CoScan: Cooperative Scan Sharing in the Cloud

Xiaodan Wang
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD, USA
xwang@cs.jhu.edu

Anish Das Sarma*
Google Research
Mountain View, CA, USA
anish.dassarma@gmail.com

Christopher Olston
Yahoo! Research
Sunnyvale, CA, USA
olston@yahoo-inc.com

Randal Burns
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD, USA
randal@cs.jhu.edu

ABSTRACT

We present *CoScan*, a scheduling framework that eliminates redundant processing in workflows that scan large batches of data in a map-reduce computing environment. *CoScan* merges Pig programs from multiple users at runtime to reduce I/O contention while adhering to soft deadline requirements in scheduling. This includes support for join workflows that operate on multiple data sources. Our solution maps well to workflows at many Internet companies which reuse data from a common set of inputs. Experiments on the PigMix data analytics benchmark exhibit orders of magnitude reduction in resource contention with minimal impact on latency.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Algorithms

1. INTRODUCTION

Internet companies process large amounts of semi-structured user content in order to correlate, mine, and extract valuable features. In turn, they have developed several distributed and scalable Cloud processing frameworks [6, 9, 16, 22] for large scale computations. Many of the data processing tasks involve multiple, inter-dependent steps that operate on large batches of continually arriving data performing data-intensive, disk-based processing. For ease of data management, a higher layer abstraction is required to facilitate the flow of data between these tasks, which are not particularly latency sensitive. Nova [27] is one such system developed at Yahoo to manage complex workflows as a graph of inter-connected tasks. Nova eases the management of a workflow's lifecycle by enabling reuse of data processing modules, stateful processing of incrementally arriving data, and support for task versioning and data provenance. The system is built on top of the Pig/Hadoop [16, 28] platform. Hadoop is a scalable, fault-tolerant system for executing map-reduce [8] jobs and Pig provides a high-level language for describing relational algebra style operations on semi-structured data that are compiled into map-reduce.

*Work done while this author was at Yahoo! Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOCC'11, October 27–28, 2011, Cascais, Portugal.
Copyright 2011 ACM 978-1-4503-0976-9/11/10 ...$10.00.

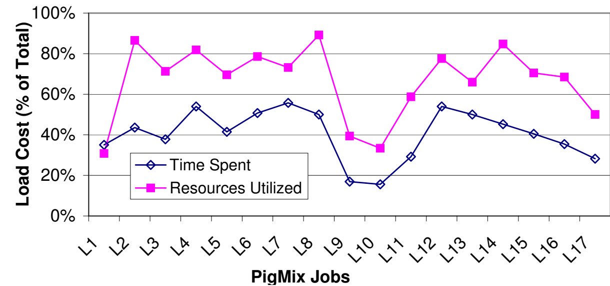


Figure 1: Fraction of time spent on loading and scanning data.

Reuse of task modules and input data at the workflow abstraction layer enables multi-query optimization [32] opportunities such as sharing common input data. Typically, data processing tasks in Nova are scheduled either periodically (once per week) or triggered in a data-driven manner (on arrival of new data in sufficient quantity). For tasks that are reused, the scheduler can predict the occurrence and cost of subsequent executions to provide better latency guarantees. Multiple tasks also contend for access to the same set of input files if, for example, several related research, production, and maintenance workflows are running simultaneously in the same cluster. The scheduler can then physically co-schedule tasks accessing the same data to eliminate redundant processing. Figure 1 reveals that the loading and processing of input data consume a large fraction of total execution cost for jobs from the PigMix [30] benchmark. In terms of running time, a vast majority of jobs spend more than 20% of the time loading input data. Loading inputs consumes an even larger fraction of system resources, approaching 90% of total cost. This difference is explained by the fact that loading of inputs is trivially parallelizable in Hadoop, resulting in many more tasks in the map step relative to the reduce step. As a result, load cost consumes more CPU and I/O resources in Hadoop but does not result in a commensurate impact on running time.

Nova workflows that periodically consume raw news content is one example in which scan sharing opportunities exist. These include: (1) de-duplication workflows that extract features from

newly created news and cluster any duplicate content, (2) geo-tagging workflows that derive location-based metadata from news context, and (3) filtering workflows that classify and label news for generating training data to use in research. These workflows consume news at varying rates and exhibit different freshness requirements. For instance, tighter latency constraints may be imposed on de-duplication by applications that assemble up-to-date new feeds. In contrast, the generation of training data can be scheduled less frequently to avoid contention with workflows used in production.

We design CoScan, a system for sharing data movement and processing costs among multi-step map-reduce workflows executing in Hadoop. CoScan identifies workflows that consume the same set of input files and merges these workflows during execution to ensure that shared data, residing in the Hadoop Distributed File System [16], are scanned only once. The cost savings are two fold. First, the merge reduces I/O cost in that shared inputs are read from disk and transferred over the network once. This reduces network contention by eliminating the need to shuffle the same, multi-terabyte data file to multiple map-reduce jobs. Moreover, shared scan eliminates redundant data processing that includes parsing and unmarshaling of records, which can be fairly heavyweight. As detailed in Section 3.1, these savings mean that fewer I/O (both disk and network) and computation resources are required to perform the same task; thereby improving overall throughput in Hadoop for highly contentious environments.

CoScan also incorporates user specified soft deadlines and attempts to meet these deadlines as closely as possible given available resources and competing workflows in the system. We define soft deadline as a constraint on workflow completion time that can be violated. However, completing a workflow prior to its soft deadline incurs a reward. Soft deadlines closely approximate performance expectations in an I/O-bound, disk-based processing system such as Nova, which is not designed for hard real-time requirements. Since applications that depend on the execution of a workflow may exhibit varying freshness requirements (*i.e.* content may update on a real-time, hourly, or daily basis), failing to meet soft deadline targets in Nova is not catastrophic. Thus, users provide each workflow with a list of soft deadlines and corresponding awards for meeting these deadlines. CoScan then attempts to meet these deadlines as best as possible by maximizing total award over all workflows.

Toward this goal, we make the following contributions:

- **Cost models for merged Pig/map-reduce workflows.** We develop and validate cost models for merged workflows that accurately quantify resource savings and its impact on workflow latency. This allows CoScan to make appropriate trade-offs between throughput and latency.

- **Algorithms for deadline-aware scheduling.** Our heuristic algorithms support both join and non-join workflows when making scheduling decisions. The scheduler takes as input user specified soft deadline functions that reward schedules for early completion. CoScan then reorders and merges workflows to maximize data sharing subject to soft deadlines constraints.

- **Support for data sharing among workflows that perform joins.** Specifically, we augment Pig's [14] multi-query optimizer to merge workflows that join two or more inputs. This includes both map and reduce-side joins.

- **Dynamic adjustments to handle cost estimation errors.** Estimating the performance of new workflows is an error prone process, especially in the presence of data skew or changes to the amount of available system resources. We develop a black box approach to capture errors and refine scheduling decisions at runtime. This allows CoScan to mask cost estimation errors and ensure that workflow latencies are not adversely affected.

We built and evaluated CoScan against the PigMix [30] benchmark, which consists of seventeen queries that test latency and scalability of internal Hadoop workloads at Yahoo. We also evaluated synthetic workloads designed to stress CoScan performance under various conditions. Experiments indicate that CoScan consistently improves throughput and reduces resource consumption, up to a factor of three, in map-reduce. In fact, throughput benefits are greater for joins (*e.g.* workflows that consume multiple input files) and scale with the amount of workflow contention.

The remainder of this paper is organized as follows. We first summarize related works in Section 2. Next, we describe our prior experience implementing scan sharing for Hadoop in Section 3, which covers the mechanisms for sharing and extensions to support the sharing of joins. We then present our main result for workflow scheduling, which balances the benefits of scan sharing with deadline constraints. The solution includes a description of the cost model and optimization problem in Section 4 followed by our algorithmic contributions in Section 5. Finally, we demonstrate the effectiveness of scan sharing using performance results obtained from PigMix in Section 6 and conclude with future works in Section 7.

2. RELATED WORK

Prior works on multi-query optimization allow database queries that are executed together to share work and eliminate redundant data access and computation [5, 12, 15, 17, 34, 38]. These works describe mechanisms for scan sharing that include pipelining of results to queries with common sub-expressions [17] and reordering of queries to maximize the probability that data in the cache is reused by subsequent queries [15]. In Crescando [34], Unterbrunner et. al. describe a scan sharing solution for main memory databases in which incoming queries are batched together and share a single cursor that continuously scans the data table. Candea et. al. [5] present a similar solution for concurrent analysis in data warehouses. Namely, incoming queries latch onto a single physical plan and share the output of continuous scans of a shared fact table. Still other works [35, 36] focus on the scheduling aspects. Queries that access the same portion of a data table are executed together in a manner that prevents query starvation and accounts for precedence constraints.

Scan sharing is studied for workloads against large datasets stored on tertiary storage in order to minimize I/O cost [25, 31, 37]. Yu and Dewitt [37] explored this in Paradise by reordering queries over data stored on magnetic tape. The reordering achieves sequential I/O by collecting data requirements during a pre-execution phase (without physically performing the I/O), reordering tape requests, and finally executing queries concurrently in one batch. Sarawagi et. al. [31] provide non-sequential processing by partitioning the data into fragments that are physically contiguous on the tertiary device and scheduling concurrent queries on a per fragment basis. Andrade et. al. [3] describe a general framework for caching and reusing intermediate results to reduce I/O.

Map-reduce jobs also benefit from scan sharing. For instance, Pig programs that share data are merged [14] into a single map-reduce job with multiple branching pipelines so that the data is scanned only once. This optimization proved effective in lowering resource footprint and improving overall system throughput

for workflows in Nova [27], a data management system developed at Yahoo. Hive [33] provides similar mechanisms for sharing the work of loading and parsing data that is accessed by multiple queries. Nykiel et al. [26] further extend these works by providing a cost-based optimization to scan sharing in map-reduce. Specifically, they employ a dynamic program to identify cases in which the cost of scan sharing outweigh potential benefits. In addition to the mechanisms for scan sharing, our paper also tackles policy.

Agrawal et al. [2] provide a solution to maximize scan sharing in an online setting. Map-reduce jobs are grouped into batches so that sequential scans of large files are shared among as many simultaneous jobs as possible. Average job completion time is minimized by modeling job arrival rates according to a stationary process and scheduling the least sharable jobs first. Scheduling also minimizes the relative difference in completion time between similar jobs regardless of scan sharing. Rather than optimize for aggregate metrics, CoScan attempts to meet individual job deadlines. Users provide soft deadlines and corresponding rewards as input to the system. CoScan then schedules workflows in order to extract as much benefit from scan sharing as possible while meeting deadline requirements. To the best of our knowledge, CoScan is also the first system that explores scan sharing for map and reduce-side joins that share one or more input files.

Multiple works on scheduling theory study the problem of multi-criteria optimization [18, 19, 23] in both single and parallel computing environments. Lai et al. [21] implement an auction-based resource allocation scheme in Tycoon in which users are allotted resources based on the amount they are willing to pay. Dua et al. [10] put forth fraction of soft deadlines met, rather than throughput or response time, as an optimization criteria. They argue that meeting soft deadlines ensures fairness across different applications with varying service level requirements. Similarly, Abbott et al. [1] describe real time scheduling using soft deadlines in which the value of completing a task decreases with time. CoScan optimizes for soft deadlines in the presence of scans sharing among workflows.

3. SCAN SHARING IN NOVA

In this section, we describe our prior experience implementing scan sharing for Nova [27], which is a data processing and management system for Hadoop workflows at Yahoo. We also summarize extensions in CoScan that support fine-grained deadline semantics and scan sharing among join workflows. While Nova provides a mechanism for multiple Pig programs that are executed together to share the same input file, it does not address the scheduling aspects. The scheduling of scan sharing in the presence of deadline constraints is the primary focus of this work.

3.1 Merging Pig Workflows

Nova provides a high level abstraction for managing Hadoop workflows that consume data from the Hadoop Distributed File System [16]. Nova pushes continually-arriving data through each workflow, which is defined as an acyclic data flow graph of map-reduce programs. Additionally, Nova supports stateful, incremental processing on new data and trigger-based execution. Thus, users define triggers for workflows that specify the frequency of execution (*i.e.* re-running when a sufficiently large batch of new data arrives or periodically on an hourly or daily basis). Example use cases include incremental processing of user logs to match ad content or news feeds for de-duplication and ranking search results.

We implemented scan sharing in Nova by identifying workflows that read from the same input file and merging them into a single workflow. To make the example concrete, consider Figure 2 in which two workflows, consisting of Pig programs, share file $c2s0$.

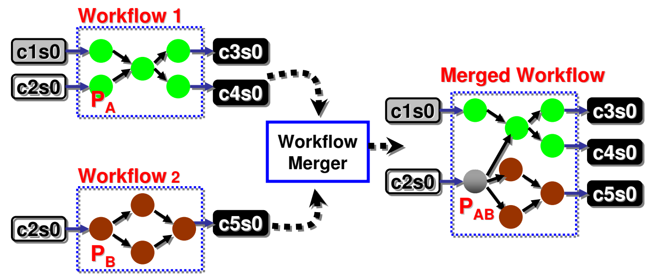


Figure 2: Merging two Pig workflows.

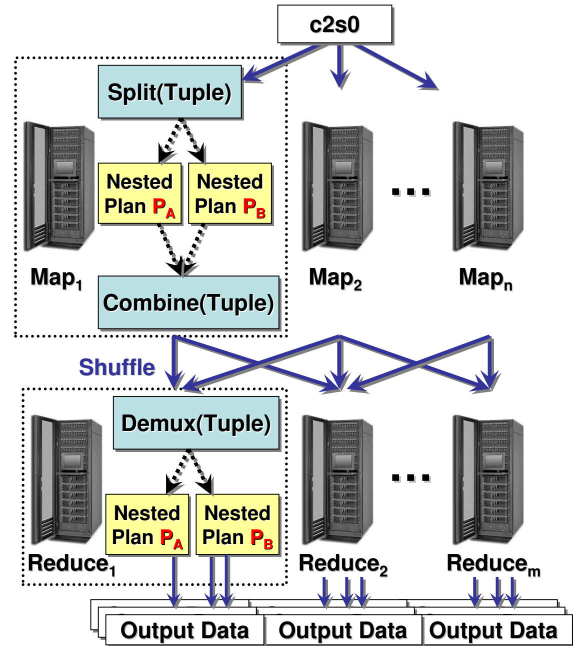


Figure 3: Nested execution of Pig program P_{AB}.

Nova eliminates redundant processing of $c2s0$ by first combining the workflows into a single workflow through simple XML rewriting. Next, Nova replaces the two Pig programs (P_A and P_B) that directly consume $c2s0$ with a single Pig program (P_{AB}) in the data flow graph. Functionally, P_{AB} is a syntactic rewrite that is no different, semantically, than the two original programs. At runtime however, the rewrite ensures that Pig's multi-query optimization amortizes I/O and data processing costs for file $c2s0$ using a single map-reduce job. (Pig's optimizer supports scan sharing on inputs for unary operations such as filter and aggregation but not joins). Finally, the merged workflow is submitted for execution in place of the two original workflows. Currently, our scan sharing implementation only supports map-reduce programs written in Pig.

Scan sharing is achieved in the merged program P_{AB} through nested execution of P_A and P_B within a single map-reduce job to eliminate redundant I/O and processing overhead (Figure 3). First, each record consumed by the map task is split into two branches corresponding to the map-side computations of P_A and P_B respectively. A combine operation then takes the intermediate key/value pairs emitted from both map-side branches and appends an index to each key to identify the branch which produced the key/value pair. This indexed key is then used to partition and shuffle the intermediate results to the appropriate branch in the reduce task. Con-

solidating computation into a single map-reduce job in this fashion means that file $c2s0$ is read from disk, transferred over the network, and unmarshaled only once. Furthermore, start-up and tear-down overhead is reduced due to the launching of fewer map-reduce jobs. However, these benefits are achieved by sacrificing parallelism. We moderate the impact of scan sharing on latency by increasing the number of reduce tasks in CoScan (Section 5.4).

3.2 CoScan Extension

CoScan improves scan sharing in Nova by providing fine-grained soft deadline guarantees on workflow completion. In Nova, workflow instances that are subject to scan sharing are queued by the scheduler for as long as a user specified maximum queue time. Two workflows are merged if (1) they read from the same input file, and (2) neither workflow's maximum queue time has expired. Finally, a merged workflow is submitted for execution when the minimum queue time of its constituent workflow instance expires. In CoScan, users specify soft-deadline requirements for each workflow that reward the scheduler for early completion. CoScan also tracks prior executions of each workflow in order to estimate its expected completion time. (Note that when workflows are merged, the latency of constituent workflows are distorted). CoScan then determines *when* each workflow executes and *which* workflows to merge in order to maximize scan sharing subject to deadline constraints.

CoScan also supports workflows containing Pig programs that join multiple input files. Our implementation in Nova did not support scan sharing for join workflows due to a limitation in Pig's multi-query optimizer. Specifically, the optimizer cannot merge multiple joins into a single map-reduce job such that inputs are shared. We modified Pig's multi-query optimizer for CoScan to support three types of common join operations. The first is a reduce-side join which groups records into bags based on the join key at the map-side and joins records within each bag at the reduce-side. Two specialized map-side joins are also supported: replicated join (one input fits in memory and is replicated at each map task) and sort-merge join (both inputs are sorted and an indexed seek on the right input is performed for each key from the left input).

4. INELASTIC COMPUTE MODEL

In this section, we construct an Inelastic Compute (IC) Model for the scheduling of scan sharing optimizations in map-reduce. (Inelastic because we focus on environments with a fixed amount of computation resources). The IC model optimizes for a set of input workflows that operate on files residing on the cluster. Workflows that operate on the same inputs can be merged and executed more efficiently by loading the inputs only once. This is beneficial if reducing the utilization of I/O and CPU resources leads to lower operating costs (*i.e.* resources offered on a pay-per-use pricing model). In other cases, a user may be willing to pay the cluster operator a bonus for completing workflows early and would like to prevent scan sharing from distorting workflow latencies (*i.e.* failing to meet deadlines that would otherwise have been met by executing two workflows separately). In this section, we balance the benefits of scan sharing with awards from meeting soft deadlines.

4.1 Performance Metric

We optimize along two performance dimensions in CoScan: resource utilization (overall system throughput) and soft deadlines (job latency). Resource utilization measures the aggregate amount of computation resources utilized by map and reduce tasks in a Hadoop cluster. In particular, each physical machine is allocated a fixed number of slots to run map and reduce tasks that are managed by Hadoop. Slot time is the amount of time that a map or reduce task occupies a given slot for computation. This includes time spent waiting to copy input files over the network and CPU time to apply map and reduce functions. We approximate resource usage by summing slot time over all map and reduce tasks. CoScan minimizes this metric by using fewer mapper and reducer slots to perform the same amount of work. This frees additional capacity (allowing the system to scale to more concurrent workflows) and improves overall throughput.

Soft deadlines provide user specified rewards for job completion. (Rewards can vary depending on the importance of a job and its deadline). Meeting a soft deadline requires that CoScan minimize a job's makespan, which is defined as the total elapsed time from start (submission of a workflow) to finish (returning the output to users). While merging workflows with shared inputs leads to consistent reduction in resource utilization, it distorts individual workflow makespans relative to executing the constituent workflows independently. As illustrated in Section 3.1, both map and reduce functions for constituent workflows are nested within the same map-reduce job. This reduces parallelism such that the makespan of merged workflows scales linearly with the number of constituent workflows. In the worst case, map and reduce functions of constituent workflows are applied serially, one after another. For latency sensitive jobs, this means that scan sharing are sometimes avoided in order to meet soft deadlines.

4.2 Model Inputs

This section describes inputs to our execution model and the cost estimation framework on which CoScan scheduling decisions are based. Each workflow or job first loads a set of input files and then applies map and reduce functions on the data. The load cost (*e.g.* I/O and unmarshaling of inputs) can be shared among jobs that operate on the same input. Each job must also meet certain soft deadline requirements. This leads to the optimization problem: maximize the benefits of sharing without violating soft deadlines.

Definition 1. The total time required to scan an input file F in a fixed sized cluster is $s_m(F)$ while incurring $s_u(F)$ resources across all map tasks.

The execution time and amount of cluster resources required to scan an input file are estimated by loading the input using a map-only job, eliminating all records using a filter operator, and storing the empty result. In practice, CoScan relies on the total size of the inputs and number of available map slots in the cluster to approximate the makespan performance of scans. (Input size is used to approximate the number of file splits and, hence, the number of map tasks required). Scan cost estimates are accurate even for join jobs that scan multiple inputs.

Definition 2. A job J is defined by a set of input files $f(J)$ it scans, a fixed processing cost ($c_m(J)$ for time and $c_u(J)$ for resources required), and a deadline $d(J)$.

Executing a job requires that we scan all inputs in $f(J)$ and apply some fixed processing on the data. Thus, the makespan (job execution time) and resources utilization of a job J can be estimated by $\sum_{F \in f(J)} s_m(F) + c_m(J)$ and $\sum_{F \in f(J)} s_u(F) + c_u(J)$ respectively. Moreover, J is a join job if $|f(J)| > 1$ (*e.g.* operates on multiple inputs). Next, we formally define soft deadlines, which dictate the completion time of jobs.

Definition 3. A soft deadline d is defined by n pairs (t_i, p_i) for $1 \leq i \leq n$ in which $0 < t_i < t_{i+i}$ and $p_i > p_{i+1}$.

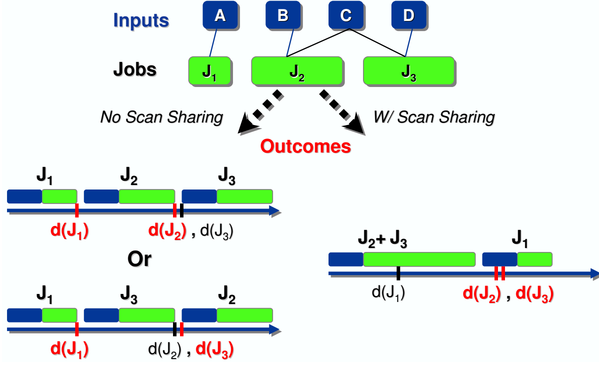


Figure 4: Scheduling outcomes (with and without scan sharing). Deadlines met are shown in bold.

If the job completes by time t_i, then p_i points are awarded to the scheduler. (We assume that the earliest deadline, t_1, is at least as large as the job's makespan). For instance, latency sensitive jobs are assigned tighter deadlines and larger awards compared with long running jobs in which users are willing to wait. CoScan prioritizes jobs that are submitted together based on point assignment and attempts to meet soft deadlines on a best-effort basis.

Definition 4. The upper bound on the time required to execute a merged job consisting of J_1 and J_2 is $s_m(f(J_1) \cup f(J_2)) + c_m(J_1) + c_m(J_2)$.

We bound makespan in the worst case by allowing the initial scan cost to be shared while assuming that processing costs ($c_m(J_1)$ and $c_m(J_2)$) are not parallelizable. Note that $s_m(f(J_1) \cup f(J_2))$ estimates the makespan of scanning all inputs as one unit. For small inputs, multiple files can be loaded in parallel in a single wave of map tasks. For inputs that are too large to be loaded in a single wave, scan cost is approximated by summing the cost of scanning each individual file (*e.g.* $\sum_{F \in f(J_1) \cup f(J_2)} s_m(F)$). Moreover, makespan performance vary wildly for merged jobs. While makespan of some jobs approach the estimated upper bound, other jobs (*i.e.* joins) tend to finish well under these bounds. Several factors contribute to these differences including the amount of network congestion and data skew. While better makespan estimates can be derived through in-depth analysis of the map-reduce pipeline [7, 13, 24], these techniques are beyond the scope of this paper.

Definition 5. The resource utilization of a merged job consisting of J_1 and J_2 is $\sum_{F \in f(J_1) \cup f(J_2)} s_u(F) + c_u(J_1) + c_u(J_2)$.

Considering scan cost alone, we expect savings in resource usage to converge sublinearly with the number of jobs merged for non-join jobs. Specifically, merging two jobs eliminates one redundant scan (50% savings), three jobs eliminates two redundant scans (66% savings), and so on. This model remains accurate for up to six-way merges. In addition, savings in resource utilization is fairly insensitive to changes in resource contention or cluster configuration (*i.e.* addition or removal of machines).

4.3 Optimization Problem

Given a set of jobs, CoScan finds a schedule that executes all jobs while maximizing the aggregate number of points awarded from meeting soft deadlines over all jobs. In particular, the scheduler shuffles the execution order of jobs and employs scan sharing when appropriate in order to balance resource usage with rewards for early job completion.

To illustrate various dimensions to this problem, consider the scheduling of jobs J_1, J_2, and J_3 which scan four inputs in Figure 4. For simplicity, only the earliest soft deadline for each job is marked in the three scheduling outcomes. (Deadlines met are in bold). J_1's deadline is the most restrictive and satisfying all three deadlines simultaneously is not possible. If no scan sharing is employed, we are left with two outcomes that satisfy soft deadlines for either J_1 and J_2 or J_1 and J_3. With scan sharing on input C, we can merge and satisfy the deadlines for both J_2 and J_3. The optimal outcome depends on the number of points awarded for the early completion of J_1. If every job is weighted the same, then scan sharing is preferred since it reduces resource utilization and overall runtime (*e.g.* increased throughput).

Formally, we define a schedule $S = [s_1, ..., s_n]$ as a complete execution plan of a set of input jobs $\mathbb{J}$. An element s_i consists of a set of one or more jobs ($s_i \subseteq \mathbb{J}$) that share inputs and are merged during execution. Moreover, the elements in S are ordered; that is, all jobs in s_i are executed prior to s_j if $i < j$. Using Figure 4 as example, the three schedules corresponding to each outcome are: $[\{J_1\}, \{J_2\}, \{J_3\}]$, $[\{J_1\}, \{J_3\}, \{J_2\}]$, and $[\{J_2, J_3\}, \{J_1\}]$. Next, let μ be a function in which $\mu(s_i)$ is the resource utilization of executing jobs in s_i (Definition 5). Let $\tau^S(s_i)$ be the estimated completion time of jobs in s_i under schedule S. Thus, $\tau^S(s_1)$ is the makespan of s_1 (Definition 4), $\tau^S(s_2)$ is $\tau^S(s_1)$ plus the makespan of s_2, and so on. Finally, let ρ be a function in which $\rho(\tau^S(s_i))$ denotes the number of points awarded for the completion of jobs in s_i by time $\tau^S(s_i)$.

Provided a set of jobs, a set of input files, and a fixed sized computing cluster, the optimal schedule S is one that maximizes total points awarded ($\sum_{i=1}^{n} \rho(\tau^S(s_i))$) while utilizing the least amount of resources ($\sum_{i=1}^{n} \mu(s_i)$). In the following section, we describe two algorithms that solve this optimization problem.

5. SCHEDULING ALGORITHM

This section presents our algorithmic results that aim to optimize the number of points awarded by employing scan sharing. We start by showing that solving the general optimization problem is intractable (Section 5.1). Therefore, we resort to efficient heuristic solutions and present two main algorithmic techniques. We first present a suite of algorithms based on greedy ordering (GO) of jobs (Section 5.2) that are highly efficient but potentially sensitive to the ordering. Any of the greedy algorithms may be used to bootstrap our second set of approaches based on local improvements, with simulated annealing-style selection that provide more robust solutions (Section 5.3). We conclude with a description of optimizing for the parallelism of reduce tasks, handling errors in cost estimates, and applying our algorithms in the online setting (Section 5.4).

5.1 Complexity

We show that it is intractable to optimally solve our scheduling problem, meaning that there does not exist any polynomial time exact algorithm.

THEOREM 1. *Maximizing points awarded in the Inelastic Compute Model is NP-complete when all jobs scan a single file and have soft deadlines.*

PROOF. **NP-hard:** There is a direct reduction from the 0-1 Knapsack problem, a well-known NP-hard problem [20]. Let the maximum weight of the knapsack be W. Each item i with weight w_i becomes a distinct input file that requires time w_i to scan. (There is exactly one job corresponding to each file). The value v_i of an item

```
Inputs:  jobs 𝕁, files F, deadline D
01  S = sort 𝕁 in ascending order by D
02  for job set s_i ∈ S in order, 1 ≤ i < n
03   for job set s_k ∈ S in order, i < k ≤ n
04    if jobs in s_i and s_k share inputs
05      S' = S ∪ {merge(s_i, s_k)} - {s_i, s_k}
06      if morePoints(S', S)
07        set S = S'
08  return S
```

Figure 5: Greedy ordering algorithm.

i becomes the number of points awarded upon completing the corresponding job before its soft deadline W. We then assign all jobs a single soft deadline of W, after which zero points are awarded.

NP: It is easy to verify that the problem is NP. Given a schedule, we can compute the total points awarded in polynomial time, based on the completion time of each job. □

5.2 Greedy Ordering (GO)

Greedy Ordering (GO) arrives at a feasible schedule by first sorting the jobs based on their soft deadlines, and at each iteration, merge as many jobs that scan the same inputs as possible in sorted order. Our solution shares some similarities with the *earliest deadline first* algorithm [4], but exhibits key differences: a job's makespan is decoupled from its deadline, alternative job sort orders are explored, and jobs can be merged to reduce overall runtime. Our algorithm greedily obtains an ordering of jobs, and then performs merges; for the ordering, we sort jobs using one of the following methods:

- **Earliest soft deadline**: jobs are ranked in increasing order of deadlines in which the earliest soft deadline pair (t_1, p_1) is used. Intuitively, meeting the earliest soft deadline yields the most points for a job. This ordering allows jobs with the most restrictive deadlines to be scheduled first.
- **Steepest point drop**: jobs are scheduled in order of deadlines that if violated, result in the largest loss of points. In particular, the loss of points between two consecutive deadlines t_i and t_j is $p_i - p_j$. The steepest such loss is a pair of deadlines in which $p_i - p_j$ is maximal. Jobs are ordered by t_i such that they are scheduled prior to incurring this loss.
- **Steepest slope**: similar to *steepest point drop* order except that the loss between two consecutive deadlines t_i and t_j is computed as $\frac{p_i - p_j}{t_j - t_i}$. This normalizes loss by the amount of slack, in terms of completion time, between two consecutive deadlines. Scheduling jobs prior to t_i ensures that CoScan avoids a high point loss per unit of time.
- **Normalized deadline**: jobs are scheduled in order of decreasing number of points acquired per unit of time. Specifically, we order jobs based on the earliest soft deadline pair as follows: $\frac{p_1}{t_1}$. This favors jobs that award the most number of points per unit of time for early completion.

We present a simple example to illustrate the various orderings. Consider a job J with soft deadlines given by the set $\{(10s, 28pts), (25s, 13pts), (28s, 5pts)\}$. The earliest soft deadline method simply considers $10s$ as the deadline for ordering J. The steepest point drop occurs between $10s$ and $25s$ for a total drop of $15pts$. In contrast, the steepest slope is between $25s$ and $28s$, with a slope of $\frac{13-5}{28-25} = \frac{8}{3}$ points per second. Finally, the normalized deadline method considers $\frac{28}{10}$ points per second.

Figure 5 illustrates the pseudo-code for GO. Line 1 sorts (using any of the greedy orderings described above) jobs in deadline order and uses the resulting sequence of jobs as the initial schedule S. Lines 2-4 iterate over each set of jobs s_i in S and attempt to merge s_i with jobs in s_k if they scan the same input files. Line 5 rewrites the schedule to perform scan sharing by substituting job sets s_i and s_k with a merged job. Finally, lines 6-7 replace the current schedule if the new schedule S' either increases the number of points awarded or reduces resource utilization without sacrificing points awarded. Note that in practice, merging too many Hadoop jobs together can result in memory issues (*e.g.* running out of heap space). We limit merges to a maximum of six jobs to avoid cascading failures.

```
Inputs:  jobs 𝕁, files F, deadline D
01  S = sort 𝕁 in ascending order by D
02  improved = true
03  while improved do
04   S' = S, improved = false
05   for each pair (s_i, s_k) ∈ S do
06    if jobs in s_i and s_k share inputs
07      S'' = S ∪ {merge(s_i, s_k)} - {s_i, s_k}
08      if morePoints(S'', S') {set S' = S''}
09    S'' = swap(S, s_i, s_k)
10    if morePoints(S'', S') {set S' = S''}
11   for each merged job s_i ∈ S do
12    for each job J ∈ s_i do
13      S'' = split(S, J)
14      if morePoints(S'', S') {set S' = S''}
15   if morePoints(S', S)
16    set S = S', improved = true
17  return S
```

Figure 6: Local improvement algorithm.

5.3 Local Improvement (LI and LI-J)

We start by describing LI, the algorithm that gradually improves a schedule by applying incremental improvements. We then enhance local improvement with LI-J, which is optimized for workloads with lots of join jobs. LI-J accounts for the degree in which an input is shared when selecting inputs to apply scan sharing.

The Local Improvement (LI) algorithm starts with a feasible solution and applies incremental improvements until the schedule cannot be improved further. LI starts from a schedule in which jobs are sorted in deadline order (Section 5.2) and none of the jobs are merged; we may use any greedy ordering to bootstrap LI. At each step, we choose among a polynomial number of possible refinements to the schedule that includes: merging a pair of jobs, swapping the order of two jobs, or splitting a job that was previously merged. We keep the refinement which yields the most points and gradually increase the number of points awarded with each iteration. In addition, we apply simulated annealing style random start at any point to prevent the solution from being stuck at a local minimum. While LI converges on a good solution fairly quickly and provides a modest improvement over GO, it does so at the expense of significantly more computation time on large workloads.

The pseudo-code for LI is provided in Figure 6. We start with an initial schedule on line 1 and improve upon the number of points awarded. Lines 5-10 iterate over every pair of job sets s_i and s_k and make the following refinements: merge the two sets of jobs or swap the execution order of the jobs. At each step, only the best

```
Inputs:  jobs 𝕁, files F, deadline D
01  S = sort 𝕁 in ascending order by D
02  F = sort F in order of significance
03  for each file f ∈ F in order do
04   𝕁′ = scansInput(𝕁,f)
05   for each J_i ∈ 𝕁′ in order, 1 ≤ i < n
06    for each J_k ∈ 𝕁′ in order, i ≤ k ≤ n
07     s_i = set(S,J_i), s_k = set(S,J_k)
08     S′ = S ∪ {merge(s_i,s_k)} - {s_i,s_k}
09     if morePoints(S′,S) {set S = S′}
10   𝕁 = 𝕁 - 𝕁′
11  improved = true
12  while improved do
13   S′ = S, improved = false
14   for each pair (s_i,s_k) ∈ S do
15    S″ = swap(S,s_i,s_k)
16    if morePoints(S″,S′) {set S′ = S″}
17   if morePoints(S′,S)
18    set S = S′, improved = true
19  return S
```

Figure 7: Local improvement with join extension.

refinement (*e.g.* one that increases points awarded or reduces resource utilization) is kept. Lines 11-14 evaluate refinements that split jobs that were previously merged. Finally, lines 15-16 continue until none of the refinements improve upon the schedule from the prior iteration.

We extend local improvement to account for joins in LI-J; that is, jobs with multiple input files that can benefit from scan sharing. This is accomplished by first sorting the input files in descending order of significance: a product of file size and its degree of sharing (number of jobs that scan the file). This metric is a proxy for the total amount of potential savings in resource utilization and is used in other works to address the file-bundling problem [29]. Next, local improvement is applied in two phases, beginning with the merging of jobs in file order. Namely, for jobs that scan a given file, LI-J determines the maximal set of jobs that can be merged without sacrificing the number of points awarded. In the second phase, we swap the execution order of jobs incrementally until no further improvements are possible.

The pseudo-code for LI-J is shown in Figure 7. In lines 1-2, we sort the jobs in deadline order and files in descending order of significance. Lines 3-10 illustrate the first phase of local improvement in which we iterate over each file in order. For a file f, we extract all jobs that scan f in $\mathbb{J}'$. Then for every pair of jobs in $\mathbb{J}'$, we merge the corresponding job sets in the schedule S if more points are obtained. Finally, we remove jobs in $\mathbb{J}'$ from $\mathbb{J}$ and proceed to the next file. In the second phase (lines 11-18), pairs of jobs are swapped until no further improvements can be made to S.

5.4 Additional Optimizations

CoScan includes two optimizations that (1) improve makespan performance by increasing the parallelism of reduce tasks and (2) make scheduling robust to errors in cost estimation. The first optimization arose from the fact that Pig's multi-query optimizer sets the reduce parallelism (*i.e.* number of reduce tasks) of a merged job to the maximum parallelism among its constituent jobs. Often times, this policy leads to considerable increase in makespan as more jobs are merged. By increasing the parallelism of reduce tasks, we observed a factor of two to three reduction in latency. This works up to a point; that is, if the number of reduce tasks saturate the number of reduce slots available in Hadoop, then performance is negatively impacted (*i.e.* overhead from idle reduce tasks and additional start-up and tear down). Thus, CoScan sets the number of reduce tasks by summing the parallelism of the constituent jobs and capping this amount to be no more than the number of available reduce slots.

CoScan relies on past execution of jobs for cost estimation, which may lead to errors (*i.e.* data skew in the input file or configuration change in the Hadoop cluster). Many works [7, 13, 24] exist that estimate the running time of jobs by sampling the input data or performing a static analysis of the execution pipeline. These techniques are complementary and can be applied in CoScan to improve cost estimation. Our goal is to demonstrate the value of scan sharing and not design a better cost estimator. Instead, we develop a black box approach to capture estimation errors at runtime, adjust schedules dynamically, and make CoScan robust to noise.

CoScan adjusts schedules at runtime to mask cost estimation errors and ensures that latency of subsequent jobs are not adversely affected. We first observe, for a completed job, the error between actual and estimated performance. We then record this error, which is keyed by the input file scanned and operation performed. Finally, we adjust cost estimates for similar jobs and re-run the scheduling algorithm for the affected jobs. Consider a simple example with jobs J_1 (*filter*), J_2 (*group-by* with *aggregation*), and J_3 (*filter* with *group-by*) that scan input A. Upon completion of J_1 and J_2, we observe that actual makespan is 10% and 5% more than estimated makespan respectively. CoScan accumulates these errors by recording the following key-value pairs: {*A::filter, 10%*}, {*A::group-by, 5%*}, and {*A::aggregation, 5%*}. The scheduling algorithm is then re-run after the estimated makespan of J_3 is increased by 7.5% (using {*A::filter, 10%*} and {*A::group-by, 5%*}).

Finally, we note that our scheduling algorithms naturally extend to the online setting, where the entire workload is not known apriori and jobs arrive on-the-fly. Specifically, we first employ the offline algorithms to merge and order the jobs seen so far. As new jobs arrive, we re-apply our algorithms for the new workload to obtain a revised schedule.

6. EXPERIMENTS

We implement CoScan for the PigMix [30] web analytics benchmark, which consists of seventeen Pig jobs designed to test latency and scalability requirements of internal workflows at Yahoo. The queries are grouped into roughly the following types: group by with aggregation (L1, L4, L6, L7, L8, L12, L17), nested group by with aggregation (L15, L16), order by (L9, L10), distinct (L11), and joins (L2, L3, L5, L13, L14). The inputs consist of eight data files that include user identification and behavior logs. Consider the following PigMix scripts (altered for ease of presentation) that illustrate use cases for both join and non-join jobs:

L2.pig *(Replicated Join)*

```
A = load 'page_views' as (user, action,
    timespent, query_term, ip, timestamp,
    revenue, page_info, links);
B = foreach A generate user, revenue;
alpha = load 'power_users' as (name,
    phone, address, city, state, zip);
beta = foreach alpha generate name;
C = join B by user, beta by name
    using "replicated";
store C into 'L2out';
```

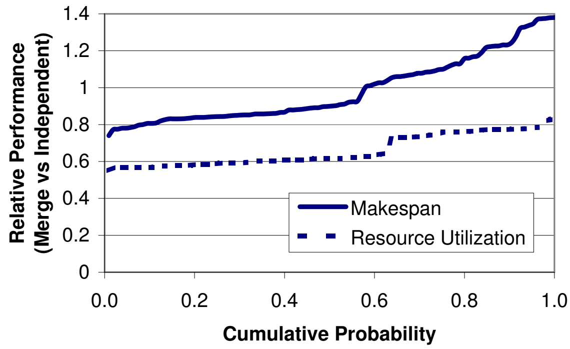


Figure 8: Makespan and resource usage of pairwise merges.

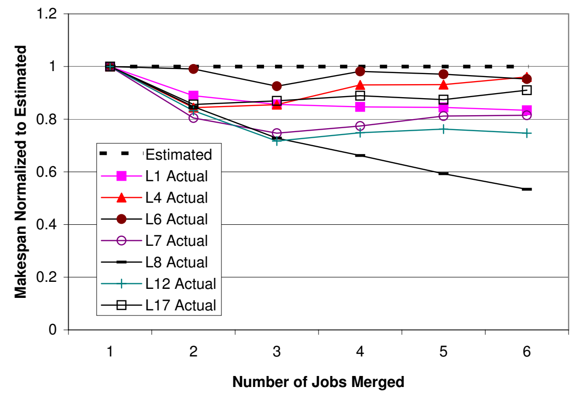


Figure 9: Actual vs estimated makespan.

L4.pig *(Group-by with Aggregation)*

```
A = load 'page_views' as (user, action,
    timespent, query_term, ip, timestamp,
    revenue, page_info, links);
B = foreach A generate user, action;
C = group B by user;
B = foreach C {
    aleph = B.action;
    beth = distinct aleph;
    generate group, COUNT(beth); }
store D into 'L4out';
```

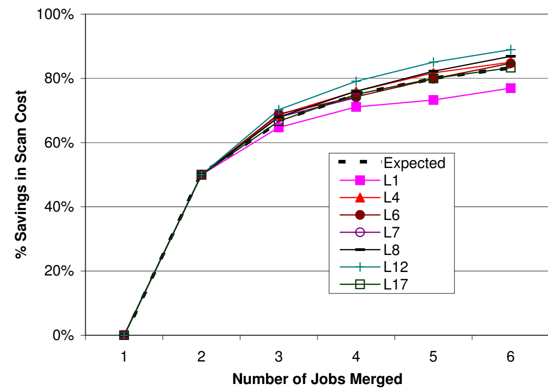


Figure 10: Actual vs estimated savings in scan cost.

L2.pig first loads the *page_views* (behavior log for users) and *power_users* (group of highly active users) tables. The two tables are then joined to determine a list of revenues that power users generated. A replicated join is used which copies (into memory) the *power_users* table for each map task and evaluates the join at the map-side. L4.pig computes the number of unique actions logged for each user. This is accomplished by first projecting each user's actions at the map-side. On the reduce-side, actions are grouped by user and a count is performed on the list of distinct actions.

Our evaluation studies the effectiveness of our scheduling algorithms, performance benefits from join-based optimization, and robustness of cost estimates to noise. We compare GO and LI to NS with respect to makespan and resource utilization (Section 4.1). NS evaluates each job independently with no scan sharing. We first characterize our workload before presenting our main results for PigMix. We then employ synthetic workloads to test the generality of our solution by varying skew in the input data, degree of sharing between jobs, and number of join jobs in the workload.

For experiments, we deployed an eight node Hadoop cluster with 2.4GHz Quad Core CPU and 4GB of RAM per node. Each node is configured as RAID 0 (two 500GB disks) with an HDFS block size of 128MB and allocates four map and two reduce slots. We also generate a 500GB version of the PigMix dataset in which the primary data file, *page_views*, contains 625 million rows.

6.1 Workload

Figure 8 illustrates the performance distribution of two-way merges for pairs of PigMix jobs that share at least one input. Merge performance is relative; that is, makespan and resource utilization is normalized by the cost of executing the pair of jobs independently. (Thus, a relative performance of 1 indicates that there is no performance difference). Reduction in resource utilization is substantial in that between 20% and 45% fewer resources are used by map and reduce tasks to complete the same amount of work. Surprisingly, job latencies are not significantly impacted by merging (up to a 38% increase in makespan) and in fact, improves slightly when join jobs are merged. This is because merging jobs reduces I/O contention (*i.e.* less network congestion) compared with executing the two jobs separately.

We also validate our cost model for estimating makespan and resource utilization in PigMix. Figure 9 plots actual versus estimated makespan performance as we merge an increasing number of jobs of the same type. Recall that our cost model bounds the makespan of merged jobs in the worst case. Thus, we plot actual makespan normalized by the estimated upper bound. While most jobs outperform these bounds, those containing expensive aggregation operations (*i.e.* L4 and L6) track closely with the estimated makespan. Next, consider Figure 10 which compares actual versus estimated savings in scan cost as we merge an increasing number of jobs. For simplicity, we only show non-join jobs in which merged jobs scan the same input. As expected, merging two jobs eliminates one redundant scan (50% savings), three jobs eliminates two redundant scans (66% savings), and so on.

Figure 11 shows the cumulative probability of errors in cost estimation for both single and merged jobs. (Measurements were obtained from the execution of a one hundred job PigMix workload). Resource utilization can be reliably estimated with actual utilization deviating by no more than 5% for most jobs. Error for the remaining jobs do not exceed 15%. For makespan, we measure the amount by which a job's actual execution time exceeds estimated time (Definition 4). 88% of jobs finish prior to the estimated

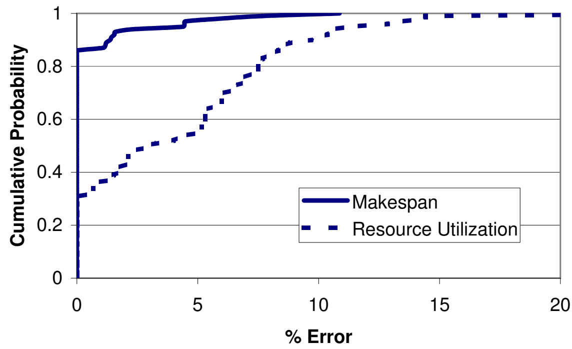


Figure 11: Error in cost estimation.

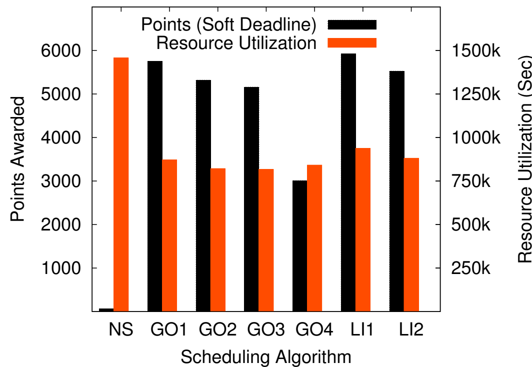


Figure 12: Points awarded and resource usage by algorithm.

time and, with the exception of two jobs, the remaining jobs finish within 5% of the estimated time. Thus, CoScan estimates the performance of merged jobs with reasonably high accuracy.

6.2 Results

We evaluate the performance of a one hundred job PigMix workload across various scheduling algorithms. The workload draws uniformly from the seventeen types of PigMix jobs in which the maximum amount of points a job can receive is drawn from a Zipf distribution. We employ piece-wise linear decay functions for soft deadlines, but supporting alternative decay functions would not affect our core algorithms. Moreover, job arrival is staggered in

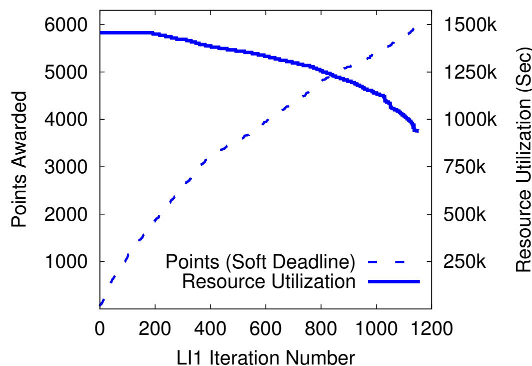


Figure 13: Best schedule over time for local improvement (LI).

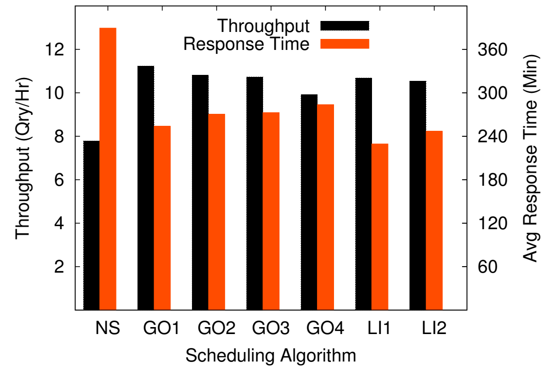


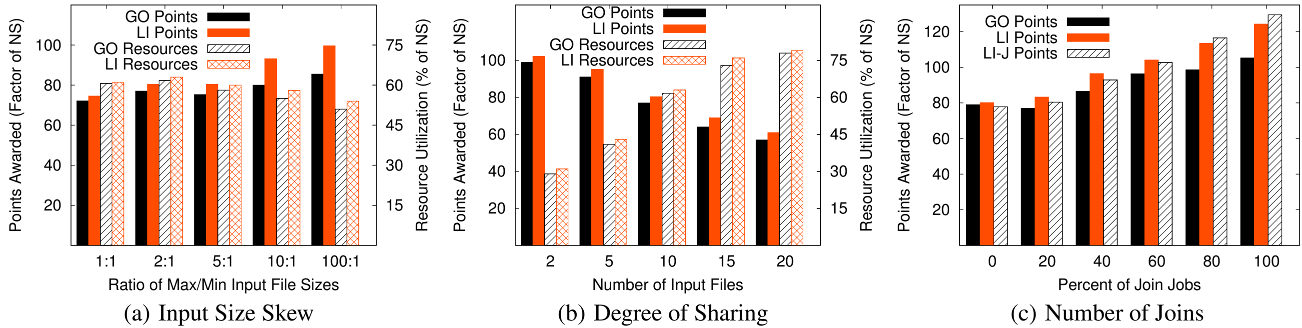
Figure 14: Throughput and response time by algorithm.

which two consecutive jobs are submitted between seven and ten minutes apart and up to two dozen jobs can be in-flight at a given point in time. The workload is evaluated against seven scheduling algorithms: NS which orders jobs by earliest soft deadline without employing scan sharing, GO using four different orderings (GO_1 for earliest soft deadline, GO_2 for steepest point drop, GO_3 for steepest slope drop, and GO_4 for normalized deadline), and LI bootstrapped using two different orderings (LI_1 for earliest soft deadline and LI_2 for arrival order).

Figure 12 evaluates the seven algorithms in terms of the number of points awarded from meeting soft deadlines and the total amount of resources utilized. Of note is that by employing scan sharing, the ability to meet soft deadlines improves by orders of magnitude. (Scan sharing reduces resource usage by roughly 45% over NS). Across the four greedy ordering algorithms, we find that sorting jobs based on the earliest soft deadline yields the best performance in terms of points awarded. Recall that GO_1 schedules the most time constrained jobs first. However, if slack exists between a job's deadline and its estimated makespan, then GO_1 attempts scan sharing for one of the input files by merging the next most time constrained job. Sorting jobs by the steepest point (GO_2) or slope (GO_3) drop perform slightly worse as meeting the earliest deadline dominates over the marginal amount of points gained from meeting a later deadline. GO_4 performs significantly worse because by scheduling jobs with higher point value first, jobs that are time constrained may fail to meet their deadlines.

Finally, we note that local improvement is sensitive to the initial ordering, with a clear advantage for sorting jobs by the earliest deadline (LI_1). LI_1 provides marginal improvement in points awarded over GO_1 by swapping jobs to explore additional sort orders. This is not without a cost. Adding an additional job to the workload increases running time by 5 seconds for LI_1 compared with 8 milliseconds for GO_1. Also, LI_1 actually merges fewer jobs than GO_1 as evident by the higher resource utilization. Figure 13 provides insights into the scheduling decision of LI_1. For each iteration, we plot points awarded and resource usage for the best schedule discovered so far. Toward the end, a marginal improvement in points awarded requires large reductions in resource utilization (half of the reduction is used to achieve the final 10% gain in points). Thus, finding scan sharing opportunities that do not violate existing soft deadlines becomes more difficult with each iteration.

In measuring system throughput and average job response time (Figure 14), scan sharing yields up to 57% increase in throughput and 41% reduction in response time. In broad terms, lower resource utilization translates into improved throughput and more



(a) Input Size Skew (b) Degree of Sharing (c) Number of Joins

Figure 15: Sensitivity of GO and LI algorithms to workload changes.

points awarded results in lower response time. As such, GO$_1$ and LI$_1$ exhibit the best throughput and response time performances respectively. However, the differences are small and indicate that, at the margins, LI$_1$ merges fewer jobs in exchange for better job latencies. For example, a light-weight map-only job should not be merged with a a reduce-heavily job to avoid distorting the latency of the light-weight job. For subsequent results, we use GO$_1$ and LI$_1$ to represent greedy ordering and local improvement.

We study the generality of our algorithms by varying the input files and number of joins using synthetic workloads. The workloads consist of one hundred jobs based on the PigMix benchmark. In Figure 15(a), we vary the skew in the sizes of input files as measured by the ratio of the largest to smallest file size. We compare points awarded and resource utilization for GO and LI. (Performance is normalized against that of NS). While GO and LI are awarded a similar number of points when skew is low, LI provides a nearly 20% improvement when shared inputs differ by a factor of ten or more in size. This is because scan sharing for larger files provides greater savings in resource utilization. Since GO does not account for input sizes when exploiting scan sharing opportunities, it may choose to merge jobs that scan smaller inputs over those that scan larger inputs.

Figure 15(b) compares GO and LI as we vary the degree in which inputs are shared. Specifically, by reducing the number of inputs from twenty to two, we increase the number of jobs that contend for the same file. Again, points awarded and resource utilization are normalized against that of NS. As expected, the difference in points awarded falls drastically as we increase the number of inputs. This is because fewer scan sharing opportunities are available. Likewise, decreasing the number of inputs reduces resource utilization to 41% (factor of two) and 29% (factor of three) of NS for two and five inputs respectively. Clearly, CoScan benefits from high contention when the same inputs are scanned by a large fraction of jobs.

Next, we study the join extension to local improvement (LI-J) by varying the fraction of join jobs (mostly two-way joins) in the workload (Figure 15(c)). In comparing the number of points awarded without joins, the three algorithms exhibit similar performance with a slight advantage for LI. Recall that LI-J merges jobs in a predefined order based on decreasing significance of input files (Section 5.3) whereas LI explores all possible merges. When a join job offers multiple inputs on which to perform scan sharing, LI-J favors sharing the input that leads to the largest reduction in resource utilization. The results show that LI consistently outperforms LI-J at a low fraction of joins (60% or less) and, with more than 80% joins, LI-J yields 4% and 22% improvement over LI and GO respectively. An additional benefit is that the running time of LI-J is significantly lower than that of LI.

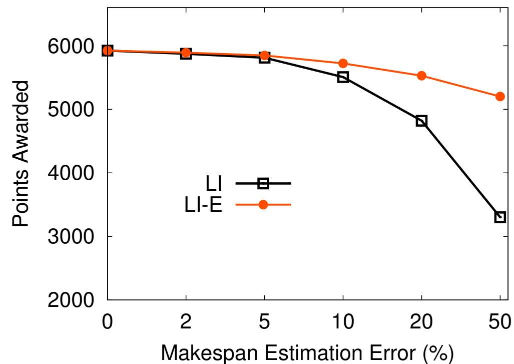


Figure 16: Robustness of LI to errors in makespan estimation.

Figure 16 studies the robustness of LI to cost estimation errors and the effectiveness our solution (Section 5.4) to mask noise in the estimation of makespan at runtime (denoted by LI-E). We add noise to the estimated makespan of individual operations (*i.e.* scanning a given input) and vary the percent spread in the amount of error applied. We then plot the number of actual points awarded for LI and LI-E. The results demonstrate that LI is fairly robust to small errors in the estimation of job makespans (10% error leads to a 7% reduction in points awarded). However, large errors (both over and under-estimation) reduce the effectiveness of LI by nearly 50%. Although, this performance still exceeds that of NS. By comparison, capturing noise in makespan estimation and reordering jobs at runtime in LI-E ensure that CoScan still meets most of the soft deadlines even with large errors in cost estimation (losing 11% of the points at 50% error). Thus, scheduling based on scan sharing ensures efficient utilization of cluster resources and allows the system to meet more soft deadlines even in the presence of cost estimation errors.

7. DISCUSSION

We evaluated several techniques that demonstrate the effectiveness of scan sharing for join and non-join jobs using the Pig/Hadoop [16, 28] platform as a motivating application. These include greedy ordering and local improvement algorithms that balance the benefits of scan sharing with meeting soft deadlines, join extensions optimized for workloads with lots of joins, and runtime techniques that dynamically adjust the schedule to make CoScan robust to errors in cost estimation. Experiments demonstrate that judicious application of scan sharing allows the scheduler to simultaneously

meet more deadlines and lower resource usage. These benefits scale with contention, leading to a three-fold reduction in resource usage when many jobs contend for a small number of files.

In ongoing work, we plan to apply our scheduling results to clusters with elastic resources in which cluster size (number of physical machines) can expand or shrink (*i.e.* Amazon EC2 pricing model [11]). As such, CoScan can allocate additional machines for a job in order to speed-up execution and reduce makespan at the expense of increased resource usage. For example, while a customer may be willing to pay a monetary bonus for early job completion, consuming more resources results in higher operating costs. The goal in the elastic model is to balance soft deadlines with resource usage by maximizing the net reward after operating costs.

We also plan to validate the generality of our approach in other data processing systems (such as Hive [33] and BigTable [6]). This includes implementing specific multi-query optimization mechanisms to support scan sharing and validating our cost model for scheduling. In addition, CoScan requires better cost estimators that predict the execution of new jobs for which prior performance is not available. While CoScan depends on the quality of cost estimates and Pig-specific optimization techniques, scheduling based scan sharing will ensure lower resource footprint and improved throughput performance irrespective of the system.

8. ACKNOWLEDGMENTS

We thank the members of the Yahoo Nova team for their feedback and support. This material is based upon work supported by the National Science Foundation under Grants CMMI-0941530, AST-0939767, and CCF-0937810.

9. REFERENCES

[1] R. Abbott and H. Garcia-Molina. Scheduling Real-time Transactions. *SIGMOD Rec.*, 17:71–81, March 1988.

[2] P. Agrawal, D. Kifer, and C. Olston. Scheduling Shared Scans of Large Data Files. In *VLDB*, 2008.

[3] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient Execution of Multiple Query Workloads in Data Analysis Applications. In *SC*, 2001.

[4] P. Brucker. *Scheduling Algorithms (4th Ed.)*. Springer, 2004.

[5] G. Candea, N. Polyzotis, and R. Vingralek. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In *VLDB*, 2009.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[7] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating Progress of Execution for SQL Queries. In *SIGMOD*, 2004.

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

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

[10] A. Dua and N. Bambos. Scheduling with Soft Deadlines for Input Queued Switches. In *Allerton*, 2006.

[11] Amazon EC2. http://aws.amazon.com/ec2.

[12] P. M. Fernandez. Red Brick Warehouse: A Read-mostly RDBMS for Open SMP Platforms. *SIGMOD Rec.*, 23:492–502, May 1994.

[13] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. *SIGMOD Rec.*, 21:9–18, June 1992.

[14] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of MapReduce: The Pig Experience. *PVLDB*, 2(2):1414–1425, 2009.

[15] A. Gupta, S. Sudarshan, and S. Vishwanathan. Query Scheduling in Multiquery Optimization. In *IDEAS*, 2001.

[16] Apache. Hadoop: Open-Source Implementation of MapReduce. http://hadoop.apache.org.

[17] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, 2005.

[18] H. Hoogeveen. Multicriteria Scheduling. *European Journal of Operational Research*, 167:592–623, 2005.

[19] D. Karger, C. Stein, and J. Wein. Scheduling Algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*. CRC Press, 1997.

[20] R. M. Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[21] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An Implementation of a Distributed, Market-based Resource Allocation System. *Multiagent Grid Syst.*, 1:169–182, August 2005.

[22] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Sys. Rev.*, 44(2):35–40, 2010.

[23] J. Lenstra, A. R. Kan, and P. Brucker. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[24] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the Progress of MapReduce Pipelines. In *ICDE*, 2010.

[25] J. Myllymaki and M. Livny. Relational Joins for Data on Tertiary Storage. In *ICDE*, 1997.

[26] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *Proc. VLDB Endow.*, 3:494–505, September 2010.

[27] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: Continuous Pig/Hadoop Workflows. In *SIGMOD*, 2011.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[29] E. Otoo, D. Rotem, and A. Romosan. Optimal File-Bundle Caching Algorithms for Data-Grids. In *SC*, 2004.

[30] Pig Performance Benchmark. https://issues.apache.org/jira/browse/PIG-200.

[31] S. Sarawagi. Query Processing in Tertiary Memory Databases. In *VLDB*, 1995.

[32] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. In *VLDB*, 2009.

[34] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and

D. Kossmann. Predictable Performance for Unpredictable Workloads. In *VLDB*, 2009.

[35] X. Wang, R. Burns, and T. Malik. LifeRaft: Data-Driven, Batch Processing for the Exploration of Scientific Databases. In *CIDR*, 2009.

[36] X. Wang, E. Perlman, R. Burns, T. Malik, T. Budavári, C. Meneveau, and A. Szalay. JAWS: Job-Aware Workload Scheduling for the Exploration of Turbulence Simulations. In *SC*, 2010.

[37] J.-B. Yu and D. J. DeWitt. Query Pre-Execution and Batching in Paradise: A Two-Pronged Approach to the Efficient Processing of Queries on Tape-Resident Raster Images. In *SSDBM*, 1997.

[38] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.