A cost-based query optimizer for online processing of SQL queries

Aleksandar Vitorovic, Tian Guo

Dept. of Computer Science
EPFL
firstname.lastname@epfl.ch

1. MOTIVATION

There is great interest in analyzing large amounts of data through cloud computing. In order to standardize data processing, some of these systems provide database operators. Cost-based query optimization is one of the gems of database systems, and applying it in cloud systems is a challenging task. We distinguish two major types of cloud computing systems: batch and online systems. Map-reduce is a programming and computing paradigm that is suitable for analytics as a batch job. Frequently, (approximate) analytics results have to be available anytime, and batch jobs with their considerable latencies are not acceptable.

Both batch systems such as Hadoop and online systems such as Squal/Storm, wire together compute nodes to perform their computations. For example, a Hadoop job with consist of a sequence of map and reduce computations; the amount of work done in such a stage varies. Thus, ideally, the number of compute nodes for each such stage should depend on the work performed there. Hadoop does the assignment of workers to stages automatically, but very naively. Several batch systems translate SQL into MapReduce plans applying an optimizer. Hive [3] currently has a rule-based optimizer, which is able to recognize and optimize star joins. However, as mentioned in [1], Hive doesn't enforce HDFS to collocate relations that are to be joined. Haloop [2] performs well with iterative data analysis tasks. It optimizes existing MapReduce plan via reusing data across many iterations. In the context of online processing, [4] introduces a cost-based query optimizer for parallel queries, but focuses on the aspect of failures.

Our cost-based optimizer is implemented on top of Squal/Storm. Storm executes topologies (collections of wired stages) very efficiently, but requires the programmer to define the topology by hand. Squall implements relational algebra operators (select, project, distinct, aggregation) as the stages for the topology, yet translating SQL query to a topology is done manually. The topology generated from SQL is actually a query plan, and the operator performed on a stage is denoted as a query component. The goal of this work is to build a SQL to query plan translator using a cost-based optimizer which will minimize the full query execution time.

2. PROJECT DESCRIPTION

Our study will consist of three major tasks. First, we have to reveal all the factors relevant to the cost function, taking both query and execution environment into account. Second, we will develop a simple greedy-based query optimizer, as an intermediate step towards a full cost-based optimizer. Third, we plan to develop and evaluate the cost-based optimizer, which will empower all the ideas grasped during the previous two tasks.

2.1 The cost function

We have to build an appropriate model of online parallel query processing. We cannot just take the ideas from batch processing due to the following reasons. Batch processing is fundamentally different since the computation is performed in stages such that no stage starts before all the preceding stages are completely finished. Online systems propagate tuple fully as soon as it arrives, and beside the full query execution time, the average latency of the tuple is also a very important factor. In order to build a model which will estimate cost appropriately, we plan to experiment with different TPCH queries.

We already have in mind some relevant factors:

1. **Query factors:** (Intermediate) relation: size, selectivity, number of possible different join condition values.

2. **Execution environment factors:** Total number of nodes, Main memory size (sometimes against number of nodes). Scalability limit (for a given query give information about scalability factor/number of nodes which are feasible). Parallelism of data source?

3. **Query component parallelism:** Multiple query components might be pipelined within a single node. In addition, the majority of the query components (including DataSourceComponent) supports parallelism - that is, single query component might be scheduled to a user-specified number of physical nodes. Thus, data-parallelism and scalability on the operator level is achieved.

The goal of this task is to find more of these factors and assign them the appropriate weight in the cost function. The query plan will also contain a parallelism for each query component.

It would be very interesting to see whether we can model each node instantiation (partition) of a parallel query component separately in the cost function. We have to check whether this fine granularity for the cost has its justification in the final query plan chosen.

For a fixed parallelism of each DataSourceComponent, we can generate the optimal plan without perito optimal system. The basic idea is that each following component have to use minimum amount of hardware to avoid being bottleneck (this can be modeled by a formula). The only important parameter is the total number of nodes, and Selinger's approach can be used for that. The formula takes into account throughput, the number of tuples sent and number of nodes from the previous levels and computes all of this on the output of the current level. In the next revision, one might take into account:

1. overhead of parallel processing (shuffling price)

2. preaggregations

3. number of columns in a tuple

4. ever-growing data structures in memory, it might take more time to a tuple to be processed.

The problem is how to generate all the possible combinations for the parallelism of each DataSourceComponent. Some of them might be correlated.

When reasing about dynamic programming, we have to know that subqueries are considered equivalent if they have the same intermediate size and partitioning (akin to intermediate size and ordering in the Selinger paper).

The parallelism of a component might be lower bounded (3GB size, 1GB memory per node => at least 3 nodes). An upper bound is if we have 3 different groupBy (joinCondition) values we cannot have more than 3 nodes. Obviously, lower bound might be higher than upper bound, than we simply say "query cannot be executed".

The parallelism of AggregationOperator component cannot exceed the number of GroupBy different values. In Hyracks with (8, 8, 8, 5) we have load balancing problem - one node is responsible for two keys. We might alleviate this problem with CustomStreamGrouping (Storm concept). Even if we fix this, there might be skew in how many tuples are there for each key (Hive optimization i).

There is a scenario where a number of nodes and execution time cannot be achieved at the same time. Assume a query "SELECT SUM(A) FROM R, S, T WHERE R.B = S.B and S.C = T.C GROUP BY B ". Query Optimizer might generate the (R join S) join T query plan. If we want to have all the results for a particular B on a single node, additional OperatorComponent performing only Aggregation is necessary. This is course incur additional latency, so a plan R join (S join T) might be more suitable.

Interestingly, the min-bottleneck formula works for all these scenarios. It only cannot include stuff such as one-time latency (broadcast join in online-batch systems) and recoverability (such as merge join lose part of the result if one tuple fails).

Unfortunately, Storm automatically assigns stages to physical nodes, without a possibility for plugging a user-defined node allocation policy.

Cardinalty problem can be found in Cow book at 15.2.1. However, we have to model the variance of of materialized relations and number of joined tuples over the time. We estimate by comapring the number of tuples processed to the total number of tuples.

cardinality - join size result parameter

Treat each partition separately in terms of the cost? Fine granularity in terms of the cost? Take time into account.

2.2 The greedy-based optimizer

The greed-based optimizer will serve to compare against the cost-based optimizer, hopefully showing that the latter is ubiquitous for many workloads. Starting from a forest of single-relation subtrees, it will always choose to join two subtrees which has the lowest join cost. We also plan to apply projection and selection as early as possible in the query plan.

We opt to join the smallest relations (data sources) first. Since some tables might be very large to perform join between them on the first level, joining them is deferred until a small enough table for joining with is generated. Our heuristics assures that all the relations are included in the plan on second level at the latest. An improvement to this heuristics would be to take into account not only table size, but also the selectivity.

Preaggregation: prioritize tables which has more tuples to preaggregate - HyracksPreAgg, TPCH3, TPCH7 (in HyracksPreAgg it brings 10% speedup). We don't have to keep the storage for one who is going after.

Two level aggregation: first level is where the join is, the second contains as many nodes as groupBy. First level sent to the second level after n tuples is received.

HierarchyExtractor.isHashedBy - the method can be extended as in Nephele - superkey/key property. Also, the Unique property can be powerfully exploited in preaggregations.

2.3 The cost-based optimizer

Using the results from the previous two tasks, we will develop a list of queries which might profit from the cost-based optimizer, and evaluate the claims experimentally. Special attention will be dedicated to the plans which have different height of stages, exploring whether it significantly affects the full query execution time and/or latency.

The algorithm from 15.4.2 from Gehrke book has to be changed because this generates lefty plans. Either local optimizations or joining always with the highest degree intermediate join representation. Greedy based algorithm generates suboptimal solutions (tweaked for TPCH7, but do not work efficiently for TCPH8, or Hyracks). The formula for assigning parallelism works for TPCH7 (joined two small tables), but it doesn't work for Hyracks query (2 big tables), or if two tables are really small (NATION join REGION) in TPCH5, TPCH8.

3. EXPERIMENTS CONCLUSIONS

It is better if we run 2 workers per node. Network traffic between zones is bigger than network contention per node (two workers use the same network interface).

4. SCHEDULE

We divided our project into several smaller tasks. The proposed schedule for particular tasks is shown in Figure 1.

5. MILESTONE

We expect to acquire numerous ideas from tasks 1-3, so we assign one month for implementing them in the cost-based optimizer. For analyzing the results we dedicate two weeks, and for the final paper and presentation one week.

#	Task	Due date
1	TPCH experiments	12.03.2012.
2	Building the cost model	19.03.2012.
3	The greedy-based optimizer	09.04.2012.
4	The cost-based optimizer	07.05.2012.
5	Analyzing the results	21.05.2012.
6	Write the paper	28.05.2012.

Figure 1: Proposed schedule.

By the project milestone we expect to have the model and the greedy-based optimizer completed.

6. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2:922–933, August 2009.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *36th International Conference on Very Large Data Bases*, pages 285–296, Singapore, September 14–16, 2010.

[3] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2:1626–1629, August 2009.

[4] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 241–252, New York, NY, USA, 2011. ACM.