Continuous Cloud-Scale Query Optimization and Processing

Nicolas Bruno
Microsoft Corp.
nicolasb@microsoft.com

Sapna Jain
IIT Bombay
sapnakjain@gmail.com

Jingren Zhou
Microsoft Corp.
jrzhou@microsoft.com

ABSTRACT

Massive data analysis in cloud-scale data centers plays a crucial role in making critical business decisions. High-level scripting languages free developers from understanding various system trade-offs, but introduce new challenges for query optimization. One key optimization challenge is missing accurate data statistics, typically due to massive data volumes and their distributed nature, complex computation logic, and frequent usage of user-defined functions. In this paper we propose novel techniques to adapt query processing in the SCOPE system, the *cloud-scale computation environment* in Microsoft Online Services. We continuously monitor query execution, collect *actual* runtime statistics, and adapt parallel execution plans as the query executes. We discuss similarities and differences between our approach and alternatives proposed in the context of traditional centralized systems. Experiments on large-scale SCOPE production clusters show that the proposed techniques systematically solve the challenge of missing/inaccurate data statistics, detect and resolve partition skew and plan structure, and improve query latency by a few folds for real workloads. Although we focus on optimizing high-level languages, the same ideas are also applicable for MapReduce systems.

1. INTRODUCTION

An increasing number of companies rely on the results of massive data computation for critical business decisions. Such analysis is crucial to improve service quality, support novel features, and detect changes in patterns over time. Usually the scale of the data volumes to be stored and processed is so large that traditional, centralized database system solutions are no longer practical. Several companies have thus developed distributed data storage and processing systems on large clusters of thousands of shared-nothing commodity servers [2, 26, 10, 13].

In the MapReduce model, developers provide *map* and *reduce* functions in procedural languages like `Java`, which perform data transformation and aggregation. The underlying runtime system achieves parallelism by partitioning the data and processing each partition concurrently, handling load-balancing, outlier detection, and failure recovery.

This approach, however, has its own set of limitations. Developers are required to translate their application logic to the MapReduce model in order to achieve parallelism. For some applications this mapping is very unnatural, especially for simple operations like projection and selection. The resulting custom code is error-prone and hardly reusable. Moreover, optimizing and reasoning over complex, opaque, multi-step MapReduce jobs is very challenging.

High level scripting languages were recently proposed to address these limitations (e.g., Sopremo/Meteor [4], Jaql [5], Asterix [6], Tenzing [8], Dremel [20], Pig [21], Hive [24], DryadLINQ [25], and SCOPE [26, 27]). These languages offer a single machine programming abstraction and allow developers to focus on application logic, while providing systematic optimizations for the underlying distributed computation. As in traditional database systems, such optimization techniques rely on data statistics to choose the best execution plan in a cost-based manner [27, 17]. Collecting accurate statistics and effectively exploiting such information has a profound impact on optimization quality.

We now illustrate the importance and challenges of calculating accurate statistics in a large-scale distributed environment. Consider the simple query in Figure 1(a), which joins results of a user-defined aggregate and a user-defined predicate over two inputs extracted from raw logs using user-defined extractors. Figure 1(b) shows a possible execution plan for this query. Specifically, after processing the raw log `r.txt` using a user-defined extractor `RExtractor`, we partition the data by column {`a`}, which is required for both the aggregation[1] and subsequent join. We do the same for the result of filtering data with `UDFilter(a, d)` from the raw log `s.txt` using a user defined extractor `SExtractor`. This plan is attractive in general compared to the alternative in Figure 1(c), which repartitions the data from `r.txt` twice. The reason is that data shuffling is one of the most expensive operations in distributed execution plans.

However, suppose that some values of `a` in input `r.txt` are more frequent than others. In this case, the initial repartitioning by {`a`} in Figure 1(b) would result in some partitions being much larger than others, which results in much longer latency and the possibility of job failures in extreme situations. In such a case, the plan in Figure 1(c) is more

[1]For correctness, the input to the aggregation operator must be partitioned by any subset of the grouping columns.

Proceedings of the VLDB Endowment, Vol. 6, No. 11

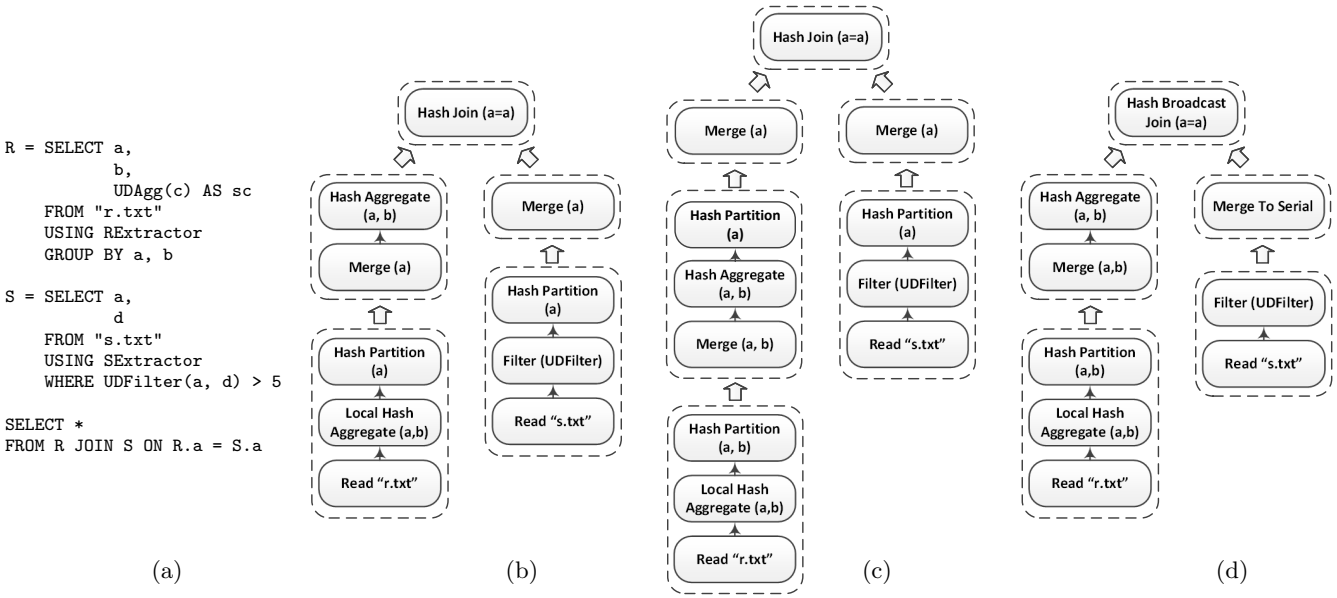


Figure 1: Alternatives to Execute a Distributed Query

attractive, because it first repartitions by {a,b}, which does not have skewed distribution, and subsequently reduces data before repartitioning it on {a} for the join.

Now suppose that the user-defined predicate on input s.txt is very selective and thus drastically reduces the size of the input. Then, the plan in Figure 1(d) is a better choice because it repartitions the data from r.txt by {a,b} (addressing the skew issue described above), and performs a broadcast-join alternative that does not require a second repartitioning operation. This is especially important when subsequent operators require the input partitioned by columns {a,b}. At the same time, however, this plan would result in very inefficient executions (or failures) if the user-defined predicate is not selective enough.

The previous examples illustrate that the choice of a plan heavily depends on properties of the input data and user-defined functions. Several aspects of highly distributed systems make this problem more challenging compared to traditional database systems. First, it is more difficult to obtain and maintain good quality statistics due to the distributed nature of the data, its scale, and common design choices that do not even account for accurate statistics on input unstructured data. Second, most queries heavily rely on user-defined code, which makes the problem of statistical inference much more challenging during optimization. Finally, input scripts typically consist of hundreds of operators, which magnify the well-known problem of propagation of statistical errors. This negatively impacts the quality of the resulting execution plans, and might lead, as illustrated above, to choosing plans with partition skew or wrong degree of parallelism. As a result, the performance impact could be in the orders of magnitude.

On the other hand, queries in such a cloud-scale distributed environment consume a large amount of system resources and typically run for minutes or hours. Query optimization time is almost negligible compared to its execution time. Additionally, certain trade-offs between performance and fault-tolerance result in a different execution model compared to that of centralized system. These design choices result in unique opportunities to monitor and adjust execution plans dynamically and cheaply.

In this paper we propose a solution based on *continuous query optimization*. This approach is different from the traditional one, which focuses on collecting and propagating data statistics *before* query execution. Instead, we continuously monitor query execution, collect *actual* runtime statistics, and adapt execution plans as the query executes. Particularly, we seamlessly integrate the query optimization, execution, and scheduling components at runtime. The query optimizer is triggered whenever new runtime statistics become available. If a better plan is found, we intelligently adapt the current execution plan with minimal changes. Experiments on a large-scale SCOPE production system at Microsoft show that the proposed approach systematically solves the challenges of missing/inaccurate data statistics and improve query latency by a few folds for real-world queries. Although this paper uses SCOPE as the underlying platform, the ideas are applicable to any distributed system that deals with large-scale data computation.

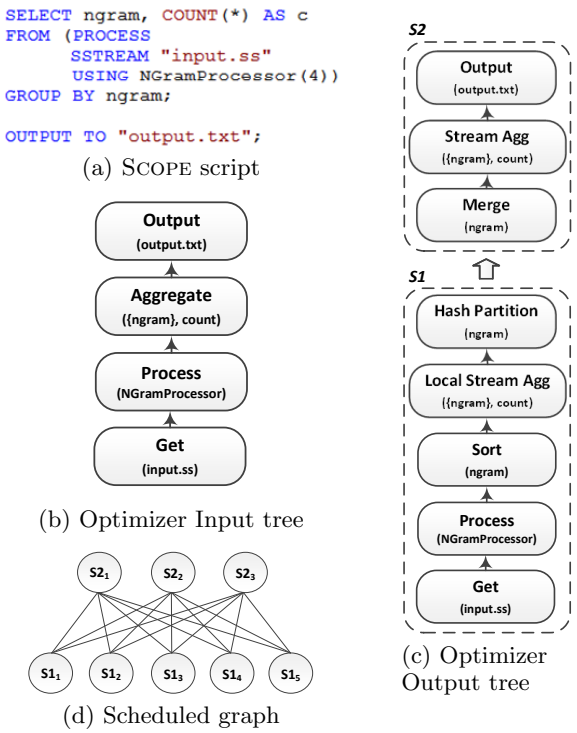
The rest of the paper is structured as follows. In Section 2 we review necessary background on SCOPE. In Section 3 we describe continuous query optimization. Section 4 reports an experimental evaluation of our approach. Section 5 discusses related work, and Section 6 concludes the paper.

2. THE SCOPE SYSTEM

In this section we describe the main architecture of the SCOPE computation system at Microsoft [13, 26].

2.1 The Language and Data Model

The SCOPE language is declarative and intentionally reminiscing SQL. The select statement is retained along with joins variants, aggregation, and set operators. Like SQL, data is modeled as sets of rows composed of typed columns, and every rowset has a well-defined schema. At the same time, the language is highly extensible and is deeply integrated with the .NET framework. Users can easily define



(a) SCOPE script

(b) Optimizer Input tree

(c) Optimizer Output tree

(d) Scheduled graph

Figure 2: Compiling and Executing in SCOPE

their own functions and implement their own versions of relational operators: *extractors* (parsing and constructing rows from a raw file), *processors* (row-wise processing), *reducers* (group-wise processing), *combiners* (combining rows from two inputs), and *outputters* (formatting and outputting final results). This flexibility allows users to solve problems that cannot be easily expressed in SQL, while at the same time enables sophisticated reasoning of scripts.

In addition to unstructured data, SCOPE supports *structured streams*. Like tables in a database, a structured stream has a well-defined schema that every record follows. A structured stream is self-contained and includes, in addition to the data itself, rich metadata information such as schema, structural properties (i.e., partitioning and sorting information), and statistical information on data distributions [26].

Figure 2(a) shows a simple SCOPE script that counts the different 4-grams of a given single-column structured stream. In the figure, `NGramProcessor` is a `C#` user-defined operator that outputs, for each input row, all its n-grams (4 in the example). Conceptually, the intermediate output of the processor is a regular rowset that is processed by the main outer query (note that intermediate results are not necessarily materialized between operators at runtime).

2.2 Query Compilation and Optimization

A SCOPE script goes through a series of transformations before it is executed in the cluster. Initially, the SCOPE compiler parses the input script, unfolds views and macro directives, performs syntax and type checking, and resolves names. The result of this step is an annotated abstract syntax tree, which is passed to the query optimizer. Figure 2(b) shows an input tree for the sample script.

The SCOPE optimizer is a cost-based transformation engine that generates efficient execution plans for input trees. Since the language is heavily influenced by SQL, SCOPE is able to leverage existing work on relational query optimization and perform rich and non-trivial query rewritings that consider the input script in a holistic manner. The optimizer returns an execution plan that specifies the steps that are required to efficiently execute the script. Figure 2(c) shows the output from the optimizer, which defines specific implementations for each operation (e.g., stream-based aggregation), data partitioning operations (e.g., the partition and merge operators), and additional implementation details (e.g., the initial sort after the processor, and the unfolding of the aggregate into a local/global pair).

The backend compiler then generates code for each operator and combines a series of operators into an execution unit or *stage*, obtained by splitting the output tree into components that would be processed by a single node. The output of the compilation of a script thus consists of (i) a graph definition file that enumerates all stages and the data flow relationships among them, and (ii) the assembly itself, which contains the generated code. This package is sent to the cluster for execution. Figure 2(c) shows dotted lines for the two stages corresponding to the input script.

2.3 Job Scheduling and Runtime

The execution of a SCOPE script is coordinated by a Job Manager (or JM). The JM is responsible for constructing the job graph and scheduling work across available resources in the cluster. As described above, a SCOPE execution plan consists of a DAG of stages that can be scheduled and executed on different machines independent of each other. A stage represents multiple instances, or *vertices*, which operate over different partitions of the data (see Figure 2(d)). The JM maintains the job graph and keeps track of the state and history of each vertex in the graph. When all inputs of a vertex become ready, the JM considers the vertex *runnable* and places it in a scheduling queue. The actual vertex scheduling order is based on vertex priority and resource availability. One scheduling principle is based on *data locality*. That is, the JM tries to schedule the vertex on a machine that stores or is close to its input whenever possible. If the selected machine is temporarily overloaded, the JM may scale back to another machine that is close in network topology so reading the input can be done efficiently with minimum network traffic. Additionally, the nature of distributed computation on top of unreliable hardware results in the JM sometimes launching redundant vertices, in what is known as speculative execution. If some machine is slow or unresponsive, a redundant computation in a different machine can be leveraged rather than increasing the execution critical path waiting for a slow partial result.

During execution, a vertex reads inputs either locally or remotely. Operators within a vertex are processed in a pipelined fashion, similar to a single-node database engine. Every vertex is given enough memory to satisfy its requirements (e.g., hash tables or external sorts), up to a fraction of total available memory, and a fraction of the available processors. This procedure sometimes prevents a new vertex from being run immediately on a busy machine. Similarly to traditional database systems, each machine uses admission control techniques and queues outstanding vertices until the

required resources are available. The final result of a vertex is written to local disks (non-replicated for performance reasons), waiting for the next vertex to *pull* data.

The *pull* execution model and materialization of intermediate results has many advantages in the context of highly distributed computation. First, it does not require both producer and consumer vertices to run concurrently, which greatly simplifies job scheduling. Second, in case of failures, which are inevitable, all that is required is to rerun the failed vertex from the cached inputs. Only a small portion of a query plan may need to be re-executed. Finally, writing intermediate results frees system memory to execute other vertices and simplifies computation resource management.

3. CONTINUOUS QUERY OPTIMIZATION

In previous sections we discussed two important facts regarding query processing in distributed settings. First, the quality of query optimization is directly influenced by the quality of statistics on the underlying data distributions. Second, it is very difficult, and sometimes impossible, to obtain good quality statistics on intermediate computation results before the query executes. This conflict points to a fundamental design limitation in the current processing cycle of complex distributed applications. In this section we describe our approach for dealing with uncertainty during optimization, which essentially (i) continuously gathers statistics as queries are executed, (ii) reoptimizes the running query with new, more accurate information, and (iii) modifies the current execution graph on the fly if significant differences are found based on newly discovered information.

Our approach is inspired by similar solutions in the context of traditional centralized database systems, but adapted to deal with and leverage some unique characteristics of the distributed computation model. First, optimization time in our environment is a negligible fraction of the overall query latency, which needs to synchronize computation across large number of commodity machines. This is different in centralized systems, where cheap queries can execute in milliseconds, making any additional work a potential performance regression. Additionally, the nature of distributed computation on top of unreliable hardware has a profound effect on design choices. Speculative execution of vertices is not needed nor wanted in traditional centralized solutions, which cannot afford to "waste" resources in anything non-essential. Also, to gracefully deal with failures, distributed queries are decomposed into small-running stages, which are processed in a sequential fashion by writing intermediate results into stable storage. Traditional centralized systems push pipelining to an extreme, which is very good for absolute latency minimization in presence of reliable components, but a bad alternative if failures are common. This last point has proven to be a challenge for optimization approaches in centralized systems, since plan changes either result in significant intermediate work being discarded or require explicit buffering between pipelined stages. Our approach leverages the staged execution model by making each stage the unit of computation.

3.1 Architecture

Figure 3 describes the overall architecture of our approach by following a sample job throughout the system. Initially, a new job is submitted to the cluster for execution (step 1 in the figure). The compiler parses the input script, performs

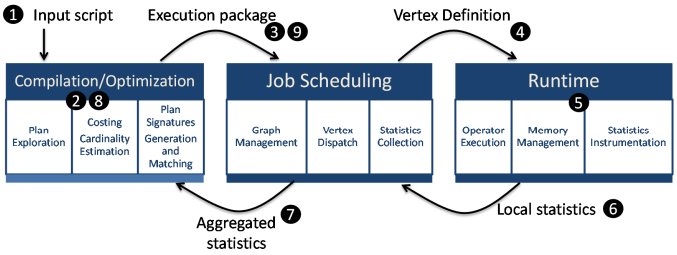


Figure 3: Continuous Optimization Architecture

syntax and type checking, and passes an annotated abstract syntax tree to the query optimizer. The query optimizer explores many semantically equivalent rewritings (e.g., those in Figure 1), estimates the cost of each alternative, and picks the most efficient one (step 2 in the figure).

We extend the query optimizer to annotate each node in the plan with a *signature* that uniquely identifies the computation it performs (see Section 3.2). Additionally, we instrument the generated code to collect specific statistics on vertex results (see Section 3.3).

The initial package produced by the optimizer is then sent to the JM (step 3 in the figure). The JM schedules the execution graph, transferring runnable vertices to available machines in the cluster (step 4 in the figure). The runtime instance on each machine executes vertices as usual, but concurrently collects statistics requested by the optimizer and instrumented during code generation (step 5 in the figure). As a vertex progresses, it periodically sends back to the JM aggregated statistical information (step 6 in the figure).

The JM aggregates information as vertices finish executing. Depending on an optimization policy, it eventually attempts to repair the currently executing plan (step 7 in the figure). For that purpose, it calls the optimizer asynchronously, passing the collected statistics associated with the corresponding plan signatures (see Section 3.4). The optimizer is not only aware of statistical information, but also of the fact that certain portions of the currently executing plan are already materialized as intermediate results. The optimizer reoptimizes the input query with this additional information (step 8 in the figure). The resulting plan is returned to the JM (step 9 in the figure). When the JM obtains a new execution plan, it repairs the current executing plan with the information in the new package, and seamlessly continues the job execution as normal (see Section 3.5). Steps 4-9 are repeated until the job finishes.

3.2 Plan Signatures

Plan signatures uniquely identify a logical query fragment during optimization. This is similar to the notion of view matching technology in traditional database systems. View matching is a very flexible approach, but at the same time it is very difficult to extend beyond select-project-join queries with aggregation. Traditional view matching is able to perform partial matching on different queries and compensate differences using additional relational operators. In our scenario we continuously reoptimize the *same* query, so we take a different approach that can handle any operator tree including user-defined operators in SCOPE.

Specifically, for every operator subtree we traverse back the sequence of rules applied during optimization [11] un-

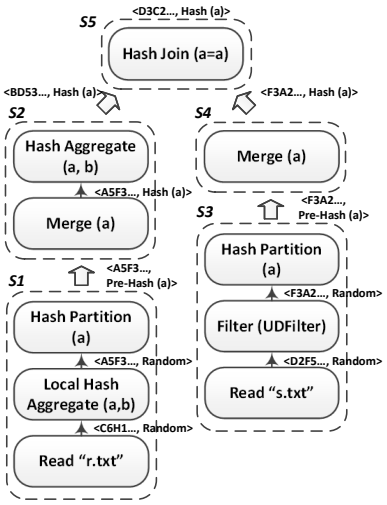


Figure 4: Signatures and Physical Properties

til we reach the initial semantically equivalent expression[2]. Since the optimizer is deterministic, this initial expression can be used as the canonical representation for the operator tree. We then recursively serialize the representation of the canonical expression and compute a 64-bit hash value for each subtree, which serves as the plan signature[3]. All fragments producing the same result are grouped together and have the same plan signature.

Figure 4 shows an execution plan annotated with signatures. Note that the `partition` and `merge` nodes in the figure have the same signature as the `read` child node, because they produce the same logical result with different physical properties (i.e., partitioning). To distinguish these operators during execution, the optimizer associates each operator with both its signature and the corresponding delivered properties. For instance, consider stage $S1$ in Figure 4. The `read` node in $S1$ is randomly partitioned, while the `hash partition` node is *hash pre-partitioned* by $\{a\}$ (i.e., while it is not yet partitioned by $\{a\}$, each vertex already generated the corresponding partitioning buckets).

3.3 Statistics instrumentation

As explained in Section 3.1, we instrument the generated code to capture statistics during execution and thus enable reoptimization. Statistics need to satisfy certain properties in our framework. First, statistics collection needs to have low overhead and consume little additional memory, as it is always enabled during normal execution. Second, statistics must be composable, because each vertex instance computes partial values that are then sent to and aggregated by the JM. Finally, statistics must be actionable, as there is no point in accurately gathering statistics unless the query optimizer is able to leverage them.

[2]Every rule transforms one plan into another, and we tag each resulting plan with its generating rule and source. That way, we can always track the plan back to the original logical expression.

[3]Modulo hash collisions, the signatures of two plan fragments match if and only if they produce semantically equivalent results. In practice, hash collisions are so rare that we can hash values as signatures, but other mechanisms are possible (e.g., using the serialization of the logical tree as the signature itself).

We conceptually model statistics as very lightweight user-defined aggregates whose execution is interleaved with normal vertex processing. This design allows to easily add new statistics into the framework as needed. Every operator in the runtime is augmented with a set of statistics to compute. On startup, the operator calls an `initialize` method on the statistics object. On producing each output row, the operator invokes an `increment` method on the statistics object passing the current row. Before closing, the operator invokes a `finalize` method on the statistics object, which computes statistics final results. Each statistics object also implements a `merge` method used by the JM to aggregate partial values across stages[4]. Examples include:

Cardinality and average row size. The object initializes cardinality and row size counters to zero, increments the cardinality counter and adds the size of the current row to the size counter for each `increment` function call, and returns the cardinality and average row size upon finalization.

Inclusive user-defined operator cost. The object initializes a timer on creation, does nothing on each increment function call, and returns the elapsed time at finalization.

Partitioning histogram. Histograms are only used for partitioning operators, and maintain the number of rows output to each partition bucket by a given vertex. We use one counter per output partition and initialize the array with zeros at the beginning. Each `increment` call simply increments the corresponding counter in the array, and we return the array of counters upon finalization. Merging two histograms is trivially done by adding each counter independently. These histograms provide a very low-overhead mechanism to calculate the number of rows that would end up in each partition *before* performing the aggregation itself, and are used to repair inherent data skew generated by suboptimal choices of partitioning columns.

As an example, a vertex of stage $S3$ in Figure 4 executes three operators: `read`, `filter` and `hash partition`. The vertex collects cardinality and average row sizes at the `read` and `filter` operators, and partition histograms at the `hash partition` operator. Suppose the `read` operator produces 1.5 million rows constituting $1.5GB$ data, the `filter` operator reduces the number of rows to $1,000$ constituting $0.35MB$ of data and the `hash partition` operator sends 4 rows to the first partition, 5 rows to the second, and so on. Upon completion, the vertex returns to the JM $\{\langle D2F5..., (1.5M, 1K)\rangle, \langle F3A2..., (1K, 350)\rangle, \langle F3A2..., [4, 5, ...]\rangle\}$. Note that the partitioning histogram is associated to a specific partition function, which is known by the JM.

3.4 Query Reoptimization

The JM is responsible for orchestrating the new continuous optimization approach. It does so by gathering statistics from vertices, preparing a statistical package sent to the query optimizer, and transitioning to a new execution plan if necessary. The JM in Scope has traditionally been structured to respond to different events, such as *vertex is*

[4]Some common statistics are optimized further by directly generating code that performs the initialization, increment, and finalization methods without requiring virtual function calls.

```
ON VertexCompleted (V: vertex in stage S)
01  SP = global statistics package
02  MP = global materialization package
03  if (V is completed for the first time)
04     updateStatisticsPack(SP, V.stats)
05     updateMaterializationPack(MP, V)
06  if (reoptimization policy)
07     newPlan = optimize(query, SP, MP)
08     if (newPlan != currentPlan)
09        replace currentPlan with newPlan
10   // additional vertexCompleted handling...
```

Figure 5: VertexCompleted Handler

runnable, or *vertex finished execution*. We introduce continuous optimization by augmenting the *vertex finished execution* handler[5]. The pseudocode in Figure 5 details the additional actions that take place in the JM on completion of each vertex, enabling continuous optimization. In the following sections we explain this algorithm in detail.

3.4.1 Data Structures

The JM maintains two additional structures while the job is executing. The *statistics package* (or `SP` in line 1 in Figure 5), is a global collection of plan signatures and aggregated statistics returned by completed vertices. The *materialization package* (or `MP` in line 2) is a global collection that tracks intermediate results materialized so far. Specifically, it contains plan signatures along with the fraction of vertices that already completed execution and the corresponding paths of already materialized intermediate results.

Updating Data Structures: When a vertex finishes execution, the JM obtains the statistics gathered during execution. Note that due to vertex failures, the same vertex instance might have been executed multiple times (each time returning the same results due to determinism assumptions). If this is the first time the vertex has finished execution, both the statistics and materialization packages are updated (lines 3-5). To update statistics for a vertex in `SP`, the JM uses the specific `merge` method defined on the corresponding statistics objects. Note that while the statistical package is updated for every plan signature in the vertex, the materialization package is only processed for the *root* plan signature of the given vertex. The reason is that an internal operator in a vertex is not materialized to disk and cannot be reused as an intermediate result. In Figure 4, when a vertex of stage $S3$ finishes, the output of `Read(s.txt)` is not available, as it is pipelined through the `Filter` operator.

3.4.2 Reoptimization policy

After the statistics and materialization packages are updated due to the completion of a vertex in the current job, a policy decides whether to reoptimize the job based on the current information (line 6 in Figure 5). An extreme policy is to reoptimize the input query every time a vertex completes execution. While it provides the finest granularity, it might impose unnecessary overhead to the JM. The reason is that query optimization, while efficient, still consumes resources, and our infrastructure is optimized for scheduling and executing a large number of small vertices. It is not uncommon for the JM to have thousands of outstanding executing vertices, and tens of completed vertices per second. A more reasonable alternative is to wait until a whole stage is finished (i.e., whenever all vertices of a given stage finish executing). This option provides a good tradeoff between optimization overhead and the granularity of the statistics. Note that multiple stages are executing concurrently, so completion of all vertices in one stage will necessarily happen while other stages have partially materialized results. In our experiments we reoptimize the input query every time all the vertices in a stage are completed, and put no limit to the number of reoptimizations on a given query. As shown in the experimental evaluation in Section 4, this simple policy provides remarkably robust results.

3.4.3 Reoptimization

SCOPE uses a transformation-based optimizer based on the Cascades framework [11], which translates input scripts into efficient execution plans. Transformation-based optimization can be viewed as divided into two phases, namely, logical exploration and physical optimization. Logical exploration applies transformation rules that generate new semantically equivalent logical expressions. All equivalent logical expressions are grouped together in a data structured called *memo*. Examples of such rules include pushing selection predicates closer to the sources, and transforming an associative reducer into a local and global pair. Implementation rules, in turn, convert logical operators to physical operators, cost each alternative, and return the most efficient one. Examples include using sort- or hash-based algorithms to implement an aggregate query, or deciding whether to perform a pairwise or broadcast join variant.

To enable continuous optimization, we extended the optimizer to accept, in addition to the input script, optional statistics and materialization packages (see Section 3.4.1). The optimizer also computes signatures of each expression it processes, which is done with very little overhead. We next describe how we extend the optimizer to leverage statistics and materialization packages provided by the JM.

Statistics Package

The statistics package produced by the JM contains a set of signatures with statistical information on corresponding query subplans. To leverage this information, we modify the entry point to the derivation of statistical properties in the optimizer to perform a lookup on the statistics package. For instance, deriving the cardinality of an expression starts by checking whether the signature of the expression matches an instance in the statistics package. If so, cardinality estimation is bypassed and replaced by the value specified in the package. Since all equivalent expressions have the same signature, this technique produces consistent results.

We also extended the optimizer with a new property called *skew*. *Skew* values are associated with a set of columns C and approximately model the skew in the number of rows per partition by C in an intermediate result. Specifically, we use the ratio between largest partition count and the average partition count as the basis of skew values. Skew values range from zero (when all partitions are equally large) to one (when a single partition contains all values and the others are empty). We make the simplifying assumption that this ratio is independent on the number of partitions, which

[5]Finer granularity is possible by periodically sending to the JM partial statistics from running vertices. We do not consider such alternatives and uses vertices as the unit of reoptimization.

is not always true but nevertheless provides a robust mechanism to reason with skewed data distributions. Whenever the statistics package contains a histogram of N frequencies f_i for a given set of columns C, we compute the skew value for C as $skew(C) = (\frac{\max f_i}{\sum f_i/N} - 1) \cdot \frac{1}{N-1}$ (so $0 \leq skew(C) \leq 1$).

We leverage skew values while performing cost estimation. The optimizer models overall latency, which is obtained by adding the cost of each operator on a per-partition basis (e.g., if a reducer is done over 100 partitions, we compute the cost of the reducer using an input cardinality that is $1/100th$ of the total input cardinality). We leverage skew values during cost estimation by calculating the cost of the operator for the worst case (i.e., the instance with the largest cardinality values). Suppose the current operator handles P partitions on columns C, and $skew(C) = S$. If the total cardinality is CT, the original cardinality estimation per partition is simply CT/P. Instead, we compute the cardinality of the largest partition as $CT/P \cdot (1 + S \cdot (P - 1))$, and thus model data skew.

For instance, suppose that there is a join operator over columns (a, b, c). The optimizer needs to partition the join inputs by any subset of (a, b, c). If we have skew information on some subset of (a, b, c) (say b), which would result in outliers during execution time, we will cost such alternatives much higher than others that partition on a superset of b, and thus have no (known) skew.

Materialization Package

The JM also creates a materialization package that contains information about intermediate materialized results. We incorporate this information into the query optimizer by adding new exploration rules that fire whenever the signature of the expression being explored matches a signature in the materialization package. When this happens, we generate a new alternative expression that simply reads the intermediate data. Leveraging the property derivation infrastructure in the optimizer, we set the physical properties (e.g., partitioning and sorting) of such intermediate result as dictated by the materialization package. This rule is conceptually very similar to materialized view matching rules in centralized database systems, but due to the use of signatures we are able to match arbitrary query fragments.

Partial Materialized Views: Recall that the materialization package contains entries for all stages with vertices in either *completed* or *executing* state. The reason is that vertices from different stages often execute concurrently. If we only capture stages that finished all its vertices and ignore stages that have only a subset of vertices completed and materialized, we would waste significant resources. Discarded stages might have been executing for a long time and in fact almost ready to be used when a new optimization happen. To address these scenarios, we model these intermediate results as *partial materialized views*. A partial materialized view is the same as a regular materialized view, but has an additional cost associated with it. This extra cost models the *remaining work* needed to complete the execution of all remaining vertices for the stage[6]. We approximate the extra cost of a partial materialized view by (i) obtaining the original plan used to execute the partial materialized view, (ii) rederiving data properties on this plan using the statistical package, (iii) recosting the original plan under the new statistical model, and (iv) adjusting the resulting cost based on fraction of nodes that already finished execution. The result of this procedure is the cost of the remainder of the actual plan being executed in the server, in optimizer units under the revised statistical model. The execution plan chosen by the optimizer might refer to a partial materialized view if the optimizer judges that the cost of waiting for the current stage to finish is smaller than the cost of restarting the work using a different plan.

[6]The unit of execution is still a vertex. Partial materialized views enable reasoning with stages for which some vertex already finished, but do not handle partially executed vertices.

```
MergePlan (P: new plan, G: current execution graph)
01  M = empty map from nodes in P to nodes in G
02  foreach node p in a bottom-up traversal of P
03     foreach pi ∈ children(p)
04        gi = M.lookup(pi)
05     g = G.Locate(p, {gi})
06     if (g not present)
07        g = G.Insert(p, {gi})
08     M.add(p → g)
09  Garbage-collect unreachable nodes in G
```

Figure 6: Merging a New Plan in the JM

3.5 Adapting Execution Plans

The output from the query optimizer is a new execution plan for the input query that very frequently depends on intermediate results specified in the materialization package. When the JM receives a new execution plan, it *merges* it with the currently executing graph and continues execution. This is done by a bottom-up traversal of the new execution plan, inserting new or reusing existing nodes in the current graph. Figure 6 illustrates this procedure.

Specifically, we maintain a map from nodes in the new execution plan to stages in the job currently executing in the JM (line 1 in the figure). In lines 2-8 we process nodes in the new execution plan in a bottom-up manner. For each node `p` in the new execution plan, we determine whether the node is already part of the currently executing graph. For that purpose, we first determine the nodes in the current graph that correspond to the children of `p` (which would have been added to the map `M` in previous iterations of the bottom-up enumeration). Then, we try to *locate* a node in the current graph that corresponds to `p` in line 5. This method searches `G` for a node that performs the same semantic operations as `p` (i.e., has the same signature), and consumes inputs from children that (recursively) perform the same operations as well. If we do not find such `g` node in the running graph, the current `p` node is part of a different execution for the input query, and so we instantiate it in the graph as traditionally for the initial graph, and connect its inputs to those in `gi`. We then add the new mapping from `p` to `g` in the mapping table `M`. At the end there will be nodes in `G` that are no longer reachable since they are part of the previous execution plan. These nodes are garbage-collected and discarded (line 9).

A small but important special case occurs whenever the current node `p` in the new plan reuses an intermediate result in the materialization package. Recall that the optimizer treats intermediate results as materialized views, which are represented with traditional `read` operations. In this situation, line 5 would not succeed in finding the corresponding node in `G`. Whenever we insert such a node in the graph in line 7, we find the node in the original graph producing the intermediate result consumed by `p`, and connect these

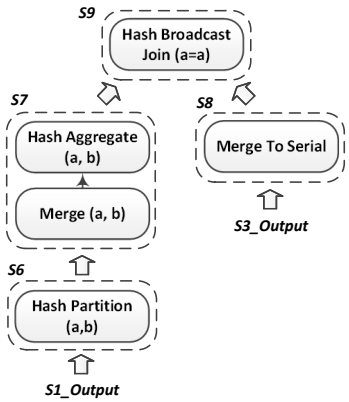


Figure 7: Adapting a Query Plan (see Figure 4)

two nodes together. By modifying the graph in this way, the JM can still use traditional fault tolerance mechanisms to recover from failed nodes. If a machine hosting non-replicated intermediate results fails, the JM re-runs the parent vertex corresponding to the missing input and continues as usual.

As an example, suppose that the JM executes stages $S1$ and $S3$ in Figure 4 and learns that $S1$ is skewed on column a and $S3$ is much smaller in volume than estimated. The optimizer might then produce a plan like in Figure 7, which reuses intermediate materialized results from $S1$ and repartitions the data by (a, b), merges in a serial plan the result from $S3$, and changes the join to using a broadcast variant.

4. EXPERIMENTAL EVALUATION

We implemented the continuous query optimization approach in SCOPE, which is deployed on production clusters consisting of tens of thousands of machines at MICROSOFT. SCOPE clusters execute tens of thousands of jobs daily, reading and writing tens of petabytes of data in total, and powering different online services. Each machine has two six-core AMD Opteron processors running at 1.8GHz, 24 GB of DRAM, and four 1TB SATA disks. All machines run Windows Server 2008 R2 Enterprise X64 Edition.

The following report uses queries chosen from real workload and evaluated on a production cluster. In our production workload, the main performance bottlenecks of problematic queries are sub-optimal degree of parallelism and data partition skew due to suboptimal partitioning keys chosen by the optimizer. This usually happens due to incorrect estimation of cardinality and key distribution across partitions. Most of the cardinality estimation errors occurs due to black-box user-defined functions, whose selectivity varies in a wide range, and unknown data distribution, either from inputs or from intermediate results. Incorrect cardinality estimation also contributes to other sub-optimal plan choices, such as join reordering, choices of join and aggregation operations. Due to limited space, we focus our experiments on optimizing the choice of degree of parallelism and partition keys that demonstrate important performance gains.

In Section 4.1 we present a case study and demonstrate how continuous query optimization impacts query execution during the lifetime of a query. We then summarize experimental results for different classes of real user queries in Section 4.2. Finally, we discuss plan convergence in Section 4.3 and optimization overhead in Section 4.4. Due to confidential data/business information, we report relative performance comparisons rather than absolute numbers.

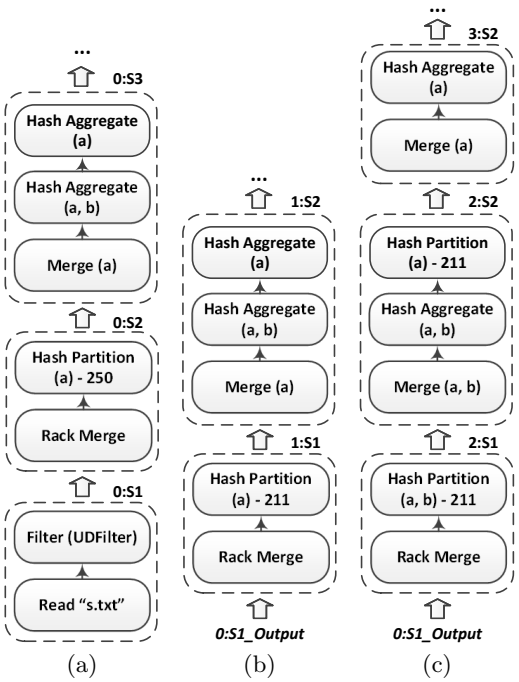


Figure 8: Execution Plans for a Simple Query

4.1 Case Study

As a case study, consider the following analytical query. It first applies a user-defined filter on top of unstructured input data, and performs a non-associative user-defined aggregate on columns $\{a, b\}$. It then performs a second non-associative aggregate on the result by column $\{a\}$.

```
A = SELECT a, b, UDAgg(c) AS sc
    FROM "r.txt" USING UDExtractor()
    WHERE UDFilter(a, b) > 10
    GROUP BY a, b

SELECT a, UDAgg(b) AS sb, UDAgg(sc) AS ssc
FROM A
GROUP BY a
```

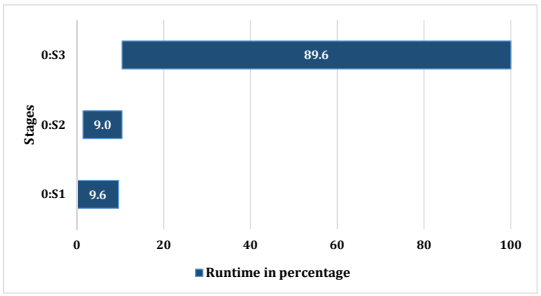
4.1.1 Baseline

As a baseline, we used the original SCOPE optimizer as described in [26], which is rather sophisticated but unaware of any execution feedback. Without knowledge on the selectivity of the user defined predicate `UDFilter` or distribution of values in columns a and b, the optimizer generates the plan in Figure 8(a), which consists of three stages (we denote a stage S generated during iteration i as i:S):

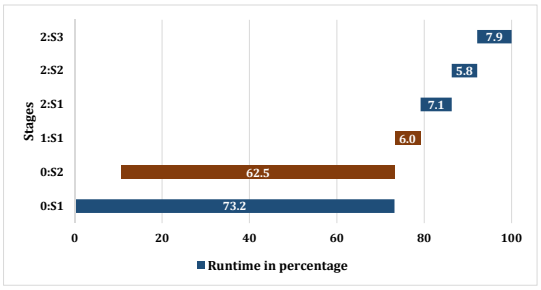
0:$S1$ reads the data using a user-defined extractor and applies the user-defined filter.

0:$S2$ partially aggregates the data from different machines in the same rack, to minimize inter-rack data transfer. It then hash partitions the data on $\{a\}$ into 250 partitions, which satisfies the requirements from both `GROUP BY` operations.

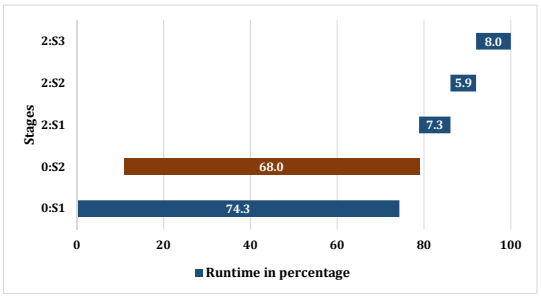
0:$S3$ aggregates the tuples belonging to same partition across all the vertices and then computes both aggregations in a pipelined manner.



(a) Baseline



(b) CQO without Partial Materialized Views



(c) CQO

Figure 9: Normalized Stage Runtime

Figure 9(a) shows the relative runtime of stages during the lifetime of the input query without continuous query optimization (denoted *Baseline* in the figure). Note that 0:$S3$, which merges the data partitioned by $\{a\}$ and computes the aggregate consumes 89% of the total latency due to data skew. The overall query latency is bounded by the most expensive vertex processing the largest partition.

4.1.2 *Continuous optimization*

Consider now continuous optimization (without partial materialized views). As the query executes, the JM learns that there is less data generated by `UDFilter` than the optimizer estimation, so 250 partitions are too many. At the same time, data has significant skew on column a. With our continuous optimization approach (without partial materialized views), the query is executed as follows.

When stage 0:$S1$ finishes execution, the JM gets accurate statistics and reoptimizes the query. The optimizer detects that the user-defined filter reduced the data size more than expected and generates a different plan with reduced degree of parallelism (211). The JM dynamically modifies the graph to the execution plan shown in Figure 8(b), discarding stages 0:$S2$ and 0:$S3$, and connecting the output of stage 0:$S1$ to the new stage 1:$S1$. Vertices from stage 1:$S1$ begin executing, and after they all complete, the next optimization detects partition skew on a and changes the plan to first partitioning on (a, b) to compute the aggregate on $\{a, b\}$, and then repartitioning on $\{a\}$ to compute the aggregate on $\{a\}$. The JM discards stages 1:$S1$ and 1:$S2$ and connects output of stage 0:$S1$ to stage 2:$S1$, as shown in Figure 8(c). Vertices from stage 2:$S1$ start executing and eventually the job finishes without further plan changes.

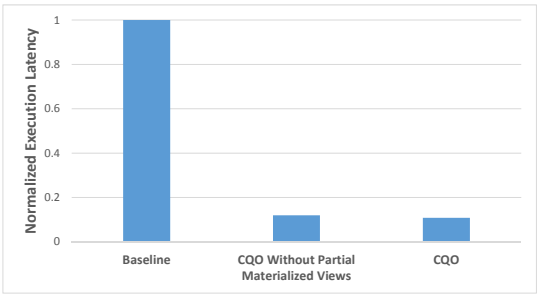


Figure 10: Query Latency

Figure 9(b) shows the relative runtime of stages with continuous query optimization (denoted *CQO without partial views* in the figure). During execution, we discard stages 0:$S2$, 0:$S3$, 1:$S1$ and 1:$S2$. Among these, 0:$S3$ and 1:$S2$ are discarded before they start executing with no overhead. Stage 0:$S2$ starts executing along with stage 0:$S1$ and is discarded right after reoptimization. While it does not result in latency overhead, it slightly increases resource consumption. The job scheduler gives priority to vertices on earlier stages, so 0:$S2$ does not cause delay in vertices of stage 0:$S1$. Finally, discarded stage 1:$S1$ caused increase on latency and wasted work (around 6%), which is a relatively small price paid for dynamically detecting and handling partition skew. Overall, query latency is reduced by 8.4x (see Figure 10).

4.1.3 *Leveraging partial materialized views*

During the first plan switch, stage 0:$S2$ was already 80% complete. So, when the optimizer considers partial materialized views, the cost with over-partitioning (20% of the cost of stage 0:$S2$) is cheaper than repartitioning with a slightly smaller number of partitions (211) so the optimizer does not change the plan and stage 0:$S2$ does not get discarded. When stage 0:$S2$ finishes, the optimizer detects partition skew and changes the plan directly to the final one in Figure 8(c). Figure 9(c) shows the relative runtime of stages in this case. The only overhead in terms of latency is the time 0:$S2$ takes after 0:$S1$ finishes, and the overall latency is further reduced by 9.4%, as shown in Figure 10.

4.2 Performance Evaluation

We next summarize results on evaluating continuous optimization on a representative workload consisting of queries both externally obtained from different customers (which include advertising and revenue, click information, and statistics about user searches), and internally used to administer the SCOPE cluster[7]. The queries in the workload have large variability in resource usage, with latencies that range

[7]All operational data generated by SCOPE is stored in the cluster itself, so analysis and mining of interesting cluster properties can be done by using SCOPE.

```
SELECT sd, se
       UDF(R.a, S.a),
       UDF(R.b, S.b),
       UDF(R.c, S.c),
FROM (SELECT a, b, c, d,
             UDA(e) AS se
      FROM <...> AS T
      GROUP BY a, b, c, d
      ) AS R
FULL OUTER JOIN
   (<...>) AS S
ON R.a == S.a AND
   R.b == S.b AND
   R.c == S.c
```

(a) Q2: skew on $T(a, b, c)$

```
SELECT a, b
       UDA(c) AS sc,
       UDA(sd) AS ssd
FROM (SELECT a, b, c,
             UDAgg(d) AS sd
      FROM (PROCESS
            (SELECT a, b, c, d
             FROM <...>) AS T
            USING UDProcessor
           ) AS R
      GROUP BY a, b, c
      )
GROUP BY a, b
```

(b) Q3: Processor doubles volume and skew on $R(a, b)$

```
SELECT a,
       RecursiveUDA(b) AS sb
FROM (SELECT a, b FROM <...>
      UNION ALL
      SELECT a, b FROM <...>
      UNION ALL
      SELECT a, b FROM <...>
      ) AS R
WHERE UDF(a, b) > 5
GROUP BY a
```

(c) Q5: UDF and RecursiveUDA reduce data volume to 800MB

```
SELECT R.a, R.b, R.sc, S.d
FROM (SELECT a, b,
             UDAgg(c) AS sc
      FROM (<...>)
      GROUP BY a, b) AS R
JOIN (SELECT a, d
      FROM (<...>)
      WHERE UDF(a, d) > 5
     ) AS S
ON R.a == S.a
```

(d) Q6: UDF reduces S to 500MB and skew on R(a)

Figure 11: Query Fragments of Jobs Used in the Experimental Evaluation

from minutes to several hours, and process from gigabytes to many terabytes. Rather than showing full complex queries, we highlight in Figure 11 specific patterns and sometimes show simplified query fragments that illustrate the main ideas without providing much unnecessary detail.

Partition skew: Most queries suffer from some form of partition skew. With continuous query optimization, the optimizer detects skew right after the partitioning stage (before the expensive merge state) and modifies the plan to avoid skew if possible by choosing a different partition key.

Over-partitioning: Continuous optimization avoids over-partitioning by reducing the degree of parallelism in the plan. Query 5, shown in Figure 11(c), is a case of extreme over-partitioning. Specifically, `UDF` reduces the data volume 500 times to under $1GB$. Continuous optimization detects it, aggregates S into a single node, and changes the remaining plan into a serial plan. By avoiding overpartitioning, query 5 reduces latency by 2.9x (see Figure 12).

Under-partitioning: Under-partitioning is more dangerous than over-partitioning, because nodes process too much data, not only increasing latency but also recovery time in case of failures. For that reason, the optimizer conservatively prefers over-partitioning in absence of accurate information. Query 7 suffers from under-partitioning, and by suitably increasing the number of partitions after optimization, the query latency is reduced by 1.3x (see Figure 12).

Join variants: Figure 11(d) shows a fragment of query 6, in which R has partition skew on the join column a, causing a parallel pair-wise join to suffer from partition skew. However, predicate UDF reduces the volume of data from S by 200x to $500MB$. After optimization, we modify the plan to use a parallel broadcast join variant, which transfers the whole S to all the partitions of R and thus avoids partition skew. Query latency is reduced by 1.4x by this plan change (over 2.5x for the query fragment in the figure).

Reusing intermediate results: Intermediate materialized results are given to the optimizer as a choice to recomputation. If the size of intermediate data is much larger than the input relations, it may be cheaper to re-compute intermediate results than reading them. In most queries, the optimizer chooses to reuse intermediate results. In query $Q3$, however, the optimizer chooses to recompute $UDProcessor$ instead of using its output as a materialized view. Figure 11(b) shows the interesting fragment of this query. Originally, the optimizer chooses to partition R (the output of `UDProcessor`) on $\{a, b\}$, so that both aggregates can be computed locally on each partition without data shuffling. Later on, optimization detects that R has data skew on $\{a, b\}$ and modifies the plan to first partition on $\{a, b, c\}$, compute the aggregate, partition on $\{a, b\}$, and compute the second aggregate. In the modified plan, the optimizer recomputes $UDProcessor$, since it doubles the input data size, and reading its output is more expensive than recomputing it.

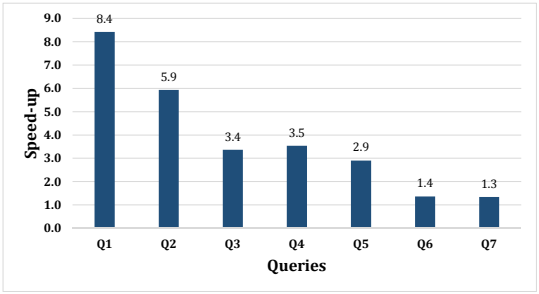


Figure 12: Query Latency Speedup

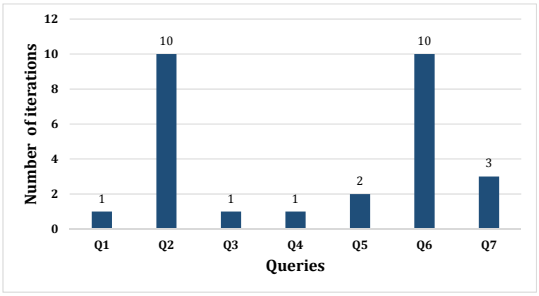


Figure 13: Number of Iterations before Convergence

Overall, the queries in this section resulted in speedup values ranging from 1.3x to 8.4x (see Figure 12).

4.3 Plan Convergence

An important practical consideration in our approach is convergence to a final plan. Figure 13 shows the number of plan changes observed while executing the queries in the previous section. We can see that the system converges to the final plan in just a couple of iterations in most cases. Queries 2 and 6 are outliers and take 10 iterations to converge. Figure 11(a) shows the relevant fragment of query 2, where the intermediate relation T has partition skew on (a, b, c) (and therefore, on all of its subsets). The aggregate and subsequent join operators require T to be partitioned on

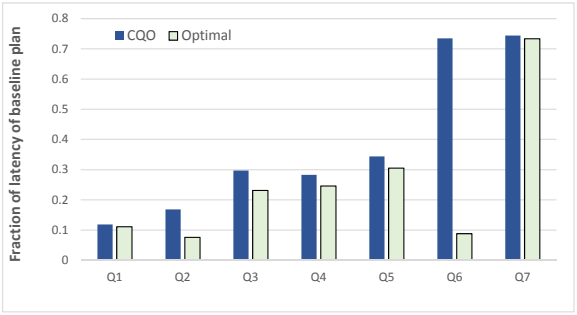


Figure 14: Latency Slowdown Compared to Optimal

any subset of (a, b, c). Without knowledge about partition skew, the optimizer tries to partition it on a (fewer partitioning columns result in cheaper executions due to fewer processing time on the partitioning function). After detecting skew, the optimizer attempts all other subsets before reaching (a, b, c). The remaining two iterations change the degree of parallelism. A more intelligent reoptimization approach that takes into account the history of previous reoptimizations would cut the number of plan changes significantly, choosing (a, b, c) directly on the second reoptimization. However, it is interesting to see that even in a scenario that changes the plan 10 times, the system converged to the final plan with an overall latency, including all *false starts*, that was almost 6 times better than for the baseline[8].

4.4 Continuous Optimization Overhead

In the previous sections we evaluated the overall improvements of our continuous optimization approach. It is also important to analyze the overhead of our proposed framework, and understand how much additional work it performs compared to a optimistic best-case scenario, where the optimizer knows *in advance* all interesting data properties. This optimal solution is of course unfeasible to implement in practice, but gives a lower bound on the execution latency of input queries. Figure 14 shows the fraction of latency that our approach (*CQO* in the figure) and the optimal approach (*Optimal* in the figure) took compared to the baseline alternative without continuous optimization.

Most queries using *CQO* are just a few percentage points above the optimal strategy, which validates our approach. The main outlier is query $Q6$. While *CQO* results in a 25% reduction in latency, the figure shows that there is vast room for improvement, as the optimal strategy reduces latency by over 90%. Query $Q6$ is one instance that changes the plan multiple times to recover from a multi-column skew. As discussed earlier, there are mechanisms to react better to multiple reoptimizations (such as more aggressively choosing partitioning columns based on the history of reoptimizations). These alternatives are part of future work.

5. RELATED WORK

Several data processing systems have been recently developed to analyze cloud-scale data like Map-Reduce [10], Hadoop [2], Dryad [13], and Hyracks [6]. These data processing frameworks provide simplified abstractions to express parallel data flow computations, but can be difficult to use for complex scenarios. To tackle the increasing complexity of user scripts, high level programming languages were proposed, including Nephele/PACT [4], Jaql [5], Tenzing [8], Dremel [20], Pig [21], Hive [24], DryadLINQ [25], and SCOPE [26, 27]. All these languages are declarative, hide system complexities, and benefit from query optimization to generate efficient distributed execution plans [17, 27].

Virtually all of optimization techniques rely on accurate data statistics to choose the best execution plan in a cost based manner. It is well known to be very difficult, and sometimes impossible, to compute good quality statistics on intermediate computation results. This problem is not unique to cloud-scale data processing systems, as traditional query optimizers also have similar issues. To deal with the uncertainties of run-time resources and parameter markers in queries, early work [9, 12] proposes generating multiple plans at compile time. When unknown quantities are discovered at run time, a decision tree mechanism decides which plan to execute. This approach suffers from a combinatorial explosion in the number of plans that need to be generated and stored at compile time.

A different approach [15, 19] uses the optimizer to generate a new plan when cardinality inconsistencies are detected. Progressive query OPtimization (POP) [19] detects cardinality estimation errors in mid-execution by comparing the estimated cardinality values against the actual run-time counts. If the actual count is outside a pre-determined validity range for that part of the plan, a re-optimization of the current plan is triggered. This approach is similar to our proposed framework as it tries to repair the current query by re-optimization. However, POP is designed for traditional centralized database systems, where query latency is small and optimization time and resource consumption is a significant proportion of total latency. Thus, POP is more conservative and uses validity ranges to trigger re-optimization, which are very difficult to compute precisely in the general case. At the same time, the underlying system in POP aggressively leverages pipelining, which reduces the instances where reoptimization can be triggered.

The DB2 LEarning Optimizer [23], in contrast, waits until a plan has finished executing to compare actual row counts to the optimizer's estimates. It learns from misestimates by adjusting the statistics to improve the quality of optimizations in future queries. Recent work [1, 7], and the basis of our approach, extends these ideas to a distributed setting, leveraging the fact that a large fraction of scripts are *parametric* and *recurring* over a time series of data. The idea is to instrument queries and piggybacking statistics collection with its normal execution, in a similar way as our approach. After collecting such statistics, it is possible to create a statistical profile that would be fed to the optimizer on a future invocation of the same job. The Stubby system [18] propose a cost-based optimizer for MapReduce systems. The system works with black box map and reduce functions and tries to find optimal configuration parameters for a MapReduce job in a cost based manner. It also collects job profile (statistics) while a job is running and uses it to optimizer similar jobs later on. These approaches focus on using query feedback to improve future query executions, but do not address improving query performance in mid-execution, which is the focus of this paper.

A more extreme approach is represented by the Eddies framework [3, 22], which does not require a complex compile-

[8]Other policies that limit the number of reoptimizations, especially in late stages of execution, might be useful in practice.

time optimizer. Instead, a run-time optimizer routes each row independently through a sequence of join operators. Per-row routing gives high opportunity for re-optimization, but imposes a big overhead in steady state. Adaptive join reordering [16] is a light-weight run-time re-optimization technique that improves both robustness and dynamically arranges the join order of pipelined join execution plans according to the observed data characteristics at run-time.

6. CONCLUSION AND FUTURE WORK

Massive data analysis in cloud-scale data centers plays a crucial role in making critical business decisions and improving quality of service. High-level scripting languages free users from understanding various system trade-offs and complexities, support a transparent abstraction of the underlying system, and provide the system great opportunities and challenges for query optimization.

In this paper, we propose a solution of *continuous query optimization* for cloud-scale systems. The approach is different from the traditional one, which focuses on collecting and propagating data statistics before query execution and utilizing them at query compilation/optimization. Instead, we continuously monitor the query execution, collect *actual* runtime statistics, and adapt execution plans as a query executes. Particularly, we seamlessly integrate the query optimizer, runtime, and scheduler components together during runtime execution. The query optimizer is triggered whenever new runtime statistics become available and generates a new optimized execution plan. If the new plan is better than the current one, we intelligently adapt the current execution plan with minimal costs. Experiments on a large-scale SCOPE production system at Microsoft show that the proposed techniques systematically solves the challenges of missing/inaccurate data statistics and improve query latency by a few folds for real-world queries.

Reoptimization opportunities are influenced by stage boundaries (i.e., by the way the system groups operators into stages). An open-ended problem and direction of future work is to make this operator grouping aware of reoptimization opportunities, rather than the current approach which performs grouping in a reoptimization-agnostic manner. Another direction for future work is to effectively exploit statistics from partially completed vertices to detect expensive predicates before a single vertex finishes execution. Finally, an open-ended question is on leveraging different conditions to trigger reoptimization, rather than the current fixed policy.

7. REFERENCES

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proceedings of NSDI*, 2012.

[2] Apache. Hadoop. http://hadoop.apache.org/.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of SIGMOD Conference*, 2000.

[4] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the ACM symposium on Cloud computing*, 2010.

[5] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[6] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of ICDE Conference*, 2011.

[7] N. Bruno, S. Agarwal, S. Kandula, M.-C. Wu, B. Shi, and J. Zhou. Recurring job optimization in scope. In *Proceedings of SIGMOD Conference*, 2012.

[8] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL implementation on the mapreduce framework. In *Proceedings of VLDB Conference*, 2011.

[9] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proceedings of SIGMOD Conference*, 1994.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI Conference*, 2004.

[11] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.

[12] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of SIGMOD Conference*, 1989.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of EuroSys Conference*, 2007.

[14] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of EuroSys Conference*, 2007.

[15] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of SIGMOD Conference*, 1998.

[16] Q. Li, M. Shao, V. Markl, K. S. Beyer, L. S. Colby, and G. M. Lohman. Adaptively reordering joins during query execution. In *Proceedings of ICDE Conference*, 2007.

[17] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. In *Proceedings of VLDB Conference*, 2012.

[18] H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for MapReduce Workflows. *PVLDB*, 5(11), 2012.

[19] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proceedings of SIGMOD Conference*, 2004.

[20] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of webscale datasets. In *Proceedings of VLDB Conference*, 2010.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of SIGMOD Conference*, 2008.

[22] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *Proceedings of ICDE Conference*, 2003.

[23] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of VLDB Conference*, 2001.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – a petabyte scale data warehouse using Hadoop. In *Proceedings of ICDE Conference*, 2010.

[25] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of OSDI Conference*, 2008.

[26] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5), 2012.

[27] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of ICDE Conference*, 2010.