Optimization and Execution of Complex Scientific Queries over Uncorrelated Experimental Data

Ruslan Fomkin and Tore Risch

Department of Information Technology, Uppsala University,
Box 337, SE-75105 Uppsala, Sweden
{Ruslan.Fomkin,Tore.Risch}@it.uu.se

Abstract. Scientific experiments produce large volumes of data represented as complex objects that describe independent events such as particle collisions. Scientific analyses can be expressed as queries selecting objects that satisfy complex local conditions over properties of each object. The conditions include joins, aggregate functions, and numerical computations. Traditional query processing where data is loaded into a database does not perform well, since it takes time and space to load and index data. Therefore, we developed SQISLE to efficiently process in one pass large queries selecting complex objects from sources. Our contributions include runtime query optimization strategies, which during query execution collect runtime query statistics, reoptimize the query using collected statistics, and dynamically switch optimization strategies. Furthermore, performance is improved by query rewrites, temporary view materializations, and compile time evaluation of query fragments. We demonstrate that queries in SQISLE perform close to hard-coded C++ implementations of the same analyses.

1 Introduction

Large volumes of data produced and shared within scientific communities are analyzed by many researchers to test scientific hypotheses. For example, in High Energy Physics (HEP) a lot of data is generated by simulation software from the Large Hadron Collider (LHC) experiment ATLAS [4]. The data is stored in files and describes effects from collisions of particles.

This paper investigates the use of query processing techniques to implement such scientific applications. Data is represented as complex objects describing measurements of independent real-world events. The analyses are selections of events applying conjunctions of complex numerical filters on each complex object.

In ATLAS, a *collision event* generates measurements of new particles summarized as a complex object. Generated objects are stored in files, which are read by the data management library ROOT [7]. Since every collision is performed independently from others, each complex object describing an event is analyzed separately and there are no joins between data from different events. Therefore each ROOT file can be processed in one pass per query as a stream of objects describing collision events. Physicists test their hypotheses on these data by selecting interesting events. An example of a scientifically interesting event is a collision event which is likely to produce Higgs bosons [5,11]. A collision event is interesting if it satisfies some

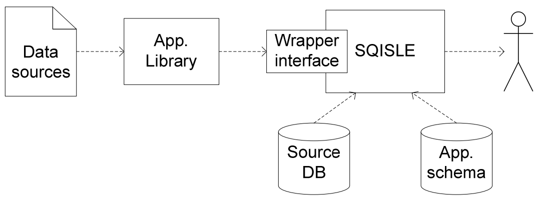


Fig. 1. Architecture of SQISLE with data flow

conditions, called *cuts*, specified over the object describing the event. The cuts are complex conditions over properties of each object involving joins, aggregate functions, and numerical computations.

Currently physicists test their hypotheses using regular programming languages, e.g., C++. The analysis programs retrieve descriptions of events from files through specific data management libraries, e.g. ROOT. However, it takes lots of efforts for physicists to code their analyses as C++ programs. Furthermore, good knowledge of programming methodologies is necessary for writing extensible and understandable programs for complex analyses.

We present a database approach to test scientific hypotheses as declarative conjunctive queries. We found that while such queries can be handled using traditional query processing techniques, performance is very poor due to slow data load and index times, space overhead of indexed data, and poor cost estimates for large queries with many aggregates and user-defined numerical functions. On the other hand, such queries can be processed very quickly using hand-coded C++ programs, but each program typically takes a scientist weeks to create. To improve performance while retaining ease of query specification, we created SQISLE (*Scientific Queries over Independent Streamed Large Events*), a query processing system that takes advantage of the special data and query characteristics of our target domain (high-energy physics) while also meeting its unique challenges. SQISLE employs a one-pass *streaming approach* to process queries where data stays in their sources, e.g. ROOT files, and is streamed through the system. SQISLE provides facilities for complex queries over streams of objects with complex structure, as required for our kind of scientific applications. The system reads complex objects from sources, e.g. description of collision events from ROOT files, through a wrapper interface and processes the objects one-by-one as they are streamed. The objects are thus analyzed in one pass by reading data sequentially without populating a database. This streaming approach requires limited memory and is shown to be efficient compared to the traditional *loading approach* where data is loaded into a database and indexed before being queried.

Instead of relying on static query optimization, SQISLE collects query statistics at runtime, uses them to reoptimize the query, and dynamically switches optimization strategies. SQISLE's performance is further improved by query rewrites, temporary view materializations, and compile time evaluation of query fragments. We show that queries in SQISLE perform close to or better than equivalent C++ implementations hand-coded by scientists.

SQISLE extends Amos II [24]. The architecture of SQISLE is illustrated by Fig. 1, where the arrows show the data flow during query execution. A scientist specifies an analysis as a query over a stream of complex objects from *data sources* processed by SQISLE through a *wrapper interface*. The scientists write their analysis queries in terms of a high level application schema (*App. schema*) that defines the structure of the streamed complex objects. The source database (*Source DB*) contains meta-data about stream sources. It is accessed in queries to locate sources containing data for the analyses and their meta-data. The wrapper interface is defined in terms of an application data management library (*App. Library*), e.g., ROOT.

To obtain efficient execution plans for complex queries over streams of complex objects, SQISLE uses runtime query optimization strategies implemented by a *profile-controller* operator. It encapsulates in each query the fragment that tests complex conditions (hypotheses) over properties of the streamed complex objects. During runtime it controls collecting statistics for the fragment, reoptimizes the fragment based on collected statistics, and dynamically switches optimization strategies. The cost-based query optimization utilizes a cost model for aggregate functions over nested subqueries from [9]. To alleviate estimation errors in large queries the fragments are decomposed into conjunctions of subqueries over which runtime statistics are measured [9].

Data from controlled scientific experiments produced with the same experimental run conditions usually have similar statistical properties. Therefore, we assume that stream data statistics are similar. For example, in the ATLAS application properties of events have the same distribution for events generated in the same experiment. Once a fragment is reoptimized based on sufficient sampled data from the beginning of the queried stream, the query execution is immediately continued using the reoptimized query execution plan for the rest of the stream without profiling overhead.

The query processing is further improved by *query rewrites*, use of *materialized views*, and *compile time evaluation*. Query rewrite rules reduce the number of predicates in queries. Views are materialized once per read complex object and the materialized results are accessed while processing the same object and then discarded. Compile time evaluation [15,23] executes some predicates of a query at compile time before query execution and replaces predicates with execution results.

By evaluating the performance of SQISLE for ATLAS queries over ROOT files with different selectivities, we show that the SQISLE query processing techniques improve performance of queries substantially. The query performance is compared with the performance of a manually coded C++ program provided by the physicists doing the same analysis. The SQISLE implementation is shown to have performance close to or better than the hard-coded C++ implementation. Ideally a C++ program should perform better than declarative programs interpreted by a DBMS, but in practice writing an efficient C++ programs requires substantial effort, which has to be repeated for new queries or for data from new experiments. Using a declarative query language for testing scientific hypotheses is thus much more efficient for research productivity than expressing the tests as more complex C++ programs.

In summary, the contributions are:

- One-pass query processing algorithms are shown to provide efficient implementations of declarative queries testing our kind of scientific hypotheses.
- Runtime query optimization of streaming queries by the profile-controller operator allows measuring real query behavior and dynamically switching execution

strategies. The runtime strategies are shown superior to a static cost-based approach.

- With the proposed runtime optimization strategies, the streaming approach is shown to be more efficient than the loading approach for the targeted kind of queries with performance close to a hard-coded C++ program.

The rest of the paper is organized as follows. Section 2 presents how an ATLAS analysis is specified as a query. The SQISLE query processing techniques are presented in Section 3. The query performance is evaluated in Section 4. Section 5 presents related work. The paper is concluded in Section 6.

2 The ATLAS Application Queries

To evaluate the query processing techniques implemented in SQISLE an ATLAS application is defined as SQISLE queries. For example, the hypothesis from [5], which searches for events producing Higgs bosons, is specified as this query:

```
1:     select e
2:     from Event e, EventFile f
3:     where  name(experiment(f)) = "bkg2" and
4:            fileid(f) < 15 and
5:            e in events(filename(f)) and                 (1)
6:            hadrtopcut(e) and jetvetocut(e) and
7:            misseecuts(e) and zvetocut(e) and
8:            threeleptoncut(e) and leptoncuts(e);
```

The query selects objects of type *Event* satisfying six cuts constituting the hypothesis from [5] called the *Six Cuts Analysis*. On lines 3-5 the query specifies the sources to query by selecting the files produced by the experiment named *bkg2*. The source database is searched in lines 3-4, while the function *events* calls a *ROOT wrapper interface function* to read objects from the selected ROOT files. The rest of the query specifies the *Six Cuts Analysis*.

Wrapper interface functions read data from sources as complex *stream objects* represented by a data type named *Sobject*. The stream objects are defined as user-defined types, and are deallocated automatically and efficiently by an incremental garbage collector when they are not referenced any more.

Each cut is a view defined as a Boolean function that returns true if the cut is fulfilled. The views are defined as declarative queries over properties of each event object *e* involving joins, aggregate functions, and complex numerical computations defined in terms of a high-level application schema. The type hierarchy of the application schema for the ATLAS application is presented in Fig. 2.

A wrapper interface function reads events and instantiates them as stream objects of type *Event*, a subtype of type *Sobject*. Each complex event object describes measurements from one collision. Objects of type *Particle* represent various kinds of particles produced by the collision, which are derived from the event object by *transformation views*. The transformation views are defined as declarative functions mapping stream objects returned by a wrapper interface function into a set of derived stream objects representing objects in terms of the application schema.

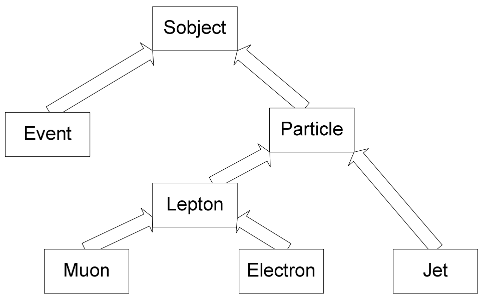


Fig. 2. Type hierarchy of an application schema for the ATLAS application

For example, the *Three Lepton Cut*, one of the simplest among the six cuts being part of the *Six Cuts Analysis*, requires that an event has exactly three *isolated leptons* with |*Eta*|<2.4 and *Pt*>7, where at least one lepton has *Pt*>20. The *Three Lepton Cut* is defined in SQISLE as this Boolean function:

```
create function threeLeptonCut (Event e) -> Boolean as
select  TRUE
where   count(isolatedLeptons(e)) = 3
        and some( select r
                  from Real r
                  where  r = Pt(isolatedLeptons(e))
                         and r > 20.0);
```

The function *isolatedLeptons* is defined as:

```
create function isolatedLeptons(Event e) ->
                                Bag of Lepton as
select l
from Lepton l
where  l in leptons(e)
       and pt(l) > 7.0
       and abs(eta(l)) < 2.4;
```

The functions *Pt* and *Eta* are defined in terms of numerical operators calculating $Pt = \sqrt{x^2 + y^2}$ and $Eta = 0.5 \cdot \ln\left(\frac{\sqrt{x^2+y^2+z^2}+z}{\sqrt{x^2+y^2+z^2}-z}\right)$, respectively, over momentum (x,y,z) of a particle. The function *leptons* is a transformation view defined as a query that returns all leptons detected by event *e*. It is defined as the union of the transformation views *electrons* and *muons*, which generate objects of types *Electron* and *Muon*, respectively. There is a detailed description of the used cuts and view definitions in [8].

Given that the cuts are defined as Boolean functions, a user query always contains the following kinds of query fragments:

- A *source access query fragment*, e.g. lines 3-5 in query (1), specifies sources to access and calls a *stream function* that emits a stream of complex objects.
- A *processing query fragment*, e.g. lines 6-8 in query (1), specifies the scientific analyses (tested hypotheses) as queries over views of complex objects, e.g. specifying cuts over events. The views are defined in terms of transformation views.

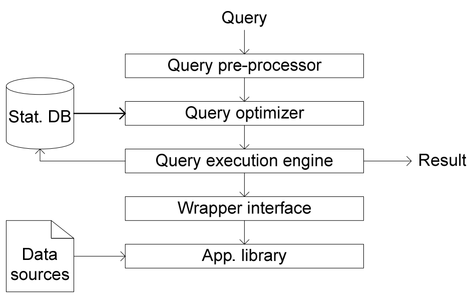


Fig. 3. Query processing steps in SQISLE

3 Query Processing in SQISLE

In [9] we implemented the above queries using the traditional loading approach where the data streams were first loaded into the DBMS and indexed before the queries were given. The loading and indexing is very time consuming. In our example, it takes about 15 seconds to load one ROOT file containing 25000 events, while the analysis alone of the 25000 events takes just 1.5 seconds, i.e. a total processing time of 16.5 seconds. Furthermore, the loading approach requires sufficient memory to store all complex objects indexed.

To avoid the high loading and indexing costs, in SQISLE, instead of preloading the data into a DBMS, the data stays in their sources, e.g. ROOT files, and are processed in one pass. The system reads the complex objects from the sources through a wrapper interface where each independent complex object is analyzed one-by-one as they are streamed.

Fig. 3 illustrates the query processing steps in SQISLE. The *query pre-processor* expands views and applies rewrite rules on the query. The cost-based *query optimizer* produces an execution plan, which is interpreted by the *execution engine*. The execution plan contains operators that call a *wrapper interface* implemented in terms of an application data management library (*App. Library,* e.g. ROOT) to access the *data sources*. To improve query processing, runtime query optimization collects data statistics for the query optimizer, which is stored in the *statistics database* (*Stat. DB*). The execution engine calls the optimizer to reoptimize the query at runtime using the collected statistics.

A general structure of an execution plan for a typical query like query (1) is presented in Fig. 4. With runtime query optimization the query pre-processor first splits the query into a source access query fragment and a processing query fragment. The figure illustrates the two corresponding subplans: the *source access plan* and the *processing plan*. The source access plan first calls *wrapper argument operators* that access the source database (*Source DB*) and the application schema meta-data (*App. schema*) to select sources and bind parameters a_1, a_2, … for a *wrapper interface operator*. The wrapper interface operator accesses each selected *data source*, e.g. a ROOT file, and generates a stream of complex objects, each bound to a *stream object variable, o*. The processing plan implements selections of *o* based on *analysis*

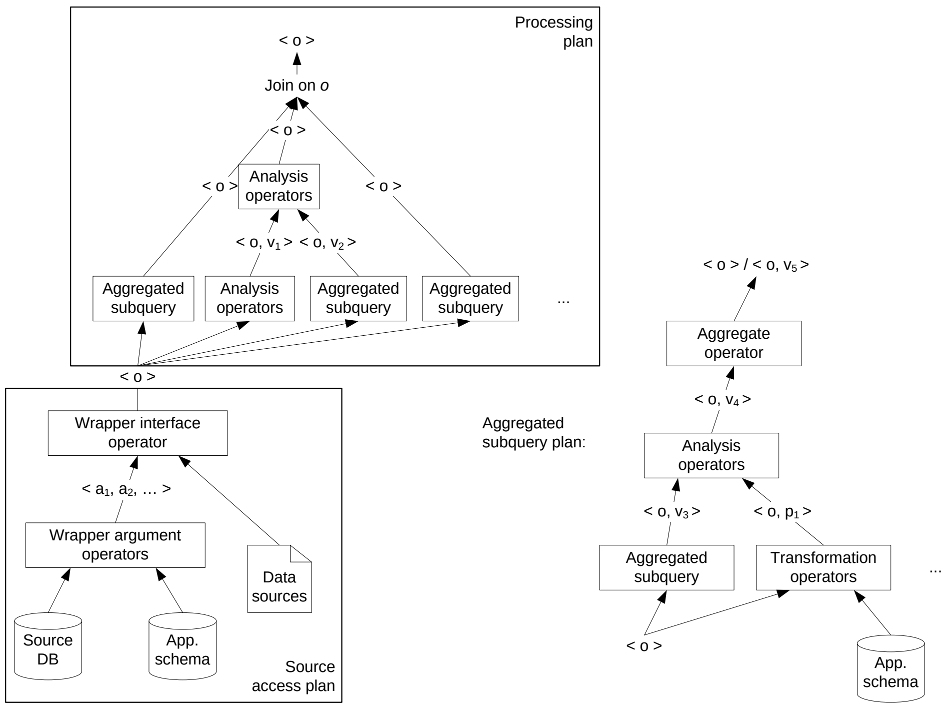


Fig. 4. General structure of a query plan

operators and *aggregated subqueries* over properties of *o*. The aggregated subqueries apply aggregate functions on nested subqueries. The nested subqueries first call transformation views to derive stream objects in terms of the application schema and then analyze properties of the derived objects. The source access plan and the processing plan are joined only on stream object variable *o*.

For query (1) the number of operators in the source access plan is 10. It produces a stream of objects representing events, which are analyzed by the processing plan. The processing plan contains 22 operators and 8 of them are calls to aggregated subqueries. The number of operators in the aggregated subquery plans is between 9 and 59, including transformation operators and analysis operators implementing the selections. Some aggregated subqueries contain calls to further aggregated subqueries.

There are many possible operator orders for a processing plan. Thus query optimization is difficult, and the query plans obtained with naïve static cost-based query processing strategies are slow. Therefore SQISLE implements runtime query optimization to collect data statistics from streams at runtime in order to adapt the query plan using collected statistics. Runtime query optimization is managed by a special operator, the *profile-controller*. During query execution it monitors whether sufficient statistics have been collected so far while processing the stream. If so, it dynamically reoptimizes the query and switches to non-profiled execution by disabling collecting and monitoring statistics.

The runtime query optimization is investigated with three strategies:

1. *Attribute statistics profiling* maintains detailed statistics on the sizes of vectors stored in each stream object attribute as the objects are read. Once the sample size is large enough the query is reoptimized using the collected statistics.
2. *Group statistics profiling* first decomposes the queries into fragments, called *groups*, which are joined only on the stream object variable, and then maintains runtime statistics of executing each group. When sufficient statistics is collected the query is reoptimized.
3. *Two-phase statistics profiling* combines the two strategies above by in a first phase collecting detailed statistics of attribute vector sizes of stream objects to optimize the group definitions, and in a second phase switching to group statistics profiling for ordering the groups.

All three strategies assume that data statistics over the stream is *stable* so that the statistics collected in the beginning of the stream is expected to be close to the statistics for the entire data stream. This is the case in scientific applications such as the ATLAS experiment, since all collision events in a stream are generated with the same experimental run conditions. The strategies perform statistics sampling at runtime until the statistics are stabilized. Cost-based query optimization utilizes a cost model for aggregate functions over nested subqueries [9]. This *aggregate cost model* estimates costs and selectivities for aggregate functions from costs and selectivities of their nested subqueries.

3.1 The Profile-Controller Operator for Runtime Query Optimization

The query pre-processor modifies the view expanded query to add the profile-controller operator and encapsulate the processing query fragment with the operator. The processing query fragment needs to be optimized carefully, since it is defined as a large condition over a complex stream object. The optimization at runtime of the encapsulated complex condition is controlled by the profile-controller operator.

The profile-controller performs the following operations for each stream object *o*:

1. It executes the processing plan parameterized by *o*.
2. It checks if profiling is enabled. If so it calls a subroutine, the *switch condition monitor*, which supervises collection of data statistics. The switch condition monitor returns true if sufficient statistics is collected. To enable different kinds of profiling the switch condition monitor can be different for different strategies and can also be dynamically changed during query execution.
3. If item two is satisfied it calls another subroutine, the *switch procedure,* which reoptimizes the processing query fragment and either switches to another runtime query optimization strategy or disables profiling. The switch procedure is also dynamically replaceable.
4. The result of the processing query fragment in item one is always emitted as result of the profile-controller operator.

3.2 Attribute statistics profiling

When attribute statistics profiling is enabled, detailed statistics on stream object attribute vector sizes are collected when each new stream object *o* is emitted by the wrapper interface operator. The means and variances of the attribute vector sizes of *o* are maintained in an internal table.

The switch condition monitor maintains statistics to check for every tenth read stream object *o* whether an estimated mean attribute vector size $\bar{x}$ is close enough to the actual mean attribute vector size with probability *1-α*. The following *attribute switch condition* is checked:

$$z_{\alpha/2} \cdot S_E \leq \delta \cdot \bar{x} \qquad (2)$$

The closeness is defined by *δ*. *α* and *δ* are provided as tuning parameters. The estimate of the mean $\bar{x}$ is calculated by $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, where x_i is the attribute vector size for the i^{th} stream object, and *n* is the number of stream objects read so far. S_E is an estimate of $\sigma / \sqrt{n}$ and calculated by $S_E = \sqrt{\frac{1}{n^2}\sum_{i=1}^{n} x_i^2 - \frac{\bar{x}^2}{n}}$.

If the attribute switch condition is satisfied for every attribute, the switch procedure is called. It reoptimizes the processing query fragment and disables profiling. After this, when the wrapper interface operator constructs a new stream object, it does not collect statistics any more. The profile-controller executes only the processing plan and does not call the switch condition monitor or the switch procedure.

When the query is started there no statistics collected and the query is initially optimized using *default statistics* where the attribute vector sizes of stream objects, e.g. the number of particles per event, is approximated by a constant (set to nine).

3.3 Group Statistics Profiling

With *group statistics profiling*, first a *stream fragmenting algorithm* is applied to the query. The algorithm decomposes the processing query fragment into smaller query fragments called *groups* [9]. The groups have only the stream variable *o* in common and thus the groups are equi-joined only on *o*. The complex object *o* is selected by the query if it satisfies the inner join of all groups.

After optimization, each group is implemented by a separate *group subplan*, which is encapsulated by a *group monitor* operator. The group monitor operator takes a group subplan and a stream object as arguments and returns the result of applying the subplan on the stream object. If profiling is enabled, it measures execution time and selectivity of the monitored subplan.

The switch condition monitor calls the query optimizer at runtime for every read stream object *o* to greedily order the executions of the monitored subplans based on available statistics on the groups. An internal table keeps track of the groups and their statistics. The switch condition is true if the order of the groups in the new processing plan is the same for a number of read stream objects in a row, called the *stable reoptimization interval* (*SI*), which is provided as a tuning parameter.

The contents of the groups and the initial join order of the groups are optimized using the default statistics before starting to execute the query.

The profile-controller operator encapsulates the entire processing plan containing all the joined groups. It invokes the dynamically optimized query processing plan at runtime. If some join fails, the entire processing plan fails. Thus, to answer the query the processing plan must execute only those first group subplans up to the first subplan that fails. No group subplans joined after the failed one need to be executed to answer the query. However, statistics still should be collected for all groups, even those that need not be executed. Thus, if profiling is enabled, the switch condition monitor executes also those remaining groups that were not executed to answer the query. In this way statistics is first collected for all groups by the switch condition monitor. The groups are then greedily reordered based on the measured estimates of the group costs and selectivities. To minimize overhead the processing plan is reoptimized once for every read complex stream object rather than for every query plan operator; there is no dynamic reordering per operator as with Eddies [3].

3.4 Two-Phase Statistics Profiling

As with group statistics profiling, with *two-phase statistics profiling* queries are first fragmented into groups before executing them. To collect runtime statistics for optimizing of group subqueries, attribute statistics profiling is enabled initially when query execution is started. When the attribute switch condition (2) is satisfied, the entire query is reoptimized, including the groups, and attribute statistics profiling is disabled. Then the switch condition monitor and switch procedure are changed to perform group statistics profiling to produce a further optimized group join order.

The main advantage with the two-phase statistics profiling is that it enables optimization of group subqueries based on collected attribute statistics. With group statistics profiling alone, where the attribute values are not monitored, the groups themselves must be optimized based on heuristic default statistics.

3.5 Query Rewrite Strategies

The performance is measured comparing runtime query optimization with a manually coded C++ program. It will be shown that optimized query plans of selective queries may perform better than a C++ implementation, while non-selective queries are still around 28 times slower.

In order to improve the performance of non-selective queries, their performance bottlenecks were analyzed. It was found that most of the time is spent on computing the transformation views many times for the same stream object. To remove this bottleneck, the use of rewrite rules to speed up the queries is investigated. One kind of rewrite is based on observing that the derivation of particle objects from event objects can be regarded as a two-dimensional matrix transposition. Different variants of operators for the transposition were implemented and evaluated. The chosen matrix transpose operator generates new particle stream objects as the result of the transposition and temporarily caches them as an attribute on the currently processed event object. This strategy is called *transformation view materialization*. It improves performance of non-selective queries about 1.5 – 2.5 times compared with only runtime query optimization.

Queries are further simplified in SQISLE by removing unnecessary vector constructions in queries and view definitions. Some vectors are first constructed out of variables and then only specific element values are accessed explicitly; the constructions of such vectors are removed and the original variables are instead accessed directly without vector construction and access overheads. These *vector rewrites* improve performance of non-selective queries with factor 1.5 – 2.

In addition *computational view materialization* improve query performance by temporarily saving on each processed complex object the results of numerical calculations computing properties of derived stream objects used in analysis queries, e.g. in cut definitions. This pays off when a query does the same complex numerical calculations several times per complex object. Materialization of computational views improves non-selective queries with at least another 32%.

Finally, the performance of queries is further improved by compile time evaluation [15,23], which is a general technique to evaluate predicates at query compilation time and replace them with computed values. Compile time evaluation is used to remove accesses to application schema meta-data, which simplifies the queries. Compile time evaluation improves performance of non-selective queries an additional 20%

All together the above query rewrite techniques improve performance of non-selective queries around 5 times. The execution is still about 4 times slower than C++. However, the execution plan is currently interpreted in SQISLE and further performance improvements can be made by making an execution plan compiler. This is expected to make the plan as fast as C++ also for non-selective queries.

4 Performance Evaluation

Performance experiments are made for scientific analyses expressed as queries in SQISLE for the ATLAS application. The experiments are run on a computer having 2.8 GHz Intel P4 CPU with 2GB RAM and Linux OS.

The performance is evaluated with different query processing strategies for two different kinds of queries implementing *Six Cuts Analysis* [5] and *Four Cuts Analysis* [11]. The performance of the C++ implementation is measured only for *Six Cuts Analysis*, since this implementation was the only one provided by the physicists.

Data from two different ATLAS experiments stored in ROOT files were used. The experiment *bkg2* simulates background events, which unlikely produce the Higgs bosons, so the analysis queries are very selective (*Six Cuts Analysis* has selectivity 0.018% and *Four Cuts Analysis* has selectivity 0.19%). The experiment *signal* simulates events that are likely to produce Higgs bosons, and both kinds of queries over these data are non-selective (*Six Cuts Analysis* selects 16% events and *Four Cuts Analysis* selects 58% events).

Event descriptions from the *bkg2* experiment are stored in 41 ROOT files, where each file contains 25000 event objects, i.e. a 1025000 event objects in total. Event descriptions from the *signal* experiment are stored in a single file with 8623 event objects. The sizes of the event streams are scaled by reading subsets of the files.

Two different kinds of queries are measured for the two different experiments. *Six Cuts Analysis* uses the views *bkgsixcuts* and *signalsixcuts* for experiment *bkg2* and experiment *signal*, respectively. *Four Cuts Analysis* uses the views *bkgfourcuts* and *signalfourcuts,* which are less complex than *bkgsixcuts* and *signalsixcuts*. A view parameter is used to specify the number of events to read and analyze, i.e. the stream size. The details can be found in [8].

Two kinds of measurements are made: the *total query processing time* and the *final plan execution time*. The total query processing time is the total time for optimization, profiling, and execution of a query. The final plan execution time is the time to execute the optimized plan.

4.1 Evaluated Strategies

The following strategies are evaluated:

- *Naïve query processing (NaiveQP)*. As a reference point, this strategy demonstrates performance of naive query processing without cost-based optimization. The cuts are executed in the same order as they are specified in the queries.
- *Static query processing with the aggregate cost model (StatQP)*. This reference strategy demonstrates the impact of regular static cost-based optimization based on the aggregate cost model. The aggregate cost model is enabled, but no runtime query optimization strategies. No data statistics is available when the query is optimized and default statistics are used. Since queries are very large, they are optimized using randomized optimization [14,27,22], which is able to find a good plan in terms of estimated costs. The strategy is compared with *NaiveQP* to demonstrate impact of the static cost-based query optimization using default statistics.
- *Attribute statistics profiling (AttrSP)*. The query is initially optimized with the aggregate cost model and default statistics. During execution of the query the statistics on sizes of the attribute vectors is collected and query reoptimization is performed using collected statistics. The initial optimization uses a fast greedy optimization method [16,18] and default statistics. The query reoptimization uses slow randomized optimization, which produces much better plan in terms of estimated cost than the greedy optimization.
- *Group statistics profiling (GroupSP)*. After query fragmentation into groups, the created groups and their order are initially re-optimized per event by greedy optimization using default statistics until a stable plan is obtained. Fast greedy optimization is also used to reoptimize the group order since dynamic programming [26] produced the same execution plans.
- *Two-phase statistics profiling (2PhaseSP)*. Greedy optimization is used first to produce optimized group subplans after performing attribute statistics profiling. In a second phase *GroupSP* is applied to optimize the order of the group subplans.
- *Full query processing (FullQP)*. This strategy implements query rewrites combined with the best of the above optimization strategies.

As reference *FullQP* is compared with the following C++ implementations of the same analysis:

– *Unoptimized C++ implementation (NaiveCPP)*. This strategy demonstrates the performance of a manual C++ implementation of *Six Cuts Analysis* executed in the same order as in query *bkgsixcuts*. This strategy is compared with *FullQP*.
– *Optimized C++ implementation (OptCPP)*. This strategy demonstrates the performance of *Six Cuts Analysis* implemented in a C++ program where the order of the cuts had been optimized by a researcher manually. This strategy is compared with *FullQP*.

All evaluated strategies are summarized in Table 1.

Table 1. Evaluated query processing techniques

Strategy	Aggregate cost model	Attribute statistics profiling	Group statistics profiling	Rewrites
NaiveQP	–	–	–	–
StatQP	+	–	–	–
AttrSP	+	+	–	–
GroupSP	+	–	+	–
2PhaseSP	+	+	+	–
FullQP	+	+	+	+
NaiveCPP	C++ implementation with suboptimal order of cuts			
OptCPP	C++ implementation with the cuts ordered manually by a scientist			

4.2 Experimental results

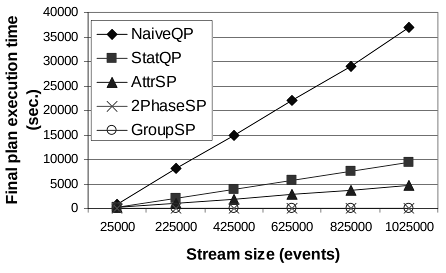
The performance of different optimization approaches without query rewrites is investigated first. Then the additional impact of the query rewrites is investigated. Finally, the best strategy is compared with hardcoded C++ implementations. The sizes of input streams in the evaluations are scaled over six points per experiment as shown in Figures 5 and 6.

Fig. 5(a) presents performance of the query plans measured by the different optimization approaches for the selective complex query *bkgsixcuts* (0.018% events selected).

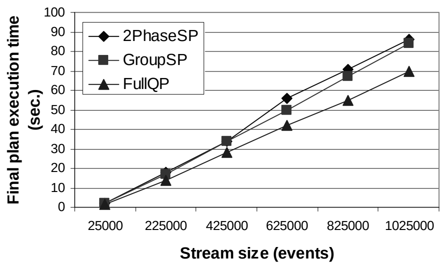
The query plan of the unoptimized processing strategy (*NaiveQP*) performs substantially worse than the other strategies. Static query optimization with the aggregate cost model (*StatQP*) gives a query plan that performs four times better than the query plan from *NaiveQP*. This demonstrates the importance of the aggregate cost model to differentiate between different aggregated subqueries.

The query plan obtained with attribute statistics profiling (*AttrSP*) performs twice better than the statically optimized plan (*StatQP*). This shows that runtime query optimization is better than static optimization.

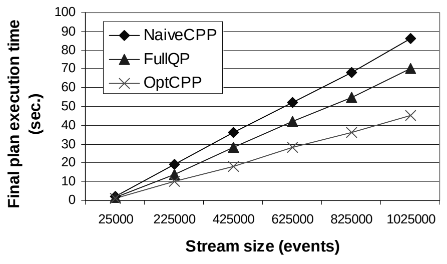
The query plans from the group statistics profiling (*GroupSP*) and two-phase statistics profiling strategies (*2PhaseSP*) perform best and substantially better than the strategies without grouping. They outperform naïve query processing (*NaiveQP*) with a factor 450 and attribute statistics profiling without grouping (*AttrSP*) with a factor



(a)



(b)



(c)

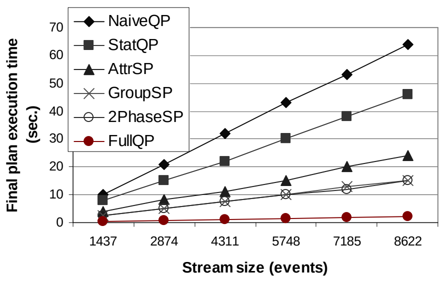
Fig. 5. Performance of different strategies for selective complex query *bkgsixcuts*: (a) and (b) show performance of different query strategies in SQISLE, while (c) compares performance of the best strategy (*FullQP*) with the C++ implementations.

50. This demonstrates that the grouped strategies *GroupSP* and *2PhaseSP* alleviate the problem of errors in the cost estimates [12] by measuring real execution time and selectivity for each group. The difference between *GroupSP* and *2PhaseSP* is insignificant (Fig. 5(b)). The total query processing times for the strategies (Table 2) demonstrate that *2PhaseSP* performs better than *GroupSP*. Thus *2PhaseSP* is chosen to optimize rewritten queries in *FullQP*. Fig. 5(b) demonstrates that the strategy with rewrites (*FullQP*) performs 17% better than the optimization strategies without rewrites (*GroupSP* and *2PhaseSP*) for the selective query *bkgsixcuts*. The query performance for the other selective query *bkgfourcuts* is similar to query *bkgsixcuts*, but with lower overheads, since the query is simpler.

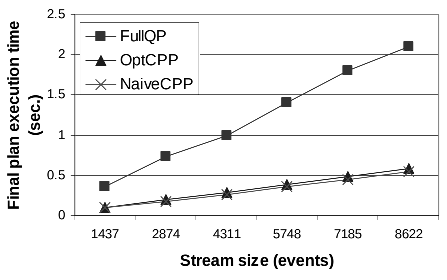
Fig. 5(c) demonstrates that for the selective query the best query processing strategy (*FullQP*) performs 20% better than unoptimized C++ (*NaiveCPP*). However, the C++ implementation where the order of cuts is optimized manually by the physicist, *OptCPP*, performs 34% better than the query plan from *FullQP*. Further performance improvements in SQISLE can be made by making an execution plan compiler, which is likely to make the plan faster than C++ for selective queries.

In conclusion, query optimization, in particular runtime query optimization, improves performance substantially for selective queries. For selective queries the impact of query rewrites is relatively insignificant compared to query optimization.

The loading approach in [9] took 15 seconds to load 25000 events. By contrast, the total processing in SQISLE with *FullQP* of the same number of events is 1.6s, which clearly shows the advantage with the streaming approach for our kind of applications.



(a)



(b)

Fig. 6. Performance of different strategies for non-selective query *signalsixcuts*: (a) SQILSE strategies, (b) performance of the best strategy (*FullQP*) and the C++ implementations.

In Table 2 the optimization strategies are compared by their *optimization overheads* obtained by subtracting the final plan execution time from the total query processing time. These overheads are independent of the stream size so the impact is negligible in practice for large streams. Table 2 contains performance measurements only for 25000 events, i.e. one file.

The optimization overhead of the ungrouped strategy *StatQP* is only the time to perform randomized optimization (29 seconds). The overhead of *AttrSP* (26 seconds) is dominated by the randomized optimization (80%). The remaining time is spent on collecting and monitoring statistics. The overheads of the grouped strategies (6.0 seconds for *GroupSP* and 4.5 seconds for *2PhaseSP*) are dominated (75%) by performing group profiling. To obtain the final execution plan *GroupSP* profiles only the first 40 events of the stream. The overhead of profiling all groups for a single event (0.15s) is substantial. The reason is that statistics are collected for all groups, including the very complex and expensive ones to get a good cost model. Therefore, it is necessary to disable profiling once stream statistics are stabilized. Notice that overheads in both the ungrouped strategies are around four times higher than overheads of the grouped strategies, because the grouped strategies use the greedy optimization, which performs well, while for ungrouped strategies the greedy optimization did not produce good plans. Therefore the slow randomized optimization is used for ungrouped strategies.

Fig. 6(a) presents performance of the query processing strategies for the non-selective query *signalsixcuts* (16% events selected). The impact of the different query optimization strategies is less significant here. The best strategies (*GroupSP* and *2PhaseSP*) are just four times faster than the slowest (*NaiveQP*). Using the aggregate

Table 2. Overhead times in seconds for query *bkgsixcuts* over events from one file

Strategy	Total query processing time	Final plan execution time	Optimization overhead
StatQP	253	224	29
AttrSP	136	110	26
GroupSP	9.4	1.9	7.5
2PhaseSP	7.9	1.9	6.0
FullQP	6.1	1.6	4.5

cost model (*StatQP*) gives a query plan that performs 28% better than *NaiveQP*. Using the attribute statistics profiling (*AttrSP*) gives a query plan that performs twice better than the query plan obtained without collecting statistics (*StatQP*). *GroupSP* and *2PhaseSP* are 35% faster than the *AttrSP*. The difference between *GroupSP* and *2PhaseSP* is again insignificant. We notice that query optimization has less impact on non-selective queries. In this case, the rewrites (*FullQP*) improve performance of the query by factor five compared to *2PhaseSP*. The other non-selective query *signalfourcuts* (58% events selected) performs similar to *signalsixcuts*.

Fig. 6(b) compares performance of the best query processing strategy (*FullQP*) with performance of the C++ implementations (*NaiveCPP* and *OptCPP*) for the non-selective query *signalsixcuts*. *FullQP* performs four times worse than both C++ implementations. The reason is that since the query is non-selective most operators are executed. Here, the cost of interpreting an operator in SQISLE is higher than the cost of executing machine instructions in C++, and we are comparing interpreted SQISLE with compiled C++. Implementing a compiler for query plans will reduce the interpretation overhead significantly.

In conclusion, query optimization, in particular runtime query optimization, improves performance significantly for all kinds of queries. For selective queries the improvements are dramatic. The impact of query rewrites is insignificant compared to query optimization of selective queries. For non-selective queries the combination of query optimization and query rewrite techniques significantly improves performance.

The evaluation demonstrates that query optimization techniques implemented in SQISLE can achieve performance for large and complex scientific queries close to a manually optimized C++ program.

5 Related Work

A visual query language for specifying HEP analyses is provided by the system PHEASANT [25]. HEP analyses are there defined in queries, which are translated into a general purpose programming language without any query optimization or simplification. By contrast, our system rewrites and optimizes queries, which is shown to give significant improvement in performance, approaching or surpassing that of hard-coded C++ programs.

Most developed DSMSs (e.g., Aurora [2], STREAM [1], TelegraphCQ [17], and XStream [10]) focus on infinite streams of rather simple objects and efficient processing of time-series operations over the streams, including stream aggregates and joins. The DSMSs are data driven and the continuous queries are rather simple. In contrast, in SQISLE the elements of the streams are complex objects (each object can be seen as a small database) and complex queries are applied on each streamed object independently from other objects. Thus the queries in SQISLE do not contain time-series operations and no join between streams is performed. Furthermore, SQISLE is demand driven, since it controls the stream flow.

In DBMSs, and in particular in DSMSs, precise statistics on data are not always available. Therefore, adaptive query processing (AQP) techniques are developed to improve query processing at query execution time by utilizing runtime feedback.

AQP systems (e.g. [1,2,3,6,13,17,19,20]) continuously adapt the execution plan of a query to reflect significant changes in data statistics. By contrast SQISLE profiles a stream until statistical properties of the streamed objects are stabilized, and then reoptimizes the query using the stable statistics. This works well for our scientific applications where large numbers of complex objects having similar statistical properties (run conditions) are processed. After the statistics are stabilized, the rest of a stream is efficiently processed without profiling overhead.

Usually DSMSs (e.g. Aurora [2], STREAM [1], and TelegraphCQ [17]) schedule operators continuously per tuple and change the execution plan if significant flow changes are detected. Such monitoring for each simple data element adds significant overhead for large queries over complex objects. For Eddies [3] this overhead is even more significant, since optimization is performed whenever a tuple is scheduled for a next operator. To avoid the high cost of monitoring individual data elements, in SQISLE the profile-controller operator monitors the execution of an entire plan once per complex input stream object until sufficient statistics are collected about the objects, after which the plan is dynamically replaced with a final plan.

Some systems [6,13,19] generate several query execution plans and adaptively switch between them during query execution. Generating many execution plans during initial optimization is not feasible for large and complex queries. By contrast SQISLE generates only one initial query execution plan which is reoptimized at runtime using collected statistics to obtain a more efficient final execution plan.

The demand driven DBMS in [20] reoptimizes the entire query at runtime and then restarts the query based on already computed materialized intermediate results. SQISLE is also demand driven but need not restart the entire query since it reoptimizes only the processing query fragment that is applied on each subsequent streamed complex object produced by the static source access plan.

Query rewrites before cost-based query optimization has been demonstrated to improve performance for different kinds of applications in, e.g., engineering [28], image processing [21], and business processing [30]. SQISLE implements several rewrite rules and shows that they are particularly important for non-selective queries.

An example of implementing a complex scientific application in a DBMS with the loading approach is the Sloan Digital Sky Survey (SDSS) project [29]. In the project huge amounts of astronomical data from the SDSS telescope are loaded into a cluster of SQL Server databases and indexed. In our application efficient query specific indexes are required for calculating query dependent aggregated properties, e.g. based on number of isolated leptons, and static query independent indexing is not sufficient. Furthermore, the performance of first loading the data into a database and then processing them as queries is shown to be around ten times slower than processing the same data in one pass by SQISLE.

6 Summary and Future Work

The implementation was presented of a data stream management system SQISLE targeted to scientific applications where data are independent objects with complex structures selected by complex queries. SQISLE reads complex objects from files through a streamed wrapper interface and processes them in one pass efficiently by utilizing novel query processing techniques. Runtime query optimization methods collect stream statistics and reoptimize queries during execution. During query execution a *profile-controller* operator monitors collected statistics, reoptimizes the processing query fragment, and switches to another strategy, e.g. into non-profiled execution. Since the complex objects contain measurements produced in controlled experiments, we assume that statistical properties of complex objects, such as average number of different kinds of particles per event, produced in the same experiment are the same. Therefore profiling is performed only at the beginning of the one-pass data processing and then disabled to reduce profiling overhead.

To verify the approach, a scientific application from the ATLAS experiment [5,11] was implemented in SQISLE. The implementation demonstrated that performance of application analysis queries in SQISLE is close to a hard-coded and manually optimized C++ implementation of the same analysis, which requires a significant effort to develop.

In summary, the following techniques in SQISLE provide efficient processing of queries over streams of complex objects:

- The profile-controller operator enables more efficient execution plans for streamed queries than static cost-based query optimization, by choosing different query optimization strategies at runtime and then disabling the profiling.
- The query optimization techniques are shown to significantly improve performance of all kinds of queries.
- The query rewrite techniques are shown to improve performance significantly for non-selective queries, while being less effective for selective queries.

SQISLE currently interprets the generated query execution plans. By compiling the executions plans into C or machine code, the performance will be significantly better than the current implementation. Further improvements can be achieved by eliminating copying data from structures used in the ROOT files and structures used in SQISLE. It can be done either by storing collision event data in the ROOT files using data format used by SQISLE or by rewriting data management in SQISLE to operate on data having the same structure as in the ROOT files. Since the performance of SQISLE is already close to C++, these changes are likely to make SQISLE perform at least as well as a C++ program manually written by a physicist.

References

1. Arasu, A., et al.: STREAM: The Stanford Stream Data Manager. In: IEEE Data Eng. Bull., vol. 26, pp. 19-26 (2003)
2. Abadi, D.J., et al.: Aurora: a new model and architecture for data stream management. In: VLDB J., vol. 12, pp. 120-139 (2003)
3. Avnur, R., Hellerstein, J.M.: Eddies: Continuously Adaptive Query Processing. In: SIGMOD Conference, pp. 261-272 (2000)
4. The ATLAS experiment, http://atlasexperiment.org/
5. Bisset, M., Moortgat, F., Moretti, S.: Trilepton+top signal from chargino-neutralino decays of MSSM charged Higgs bosons at the LHC. In: Eur.Phys.J. C, vol. 30, pp. 419-434 (2003)

6. Babu, S., et al.: Adaptive Ordering of Pipelined Stream Filters. In: SIGMOD, pp. 407-418 (2004)
7. Brun, R., Rademakers, F.: ROOT - An Object Oriented Data Analysis Framework. In: AIHENP'96 Workshop, Nucl. Inst. & Meth. in Phys. Res. A 389, pp. 81-86 (1997).
8. Fomkin, R.: Optimization and Execution of Complex Scientific Queries. Uppsala Dissertations from the Faculty of Science and Technology 80 (2009)
9. Fomkin, R., Risch, T.: Cost-based Optimization of Complex Scientific Queries. In: SSDBM, p. 1 (2007)
10. Girod, L., et al.. XStream: a Signal-Oriented Data Stream Management System. In ICDE, pp. 1180-1189 (2008)
11. Hansen, C., Gollub, N., Assamagan, K., Ekelöf, T.: Discovery potential for a charged Higgs boson decaying in the chargino-neutralino channel of the ATLAS detector at the LHC. In: Eur.Phys.J. C, vol. 44, pp. 1-9 (2005)
12. Ioannidis, Y.E., Christodoulakis, S.: On the Propagation of Errors in the Size of Join Results. In: SIGMOD, pp. 268-277 (1991)
13. Ives, Z.G., Halevy, A.Y., Weld, D.S.: Adapting to Source Properties in Processing Data Integration Queries. In: SIGMOD, pp. 395-406 (2004)
14. Ioannidis, Y.E., Kang, Y.C.: Randomized Algorithms for Optimizing Large Join Queries. In: SIGMOD, pp. 312-321 (1990)
15. Jones, N.D.: An Introduction to Partial Evaluation. In: ACM Comput. Surv., vol. 28, pp. 480-503 (1996)
16. Krishnamurthy, R., Boral, H., Zaniolo, C.: Optimization of Nonrecursive Queries. In: VLDB, pp. 128-137 (1986)
17. Krishnamurthy, S., et al.: TelegraphCQ: An Architectural Status Report. In: IEEE Data Eng. Bull., vol. 26, pp. 11-18 (2003)
18. Litwin, W., Risch, T.: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates. In: IEEE Trans. Knowl. Data Eng., vol. 4, 517-528 (1992)
19. Li, Q., et al.: Adaptively Reordering Joins during Query Execution. In: ICDE, 26-35 (2007)
20. Markl, V., et al.: Robust Query Processing through Progressive Optimization. In: SIGMOD, pp. 659-670 (2004)
21. Marathe, A.P., Salem, K.: Query processing techniques for arrays. In: VLDB J. , vol. 11, pp. 68-91 (2002)
22. Näs, J.: Randomized optimization of object oriented queries in a main memory database management system. Master's Thesis. http://user.it.uu.se/~udbl/Theses/JoakimNasMSc.pdf
23. Petrini, J.: Querying RDF Schema Views of Relational Databases. Uppsala Dissertations from the Faculty of Science and Technology 75 (2008)
24. Risch, T., Josifovski, V., Katchaounov, T.: Functional data integration in a distributed mediator system. In: Gray, P.M.D., Kerschberg, L., King, P.J.H., Poulovassilis, A. (eds.) The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data. SpringerVerlag (2003)
25. Sousa, V., Amaral, V., Barroca, B.: Towards a full implementation of a robust solution of a domain specific visual query language for HEP physics analysis. In: J. Phys.: Conf. Ser. 119 (2007)
26. Selinger, P.G., et al.: Access Path Selection in a Relational Database Management System. In: SIGMOD, pp. 23-34 (1979)
27. Swami, A.N., Gupta, A.: Optimization of Large Join Queries. In SIGMOD, pp. 8-17 (1988)
28. Sellis, T.K., Shapiro, L.D.: Query Optimization for Nontraditional Database Applications. In IEEE Trans. Software Eng., vol. 17, pp. 77-86 (1991)
29. Szalay, A.S.: The Sloan Digital Sky Survey and beyond. In: SIGMOD Rec., vol. 37, pp. 61-66 (2008)
30. Vrhovnik, M., et al.: An Approach to Optimize Data Processing in Business Processes. In: VLDB, pp. 615-626 (2007)