demonstrates that, with SkewTune, a user can focus on the application logic when implementing her UDO. She does not need to worry about the cluster details (*e.g.*, how to write the application to use N reducers instead of the natural 27).

In the figure, we also show the ideal execution time for the job. This execution time is derived from the logs of the vanilla Hadoop execution: we compute the minimal runtime that could be achieved assuming zero overhead and a perfectly accurate cost model driving the load re-balancing decisions. In the figure, we see that SkewTune adds a significant overhead compared to this ideal execution time. The key reasons for the extra latency compared with ideal are scheduling overheads and an uneven load distribution due to inaccuracies in SkewTune's simple runtime estimator. SkewTune does, however, improve the total runtimes greatly compared with vanilla Hadoop. In the rest of this section, we always decompose the runtime into *ideal* time and *extra* time. The latter accounts for all real overheads of the system and possible resource under utilization.

Figure 6(b) shows the runtime for the map phase of CloudBurst. This application uses all map slots. Hence, the cluster starts off fully utilized. However, the mappers process two datasets: the sequence reads and the reference genome. All map tasks assigned to process the former complete in under a minute. With vanilla Hadoop, the job then waits for the mappers processing the reference dataset to complete. In contrast, SkewTune re-balances the load of the mappers processing the reference dataset, which improves the completion time from 12 minutes to 3 minutes (ideal time is 66 seconds). This application is a classical example of skew and it demonstrates SkewTune's ability to both detect and mitigate that skew. Notice that skew arises even though all mappers are initially assigned the same amount of data (in bytes).

Finally, we demonstrate SkewTune's ability to help users avoid the negative performance implications of mis-configuring their jobs. Figure 6(c) shows the runtime for the map phase of PageRank. The figure shows two configurations: a good configuration and a worst-case configura-

---

[6] http://rulai.cshl.edu/rmap/
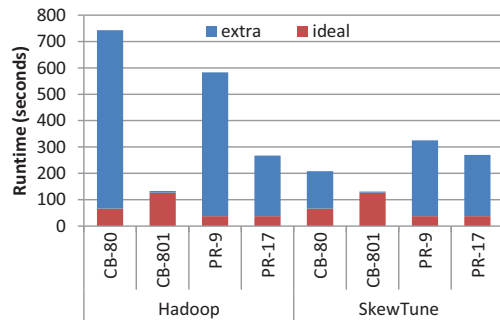
Figure 7: **Performance Consistency of Map Phase: For both PageRank (PR) and CloudBurst (CB), SkewTune delivers high-performance consistently, while Hadoop is sensitive to the initial configuration (here, the number of map tasks).**

tion. In the good case, vanilla Hadoop and SkewTune perform similarly. However, if the job is mis-configured, vanilla Hadoop leads to a significantly longer completion time while SkewTune maintains a consistent performance. To create the bad configuration, we simply changed the input data order: we sorted the nodes in the graph by increasing order of outdegree. While in practice a user may not necessarily hit the worst-case configuration for this application, the experiment shows that vanilla Hadoop is sensitive to user mis-configurations, unlucky data orders, and other unfortunate conditions. In contrast, SkewTune delivers high performance systematically, independent of these initial conditions.

5.2 Performance Consistency

In this section, we further study the consistency of the performance that SkewTune delivers. For this, we run the CloudBurst and PageRank applications but we vary the initial number of tasks. Figure 7 shows the results for the map phase of CloudBurst using either 80 or 801 mappers and PageRank using either 9 or 17 mappers. As the figure shows, Vanilla Hadoop is sensitive to these configuration parameters with up to a 7X difference in runtimes. In contrast, SkewTune's performance is significantly more stable with performance differences within 50%. The figure shows, however, that for configurations without skew in PageRank, SkewTune yields a runtime higher than that of vanilla Hadoop (3 s more). This is due to inaccurate time-remaining estimates: SkewTune missed the timing to mitigate skew of the longest map task and made an unnecessary split of another task. The overhead, however, is negligible.

5.3 Skew Mitigation Overhead

To measure some of SkewTune's overheads, we re-run the same applications as above, but we tune them to ensure low runtimes with vanilla Hadoop. We make the following tunings. For CloudBurst, we configure the number of map and reduce tasks exactly as the author recommends: We use 10 times as many map tasks and 2 times as many reduce tasks as slots. In the experiment, we thus get 801 map tasks (the last task is assigned only a small amount of data due to rounding in size) and 160 reduce tasks. For the Inverted Index, we use a hash partitioner and spread the reduce input across 140 tasks. Finally, for PageRank, we use 17 map and 17 reduce tasks with 128 MB chunks. This configuration
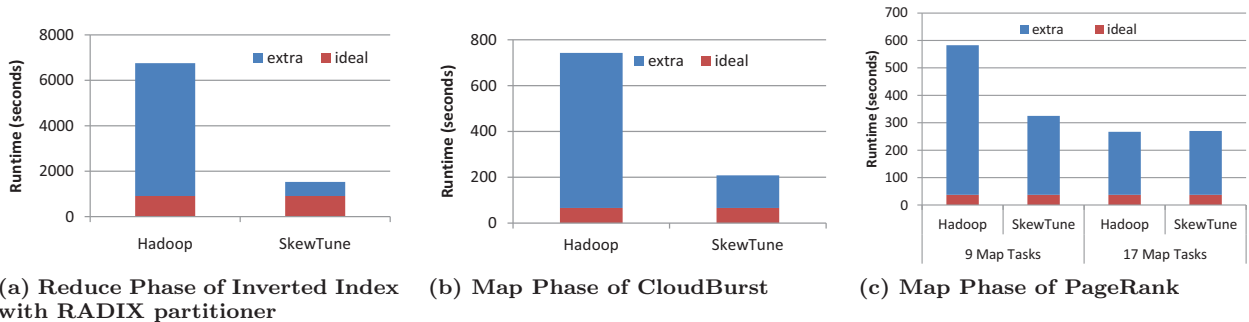
(a) Reduce Phase of Inverted Index with RADIX partitioner

(b) Map Phase of CloudBurst

(c) Map Phase of PageRank

Figure 6: UDO runtime with and without SkewTune.

Figure 8: Map Tasks without Skew

Figure 9: Reduce Tasks without Skew

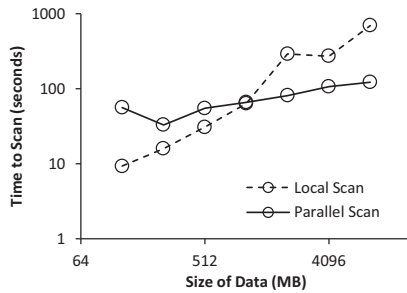| Type	Scan	Plan	< Compute	Input Bytes
Map	8.0s (3.0)	0.19s (0.08)	5.01s (3.83)	84MB (55)
Reduce	15s (15.0)	0.18s (0.19)	15.7s (10.4)	140MB (175)

Table 2: Mitigation Overhead Statistics. The average and standard deviation (number in parentheses) in seconds for each mitigation step. Size of re-partitioned data. "< Compute" represents time until the actual processing resumes. Scans are all local scans.

nique, where we replace SkewTune's range partitioning with hash partitioning thus avoiding the need to scan the remaining input data. Overall, REHASH performs slightly better than SkewTune due to its reduced overhead but it requires an extra job to recover the ordering (note that the numbers do not include such extra jobs!). SkewTune is only marginally slower than REHASH but it preserves the output order.

Detailed Mitigation Overhead Analysis: We further analyze the overhead of mitigating the skew of a single straggler by analyzing the execution logs of 32 map task mitigations and 64 reduce task mitigations from our three test applications. Overall, in these experiments, the current SkewTune prototype incurs approximately 15 sec overhead for map task skew mitigation and 30 sec for reduce tasks.

Table 2 shows the breakdown of the overhead. Interestingly, the mitigator planning phase takes less than 200 ms. It hardly incurs any overhead due to the compact summary information. We ran extra experiments (not shown due to space constraints), where we varied the interval granularity. We found the PLAN phase to be consistently fast and below 500 ms in all configurations. The most significant overhead component is the data scan, which takes approximately 10 to 15 sec for a local scan. This overhead grows linearly with the size of the input data. Because SkewTune repartitions more data for reduce tasks than map tasks in these experiments, it follows that the total overhead is larger for reduce tasks. With the same applications and datasets, parallel scans take between 20 and 22 sec. This includes the startup and tear down overhead of the MapReduce job as well as shuffling and sorting overheads when scanning map outputs. This overhead also grows linearly but with a much smaller slope as we discuss below.

"< Compute" represents the time between mitigator planning and the resumption of the data computation. In case of map mitigation, this time only includes the overhead of starting a new job. For reduce mitigation, the overhead includes another scan of the data to repartition and re-shuffle

differs from the worst-case configuration in the ordering of data (the original ordering of the dataset vs sorted by record size) and a smaller chunk size (128MB vs. 256MB).

Figures 8 and 9 show the results. As the figures show, SkewTune adds overhead but that overhead is small. In most cases when applications are already well-tuned and do not exhibit skew, the slots remain busy. SkewTune has few opportunities to improve performance or incur repartitioning overhead. As a result, performance may improve only slightly as in the case of the CloudBurst and Inverted Index reduce phases. In other cases, the runtime can slightly increase. Also with shorter overall runtimes, the overheads of stopping, planning, and re-partitioning become more pronounced. Errors in progress estimation also have more visible effects as does any unnecessarily re-partitioning of nearly completed tasks.

In Figure 9, we also show the result of the REHASH tech-

Figure 10: Overhead of Local Scan vs. Parallel Scan. Time was measured under heavy disk load. For small data sizes (< 1 GB), local scan is faster. For large data sizes (> 1 GB), parallel scan is faster.

that data.

Overhead of Local Scan vs. Parallel Scan: In all three applications and datasets, the size of remaining data during skew mitigation is small (< 1 GB). Thus, SkewTune always performs a local scan rather than a parallel scan. To evaluate the trade-off between the two approaches, we compared the performance of the two scan strategies using a synthetic workload. Figure 10 shows the results. We generated random datasets with different sizes and evenly distributed them across all 20 nodes. To simulate a realistic environment, we loaded all the disks using two background writer processes per disk and dropped the disk cache before the scan. The timing of parallel scan includes the MapReduce job startup and cleanup overhead. In our 20 node cluster, parallel scan performs better than local scan if the size of remaining data is greater than 1 GB. With smaller data, the MapReduce job overhead dominates the I/O time. However, once the data becomes large enough, the overhead pays off by reading a small amount of data per disk while local scan has to sequentially read the data from a single disk. Clearly, the gain will diminish if there exists a significant skew in the amount of distributed input data. For example, for 8 GB of data, local scan takes 890 s but parallel scan takes 679 s when a node has 7.2 GB of data.

Summary: The above experiments show that SkewTune effectively mitigates skew whether it is intrinsic to the application, caused by a misconfiguration, or due to an unfortunate input data order. SkewTune also delivers consistently fast runtimes independent of initial job configuration parameters. SkewTune's overhead is small to none when there is no skew. Finally, the greatest overhead component of re-partitioning a straggler's data comes from the data scans necessary for planning and re-allocating the data. SkewTune's ability to perform these scans in parallel when possible, however, effectively keeps these overheads low even when large datasets need to be repartitioned.

6. RELATED WORK

MapReduce Stragglers. Dean and Ghemawat first describe the straggler problem and solution (execute a redundant copy of a task-in-progress on a different node) in their original MapReduce paper [6]. Zaharia *et al.* [38] extend the approach to clusters with heterogeneous hardware. Ganeshi *et al.* [1] develop a method that improves the decision process of when to either restart a task or execute a duplicate task and where to schedule it. Restarting or du-

plicating straggling tasks, however, only helps alleviate skew problems due to inadequate resources available during task execution. In contrast, SkewTune re-allocates work among tasks to mitigate skew that is intrinsic to the computation. Finally, resource-aware scheduling techniques [1] are complementary to SkewTune.

Handling Data Skew in Parallel Systems: The data skew problem has been extensively researched in the parallel database literature, but only in the context of parallel joins [7, 29, 36, 37] and parallel aggregate operators [32]. These techniques carry over to MapReduce-type platforms. For example, the Pig system includes a SkewedJoin [10] adapted from the literature [7]. In general, however, to leverage these techniques users must implement them directly when writing their user-defined operators (*e.g.*, [30]).

Skew has also been studied previously in the context of MapReduce applications. In earlier work, we proposed SkewReduce, a system that statically optimizes the data partitioning according to user-defined cost functions [21]. The approach effectively addresses potential data skew problems, but it relies on domain knowledge from users and is limited to specific types of applications. Ibrahim *et al.* and Gufler *et al.* studied data skew in the reduce phase [12, 16]. Both approaches schedule reduce keys to the reduce tasks based on cost models. In both systems, the reduce key scheduling does not preserve the order as in the original reduce output. SkewTune not only addresses skew in both the map and reduce phases but also minimizes the side-effect of skew mitigation by preserving input order. Finally, Lin proposed an application-specific solution [25] to skew in the map phase: disproportionately large records were split into smaller ones to improve load balance between mappers.

Adaptive Processing: FLUX [31] splits an operator into mini-partitions. As the pipeline, which includes the operator, runs, FLUX observes machines and computes their percent utilization by measuring the fraction of time they spent being idle. It then moves mini-partitions from the most heavily utilized to the most lightly utilized machines. In our case this is equivalent to running many mappers and many reducers and scheduling them as resources become available. However, running a large number of small tasks has been shown to create significant overhead [21]. Instead, SkewTune only creates additional tasks when necessary.

Optimizing MapReduce Programs: Dittrich *et al.* proposed the Hadoop++ system that optimizes MapReduce jobs by leveraging indexing and join techniques [9]. Herodotou *et al.* proposed the Starfish optimization framework that hierarchically optimizes from MapReduce jobs to workflows by searching for good parameter configurations [15]. The Starfish framework utilizes dynamic profiling to capture the runtime behavior of map and reduce at the granularity of phase level and helps users fine tune Hadoop job parameters. The goal of these previous works is improving the performance of MapReduce jobs by leveraging database techniques or finding a good set of configuration parameters. SkewTune aims at automatically reacting to unexpected data skew encountered at runtime. All works including SkewTune share a common subgoal: minimize user intervention when trying to obtain the best performance out of a MapReduce system.

7. CONCLUSION

In this paper, we presented SkewTune, a system that au-

tomatically mitigates skew in a broad class of user defined operations implemented as MapReduce jobs. SkewTune requires no input from users. It is broadly applicable as it makes no assumptions about the cause of the skew but instead observes the job execution and re-balances load as resources become available. SkewTune is also capable of preserving the order and partitioning properties of the output of the original unoptimized job, making it transparently compatible with existing code, even in the context of complex workflows and advanced MapReduce algorithms.

Experimental results show that SkewTune can deliver a factor of 4X improvement over Hadoop on real and representative datasets and real, non-trivial UDOs. At the same time, it adds little to no overhead when skew is not present. Finally, it provides for much more consistent job execution times for jobs that sometimes incur skew thereby enabling more predictable performance.

Acknowledgments

8. REFERENCES

[1] 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 *Proc. of the 9th OSDI Symp.*, 2010.

[2] Apache Hadoop Project. Powered By Hadoop. http://wiki.apache.org/hadoop/PoweredBy/, 2011.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proc. of the First SOCC Conf.*, pages 119–130, 2010.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the 7th WWW Conf.*, pages 107–117, 1998.

[5] Bu et. al. HaLoop: Efficient iterative data processing on large clusters. *Proc. of the VLDB Endowment*, 3(1), 2010.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.

[7] D. DeWitt, J. Naughton, D. Schneider, and S. S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the 18th VLDB Conf.*, 1992.

[8] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *Proc. of the VLDB Endowment*, 1(1):28–41, 2008.

[9] J. Dittrich, J.-A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proc. of the VLDB Endowment*, 3(1), 2010.

[10] 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. *Proc. of the VLDB Endowment*, 2(2), 2009.

[11] Google. Freebase Data Dumps. http://download.freebase.com/datadumps/, 2010.

[12] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Handling data skew in mapreduce. In *The First International Conference on Cloud Computing and Services Science*, 2011.

[13] Hadoop. http://hadoop.apache.org/.

[14] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of the SIGMOD Conf.*, 1997.

[15] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of the Fifth CIDR Conf.*, 2011.

[16] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24, 2010.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of the EuroSys Conf.*, pages 59–72, 2007.

[18] Jimmy Lin. Cloud 9: A MapReduce library for Hadoop. http://www.umiacs.umd.edu/~jimmylin/Cloud9/docs/index.html, 2010.

[19] M. Kalyuzhnaya, D. Beck, and L. Chistoserdova. Functional metagenomics of methylotrophs. *Methods in Enzymology*, 495, 2011.

[20] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *Proc. of the PODS Conf.*, pages 223–234, 2011.

[21] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the First SOCC Conf.*, June 2010.

[22] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in mapreduce applications. In *The 5th Open Cirrus Summit*, 2011.

[23] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc. of the 22nd Scientific and Statistical Database Management Conference (SSDBM)*, 2010.

[24] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A Platform for Scalable One-Pass Analytics using MapReduce. In *Proc. of the SIGMOD Conf.*, June 2011.

[25] J. Lin. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, number July, 2009.

[26] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *Proc. of the 26nd ICDE Conf.*, Mar. 2010.

[27] O. O'Malley. Apache hadoop wins terabyte sort benchmark. http://developer.yahoo.com/blogs/hadoop/posts/2008/07/apache_hadoop_wins_terabyte_sort_benchmark/.

[28] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. R. Madden, and M. Stonebraker. A comparison of approaches to large scale data analysis. In *Proc. of the SIGMOD Conf.*, 2009.

[29] V. Poosala and Y. E. Ioannidis. Estimation of Query-Result Distribution and its Application in Parallel-Join Load Balancing. In *Proc. of the 22nd VLDB Conf.*, Sept. 1996.

[30] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, June 2009.

[31] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the SIGMOD Conf.*, June 2004.

[32] A. Shatdal and J. Naughton. Adaptive Parallel Aggregation Algorithms. In *Proc. of the SIGMOD Conf.*, 1995.

[33] T. M. Team. Apache mahout project. http://mahout.apache.org/.

[34] Tyson Condie et. al. MapReduce online. In *Proc. of the 7th NSDI Symp.*, 2010.

[35] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proc. of the SIGMOD Conf.*, June 2011.

[36] C. Walton, A. Dale, and R. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of the 17th VLDB Conf.*, 1991.

[37] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel dbms. *Proc. of the VLDB Endowment*, 2(2), 2009.

[38] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proc. of the 8th OSDI Symp.*, 2008.