Starfish: A Self-tuning System for Big Data Analytics

Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong,
Fatma Bilgen Cetin, Shivnath Babu
Department of Computer Science
Duke University

ABSTRACT

Timely and cost-effective analytics over "Big Data" is now a key ingredient for success in many businesses, scientific and engineering disciplines, and government endeavors. The Hadoop software stack—which consists of an extensible MapReduce execution engine, pluggable distributed storage engines, and a range of procedural to declarative interfaces—is a popular choice for big data analytics. Most practitioners of big data analytics—like computational scientists, systems researchers, and business analysts—lack the expertise to tune the system to get good performance. Unfortunately, Hadoop's performance out of the box leaves much to be desired, leading to suboptimal use of resources, time, and money (in pay-as-you-go clouds). We introduce Starfish, a self-tuning system for big data analytics. Starfish builds on Hadoop while adapting to user needs and system workloads to provide good performance automatically, without any need for users to understand and manipulate the many tuning knobs in Hadoop. While Starfish's system architecture is guided by work on self-tuning database systems, we discuss how new analysis practices over big data pose new challenges; leading us to different design choices in Starfish.

1. INTRODUCTION

Timely and cost-effective analytics over "Big Data" has emerged as a key ingredient for success in many businesses, scientific and engineering disciplines, and government endeavors [6]. Web search engines and social networks capture and analyze every user action on their sites to improve site design, spam and fraud detection, and advertising opportunities. Powerful telescopes in astronomy, genome sequencers in biology, and particle accelerators in physics are putting massive amounts of data into the hands of scientists. Key scientific breakthroughs are expected to come from computational analysis of such data. Many basic and applied science disciplines now have computational subareas, e.g., computational biology, computational economics, and computational journalism.

Cohen et al. recently coined the acronym *MAD*—for *Magnetism*, *Agility*, and *Depth*—to express the features that users expect from a system for big data analytics [6].

Magnetism: A magnetic system attracts all sources of data irrespective of issues like possible presence of outliers, unknown schema or lack of structure, and missing values that keep many useful data sources out of conventional data warehouses.

Agility: An agile system adapts in sync with rapid data evolution.

Depth: A deep system supports analytics needs that go far beyond conventional rollups and drilldowns to complex statistical and machine-learning analysis.

Hadoop is a MAD system that is becoming popular for big data analytics. An entire ecosystem of tools is being developed around Hadoop. Figure 1 shows a summary of the Hadoop software stack in wide use today. Hadoop itself has two primary components: a MapReduce execution engine and a distributed filesystem. While the Hadoop Distributed FileSystem (HDFS) is used predominantly as the distributed filesystem in Hadoop, other filesystems like Amazon S3 are also supported. Analytics with Hadoop involves loading data as files into the distributed filesystem, and then running parallel MapReduce computations on the data.

A combination of factors contributes to Hadoop's MADness. First, copying files into the distributed filesystem is all it takes to get data into Hadoop. Second, the MapReduce methodology is to interpret data (lazily) at processing time, and not (eagerly) at loading time. These two factors contribute to Hadoop's magnetism and agility. Third, MapReduce computations in Hadoop can be expressed directly in general-purpose programming languages like Java or Python, domain-specific languages like R, or generated automatically from SQL-like declarative languages like HiveQL and Pig Latin. This coverage of the language spectrum makes Hadoop well suited for deep analytics. Finally, an unheralded aspect of Hadoop is its extensibility, i.e., the ease with which many of Hadoop's core components like the scheduler, storage subsystem, input/output data formats, data partitioner, compression algorithms, caching layer, and monitoring can be customized or replaced.

Getting desired performance from a MAD system can be a non-trivial exercise. The practitioners of big data analytics like data analysts, computational scientists, and systems researchers usually lack the expertise to tune system internals. Such users would rather use a system that can tune itself and provide good performance automatically. Unfortunately, the same properties that make Hadoop MAD pose new challenges in the path to self-tuning:

- *Data opacity until processing:* The magnetism and agility that comes with interpreting data only at processing time poses the difficulty that even the schema may be unknown until the point when an analysis job has to be run on the data.
- *File-based processing:* Input data for a MapReduce job may be stored as few large files, millions of small files, or anything in between. Such uncontrolled data layouts are a marked contrast to the carefully-planned layouts in database systems.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

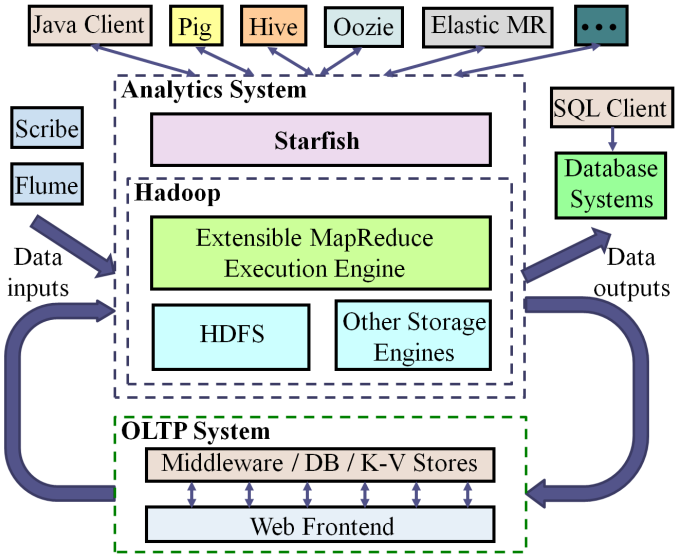

Figure 1: Starfish in the Hadoop ecosystem

- *Heavy use of programming languages:* A sizable fraction of MapReduce programs will continue to be written in programming languages like Java for performance reasons, or in languages like Python or R that a user is most comfortable with while prototyping new analysis tasks.

Traditional data warehouses are kept nonMAD by its administrators because it is easier to meet performance requirements in tightly controlled environments; a luxury we cannot afford any more [6]. To further complicate matters, three more features in addition to MAD are becoming important in analytics systems: *Data-lifecycle-awareness*, *Elasticity*, and *Robustness*. A system with all six features would be *MADDER* than current analytics systems.

Data-lifecycle-awareness: A data-lifecycle-aware system goes beyond query execution to optimize the movement, storage, and processing of big data during its entire lifecycle. The intelligence embedded in many Web sites like LinkedIn and Yahoo!—e.g., recommendation of new friends or news articles of potential interest, selection and placement of advertisements—is driven by computation-intensive analytics. A number of companies today use Hadoop for such analytics [12]. The input data for the analytics comes from dozens of different sources on user-facing systems like key-value stores, databases, and logging services (Figure 1). The data has to be moved for processing to the analytics system. After processing, the results are loaded back in near real-time to user-facing systems. Terabytes of data may go through this *cycle* per day [12]. In such settings, data-lifecycle-awareness is needed to: (i) eliminate indiscriminate data copying that causes bloated storage needs (as high as 20x if multiple departments in the company make their own copy of the data for analysis [17]); and (ii) reduce resource overheads and realize performance gains due to reuse of intermediate data or learned metadata in workflows that are part of the cycle [8].

Elasticity: An elastic system adjusts its resource usage and operational costs to the workload and user requirements. Services like Amazon Elastic MapReduce have created a market for pay-as-you-go analytics hosted on the cloud. Elastic MapReduce provisions and releases Hadoop clusters on demand, sparing users the hassle of cluster setup and maintenance.

Robustness: A robust system continues to provide service, possibly with graceful degradation, in the face of undesired events like hardware failures, software bugs [12], and data corruption.

1.1 Starfish: MADDER and Self-Tuning Hadoop

Hadoop has the core mechanisms to be MADDER than existing analytics systems. However, the use of most of these mechanisms has to be managed manually. Take elasticity as an example. Hadoop supports dynamic node addition as well as decommissioning of failed or surplus nodes. However, these mechanisms do not magically make Hadoop elastic because of the lack of control modules to decide (a) when to add new nodes or to drop surplus nodes, and (b) when and how to rebalance the data layout in this process.

Starfish is a MADDER and self-tuning system for analytics on big data. An important design decision we made is to build Starfish on the Hadoop stack as shown in Figure 1. (That is not to say that Starfish uses Hadoop as is.) Hadoop, as observed earlier, has useful primitives to help meet the new requirements of big data analytics. In addition, Hadoop's adoption by academic, government, and industrial organizations is growing at a fast pace.

A number of ongoing projects aim to improve Hadoop's peak performance, especially to match the query performance of parallel database systems [1, 7, 10]. Starfish has a different goal. The peak performance a manually-tuned system can achieve is not our primary concern, especially if this performance is for one of the many phases in the data lifecycle. Regular users may rarely see performance close to this peak. Starfish's goal is to enable Hadoop users and applications to get good performance automatically throughout the data lifecycle in analytics; without any need on their part to understand and manipulate the many tuning knobs available.

Section 2 gives an overview of Starfish while Sections 3–5 describe its components. The primary focus of this paper is on using experimental results to illustrate the challenges in each component and to motivate Starfish's solution approach.

2. OVERVIEW OF STARFISH

The *workload* that a Hadoop deployment runs can be considered at different levels. At the lowest level, Hadoop runs MapReduce *jobs*. A job can be generated directly from a program written in a programming language like Java or Python, or generated from a query in a higher-level language like HiveQL or Pig Latin [15], or submitted as part of a MapReduce job *workflow* by systems like Azkaban, Cascading, Elastic MapReduce, and Oozie. The execution plan generated for a HiveQL or Pig Latin query is usually a workflow. Workflows may be ad-hoc, time-driven (e.g., run every

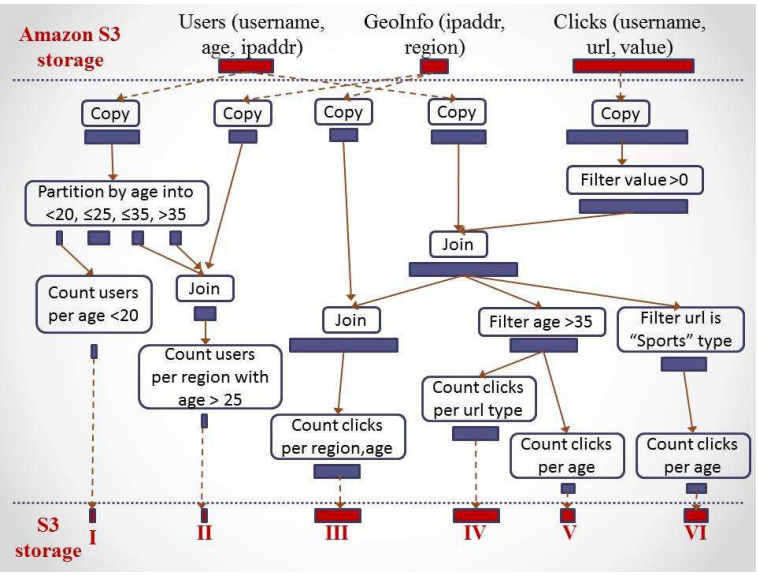

Figure 2: Example analytics workload to be run on Amazon Elastic MapReduce

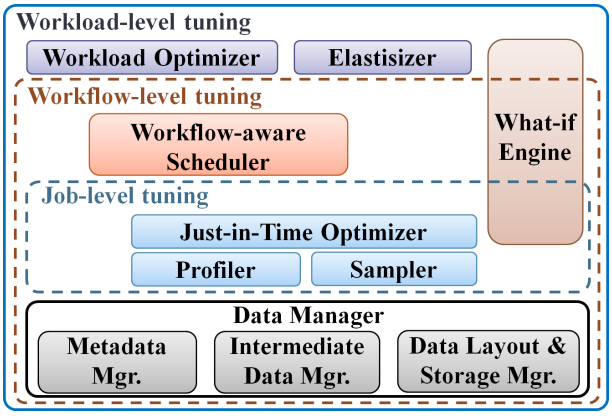

Figure 3: Components in the Starfish architecture

hour), or data-driven. Yahoo! uses data-driven workflows to generate a reconfigured preference model and an updated home-page for any user within seven minutes of a home-page click by the user.

Figure 2 is a visual representation of an example workload that a data analyst may want to run on demand or periodically using Amazon Elastic MapReduce. The input data processed by this workload resides as files on Amazon S3. The final results produced by the workload are also output to S3. The input data consists of files that are collected by a personalized Web-site like `my.yahoo.com`.

The example workload in Figure 2 consists of workflows that load the files from S3 as three datasets: Users, GeoInfo, and Clicks. The workflows process these datasets in order to generate six different results I-VI of interest to the analyst. For example, Result I in Figure 2 is a count of all users with age less than 20. For all users with age greater than 25, Result II counts the number of users per geographic region. For each workflow, one or more MapReduce jobs are generated in order to run the workflow on Amazon Elastic MapReduce or on a local Hadoop cluster. For example, notice from Figure 2 that a join of the Users and GeoInfo datasets is needed in order to generate Result II. This logical join operation can be processed using a single MapReduce job.

The tuning challenges present at each level of workload processing led us to the Starfish architecture shown in Figure 3. Broadly, the functionality of the components in this architecture can be categorized into job-level tuning, workflow-level tuning, and workload-level tuning. These components interact to provide Starfish's self-tuning capabilities.

2.1 Job-level Tuning

The behavior of a MapReduce job in Hadoop is controlled by the settings of more than 190 configuration parameters. If the user does not specify parameter settings during job submission, then default values—shipped with the system or specified by the system administrator—are used. Good settings for these parameters depend on job, data, and cluster characteristics. While only a fraction of the parameters can have significant performance impact, browsing through the Hadoop, Hive, and Pig mailing lists reveals that users often run into performance problems caused by lack of knowledge of these parameters.

Consider a user who wants to perform a join of data in the files users.txt and geoinfo.txt, and writes the Pig Latin script:

```
Users = Load 'users.txt' as (username: chararray,
            age: int, ipaddr: chararray)
GeoInfo = Load 'geoinfo.txt' as (ipaddr: chararray,
            region: chararray)
Result = Join Users by ipaddr, GeoInfo by ipaddr
```

The schema as well as properties of the data in the files could have been unknown so far. The system now has to quickly choose the join execution technique—given the limited information available so far, and from among 10+ ways to execute joins in Starfish—as well as the corresponding settings of job configuration parameters.

Starfish's *Just-in-Time Optimizer* addresses unique optimization problems like those above to automatically select efficient execution techniques for MapReduce jobs. "Just-in-time" captures the online nature of decisions forced on the optimizer by Hadoop's MADDER features. The optimizer takes the help of the *Profiler* and the *Sampler*. The Profiler uses a technique called *dynamic instrumentation* to learn performance models, called *job profiles*, for unmodified MapReduce programs written in languages like Java and Python. The Sampler collects statistics efficiently about the input, intermediate, and output *key-value spaces* of a MapReduce job. A unique feature of the Sampler is that it can sample the execution of a MapReduce job in order to enable the Profiler to collect approximate job profiles at a fraction of the full job execution cost.

2.2 Workflow-level Tuning

Workflow execution brings out some critical and unanticipated interactions between the MapReduce task scheduler and the underlying distributed filesystem. Significant performance gains are realized in parallel task scheduling by moving the computation to the data. By implication, the data layout across nodes in the cluster constrains how tasks can be scheduled in a "data-local" fashion. Distributed filesystems have their own policies on how data written to them is laid out. HDFS, for example, always writes the first replica of any block on the same node where the writer (in this case, a map or reduce task) runs. This interaction between data-local scheduling and the distributed filesystem's block placement policies can lead to an *unbalanced* data layout across nodes in the cluster during workflow execution; causing severe performance degradation as we will show in Section 4.

Efficient scheduling of a Hadoop workflow is further complicated by concerns like (a) avoiding cascading reexecution under node failure or data corruption [11], (b) ensuring power proportional computing, and (c) adapting to imbalance in load or cost of energy across geographic regions and time at the datacenter level [16]. Starfish's *Workflow-aware Scheduler* addresses such concerns in conjunction with the *What-if Engine* and the *Data Manager*. This scheduler communicates with, but operates outside, Hadoop's internal task scheduler.

2.3 Workload-level Tuning

Enterprises struggle with higher-level optimization and provisioning questions for Hadoop workloads. Given a workload consisting of a collection of workflows (like Figure 2), Starfish's *Workload Optimizer* generates an equivalent, but optimized, collection of workflows that are handed off to the Workflow-aware Scheduler for execution. Three important categories of optimization opportunities exist at the workload level:

A. *Data-flow sharing*, where a single MapReduce job performs computations for multiple and potentially different logical nodes belonging to the same or different workflows.

B. *Materialization*, where intermediate data in a workflow is stored for later reuse in the same or different workflows. Effective use of materialization has to consider the cost of materialization (both in terms of I/O overhead and storage consumption [8]) and its potential to avoid cascading reexecution of tasks under node failure or data corruption [11].

C. *Reorganization*, where new data layouts (e.g., with partitioning) and storage engines (e.g., key-value stores like HBase and databases like column-stores [1]) are chosen automatically and transparently to store intermediate data so that downstream jobs in the same or different workflows can be executed very efficiently.

While categories A, B, and C are well understood in isolation, applying them in an integrated manner to optimize MapReduce workloads poses new challenges. First, the data output from map tasks and input to reduce tasks in a job is always materialized in Hadoop in order to enable robustness to failures. This data—which today is simply deleted after the job completes—is key-value-based, sorted on the key, and partitioned using externally-specified partitioning functions. This unique form of intermediate data is available almost for free, bringing new dimensions to questions on materialization and reorganization. Second, choices for A, B, and C potentially interact among each other and with scheduling, data layout policies, as well as job configuration parameter settings. The optimizer has to be aware of such interactions.

Hadoop provisioning deals with choices like the number of nodes, node configuration, and network configuration to meet given workload requirements. Historically, such choices arose infrequently and were dealt with by system administrators. Today, users who provision Hadoop clusters on demand using services like Amazon Elastic MapReduce and Hadoop On Demand are required to make provisioning decisions on their own. Starfish's *Elastisizer* automates such decisions. The intelligence in the Elastisizer comes from a search strategy in combination with the What-if Engine that uses a mix of simulation and model-based estimation to answer what-if questions regarding workload performance on a specified cluster configuration. In the longer term, we aim to automate provisioning decisions at the level of multiple virtual and elastic Hadoop clusters hosted on a single shared Hadoop cluster to enable Hadoop *Analytics as a Service*.

2.4 Lastword: Starfish's Language for Workloads and Data

As described in Section 1.1 and illustrated in Figure 1, Starfish is built on the Hadoop stack. Starfish interposes itself between Hadoop and its clients like Pig, Hive, Oozie, and command-line interfaces to submit MapReduce jobs. These Hadoop clients will now submit workloads—which can vary from a single MapReduce job, to a workflow of MapReduce jobs, and to a collection of multiple workflows—expressed in *Lastword*[1] to Starfish. Lastword is Starfish's language to accept as well as to reason about analytics workloads.

Unlike languages like HiveQL, Pig Latin, or Java, Lastword is not a language that humans will have to interface with directly. Higher-level languages like HiveQL and Pig Latin were developed to support a diverse user community—ranging from marketing analysts and sales managers to scientists, statisticians, and systems researchers—depending on their unique analytical needs and preferences. Starfish provides language translators to automatically convert workloads specified in these higher-level languages to Lastword. A common language like Lastword allows Starfish to exploit optimization opportunities among the different workloads that run on the same Hadoop cluster.

A Starfish client submits a workload as a collection of workflows expressed in Lastword. Three types of workflows can be represented in Lastword: (a) physical workflows, which are directed graphs[2] where each node is a MapReduce job representation; (b) logical workflows, which are directed graphs where each node is a logical specification such as a select-project-join-aggregate (SPJA) or a user-defined function for performing operations like partitioning, filtering, aggregation, and transformations; and (c) hybrid workflows, where a node can be of either type.

An important feature of Lastword is its support for expressing metadata along with the tasks for execution. Workflows specified in Lastword can be annotated with metadata at the workflow level or at the node level. Such metadata is either extracted from inputs provided by users or applications, or learned automatically by Starfish. Examples of metadata include scheduling directives (e.g., whether the workflow is ad-hoc, time-driven, or data-driven), data properties (e.g., full or partial schema, samples, and histograms), data layouts (e.g., partitioning, ordering, and collocation), and runtime monitoring information (e.g., execution profiles of map and reduce tasks in a job).

The Lastword language gives Starfish another unique advantage. Note that Starfish is primarily a system for running analytics work-

[1] *La*nguage for *St*arfish *Wor*kloads and *D*ata.

[2] Cycles may be needed to support loops or iterative computations.

	WordCount		TeraSort	
	Rules of Thumb	Based on Job Profile	Rules of Thumb	Based on Job Profile
io.sort.spill.percent	0.80	0.80	0.80	0.80
io.sort.record.percent	0.50	0.05	0.15	0.15
io.sort.mb	200	50	200	200
io.sort.factor	10	10	10	100
mapred.reduce.tasks	27	2	27	400
Running Time (sec)	785	407	891	606

Table 1: Parameter settings from rules of thumb and recommendations from job profiles for WordCount and TeraSort

loads on big data. At the same time, we want Starfish to be usable in environments where workloads are run directly on Hadoop without going through Starfish. Lastword enables Starfish to be used as a recommendation engine in these environments. The full or partial Hadoop workload from such an environment can be expressed in Lastword—we will provide tools to automate this step—and then input to Starfish which is run in a special *recommendation mode*. In this mode, Starfish uses its tuning features to recommend good configurations at the job, workflow, and workload levels; instead of running the workload with these configurations as Starfish would do in its normal usage mode.

3. JUST-IN-TIME JOB OPTIMIZATION

The response surfaces in Figure 4 show the impact of various job configuration parameter settings on the running time of two MapReduce programs in Hadoop. We use WordCount and TeraSort which are simple, yet very representative, MapReduce programs. The default experimental setup used in this paper is a single-rack Hadoop cluster running on 16 Amazon EC2 nodes of the c1.medium type. Each node runs at most 3 map tasks and 2 reduce tasks concurrently. WordCount processes 30GB of data generated using the RandomTextWriter program in Hadoop. TeraSort processes 50GB of data generated using Hadoop's TeraGen program.

Rules of Thumb for Parameter Tuning: The job configuration parameters varied in Figure 4 are *io.sort.mb*, *io.sort.record.percent*, and *mapred.reduce.tasks*. All other parameters are kept constant. Table 1 shows the settings of various parameters for the two jobs based on popular rules of thumb used today [5, 13]. For example, the rules of thumb recommend setting *mapred.reduce.tasks* (the number of reduce tasks in the job) to roughly 0.9 times the total number of reduce slots in the cluster. The rationale is to ensure that all reduce tasks run in one wave while leaving some slots free for reexecuting failed or slow tasks. A more complex rule of thumb sets *io.sort.record.percent* to $\frac{16}{16+avg\_record\_size}$ based on the average size of map output records. The rationale here involves source-code details of Hadoop.

Figure 4 shows that the rule-of-thumb settings gave poor performance. In fact, the rule-of-thumb settings for WordCount gave one of its worst execution times: *io.sort.mb* and *io.sort.record.percent* were set too high. The interaction between these two parameters was very different and more complex for TeraSort as shown in Figure 4(b). A higher setting for *io.sort.mb* leads to better performance for certain settings of the *io.sort.record.percent* parameter, but hurts performance for other settings. The complexity of the surfaces and the failure of rules of thumb highlight the challenges a user faces if asked to tune the parameters herself. Starfish's job-level tuning components—Profiler, Sampler, What-if Engine, and Just-in-Time Optimizer—help automate this process.

Profiling Using Dynamic Instrumentation: The Profiler uses *dynamic instrumentation* to collect run-time monitoring information from unmodified MapReduce programs running on Hadoop. Dynamic instrumentation has become hugely popular over the last few years to understand, debug, and optimize complex systems [4]. The dynamic nature means that there is zero overhead when instrumentation is turned off; an appealing property in production deployments. The current implementation of the Profiler uses BTrace [2], a safe and dynamic tracing tool for the Java platform.

When Hadoop runs a MapReduce job, the Starfish Profiler dynamically instruments selected Java classes in Hadoop to construct a *job profile*. A profile is a concise representation of the job execution that captures information both at the task and subtask levels. The execution of a MapReduce job is broken down into the *Map Phase* and the *Reduce Phase*. Subsequently, the Map Phase is divided into the *Reading*, *Map Processing*, *Spilling*, and *Merging* subphases. The Reduce Phase is divided into the *Shuffling*, *Sorting*, *Reduce Processing*, and *Writing* subphases. Each subphase represents an important part of the job's overall execution in Hadoop.

The job profile exposes three views that capture various aspects of the job's execution:

1. *Timings view:* This view gives the breakdown of how wall-clock time was spent in the various subphases. For example, a map task spends time reading input data, running the user-defined map function, and sorting, spilling, and merging map-output data.

2. *Data-flow view:* This view gives the amount of data processed in terms of bytes and number of records during the various subphases.

3. *Resource-level view:* This view captures the usage trends of CPU, memory, I/O, and network resources during the various subphases of the job's execution. Usage of CPU, I/O, and network resources are captured respectively in terms of the time spent using these resources per byte and per record processed. Memory usage is captured in terms of the memory used by tasks as they run in Hadoop.

We will illustrate the benefits of job profiles and the insights gained from them through a real example. Figure 5 shows the Timings view from the profiles collected for the two configuration parameter settings for WordCount shown in Table 1. We will denote the execution of WordCount using the "Rules of Thumb" settings from Table 1 as Job A; and the execution of WordCount using the "Based on Job Profile" settings as Job B. Note that the same WordCount MapReduce program processing the same input dataset is being run in either case. The WordCount program uses a *Combiner* to perform reduce-style aggregation on the map task side for each spill of the map task's output. Table 1 shows that Job B runs 2x faster than Job A.

Our first observation from Figure 5 is that the map tasks in Job B completed on average much faster compared to the map tasks in Job A; yet the reverse happened to the reduce tasks. Further exploration of the Data-flow and Resource views showed that the Combiner in Job A was processing an extremely large number of records, causing high CPU contention. Hence, all the CPU-intensive operations in Job A's map tasks (executing the user-provided map function, serializing and sorting the map output) were negatively affected. Compared to Job A, the lower settings for *io.sort.mb* and *io.sort.record.percent* in Job B led to more, but individually smaller, map-side spills. Because the Combiner is invoked on these individually smaller map-side spills in Job B, the Combiner caused far less CPU contention in Job B compared to Job A.

On the other hand, the Combiner drastically decreases the amount of intermediate data that is spilled to disk as well as transferred over the network (*shuffled*) from map to reduce tasks. Since the map

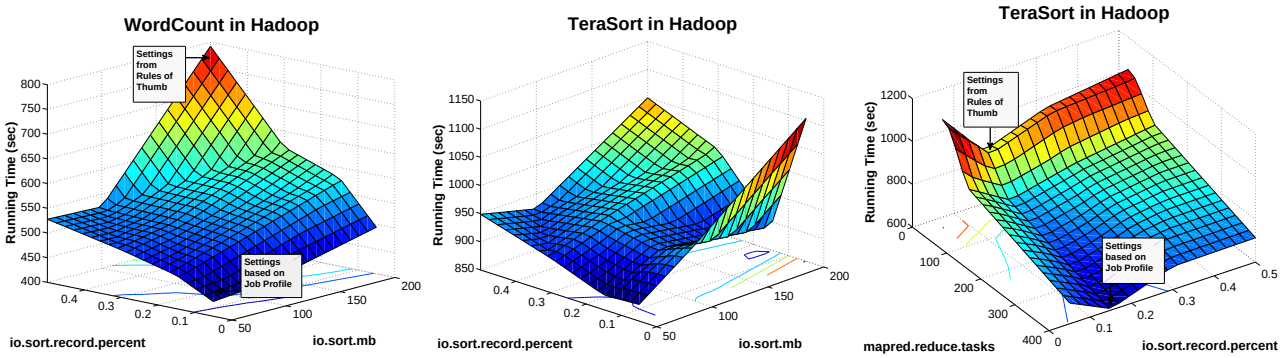

Figure 4: Response surfaces of MapReduce programs in Hadoop: (a) WordCount, with io.sort.mb $\in$ $[50, 200]$ and io.sort.record.percent $\in$ $[0.05, 0.5]$ (b) TeraSort, with io.sort.mb $\in$ $[50, 200]$ and io.sort.record.percent $\in$ $[0.05, 0.5]$ (c) TeraSort, with io.sort.record.percent $\in$ $[0.05, 0.5]$ and mapred.reduce.tasks $\in$ $[27, 400]$

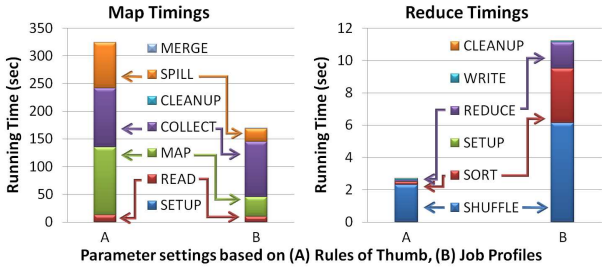

Figure 5: Map and reduce time breakdown for two WordCount jobs run with different settings of job configuration parameters

tasks in Job B processed smaller spills, the data reduction gains from the Combiner were also smaller; leading to larger amounts of data being shuffled and processed by the reducers. However, the additional local I/O and network transfer costs in Job B were dwarfed by the reduction in CPU costs.

Effectively, the more balanced usage of CPU, I/O, and network resources in the map tasks of Job B improved the overall performance of the map tasks significantly compared to Job A. Overall, the benefit gained by the map tasks in Job B outweighed by far the loss incurred by the reduce tasks; leading to the 2x better performance of Job B compared to the performance of Job A.

Predicting Job Performance in Hadoop: The job profile helps in understanding the job behavior as well as in diagnosing bottlenecks during job execution for the parameter settings used. More importantly, given a new setting S of the configuration parameters, the What-if Engine can use the job profile and a set of models that we developed to estimate the new profile if the job were to be run using S. This what-if capability is utilized by the Just-in-Time Optimizer in order to recommend good parameter settings.

The What-if Engine is given four inputs when asked to predict the performance of a MapReduce job J:

1. The job profile generated for J by the Profiler. The profile may be available from a previous execution of J. Otherwise, the Profiler can work in conjunction with Starfish's Sampler to generate an approximate job profile efficiently. Figure 6 considers approximate job profiles later in this section.

2. The new setting S of the job configuration parameters using which Job J will be run.

3. The size, layout, and compression information of the input dataset on which Job J will be run. Note that this input dataset can be different from the dataset used while generating the job profile.

4. The cluster setup and resource allocation that will be used to run Job J. This information includes the number of nodes and network topology of the cluster, the number of map and reduce task slots per node, and the memory available for each task execution.

The What-if Engine uses a set of *performance models* for predicting (a) the flow of data going through each subphase in the job's execution, and (b) the time spent in each subphase. The What-if Engine then produces a *virtual job profile* by combining the predicted information in accordance with the cluster setup and resource allocation that will be used to run the job. The virtual job profile contains the predicted Timings and Data-flow views of the job when run with the new parameter settings. The purpose of the virtual profile is to provide the user with more insights on how the job will behave when using the new parameter settings, as well as to expand the use of the What-if Engine towards answering hypothetical questions at the workflow and workload levels.

Towards Cost-Based Optimization: Table 1 shows the parameter settings for WordCount and TeraSort recommended by an initial implementation of the Just-in-Time Optimizer. The What-if Engine used the respective job profiles collected from running the jobs using the rules-of-thumb settings. WordCount runs almost twice as fast at the recommended setting. As we saw earlier, while the Combiner reduced the amount of intermediate data drastically, it was making the map execution heavily CPU-bound and slow. The configuration setting recommended by the optimizer—with lower *io.sort.mb* and *io.sort.record.percent*—made the map tasks significantly faster. This speedup outweighed the lowered effectiveness of the Combiner that caused more intermediate data to be shuffled and processed by the reduce tasks.

These experiments illustrate the usefulness of the Just-in-Time Optimizer. One of the main challenges that we are addressing is in developing an efficient strategy to search through the high-dimensional space of parameter settings. A related challenge is in generating job profiles with minimal overhead. Figure 6 shows the tradeoff between the profiling overhead (in terms of job slowdown) and the average relative error in the job profile views when profiling is limited to a fraction of the tasks in WordCount. The results are promising but show room for improvement.

4. WORKFLOW-AWARE SCHEDULING

Cause and Effect of Unbalanced Data Layouts: Section 2.2 mentioned how interactions between the task scheduler and the policies employed by the distributed filesystem can lead to unbalanced data layouts. Figure 7 shows how even the execution of a single large

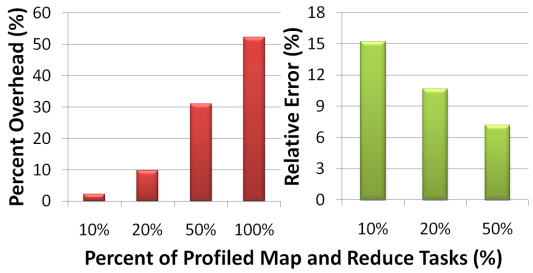

Figure 6: (a) Relative job slowdown, and (b) relative error in the approximate views generated as the percentage of profiled tasks in a job is varied

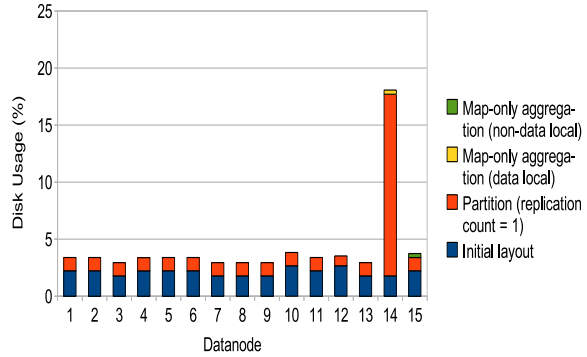

Figure 7: Unbalanced data layout

job can cause an unbalanced layout in Hadoop. We ran a partitioning MapReduce job (similar to "Partition by age" shown in Figure 2) that partitions a 100GB TPC-H Lineitem table into four partitions relevant to downstream workflow nodes. The data properties are such that one partition is much larger than the others. All the partitions are replicated once as done by default for intermediate workflow data in systems like Pig [11]. HDFS ends up placing all blocks for the large partition on the node (Datanode 14) where the reduce task generating this partition runs.

A number of other causes can lead to unbalanced data layouts rapidly or over time: (a) skewed data, (b) scheduling of tasks in a data-layout-unaware manner as done by the Hadoop schedulers available today, and (c) addition or dropping of nodes without running costly data rebalancing operations. (HDFS does not automatically move existing data when new nodes are added.) Unbalanced data layouts are a serious problem in big data analytics because they are prominent causes of task failure (due to insufficient free disk space for intermediate map outputs or reduce inputs) and performance degradation. We observed a more than 2x slowdown for a sort job running on the unbalanced layout in Figure 7 compared to a balanced layout.

Unbalanced data layouts cause a dilemma for data-locality-aware schedulers (i.e., schedulers that aim to move computation to the data). Exploiting data locality can have two undesirable consequences in this context: performance degradation due to reduced parallelism, and worse, making the data layout further unbalanced because new outputs will go to the over-utilized nodes. Figure 7 also shows how running a map-only aggregation on the large partition leads to the aggregation output being written to the over-utilized Datanode 14. The aggregation output was small. A larger output could have made the imbalance much worse. On the other hand, non-data-local scheduling (i.e., moving data to the computation) incurs the overhead of data movement. A useful new feature in Hadoop will be to piggyback on such data movements to rebalance the data layout.

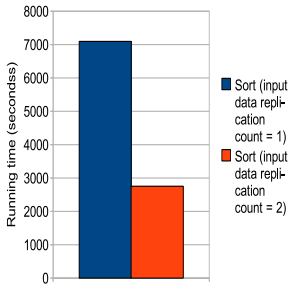

Figure 8: Sort running time on the partitions

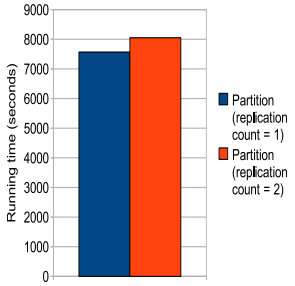

Figure 9: Partition creation time

We ran the same partitioning job with a replication factor of two for the partitions. For our single-rack cluster, HDFS places the second replica of each block of the partitions on a randomly-chosen node. The overall layout is still unbalanced, but the time to sort the partitions improved significantly because the second copy of the data is spread out over the cluster (Figure 8). Interestingly, as shown in Figure 9, the overhead of creating a second replica is very small on our cluster (which will change if the network becomes the bottleneck [11]).

Aside from ensuring that the data layout is balanced, other choices are available such as collocating two or more datasets. Consider a workflow consisting of three jobs. The first two jobs partition two separate datasets R and S (e.g., Users and GeoInfo from Figure 2) using the same partitioning function into n partitions each. The third job, whose input consists of the outputs of the first two jobs, performs an equi-join of the respective partitions from R and S. HDFS does not provide the ability to collocate the joining partitions from R and S; so a join job run in Hadoop will have to do non-data-local reads for one of its inputs.

We implemented a new block placement policy in HDFS that enables collocation of two or more datasets. (As an excellent example of Hadoop's extensibility, HDFS provides a pluggable interface that simplifies the task of implementing new block placement policies [9].) Figure 10 shows how the new policy gives a 22% improvement in the running time of a partition-wise join job by collocating the joining partitions.

Experimental results like those above motivate the need for a Workflow-aware Scheduler that can run jobs in a workflow such that the overall performance of the workflow is optimized. Workflow performance can be measured in terms of running time, resource utilization in the Hadoop cluster, and robustness to failures (e.g., minimizing the need for cascading reexecution of tasks due to node failure or data corruption) and transient issues (e.g., reacting to the slowdown of a node due to temporary resource contention). As illustrated by Figures 7–10, good layouts of the initial (base), intermediate (temporary), and final (results) data in a workflow are vital to ensure good workflow performance.

Workflow-aware Scheduling: A Workflow-aware Scheduler can ensure that job-level optimization and scheduling policies are co-

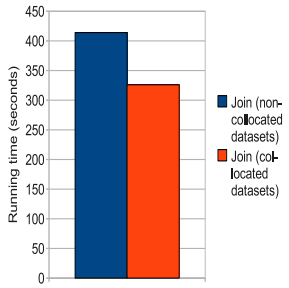

Figure 10: Respective execution times of a partition-wise join job with noncollocated and collocated input partitions

ordinated tightly with the policies for data placement employed by the underlying distributed filesystem. Rather than making decisions that are locally optimal for individual MapReduce jobs, Starfish's Workflow-aware Scheduler makes decisions by considering producer-consumer relationships among jobs in the workflow.

Figure 11 gives an example of producer-consumer relationships among three Jobs P, $C1$, and $C2$ in a workflow. Analyzing these relationships gives important information such as:

- What parts of the data output by a job are used by downstream jobs in the workflow? Notice from Figure 11 that the three writer tasks of Job P generate files *File1*, *File2*, and *File3* respectively. (In a MapReduce job, the writer tasks are map tasks in a map-only job, and reduce tasks otherwise.) Each file is stored as blocks in the distributed filesystem. (HDFS blocks are 64MB in size by default.) *File1* forms the input to Job $C1$, while *File1* and *File2* form the input to Job $C2$. Since *File3* is not used by any of the downstream jobs, a Workflow-aware Scheduler can configure Job P to avoid generating *File3*.
- What is the unit of data-level parallelism in each job that reads the data output by a job? Notice from Figure 11 that the data-parallel reader tasks of Job $C1$ read and process one data block each. However, the data-parallel reader tasks of Job $C2$ read one file each. (In a MapReduce job in a workflow, the data-parallel map tasks of the job read the output of upstream jobs in the workflow.) While not shown in Figure 11, jobs like the join in Figure 10 consist of data-parallel tasks that each read a group of files output by upstream jobs in the workflow. Information about the data-parallel access patterns of jobs is vital to guarantee good data layouts that, in turn, will guarantee an efficient mix of parallel and data-local computation. For *File2* in Figure 11, all blocks in the file should be placed on the same node to ensure data-local computation (i.e., to avoid having to move data to the computation). The choice for *File1*, which is read by both Jobs $C1$ and $C2$, is not so easy to make. The data-level parallelism is at the block-level in Job $C1$, but at the file-level in Job $C2$. Thus, the optimal layout of *File1* from Job $C1$'s perspective is to spread *File1*'s blocks across the nodes so that $C1$'s map tasks can run in parallel across the cluster. However, the optimal layout of *File1* from Job $C2$'s perspective is to place all blocks on the same node.

Starfish's Workflow-aware Scheduler works in conjunction with the What-if Engine and the Just-in-Time Optimizer in order to pick the job execution schedule as well as the data layouts for a workflow. The space of choices for data layout includes:

1. What block placement policy to use in the distributed filesystem for the output file of a job? HDFS uses the *Local Write* block placement policy which works as follows: the first replica of any block is stored on the same node where the block's writer (a map or reduce task) runs. We have implemented a new *Round Robin* block placement policy in HDFS where the blocks written are stored on the nodes of the distributed filesystem in a round robin fashion.

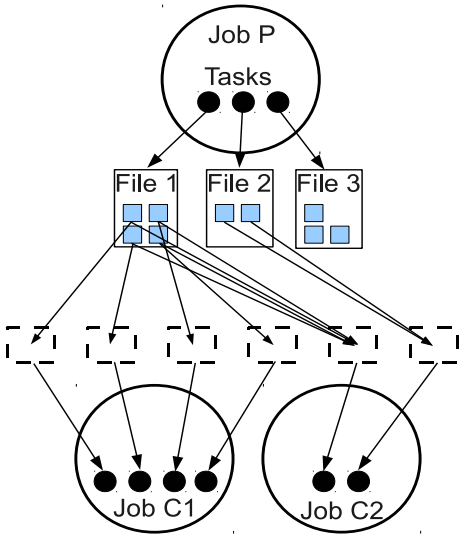

Figure 11: Part of an example workflow showing producer-consumer relationships among jobs

2. How many replicas to store—called the *replication factor*—for the blocks of a file? Replication helps improve performance for heavily-accessed files. Replication also improves robustness by reducing performance variability in case of node failures.
3. What size to use for blocks of a file? For a very big file, a block size larger than the default of 64MB can improve performance significantly by reducing the number of map tasks needed to process the file. The caveat is that the choice of the block size interacts with the choice of job-level configuration parameters like *io.sort.mb* (recall Section 3).
4. Should a job's output files be compressed for storage? Like the use of Combiners (recall Section 3), the use of compression enables the cost of local I/O and network transfers to be traded for additional CPU cost. Compression is not always beneficial. Furthermore, like the choice of the block size, the usefulness of compression depends on the choice of job-level parameters.

The Workflow-aware Scheduler performs a cost-based search for a good layout for the output data of each job in a given workflow. The technique we employ here asks a number of questions to the What-if Engine; and uses the answers to infer the costs and benefits of various choices. The what-if questions asked for a workflow consisting of the producer-consumer relationships among Jobs P, $C1$, and $C2$ shown in Figure 11 include:

(a) What is the expected running time of Job P if the Round Robin block placement policy is used for P's output files?

(b) What will the new data layout in the cluster be if the Round Robin block placement policy is used for P's output files?

(c) What is the expected running time of Job $C1$ ($C2$) if its input data layout is the one in the answer to Question (b)?

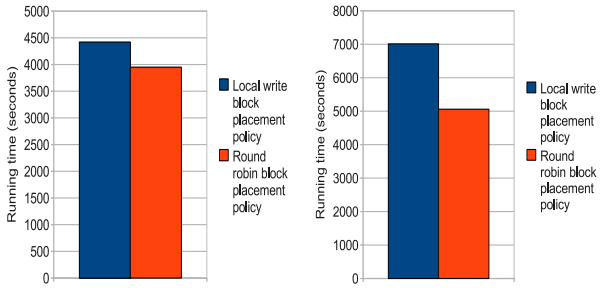

Figure 12: Respective running times of (a) a partition job and (b) a two-job workflow with the (default) Local Write and the Round Robin block placement policies used in HDFS

(d) What are the expected running times of Jobs $C1$ and $C2$ if they are scheduled concurrently when Job P completes?

(e) Given the Local Write block placement policy and a replication factor of 1 for Job P's output, what is the expected increase in the running time of Job $C1$ if one node in the cluster were to fail during $C1$'s execution?

These questions are answered by the What-if Engine based on a simulation of the main aspects of workflow execution. This step involves simulating MapReduce job execution, task scheduling, and HDFS block placement policies. The job-level and cluster-level information described in Section 3 is needed as input for the simulation of workflow execution.

Figure 12 shows results from an experiment where the Workflow-aware Scheduler was asked to pick the data layout for a two-job workflow consisting of a partition job followed by a sort job. The choice for the data layout involved selecting which block placement policy to use between the (default) Local Write policy and the Round Robin policy. The remaining choices were kept constant: replication factor is 1, the block size is 128MB, and compression is not used. The choice of collocation was not considered since it is not beneficial to collocate any group of datasets in this case.

The Workflow-aware Scheduler first asks what-if questions regarding the partition job. The What-if Engine predicted correctly that the Round Robin policy will perform better than the Local Write policy for the output data of the partition job. In our cluster setting on Amazon EC2, the local I/O within a node becomes the bottleneck before the parallel writes of data blocks to other storage nodes over the network. Figure 12(a) shows the actual performance of the partition job for the two block placement policies.

The next set of what-if questions have to do with the performance of the sort job for different layouts of the output of the partition job. Here, using the Round Robin policy for the partition job's output emerges a clear winner. The reason is that the Round Robin policy spreads the blocks over the cluster so that maximum data-level parallelism of sort processing can be achieved while performing data-local computation. Overall, the Workflow-aware Scheduler picks the Round Robin block placement policy for the entire workflow. As seen in Figure 12(b), this choice leads to the minimum total running time of the two-job workflow. Use of the Round Robin policy gives around 30% reduction in total running time compared to the default Local Write policy.

5. OPTIMIZATION AND PROVISIONING FOR HADOOP WORKLOADS

Workload Optimizer: Starfish's Workload Optimizer represents the workload as a directed graph and applies the optimizations listed in Section 2.3 as graph-to-graph transformations. The optimizer

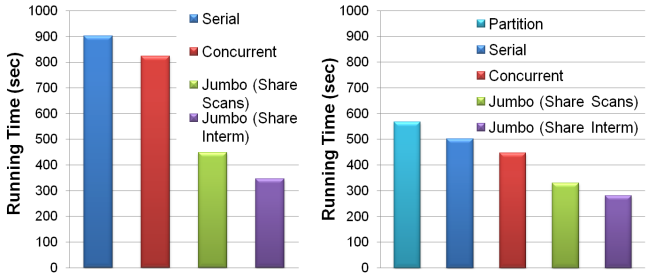

Figure 13: Processing multiple SPA workflow nodes on the same input dataset

uses the What-if Engine to do a cost-based estimation of whether the transformation will improve performance.

Consider the workflows that produce the results IV, V, and VI in Figure 2. These workflows have a join of Users and Clicks in common. The results IV, V, and VI can each be represented as a Select-Project-Aggregate (SPA) expression over the join. Starfish has an operator, called the *Jumbo operator*, that can process any number of logical SPA workflow nodes over the same table in a single MapReduce job. (MRShare [14] and Pig [15] also support similar operators.) Without the Jumbo operator, each SPA node will have to be processed as a separate job. The Jumbo operator enables sharing of all or some of the map-side scan and computation, sorting and shuffling, as well as the reduce-side scan, computation, and output generation. At the same time, the Jumbo operator can help the scheduler to better utilize the bounded number of map and reduce task slots in a Hadoop cluster.

Figure 13(a) shows an experimental result where three logical SPA workflow nodes are processed on a 24GB dataset as: (a) *Serial*, which runs three separate MapReduce jobs in sequence; (b) *Concurrent*, which runs three separate MapReduce jobs concurrently; (c) using the Jumbo operator to share the map-side scans in the SPA nodes; and (d) using the Jumbo operator to share the map-side scans as well as the intermediate data produced by the SPA nodes. Figure 13(a) shows that sharing the sorting and shuffling of intermediate data, in addition to sharing scans, provides additional performance benefits.

Now consider the workflows that produce results I, II, IV, and V in Figure 2. These four workflows have filter conditions on the age attribute in the Users dataset. Running a MapReduce job to partition Users based on ranges of age values will enable the four workflows to prune out irrelevant partitions efficiently. Figure 13(b) shows the results from applying partition pruning to the same three SPA nodes from Figure 13(a). Generating the partitions has significant overhead—as seen in Figure 13(b)—but possibilities exist to hide or reduce this overhead by combining partitioning with a previous job like data copying. Partition pruning improves the performance of all MapReduce jobs in our experiment. At the same time, partition pruning decreases the performance benefits provided by the Jumbo operator. These simple experiments illustrate the interactions among different optimization opportunities that exist for Hadoop workloads.

Elastisizer: Users can now leverage pay-as-you-go resources on the cloud to meet their analytics needs. Amazon Elastic MapReduce allows users to instantiate a Hadoop cluster on EC2 nodes, and run workflows. The typical workflow on Elastic MapReduce accesses data initially from S3, does in-cluster analytics, and writes final output back to S3 (Figure 2). The cluster can be released when the workflow completes, and the user pays for the resources used. While Elastic MapReduce frees users from setting up and maintaining Hadoop clusters, the burden of cluster provisioning is still

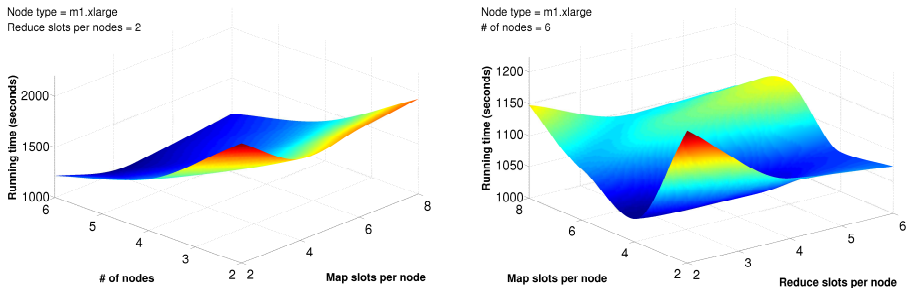

Figure 14: Workload performance under various cluster and Hadoop configurations on Amazon Elastic MapReduce

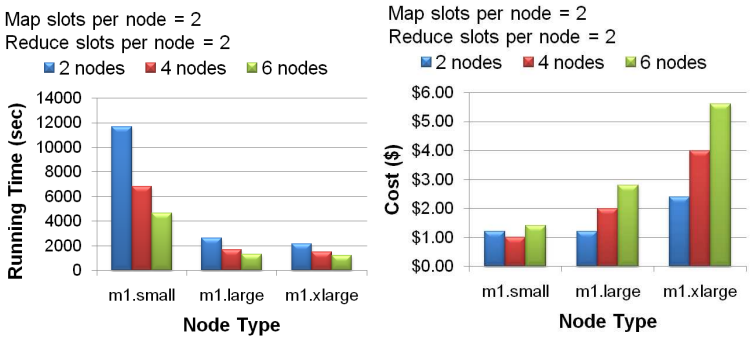

Figure 15: Performance Vs. pay-as-you-go costs for a workload on Amazon Elastic MapReduce

on the user. Specifically, users have to specify the number and type of EC2 nodes (from among 10+ types) as well as whether to copy data from S3 into the in-cluster HDFS. The space of provisioning choices is further complicated by Amazon Spot Instances which provide a market-based option for leasing EC2 nodes. In addition, the user has to specify the Hadoop-level as well as job-level configuration parameters for the provisioned cluster.

One of the goals of Starfish's Elastisizer is to automatically determine the best cluster and Hadoop configurations to process a given workload subject to user-specified goals (e.g., on completion time and monetary costs incurred). To illustrate this problem, Figure 14 shows how the performance of a workload W consisting of a single workflow varies across different cluster configurations (number and type of EC2 nodes) and corresponding Hadoop configurations (number of concurrent map and reduce slots per node).

The user could have multiple preferences and constraints for the workload, which poses a multi-objective optimization problem. For example, the goal may be to minimize the monetary cost incurred to run the workload, subject to a maximum tolerable workload completion time. Figures 15(a) and 15(b) show the running time as well as cost incurred on Elastic MapReduce for the workload W for different cluster configurations. Some observations from the figures:

- If the user wants to minimize costs subject to a completion time of 30 minutes, then the Elastisizer should recommend a cluster of four m1.large EC2 nodes.
- If the user wants to minimize costs, then two m1.small nodes are best. However, the Elastisizer can suggest that by paying just 20% more, the completion time can be reduced by 2.6x.

To estimate workload performance for various cluster configurations, the Elastisizer invokes the What-if Engine which, in turn, uses a mix of simulation and model-based estimation. As discussed in Section 4, the What-if Engine simulates the task scheduling and block-placement policies over a hypothetical cluster, and uses performance models to predict the data flow and performance of the MapReduce jobs in the workload. The latest Hadoop release includes a Hadoop simulator, called *Mumak*, that we initially attempted to use in the What-if Engine. However, Mumak needs a workload execution trace for a specific cluster size as input, and cannot simulate workload execution for a different cluster size.

6. RELATED WORK AND SUMMARY

Hadoop is now a viable competitor to existing systems for big data analytics. While Hadoop currently trails existing systems in peak query performance, a number of research efforts are addressing this issue [1, 7, 10]. Starfish fills a different void by enabling Hadoop users and applications to get good performance automatically throughout the data lifecycle in analytics; without any need on their part to understand and manipulate the many tuning knobs available. A system like Starfish is essential as Hadoop usage continues to grow beyond companies like Facebook and Yahoo! that have considerable expertise in Hadoop. New practitioners of big data analytics like computational scientists and systems researchers lack the expertise to tune Hadoop to get good performance.

Starfish's tuning goals and solutions are related to projects like Hive, Manimal, MRShare, Nectar, Pig, Quincy, and Scope [3, 8, 14, 15, 18]. The novelty in Starfish's approach comes from how it focuses *simultaneously* on different workload granularities—overall workload, workflows, and jobs (procedural and declarative)—as well as across various decision points—provisioning, optimization, scheduling, and data layout. This approach enables Starfish to handle the significant interactions arising among choices made at different levels.

7. ACKNOWLEDGEMENTS

We are extremely grateful to Amazon Web Services for awarding us a grant for the use of resources on the Amazon Elastic Compute Cloud (EC2) and Elastic MapReduce.

8. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.

[2] *BTrace: A Dynamic Instrumentation Tool for Java*. http://kenai.com/projects/btrace.
[3] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*, 2010.
[4] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference*, 2004.
[5] Cloudera: 7 tips for Improving MapReduce Performance. http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance.
[6] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.
[7] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3(1), 2010.
[8] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, 2010.
[9] Pluggable Block Placement Policies in HDFS. issues.apache.org/jira/browse/HDFS-385.
[10] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1), 2010.
[11] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making Cloud Intermediate Data Fault-tolerant. In *SoCC*, 2010.
[12] TeraByte-scale Data Cycle at LinkedIn. http://tinyurl.com/lukod6.
[13] Hadoop MapReduce Tutorial. http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html.
[14] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1), 2010.
[15] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *USENIX Annual Technical Conference*, 2008.
[16] A. Qureshi, R. Weber, H. Balakrishnan, J. V. Guttag, and B. V. Maggs. Cutting the Electric Bill for Internet-scale Systems. In *SIGCOMM*, 2009.
[17] Agile Enterprise Analytics. Keynote by Oliver Ratzesberger at the Self-Managing Database Systems Workshop 2010.
[18] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *ICDE*, 2010.

APPENDIX

A. STARFISH'S VISUALIZER

When a MapReduce job executes in a Hadoop cluster, a lot of information is generated including logs, counters, resource utilization metrics, and profiling data. This information is organized, stored, and managed by Starfish's Metadata Manager in a catalog that can be viewed using Starfish's *Visualizer*. A user can employ the Visualizer to get a deep understanding of a job's behavior during execution, and to ultimately tune the job. Broadly, the functionality of the Visualizer can be categorized into *Timeline views*, *Data-flow views*, and *Profile views*.

A.1 Timeline Views

Timeline views are used to visualize the progress of a job execution at the task level. Figure 16 shows the execution timeline of map and reduce tasks that ran during a MapReduce job execution. The user can observe information like how many tasks were running at any point in time on each node, when each task started and ended, or how many map or reduce waves occurred. The user is able to quickly spot any variance in the task execution times and discover potential load balancing issues.

Moreover, Timeline views can be used to compare different executions of the same job run at different times or with different parameter settings. Comparison of timelines will show whether the job behavior changed over time as well as help understand the impact that changing parameter settings has on job execution. In addition, the Timeline views support a *What-if* mode using which the user can visualize what the execution of a job will be when run using different parameter settings. For example, the user can determine the impact of decreasing the value of *io.sort.mb* on map task execution. Under the hood, the Visualizer invokes the What-if Engine to generate a virtual job profile for the job in the hypothetical setting (recall Section 3).

A.2 Data-flow Views

The Data-flow views enable visualization of the flow of data among the nodes and racks of a Hadoop cluster, and between the map and reduce tasks of a job. They form an excellent way of identifying data skew issues and realizing the need for a better partitioner in a MapReduce job. Figure 17 presents the data flow among the nodes during the execution of a MapReduce job. The thickness of each line is proportional to the amount of data that was shuffled between the corresponding nodes. The user also has the ability to specify a set of filter conditions (see the left side of Figure 17) that allows her to zoom in on a subset of nodes or on the large data transfers. An important feature of the Visualizer is the *Video mode* that allows users to play back a job execution from the past. Using the Video mode (Figure 17), the user can inspect how data was processed and transfered between the map and reduce tasks of the job, and among nodes and racks of the cluster, as time went by.

A.3 Profile Views

In Section 3, we saw how a job profile contains a lot of useful information like the breakdown of task execution timings, resource usage, and data flow per subphase. The Profile views help visualize the job profiles, namely, the information exposed by the Timings, Data-flow, and Resource-level views in a profile; allowing an in-depth analysis of the task behavior during execution. For example, Figure 5 shows parts of two Profile views that display the breakdown of time spent on average in each map and reduce task for two WordCount job executions. Job A was run using the parameter settings as specified by rules of thumb, whereas Job B was run using the settings recommended by the Just-in-time Optimizer (Table 1 in Section 3). The main difference caused by the two settings was more, but smaller, map-side spills for Job B compared to Job A.

We can observe that the map tasks in Job B completed on average much faster compared to the map tasks in Job A; yet the reverse happened to the reduce tasks. The Profile views allow us to see exactly which subphases benefit the most from the parameter settings. It is obvious from Figure 5 that the time spent performing the map processing and the spilling in Job B was significantly lower compared to Job A.

On the other hand, the Combiner drastically decreases the amount of intermediate data spilled to disk (which can be observed in the Data-flow views not shown here). Since the map tasks in Job B processed smaller spills, the reduction gains from the Combiner were also smaller; leading to larger amounts of data being shuffled and processed by the reducers. The Profile views show exactly how much more time was spent in Job B for shuffling and sorting the intermediate data, as well as performing the reduce computation. Overall, the benefit gained by the map tasks in Job B outweighed by far the loss incurred by the reduce tasks, leading to a 2x better performance than Job A.

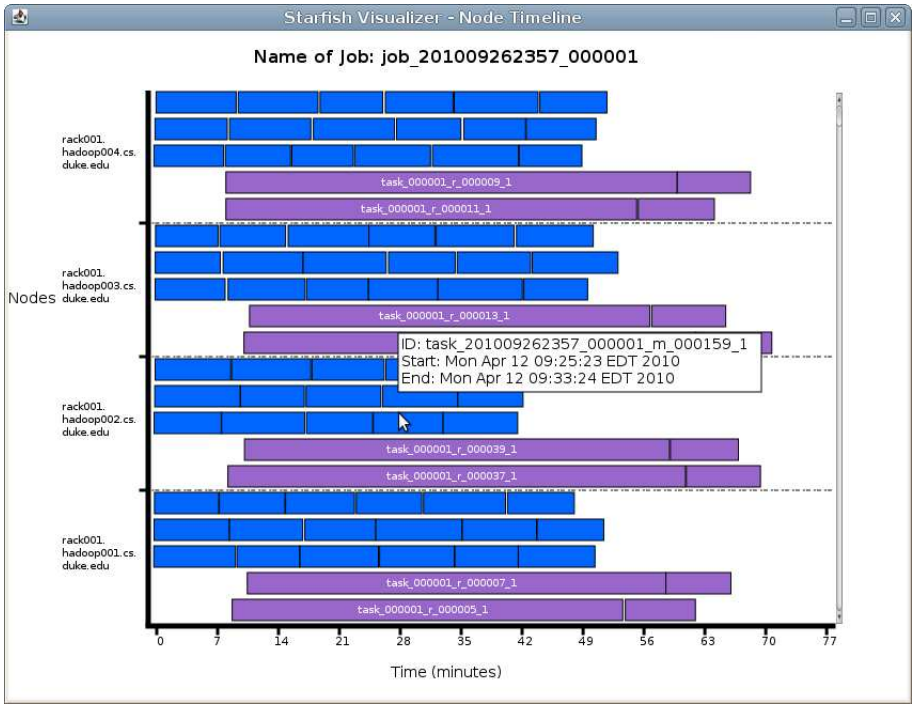

Figure 16: Execution timeline of the map and reduce tasks of a MapReduce job

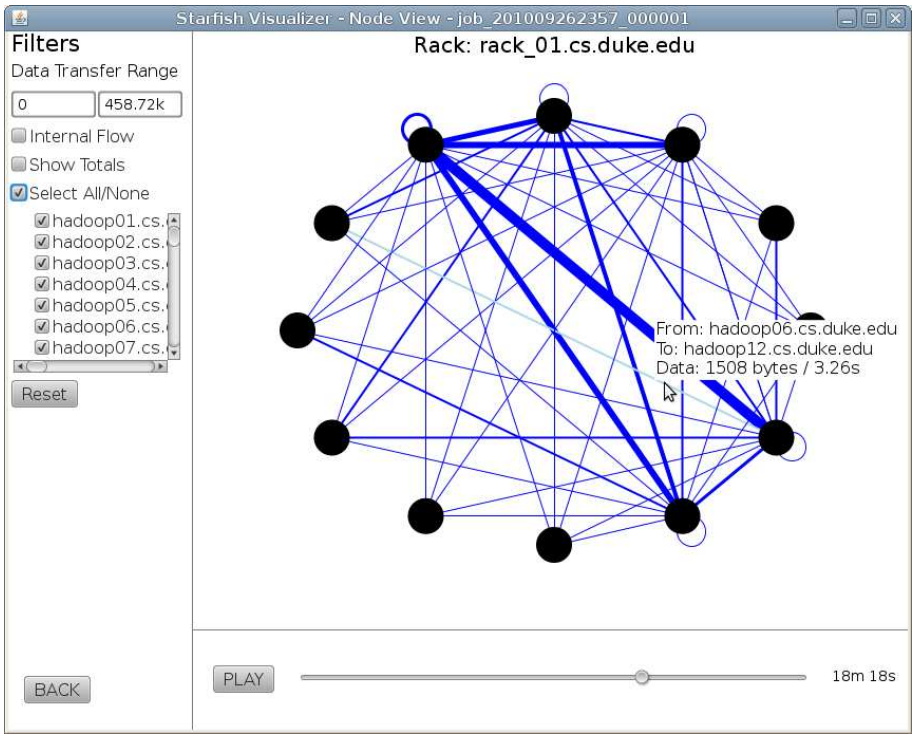

Figure 17: Visual representation of the data-flow among the Hadoop nodes during a MapReduce job execution