Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions

YongChul Kwon, Magdalena Balazinska,
Bill Howe
University of Washington
{yongchul,magda,billhowe}@cs.washington.edu

Jerome Rolia
HP Labs
jerry.rolia@hp.com

ABSTRACT

Scientists today have the ability to generate data at an unprecedented scale and rate and, as a result, they must increasingly turn to parallel data processing engines to perform their analyses. However, the simple execution model of these engines can make it difficult to implement efficient algorithms for scientific analytics. In particular, many scientific analytics require the extraction of features from data represented as either a multidimensional array or points in a multidimensional space. These applications exhibit significant computational skew, where the runtime of different partitions depends on more than just input size and can therefore vary dramatically and unpredictably. In this paper, we present SkewReduce, a new system implemented on top of Hadoop that enables users to easily express feature extraction analyses and execute them efficiently. At the heart of the SkewReduce system is an optimizer, parameterized by user-defined cost functions, that determines how best to partition the input data to minimize computational skew. Experiments on real data from two different science domains demonstrate that our approach can improve execution times by a factor of up to 8 compared to a naive implementation.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*parallel databases*

General Terms

Algorithms, Design, Experimentation

1. INTRODUCTION

Science is becoming data-intensive [23, 34]. As a result, scientists are moving data analysis activities off of the desktop and onto clusters of computers — public and private "clouds". Programming these clouds for scientific data analysis remains a challenge, however, despite the proliferation of parallel dataflow frameworks such as MapReduce [6] and Dryad [19].

Specialized data models represent one source of challenges. In particular, multi-dimensional arrays and other order-sensitive data structures are common, and data values are often associated with coordinates in space or time. For example, images in astronomy are 2D arrays of pixel intensities, and each element corresponds to a point in the sky and a time at which the image was taken. Climate and ocean models use arrays or meshes to describe 3D regions of the atmosphere and oceans, simulating the behavior of these regions over time by numerically solving the governing equations. Cosmology simulations model the behavior of clusters of 4D particles to analyze the origin and evolution of the universe. Flow cytometry technology uses scattered light to recognize microorganisms in water, generating an enormous volume of events in a 6D space corresponding to different wavelengths of light. These application domains all must reason about the *space in which the data is embedded* as well as the data itself. The spatial relationships of the data pose challenges for conventional "loosely-coupled" shared-nothing programming paradigms because data must be range-partitioned across nodes rather than hash-partitioned, complicating load balancing.

A second source of challenges result from specialized science algorithms that must be re-expressed in these new cloud platforms. The translation is challenging even if these platforms provide high-level, declarative interfaces [16, 25, 41]. Indeed, science algorithms often need to extract some features from the data (*e.g.*, clustering, image segmentation, and object recognition) possibly also tagging the original data with these features. Engineering the translation of such algorithms in a way that preserves performance is nontrivial. For example, in prior work, we found that a naïve implementation of a data clustering algorithm on a real astronomy simulation dataset took 20 hours to complete on an 8-node Dryad [19] cluster. In contrast, an optimized version took only 70 minutes, but took multiple weeks to develop and debug by a team of domain and computer scientists [21].

In this paper, we address these challenges by presenting and evaluating SkewReduce, a new shared-nothing parallel data processing system designed to support *spatial feature extraction* applications common in scientific data analysis.

We observe that these applications share a common structure that can be parallelized using the following strategy: (1) Partition the multidimensional space and assign each node a contiguous region, (2) run a serial form of the analysis locally on each region, extracting locally found features

and labeling the input data with these features if necessary, (3) efficiently merge the local results by considering only those features that cross region boundaries, re-labeling the input data as necessary. Although this formulation is simple and sound, a naïve implementation on existing parallel data processing engines is dominated by skew effects and other performance problems.

The standard approach to handling skew in parallel systems is to assign an equal number of data values to each partition via hash partitioning or clever range partitioning. These strategies effectively handle *data skew*, which occurs when some nodes are assigned more data than others. *Computation skew*, more generally, results when some nodes take longer to process their input than other nodes and can occur even in the absence of data skew — the runtime of many scientific tasks depends on the data values themselves rather than simply the data size [17].

Existing parallel processing engines offer little support for tolerating general computation skew, so scientific programmers are typically forced to develop ad hoc solutions. At realistic scales, these ad hoc solutions not only require intimate familiarity with the source data, but also expertise in distributed programming, scheduling, out-of-core processing, performance monitoring and tuning, fault-tolerance techniques, and distributed debugging. SkewReduce efficiently reduces computational skew and helps scientific programmers express their solutions in popular parallel processing engines such as MapReduce.

In addition to skew, two other sources of performance problems are the merge and data labeling steps. Because of large data volumes, it may not be efficient or even possible to execute the merge phase on a single node. Instead, feature reconciliation must be performed incrementally in a hierarchical fashion. Similarly, intermediate results must be *set aside* to disk during the merge phase, then re-labeled in parallel after the merge phase is complete to obtain the final result. While both these strategies can be implemented in existing systems, doing so is non-trivial. Additionally, the same type of translation is repeated independently for each new feature extracting application.

Approach Informed by the success of MapReduce [6], SkewReduce is a new parallel computation framework tailored for spatial feature extraction problems to address the above challenges. To use the framework, the programmer defines three (non-parallel) data processing functions and two cost functions to guide optimization. Given this suite of functions, the framework provides a parallel evaluation plan that is demonstrably efficient and — crucially — skew-tolerant. The plan is then executed in a Hadoop cluster. We show that this framework delivers significant improvement over the status quo. The improvement is attributable primarily to the reduction of skew effects, as well as the elimination of performance issues in the merge and labeling steps. Further, we argue that the cognitive load for users to provide the suite of control functions is significantly less than that required to develop an ad hoc parallel program. In particular, the user remains focused on their application domain: they specify their analysis algorithm and reason about its complexity, but do not concern themselves with distributed computing complications.

Overall, SkewReduce is a powerful new framework that provides a complete and general solution to an important class of scientific analysis tasks.

Technical contributions: We deliver the following contributions: we present the SkewReduce system for efficiently processing spatial feature extraction scientific user-defined functions. SkewReduce comprises (1) a simple API for users to express multidimensional feature extraction analysis tasks (Section 3.1) and (2) a static optimization engine designed to produce a *skew-resistant plan* for evaluating these tasks (Section 3.2 and 3.3). SkewReduce is implemented using Hadoop [15]. (3) We demonstrate the efficacy of our framework on real data from two different science domains (Section 4). The results show that SkewReduce can improve query runtime by a factor of up to 8 compared with an unoptimized implementation.

2. MOTIVATION

We begin by describing three motivating applications from different scientific domains. We then discuss the commonalities between these applications and the challenges that arise when trying to implement them on a MapReduce-type platform.

Astronomy Simulation. Cosmological simulations are used to study the structural evolution of the universe on distance scales ranging from a few million light-years to several billion light-years. In these simulations, the universe is modeled as a set of particles. These particles represent gas, dark matter, and stars and interact with each other through gravity and fluid dynamics. Every few simulation timesteps, the simulator outputs a *snapshot* of the universe as a list of particles, each tagged with its identifier, location, velocity, and other properties. The data output by a simulation can thus be stored in a relation with the following schema:

`Particles(`*id*, *time*, x, y, z, v_x, v_y, v_z, $\cdots$`)`

State of the art simulations (*e.g.*, Springel *et al.* 2005 [31]) use over 10 billion particles producing a data set size of over 200 GB per snapshot and are expected to significantly grow in size in the future.

Astronomers commonly used various sophisticated clustering algorithms [13, 20, 37] to recognize the formation of interesting structures such as galaxies. The clustering algorithm is typically executed on one snapshot at a time [21]. Given the size of individual snapshots, however, astronomers would like to run their clustering algorithms on a parallel data processing platform in a shared-nothing cluster.

Flow Cytometry. A flow cytometer measures scattered and fluoresced light from a stream of particles, using data analysis to recognize specific microorganisms. Originally devised for medical applications, it has been adapted for use in environmental microbiology to determine the concentrations of microbial populations. Similar microorganisms exhibit similar intensities of scattered light, as in Figure 1.

In an ongoing project in the Armbrust Lab at the University of Washington [2], flow cytometers are being continuously deployed on ocean-going vessels to understand the ocean health. All data is reported to a central database for ad hoc analysis and takes the form of points in a 6-dimensional space, where each point represents a particle or organism in the water and the dimensions are the measured properties.

As in the astrophysics application, scientists need to cluster the resulting 6D data. As their instruments increase in sophistication, so does the data volume, calling for efficient analysis techniques that can run in a shared-nothing cluster.

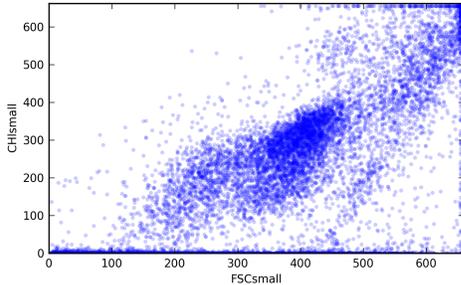**Image Processing**. As a final example, consider the

Figure 1: A scatter plot of flow cytometry measurements. Each point represents an organism and clusters represent populations. The axes correspond to different wavelengths of light.

problem of analyzing collections of 2D images. In many scientific disciplines, scientists process such images to extract objects (or features) of interest: galaxies from telescope images, hurricanes from satellite pictures, etc. As these images grow in size and number, parallel processing becomes necessary.

General Feature Extracting Applications. Each of these scientific applications follow a similar pattern: data items (events, particles, pixels) are embedded in a metric space, and the task is to identify and extract emergent features from the low-level data (populations, galaxies). These algorithms then typically return (a) a set of features (significantly smaller than the input data), (b) a modified input dataset with each element tagged with the corresponding feature (potentially as large as the input), or (c) both. For example, the output of the astronomy clustering task is a list of clusters with the total number of particles in each and a list of the original particles annotated with their cluster identifier.

Parallel Implementation Challenges. A straightforward way to parallelize such feature extraction applications in a compute-cluster with N nodes is the following: (1) split the input into N equal-sized hypercubes, (2) extract features in each partition and annotate the input with these initial features, (3) reconcile features that span partition boundary, relabeling the input as appropriate. With existing parallel processing systems, there are several challenges with expressing this seemingly simple algorithm in a manner that achieves high performance.

First, the data distribution in many scientific applications is highly skewed. Even worse, the processing time of many feature-extraction algorithms depends not only on the number of data points but also on their distribution in space. For example, in a simple clustering algorithm used in astrophysics called "friends-of-friends" [5], clusters correspond to connected components of the graph induced by the "friend" relationship — two particles are friends if they are within a given distance threshold. To identify a cluster, the algorithm starts with a single point, then searches a spatial index to find its immediate friends. For each such friend, the algorithm repeats the search recursively. In a sparse region with N particles, the algorithm completes in $O(N \log N)$ time (*i.e.*, all particles are far apart). In a dense region, however, a single particle can be a friend of all the other particles and vice versa. Thus, the algorithm takes $O(N^2)$ time. In the two simulation snapshots that we received from astronomers [21], we found that the number of friends associated with a given particle varied between 2 and $387,136$. As a result, without additional optimizations, a dense region takes much longer to process than a sparse one even when both contain the same number of total particles [21]. The consequence is a type of computational skew, where some data partitions require dramatically more time than others to process. Computational skew is the reason that the naïve parallel implementation of the astronomy clustering application mentioned in Section 1 required over 20 hours, while an optimized one took only 70 minutes on the same dataset [21]. Our key motivation is that *existing platforms do nothing to reduce computational skew.* In our case, developing a skew-resistant algorithm (by optimizing index traversal to avoid quadratic behavior in the dense region) required significant effort from multiple experts over several weeks [21].

Second, the feature reconciliation phase (which we refer to as the "merge" phase) can be both CPU and memory intensive. For example, to reconcile clusters at the boundary of two data partitions requires processing all particles within a small distance of that boundary. If the space is initially carved into N partitions, it may not be efficient or even possible for a single node to reconcile the data across all these partition boundaries in one step. Instead, reconciliation should be performed in a hierarchical fashion, reconciling increasingly large regions of the space, while keeping the amount of data to process at each step approximately constant (*i.e.*, the memory requirement cannot increase as we move up the hierarchy). At the same time, while the local data processing and later merge steps proceed, the input data must be labeled and re-labeled as necessary, *e.g.*, to track feature membership. While it is possible to implement both functions using existing systems, expressing them using current APIs is non-trivial.

Problem Statement Summary. The goal of SkewReduce is to enable scientists to *easily express* and *efficiently execute* feature-extraction applications at very large scale without consideration of resource constraints and data or computation skew issues.

3. SkewReduce

SkewReduce has two components. The first component is an API for expressing spatial feature-extraction algorithms such as the ones above. We present the API in Section 3.1. The functions in our API are translated into a dataflow that can run in a MapReduce-type platform [6, 15, 19]. The second component of SkewReduce is a static optimizer that partitions the data to ensure skew-resistant processing if possible. The data partitioning is guided by a user-defined cost function that estimates processing times. We discuss the cost functions in Section 3.2 and the SkewReduce optimizer in Section 3.3.

3.1 Basic SkewReduce API

Informed by the success of MapReduce [6], the basic SkewReduce API is designed to be a minimal control interface allowing users to express feature extraction algorithms in terms of serial programs over familiar data structures. The *basic SkewReduce API* is the minimal interface that must be implemented to use our framework. The basic API

Table 1: Summary of notation

T	A record in the original input data file assigned to a region (*e.g.*, a particle in an astronomy simulation)
S	A record set aside during the process phase or merge phase. (*e.g.*, a particle far away from a partition boundary tagged with a local cluster id).
F	An object representing a set of features extracted during the process phase for a given region. May not be relational. Includes enough information to allow reconciliation of features extracted in different partitions (*e.g.*, the clusters identified so far and the particles near a partition boundary)
Z	A record in the final result set (*e.g.*, a particle tagged with a global cluster id)

is

$$
\begin{aligned}
\text{process} &:: \langle\text{Seq. of } T\rangle \rightarrow \langle F, \text{Seq. of } S\rangle \\
\text{merge} &:: \langle F, F\rangle \rightarrow \langle F, \text{Seq. of } S\rangle \\
\text{finalize} &:: \langle F, S\rangle \rightarrow \langle\text{Seq. of } Z\rangle
\end{aligned}
$$

The notation used in these types is defined in Table 1. At a high-level, T refers to the input data. F is the set of features and S is an output data field that must be tagged with the features F to form Z. The above three functions lead to a very natural expression of feature extracting algorithms: First, partition the data (not shown). Second, apply **process** to each partition to get an initial set of local features and an initial field. Third, **merge**, or reconcile, the output of each local partition to identify a global set of features. Finally, adjust the output of the original **process** functions given the final, global structures output by **merge**. For example, in the case of the astronomy simulation clustering task, **process** identifies local clusters in a partition of the 3D space. **merge** hierarchically reconciles local clusters into global clusters. Finally, the **finalize** function relabels particles initially tagged by **process** with a local cluster ID using the appropriate global cluster ID.

The functions of the SkewReduce API loosely correspond to the API for distributed computation of algebraic user-defined aggregates found in OLAP systems and distributed dataflow frameworks. For example, Yu *et al.* propose a parallel aggregation framework consisting of functions **initialreduce**, **combine**, and **finalreduce** [40]. The function **initialreduce** generates intermediate partial aggregates, **combine** merges partial aggregates, and the final aggregate value can be further transformed by **finalreduce**.

The distinguishing characteristic of our API is that our analog of the **initialreduce** and **finalreduce** functions return two types of data: a representation of the extracted features, and a representation of the "tagged" field. A given algorithm may or may not use both of these data structures, but we have found that many do.

We now present the three functions in SkewReduce's API in more detail.

3.1.1 Process: Local Computation with Set-Aside

The **process** function locally processes a sequence of input tuples producing F, a representation of the extracted features, and Seq. of S, a sequence of tuples that are set aside from the hierarchical reconciliation. In our astronomy simulation use-case, **process** performs the initial clustering of particles within each partition. Although we can forward all the clustering results to the **merge** function, only par-

ticles near the boundary of the fragment are necessary to merge clusters that span two partitions. Thus, **process** can optionally *set aside* those particles and results that are not required by the following **merge**s. This optimization is not only helpful to reduce the memory pressure of **merge** but also improves overall performance by reducing the amount of data transferred over the network. In this application, our experiments showed that almost 99% of all particles can thus be set aside after the Process phase (Figure 9).

3.1.2 Merge: Hierarchical Merge with Set-Aside

The **merge** function is a binary operator that combines two intermediate results corresponding to two regions of space. It takes as input the features from each region and returns a new merged feature set. The two feature set arguments are assumed to fit together in the memory of one node. This constraint is a key defining characteristic of our target applications. This assumption is shared by most user-defined aggregate frameworks [26, 32, 40]. However, SkewReduce provides more flexibility than systems designed with trivial aggregation functions such as sum, count, average in mind. Specifically, we acknowledge that the union of all feature sets may not fit in memory, so we allow the **merge** function to set aside results at each step. In this way, we ensure that the size of any value of type F does not grow larger than memory. We acknowledge that some applications may not exhibit this property, but we have not encountered them in practice. We assume that both functions **process** and **merge** set aside data of the same form. This assumption may not hold in general, but so far we have found that applications either set aside data in the **process** phase or in the **merge** phase, but not both.

In our running example, the **merge** function combines features from adjacent regions of space, returning a new feature object comprising the bounding box for the newly merged region of space, the cluster id mappings indicating which local clusters are being merged, and the particles near the boundary of the new region. Figure 2 illustrates the merge step for four partitions P1 through P4. The outer boxes, Pi, represent the cell boundaries. The inner boxes, I, are a fixed distance ϵ away from the corresponding edge of the region. The local clustering step, **process**, identified a total of six clusters labeled C1 through C6. Each cluster comprises points illustrated with a different shade of gray and shape. However, there are only three clusters in this dataset. These clusters are identified during the hierarchical **merge** step. Clusters C3, C4, C5, and C6 are merged because the points near the cell boundaries are within distance ϵ of each other. In Figure 2, C2 does not merge with any other cluster because all points in C2 are sufficiently far from P1's boundary. We can thus safely discard C2 before merging: These points are not needed during the merge phase. In general, we can discard all the points in the larger I regions before merging, reducing the size of the input to the merging algorithm. This reduction is necessary to enable nodes to process hierarchically larger regions of space without exhausting memory.

3.1.3 Finalize: Join Features with Set-Aside Data

The **finalize** function can be used to implement a join between the final collection of features and the input representation as output by the **process** and **merge** functions. This function is useful for tagging the original data elements with
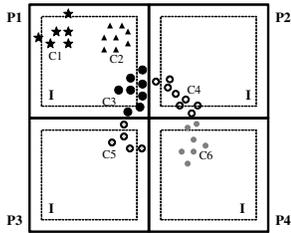
Figure 2: Illustration of the merge step of the clustering algorithm in the SkewReduce framework. Data is partitioned into four chunks. Points with the same shape are in the same global cluster. Point with different colors but with identical shapes are in different local clusters (*e.g.*, the circles in the middle of the figure). Each P*i* labels the cell boundary and each *I* labels the interior region. Only points outside of *I* are needed in the subsequent merge phase. After the hierarchical merge phase, three cluster mappings are generated: (C4,C3), (C5,C3), and (C6,C3). Such mappings are used to relabel local cluster ids during the finalize phase.

their assigned feature. The finalize function accepts the final feature set from the merge phase and a *single tuple* set aside during processing. The SkewReduce framework manages evaluation of this function over the entire distributed dataset.

Our emphasis on distinguishing "features" and "set aside" data may at first appear to be over-specialized to our particular examples, but we find the idiom to be quite general. To understand why, consider the analogous distinction between *vector* and *raster* representations of features. For example, Geographic Information Systems (GIS) may represent a geographic region as an image with each pixel assigned a value of "road", "waterway", "building", etc. (the raster representation). Alternatively, these objects may be represented individually by line segments, polygons, or some other complex object (the vector representation). Neither representation is ideal for all algorithms, so both are frequently computed and maintained. In our running example, the tagged particles are analogous to the raster representation — each point in the original dataset is labeled with the feature to which it contributes.

The user thus specifies the above three functions. Given these functions, SkewReduce automatically partitions the input data into hypercubes and schedules the execution of the process, merge, and finalize operators in a Hadoop cluster. We further discuss the details of the Hadoop translation in Section 4. The partition plan is derived by SkewReduce's optimizer, as we discuss below.

In many application domains the process function satisfies the following property:

DEFINITION 3.1. **process** *Monotonicity For datasets* R, S *where* $R \subseteq S$, *the execution time* time[*process*(R)] $\leq$ time[*process*(S)] *(Intuition: as data size increases, so must the local processing cost).*

The SkewReduce's optimizer is designed primarily for applications where this property holds. However, it can still handle applications that violate this property, as we discuss in Section 3.3.

For the applications we encounter in practice, we find that process is far more expensive than merge, which causes aggressive partitioning to be generally beneficial. In these cases, the limiting factor in partitioning is the scheduling overhead. In contrast, if merge is expensive or comparable relative to process, partitioning simply ensures that no node is allocated more data than will fit in its memory.

Optional Pre-Processing. The process function operates on a set of records Seq. of T. In some applications, especially those operating on arrays, individual records are not cells but rather small neighborhoods of cells, sometimes called *stencils*. This distinction is not an issue for process, which receives as input a contiguous block of cells and can thus extract stencil neighborhoods unilaterally. However, since the optimizer operates on a sample of the input data, SkewReduce must apply a pre-processing step that extracts application-defined *computational units* before sampling them. For this reason, although not part of the basic API, we allow a user to provide a custom function to transform a sequence of "raw" records into a sequence of computational units, Seq. of T.

3.2 Cost Functions

We have presented the basic SkewReduce API, but we have not explained how skew is handled. Both the process and merge phases of the API are crucially dependent on the initial partitioning of data into regions. Feature extraction applications often exhibit both data skew and computational skew, and both are determined by how the data are partitioned. Datasets prone to significant data and computational skew (usually due to extreme variations in data density) can be processed efficiently if an appropriate partition-and-merge plan can be found. As we will show, plan quality can be improved dramatically if the user can estimate the runtime costs of their process and merge functions.

We allow the user to express these costs by providing two additional cost functions C_p and C_m, corresponding to process and merge, respectively. These cost functions operate serially on samples of the original dataset returning a real number; that is:

$$C_p \quad :: \quad (S, \alpha, B) \to \mathbb{R}$$
$$C_m \quad :: \quad (S, \alpha, B) \times (S, \alpha, B) \to \mathbb{R}$$

where S is a sample of the input, α is the sampling rate, and B is a bounding hypercube.

The cost functions accept both a representation of the data (the sample S) and a representation of the region (the bounding hypercube B, represented as a sequence of ranges, one for each dimension). The cost of the feature extraction algorithms we target is frequently driven by the distribution of the points in the surrounding space. One approach to estimate cost inexpensively is therefore to build a histogram using the bounding hypercube and the sample data and compute an aggregate function on that histogram. The sampling rate α allows the cost function to properly scale up the estimate to the overall dataset. When discussing cost functions in the remainder of this paper, we omit the bounding hypercube and sampling rate parameters when they are clear from the context.

Given a representative sample, the cost functions C_p and C_m must be representative of actual runtimes of the process and merge functions. More precisely, the functions must satisfy the following properties.

- **Fidelity** For samples R, S, if $C_p(R) < C_p(S)$, then time[process(R)] < time[process(S)] (intuition: the true cost and the estimated cost impose the same total order on datasets). Similarly, for samples R, S, T, U, if $C_m(R, S) < C_m(T, U)$, then time[merge(R, S)] < time[merge(T, U)].
- **Boundedness** For some constants ρ_p and ρ_m and samples R and S, time[process(R)] $= \rho_p C_p(R)$ and time[merge(R, S)] $= \rho_m C_m(R, S)$

For the boundedness condition, we can estimate the constant factors ρ_p and ρ_m in at least two ways. The first method is to run the process and merge algorithms over a data sample and compute the constants. This type of approach is related to curve fitting for UDF cost estimation [4]. The second method is to derive new constants for a new cost function from past executions of the same analysis.

Many MapReduce-style analytic systems are running on top of chunk-based distributed file systems such as GFS, HDFS, and S3 and use the chunk as a unit of task distribution and computation. SkewReduce takes a similar approach and requires that the process and merge functions have the ability to process at least one chunk-size of input data without running out of memory. Alternatively, we could optionally allow users to specify memory usage estimation functions that take a form analogous to the cost functions above. In both cases, the optimizer ensures a partition plan with sufficient granularity that no operator runs out of memory.

3.3 SkewReduce's Optimizer

There are two potential optimization goals for a SkewReduce application: minimize execution time or minimize resource usage. SkewReduce's current optimizer adopts a traditional approach and *minimizes the query execution time subject to a constraint on the number of available machines in a cluster*. This constraint can be dictated by the size of a locally available cluster or by monetary reasons when using a pay-as-you-go platform such as Amazon EC2 [1]. SkewReduce's optimizer could be used to try alternative cluster sizes if a user tries to find some desired price-performance trade-off, but we leave it for future work to automate such exploration.

SkewReduce's optimizer is designed to operate on a small sample of the entire dataset, so that the optimizer can execute on a user's desktop before the user acquires or even just reserves any resources on a large cluster. In this paper, we do not address the problem of how the user generates such a sample. Such samples are already commonly used for debugging in these environments.

At a high level, SkewReduce's optimizer thus works as follows: given a sample S of the input data, process and merge functions and their corresponding cost functions C_p and C_m, a compute cluster-size constraint of M nodes, and a scheduling algorithm, the optimizer attempts to find the *partitioning plan* that minimizes the total query execution time. The user-supplied cost functions and the scheduling algorithm guide the optimizer's search for such best plan. SkewReduce works best with a task scheduler that minimizes makespan subject to task dependencies. However, it uses the scheduler as a black box and can therefore work with various schedulers.

Since the scheduler is modeled as a black box and the cost functions may not be completely accurate, SkewReduce

does not guarantee to generate an optimal plan. However, our experiments in Section 4 show that it finds very efficient plans in practice (Figure 3).

We begin by defining the SkewReduce partition plan, execution plan, and the optimization problem more precisely.

Partition Plan: A SkewReduce partition plan is a full binary tree where all intermediate nodes represent merge operators and all leaf nodes represent process operators. Each node in the tree is associated with a bounding hypercube defining the region of space containing all data in the partition. The hypercubes at a given height in the tree partition the space; there are no gaps or overlaps.

Valid Partition Plan: A partition plan is valid if no node in the plan is expected to receive more data than will fit in memory. The memory size is applied after scaling the sample data back to the original input data size, assuming the sample, S, is representative. For example, if a 1% data sample leads to a partition with $2,000$ particles, and we know that a single node cannot process more than $100,000$ particles, the plan will not be valid since $2,000 * 100 > 100,000$.

Execution Plan: A SkewReduce execution plan comprises a partition plan and its corresponding schedule using a job scheduling algorithm schedule. A valid execution plan is a valid partition plan and its schedule.

Optimization Problem: Given a sample S of the input data, process and merge functions with their corresponding cost functions and constants (ρ_p, ρ_m), a compute cluster of M nodes, a scheduling algorithm and constant operator scheduling delay (Δ), return the valid execution plan that is estimated to minimize query runtime.

3.3.1 Optimizing the Partition Plan

The search space of the optimizer is the set of all possible partitions of the hypercube defined by the input data. The optimizer enumerates potentially interesting partition plans in this search space using a greedy strategy. This greedy strategy is motivated by the fact that all process cost functions are assumed to be monotonic (Section 3.2).

Starting from a single partition that corresponds to the entire hypercube bounding the input data I, and thus also the data sample, S, the optimizer greedily splits the most expensive leaf partition in the current partition plan. The optimizer stops splitting partitions when two conditions are met: (a) All partitions can be processed and merged without running out of memory; (b) No further leaf-node split improves the runtime: *i.e.*, further splitting a node increases the expected runtime compared to the current plan. Algorithm 1 summarizes this approach.

In order for a partition split to decrease the runtime, the savings in process processing times must outweigh the cost of the extra merge including scheduling overheads. More specifically, in the algorithm, the runtime after the split for the partition is estimated to be the sum of the runtime of the slower of the two new process operators (given that they will most likely be processed in parallel), the runtime of merge, and the task scheduling delay (Δ) for the merge operator. This is compared to the estimated runtime before the split, which was simply the time to run the process operator (line 9). Additionally, the resulting parallel execution plan must be valid and must improve the total estimated runtime. We estimate the total runtime by running the black-box scheduling algorithm. The algorithm updates the current best plan

Algorithm 1 Searching Optimal Partitioning Plan

Input: p_0: root partition
$\quad$ M: the number of machines
Output: P: the best partitioning plan
$\quad$ $bestCost$: the best cost to run P
$\quad$ $schedule$: the schedule of P
1: $P \leftarrow \{p_0\}$
2: $bestCost \leftarrow schedule\_cost(P, M)$
3: $L \leftarrow \{p_0\}$ // all leaf partitions in P
4: **while** $L \neq \emptyset$ **do**
5: $\quad$ $p \leftarrow$ choose the most expensive partition
6: $\quad$ $p_l, p_r \leftarrow$ find best split of p
7: $\quad$ $c \leftarrow \rho_p \max\{C_p(p_l), C_p(p_r)\} + \rho_m C_m(p_l, p_r) + \Delta$
8: $\quad$ $force \leftarrow p$ does not satisfy memory requirement
9: $\quad$ **if** $c < \rho_p C_p(p)$ or $force$ **then**
10: $\quad\quad$ $s \leftarrow schedule\_cost(P \cup \{p_l, p_r\}, M)$
11: $\quad\quad$ **if** $s < bestCost$ or $force$ **then**
12: $\quad\quad\quad$ $L \leftarrow L \cup \{p_l, p_r\}$
13: $\quad\quad\quad$ $P \leftarrow P \cup \{p_l, p_r\}$ // p becomes an internal node.
14: $\quad\quad\quad$ $bestCost \leftarrow s$
15: $\quad\quad$ **end if**
16: $\quad$ **end if**
17: $\quad$ $L \leftarrow L - \{p\}$
18: **end while**
19: **if** all $p \in P$ satisfies memory requirement **then**
20: $\quad$ **return** $(P, bestCost, schedule(P))$
21: **else**
22: $\quad$ **raise** failed to find a valid plan
23: **end if**

Algorithm 2 Searching best split for a given partition

Input: B: bounding hypercube
$\quad$ S: sample data bounded by B
Output: $bestSplit$: axis and splitting point
1: $bestCost \leftarrow \infty$
2: $bestSplit \leftarrow null$
3: $A \leftarrow chooseAxes(B, S)$
4: **for all** $axis \in A$ **do**
5: $\quad$ $split \leftarrow$ find best split point along $axis$
6: $\quad$ $B_l, B_r \leftarrow$ split B at $split$ along $axis$
7: $\quad$ $c \leftarrow \max\{C_p(B_l, S), C_p(B_r, S)\}$
8: $\quad$ **if** $c < bestCost$ and satisfies merge memory requirement **then**
9: $\quad\quad$ $bestSplit \leftarrow (axis, split)$
10: $\quad\quad$ $bestCost \leftarrow c$
11: $\quad$ **end if**
12: **end for**
13: **return** $bestSplit$

---

only when the runtime improves (line 10-11) or if a split is mandatory due to memory constraints. We perform this two-level filtering to reduce the number of calls to the scheduler function.

The optimizer returns an error if no valid partition plan exists. That is, if in all considered partition plans at least one process or one merge operators run out of memory (line 22).

The algorithm uses two key subroutines: finding the best point where to split a partition in two (line 5) and estimating the cost of a schedule in terms of processing time (line 10). In the following subsections, we discuss each of these two subroutines and SkewReduce's default implementation of these routines. Alternatively, the user may also supply custom implementations.

3.3.2 Partition Splitting

When splitting a hypercube in two, the optimizer has two choices to make: which axis to use for the split and at what point along this axis to perform the split.

An ideal split should partition the data into two subpartitions with identical real runtimes. In contrast, the worst split creates two subpartitions with very different real runtimes, with the runtime for the slower subpartition similar to the pre-split runtime.

Algorithm 2 shows the optimizer's approach to choosing the split axis and split point for a given partition. This algorithm applies the user-defined cost functions on the data sample, S, to estimate execution times.

For a low dimensional data, typically 3 to 4, the optimizer exhaustively tries to split the data along each of the available axes because the optimization process is low-overhead (as we

show later in Figure 8). For a high dimensional data, the user can supply a heuristic to filter out bad split axes to improve optimization time. We define the best split to be the one that minimizes the maximum cost C_p of any of the subpartitions created without violating the merge memory requirement.

To select the point along an axis where to split the data, different algorithms are possible. We present and compare three strategies. All three methods require that the examined sample data be sorted along the splitting axis with tie-breaking using values in other dimensions. Thus, we sort the sample data before run the strategy.

Discrete: The Discrete approach considers splitting the data at each one of n uniformly-spaced points along the splitting-axis. n is given as a parameter. For each point, the discrete strategy computes the cost of splitting the data at that point. The discrete approach is thus the most general strategy because it can work even when the cost function is not monotonic. It simply tries all possible splitting points assuming a given minimum granularity. On the other hand, this strategy may not return the best estimated splitting point, especially if n is small.

Binary Search: This approach requires that cost functions be monotonic and performs a binary search for the best split point. The algorithm terminates after examining all $\log |S|$ candidate split points. Binary search always returns the optimal split as estimated by the cost function.

Incremental Update: The Incremental Update approach requires that the cost function be monotonic and incrementally updatable. That is, whenever the cost function is updated with a sample through an API call, the new cost is returned. Given these restrictions, the Incremental Update approach achieves the best optimization performance. The approach searches for the best split point in two phases. The algorithm starts with two empty subpartitions. It continuously adds samples to these subpartitions starting at both ends of partitioning axis. Each new data point is added to the partition currently estimated to have the lower runtime. The algorithm terminates when all samples have been assigned to a subpartition and the splitting point is the mid-point between the last sample inserted into each partition.

If multiple points fall on the partition boundary, the algo-

rithm enters a second phase, where it computes the fraction of such points that were assigned to each partition. At runtime, when the entire dataset is partitioned, points on the same partition boundary are randomly distributed to sub-partitions according to these precomputed proportions.

3.3.3   Estimating the Cost of a Schedule

The newly split partitions are only added if the candidate plan yields a better total runtime than the current plan. We estimate the runtime by calling a black box scheduling function schedule. To match the units of the operator costs to those of the scheduling overheads, we scale the process and merge costs using the pre-computed ρ_p, ρ_m constants, thus converting these costs into time units.

Converting a schedule to a cost estimate is straight forward; we invoke the scheduling algorithm with the costs of all operators and M slots as input then take the total runtime. While we leave the scheduling algorithm as a black box, we found that Longest Processing Time (LPT) scheduling algorithm [14] works well in practice and satisfies all necessary features such as job dependency and multiple slots. Thus, we use LPT algorithm in the prototype.

4.   EVALUATION

In this section, we evaluate the performance of SkewReduce on the friends-of-friends clustering task over datasets from two different domains: astronomy and oceanography (see Section 2). Table 2 summarizes the properties of the two datasets. We implemented friends-of-friends in a straightforward fashion without any optimizations, and using a standard KD-tree for storing local data and looking up friends.

Summary. We answer the following questions: (1) Does SkewReduce improve task completion times compared to uniform data partitioning, and, if so, is the difference significant? (2) How important is the fidelity of the cost model for SkewReduce's optimization? (3) How does the sample size affect cost estimates and ultimately performance? (4) What is the overhead of scheduling and optimization in SkewReduce? Our results show that SkewReduce imposes a negligible overhead (Figure 8) and can decrease total runtime by a factor of 2 or more compared to uniform data partitioning (Figure 3). We also find that small sample sizes of just 1% suffice to guide optimization, but the quality of the resulting plan does depend on the characteristics of the sample (Figures 6 and 7). Finally, a cost function that better captures the analysis algorithms helps SkewReduce find better plans, but even an approximate cost function can improve runtime compared to not using SkewReduce at all (Figures 4 and 5).

Implementation. The SkewReduce prototype consists of two Java classes: the SkewReduce optimizer and the SkewReduce execution engine. The optimizer takes the cost model and sample data as input and produces an optimized partition plan and a corresponding schedule. The execution engine converts the plan into a graph of Hadoop jobs and submits them to Hadoop according to the schedule from the optimizer. SkewReduce deploys a full MapReduce job for the initial data partitioning task (if necessary) and for each finalize operator, but deploys a map-only job for each process or merge operator. This design gives us better control over the timing of the schedule because Hadoop only supports user specified priorities at the job level rather than at the task level.

SkewReduce minimizes the scheduling overhead by using

Table 2: Datasets used in the evaluation

Dataset	Size	# items	Description
Astro	18 GB	900 M	Cosmology simulation
Seaflow	1.9 GB	59 M	Flow Cytometry

Table 3: Cost-to-time conversion constant for cost models (ρ_p, ρ_m,scale)

	Data Size			Histogram 1D			Histogram 3D		
Astro	83	4.3	10^{-6}	1500	2.9	10^{-12}	3.0	40	10^{-7}
Seaflow	4.8	1.6	10^{-5}	9.3	130	10^{-12}	6.0	200	10^{-8}

asynchronous job completion notifications of the Hadoop client API. Optionally, the user can implement the finalize operator as a Pig script [25] instead of a MapReduce program.

Setup. We perform all experiments in an eight-node cluster running Hadoop 0.20.1 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 (RAID 0). 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 compare SkewReduce to various uniform data partitioning algorithms. We use the LPT scheduling algorithm for the SkewReduce optimizer. Uniform alternatives cannot use this approach because they do not have any way to estimate how long different tasks will take to process the same amount of data.

Default Optimization Parameters. SkewReduce's optimizer assumes a MapReduce job scheduling overhead (Δ) of 10 seconds [28]. Unless indicated otherwise, experiments use a sample size of 1%. The default cost function builds a 3D equi-width histogram of the data. Each bucket covers a range equal to the friend distance threshold along each dimension. The cost is computed as the sum of squared frequencies for all buckets. Each frequency is scaled back by the sample size (*e.g.*, for a 1% sample, all bucket frequencies are multiplied by 100) before squaring. The intuition behind this cost model is this: To identify a cluster, the friends-of-friends algorithm starts with a point and recursively finds friends and friends-of-friends using the KD-tree until no new friends can be added. This process yields quadratic runtime in dense regions, since every point is a friend of every other point. We obtain the conversion constants ρ_p, ρ_m (shown in Table 3) by executing 10 micro-benchmark runs of the analysis task over a 1% data sample.

4.1   Overall SkewReduce Performance

In this section, we present experimental results that answer the following question: **Q: Does SkewReduce improve task completion times in the presence of computational skew compared to uniform data partitioning? Is the improvement significant?**

To answer this question, we measure the total runtime of the plans generated by SkewReduce for both datasets. We compare them against the runtimes of a manually crafted plan called *Manual* and plans with various uniform partitioning granularities: *Coarse*, *Fine*, *Finer*, and *Finest*. All plans are generated from the same 1% data sample. *Coarse* mimics Hadoop, which assigns a Map task to each HDFS chunk. Similarly, *Coarse* partitions the data into fragments that each contains the same number of data points. It does so by repeatedly splitting the region to bisect the data, one axis at a time in a round robin fashion, just like a KD-tree
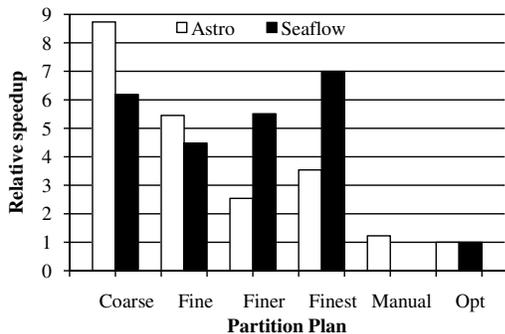
Completion time (hours for Astro, minutes for Seaflow)

Dataset	Coarse	Fine	Finer	Finest	Manual	Opt
Astro	14.1	8.8	4.1	5.7	2.0	1.6
Seaflow	87.2	63.1	77.7	98.7	-	14.1

Figure 3: Relative speed of different partitioning strategies compared with the optimized plan (Opt). The table shows the actual completion time for each strategy (units are hours for Astro and minutes for Seaflow). Manual plan is shown only for the Astro dataset. Overall, SkewReduce's optimization significantly improves the completion time.

using a Recursive Coordinate Bisection (RCB) scheme [3]. *Coarse* stops splitting when the size of each partition is less than 128 MB. *Fine* stops splitting only when each partition is 16 MB. *Finer* and *Finest* partition the *Fine* partitions further until each partition holds 4 MB and 2 MB, respectively. Finally, we prepared the *Manual* plan by tweaking the *Fine* plan based on the execution results: we merged partitions experiencing no skew and split slow partitions further. We prepared a manual plan only for the Astro dataset due to the tedious nature of this task. Figure 3 shows the relative completion times of all plans compared to the optimized plan, labeled as *Opt*. We also report the actual completion time of each plan in the accompanying table.

The results from both datasets illustrate that fine-grained uniform splitting only improves performance up to a certain point before runtimes increase again due to overheads associated with scheduling and executing so many partitions. The SkewReduce optimizer's plan, however, guided by user-defined cost functions, is more than twice as fast as the best uniform plan. For the Astro dataset, SkewReduce improves the completion time of the clustering task by a factor of more than 8 compared with *Coarse*, which is the strategy equivalent to the default approach in MapReduce-type systems. SkewReduce's performance is even a bit better than the Manual plan. For the Seaflow dataset, the *Opt* runtime is a factor of 3 better than *Fine* and a factor of 6 better than *Coarse*.

Overall, SkewReduce can thus significantly improve the runtime of this analysis task.

4.2 Cost Model Fidelity

In this section, we start to study the parameters that affect SkewReduce's performance. In particular, we answer the following question: **Q: How important is the fidelity of the cost model for SkewReduce's optimization?**

The fidelity of a cost function is related to the Fidelity property defined in Section 3.2. Given two partitions R and S, if a cost function C_A is more likely to produce cost estimates that reflect the correct execution time order than

Table 4: Cost functions for evaluation

Function	Fidelity	Description
Data Size	Low	The number of data items
Histogram 1D	Medium	Sum of squared freq., 10 buckets
Histogram 3D	High	Sum of squared freq., all buckets

Estimated runtime and percent error for Astro data (hour)

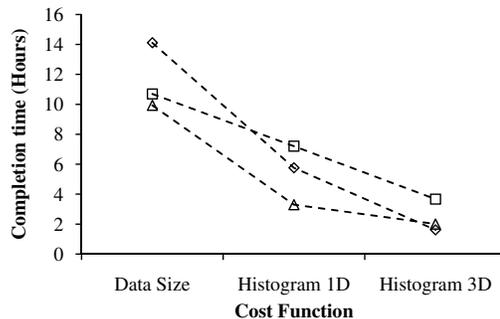| Sample	Data Size	Histogram 1D	Histogram 3D
1	6.64 (-53.0%)	7.59 (31.8%)	2.84 (76.2%)
2	6.64 (-37.9%)	7.21 (8.20%)	2.94 (-20.0%)
3	6.64 (-33.2%)	3.30 (135%)	2.73 (36.2%)

Figure 4: Completion times of plans for the Astro dataset using different cost functions. The x-axis is cost functions in increasing order of fidelity and y-axis is completion time in hours. Each line represents the real runtimes of the plans derived from the same sample. The table shows estimated runtimes and percent errors with respect to completion times.

a cost function C_B, we say that C_A is a higher-fidelity cost function than C_B.

To answer this question, we compare the performance of SkewReduce using different cost functions and find that plan quality is sensitive to the fidelity of the cost function to the actual algorithm. In this experiment, we compare three cost functions: the 3D histogram function described previously, a simpler but less faithful 1D histogram, and simply the data size as a cost proxy. Table 4 summarizes the three functions.

For each dataset, we prepared three independent 1% data samples. We then ran the SkewReduce optimizer with the different cost functions for each data sample and compared the execution times of the resulting partition plans.

Figure 4 shows the result for the Astro dataset. The x-axis shows the cost function in order of increasing fidelity. Each dashed line represents the execution time of plans generated from the same sample. Overall, the fidelity of the cost function significantly affects the total runtime.

The Data Size cost function leads to a plan essentially equivalent to *Coarse* in Figure 3 thus there is no big improvement. Interestingly, the *Histogram 1D* function yields a runtime close to the second best *Finest* from Figure 3. Hence, even a cost function with limited fidelity to the actual algorithm can help compare to not using SkewReduce. The most faithful *Histogram 3D* function yields the best plan and significantly improves the execution time compared to the least faithful cost function, Data Size. Across all three samples, the higher fidelity cost function yields the better plan. The estimated runtimes and percent errors with respect to the real execution times are shown in the accompanying table in Figure 4. The expected runtimes significantly deviate from the actual runtimes because of discrepancies
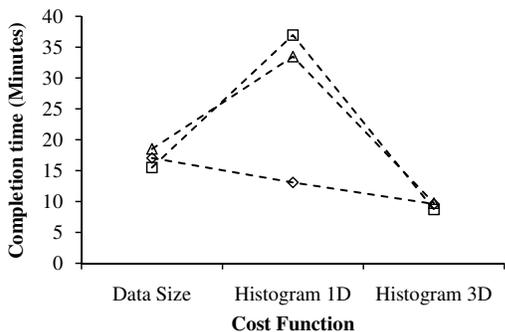
Estimated runtime and percent error for Seaflow data (minutes)

Sample	Data Size	Histogram 1D	Histogram 3D
1	3.84 (-77.5%)	6.71 (-48.8%)	1.41 (-85.3%)
2	3.84 (-75.2%)	6.71 (-81.8%)	1.41 (-83.9%)
3	3.84 (-79.2%)	6.72 (-79.9%)	1.41 (-85.5%)

Figure 5: Completion time of plans for Seaflow dataset using different cost functions. Note that the y-axis is in minutes. Each line represents the real runtimes of the plans derived from the same sample. The table shows estimated runtimes and percent errors with respect to completion times. Histogram 1D performs badly because it significantly overestimates the cost of a partition.

between the cost function and the real algorithm, as well as the sample and the real data.

Figure 5 shows the results of the same experiment on the Seaflow dataset. Here, the Histogram 1D cost function yields a worse plan than the Data Size cost function in two out of three cases, while Histogram 3D consistently produces the best plans.

The anomaly is due to characteristics of the Seaflow dataset. Unlike the Astro dataset, all domains of the Seaflow dataset are 16-bit unsigned integers with significant repetition (*e.g.*, values near 0 along the x-axis in Figure 1). Histogram 1D tends to overestimate the cost of a partition compared with Histogram 3D because it approximates the cost using a fixed number of buckets (Table 4). With small domains and many repeated values in the dataset, the error becomes significant compromising fidelity and eventually affecting the optimization. While the resulting plans are worse than those produced by Data Size, the execution time is only half of the best uniform partitioning strategy (*Fine*) in Figure 3. Thus, a less faithful cost function may not produce a good plan consistently but still yields better results than a uniform strategy.

Finally, the Data Size cost function significantly improves the runtime compared with the *Coarse* and *Fine* plans from Figure 3, while it seems that the two should be equivalent. The difference is attributable to SkewReduce's ability to select the partitioning axis that yields the best splits.

In summary, a high fidelity cost function benefits SkewReduce's optimization and improves the runtime significantly. However, even approximate cost functions can yield better plans than ignoring computation costs and splitting only based on dataset sizes.

4.3 Sample Size

In this section, we examine the effects of the sample size on SkewReduce's performance and answer the following question: **Q: What sample sizes are required for SkewReduce to generate good plans?**
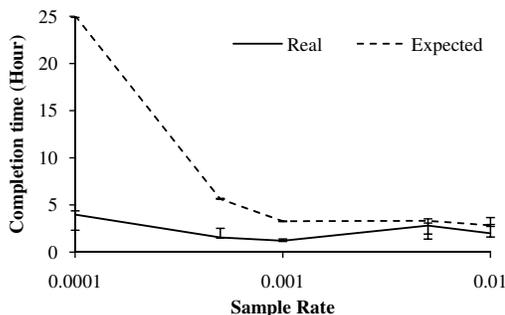

Figure 6: Completion time for the Astro dataset with varying sample rates. Error bars show the minimum and maximum values obtained for each sampling rate.
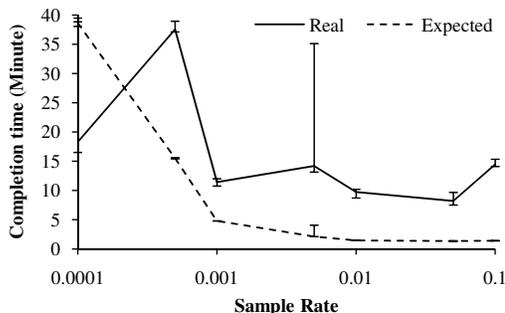

Figure 7: Completion time for the Seaflow dataset with varying sample rates. Error bars show the minimum and maximum values obtained for each sampling rate.

SkewReduce's optimization is based solely on the sample, and an unrepresentative sample may affect the accuracy of the optimizer's cost estimates. To measure the effect on accuracy, we prepared three independent samples with varying sampling rates, then generated and executed an optimized plan using the best cost function, Histogram 3D.

Figures 6 and 7 show the results from the Astro and Seaflow datasets, respectively. In both figures, the optimizer's cost estimates improve as the sample size increases but the convergence is not smooth. Surprisingly, the estimated runtime of the Astro dataset does not fluctuate as much as that of the Seaflow dataset even at lower sampling rates. The reason is that the extreme density variations in the Astro dataset that drive the performance are still captured even in a small sample. In contrast, the Seaflow sample may or may not exhibit significant skew. We also find that a larger sample does not always guarantee a better plan. In Figure 7, the sampling rate of 10% does not yield a better plan than a 5% sampling rate. The conclusion is that the quality of optimization may vary subject to the representativeness of the sample. Interestingly, the runtime of this suboptimal plan is still a factor of 2 improvement compared to the plans based on uniform partitioning as shown in Figure 3.

4.4 SkewReduce Overhead

We finally study SkewReduce's overhead and answer the following question. **Q:** *How long does SkewReduce's optimization take compared with the time to process the query?*

Figure 8 shows the runtime of the prototype optimizer using the Data Size and the Histogram 3D cost functions for each dataset. At a 1% sampling rate, the optimization takes
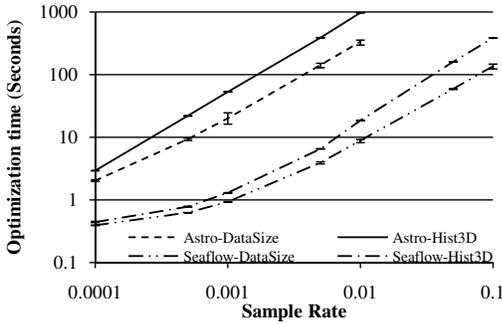
Figure 8: Optimization time with varying sample rates and cost functions. With a 0.01 sample rate, there are 590K samples for the Seaflow dataset and 9.1M samples for Astro.
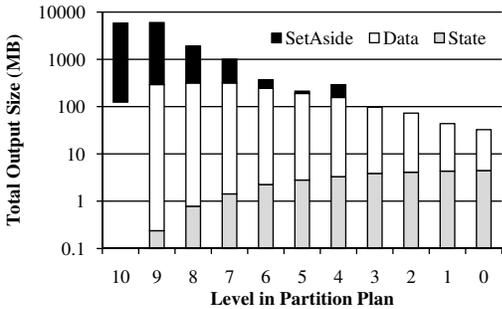

Figure 9: Aggregate output data size produced at each level of an optimized partition plan for the Astro dataset. The x-axis is the level in the partition tree with level 10 representing the leaves. The y-axis is in log scale. Data and State together form the Feature object (F). Data corresponds to boundary particles with cluster ids and State holds the cluster mappings found so far. Overall, a significant amount of data is set-aside and only a small amount of data is passed to merge.

18 seconds using 594K samples from the Seaflow dataset and 15 minutes using 9.1 M samples from the Astro dataset. Considering that the prototype is not parallelized and does not manage memory in any sophisticated way, the runtime is still a small fraction of the actual runtime of the algorithm for each dataset. With an efficient parallel implementation, the SkewReduce optimizer could potentially run with a more complex cost function or use multiple samples to produce a better plan.

4.5 Data Volume between Operations

Lastly, we briefly consider the performance implications of SkewReduce's capability to set-aside some data during the process and merge steps. The amount of data transferred between stages is known to be a bottleneck of MapReduce [12]. In SkewReduce, however, this is more than a bottleneck because a flood of data between levels eventually exceeds the memory bounds of merge. In this section, we answer the following question. **Q:** *Does SkewReduce feed too much data to merge? How much data is set aside?*

In Figure 9, we analyze the total amount of data generated at each level of the partition tree during one execution of the clustering algorithm on the Astro dataset. Overall, a total of 13.7 GB of data was set aside by process, which corresponds to almost 99% of the input data. The total

data passed to all merge operators is only 2 GB and the top-level merge has received the most amount of data (39 MB), however, this is only one third of the 128 MB chunk size. Thus, through its API, SkewReduce guides application writers toward an efficient implementation of their feature extraction applications, setting aside significant amounts of data when possible, reducing traffic between operators, and helping to satisfy the per node memory requirement. We acknowledge that the amount of data that is set aside data will vary depending on the dataset and the application.

5. RELATED WORK

Effective handling of skew is an important problem in any parallel system because improper skew handling can counter all the benefits of parallel processing [10].

In parallel database research, four types of data skew have been identified by Wolf *et al.* [35], and extensively researched by many groups, especially, in the context of the Join operation [11, 18, 35, 36, 39, 38]. Shatdal *et al.* investigated skew problems in aggregation algorithms [30]. Recent adaptive query processing research also mostly focuses on relational operators [7]. SkewReduce approaches the same problem from a different angle. Instead of focusing on specialized implementations of an operator, SkewReduce requests that users provide cost models for their non-relational algorithms and it performs cost-based static partitioning optimization.

Scientific simulation communities have long studied load imbalance problems in parallel systems. Just as in parallel database systems, there exist many mature infrastructures to run parallel simulations in a scalable manner [8, 24, 27]. The primary technique for attacking skew is adaptively repartitioning (or regridding) the data by periodically monitoring the application runtime statistics or through explicit error measures of the parallel simulation [9]. The SkewReduce optimization resembles these programs, but uses sampling to optimize statically. Also, the partitioning is strictly guided by the user cost functions rather than errors in the simulation. Several cosmological simulations partition the workload based on gravitational potential and construct a tree to balance parallel spatial index lookup as well as computation [33]. SkewReduce shares the same spirit with those simulations but provides a generic, domain-independent framework to statically optimize the partition plan using user-defined cost functions and execute it in a shared-nothing cluster.

MapReduce and similar large scale data analysis platforms handle machine skew using speculative execution [6, 19, 15]. Speculative execution simply re-launches slow tasks on multiple different machines and takes the result of the first replica to complete. Speculative execution is effective in heterogeneous computing environments or when machines fail. However, it is not effective against data skew because rerunning the skewed data partition even on a faster machine can still yield a very high response time. Lin analyzed such impact of data skew of a MapReduce program [22]. Handling data skew in these systems is, in general, at a relatively early stage. Pig supports skewed join as proposed by DeWitt *et al.* [11] in its latest 0.5.0 release. To the best of our knowledge, this is the only effort to handle data skew problems in MapReduce-based systems. Qiu *et al.* implemented three applications for bioinformatics using cloud technologies and reported their experience and measurement results [29]. Although they also found skew problems in two

applications, they discussed a potential solution rather than tackling the problem. SkewReduce is aiming to offer a more general skew-resistant solution to applications running on these types of platforms.

6. CONCLUSION

In this paper, we presented SkewReduce, a new API for feature-extracting scientific applications and an implementation that leads to an efficient execution of these applications. At the heart of SkewReduce is a static optimizer that leverages user-supplied cost functions to generate parallel processing plans, which significantly reduce the impact of computational skew inherent in many of these applications. Through experiments on two real datasets, we showed that SkewReduce can improve application run times by a factor of two to eight and takes only seconds to minutes to run, facilitating offline resource planning. In future work, we plan to extend SkewReduce to react to unanticipated skew at runtime by dynamically repartitioning the data and computation as necessary.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://www.amazon.com/gp/browse.html?node=201590011.
[2] Oceanic remote chemical analyzer (ORCA). http://armbrustlab.ocean.washington.edu/.
[3] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *Computers, IEEE Transactions on*, C-36(5), 1987.
[4] J. Boulos and K. Ono. Cost estimation of user-defined methods in object-relational database systems. *SIGMOD Record*, 28(3), 1999.
[5] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White. The evolution of large-scale structure in a universe dominated by cold dark matter. *Astroph. J.*, 292:371–394, May 1985.
[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.
[7] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):139, 2007.
[8] K. Devine, E. Boman, R. Heapby, B. Hendrickson, and C. Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engg.*, 4(2), 2002.
[9] K. Devine, E. Boman, and G. Karypis. *Partitioning and load balancing for emerging parallel applications and architectures*, chapter 6. 2006.
[10] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *CACM*, 35(6), 1992.

[11] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proc. of the 18th VLDB Conf.*, 1992.
[12] DeWitt et. al. Clustera: an integrated computation and data management system. In *Proc. of the 34th VLDB Conf.*, 2008.
[13] J. M. Gelb and E. Bertschinger. Cold dark matter. 1: The formation of dark halos. *Astroph. J.*, 436:467–490, Dec. 1994.
[14] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.
[15] Hadoop. http://hadoop.apache.org/.
[16] Hive. http://hadoop.apache.org/hive/.
[17] B. Howe, D. Maier, and L. Bright. Smoothing the roi curve for scientific data management applications. In *Proc. of the Third CIDR Conf.*, 2007.
[18] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. of the 17th VLDB Conf.*, 1991.
[19] 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.*, 2007.
[20] S. R. Knollmann and A. Knebe. AHF: Amiga's Halo Finder. *Astroph. J. Suppl.*, 182:608–624, June 2009.
[21] Kwon et. al. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. Technical Report UW-CSE-09-06-01, Dept. of Comp. Sci., Univ. of Washington, 2009.
[22] 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*, volume i, 2009.
[23] R. P. Mount. The office of science data-management challenge. Technical report, Department of Energy, 2004.
[24] L. Oliker and R. Biswas. Plum: parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 52(2), 1998.
[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of the SIGMOD Conf.*, 2008.
[26] Oracle. http://www.oracle.com/database/.
[27] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate: Enabling Autonomic Applications on the Grid. *Cluster Computing*, 9(2), 2006.
[28] Pavlo et. al. A comparison of approaches to large-scale data analysis. In *Proc. of the SIGMOD Conf.*, 2009.
[29] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon. Cloud technologies for bioinformatics applications. In *MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10, 2009.
[30] A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *Proc. of the SIGMOD Conf.*, 1995.
[31] Springel et. al. Simulations of the formation, evolution and clustering of galaxies and quasars. *NATURE*, 435:629–636, June 2005.
[32] Sql server. http://www.microsoft.com/sqlserver/.
[33] J. G. Stadel. *Cosmological N-body simulations and their analysis*. PhD thesis, University of Washington, 2001.
[34] A. Szalay and J. Gray. 2020 computing: Science in an exponential world. *Nature*, 440:413–414, 2006.
[35] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *Proc. of the 17th VLDB Conf.*, 1991.
[36] R. S. Wei Li, D Gao. Skew handling techniques in sort-merge join. In *Proc. of the SIGMOD Conf.*, 2002.
[37] D. H. Weinberg, L. Hernquist, and N. Katz. Photoionization, Numerical Resolution, and Galaxy Formation. *Astroph. J.*, 477:8–+, Mar. 1997.
[38] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel DBMS. In *VLDB*, 2009.
[39] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *Proc. of the SIGMOD Conf.*, 2008.
[40] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proc. of the 22nd SOSP Symp.*, 2009.
[41] Yu et. al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of the 8th OSDI Symp.*, 2008.