Cost-based Optimization of Complex Scientific Queries

Ruslan Fomkin and Tore Risch
Department of Information Technology, Uppsala University
Box 337, SE-751 05, Uppsala
{Ruslan.Fomkin, Tore.Risch}@it.uu.se

Abstract

High energy physics scientists analyze large amounts of data looking for interesting events when particles collide. These analyses are easily expressed using complex queries that filter events. We developed a cost model for aggregation operators and other functions used in such queries and show that it substantially improves performance. However, the query optimizer still produces suboptimal plans because of estimate errors. Furthermore, the optimization is very slow because of the large query size. We improved the optimization by a profiled grouping strategy where the scientific query is first automatically fragmented into subqueries based on application knowledge. Each fragment is then independently profiled on a sample of events to measure real execution cost and cardinality. An optimized fragmented query is shown to execute faster than a query optimized with the cost model alone. Furthermore, the total optimization time, including fragmentation and profiling, is substantially improved.

1. Introduction

Modern databases can provide tools for efficient processing of large amounts of scientific data involving complex application-specific analyses [16]. Scientific analyses can be specified as high-level queries calling user defined functions (UDFs) in an extensible DBMS. Query optimization provides scalability and high performance without any need for the scientist to spend time on low-level programming. Furthermore, as queries are easily specified and changed, new theories, e.g. implemented as filters, can be tested quickly.

Our application is High Energy Physics (HEP), where lots of data to be analyzed are generated by simulation software from the Large Hadron Collider (LHC) experiment ATLAS [1]. The data describes effects from collisions of pairs of particles. A description of a collision is called an *event*. Thus our application data are sets of independent events, where each event has properties that describe sets of particles of various types produced by the collision. Scientists define the analysis queries in terms of these *event properties*. As every collision is simulated independent of other collisions, the queries contain no joins between properties of different events. The scientist searches for events satisfying certain conditions, called *cuts*, and the query results are sets of interesting events. A typical query is a conjunction of a number of cuts. Queries over events are complex since the cuts are complex containing many predicates applied on properties of each event. The query conditions involve selections, arithmetic operators, aggregates, UDFs, and joins. The aggregates compute complex derived event properties. For example, a complex query is to look for the events producing Higgs bosons [18] by applying scientific theories expressed as cuts.

These complex queries need to be optimized for efficient and scalable execution. However, optimizing such complex queries is challenging because:

- The queries contain many joins.
- The size of the queries makes optimization slow.
- The cut definitions contain many more or less complex aggregates.
- The filters defining the cuts use many numerical UDFs.
- There are dependencies between event properties that are difficult to find or model.
- The UDFs cause dependencies between query variables.

We first investigated whether cost-based optimization improves query execution compared to no optimization. We developed a *static cost model* for the operations occurring in our kind of queries. As a comparison we also manually optimized a reference query by experimenting with different orders of cuts and measuring the actual execution times. Since the queries are very large, regular dynamic programming [27] could not be used. Instead randomized optimization [21,24,29] running for a long time and

greedy heuristic optimization [22,23] were used. Performance measurements showed that cost-based optimization produced a substantially faster execution plan (1000 times) than an unoptimized one.

For some data sets, our manually optimized plan was still somewhat faster. The main reason for this is that the static cost model becomes unreliable for large plans [20] because i) there are dependencies between query variables and ii) the cost estimate errors are compounded by the very large queries. It is difficult to define a cost model dealing with the dependencies. Another problem is that the time to optimize the query to produce a good plan is substantial; it took around half minute by randomized optimization to find a sufficiently good plan for a test query.

To alleviate this, we developed a *profiled grouping* method where the query is first split into query fragments, called *groups*, where each group has no join with other groups on event properties. Then each group is optimized separately and profiled for real execution time over a sample set of events in order to obtain measurements of actual selectivities and costs per group. Finally the join order of the groups representing the query is optimized by the cost-based query optimizer using the *profiled group cost model*.

Profiled grouping is based on measuring real execution time of different query fragments rather than static cost model estimates. In addition, the number of groups is much smaller than the number of predicates in the ungrouped query. Therefore the query optimization time is improved substantially by the grouping. Furthermore, profiled grouping turns out to be less sensitive to optimization errors, so even a greedy optimization method combined with profiled grouping produces better plans than an ungrouped approach.

An important problem is how to fragment the query. The set of all possible groups is very large and therefore a heuristic method for forming the groups is used. The *grouping heuristic* uses knowledge that in our application each event is analyzed independent of other events when selecting the events satisfying conjunctions of cuts. The grouping heuristic fragments a conjunctive query into groups where joins between groups are performed only on the event identifier; no joins are made between event properties from different groups.

We implemented the static cost model, profiled grouping, and the application query in an object-relational DBMS AMOS II [26] and evaluated the effectiveness of both ungrouped strategies and profiled grouping in combination with different optimization strategies: dynamic programming [27], randomized optimization [21,24,29], and greedy heuristic optimization [22,23]. As references we also compared with a best effort manual optimization. The measurements were made with two data sets, one with high selectivities of the cuts and one with low selectivities. We show that for high selectivity data sets profiled grouping combined with any optimization method produces better plans than the ungrouped strategies.

The rest of the paper is organized as follows. Section 2 describes the application and a test query used in the rest of the paper. The static cost model is presented in Sec. 3. Profiled grouping is described in Sec. 4. It is followed by performance measurements for the query execution strategies in Sec. 5. Related work is discussed in Sec. 6. Section 7 concludes and discusses on-going and future work.

2. High energy physics queries

Our test application analyzes data files produced by the ATLAS simulation software searching for events producing charged Higgs bosons [5,18]. Event files are associated with *meta-data conditions* for the file production that describe, e.g., experiment settings and what kinds of events were produced. A simulated collision event produces a number of general measurements about the collision and measurements about particles generated by the collision. Measurements about the collision include, e.g., missing momentum in x and y directions (*PxMiss* and *PyMiss*). Examples of generated particles are *electrons*, *muons*, and *jets*, and measurements about them are the ID-number of the type of a particle (*Kf*), momentum in x, y, and z directions (*Px*, *Py*, and *Pz*), and the amount of energy (*Ee*).

The analysis of the events consists of selecting those events that can potentially contain charged Higgs bosons. A number of predicates, called *cuts*, are applied to each event and events that satisfy all cuts are selected. Selectivities of cuts are similar for event sets from files with the same meta-data condition.

The scientists experiment with combinations of different cuts. An example of a cut, named the *three lepton cut,* is to select an event if it has exactly three *isolated leptons* and at least one isolated lepton has *Pt* bigger than 20 GeV. An isolated lepton is a lepton, which has absolute value of *Eta* smaller than 2.4 GeV and *Pt* bigger than 7 GeV. *Pt* and *Eta* are computational functions on event properties.

We implemented our application *ALEH* (Analysis LHC Events for containing Higgs bosons) as an extension of an object-relational main memory DBMS AMOS II [26]. The events are delivered in binary files managed by the ROOT library [7]. A ROOT wrapper is implemented to load events from ROOT files into

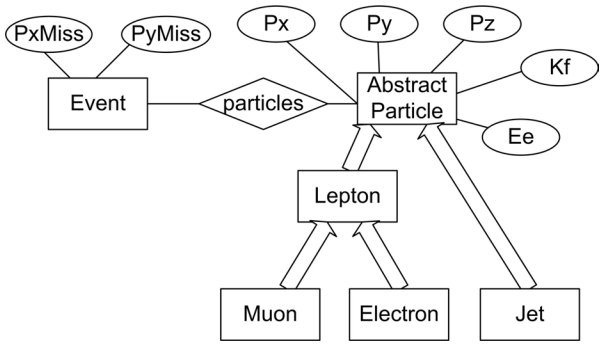

Figure 1. Schema of the application data.

main memory. An object-relational schema of the collision events is implemented in the query language AmosQL [13][1] and presented in Fig. 1. Events are represented by *Event* entity with two attributes *PxMiss* and *PyMiss*. Particles are represented by objects with attributes *Kf*, *Px*, *Py*, *Pz*, and *Ee* and relationships to *Events*. Particles of different types are represented by different entity subtypes *Muon*, *Electron*, and *Jet*. *Muon* and *Electron* are generalized by an abstract entity *Lepton*, which is used in definitions of some cuts.

A number of numerical UDFs, e.g. *Pt* and *Eta*, are defined in the database in order to make the analyses. The cuts are expressed as functions (parameterized queries) in terms of these UDFs in the query language AmosQL. The analysis is usually defined as conjunctions of several different cuts, where each cut is defined as a conjunction of many predicates. As each event is always analyzed independently of other events, the analysis queries have the important property that no joins are performed between events. In general the queries have the form $\{e \mid d(e) \wedge c_1(e) \wedge c_2(e) \wedge ...\}$, where c_i are cuts and *d(e)* is a domain predicate to scan the events.

For example, a general query is:

```
SELECT ev
FROM Event ev
WHERE jetVetoCut(ev) AND
      zVetoCut(ev) AND
      topCut(ev) AND
      missEeCuts(ev) AND
      leptonCuts(ev) AND
      threeLeptonCut(ev); .
```
(1)

Here the functions *jetVetoCut*, *zVetoCut*, *topCut*, *missEeCuts*, *leptonCuts*, and *threeLeptonCut* are examples of cuts that provide necessary conditions for the collision event *ev* to produce a Higgs boson according the theory in [5,18][2]. The domain predicate is generated by the FROM clause. This general query is a reference query for the rest of the paper.

For example, the definition of the three lepton cut is:

```
CREATE FUNCTION threeLeptonCut
  (Event ev) -> Boolean AS
SELECT TRUE
WHERE COUNT(isolatedLeptons(ev))=3
  AND SOME(SELECT r
    FROM Real r
    WHERE r=Pt(isolatedLeptons(ev))
      AND r>20.0);
```

The function *isolatedLeptons* has the definition:

```
CREATE FUNCTION isolatedLeptons
  (Event ev) -> Lepton AS
SELECT l FROM Lepton l
WHERE  l=leptons(ev) AND
       Abs(Eta(l))<2.4 AND Pt(l)>7.0;
```

The *Pt* and *Eta* functions call UDFs *Pt* and *Eta* over momentum triple for the given particle *l*. The formulas of *Pt* and *Eta* are:

$$\sqrt{x^2 + y^2} \text{ and} \tag{2}$$

$$0.5 \cdot \ln\left(\frac{\sqrt{x^2 + y^2 + z^2} + z}{\sqrt{x^2 + y^2 + z^2} - z}\right) \text{ respectively.} \tag{3}$$

Before query optimization, functions are expanded as views and the query is represented in domain calculus. The plan for the query (1) is a conjunction of 51 predicates. The predicates are comparisons, numerical operations, aggregates, UDF calls, and joins. The large size of the query makes it difficult to optimize and dynamic programming [27] cannot be used. We were able to optimize it using randomized optimization [21,24,29], which, however, uses a lot of time to produce a good plan.

Another problem is that there are many dependencies between predicates. This makes it difficult to estimate the cost. For example, a part of an unoptimized predicate in the definition of function *isolatedLeptons* is the conjunction:

```
Eta(m) = em
Abs(em) = aem
aem < 2.4
Pt(m) = pm
pm > 7.0
```

Here *m* is the momentum triple of a lepton of a given event, and *em*, *aem*, and *pm* are query variables containing results of the UDFs *Eta*, *Abs*, and *Pt*. It is

[1] The source code of the schema in AmosQL can be found in http://user.it.uu.se/~ruslan/ALEH/schema.amosql.txt.

[2] Definitions of the cuts in AmosQL can be found in http://user.it.uu.se/~ruslan/ALEH/cuts.amosql.txt.

difficult to estimate selectivities for such predicates defined in terms of UDFs. For example, the estimate of the selectivity of the comparison *aem* < 2.4 depends on original data distribution of event properties and on the distribution of results from the functions *Eta* and *Abs* that are applied on the these properties to calculate *aem*. Because of the data dependencies the selectivity estimates contain large errors. The same holds for the comparison *pm* > 7.0, etc. Furthermore, there is also a dependency between the two comparisons, as they operate on the same event properties. Such dependencies influence cost and cardinality estimates and suboptimal execution plans are chosen [20].

To alleviate the problems of slow optimization and data dependencies, we investigated a profiled grouping strategy based on measuring real costs of query fragments. Each group is individually optimized using the static cost model. Then the optimized groups are profiled over event set samples. Finally, the so obtained profiled group cost model is used to optimize the fragmented query. In our measurements, we compare this approach to a cost-based approach without applying the profiled grouping method.

3. Static cost model

We developed a cost model for aggregates and numerical operations used in our application, assuming data independence between predicates. Tables 1 and 2 in the Appendix define the static cost model for the operators. The static cost model is rather ad hoc, but, as will be shown, it still produces good execution plans for our test query, in particular in combination with profiled grouping. It is defined so that the operators are comparable. For example, *SOME* and *NOTANY* should be complementary and *SOME* is a special case of *ATLEAST*.

The costs of complex numerical operators are approximated according their measured execution time. The cost of basic numerical operators such as *plus*, *minus*, and *times* is set to one. The costs for the numerical operators that are used in the ALEH query are presented in Table 1 in Appendix. The selectivities of the numerical operators are always one.

The costs and cardinalities of aggregates are based on the estimated costs and cardinalities of their subqueries. The estimates of the cost and the cardinality of a subquery assume independence between predicates of the subquery.

The cost of an aggregate depends on the estimated number of tuples produced by its subquery *sq*. For aggregate *SUM*(*sq*) all tuples emitted by *sq* have to be processed, while for other aggregates, such as *SOME* and *NOTANY,* only a limited number of tuples emitted by *sq* are processed. Therefore the cost of an aggregate is the cost of producing the required tuples by *sq* plus the cost of processing the emitted tuples by the aggregation operator. The cost per produced tuple by subquery *sq* is the estimated total cost of executing the subquery, *cost*(*sq*), divided by its estimated cardinality, *card*(*sq*), i.e. $\frac{cost(sq)}{card(sq)}$. The cost for the aggregation operator to process one received tuple from *sq* is set to one. For example, *SUM*(*sq*) has the cost *cost*(*sq*)+*card*(*sq*). The cost of *SOME*(*sq*) when *card*(*sq*)<1 is *cost*(*sq*). If *sq* emits at least one tuple the cost becomes $\frac{cost(sq)}{card(sq)}+1$ since only the first tuple is processed by SOME. Analogous cost model formulas are developed for other aggregation operators (Table 1).

The selectivity of *SUM*(*sq*) is always one. The selectivities of *SOME*(*sq*) and *NOTANY*(*sq*) depend on the estimated selectivity of *sq*. If *sq* emits fewer than one result tuple the selectivity of *SOME*(*sq*) is set proportional to *card*(*sq*), $\frac{card(sq)}{2}$. Otherwise it is set to $1-\frac{1}{2\cdot card(sq)}$. Basically, the model converges to one as *card*(*sq*) increases since it becomes more and more likely that *SOME* is true. The factor two allows *NOTANY* to have a complementary model (see Table 1).

The selectivity of predicate *COUNT*(*sq*)=*N*, and *ATLEAST*(*sq*)=*N*, where *N* is known, depends on the relationship between *N* and *card*(*sq*). For example, for *COUNT*(*sq*)=*N* if *card*(*sq*)<*N* the selectivity is increasing until *N* tuples are emitted from *sq*, and it is computed as $\frac{card(sq)}{3\cdot N}$. After *N* tuples are emitted the selectivity goes down and is therefore computed as $\frac{N}{3\cdot card(sq)}$. The selectivity is set to 1/3 when *card*(*sq*) is estimated to be *N*.

4. Profiled grouping

The profiled grouping fragments the query into groups where the groups are joined only on the event variable *e*. The groups are *minimal* in the sense that none of the groups can be split further into subgroups joined only on the event variable *e*. Thus, a fragmented query has the form $\{e \mid d(e) \wedge g_1(e) \wedge g_2(e) \wedge ...\}$, where $d(e)$ is the domain predicate and $g_i(e)$ are groups and $g_i(e)$ cannot be further fragmented, i.e.

$\neg\exists(g_{i1}(e), g_{i2}(e)) : g_i(e) = g_{i1}(e) \wedge g_{i2}(e)$. Notice that the original cuts do not fulfill the minimality as some of the cuts can be split into further groups. For example, the definition of *threeLeptonCut* forms two groups:

```
COUNT(isolatedLeptons(ev))=3
```

and

```
SOME(SELECT r
     FROM real r
     WHERE r=Pt(isolatedLeptons(ev))
       AND r>20.0)
```

The result of the grouping is a set of subqueries where each predicate from the original query belongs to exactly one group.

After the groups are formed each group is optimized using the static cost model and assuming that *e* is bound by the domain predicate *d*(*e*). Both randomized and greedy optimization was used and compared, with no significant impact on the final execution efficiency. Therefore, in our measurements we show the time to do the cheap greedy optimization only.

Since each group is a complex conjunctive query a static cost model may not produce good estimates [20]. Therefore we wrap each group and profile it on a sample of the set of events that are queried. This requires that the queried data are already loaded to the main memory by the ROOT wrapper. The profiler executes each group on the same sample set and calculates selectivity and real cost estimates for each group and these estimates are then used for cost-based reordering of the groups.

In the experiments we varied the number of events used in the sample set. Based on this we estimated the required sample size to obtain sufficiently efficient optimization.

Finally, the join order of groups is optimized using the profiled group cost model obtained by the profiling.

Figure 2 shows our grouping algorithm. The input is a conjunctive query predicate *S* and an event variable *varE*. The output is a conjunction of groups, *Groups,* representing *S*. On lines (3-5) the algorithm forms a new group by picking one predicate at a time from *S*. The variable *V* will contain the set of variables to be processed in order to form the group. On line (6) *V* is initialized to the variables in *p*, except the event variable. On lines (7-9) the algorithm processes one variable at a time from *V* and on lines (10-11) it searches for all predicates that use the processed variable. Each predicate using the processed variable is added to the new group on lines (12-13) and its other variables are added to set of unprocessed variables *V* on line (14). The group is formed on line (15) when no more variables in the group need to be processed. The

```
 1: Groups = {}
 2: while (S != {})
 3:  pick a predicate p from S
 4:  S = S \ p
 5:  G = {p}
 6:  V = variables(p) \ varE
 7:  while (V !={})
 8:   pick a variable v from V
 9:   V = V \ v
10:   for each q in S
11:    if v   variables(q) then
12:     G = G∪q
13:     S = S \ q
14:     V = V∪variables(q)\{v,varE}
15:  Groups = Groups ∪ {G}
16: return Groups
```

Figure 2. Pseudo-code of the grouping algorithm.

algorithm stops forming groups when all predicates in *S* have been moved to some group in *Groups*.

5. Performance measurements

To investigate the effectiveness of our approaches we evaluated the following strategies both with respect to execution time and time to do the optimization:

Unoptimized plan (UNOPT). The unoptimized plan is obtained directly from our query (1) by using a very *simple cost model*, where all aggregate operations have the same cost and all UDFs also have the same cost. Thus the query optimizer does not change the order of aggregates and UDFs and their execution order is the same as the order of the cuts in the query.

Best manual effort plan (MAN). We use the same simple cost model as for UNOPT but we manually reordered the plan, by extensive experimentation with different cut orders, to get the plan that was fastest to execute. The best effort query formulation is:

```
SELECT ev
FROM Event ev
WHERE  threeLeptonCut(ev)
  AND  leptonCuts(ev)
  AND  missEeCuts(ev)
  AND  zVetoCut(ev)
  AND  topCut(ev)
  AND  jetVetoCut(ev);
```

Ungrouped strategies (UR and UG). Query (1) was optimized without grouping using the static cost model after the database was populated. Because of the large number of predicates in the query, the query optimizer could not use dynamic programming. Instead

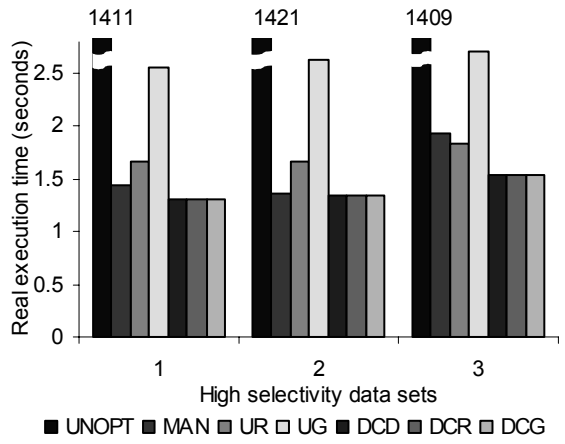

Figure 3. Comparing execution times for three data sets with high selectivity.

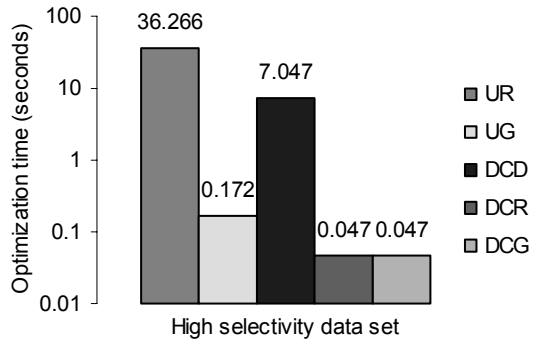

Figure 4. Comparing optimization time (logarithmic scale).

randomized optimization (UR) [21,29] and greedy optimization (UG) [22,23] were used. For randomized optimization we used a mixture of iterative improvement and sequence heuristics [24] to obtain a likely optimal plan. We first made extensive experiments to determine the minimal number of iterations to get a converged plan. For comparing optimization time of UR with other strategies we used the time to find the converged plan. This can be regarded as a best case for the time to do randomized optimization.

Profiled group cost model (DCD, DCR, and DCG). We evaluated our profiled grouping strategy. Because the grouping decomposes a flat query with 51 predicates to a join of 8 groups, dynamic programming optimization can be used to optimize the join order of the groups (DCD). We also optimized the group join order using randomized (DCR) and greedy (DCG) optimization.

5.1 Experimental setup

The experiments were performed on a PC with an Intel Pentium 4 CPU 2.40 GHz, 1 GB of RAM.

The same large query (1) was used in all the performance studies. As test cases we used real data sets produced by ATLAS. The evaluation was first performed on data sets with high query selectivity, where only 0.0013% of the events satisfy the query. Each data set contains 25000 events. As comparison, the performance was also measured for a data set with low query selectivity where 10% of the events passed the query. It contained 8623 events.

5.2 Experimental results

Figure 3 shows the execution times for three data sets with high query selectivity. All optimization strategies (MAN, UR, UG, DCD, DCR, or DCG) produced plans being a factor 1000 faster than the unoptimized plan (UNOPT), so optimization certainly pays off. Profiled grouping strategies (DCD, DCR, and DCG) perform best for all three data sets, independent on what optimization method is used for joining the groups. The best ungrouped strategy (UR) produces a plan that performs 18% worse than any of the profiled grouping strategies. Not surprisingly, randomized optimization for ungrouped queries (UR) produced much better plans than corresponding greedy optimization (UG).

Figure 4 measures the time to do the query optimization. All profiled grouping strategies (DCD, DCR, and DCG) are significantly faster than ungrouped randomized optimization (UR). With profiled grouping both randomized (DCR) and greedy (DCG) optimization methods find the same optimal plan much faster than dynamic programming (DCD). Ungrouped greedy optimization UG is rather fast but it produces a bad execution plan (Fig. 3).

The effectiveness of DCD, DCR, and DCG also depends on the profiling time. The profiling should be done for every query so this adds to the optimization time. The query execution performance for different profiling sample sizes is presented in Figure 5. The performance is independent of the optimization method (DCD, DCR, or DCG) but is proportional to the sample size. Different data sets require different sample sizes for optimal query performance. Query plans that were obtained with small samples are noticeably worse than query plans with large samples. The smallest sizes of the samples for which good plans are produced depend on the data sets. For example, good plans for data set one starts with a sample size of 40 events, taking approximately 5.5 seconds to profile. Data set two

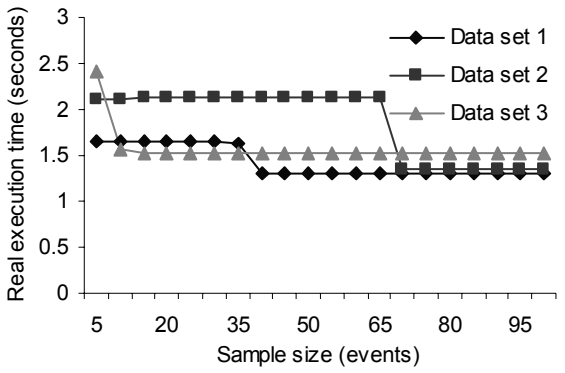

Figure 5. Execution performance for different sample sizes.

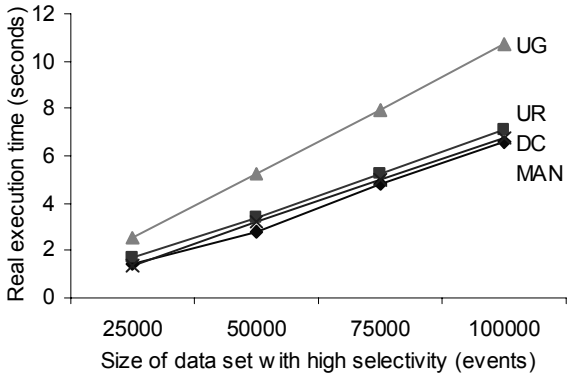

Figure 6. Scaling the data size with high selectivity queries.

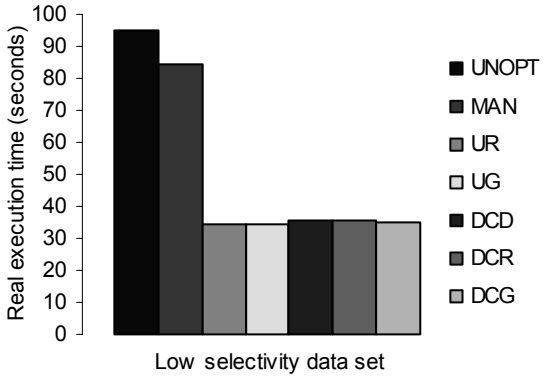

Figure 7. Comparing optimization strategies for low selectivity data.

requires 70 events (9.5 seconds), and data set three requires 15 events (2 seconds). Based on these measurements the sample sizes are conservatively set to 70 by default. The user can tune the system by changing the sample size. Notice that, even with the conservative sample setting ungrouped randomized optimization (UR) is still much slower to optimize than grouped optimization when adding the profiling time.

In Figure 6 we investigate the execution times of the optimization strategies when scaling the data size with the high selectivity data sets. With profiled grouping all three optimization methods find the same optimal plan and therefore the three strategies are presented as one curve (DC). The profiled group cost model for the query was obtained by profiling only data set one on the first 40 events. The measurements were done for 25 000 events (data set one), 50 000 events (data sets one and two), 75 000 events (data sets one, two, and three), and 100 000 events (data sets one, two, three and one more). The reference query was optimized using the static cost model (UR, UG) for each size of the data set. The execution time increases linearly with the data set size, since all events of a data set are always processed. The query plan from the profiled grouping strategies performs always better than any query plan from an ungrouped strategy.

The profiled grouping strategies scaled well using an execution plan obtained by profiling a single sample. This indicates that the profiled group cost model can be obtained once on a single sample data set and then it can be used for all data sets having the same query selectivity. We assume that data sets having the same meta-data condition also have the same selectivity.

Finally, Figure 7 shows the performance for a data set with low query selectivity. Here the impact of query optimization is less significant. The manual plan turns out to be slower than any optimized plan since it was obtained for high selectivity data sets. A new manual plan would have to be developed here (with great effort). This shows that automatic query optimization can improve the effectiveness of the scientists, in particular since they currently implement the cuts in C++ manually [7] using manual optimization. The profiled grouping strategies (DCD, DCR, and DCG) performed 5% worse than the ungrouped strategies (UR and UG), indicating that the grouping here provides less good heuristics.

6. Related work

Most work on cost-based query optimization concentrates on statistics for adequate cost estimates in Select-Project-Join (SPJ) queries only, e.g. [4,12,19,25,28]. A number of works present optimization of queries with UDFs [6,9,17] not including cost models for aggregates. [10,14,30] describe optimizations of queries with aggregates without defining cost model for aggregates. In [8] the cost of GROUP BY in SQL is estimated solely by the number of expected groups in order to push it down the query plan, without considering the costs of executing

predicates. By contrast, we developed a cost model for aggregates used in scientific queries that takes into account both the cardinality and costs of the subqueries and the cost of executing the aggregation operator itself.

Eddies [2,11] optimize queries at run time per tuple, not taking expensive UDFs into account. This incurs a run time overhead compared to preoptimized queries.

Recent work in [3,15] use statistics on views to handle data dependencies during optimization of simple SPJ queries, without collecting statistics for large numerical queries or automatically fragmenting the queries into views.

In our approach, to alleviate the data dependency problem, we complement a simple static cost model for scientific queries by fragmenting queries into groups. Then we dynamically generate a profiled group cost model by measuring real executions over data samples. The profiled group cost model is used for join ordering of the groups.

7. Conclusion and future work

We developed a static cost model for aggregation operators and functions used in scientific queries from the ATLAS project. We showed that optimization of large scientific queries can reduce execution time by a factor 1000. Automatic query optimization can improve the effectiveness of the scientists, in contrast to manually implementing the queries in C++ [7] as they currently do. Furthermore, data sets from different experiments will have different optimal execution plans and it is very costly to manually construct them.

Scientific work in particle physics includes experimenting with different cuts to implement new theories [5,18]. The flexibility to specify the cuts using non-procedural database queries could improve the effectiveness of the scientific work.

Complex scientific queries are very large having many predicates. This makes cost-based optimization difficult and slow. Furthermore, the predicates contain many dependent variables. It is difficult or even impossible to define a reliable cost model dealing with large predicates with many dependencies. Therefore, as an alternative, we developed a new method, *profiled grouping*, where the query is first fragmented into groups and then the execution of each group is measured on samples of real data. The profiled group cost model is finally used in cost-based optimization of the group join order.

We evaluated both the static cost model and the profiled grouping method on real data. We investigated the time to do the optimization for both approaches and with different optimization strategies, i.e. dynamic programming, randomized optimization, and greedy optimization. Our results show that the profiled grouping gives significant improvement in optimization time compared with an ungrouped strategy and produces better execution plans. A greedy approach with the static cost model also has fast optimization, but the plan is around twice slower than the other plans. Still, it is shown to be much better than no optimization at all.

There are several issues for future work. One issue is how to dynamically adapt the group statistics during query execution. Another one is to investigate the impact of collecting more detailed statistics on groups. It should also be investigated whether our grouping heuristic can be improved further. The static cost model can be evaluated more carefully. It should also be evaluated how well query selectivities correlate with meta-data conditions.

Currently our approach requires loading queried data into main memory. This is insufficient if the amount of data to analyze is bigger than the amount of available main memory. We are investigating a stream approach, where events are streamed one by one from a file into the DBMS. Since events are queried independently from each other only one event has to be stored in the main memory at a time.

An open question how difficult it is to define a cost model dealing with all the dependencies for large scientific queries, or if it is even possible to define it.

8. References

[1] The ATLAS experiment, 2006, http://atlas.ch/index.html.

[2] R. Avnur and J.M. Hellerstein, "Eddies: continuously adaptive query processing", *SIGMOD*, pp. 261-272, 2000.

[3] N. Bruno and S. Chaudhuri, "Conditional selectivity for statistics on query expressions", *SIGMOD*, pp. 311-322, 2004.

[4] N. Bruno, S. Chaudhuri, and L. Gravano, "STHoles: a multidimensional workload-aware histogram", *SIGMOD*, pp. 211-222, 2001.

[5] M. Bisset, F. Moortgat, and S. Moretti, "Trilepton + top signal from chargino-neutralino decays of MSSM charged Higgs bosons at the LHC", *Eur.Phys.J. C30*, pp. 419-434, 2003.

[6] J. Boulos and K. Ono, "Cost estimation of user-defined methods in object-relational database systems", *SIGMOD Rec. 28(3)*, pp. 22-28, 1999.

[7] R. Brun and F. Rademakers, "ROOT - An object oriented data analysis framework", *Proc. AIHENP'96 Workshop*, *Nucl. Inst. & Meth. In Phys. Res. A 389*, pp. 81-86, 1997. See also http://root.cern.ch/.

[8] S. Chaudhuri and K. Shim, "Optimizing queries with aggregate views", *EDBT*, pp. 167-182, 1996.

[9] S. Chaudhuri and K. Shim, “Optimization of queries with user-defined predicates”, *TODS 24(2)*, pp. 177-228, 1999.
[10] S. Cohen, “User-defined aggregate functions: bridging theory and practice”, *SIGMOD*, pp. 49-60, 2006.
[11] A. Deshpande and J.M. Hellerstein, “Lifting the burden of history from adaptive query processing”, *VLDB*, pp. 948-959, 2004.
[12] C. Estan and J.F. Naughton, “End-biased samples for join cardinality estimation”, *ICDE*, p. 20, 2006.
[13] S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, and M. Sköld, “Amos II Release 8 User's Manual”, Uppsala DataBase Laboratory, 2006. Available at http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html
[14] C. Galindo-Legaria, M. Joshi, “Orthogonal optimization of subqueries and aggregation”, *SIGMOD*, pp. 571-581, 2001.
[15] C.A. Galindo-Legaria, M. Joshi, F. Waas, and M. Wu, “Statistics on views”, *VLDB*, pp. 952-962, 2003.
[16] J. Gray, D.T. Liu, M.A. Nieto-Santisteban, A. Szalay, D.J. DeWitt, and G. Heber, "Scientific data management in the coming decade”, *SIGMOD Record 34(4)*, pp. 34-41, 2005.
[17] J.M. Hellerstein, “Optimization techniques for queries with expensive methods”, *TODS 23(2)*, pp. 113-157, 1998.
[18] C. Hansen, N. Gollub, K.Assamagan, and T. Ekelöf, “Discovery potential for a charged Higgs boson decaying in the chargino-neutralino channel of the ATLAS detector at the LHC”, *Eur.Phys.J. C44S2*, pp. 1-9, 2005.
[19] P.J. Haas and A.N. Swami, “Sequential sampling procedures for query size estimation”, *SIGMOD*, pp. 341-350, 1992.
[20] Y.E. Ioannidis and S. Christodoulakis, “On the propagation of errors in the size of join results”, *SIGMOD*, pp. 268-277, 1991.
[21] Y.E. Ioannidis, and Y. Kang, "Randomized algorithms for optimizing large join queries”, *SIGMOD*, pp. 312-321, 1990.
[22] R. Krishnamurthy, H. Boral, and C. Zaniolo, “Optimization of nonrecursive queries”, *VLDB*, pp. 128-137, 1986.
[23] W. Litwin, and T. Risch, “Main memory oriented optimization of OO queries using typed datalog with foreign predicates”, *TKDE 4(6)*, pp. 517-528, 1992.
[24] J. Näs, “Randomized optimization of object oriented queries in a main memory database management system”, *Master's Thesis No: LiTH-IDA-Ex-93/25*, 1993. Available at http://user.it.uu.se/~udbl/Theses/JoakimNasMSc.pdf
[25] V. Poosala and Y.E. Ioannidis, “Selectivity estimation without the attribute value independence assumption”, *VLDB*, pp. 486-495, 1997.
[26] T. Risch, V. Josifovski, and T. Katchaounov, “Functional data integration in a distributed mediator system”, In P. Gray, L. Kerschberg, P. King and A. Poulovassilis (eds.), *Functional Approach to Data Management - Modeling, analyzing and integrating heterogeneous data*, Springer, 2003.
[27] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, “Access path selection in a relational database management system”, *SIGMOD*, pp. 23-34, 1979.
[28] U. Srivastava, P.J. Haas, V. Markl, M. Kutsch, and T.M. Tran, “ISOMER: consistent histogram construction using query feedback”, *ICDE*, p. 39, 2006.
[29] A. Swami and A. Gupta, “Optimization of large join queries”, *SIGMOD*, pp. 8-17, 1988.
[30] W.P. Yan and P. Larson, “Eager aggregation and lazy aggregation”, *VLDB*, pp. 345-357, 1995.

Appendix

Table 1. Costs of ALEH numerical operations, where x, y, and z are numbers (integers or reals), i is integer, v, v1, v2, and v3 are vectors, and vs is bag of vectors.

Numerical operation	Cost	Formula
PLUS(*x*,*y*)=*z*	1	$x+y=z$
TIMES(*x*,*y*)=*z*	1	$x \cdot y = z$
ABS(*x*)=*y*	1	y is absolute value of x
v[*i*]=*x*	1	x is element i of vector v
TIMES(*v1*,*v2*)=*x*	5	x is scalar product of two vectors *v1* and *v2*
SQRT(*x*)=*y*	1	y is square root of x
PLUS(*v1*,*v2*)=*v3*	15	$v3[i]=v1[i]+v2[i]$ for all i
LOG(*x*)=*y*	2	y is natural logarithm of x
ATAN2(*x*,*y*)=*z*	2	z is arctangent of x/y
CEILING(*x*)=*y*	1	y is ceiling of x
COS(*x*)=*y*	2	y is cosine of x
MAGNITUDE(*v*)=*x*	9	$x = \sqrt{v[0]^2 + v[1]^2 + v[2]^2}$, where v is a 3D vector
ETA(*v*)=*x*	16	$x = 0.5 \cdot \ln\left(\frac{\sqrt{v[0]^2 + v[1]^2 + v[2]^2} + v[2]}{\sqrt{v[0]^2 + v[1]^2 + v[2]^2} - v[2]}\right)$, where v is a 3D vector
PT(*v*)=*x*	6	$x = \sqrt{v[0]^2 + v[1]^2}$, only first two dimensions of 3D vector v are used in the calculation
SUM(*vs*)=*v*	36	v is sum of all vectors in bag of vector *vs*

Table 2. Cost model for aggregation operators over subquery sq, where cost(sq) is the estimated total cost of executing sq, and card(sq) is the estimated cardinality of sq.

Aggregate	Cost	Selectivity
SOME(*sq*)	if $card(sq) < 1$ then $cost(sq)$ else $\frac{cost(sq)}{card(sq)} + 1$	if $card(sq) < 1$ then $\frac{card(sq)}{2}$ else $1 - \frac{1}{2 \cdot card(sq)}$
NOTANY(*sq*)	if $card(sq) < 1$ then $cost(sq)$ else $\frac{cost(sq)}{card(sq)} + 1$	if $card(sq) < 1$ then $1 - \frac{card(sq)}{2}$ else $\frac{1}{2 \cdot card(sq)}$
COUNT(*sq*)=*N*	if $card(sq) < N+1$ then $cost(sq) + card(sq)$ else $(N+1)\frac{cost(sq)}{card(sq)} + N + 1$	if $card(sq) < N$ then $\frac{card(sq)}{3 \cdot N}$ else $\frac{N}{3 \cdot card(sq)}$
ATLEAST(*sq*)=*N*	if $card(sq) < N$ then $cost(sq) + card(sq)$ else $N\frac{cost(sq)}{card(sq)} + N$	if $card(sq) < N$ then $\frac{card(sq)}{2 \cdot N}$ else $1 - \frac{N}{2 \cdot card(sq)}$
SUM(*sq*)	$cost(sq) + card(sq)$	1