Recurring Job Optimization for Massively Distributed Query Processing

Nicolas Bruno
nicolasb@microsoft.com
Microsoft Corp.

Sapna Jain
sapnakjain@gmail.com
IIT Bombay

Jingren Zhou
jrzhou@microsoft.com
Microsoft Corp.

Abstract

Companies providing cloud-scale data services have increasing needs to store and analyze massive data sets. For cost and performance reasons, processing is typically done on large clusters of tens of thousands of commodity machines. Developers use high-level scripting languages that simplify 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 describe a technique to optimize a class of jobs that are recurring *over time in a cloud-scale computation environment. By leveraging information gathered during previous executions we are able to obtain accurate statistics for new instances of recurring jobs, resulting in better execution plans. Experiments on a large-scale production system show that our techniques significantly improve cluster utilization.*

1 Introduction

An increasing number of applications require distributed data storage and processing infrastructure over large clusters of commodity hardware for critical business decisions. The MapReduce programming model [7] helps programmers write distributed applications on large clusters, but requires dealing with complex implementation details (e.g., reasoning with data distribution and overall system configuration). High level scripting languages were recently proposed to address these limitations, such as Nephele/PACT [1], Jaql [2], Hyracks [3], Tenzing [5], Dremel [15], Pig [16], Hive [18], DryadLINQ [19], and SCOPE [20, 21]. 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 [21, 13].

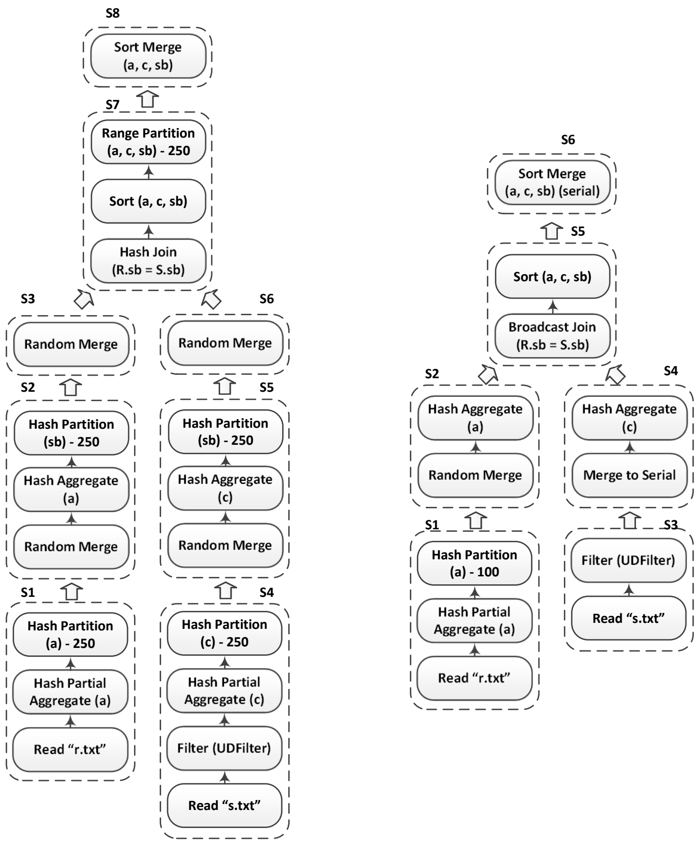
Consider the sample script shown in Figure 1(a), which first joins the result of user-defined aggregates and performs a global sort on the join result before writing the final answer. Figure 1(b) shows a possible execution plan for this script. The plan first partitions the input data sets on the group by columns (*a* and *c*, respectively) and computes user defined aggregates in parallel. It then re-partitions the aggregate outputs on the join column *sb* and evaluates the join. Finally, it range-partitions the join output on the global sort columns and locally sorts each partition.

```
R = SELECT a, UDAgg(b) AS sb
    FROM "r.txt" USING RExtractor
    GROUP BY a

S = SELECT c, UDAgg(b) AS sb
    FROM "s.txt" USING SExtractor
    GROUP BY c
    HAVING UDFilter (b, c) > 5

SELECT *
FROM R JOIN S
ON R.sb = S.sb

OUTPUT to "t.txt" USING TOutputter
```



(a) Script (b) Original Plan (c) Alternative Plan

Figure 1: Alternatives to Execute a Distributed Query

Now, suppose that the user-defined filter `UDFilter(b,c) > 5` on `s.txt` is very selective and drastically reduces the data size. In this case, it would be better to merge all data on a single machine to compute the user-defined aggregate and use a broadcast join variant (which does not require re-partitioning on the join column) to obtain the join result, as shown in Figure 1(c). This alternative also assumes that the user-defined aggregate on *r.txt* and the join are selective, so it reduces the degree of parallelism from 250 to 100 and changes the post-join range-partitioning with a serial alternative. Clearly, this alternative plan is better than the previous one for the given cardinality assumptions, but it will perform really badly if those assumptions are not valid.

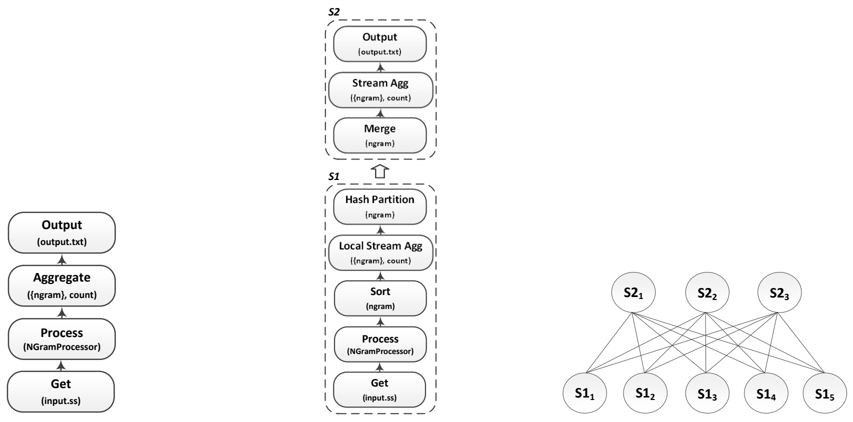
The previous examples illustrate that the choice of a plan heavily depends on properties of the input data and user-defined functions. Several aspects of highly distributed systems make this problem more challenging compared to traditional database systems. First, it is more difficult to obtain and maintain 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 over input unstructured data. Even if we can collect statistics for base tables, the nature of user scripts, which typically rely on user-defined code, makes the problem of statistical inference beyond selection and projection more difficult during optimization. Additionally, the cost of user defined code is another important source of information for cost-based query optimization. Such information is crucial for the optimizer to choose the optimal degree of parallelism for the final execution plan and when and where to execute the user code. 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, with a potential performance impact in the orders of magnitude.

At the same time, a large proportion of queries in such a cloud-scale distributed environments are *parametric* and *recurring* over a time series of data. The input datasets usually come in regularly, say, hourly or daily, and

```
SELECT ngram, COUNT(*) AS c
FROM (PROCESS
      SSTREAM "input.ss"
      USING NGramProcessor(4))
GROUP BY ngram;

OUTPUT TO "output.txt";
```

(a) SCOPE script



(b) Optimizer Input tree (c) Optimizer Output tree (d) Scheduled graph

Figure 2: Compiling and Executing in SCOPE

more importantly, they share similar data distribution and characteristics. The same business logic is thus applied to incoming datasets that are similar. We denote those jobs as *recurring*. In this work we describe mechanisms to capture data statistics concurrently with job execution, and automatically exploit them for optimizing recurring jobs. We achieve this goal by instrumenting different job stages and piggybacking statistics collection with the normal execution of a job. After collecting such statistics, we show how to feed them back to the query optimizer so that future invocations of the same (or similar) jobs take advantage of accurate statistics.

The rest of the paper is structured as follows. In Section 2 we review necessary background on distributed computation engines, using the SCOPE system for illustration. In Section 3 we describe our solution for recurring job optimization. Section 4 reports an experimental evaluation of our approach on real-world data. Section 5 reviews related work and Section 6 concludes the paper.

2 Background

We now describe the main components of distributed computation engines, using the SCOPE system as an example. The ideas and results in this paper, however, are applicable to other engines as well.

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 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. 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).

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. 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.

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. 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. During execution, a vertex reads inputs either locally or remotely. Operators within a vertex are processed in a pipelined fashion, in a way very similar to a single-node database engine. Every vertex is given enough memory and processing power to satisfy its requirements, which 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.

3 Recurring Job Optimization

The architecture of our approach to process recurring jobs is shown in Figure 3. We next describe our technique in detail by illustrating the steps a typical job follows through the system:

1. Initially, a script is submitted to the cluster for execution. The script might be recurring or new, and it is assumed to be annotated with parametric information (e.g., usually the input datasets change every day following a simple expression pattern).
2. The compiler parses the input script, performs syntax and type checking, and passes an annotated abstract syntax tree to the query optimizer. The query optimizer explores many semantically equivalent rewritings, estimates the cost of each alternative, and picks the most efficient one. While doing costing, the optimizer relies on cardinality estimates and other statistics for each plan alternative (see Section 3.4).
3. We extend the query optimizer to generate *signatures* for plan subtrees (explained in Section 3.1). During plan exploration we collect all the signatures that are associated with execution alternatives, and before implementation and costing we probe the *statistics repository* for signature matches. The repository is a new service that stores plan signatures and the corresponding runtime statistics gathered during execution of previous jobs. The optimizer relies on such feedback to produce a more effective execution plan. Signatures can be matched not only on the same recurring job, but also on similar jobs that share common subexpressions. During optimization, the optimizer instruments the resulting execution plan to collect additional statistics during execution: the resulting execution plan might contain sub-plans not yet seen by the statistics repository, and also data properties might change over time, invalidating previous estimates.

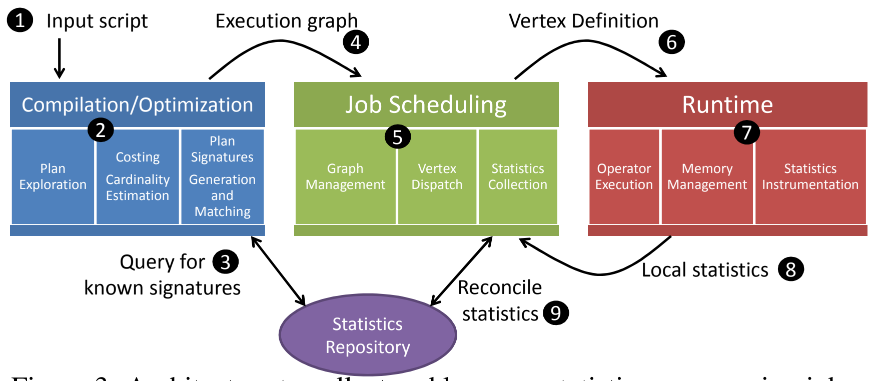


Figure 3: Architecture to collect and leverage statistics on recurring jobs.

4. The resulting execution plan is passed to the job scheduler in the form of a directed acyclic graph, where each vertex is an execution unit which looks similar to a single-node relational query plan, and each edge corresponds to data transfer due to repartitioning operators.
5. The scheduler manages the execution graph to achieve load-balancing, outlier detection and fault tolerance, among others.
6. The job manager transfers the vertex definition of new execution units to be run in cluster machines, and monitors the health and progress of each instance.
7. The runtime, which can be seen as a single-instance database engine, executes the vertex definition and concurrently collects statistics requested by the optimizer and instrumented during code generation (see Section 3.2).
8. When the vertex finishes execution, as part of the last heartbeat to the job scheduler, it sends back the aggregated statistical information, which is collected and further aggregated over all vertex instances.
9. Before finishing execution of the whole job graph, the job manager contacts the statistics repository and inserts the new statistics, to be consumed by future jobs. In case of duplicate signatures, the repository reconciles them using different policies (see Section 3.3). Periodically a background task discards statistics in the repository that exceed a certain age.

We next describe some components in additional detail.

3.1 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 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. For scalability purposes, and because many jobs are naturally recurring in our environment, we take a slightly different approach that can handle any operator tree including user-defined operators in SCOPE. This approach has some advantages over traditional view matching for our scenarios. First, we can handle any logical operator tree including used defined operators in SCOPE, which are not considered in traditional view matching. Second, signatures are very compact and easy to manipulate inside the optimizer and across components. Finally, querying and updating the statistics repository for signature matches is a very efficient lookup operation.

Specifically, for every operator subtree we traverse back the sequence of rules applied during optimization [9] until we reach the initial semantically equivalent expression[1]. Since the optimizer is deterministic, this initial

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

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[2]. All fragments producing the same result are grouped together and have the same plan signature.

Parametric signatures. We relax the construction of signatures to accept parametric scripts. The reason is that recurring scripts are virtually identical, but differ on certain constants or input names (e.g., there are predicates that filter the current day). In absence of parametric signatures, every script would result in different signatures and there would not be any possibility of reusing statistical information. In our approach, customers are required to specify that certain constants (including input names) are parameters. In that case, the signature skips the actual value of the constant and replaces it with a canonical value, which allows the system to match subexpressions that differ only on parameters.

3.2 Statistics instrumentation

The compiler instruments generated code to capture statistics during execution and thus enable recurring job optimization. Statistics need to satisfy certain properties to be useful 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.

We 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. Examples of implemented statistics 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 counter and the 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.

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

3.3 Statistics Management

The job manager maintains a *statistics package* data structure while the job is executing. A statistics package is a transient collection of plan signatures and aggregated statistics returned by completed vertices. Every time a vertex finishes execution, it sends the collected statistics to the job manager in the last heartbeat. The job manager uses the `merge` operation on statistics to aggregate all partial statistics for each signature.

The statistics repository is implemented as a key-value service, where keys are signatures and values are serializations of the corresponding statistics. Every time a job finishes, the job manager updates the statistics

[2]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).

repository with all the statistics it gathered during execution. For that purpose, it creates a new entry for each signature (or merges statistics if the signature already exists in the repository). The specific merging behavior is dictated by a policy. Simple alternatives include keeping the last instance, or performing a weighted average with aging. Additionally, the job manager stores all statistics for the job under the signature that corresponds to the root of the corresponding script, to enable the *single-job* optimization mode discussed in Section 3.5.

3.4 Optimizing Recurring Jobs

SCOPE uses a transformation-based optimizer based on the Cascades framework [9], 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 down* selection predicates, and transforming associative reducers into a local/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 recurring job optimization, we extended the optimizer to compute signatures of each expression it processes, which is done with very little overhead. Additionally, the optimizer leverages statistical information stored in the global repository. For that purpose, we modify the entry point to any 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 repository. If so, cardinality estimation is bypassed and replaced with the value found in the repository. Since all equivalent logical expressions have the same signature, this technique produces consistent results.

3.5 Optimization modes

There are two different ways to leverage statistics during optimization of recurring jobs, with different trade-offs in both performance and accuracy. The first approach, denoted *single-job mode*, performs a single call to the statistics repository with the signature corresponding to the root of the whole script, and obtains all signatures associated to the given script in previous executions. The second approach, denoted *any-job* model, performs one call to the statistics repository for each signature is about to process. The second approach is clearly more expensive as it requires a single service call per expression, but can be used to optimize queries that share script fragments with other (unrelated) previously executed jobs.

4 Experimental Evaluation

In this section, we report an experimental evaluation of an implementation of our approach on SCOPE using real queries on a large production cluster consisting of tens of thousands of machines at Microsoft. Each machine in the cluster has two six-core AMD Opteron processors running at $1.8GHz$, $24GB$ of DRAM, and four $1TB$ SATA disks. All machines run Windows Server 2008 R2 Enterprise X64 Edition. The cluster runs tens of thousands of jobs daily, reading and writing petabytes of data, powering different online services. Due to confidential data/business information, we report performance trends rather than actual numbers for all the experiments. We present a case study on a single query in Section 4.1 and show how recurring job optimization improves subsequent executions. In Section 4.2 we summarize a performance evaluation on real-world queries.

4.1 Case Study

Consider the simple query in Figure 1(a). Without knowing the selectivity of the user-defined predicate, aggregates and join operators, the optimizer generates the plan shown in Figure 1(b), which consists of eight stages and five complete data repartition stages. Stage $S1$ reads `r.txt`, computes a local aggregate, and hash-partitions the resulting data on column (a) into 250 partitions. Stage $S4$ reads *s.txt*, applies the user-defined predicate, computes a local aggregate, and hash-partitions the data on column (c) into 250 partitions. Stages $S2$ and $S5$ aggregate the tuples that belong to the same partition across all vertices, compute the user-defined aggregate and hash-partition the data on column (sb) into 250 partitions. Stages $S4$ and $S6$ aggregate the tuples that belong to same partition across all vertices. Stage $S7$ performs a pairwise join, locally sorts the join output on columns (a, c, sb) and range-partitions the data into 250 partitions. Finally, stage $S8$ aggregates the tuples that belong to the same partition across all vertices maintaining the sort order. During execution, the runtime collects different data statistics in all operators as described in Section 3.

When the recurring job is submitted the next day, the optimizer knows the selectivity of all operators and other data statistics and chooses the plan shown in Figure 1(c), which consists of only six stages with reduced degree of parallelism, only one complete data repartitioning and two partial data shuffles (moving data to a single machine). Stage $S1$ reads *r.txt*, computes a local aggregate, and hash-partitions the data on column (a) into 100 partitions (instead of 250). Stage $S2$ aggregates the tuples that belong to same partition across all vertices and computes the user-defined aggregate. Stage $S3$ reads *s.txt*, applies the user-defined predicate and computes a local aggregate. Now, instead of partitioning data on the group by column to compute a parallel aggregate, the optimizer leverages the knowledge that data volume is small, and stage $S4$ aggregates all the data on a single machine computing the user-defined aggregate serially. Another data partitioning on the join columns is avoided by using a broadcast join variant in stage $S5$, followed by a local sort of the join output on columns (a, c, sb). Since the join is very selective and significantly reduces the data volume, the repartitioning operation required by the global sort is replaced by a merge operation in stage $S8$.

By gathering and exploiting statistics, the resource consumption of the job is reduced $6.5x$ and latency is reduced $3.2x$. In this example, a significant fraction of the improvement comes from structural changes in the plan (e.g., introducing broadcast join variants), and a better determination of the degree of parallelism of repartitioning operators, which is overestimated due to complex filter predicates and user defined operators.

4.2 Performance Evaluation

In this section we report results of evaluating our approach on a real workload. We observed that roughly 40% of all the jobs submitted to our production clusters in a month have a recurring patterns and potentially benefit from recurring job optimization. We now summarize our findings on a representative workload that consists of 6 queries used internally to administer the cluster [3], and 7 queries from different customers (which include advertising and revenue, click information, and statistics about user searches). The queries in the workload have large variability in resource usage, with latencies that range from minutes to several hours, and process from gigabytes to many terabytes. We computed the resource consumption as total machine-hours used by each query with and without recurring job optimization. Figure 4 summarizes our results, and we can see that resource consumption is reduced from 10% to 80% by using recurring job optimization. Most of these improvements come from better structural plans and changes in the required degree of parallelism, as illustrated in the previous section. At the same time, we observed negligible overhead for collecting runtime statistics during query execution.

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

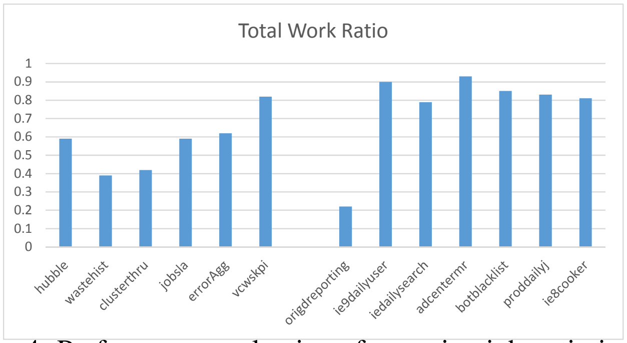


Figure 4: Performance evaluation of recurring job optimization.

5 Related work

The problem of accurate statistics estimation is not unique to cloud-scale data processing systems, as traditional query optimizers also have similar issues. To deal with parameter markers in queries, early work [6, 10] proposes generating multiple plans at compile time. When unknown quantities are discovered at runtime, 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. Progressive query OPtimization [14] uses the optimizer to generate a new plan when cardinality inconsistencies are detected. This work detects cardinality estimation errors in mid-execution by comparing the estimated cardinality values against the actual runtime 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 has high overhead of plan switching and wasted work, which need to be paid in every run of the job.

More closely related to our work is the DB2 LEarning Optimizer (LEO) [17], which waits until a plan has finished executing to compare actual row counts to the optimizer's estimates. It learns from mis-estimates by adjusting the statistics to improve the quality of optimizations in future queries. LEO does not capture and detect job recurrence over time series of data. It tries to compute adjustments to base table statistics and selectivity of predicates from output data sizes of operators, which can still be inaccurate and prone to oscillations. Our work can be seen as adapting techniques that exploit statistics on views during optimization [4, 8] to a distributed environment with explicit recurring jobs. The main contributions of our work are the generic signatures used to match plan fragments, the lightweight generated collectors for statistics, the automatic determination of which statistics to collect and store, and the policies around the statistics repository.

Some recent work [11, 12] extends self-tuning techniques to distributed environments. Without user intervention, these approaches can (i) fine tune system parameters based on workload utilization, and (ii) perform some optimizations across jobs by improving interactions between the scheduler and the underlying file system. In contrast to our work, these approaches cannot make structural decisions on how to optimize and execute individual queries, as they are not integrated with a query optimizer. Our techniques complement such approaches by additionally changing the structure of the execution plans (such as switching to broadcast join variants), and further improving cluster utilization.

6 Conclusion

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. One key optimization challenge, which is amplified in cloud-based systems, is missing accurate data statistics. In this paper, we propose a solution for recurring job optimization for cloud-scale systems. The approach monitors query execution, collects *actual* runtime statistics, and adapts future execution plans as the recurring job is subsequently submitted again. Experiments on a large-scale production system show that the proposed techniques systematically address the challenges of missing/inaccurate data statistics and improve overall resource consumption.

References

[1] D. Battré et al. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the ACM symposium on Cloud computing*, 2010.

[2] 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.

[3] 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.

[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings SIGMOD*, 2002.

[5] 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.

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

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

[8] C. Galindo-Legaria, M. Joshi, F. Waas, and M.-C. Wu. Statistics on views. In *Proceedings of VLDB*, 2003.

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

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

[11] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of VLDB Conference*, 2011.

[12] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of CIDR*, 2011.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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