Re-optimizing Data-Parallel Computing

Sameer Agarwal[1,3], Srikanth Kandula[1], Nicolas Bruno[2], Ming-Chuan Wu[2], Ion Stoica[3], Jingren Zhou[2]
[1]*Microsoft Research*, [2]*Microsoft Bing*, [3]*University of California, Berkeley*

Abstract– Performant execution of data-parallel jobs needs good execution plans. Certain properties of the code, the data, and the interaction between them are crucial to generate these plans. Yet, these properties are difficult to estimate due to the highly distributed nature of these frameworks, the freedom that allows users to specify arbitrary code as operations on the data, and since jobs in modern clusters have evolved beyond single map and reduce phases to logical graphs of operations. Using fixed apriori estimates of these properties to choose execution plans, as modern systems do, leads to poor performance in several instances. We present RoPE, a first step towards re-optimizing data-parallel jobs. RoPE collects certain code and data properties by piggybacking on job execution. It adapts execution plans by feeding these properties to a query optimizer. We show how this improves the future invocations of the same (and similar) jobs and characterize the scenarios of benefit. Experiments on Bing's production clusters show up to 2× improvement across response time for production jobs at the 75th percentile while using 1.5× fewer resources.

1. INTRODUCTION

In most production clusters, a majority of data parallel jobs are logical graphs of map, reduce, join and other operations [5, 6, 21, 24].

An execution plan represents a blueprint for the distributed execution of the job. It encodes, among other things, the sequence in which operations are to be done, the columns to partition data on, the degree of parallelism and the implementations to use for each operation.

While much prior work focuses on executing a given plan well, such as dealing with stragglers at runtime [1], placing tasks [15, 25] and sharing the network [8, 23], little has been done in choosing appropriate execution plans.

Execution plan choice can alleviate some runtime concerns. For example, outliers are less bothersome if even the most expensive operation is given enough parallelism.

But plan choice can do much more– it can avoid needless work (for example, by deferring expensive operations till after simpler or more selective operations) and it can trade-off one resource type for another to speed up jobs (for example, in certain cases, some extra network traffic can avoid a read/write pass on the entire data set). However, plan choice is more challenging because the space of potential plans is large and also because the appropriateness of the plan depends on the interplay between code, data and cluster hardware.

Early data-parallel systems force developers to specify the execution plan (e.g., Hadoop). To shield developers from coping with these details, some recent proposals raise the level of abstraction. A few use hand-crafted rules to generate execution plans (e.g., HiveQL [24]) or use compiler techniques (e.g., FlumeJava [6]). Other declarative frameworks cast the execution plan choice as the traditional query optimization problem (e.g., SCOPE [5], Tenzing [7], Pig [21]).

A central theme, across all schemes, is the absence of insight into certain properties of the code (such as expected CPU and memory usage), the data (such as the frequency of key values), and the interaction between them (such as the selectivity of an operation). These properties crucially impact the choice of execution plans.

Our experience with Bing's production clusters shows that these code and data properties vary widely. Hence, using fixed apriori estimates leads to performance inefficiency. Even worse, not knowing these properties constrains plan choice to be pessimistic; techniques that provide gains in certain cases but not all cannot be used.

This paper presents RoPE[1], a first step towards re-optimizing data parallel jobs, i.e., adapting execution plans based on estimates of code and data properties. To our knowledge, we are the first to do so. The new domain brings challenges and opportunities. Accurately estimating code and data properties is hard in a distributed context. Predicting these properties by collecting statistics on the raw data stored in the file-system is not practical due to the prevalence of user-defined operations. But, knowing these properties enables a large space of improvements that is disjoint from prior work and so are the methods to achieve these improvements.

The sheer number of jobs indicates that the estimation and use of properties has to be automatic. RoPE piggybacks estimators with job execution. The scale of the data and distributed nature of computation means that no single task can examine all the data. Hence, RoPE collects statistics at many locations and uses novel ways to compose them. To keep overheads small, RoPE's collectors can only keep a small amount of state and work in a single pass over data. Collecting meaningful properties, such as the number of distinct values or heavy hitters, under these

[1] *Re*optimizer for *P*arallel *E*xecutions

constraints precludes traditional data structures and leads to some interesting designs.

The flexibility allowed for users to define arbitrary code leads to a much tighter coupling between data and computation in data parallel clusters. As long as they conform to well-defined interfaces, users can submit jobs with binary implementations of operations. In this context, predicting code properties becomes even more difficult. Traditional database techniques can project statistics on the raw data past some simple operations with alphanumeric expressions but doing so through multiple operations, more complex expressions, potentially dependent columns and user-defined operations introduces impractically large error [3]. Rather than predicting, RoPE instruments job execution to measure properties directly.

We find traditional work on adaptive query optimization to be specific to the environment of one database server [2, 3, 16] and the resulting space of optimizations. For example, a target scenario minimizes the reads from disk by keeping one side of the join in the server's memory. RoPE translates these ideas to the context of distributed systems and parallel plans. In doing so, RoPE uses a few aspects of the distributed environment that make it a better fit for adaptive optimization. Unlike the case of a database server where most queries finish quickly and the server has to decide whether to use its constrained resources to run the query or to re-optimize it, map-reduce jobs last much longer and the resources to re-optimize are only a small fraction of those used by the job. Further, if a better plan becomes available, transitioning from the current plan to the better plan is tricky in databases [16] whereas data parallel jobs have many inherent barriers at which execution plans can be switched.

In Bing's production clusters, we observe that many key jobs are re-run periodically to process newly arriving data. Such recurring jobs contribute 40.32% of all jobs, 39.71% of all cluster hours and 26.07% of the intermediate data produced. We also observe that the code and data properties are remarkably stable across recurring jobs despite the fact that each job processes new data. These jobs are RoPE's primary use-case.

RoPE adapts the execution plans for future invocations of such jobs by feeding the observed properties into a cost-based query optimizer (QO). Our prototype is built atop SCOPE [5], the default engine for all of Bing's map-reduce clusters, but our techniques can be applied to other systems. The optimizer evaluates various alternatives and chooses the plan with the least expected job latency. Additionally, we modified the optimizer to use the actual code and data properties while estimating costs. Working with a QO enables RoPE to not only perform local changes such as changing the degree of parallelism of operations, but also changes that require a global context, such as reordering operations, choosing appropriate operation implementations and grouping operations that have little work to do into a single physical task.

We find jobs that are not completely identical often have common *parts*. Further, during the execution of a job, while some of the global changes are logically impossible due to operations that have already executed, other changes remain feasible. RoPE can help in both these cases since (a) the query optimizer can work with incomplete estimates and (b) the code and data properties are linked to the sub-graph at which they were collected and can be matched to other jobs with an identical sub-graph.

Our contributions include:

- Based on experiences and measurements on Bing's production clusters, we describe scenarios where knowledge of code and data properties can improve performance of data-parallel jobs. A few of these scenarios are novel (§2).
- Design of the first re-optimizer for data-parallel clusters, which involves collecting statistics in a distributed context, matching statistics across sub-graphs and adapting execution plans by interfacing with a query optimizer (§3).
- Results from a partial prototype, deployed on production clusters, which show RoPE to be effective at reoptimizing jobs (§4, §5). Production jobs speed up by over 2× at the 75th percentile while using 1.5× fewer resources. RoPE achieves these gains by designing better execution plans that avoid wasteful work (reads, writes, network shuffles) and balance operations that run in parallel.

A user would expect her data-parallel jobs to run quickly. She would expect this even though the code is unknown, even though the data properties are hard to estimate, even though the code and data interact in unpredictable ways, and even though the code, the data and the cluster hardware and software keep evolving. RoPE is a first step towards reoptimizing data-parallel computing.

2. COST OF IGNORING CONTEXT

It is not uncommon for data-parallel computing frameworks such as Dryad and MapReduce to process petabytes of data each day. However, their inability to leverage data and computation statistics renders them unable to generate execution plans that are better suited for the jobs they run and prevents them from utilizing historical context to improve future executions. Here, we describe axiomatic scenarios of such inefficiency and quantify both their impact and frequency of occurrence.

2.1 Background

Our experience is rooted in Bing's production clusters consisting of thousands of multi-core servers participating in a distributed file system that supports

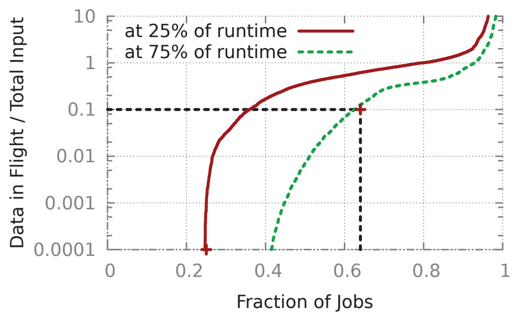


Figure 1: Data that remains in flight when a job has executed for 25% (or 75%) of its running time.

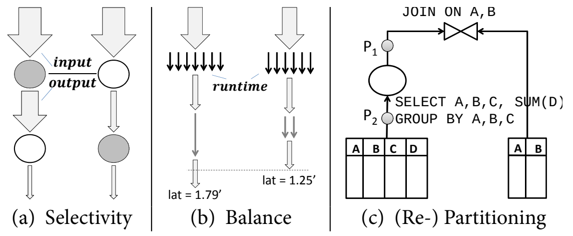
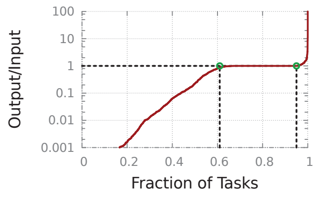


Figure 2: Motivating examples for re-optimizing data parallel computing



(a) CDF

Range	Fraction
0	3.7%
(0, .001)	12.9%
[.001, .1)	27.2%
[.1, .5)	11.3%
[.5, .9)	6.7%
[.9, 1]	32.8%
> 1	5.4%
> 2	.35%
> 10	.01%

(b) readout

Figure 3: Variation in selectivity across tasks

both structured and unstructured data. Jobs are written in SCOPE [5], a SQL-like mashup language with support for arbitrary user-defined operators. That is, users specify their data parallel jobs within a declarative framework (e.g., `select, join, group by`) but are allowed to declare their own implementations of operators as long as they fit the templates provided (e.g., `extractor, processor, combiner, reducer`). A compiler translates the query into an execution plan which is then executed in parallel on a Dryad-like [14] runtime engine. Plans are directed acyclic graphs where edges represent dataflow and nodes represent work that can be executed in parallel by many tasks. A task can consist of multiple operations. A job manager orchestrates the execution of the job's plan by issuing tasks when their inputs are ready, choosing where tasks run on the cluster, and reacting to outliers and failures. To facilitate better resource allocation across concurrent jobs, individual job managers work in close contact with a per-cluster global manager.

Unless otherwise specified, our results here use a dataset that contains all the events from a large production cluster in Bing. The events encode for each entity (job/ operation/ task/ or network transfer), the start and end times of the entity, the resources used, the completion status, and its dependencies with other entities. Our experiments are based on examining all events during the month of September 2011 on a cluster consisting of tens of thousands of servers.

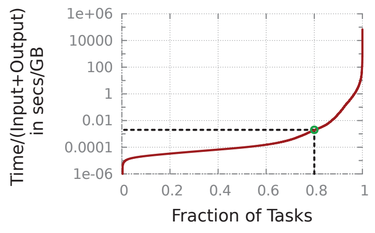
2.2 Little Data

We notice that while most map-reduce programs start off by reading a large amount of data, each successive operation (filters, reduces, etc.) produces considerably fewer output compared to its input. Hence, more often than not, just after a small number of these consecutive operations there is very little data left to process. We call this the *little data* case. The little-data observation can be used to optimize the tail of most jobs. In some cases, the degree of parallelism on many operations in the tail can be reduced. This saves scheduling overhead on the un-necessary tasks. In other cases, multiple operations in the tail can be coalesced into a single physical operation, i.e., one group of tasks executes these operations in parallel. This avoids needless checkpoints to disk. In yet other cases, broadcast joins can be used instead of pair-wise joins (see §2.6) thereby saving on network shuffle and disk accesses. The challenge however is that the reduction factors are unknown apriori and vary by several orders of magnitude.

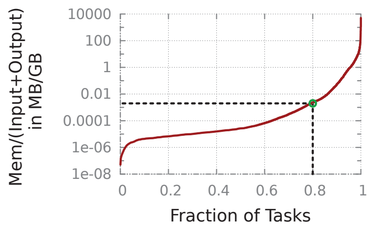
Fig. 1 plots the fraction of input data that remains *in flight* after jobs have been running for 25% (and 75%) of their runtime. We compute the data in-flight at any time by taking a cut of the job's execution graph at that time and adding up the data exchanged between tasks that are on either side of this cut. For convenience, we place tasks running at that time to the left of the cut. We see that while some jobs have more data in flight than their input (above $y = 1$ line), most of the jobs have much fewer. In fact for over 20% of jobs, the data in flight reduces to less than $\frac{1}{10^4}$ of their input within a quarter of the job's running time and over 60% of jobs have less than a tenth of data in flight after three quarters of their running time. This means that little data, and the above optimizations, can be brought to bear.

2.3 Varying Selectivity, Reordering

Consider a pair of commutative operations. Ordering them, so that the more selective operation (one with a lower output to input ratio) runs first will *avoid work* thereby saving compute hours, disk accesses and network shuffle. See Fig. 2(a), where the width of block arrows represent the data flow between a pair of commutative operators (indicated by the circles). Evidently, the plan on the right avoids processing unnecessary data and potentially saves significant cluster cycles by appropriately ordering the operators based on their selectivity. These pairs hap-



(a) Runtime/Data Processed



(b) Mem Usage/ Data Processed

Figure 4: Variation in Operation costs; in time to process and memory usage, per unit data processed

pen often, due to operations that are independent of each other (e.g. operations on different columns) or are commutative. Identifying these pairs can be hard in general but SCOPE's declarative syntax allows the use of standard database techniques to discover such pairs. Finding the selectivities of operations remains a challenge.

Standard database techniques to predict operator selectivity are hard to translate to map-reduce like frameworks due to the complexity of expressions and long sequences of operations. The selectivity of alphanumeric expressions (e.g. `select on X=30`) can be predicted by using clever histograms on the raw data (e.g. equi-depth) but creating these histograms requires many passes over the data. Predicting the selectivity for user-defined operations (e.g. `select when columnvalue.BeginsWith("http://bing")`) is an open problem [3]. We see such code in a majority of jobs. Moreover, the prediction errors grow exponentially with the length of the sequence of operations [3]. Computing more detailed synopsis on a random sample is often of only marginal benefit [18]. Finally, correlations between sets of columns, as is common, increases prediction error (e.g. `select on X=30 and Y=10` can produce just as much data as the `select on X=30` or much less). RoPE estimates selectivity by direct instrumentation.

Fig. 3(a) plots a CDF of the selectivity (ratio of output to input) of operations in our dataset. Note the y axis is in log scale. About 5% of operations produce more data than they consume (above $y = 1$). These are typically (outer) joins. About 34% have output roughly equalling input. The remaining 60% operations produce less output than their input but the selectivity varies widely– 17% produce fewer than $\frac{1}{1000}$'th of their input, and the coefficient of variation ($\frac{stdev}{mean}$) is 1.3 with a range from 0 to 171. This means that if these selectivities were available, there is substantial room to reorder operators.

2.4 Varying Costs, Balance

Suppose we figured out selectivities and picked the right order. Uniquely for data parallel computing, we need to choose the number of parallel instances for each operation. Choosing too few instances will cause that operation to become a bottleneck as per Amdahl's law. On the other hand, choosing too many leads to needless per-task scheduling, queuing and other startup overhead. *Balance*, i.e., ensuring that each operation has enough parallelism and takes roughly the same amount of time, can improve performance significantly [22]. See Fig. 2(b) where block arrows again represent the dataflow and the thin arrows now represent the tasks in each of the two operations. Here, using one less task for the upstream operation and one more for the downstream operation reduces job latency by over 30%.

Achieving balance in the context of general data parallel computing is hard because the costs (runtime, memory etc.) of the operators are unknown apriori. These costs depend on the amounts of data processed, the types of computation performed and also on the type of data. For example, a complex sentence can take longer to translate than a simpler one of the same length. Even worse, *late-binding*, i.e., deferring the choice of the amount of parallelism to the runtime is hard because local changes have global impact; for example, the number of partitions output by the map phase restricts the maximum number of reduce tasks at the next stage. RoPE estimates these costs to generate balanced execution plans.

Fig. 4(a) plots a CDF of the runtime per unit byte read or written by operations in the dataset. The analogous plot for memory used by tasks is in Fig. 4(b). Note again that the y axes are in log scale. While as variable as their selectivity– the middle 50^{th} percent of operators have costs spread over two orders of magnitude– we find that operator costs skew more towards higher values. Per unit data processed, 20% of operations take over 100X more time and memory than the average over the remaining operations. It is crucial to identify these heavy operations, to offset their costs by increasing the parallelism and for the case of memory, to place tasks so that they do not compete with other memory hungry tasks.

A consequence of unpredictable data selectivity and operator costs is the lack of balance. We find that our compiler both underestimates and overestimates an operation's work. Fig. 5 plots a CDF of the runtime of the median task in each stage. The y axis is in log scale. The median task in a stage is unlikely to be impacted by failures or be an outlier. So, if the compiler apportions parallelism well, the median task in each stage should take

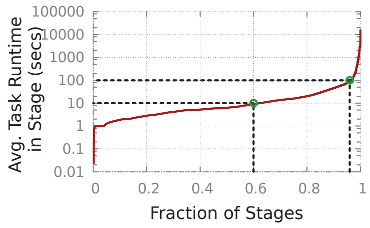


Figure 5: Imbalance

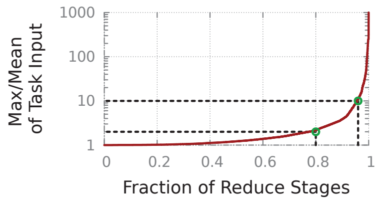


Figure 6: Skewness

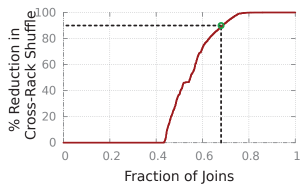


Figure 7: Replacing Pair-wise Joins with Broadcast Joins

about the same time. This duration could be chosen to trade-off fault-tolerance vs. the cost of checkpointing to disk. However, we see that while roughly 60% of all tasks finish within 10 seconds, 4% take over 100s with the last 1% taking over 1000s. We found the tail dominated by tasks with user defined operators. Setup and scheduling overheads outweigh the useful work in short-lived tasks whereas the long-lived tasks are bottlenecks in the job.

2.5 Partition Skew and Repartitioning

A key to efficient data-parallel computing is to avoid skews in partition and to re-partition only when needed. Consider the example in Fig. 2(c). The naive implementation would partition the data twice– once on A, B, C (at P_1), followed by a network shuffle and a reduce to compute the sum, and then again on A, B (at P_2) followed by another shuffle for the join. It is tempting to just partition the data once, say on A, B to avoid the network shuffle and the pass over data. Note that partitioning on fewer keys does not violate the correctness of the reduce that computes SUM(D). Each reduce task will now compute many rows, one per distinct value of C, rather than just the one row they would have produced were the map to partition on all three columns. However, if there is not enough entropy on A and B, i.e., only a few distinct values have many records, then partitioning on the sub-group can make things worse. A few reduce tasks might receive a lot of data to process while other reduce tasks have none, and the overall parallel execution can slow down.

Fig. 6 estimates how skewed the partitions can be in our cluster. Note that our compiler is conservative and does not partition on sub-groups to avoid re-partitioning. Yet significant skew happens. We define skew as the ratio of the maximum data processed by a reduce task to the average over other tasks in that reduce phase. We see that in 20% of the reduce stages, the largest partition is twice as large as the average and in about 5% of the stages the largest partition is over ten times larger than the average. Such skew causes unequal division of work and bottlenecks but can be avoided if the data properties are known.

2.6 From operations to implementations

Often, the same operation can be implemented in several ways. While choosing the appropriate implementation can result in significant improvements, doing so requires appropriate context that is not available in today's systems. For example, consider `Join`. The default implementation `PairJoin`, involves a map-reduce operation on each side that partitions data on the join columns. This causes three read/write passes on each of the sides and at least one shuffle each across the network. However, if one of the sides is smaller, perhaps due to the *little data* case (§2.2), one could avoid shuffling the larger side and complete the join in one pass on that side. The trick is to broadcast the smaller side to each of the tasks that is operating in parallel on the larger side. The problem though is that when used inappropriately, a `BroadcastJoin` can be even more expensive than a `PairJoin`.

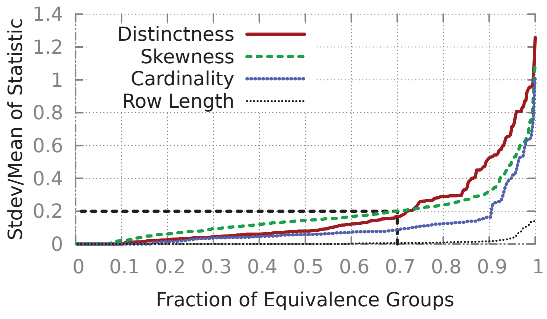


Figure 8: Stability of data properties across recurring jobs

Fig. 7 plots the potential benefits of replacing `PairJoins` with `BroadcastJoins`. It shows the amount of data shuffled in either case. We see that about 40% of joins in the dataset would see no benefit. This can happen if both join inputs are considerably large and/or when the parallelism on the larger side is so much that broadcasting the smaller input dataset to too many locations becomes a bottleneck. However, off the remaining joins, the median join shuffles 90% less data when using broadcast joins.

2.7 Recurring Jobs

We find that many jobs in the examined cluster repeat and are re-run periodically to execute on the newly arriving data. Such recurring jobs contribute to 40.32% of all jobs, 39.71% of all cluster hours and 26.07% of intermediate data produced. If the extracted statistics are stable per recurring job, i.e., the operations behave statistically similar to the previous execution when running on newer data of the same stream, then RoPE's instrumentation would suffice to re-optimize future invocations.

Fig. 8 plots the average difference between statistics collected at the same location in the execution plan across recurring jobs. We picked all of the recurring jobs from one business group and instrumented five different runs of each job. While most of these jobs repeated daily, a few repeated more frequently. The figure has four distributions, one per data property that RoPE measures. We defer details on the specifics of the properties (see Table 1, §3.1) but note that while some properties, such as row length, are more predictable than others, the overall statistics are similar across jobs– the ratio $\frac{stdev}{mean}$ is less than 0.2 for 70% of the locations.

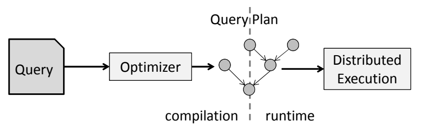
2.8 Current Approaches, Alternatives

To the best of our knowledge, we are the first to re-optimize data parallel jobs by leveraging data and computation statistics. Current frameworks use best-guess estimates on the selectivity and costs of operations. Such rules-of-thumb, as we saw in §2.2–§2.6, perform poorly. Hadoop and the public description of MapReduce leave the choice of execution plans, the number of tasks per machine and even low-level system parameters such as buffer sizes to the purview of the developer. HiveQL [24] uses a rule-based optimizer to translate scripts written in a SQL-like language to sequences of map-reduce operations. Starfish [12] provides guidance on system parameters for Hadoop jobs. To do so, it builds a machine learning classifier that projects from the multi-dimensional parameter space to expected performance but does not explore semantic changes such as reordering. Ke et. al. [17] propose choosing the number of partitions based on operator costs. In contrast, RoPE can perform more significant changes to the execution plan, similar to FlumeJava [6] and Pig [21], but additionally does so based on actual properties of the code and data.

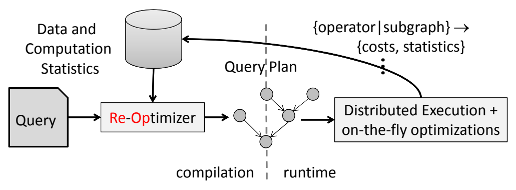
It is tempting to ask end-users to specify the necessary context, for e.g., tag operations with cost and selectivity estimates. We found this to be of limited use for a few reasons. First, considerable expertise and time is required to hand-tune each query. Users often miss opportunities to improve or make mistakes. Second, exposing important knobs to a wide set of users is risky. Unknowingly, or greedily, a user can hog the cluster and deny service to others; for instance, by tricking the system to give her job a high degree of parallelism. Finally, changes in the script, the cluster characteristics, the resources available at runtime or the nature of data being processed can require re-tuning the plan. Hence, RoPE re-optimizes execution plans by automatically inferring these statistics.

2.9 Experience Summary, Takeaways

Note that all of the problems described here happen deterministically. They are not due to heterogeneity or runtime variations [1] or due to poor placement of tasks [15, 25] or due to sharing the cluster [13, 23]. We believe that



(a) Current Cluster(s)


Data and Computation Statistics
{operator|subgraph} → {costs, statistics}
Query Plan
Query
Re-Optimizer
Distributed Execution + on-the-fly optimizations
compilation
runtime

(b) With RoPE

Figure 9: RoPE's architecture for re-optimization

the choice of execution plan is orthogonal to these problems that arise *during* the execution of a plan. The underlying cause is that predicting relevant data and computation statistics deep into a job's execution is necessary to find a good plan but is difficult due to the intricate coupling between data and computation.

When employed together, these improvements add up to more than their sum. Promoting a more selective operator closer to the input, can reduce the data flowing in so much that a subsequent join may be implemented with broadcast join. We found in practice that such global changes to the execution plan accrue more benefits than making singleton changes. RoPE achieves both types of re-optimizations as we will see in §3.

A few key takeaways follow.

- Being un-aware of data and computation context leads to slower responses and wasted resources.
- Unlike the case of singleton database servers, data-parallel computation provides different space for improvements and has new challenges such as coping with arbitrary user defined operations and expressions.
- Global changes to the execution plan add more value than local ones.

3. DESIGN

RoPE enables re-optimization of data parallel computing. To obtain data- and computation- context, RoPE interposes instrumentation code into the job's dataflow. An operation can be instrumented with collectors on its input(s), output(s) or both. We describe how RoPE chooses what information to collect, avoids redundancy in the locations from which stats are collected, and the algorithms to compose statistics from distributed locations in §3.1. These statistics are funneled to the job manager and are used to improve execution plans in a few different ways, each differing in the scope of possible changes and the complexity to achieve those changes.

Even though an execution plan is already chosen for a running job, RoPE uses the statistics collected during the run to improve some aspects of the job. Descendant stages that are at least a barrier away from the stages where datastats are being collected will not have begun execution. The implementation of these stages can be changed on the fly. Stages that are pipelined with the currently executing stage can be changed, since inter-task data flow happens through the disk. Some plan changes may be constrained by stages (or parts of stages) that have already run. Hence RoPE performs changes that only impact the un-executed parts of the plan, such as altering the degree of parallelism of non-reduce stages.

RoPE uses the collected statistics to generate better execution plans for new jobs. Here, RoPE can perform more comprehensive changes. Recall from §2.7 that many jobs in the examined cluster recur because they periodically execute on new data and that the extracted datastats are stable across runs of these jobs. In this case, upon a new run of a recurrent job, datastats collected from previous runs are used as additional inputs to the plan optimizer. We describe how statistics are stored so they can be matched with expressions from subsequent jobs in §3.2. We also note that partial overlaps are common among jobs [11] and our matching framework extends to cover this case. The methodology of how the statistics are used is described in §3.3. An illustrative case study of the changes that RoPE achieves is in §5.2.

3.1 Collecting contextual information

Choosing what to observe and the statistics to collect has to be done with care since the context we collect will determine the improvements that we can make. Broadly, we collect statistics about the resource usage (e.g., CPU, memory) and data properties (e.g., cardinality, number of distinct values). See Table 1 for a summary.

The nature of map-reduce jobs leads to a few unique challenges. First, map-reduce jobs examine a lot of data in a distributed fashion. There is no instrumentation point that observes all the data and even if created, such instrumentation would not scale with the data size. Hence, we require stat collection mechanisms to be *composable*, i.e., local statistics obtained by looking at parts of the data should be composable into a global statistic over all data. Second, to be applicable to a wide set of jobs, the collection method should have *low overhead*, i.e., overhead that is only a small fraction of the task that it piggybacks upon. In particular, the memory used should scale sub-linearly with data size and to limit computation cost, the statistics should be collected in a single pass. Together, these constraints are quite strict, so we adapt some pre-existing streaming and approximation algorithms.

Finally, to be useful, the statistics have to be *unambiguous*, *precise*, and have *high coverage*. By unambiguous, we mean that the statistics should contain metadata describing the location in the query tree that these statistics were collected at. Global changes to the plan, during re-optimization for subsequent jobs for example, can alter the plan so much that previously observed subtrees no longer occur. Precision is an accuracy metric; we try to match the accuracy requirements of the improvements that we would like to make with the properties of the algorithmic techniques that we use to collect statistics. For coverage, we would like to observe as many different points in the job execution as possible. However, to keep costs low, we ignore instrumenting operations whose impact on the data is predictable (e.g., the input size of a sort operation is the same as its output). Further, we only look at the interesting columns. That is, we collect column-specific statistics only on columns whose values will, at some later point in the job, be used in expressions (e.g. record stats for `col` if `select col=...`, `join by col=...`, or `reduce on col` follow).

Type	Description	Granularity
Data Properties	Cardinality	Query Subgraph
	Avg. Row Length	Query Subgraph
	# of Distinct values	Column @ Subgraph
	Heavy hitter values, their frequency	Column @ Subgraph
Code Properties	CPU and Memory Used per Data read and written	Task
Leading Statistics	Hash histogram	Column @ Subgraph
	Exact sample	Column @ Subgraph

Table 1: Statistics that RoPE collects for reoptimization

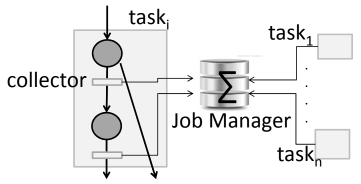


Figure 10: A task can have many operations and hence, collectors. A job manager composes individual statistics.

Implementation: RoPE interposes stat-collectors at key points in the job's dataflow. Datastat collectors are pass-through operators which keep negligible amounts of state and add little overhead. We also extend the task wrapper to collect the resources used by the task. When a task finishes, all datastats are ferried to the job manager (see Figure 10), which then composes the stats. The stats are used right away and also stored with a matching service for use with future jobs (see Figure 9(b)).

3.1.1 Data Properties

At each collection point we collect the number of rows and the average row length. This statistic will inform whether data grows or shrinks and by how much as it flows through the query tree. Composing these statistics is easy–for example, the total number of rows after a `select` operation is simply the sum of the number of rows seen by the collectors that observe the output of that `select` across all the tasks that contain that `select`.

Further, for each interesting column, we compute the number of distinct values and the heavy hitters, i.e., values that repeat in a large fraction of the rows. These statistics, as we will see shortly, help avoid skews during partition and also help pick better implementations. Computing these statistics while keeping only a small amount of state in a single pass is challenging, let alone the need to compose across different collection points.

Our solution builds on some state-of-the-art techniques that we carefully chose because we could extend them to be composable. We will only sketch the basic algorithms and focus on how we extended them.

Lossy Counting to find Heavy Hitters.

Suppose we want to identify all values that repeat more frequently than a threshold, say 1% of the data size N. Doing this in one pass, naively, requires tracking running counts of all distinct values and uses up to $O(N)$ space.

Lossy counting [19] is an approximate streaming algorithm, with parameters s, ϵ that has these properties. First, it guarantees that all values with frequency over sN will be output. Further no value with a frequency smaller than $(s-\epsilon)N$ will be output. Second, the worst case space required to do so is (sub-linear) $\frac{1}{\epsilon}\log(\epsilon N)$. In practice, we find that the usage is often much smaller. Third, the frequency estimated undercounts the true frequency of the elements by at most ϵN. The key technique is rather elegant; it tracks running frequency counts but after every $\lceil \frac{1}{\epsilon} \rceil$ records, it retires values that do not pass a test on their frequency. For more details, please refer [19].

RoPE uses a distributed form of lossy counting. Each stat collector employs lossy counting on the subset of data that their task observes with parameters $s = 2\epsilon, \epsilon$. To compute heavy hitters over all the data, we add up the frequency estimates over all collectors and report distinct values with count greater than ϵN. Interestingly, composing in this manner retains the properties of lossy counting with slight mods. A proof sketch follows.

Proof Sketch: Let N_i be the number of records observed by the i'th collector. Note that $s-\epsilon = \epsilon$ and $\sum N_i = N$ First, for the frequency estimation error, if a value is reported by stat collector i, we know that its frequency estimate is no worse off than ϵN_i. If the value is not reported, its frequency estimate is zero; but by the existence constraint, we know that the element did not occur more than $2\epsilon N_i$ times at this collector. Summing up the errors across collectors, we conclude that the global estimate is not off the true frequency by more than $2\epsilon N$. Second, for false positives, we note that we only keep values whose cumulative recorded count is greater than ϵN, that means their true frequency is at least ϵN. Third, for false negatives, suppose a value has a global frequency greater than $f > 3\epsilon N$, then it has to occur more than $3\epsilon N_i > sN_i$ times at some collector, and so will be reported. Even more, since we just showed that the cumulative error is no worse than $2\epsilon N$, its cumulative recorded count will be no worse than $f - 2\epsilon N$ which is larger than ϵN, hence RoPE will report this value after composition. Finally, the space used at each collector is $\frac{1}{\epsilon}\log(\epsilon N_i)$ meeting our requirements.

Implementation: RoPE uses $\epsilon = .01$. Micro-benchmarks show that frequency estimates are never worse off by more than ϵN and the space used is small multiples of $\log(\frac{N}{\epsilon})$.

Hash Sketches to count Distinct Values.

Counting the number of distinct items in sub-linear state and in a single pass has canonically been a hard problem. Using only $O(\log N)$ space, hash sketches [9] computes this number approximately. That is, the estimate is a random variable whose mean is the same as the number of distinct values and the standard deviation is small.

The key technique involves uniformly random hashing the values. The first few bits of the hash value are used to choose a bit vector. From the remaining bits, the first non-zero bit is identified and the corresponding bit is set in the chosen bit vector. Such a hash sketch estimates the number of distinct values because $\frac{1}{2}$ of all hash-values will be odd and have their first non-zero bit at position 1, $\frac{1}{2^2}$ will do so at position 2 and so on. Hence, the maximum bit set in a bit-vector is proportional to the logarithm of the number of distinct values. Using a few bit-vectors rather than one guards against discretization error. The actual estimator is a bit more complex to correct for additive bias. For more details, please refer [9].

RoPE uses a distributed form of hash sketches. Each stat collector maintains local hash sketches and relays these bit vectors to the job manager. The job manager maintains global bit vectors, such that the i^{th} global bit vector is an OR of all the individual i^{th} bit vectors. By doing so, the global hash sketch is exactly the same as would be computed by one instrumentation point that looks at all data. Hence, RoPE retains all the properties of hash sketches.

Implementation: If the hash values are h bits long, where $h = O(\log N)$, and the first m bits choose the bit-vector, then there are 2^m bit vectors and the size of the hash sketch is $h * 2^m$ bits. RoPE uses $m = 6$ and $h = 64$. Hence, each task's hash sketch is 512B long. Our micro-benchmarks show that hash sketches retain precision even as the number of distinct values grows to 2^{40}.

3.1.2 Operation Costs

We collect the processing time and memory used by each task. A task is a multi-threaded process that reads one or more inputs from the disk (locally or over the network) and writes its output to the local disk. However, popular data-parallel frameworks can combine more than one operation into the same task (e.g. data extraction, decompression, processing). Since such operations run within the same thread and frequently exchange data via shared memory, the per-operator processing and memory costs are not directly available. But, per-operator costs are necessary to reason about alternate plans, e.g., reordering operators. Program analysis of the underlying code could reason about how the operators interact in a task, but this analysis can be hard because the interactions are complex, e.g., pipelining. Also, profiling individual operators does not scale to the large number of UDOs. Instead, RoPE uses a simple approach that only estimates the costs of the more costly operations.

The approach works as follows. First, tasks containing costly operations are likely to be costly as well. We pick stages with costs in the top tenth percentile as *expensive*. We only use the costs of the median task in each stage to filter the impact from failures and other runtime effects. Second, not all the operators in expensive tasks are expensive. So, for each operator, we compute a *confidence* score as the fraction of the stages containing the operator that have been picked as expensive. An operator will have a high confidence score only if it exclusively occurs in expensive tasks. Third, we compute the *support* of an operator as the number of distinct expensive stages that it occurs in. Finally, we estimate the cost of operators, that have high confidence and high support scores, as the average cost of the stages containing that operator.

To validate this approach, we profiled over 200K randomly chosen stages from the production cluster. It is hard to obtain ground truth at this scale. Hence, we corroborate our results on *succinctness* – only a few among all the operations should be expensive, and *coverage* – most of the expensive tasks should contain at least one expensive operation. This method identified 22.3% and 15.6% of the operators as expensive for CPU and memory costs respectively. This number was higher than expected because there are only a few generic operations but many UDOs. Most of the costly operations were UDOs. Further, 87.9% (92.4%) of the tasks picked as expensive based on their memory (cpu) cost contained at least one expensive operator. Hence, their cost can be explained by these operations. Very few contained more than one expensive operation. The succinct set of operations identified as expensive and the high coverage of this set makes us optimistic about this simple method. Fig 11 plots the relative cost values attributed to the expensive operations.

3.1.3 *Leading Statistics*

The ability to predict behavior of future operators is invaluable, especially for on-the-fly optimizations. Doing so precisely is hard. Rather than aspiring for precise predictions in all cases, RoPE collects simple leading statistics to help with typical pain points. We collect histograms and random samples on interesting columns (i.e. columns which are involved in where/group by clauses or joins) at the earliest collection points preceding the operator at which those columns will be used.

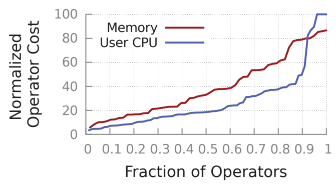


Figure 11: Normalized memory and user CPU costs for the operators identified as expensive.

We looked at several histogram generators, including equi-width, equi-depth and serial but ended up with a simpler, albeit less useful alternative. This is because none of the others satisfied our single pass and bounded memory criterion with a reasonable accuracy. For each interesting column, in an operator, we build a hash-based histogram with B buckets, that is hashed on a given column value (hash[column value] % $B \rightarrow$ bucket) and counts the frequency of all entries in each bucket. RoPE uses $B = 256$.

We also use reservoir sampling to pick a constant sized, random sample of the data flowing through the operator. For each interesting column, RoPE collects up to 100 samples but no more than 10KB.

3.2 Matching context to query expressions

As metadata to enable matching, with each stat collector we associate a hash-value that captures the location of the collector in the query graph. In particular, location in the query graph refers to a signature of the input(s) along with a topologically sorted chain of all the operators that preceded the stat collector on the execution plan. We colloquially refer to this as the query subgraph. RoPE uses 64 bit hashes to encode these query subgraphs.

3.3 Adapting query plans

We build on top of SCOPE Cloud Query Optimizer (CQO) which is a cost-based optimizer. CQO translates the user-submitted script into an expression tree, generates variants for each expression or group of expressions and finally chooses the query tree with the lowest cost. However, lacking direct knowledge of query context, the CQO uses simple apriori constants to determine the costs and selectivities of various operators as well as the data properties. By providing exactly this context, RoPE helps the CQO make better decisions.

RoPE imports statistics uses a stat-view matching technique similar to the analogous method in databases [4, 10, 18]. Statistics are matched in during the exploration stage in optimization, i.e. before implementing physical alternatives but after equivalent rewrites have been explored. The optimizer then propagates the statistics offered at one location to equivalent semantic locations, e.g., cardinality of rows remains the same after a sort operation and can be propagated. For expressions that have no direct evidence available, the optimizer makes-do by propagating statis-

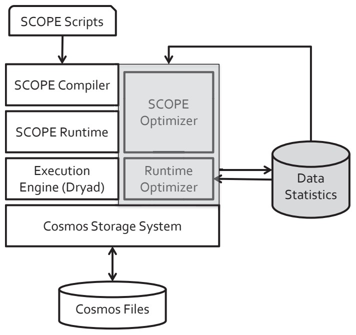


Figure 12: The RoPE prototype consists of 3 key components: a distributed statistics collection module, a pre-processor to the existing SCOPE compiler that provides matching functionality and a basic on-the-fly runtime optimizer.

tics from nearby locations with apriori estimates on the behavior of operators that lie along the path. Such uncertain estimates are deemed to be of lower quality and are used only when other estimates are unavailable.

We extended the optimizer to make use of these statistics. Cardinality, i.e., the number of rows observed at each collection point, helps estimate operator selectivity and compare reordering alternatives. Along with selectivity, the computation costs of operations are used to determine whether an operation is worth doing now or later when there is less work for it. Costs also help determine the degree of parallelism, the number of partitions, and which operations to fit within a task. Besides the choice of broadcast join, statistics also help decide when self-join or index-join are appropriate. Most of these optimizations are specific to the context of parallel executions. We believe that there is more to do with statistics than what RoPE currently does, but our prototype (§4) suffices to provide substantial gains in production (§5).

4. PROTOTYPE

The prototype collects all the statistics described in §3. Data stats are written to a distributed file system and the matching functionality (§3.2) runs as a pre-processing step of the job compiler. The statistics collection code is a few thousand lines of C#, which during code generation for each operator, gets placed into the appropriate location in the operator's dataflow. Note that not all operators collect statistics, and even when they do, they do not collect all types of statistics. Collected statistics are passed to the C++ task wrapper via reflection which piggybacks them along with task status reports to the job manager. Composing statistics required a few hundred lines of code in the job manager.

We allow the compiler to specify varying requirements across the columns. The constants also can change, to trade-off costs for improved precision, based on previous statistics or other information that the compiler has such as required accuracy of an estimate.

Parts of the code are production quality to the extent that all of our results here are from experiments that run in Bing's production clusters. Rather than implementing every optimization possible with the statistics that RoPE collects, we built a subset up to production quality code in order to deploy, run and gain experiences from Bing's production clusters. We acknowledge that our prototype is just that, and there is more benefit to be achieved.

Using these statistics required extensive changes in the SCOPE query optimizer involving several hundred lines of code spread over several tens of files.

5. EVALUATION

We built and deployed RoPE on a large production cluster that supports the Bing search engine at Microsoft.

5.1 Methodology

Cluster: This cluster constitutes of tens of thousands of 64-bit, multi-core, commodity servers; processes petabytes of data each day and is shared across many business groups. The core of the network is moderately oversubscribed and hence shuffling data across racks remains more expensive than transfers within a rack.

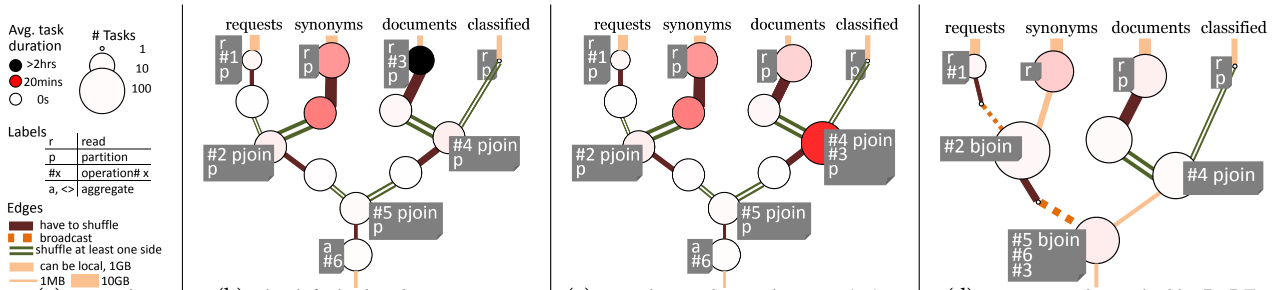
Workload Evaluated: We present results from evaluating RoPE on all the recurring jobs of a major business group at Bing and randomly chosen jobs from ten other business groups. The dataset has over 80 jobs. We repeat each job over ten times for each variant that we compare. The only modification we do to the jobs is to pipe the output to another location in the distributed file system so as to not pollute the production data. Though small, since we pick a large set of jobs, our dataset spans a wide range of characteristics. Latency of the unmodified runs varied from minutes to several hours. Tasks ranged from small tens to hundreds of thousands. The largest job had several tens of stages. User-defined operations are prevalent in the dataset, as is common across our cluster.

Compared Variants: For each job, we compare the execution plan generated by RoPE with the execution plan generated without RoPE. The latter is a strong strawman, since it uses apriori fixed estimates of operator costs and selectivities and is the current default in our production clusters. Early results from a variant that was equivalent to Hive over Hadoop, i.e., one that translated users' jobs into literal map-reduce execution plans, were significantly worse than our cluster's baseline implementation. Here, we omit those results.

Metrics: Our evaluation metrics are the reduction in job completion time and resource usage. Resources include cluster slot occupancy computed in machine-hour units, as well as the total amount of data read, written and moved across low bandwidth (inter-rack) network links.

5.2 A Case Study

Many plan changes can be performed given the statistics that RoPE collects. We report a case study that il-



(a) Legend (b) The default plan due to SCOPE (c) Reordering the costly UDO (#3) to a location with less data (d) Execution plan picked by RoPE

Figure 13: Case Study: Evolution of the execution plan as RoPE provides more statistics. See Table 2 for how well these plans do upon execution.

lustrates some of the changes to execution plans that achieved significant gains in practice. Aggregate results from applying RoPE to a wide variety of jobs are in §5.3.

Consider a job that processes four datasets. Let these be requests (R), synonyms (S), documents (D) and classified URLs (C). The goal is to compute how many requests (of a certain type) access documents (of a certain other type). Doing so, involves the following five operations:

1. Filter requests (R) by location. This operation has a selectivity of $\frac{1}{2000}x$.
2. Join requests (R) with synonyms (S). There are many synonyms per word, so the selectivity is $50x$.
3. Apply a user-defined operation (UDO) to documents (D). The selectivity is $\frac{1}{8}x$ but has a very large CPU cost per document.
4. Join documents (D) with the list of classified URLs (C). This has a selectivity of $\frac{1}{10}x$.
5. Join synonymized requests with documents containing classified URLs; has a selectivity of $\frac{4}{5}x$.
6. Finally, count number of requests per document URL.

The dataset sizes are R: 3GB, S: 18GB, D: 12GB, C: 160MB.

Figure 13(b) shows the plan computed by the unmodified optimizer which uses apriori estimates on data properties and costs. Each circle represents a stage, a collection of tasks that execute one or more operations which are listed in the adjacent label (see legend in 13(a)). Unlabeled stages are aggregates [14]. The size of the circle denotes the number of tasks allocated to that stage, in logarithmic scale. The color of the circle in linear *red*scale denotes the average time per task in that stage, darker implies longer. Figure 13(d)(13(c)) show the plan generated using (a subset of) the statistics provided by RoPE. We explain the figures as we go along. Table 2 shows how well these plans do when executed, the metrics are averaged over five different runs.

Recall that the compiler sets costs proportional to the data processed and sets selectivity based on the type of the operation– for instance, filters are assumed to be more selective than joins. These apriori estimates have mixed results. They pick the right choice for the operation on requests; the more selective filter operation (#1) is done before joining with synonyms (#2) (see Fig. 13(b) top left). However, for documents, the plan performs the user-defined filter (UDO) (#3) before the more selective join (#4) leading to a lot of wasted work. As we see in Table 2, due to the high cost of the UDO, this plan takes over $17x$ the time of the next best alternative.

By providing an estimate of the UDO's selectivity, RoPE lets the compiler join the documents dataset with the classified dataset first. Figure 13(c) depicts such a plan (see change in middle right). From Table 2, we see that since the UDO is applied on fewer data, the median execution time improves substantially. And, not many more tasks are needed since fewer documents through the UDO means fewer net work, and so the cluster hours decrease as well. But, by performing the UDO later, more data is shuffled across the network, since the join with classifieds is done on unfiltered documents.

When comparing these two plans, note that the thickness of the edges represent data volume moving between stages in logarithmic scale (see legend). Also, the color indicates the type of data movement. Dark solid lines denote data flow from a partition stage to an aggregate stage (many to many) which has to move over the network. Light solid lines indicate one-to-one data flow. Here, data-local placement can avoid movement across the network. Dotted lines indicate broadcast, i.e., the source stage's output is read by every task in the destination stage. Double edges indicate two source tasks per destination task, i.e., at least one of the sides has to be moved over the network.

With the UDO at a better place, the bottleneck moves to two new places. The average task duration of the stage with the UDO is very high (deep red). If the compiler knew the cost of the UDO, it could apportion more tasks to this stage to resolve this bottleneck. Similarly, even though requests (R) becomes small after applying the very selective filter (#1), the compiler is unaware and picks a pair-wise join for #2. This causes the large dataset S to be

Alternate Execution Plans	Performance				
	Latency (s)	Cluster Occupancy (s)	Cross Rack Shuffle (GB)	Reads+Writes To Disk (GB)	Tasks
Un-modified (Fig.13(b))	1x	1y	21.23	91.96	234
Push UDO to appropriate location (Fig. 13(c))	.057x	.201y	30.35	125.68	236
+ Replace pair-wise join with broadcast (not shown)	.016x	.227y	13.29	94.05	211
+ Balance (not shown)	.008x	.215y	12.82	90.75	341
RoPE (+push UDO even lower, little data,Fig. 13(d))	.006x	.139y	15.98	84.04	366

Table 2: Summarizes salient features of executing a typical job with and without RoPE. Overall, with RoPE latency reduces by almost 160X, while using 7X fewer cluster hours.

partitioned and shuffled across the network needlessly.

Figure 13(d) shows the plan where RoPE provides the compiler with the selectivity and costs of all the operations. Intermediate plans have been omitted for brevity. Table 2 shows that the median execution of this plan improves by another $9.5x$. A few changes are worth noting.

First, operation #2 is now implemented as a broadcast join. This not only saves cross rack shuffle and reads/writes to disk but also avoids partitioning both these datasets. Since pair-wise joins shuffle data across the network, such stages are more at risk from congestion induced outliers. We observed this with the default plan.

Second, given the high cost of the UDO (#3), the compiler instead of adding more tasks to the stage with operation #4 defers the UDO till even later, i.e., till after operations #5 and #6. Both these operations are net data reductive, however #6 is particularly so since it produces one row for each document url that is in the eventual output. The increase in cost from performing other operations on more data was lower than the benefits from performing the UDO on fewer data.

More optimizations are enabled recursively. The compiler realizes that both sides of the input for the join in operation #5 are small due to the cumulative impact of earlier operations, leading to the third improvement– implement operation #5 also as a broadcast join.

This leads to a fourth improvement that is subtle. Notice that operation #6, a *reduce* operation that needs data to be partitioned by document url to compute the number of times each URL occurs, now lies between operations #5 and #3 thereby eliminating one complete map-reduce!

To understand why, note that the join in operation #5 matches words in synonymized requests with words in the classified documents. Typically this join would require both inputs to be partitioned on words. However, here there is so little data that the left side (requests) can be broadcast and the right side (documents) can be partitioned in any way. Hence the right side is partitioned on url immediately after it is read to facilitate operation #6 and never re-partitioned thereby providing for coalescing many more operations into the same stage. The enabler for this optimization is *little data*– the later parts of most data parallel jobs can benefit from serial plans. Achieving the change, however, requires reasoning about the entire execution plan and not one stage at a time which is possible in RoPE due to the interface with a query optimizer.

Fifth, to offset the cost of the UDO, the compiler assigns more tasks (larger size) to the stage that now implements the UDO (along with operations #5 and #6). Doing so, is again a global change because the earlier operations have to partition the data enough ways. To benefit from not re-partitioning, the compiler implements the join in #4 with the same (larger) amount of parallelism.

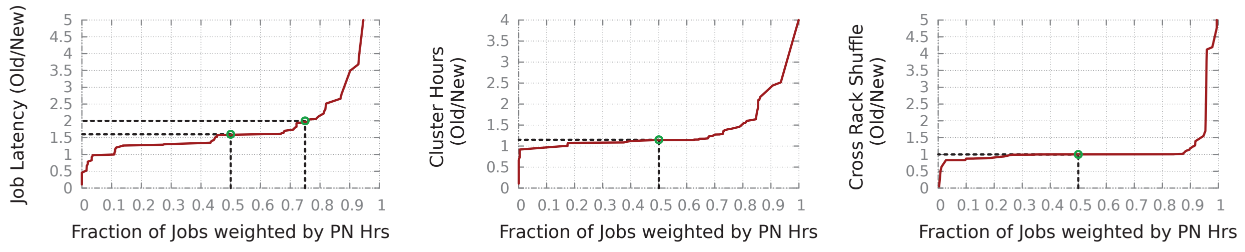
Finally, a potential change that was not performed is worth discussing. Since the classifieds dataset is small (C), it is possible to implement operation #4 also as a broadcast join. But the compiler chooses to not do that. The reason is that the documents have to be partitioned by url to facilitate the reduce in op #6. Either they are partitioned before the join op #5 or after. If the partition happens before, as is the case with the chosen plan, the large amount of work in the stage doing ops #5, #6 and #3 needs more parallelism resulting in more partitions, i.e., more tasks in op #4. But, the network costs of broadcasting C grow linearly with the number of tasks in op #4 offsetting the savings which is one additional read/write of C. Partitioning after op #5 would mean one more map-reduce on the critical path and just before the end of the job. Any outliers here would directly impact the job unlike outliers on the shuffle before op #4 which is not on the critical path since more work lies on the left side of the DAG.

Overall, we see that significant reductions result from optimizations building on top of each other. Unfortunately, the space of optimizations is not monotonic; some times using more tasks is better, whereas at other times pushing a filter after some operators but before some others is the best choice. By providing accurate estimates of code and data properties, RoPE is a crucial first step towards picking appropriate execution plans.

5.3 Aggregate results

Here, we present aggregated results across jobs in the dataset. These are runs of production jobs in a production cluster, so the noise from other jobs and other traffic on the cluster is realistic.

Figure 14(a) plots a CDF of the ratio between the job runtimes without and with RoPE. So, $y = 1$ implies no benefit. Samples below that line are regressions while those above indicate improvement. To compute a single metric from a disparate set of jobs, we weight job by the



(a) Reduction in Job Latency (b) Reduction in Cluster Hours Used (c) Reduction in data shuffled across racks

Figure 14: Aggregated results from production dataset (see 5.3)

total cluster hours that it occupies. All the results in this section share this format.

Figure 14(a) shows that 25% (50%) of the jobs speed up by over $2x$ ($1.5x$). Several significant jobs see over $5x$ reduction in their latencies. Upon inspection we find that all the samples under the $y = 1$ line are from two jobs. Both jobs have a small number of stages (and tasks). RoPE does not change their execution plans. Noise due to the cluster is the likely cause for lower performance. The vast majority of jobs see performance improvements. The reason is due to one or more of the optimizations described in the case study above.

Figure 14(b) shows that the latency savings due to RoPE are not from simply using more resources. In fact by avoiding wasteful work, RoPE speeds up jobs while reducing cluster usage. At the 50th (75th) percentile, jobs use $1.2x$ ($1.5x$) fewer cluster hours with RoPE.

Figure 14(c) shows that while the volume of cross rack shuffle is lower for some jobs, it stays the same for many and increases for only a few. This is expected since while some of the optimizations enabled by RoPE lower cross rack shuffle, others can increase it. Our optimization goal is to improve job latency and hence, on all the other metrics, we only indirectly constrain RoPE. Yet, we find that RoPE mostly achieves its gains by shuffling fewer amounts of data across the network. Figure 15(b) shows a similar pattern for the data read and written to disk. Figures 15(a) and 15(c) show that RoPE's plans mostly use fewer tasks and stages, though sometimes, for e.g., when offsetting the cost of UDOs, RoPE can use more tasks.

In summary, RoPE judiciously uses resources to improve job latency. Some gains accrue from avoiding wasted work, others from trading a little more of one type of resource for large savings on another while still others accrue from balancing the parallel plans.

6. RELATED WORK

Recent work on data-parallel cluster computing framework has mainly focused on solving issues that arise during the execution of jobs, by sharing the cluster [13, 23], tackling outliers [1], fairness vs. locality [25] and network scheduling [8]. Others incorporate new functionality such as the support for iterative and recursive control flow [20]. Orthogonally, RoPE generates better execution plans by leveraging data and execution insights.

The AutoAdmin project examined adapting physical database design, e.g., choosing which indices to build and which views to materialize, based on the data and queries.

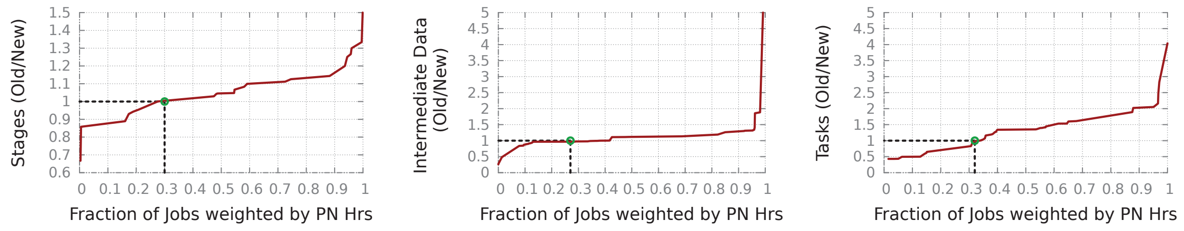
Closer to us, is the work that adapts query plans based on data. Kabra and DeWitt [16] were one of the earliest to propose a scheme that collects statistics, re-runs the query optimizer concurrently with the query, and migrates from the current query plan to the improved one, if doing so is predicted to improve performance. They mainly address the challenges of trading off re-optimization vs. doing actual work and of re-using the partial executions from the old plan to avoid wasting work that is already done. These challenges are easier in the context of map-reduce while collecting statistics is harder due to the distributed setting. Further, RoPE explores new opportunities arising due to the parallel nature of plans.

Eddies [2] adapts query executions at a much finer, per-tuple, granularity. To do so, Eddies (a) identifies points of symmetry in the plan at which re-ordering can happen without impacting the output, (b) creates tuple routing schemes that adapt to the varying selectivity and costs of operators. RoPE looks at a disjoint space of optimizations (choosing appropriate degrees of parallelism and operator implementations), which are not easily cast into Eddies' tuple routing algorithm.

Starfish [12] examines Hadoop jobs, one map followed by one reduce, and tunes low-level configuration variables in Hadoop such as io.sort.mb. To do so, it constructs a what-if engine based on a classifier trained on experiments over a wide range of parameter choices. Results show that the prescriptions from Starfish improve on developer's rules-of-thumb on non-traditional servers (e.g., fewer memory or cores). RoPE is complementary because (a) it applies to jobs that are more complex than a map followed by a reduce, (b) explores a larger space of plans and (c) uses a cost-based optimizer.

7. DISCUSSION

Recurring jobs are a first-order use case in our production system. We find that RoPE achieves meaningful improvements to such jobs. However, it is important



(a) Reduction in # of Stages (b) Reduction in intermediate writes to disk (c) Reduction in # of Tasks

Figure 15: Aggregated results from production dataset (see 5.3)

to note that statistics from a run only cover sub-graphs of operations used in that execution plan. This information may not suffice to find the optimal plan. After a few runs, we usually find all the necessary statistics since each run can explore new parts of the search space. However, plans chosen during this transition are not guaranteed to monotonically improve. In fact, there are no general ways to bound the worst case impact from plans chosen based on incomplete information. As a result, RoPE largely errs on the side of very conservative changes.

We defer to future work some advanced techniques that choose plans given uncertainity regimes over statistics or choose a set of plans, each of which has an associated validity range specified over statistics, and switch between these plans at runtime depending on the observed statistics [3]. Clearly these techniques are more complex than RoPE, the risks from picking worse plans are larger here, and to the best of our knowledge using these ideas in the context of parallel plans is an open problem.

8. FINAL REMARKS

Results from a deployment in Bing show that leveraging properties of the data, the code and their interaction significantly improves the execution of data parallel programs. The improvements derive from using statistics to generate better execution plans. Note that these improvements are mostly orthogonal to those from solving runtime issues during the execution of the plans (e.g., outliers, placing tasks). They are also in addition to those accrued by a context-blind query optimizer over literally executing the programs as specified by the users.

While RoPE leverages database ideas, we believe that the realization in the context of data parallel programs is interesting due to challenges that are new (e.g., distributed collection), or are more important in this context (e.g., user defined operations) and novel opportunities for improvement (e.g., recurring jobs, little data and the optimizations specific to parallel plans such as choosing degree of parallelism to achieve balance).

Acknowledgements

We thank Suman Nath, Peter Bodik, Andrew Ferguson, Jonathan Perry, our shepherd Roxana Geambasu and members of the UC Berkeley AMP Lab for comments that improved this work. The Cosmos team has built and keeps running the fabulous system without which this work would not have been possible.

References

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, et al. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI*, 2010.
[2] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29, May 2000.
[3] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD*, 2005.
[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, 2002.
[5] R. Chaiken, B. Jenkins, P. Larson, et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[6] C. Chambers, A. Raniwala, F. Perry, et al. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.
[7] B. Chattopadhyay, L. Lin, W. Liu, et al. Tenzing a sql implementation on the mapreduce framework. In *VLDB*, 2011.
[8] M. Chowdhury, M. Zaharia, J. Ma, et al. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, 2011.
[9] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, 2003.
[10] C. A. Galindo-Legaria, M. M. Joshi, F. Waas, and M.-C. Wu. Statistics on views. In *VLDB*, 2003.
[11] P. K. Gunda, L. Ravindranath, C. A. Thekkath, et al. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.
[12] H. Herodotou, H. Lim, G. Luo, et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.
[13] B. Hindman, A. Konwinski, M. Zaharia, et al. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
[14] M. Isard et al. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Eurosys*, 2007.
[15] M. Isard, V. Prabhakaran, J. Currey, et al. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.
[16] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.
[17] Q. Ke, V. Prabhakaran, Y. Xie, et al. Optimizing data partitioning for data-parallel computing. In *HotOS*, 2011.
[18] P.-A. Larson et al. Cardinality estimation using sample views with quality assurance. In *SIGMOD*, 2007.
[19] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.
[20] D. G. Murray and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.
[21] C. Olston, B. Reed, U. Srivastava, et al. Pig Latin: A Language for Data Processing. In *SIGMOD*, 2008.
[22] A. Rasmussen, G. Porter, M. Conley, et al. Tritonsort: a balanced large-scale sorting system. In *NSDI*, 2011.
[23] A. Shieh, S. Kandula, A. Greenberg, et al. Sharing the data center network. In *NSDI*, 2011.
[24] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.
[25] M. Zaharia, D. Borthakur, J. S. Sarma, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.