Recurring Job Optimization in Scope

Nicolas Bruno Sameer Agarwal Srikanth Kandula Bing Shi Ming-Chuan Wu Jingren Zhou
Microsoft Corp.
{nicolasb,a-saagar,srikanth,binshi,mingchuw,jrzhou}@microsoft.com

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

Keywords

SCOPE, Statistics, Recurring Jobs, Distributed Computation, Query Optimization

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 [2] 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).

Recent proposals, such as SCOPE[1], raise the level of abstraction by providing a declarative language that not only increases programming productivity but is also amenable to sophisticated optimization. Like in traditional database systems, such optimization relies on detailed data statistics to choose the best execution plan in a cost-based fashion.

However, in contrast to database systems, it is very difficult to obtain and maintain good quality statistics in a highly distributed environment that contains tens of thousands of machines. First, it is very challenging to efficiently combine a large number of individually collected local complex statistical information (e.g., histograms, distinct values) in a statistically meaningful way. Second, calculating statistics typically requires scans over the full dataset. Such operation can be overwhelmingly expensive for terabytes of data. Third, 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 even more difficult during optimization. Finally, 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. It is challenging, if not impossible, to estimate its actual cost before running the query with the real dataset.

We leverage the fact that a large proportion of scripts in this environment are *parametric* and *recurring* over a time series of data. The input datasets usually come in regularly, say, hourly or daily. The same business logic is applied to different datasets in an hourly or daily fashion. We call those scripts *recurring jobs*. Although the input datasets arrive in a time series, they share a similar data distribution and characteristics. In this paper, we describe mechanisms to capture data statistics concurrently with job execution and automatically exploit them for optimizing a class of 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. We implemented this approach in the SCOPE system at Microsoft, which runs over tens of thousands of machines and processes over 30 thousand jobs daily, 40% of which have a recurring pattern.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.
ACM 978-1-4503-1247-9/12/05.

2. SOLUTION OVERVIEW

We describe the main characteristics of our approach to collect statistics at runtime and subsequently leverage them while optimizing new scripts (see also Figure 1):

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 [3]. 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.
3. We extend the query optimizer to generate *signatures* for plan subtrees (explained below). 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. Also during optimization, the optimizer instruments the resulting execution plan to collect addi-

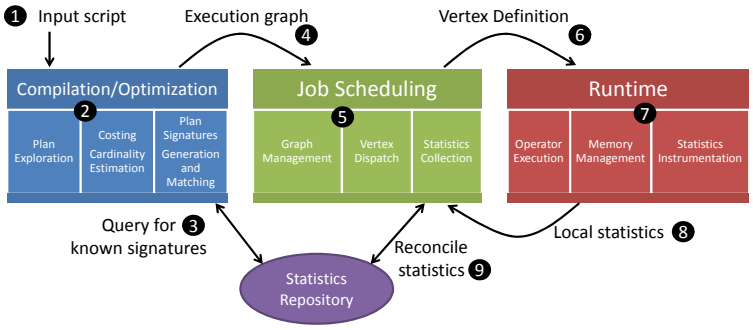

Figure 1: Architecture to collect and leverage statistics.

tional 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.

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 scheduler 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.
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 scheduler 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. Periodically a background task maintains and/or discards statistics in the repository that exceed a certain age.

We comment on some technical aspects of our approach, but omit most details due to space constraints.

What to collect: Statistics collection needs to satisfy the following properties: (i) be cheap to collect, (ii) be actionable during optimization. We compute cardinality values, average row sizes and average time to process a row by user-defined operators. Additionally, based on query requirements, we selectively gather more expensive statistics such as histograms and distributions of distinct values.

Signatures: Plan signatures are used to identify plan fragments. This is very similar to view matching technology in traditional database systems. View matching is a very flexible approach, since it obtains a canonical representation of a sub-query and is able to use compensating actions in case of partial matches. At the same time, traditional view matching does not scale well when used on all subexpressions of thousands of input queries. For scalability purposes, and exploiting the fact that many jobs are naturally recurring in our environment, we take a slightly different approach. Specifically, we serialize the representation of the canonical logical expression tree and compute a 64-bit hash value for a given query expression (we use parameterized serialization, so we do match logically equivalent trees as long as the input datasets belong to the same class). This hash value is the signature of the query expression. This approach has some advantages over traditional view matching in our scenarios. First, we can handle any logical operator tree including user defined operators in SCOPE, which are not considered in traditional view matching. Second, these 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.

3. AN EXAMPLE

Figure 2 shows an example of our techniques applied to a daily recurring job that monitors the health of the cluster and aggregates and reports various warnings and errors. After gathering and exploiting statistics, the runtime of the recurring job improves by over 30%, and incurs in 18% less I/O. In this example, a significant fraction of the improvement comes from a better determination of the degree of parallelism of expensive repartitioning operators, which is overestimated by the optimizer due to complex filter predicates and user defined operators.

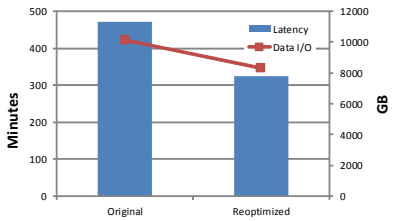

Figure 2: Performance results when using statistics.

4. REFERENCES

[1] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of VLDB Conference*, 2008.

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

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