ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from the Faculty of Science and Technology
80

Ruslan Fomkin

Optimization and Execution of Complex Scientific Queries

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in Häggsalen, Ångströmslaboratoriet, Lägerhyddsvägen 1, Polacksbacken, Uppsala, Monday, February 2, 2009 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract
Fomkin, R. 2009. Optimization and Execution of Complex Scientific Queries. Acta Universitatis Upsaliensis. *Uppsala Dissertations from the Faculty of Science and Technology* 80. 157 pp. Uppsala. ISBN 978-91-554-7382-2.

Large volumes of data produced and shared within scientific communities are analyzed by many researchers to investigate different scientific theories. Currently the analyses are implemented in traditional programming languages such as C++. This is inefficient for research productivity, since it is difficult to write, understand, and modify such programs. Furthermore, programs should scale over large data volumes and analysis complexity, which further complicates code development.

This Thesis investigates the use of database technologies to implement scientific applications, in which data are complex objects describing measurements of independent events and the analyses are selections of events by applying conjunctions of complex numerical filters on each object separately. An example of such an application is analyses for the presence of Higgs bosons in collision events produced by the ATLAS experiment. For efficient implementation of such an ATLAS application, a new data stream management system SQISLE is developed. In SQISLE queries are specified over complex objects which are efficiently streamed from sources through the query engine. This streaming approach is compared with the conventional approach to load events into a database before querying. Since the queries implementing scientific analyses are large and complex, novel techniques are developed for efficient query processing. To obtain efficient plans for such queries SQISLE implements runtime query optimization strategies, which during query execution collect runtime statistics for a query, reoptimize the query using the collected statistics, and dynamically switch optimization strategies. The cost-based optimization utilizes a novel cost model for aggregate functions over nested subqueries. To alleviate estimation errors in large queries the fragments are decomposed into conjunctions of subqueries over which runtime statistics are measured. Performance is further improved by query transformation, view materialization, and partial evaluation. ATLAS queries in SQISLE using these query processing techniques perform close to or better than hard-coded C++ implementations of the same analyses.

Scientific data are often stored in Grids, which manage both storage and computational resources. This Thesis includes a framework POQSEC that utilizes Grid resources to scale scientific queries over large data volumes by parallelizing the queries and shipping the data management system itself, e.g. SQISLE, to Grid computational nodes for the parallel query execution.

Keywords: scientific databases, query processing, data streams, cost-based query optimization, query rewritings, databases and Grids

Ruslan Fomkin, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

ISSN 1104-2516
ISBN 978-91-554-7382-2

urn:nbn:se:uu:diva-9514 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-9514)

Printed in Sweden by Universitetstryckeriet, Uppsala 2009
Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala
www.uu.se, acta@ub.uu.se

Моей семье

Contents

Abbreviations

ALEH	query system for Analysis of LHC Events for containing charged Higgs bosons
ARC	Advanced Resource Connector (earlier called the NorduGrid middleware, NG)
DB	Data Base
DBA	Data Base Administrator
DBMS	Data Base Management System
DSMS	Data Stream Management System
ER	Entity-Relationship
EER	Extended ER
HEP	High Energy Physics
LHC	Large Hadron Collider
OO	Object-Oriented
POQSEC	Parallel Object Query System for Expensive Computations
RDBMS	Relational DBMS
SALEH	Streamed ALEH
SPJ	Select-Project-Join
SQISLE	DSMS for processing Scientific Queries over Independent Streamed Large Events

1. Introduction

The scientific community produces lots of data, on which scientists perform complex analyses to test hypotheses and theories. The amount of data is usually huge so it is important to scale the analyses for large data volumes. Scientists also need to understand the analyses and be able to modify them in a simple way. Therefore the computer definition of the analyses should be simple and easy to understand by a scientist. Furthermore, the complex analyses contain many numerical operations that should be executed efficiently.

For example, in High Energy Physics (HEP) a lot of data is generated by simulation software from the Large Hadron Collider (LHC) experiment ATLAS [7]. The data describes effects from collisions of particles. A collision generates measurements of new particles, which are summarized in a collision description called an *event*. Every collision is performed independently from others, thus events are also independent. Events are stored in files, which are generated and stored using Grid infrastructures [31] that provide uniform access to pools of stored files and computational resources [35]. Physicists test their theories on these data by selecting interesting events. An event is interesting if it satisfies some conditions, which are called *cuts*. Cuts are complex conditions over properties of an independent event involving joins, aggregate functions, and complex numerical computations. An example of a scientifically interesting event is a collision event which is likely to produce Higgs bosons [15][47].

Currently physicists implement their theories using regular programming languages, e.g., C++, and write scripts for a Grid infrastructure to access event files and to execute analyses over the files. The analysis programs retrieve events from files through specific data management libraries, for example the C++ framework *ROOT* [18]. However, it takes lots of efforts for physicists to express their analyses as C++ programs. Furthermore, good knowledge of programming methodologies is necessary for writing extensible and understandable programs for complex analyses. Because of this it is often difficult to debug, understand, and modify the analysis programs. Moreover, when the amount of data grows, scientists have to manually modify programs and scripts to improve performance by code optimization and parallelization.

On the other hand database management systems (DBMSs) [44] provide high level query language interfaces to specify data analyses that scale over

large amounts of data. Query languages like SQL have been shown to enable much higher productivity than manual programming of regular programs that traverse databases [24][89]. High level query languages furthermore give flexibility for a database query optimizer to automatically generate efficient and scalable query plans [89]. Parallelization of query execution plans to run on many computing nodes is transparent for the user [76]. Furthermore, modern DBMSs can be extended with accesses to new kinds of data sources, user-defined query functions, and user-defined data types, which make it possible to use them for new applications such as scientific ones.

In this Thesis it is investigated how database query processing technologies can improve scientific analyses and novel database query processing techniques are proposed for this. It aims at answering the following research questions:

1. Can a DBMS and database queries be used to implement scientific applications and scientific analyses? In particular, how should a DBMS be extended for implementing a complex scientific application?
2. Can query processing improve performance and scalability of complex scientific analysis queries? What query rewriting and optimization techniques are needed for these?
3. How can storage and computational resources available through a Grid infrastructure be utilized for scaling scientific analyses queries over large amounts of data?

The Thesis focuses on those scientific applications where data are measurements of independent events and the analyses are selections of those events satisfying conjunctions of complex numerical filters on each event separately. Furthermore, each event has a lot of associated data and therefore can be seen as a small database, i.e. a *complex object*. The ATLAS experiment is an example of such an application, since each collision is performed independently from other collisions and each analysis is specified as a conjunction of complex conditions on each collision event. The answers to the research questions are illustrated on examples of the ATLAS application from [15] and [47].

To show the feasibility of the proposed database approach, a first prototype implementation of the ATLAS application from [15][47] was made as extensions of a main memory DBMS *Amos II* [79]. The prototype is called *ALEH* (query system for Analysis of LHC Events for containing charged Higgs bosons). Events are there modeled as objects and functions in a high-level functional data model [79], and a functional schema of event data is designed. The analyses are expressed as conjunctive queries in a functional query language. This way of implementing the application is simple and natural since it is close to the textual application description as expressed by the scientists in [15][47]. Therefore, it is more natural and

much easier for the physicists to implement the analysis in queries than in traditional way in C++ programs.

The amount of data in scientific applications is huge and the data is often stored in distributed Grid files. Therefore, a framework was implemented that connects ALEH with a Grid infrastructure called the *Advance Resource Connector*, ARC [32]. The framework is called *POQSEC* (*Parallel Object Query System for Expensive Computations*) and it utilizes resources of *Swegrid* [90]. POQSEC provides a query interface to specify the analyses, parallelizes queries into subqueries, generates job scripts for subqueries, submits jobs to ARC for execution, monitors job executions, downloads job results, and delivers results to users. POQSEC demonstrates an architecture, where not only analysis subqueries and data are shipped to computational nodes for execution but also the DBMS itself.

The implemented analysis queries and views are large and complex compared to traditional database queries. Thus naïve processing of the queries on each node takes a lot of time. It was therefore investigated how local execution on one computation node can be improved by query rewriting and optimization techniques. Two different query processing architectures were studied with regard to query performance:

- First the conventional *loading approach* was studied, where first data is loaded into a database and then queries are executed over the loaded data. The ALEH prototype uses the loading approach.
- Then the *streaming approach* was studied, where data is not loaded, but the scientific queries are executed directly over streams of data read from the files or other sources. The streaming approach is natural for those applications targeted by the Thesis, since every event is analyzed separately from other events.

The loading approach is used in ALEH to analyze query optimization of complex scientific queries. The ALEH implementation uses a functional schema to represent events and analysis queries are implemented over the functional schema. A cost-based query optimizer relies on cost models of operators used in queries. To improve the optimization of the targeted kind of scientific queries, a novel cost model is developed for aggregate functions over nested subqueries. It is shown that this substantially improves ALEH performance. However, the query optimizer still produces suboptimal plans because of estimate errors. Furthermore, the time to do optimization is very long because of the large query size.

The optimization is improved by a *profiled grouping* strategy where an analysis query is first automatically fragmented into subqueries based on application knowledge that all data are referenced by events and each event is analyzed independently. Each fragment is then independently profiled on a sample of events to measure real execution cost and fanout. An optimized fragmented query with the measured cost model is shown to execute faster than an ungrouped query optimized with the estimated cost model alone.

Furthermore, the total optimization time, including fragmentation and profiling, is substantially improved.

In ALEH the database of events is stored in main memory. The strategy of loading events into the main memory DBMS has two main disadvantages:

- The time to load the data can be substantial.
- There is normally not sufficient main memory to fit the entire data set so an even slower disk representation would be required to load all events to analyze.

To alleviate these bottlenecks a streaming approach to query processing was implemented in a new Data Stream Management System (DSMS) called *SQISLE* (*Scientific Queries over Independent Streamed Large Events*). Unlike a conventional DBMS, into which data has to be loaded before it can be queried, a DSMS [9] like SQISLE manages and analyzes streamed data not stored permanently in a DBMS, and the data streams are considered infinite and cannot be re-read in general. In SQISLE the queries are selecting complex objects streamed through the system. The streaming approach is natural for our kind of scientific applications where each event is analyzed independently from other events. Thus it is sufficient to access only one currently analyzed complex object at the time from a stream and temporarily materialize it in main memory only during the execution of an analysis query over it.

SQISLE is implemented as an extension of the research DBMS Amos II by extending its functional data model with a new data type *Sobject* to represent complex objects participating in streams. Such *stream objects* are allocated efficiently, are defined as user-defined types, and are deallocated automatically and efficiently by an incremental garbage collector when they are not referenced any more. The events streamed from sources are represented as stream objects and the transformation between the event representation in the sources and the event representation in a high-level functional application schema is defined as *transformation views* by queries. Therefore a user query always contains the following kinds of query fragments:

- A *source access query fragment* specifies sources to access and calls a *stream function* that generates a stream of events from the sources to process.
- A *processing query fragment* specifies the scientific analyses in terms of complex filters over the generated events. The processing query fragment includes transformation views.

To understand the implications of the streaming approach, the ALEH application was reimplemented in SQISLE in a streamed way. The implementation is called *SALEH* (Streamed ALEH). In SALEH events and their derived properties are represented in terms of the same functional schema as used in the loading approach. In contrast to the loading approach, where the schema is defined in terms of traditional objects, in SALEH the

functional schema is defined in terms of stream objects. The cuts as defined in ALEH can be directly used also in the processing query fragment of SALEH queries, since the cut definitions in terms of the functional schema are logically independent from the schema implementation.

In the Thesis it is shown that naïve execution of SALEH stream queries without advanced query optimization is slow. It is therefore investigated whether the query optimization strategies from the loading approach can be utilized also for the streaming approach. Since, with the streaming approach events are not stored in SQISLE, there are no statistics available for cost-based optimization about the data collections, and statistics instead must be collected dynamically during query execution. For this we introduce a new operator, the *profile-controller*, which enables different *runtime query optimization* strategies. During query execution it checks goodness of statistical estimates, and, when it has determined that sufficient statistics are collected, it dynamically reoptimizes the query and switches to query execution without profiling overhead by disabling collecting and monitoring statistics. It is shown that the runtime query optimization strategies improve performance of stream analysis queries substantially compared to naïve execution.

However, even with the profile-controller, the performance of some stream queries is still much slower than the corresponding manually coded C++ programs performing the same analyses. The bottleneck is in the transformation views, which are called many times for the same event from a file stream. Therefore, some general rewriting rules of complex expressions are introduced to improve the performance of the transformation views. Furthermore, to avoid repeated execution of them, materialization of the transformation views is implemented. In addition, materialization of nested subqueries and rewriting rules to remove unnecessary vector constructions are done for the analysis query fragments. The source access query fragment and transformation views need to access meta-data from the schema during query execution. To eliminate the access to the schema, compile time evaluation [59][77] is applied to expressions in queries accessing the schema.

All these techniques together with the presented novel query optimization techniques make performance of the stream analysis queries close to the corresponding C++ programs.

In summary the results of this Thesis are:

- It is shown that the HEP application and its analyses can be implemented in terms of high-level queries. The events are represented using a functional data model, and queries are defined using a functional query language.
- It is shown that, based on our contributions to query processing, the scientific application queries can be executed as efficiently as with a hard-coded C++ approach.

- The streaming approach is used to select complex objects from files. It is shown to perform much better than the loading approach. The streaming approach is based on the implementation of the data type *Sobject*, which efficiently represents complex objects such as events with complex structures. The streaming approach obtains efficient plans by runtime query optimization strategies utilizing the profile-controller operator, which encapsulates in each query the query fragment that tests complex conditions over event properties. It controls collection of statistics for the fragment, reoptimizes the fragment at runtime based on collected statistics, and dynamically switches optimization strategies.
- A novel cost model for aggregate functions over nested subqueries is developed, and it is shown to improve performance of complex queries with many aggregate functions over complex nested subqueries.
- The profiled grouping approach automatically fragments a query into groups and profiles each group to measure its real cost and fanout on a subset of events. It is shown that, with the profiled grouping approach and the cost model for aggregate functions, the query optimizer is able to find better performing plans than without the profiled grouping approach.
- Rewritings of query expressions and materializations of views called in a query further improve performance. It is shown that these techniques significantly improve performance of queries with low selectivities.
- The integration of a DBMS with a Grid infrastructure utilizes Grid computational resources for scalable execution of the application queries over data stored in a Grid. The integration is based on an architecture where data, queries, and a database system are shipped to computational resources accessible through the Grid infrastructure. It is shown that this architecture allows executing queries in parallel on non-dedicated external resources managed by a Grid infrastructure.

The rest of the Thesis is organized in the following way. Chapter 2 describes the ATLAS application, which motivates the Thesis, and gives background on the technologies extended in the Thesis. Chapter 3 presents contributions on the query optimization and evaluates the contributions for the loading scenario, based on our paper [38]. The stream system SQISLE and the streaming implementation of ALEH are described in Chapter 4. Chapter 5 describes integration of the DBMS with a Grid infrastructure based on our paper [37]. The chapter presents the parallel architecture of executing expensive queries in the Grid environment. It is followed by related work in Chapter 6, which describes work related to all parts of the Thesis. Chapter 7 summarizes the Thesis and presents future work.

2. Background

This chapter describes the basis for the Thesis. First, the scientific application used in the Thesis is described in Section 2.1. Related database technologies are described in Section 2.2. They are followed by description of the DBMS Amos II, which is extended in this work, in Section 2.3. Finally Section 2.4 presents Grid technologies and in particular the Advanced Resource Connector (ARC).

2.1 The ATLAS Application

Our test application is from HEP, where lots of data is produced by LHC detectors, e.g. ATLAS [7]. Currently the ATLAS experiment simulates data to test its software infrastructure and to provide test data for physicists. The physicists use the simulated data during development and testing their theories. Many more physicists are going to be involved in the analyses of real data after LHC and ATLAS detector start to produce collision events at very high rate.

2.1.1 Application Data

The data produced by the ATLAS experiment describe collisions of particles. Each collision generates new particles, which are measured by the ATLAS detector, or the measurements are simulated by the ATLAS experiment. The measurements of particles produced in a collision form a *collision event*. Each event is conditionally independent given experimental run conditions, since each collision is preformed independently. Distribution of event property values are the same for events produced with the same experimental run conditions.

The ATLAS experiment generates measurements as raw data, which are processed by several phases of ATLAS software and summarized in high-level collision descriptions [8]. This work focuses on the high-level descriptions of simulated collision events as in [47]. Each such *event* is described by *event properties*, which are general measurements about the collision and sets of generated particles of various types. An example of a general collision measurement is the missing momentum in x and y directions (*PxMiss* and *PyMiss*). The generated particles of an event are, e.g.,

electrons, *muons*, and *jets*. The particles of the events are described by the same set of properties such as 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*). Therefore, our application data are sets of independent events described by their properties.

The events are stored in files, which are usually generated on Grid computational resources and then stored on Grid storage resources or locally. The test data for [47] and this Thesis were produced in *NorduGrid* [31], and the files used in the Thesis are stored in NorduGrid storage resources. The names of the files reflect experimental run conditions and contain data partition identifiers within the experiment, thus we assume that two events are produced with the same experimental run condition if the names of the source files differ only by the partition identifiers.

Events are accessed from the files through the C++ framework ROOT [18]. ROOT is a general framework, which provides ability to store data as collection of tuples of simple C values or as collection of C++ objects. One ROOT file can contain several independent collections of data. Thus it is necessary to specify the ROOT file, the internal path to a collection, the name of the collection, and the tuple or object positions in the collection to retrieve data. ROOT also provides an interface to retrieve metadata about the files that includes, for example, which collections are stored in the file, paths to the collections, structure for each collection, and amount of data stored in each collection.

The simulated events available for this Thesis are stored in ROOT files in a collection called *h51* as tuples of simple C values. Each element of a ROOT tuple contains either a real or integer number or a C array of numbers. The element values are accessed by their position in the ROOT tuple. The metadata about the collection of tuples describe attributes and mappings of the attribute names to position identifiers and types of the corresponding elements in the tuples.

All ROOT files, which store events of the Thesis' application, have the same structure and the file names contain meta-information about stored events. Events are stored in a collection object, named *h51*, located in */ATLFAST* in the ROOT files. Examples of file names are *bkg2Events_000.root*, *bkg2Events_001.root*, and *signalEvents_000.root*. The names of the first two files describe that their events are from the same set produced in an experiment named *bkg2* and have the same distribution. The numbers *000* and *001* in the file names identify subsets of the event set. The experiment *bkg2* simulates background events, which are unlikely to produce Higgs bosons and therefore the analysis queries searching for Higgs bosons have high selectivities. The events from *signalEvents_000.root* are simulated in a different experiment named *signal* and have another distribution than the events produced in the experiment *bkg2*. The experiment *signal* produces signal events, which are likely to produce Higgs

Table 2.1. Structure of the event tuples and example of events from file *bkg2Events_000.root*. The first row contains logical names of the attributes, the second row defines positions of the attributes in the tuples, and the third row presents the types of the tuple elements. The remaining rows contain values of example event attributes, where arrays are denoted by the notation {…}.

EventId	**Nele**	**Kfele**	**Pxele**	**Pyele**	**Pzele**	**Eeele**	**Nmuo**	**Kfmuo**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
int	**int**	**int []**	**float []**	**float []**	**float []**	**float []**	**int**	**int []**
0	0	null	Null	null	null	null	0	null
1	0	null	Null	null	null	null	1	{13}
…								
3	2	{-11,11}	{-20.67, 49.11}	{98.32, 67.51}	{36.43, -29.14}	{106.8, 88.43}	1	{13}
…								

Pxmuo	**Pymuo**	**Pzmuo**	**Eemuo**	**…**	**Pxmiss**	**Pymiss**	**Pxnue**	**Pynue**
9	**10**	**11**	**12**		**54**	**55**	**56**	**57**
float []	**float []**	**float []**	**float []**		**float**	**float**	**float**	**float**
null	null	null	null		20.43	19.80	0.039	19.93
{-32.03}	{2.640}	{33.81}	{46.65}		107.5	-4.065	101.9	-10.37
…								
{-41.23}	{-21.16}	{-41.06}	{61.92}		43.77	8.846	36.94	17.30
…								

bosons and therefore the analysis queries searching for Higgs bosons have low selectivities.

The structure of the ROOT tuples is the same in all test files. Each ROOT tuple contains 58 attributes. Some of the attributes are presented in Table 2.1. Position 0 of the tuples stores a unique ID number of an event within the file (*EventId*). Attribute *Nele* at position 1 describes how many electrons are contained in the event. The properties of electrons are presented in attributes at positions 2-6. They are followed by properties of other particles of events and general event properties. For example, attributes at positions 54 and 55 contains values of the missing momentum.

Table 2.1 includes examples of values for some events. For example, event with *EventId* equal to three contains two electrons. The properties of the electrons are stored as vectors in the attributes *Kfele*, *Pxele*, *Pyele*, *Pzele*, and *Eeele*. In the example each attribute array contains two elements to store property values for both the electrons. Then one of the electrons is constructed by values stored in the attribute vectors at position zero and is uniquely identified by the source event, which is from *bkg2Events_000.root* and has *EventId* three, and the position in the source event (*particle identifier*), which is zero. The other electron is constructed by values stored in the attribute vectors at position one and is uniquely identified by the source event and the particle identifier equal to one.

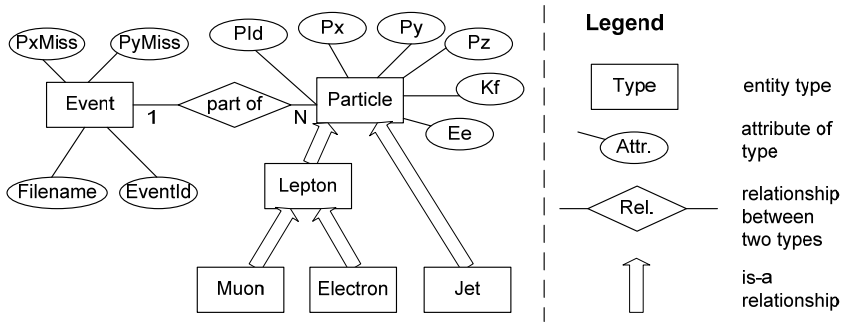

Figure 2.1. An EER diagram of the event collision data.

The above way of modeling events in the files is not natural, since every particle is split between several attributes and one attribute contains values from several particles indexed by the particle identifier. It is more natural to represent particles as instances of corresponding particle types, e.g., as electron or muon objects contained in the event objects.

An extended entity-relationship (EER) diagram [44] in Figure 2.1 models the event collision data as objects of different types. The diagram describes only those event properties, which are required by analyses in [15] and [47]. Analyses there are defined in terms of leptons and jets, which are represented by types *Lepton* and *Jet*, respectively. A lepton is either an electron or a muon, thus the types *Electron* and *Muon* are subtypes of type *Lepton*. Since all kinds of particles have the same attributes, the general type *Particle* is defined and all particle subtypes inherit its properties. The attributes of particles are the ID-number of a specific kind of a particle (*Kf*), momentum in x, y, and z directions (*Px*, *Py*, and *Pz*), the amount of energy (*Ee*), and the identifier of the particle within an Event (*PId*). Particles are contained in events. The attributes of an event are the missing momentum in x and y directions (*PxMiss* and *PyMiss*), the name of a source file (*Filename*), and the identifier within the file (*EventId*).

In the Thesis the same logical schema is defined based on this schema for both the loading and streaming approaches. The logical schema is called the *particle schema* and is defined using a functional data model [79], presented later in the Thesis (Figure 2.3). Scientific analyses of event data are specified as queries over events, which are expressed in terms of the particle schema. However, different physical implementations of the particle schema are used for the two approaches.

2.1.2 Application Analyses

Scientists analyze the event data to select interesting events. An analysis of the events consists of selecting those events that can potentially contain charged Higgs bosons [7]. A number of complex predicates, called *cuts*, are

applied to each event and the events that satisfy all cuts are selected. Selectivities of cuts are similar for the event sets that are produced with the same experimental run condition. Since events are independent, the analysis of each event is performed independently from other events.

Example 2.1. An example of a scientific analysis of the events is presented in [47]. It defines four cuts: *Jet Cut*, *Top Cut*, *Three Lepton Cut*, and *Two Lepton Cut*, and is called *Four Cuts Analysis*. *Top Cut* and *Jet Cut* are the most complex cuts defined over jets. The definition of *Top Cut* in paper [47] is:

> The *Top Cut* requirements are:
> Events must have at least three jets, each with $p_T > 20$ GeV in $|\eta| < 4.5$.
> Among these, the three jets most likely to come from the top quark are selected by minimizing $|m_{jjj} - m_t|$, where m_{jjj} is the invariant mass of the three-jet system. It is required that $|m_{jjj} - m_t| < 35$ GeV.
> Among these three top jets, the two jets most likely to come from the *W* boson is selected by minimizing $|m_{jj} - m_W|$, where m_{jj} is the invariant mass of the two-jet system. It is required that $|m_{jj} - m_W| < 15$ GeV.

Where p_T (called *Pt* in the Thesis) is calculated over the momentum of a particle by formula:

$$p_T = \sqrt{Px^2 + Py^2} \tag{2.1}$$

, η (called *Eta* in the Thesis) is calculated over the momentum of a particle by formula:

$$\eta = 0.5 \cdot \ln\left(\frac{\sqrt{Px^2 + Py^2 + Pz^2} + Pz}{\sqrt{Px^2 + Py^2 + Pz^2} - Pz}\right) \tag{2.2}$$

, the invariant mass is calculated over set of *n* particles by:

$$m = \sqrt{\sum_{i=1}^{n} Ee_i \cdot \sum_{i=1}^{n} Ee_i + \left(\sum_{i=1}^{n} Px_i \quad \sum_{i=1}^{n} Py_i \quad \sum_{i=1}^{n} Pz_i\right) \cdot \begin{pmatrix} \sum_{i=1}^{n} Px_i \\ \sum_{i=1}^{n} Py_i \\ \sum_{i=1}^{n} Pz_i \end{pmatrix}} \tag{2.3}$$

, m_T is the invariant mass of the top quark (174.3 GeV), and m_W is the invariant mass of the W boson (80.419 GeV). The definition of *Jet Cut* can be found in [47].

Three Lepton Cut and *Two Lepton Cut* are simpler than the cuts above and they are defined over leptons. The paper [47] describes *Three Lepton Cut* as:

The *Three Lepton Cut* requires:

Exactly three isolated leptons ($l = e$ or μ) with $|\eta| < 2.4$, with $p_T > 7$ GeV and at least one of which with $p_T > 20$ GeV.

Where l means a lepton, e means an electron, and μ means a muon. The definition of *Two Lepton Cut* can be found in [47]. ■

The scientists implement their cuts in some programming language and experiment with the implemented cuts and combinations of the different cuts during developing and testing their scientific theories. Currently the analyses are usually implemented in C++, which requires a lot of effort. Furthermore, the event collision data are stored in ROOT files in an unnatural way as discussed in the Section 2.1.1. Therefore, it can be difficult to understand and modify programs implementing the analyses. Furthermore, modification and extension of analyses requires code recompilation and uploading compiled binaries to external computational resources.

Example 2.2. The theory presented in [47] and Example 2.1 is result of several years of research. The work continued the theory presented in [15]. To be able to test new ideas, the requirements for the interesting events from [15] were implemented as six cuts in a C++ program, which was then modified and extended with the new ideas. The six cuts were *Hadr Top Cut*, *B Tag Cut*, *Jet Veto Cut*, *Z Veto Cut*, *Three Lepton Cut*, and *Other Cuts*. Then *Hadr Top Cut* was modified first and *B Tag Cut* was removed. The definition of the implemented and modified cuts at this point is used in the Thesis for evaluation. This analysis is called *Six Cuts Analysis* and can be found in Appendix A in natural language.

The cuts over ROOT tuples from Table 2.1 were implemented by a scientist in a C++ program without abstracting into a high level data model, e.g., as presented in Figure 2.1. Thus duplicated code was introduced, for example, in implementation of isolated leptons for electrons and muons in *Three Lepton Cut*. Global variables were used to keep intermediate results between cuts, for example, set of isolated leptons, which are used in *Three Lepton Cut*, *Jet Veto Cut*, and *Other Cuts*. As result it is difficult to understand and modify the code.

During the implementation of the cuts in the C++ program a manual optimization of the code was done. The cuts were ordered in such a way that the program should execute efficiently. The implemented order of the cuts is *Three Lepton Cut*, *Z Veto Cut*, *Hadr Top Cut*, *Jet Veto Cut*, and finally *Other Cuts*. Furthermore, materialization of temporary results of calculations is manually implemented in the C++ program by storing the temporary results in global variables, which are reset at the beginning of the analysis of each event. The results of calculating *isolated leptons*, *ok jets*, *b-tagged jets*, and *w jets* are materialized in C++ vectors. The materializations limit the

possibility to reorder cuts, since the reordering sometimes requires manually moving materialization code from one cut to another. ■

To investigate how database query processing technologies can improve scientific analyses, *Six Cuts Analysis* (Example 2.2) is implemented in a query language as six cut functions over the events modeled by a high-level schema (Figure 2.1) and *Four Cuts Analysis* (Example 2.1) is implemented as four cut functions. *Six Cuts Analysis* queries are evaluated for both the loading and streaming approaches. It is demonstrated that the query language implementation has comparable performance as the C++ implementation described in Example 2.2. *Four Cuts Analysis* queries are evaluated only for the streaming approach.

2.2 Database Technologies

Database technologies provide efficient and scalable processing of large volumes of data. The traditional way to use these technologies is to store data in a database managed by a *database management system* (*DBMS*) and then specify data processing by queries to the DBMS [44]. This approach does not suit all applications. In some cases, data can not be stored in a DBMS and instead they are streamed through a *data stream management system* (*DSMS*) [9]. In a DSMS queries are processed over streams instead of querying stored data. In other cases, data are distributed in a network or Internet and then a middleware DBMS (called a *federated* or *mediator* database) integrates the data to answer a user query [76].

The database community has developed and continues to develop technologies to support different applications to process data in efficient and scalable ways [53]. Therefore, data-intensive applications can gain a lot by utilizing appropriate database technologies. For example, the application described in Section 2.1 does not utilize any database technology for analyzing the huge amount of produced scientific data. This Thesis investigates how database technologies can be utilized for applications of this kind and develops new database techniques to achieve efficiency and scalability in execution of analysis queries.

The first step in using databases is designing a conceptual schema of data. Entity-Relationship (ER) modeling [20] is commonly used to model data on high-level. During the ER modeling entity types with their attributes are defined to model real world objects with properties. Entity types are related to each other by relationships. The result of modeling can be presented on a diagram, for example, by using the entity-relationship notation. ER model can be extended with inheritance. For example, in Figure 2.1 an extended entity-relationship (EER) notation is used to represent a conceptual schema.

The conceptual schema is implemented in a DBMS and mapped into the DBMS's *data model*. A data model is a collection of data types, operators manipulating data stored using the data types, and general integrity rules constraining the stored data [24]. The relational data model [23] is most commonly and widely used in databases, and many commercial DBMSs are based on it. Such DBMSs are called Relational DBMSs (RDBMSs). In the relational data model entity types are represented by relations, which can be seen as tables. Entities are stored as tuples (called table *rows* in the standard query language SQL [27]). Attributes of a tuple (column values in a table row) correspond to attribute values of an entity. RDBMSs maintain extents for every relation to represent its tuples. They also maintain *primary key*, *unique key*, and *foreign key* constraints on attributes. Values of the primary key attribute(s) of a relation identify uniquely tuples of the relation. Unique key on an attribute specify that values of the attribute should be unique in different tuples. Foreign key attributes of relations store relationships to other relations. RDBMSs provide support for keys on single attributes and compound keys defined over several attributes. For faster access values of some attributes are indexed. Most RDBMSs always maintain indexes on primary keys. Other attributes are indexed on requests of a database administrator (DBA).

Each DBMS implements a query language, which is used to store, modify, and search data from the RDBMS. Commercial RDBMSs implement the high-level, nonprocedural standard query language SQL [27]. A query expressed in SQL specifies which data to retrieve. How data is going to be physically accessed from a database is decided by the DBMS.

In SQL data retrievals specify data source relations, selection conditions on tuples, and which attributes to be presented in the result. If data are retrieved from more than one relation, tuples from different relations are *joined* with each other using some join condition, e.g. equality on a foreign key. A *selection condition* is specified as a set of operators on attribute values of tuples. The operators can be logical, numerical, string, or complex logical operators. Results of queries are formed by values of specified attributes and values of other attributes are projected away. Queries with joins, selection conditions, and attribute projection are called *Select-Project-Join* (*SPJ*) queries.

SQL queries can be more complex than SPJ queries. Selected tuples can be grouped and *aggregate functions* are applied over attribute values of the tuples grouped together. Selection condition of the queries can contain *nested subqueries* with aggregate functions over their results. A nested subquery can access a variable bound to a relation from the a parent query. Such a relation variable is called a *correlated variable*.

RDBMSs support *views*, which are virtual relations defined by queries on top of physical relations or other views. Views provide modularity in query definitions. Some DBMSs extend SQL to allow parameterized views.

The main limitation of the relational data model is its limited expressiveness. For example, it does not support inheritance. The Thesis uses and extends a DBMS, which is based on a *functional data model* [40]. The functional data model provides higher expressiveness than the relational data model, and naturally supports relational and object-oriented data. Functional data models are based on mathematical notion of functions. DBMSs with a functional data model, *functional DBMSs*, implement a *functional query language*. Functional query languages give ability to declaratively specify through functions complex data processing in addition to the selection of which data to retrieve.

2.2.1 Query Processing

When a DBMS receives a query to select data it processes the query in several phases. The query processing phases are presented in Figure 2.2 [52]. In the first phase a *parser* checks syntactic and semantic correctness of an input *query* and creates a *calculus representation* of the query. Then a *rewriter* transforms the calculus representation by applying different rewriting rules. One of the most important rewriting is *view expansion*, where views are substituted with their definitions.

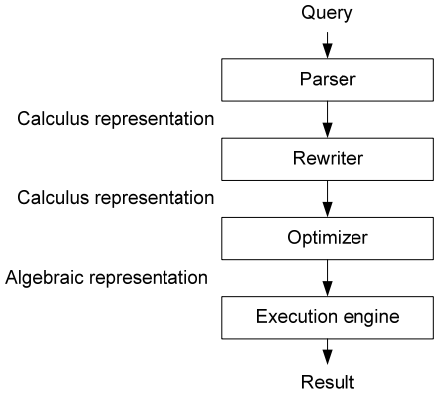

Figure 2.2. General query processing steps.

After the pre-processing phase the query *optimizer* transforms the predicates from the calculus representation of the query into *algebra operators* implementing the query. The operators are placed in an order called the *execution plan* of the query. Since there are many possible execution plans for a given query, the query optimizer has the goal to find an efficient execution plan. The query optimizer can be based on heuristics, cost models, or usually a mixture of both heuristics and cost models. In a heuristic based query optimizer heuristic rules define choice of operators and their order. In a *cost-based optimizer* the cost of each operator is estimated based on data statistics and an *operator cost model* and then the total cost of

an execution plan is minimized based on the cost model. Query optimizers of relational DBMS usually mix these two approaches. For example, RDBMSs often use a heuristic rule that selection operators should be executed as early as possible [57]. Then the order of joins and the choice of physical operators implementing joins, e.g. a *nested loop join* [44], are optimized by minimizing the cost of the final plan. The optimization is usually performed by an optimization algorithm based on dynamic programming [87]. Such algorithms can find optimal plan in terms of estimated cost. However, optimization algorithms based on dynamic programming can handle only small number of joins. Thus some DBMSs implement randomized optimization [56][82] or greedy optimization [60] to handle larger queries.

In the last phase an *execution engine* executes the execution plan by interpreting the plan. For example, a nested loop join of two relations called *outer* and *inner* relations loops over all tuples from the inner relation for each accessed tuple of outer relation to produce the join result. The *result* of the query execution is shipped to the user.

This Thesis extends a DBMS that implements all these phases. After parsing a query, several rewriting rules are applied including view expansion. The Thesis proposes additional rewriting rules to reduce the amount of operators in the execution plan. Query optimization is performed by a cost-based optimizer. A novel cost model for operators used in the application queries is presented in the Thesis. The DBMS provides three optimization algorithms: based on dynamic programming, randomized optimization, and greedy optimization. All the three algorithms are used in the Thesis. The execution plan produced by the query optimizer is interpreted during the query execution.

2.2.2 Data Stream Management Systems

There are applications, where data is constantly produced as streams. Storing such data can be inefficient or impossible. To enable queries for such applications Data Stream Management Systems (DSMSs) were developed [9]. In DSMSs analyses are specified in high level query languages similar to SQL over data which are streamed from sources [85]. It is common to assume that data is ordered in a stream, and a data stream is infinite and cannot be repeated. In a DSMS data is not available all the time and execution is performed when data arrives, *data driven* execution, while in a DBMS data is always available and execution is performed when a query is issued, *demand driven* execution.

Since a stream is assumed to be infinite and not repeatable, DSMS queries cannot be executed in the same way as by a DBMS. For example, the nested loop join in a DBMS accesses data from inner tables many times. In the case if inner relation is a stream, it cannot be called several times and data of the stream cannot be stored either. Therefore, a concept of data *windows* is

implemented in DSMSs [85]. Usually a data window contains only the most recent data. Thus operators that require accessing the same data several times are executed only over recent data and therefore the query results for the entire stream are approximated.

This Thesis investigates scalability and efficiency of query processing over complex objects streamed from sources, e.g. ROOT files in the ATLAS application, and implements a new DSMS. In contrast to data driven DSMS our DSMS is demand driven, i.e. it controls when each new complex object is produced by a stream. In DSMSs the elements of the streams are usually relatively simple records, while is our case the elements are complex objects. Since in our kind of applications each complex object is analyzed independently, our DSMS needs to process only one most recent element of the stream at a time. Furthermore, our streams are finite, thus exact query results can be obtained over entire stream. Therefore, windows and orders are not utilized.

2.2.3 Distributed Databases

Distributed database systems [76] allow to process queries on more than one database server distributed over a network. Usually DBMSs with data are preinstalled on server machines and available before queries are issued. Submitted queries are processed on distributed DBMSs transparently for the user. Distributed database systems take care on splitting a submitted query into query fragments, executing the query fragments on relevant source DBMSs, and integrating results of the query fragment executions. Traditionally distributed database systems minimize data volumes shipped over network between the distributed DBMSs.

This Thesis presents a distributed architecture, where DBMSs are not pre-installed. Instead the DBMS itself is shipped to computational resources in addition to shipped query fragments and data. This makes possible to dynamically utilize computational resources of Grids without preinstalling DBMSs.

2.3 The Functional DBMS Amos II

This Thesis extends a research DBMS Amos II [79]. Amos II provides a functional data model with user-defined data types, a functional query language, external interfaces to C/C++, Lisp, and Java, query processing with abilities to implement new rewriting rules and different optimization methods, support for wrappers and mediators, and support for distribution and stream environments.

The basic concepts of the functional data model of Amos II are *objects*, *types*, and *functions*. All data are represented by objects, which can be *literal*

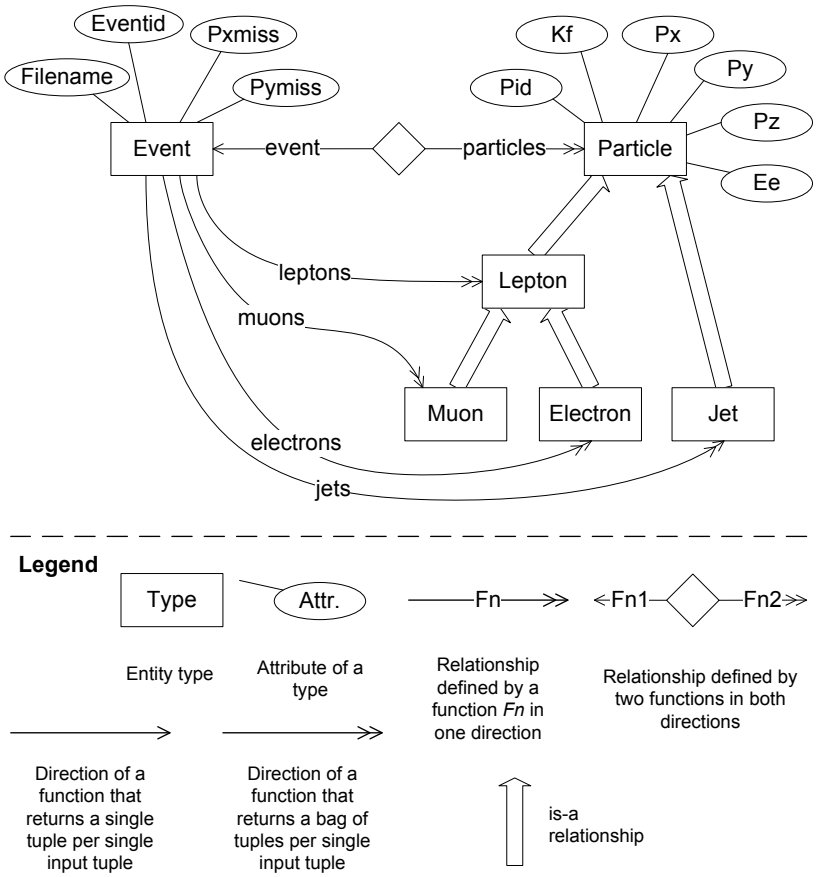

Figure 2.3. The particle schema of the event collision data in the functional data model.

objects or *surrogate objects*. Literal objects represent primitive data such as numbers, strings, and collections and belong to literal types, e.g. *Integer*, *Real*, *Charstring*, *Vector*, and *Bag*. Complex data are stored as surrogate objects, which are associated with *object identifiers* (*OIDs*). Objects are classified to *types*. Types are defined by users, are used to model real world entities, and are arranged into hierarchies. Amos II maintains extents of surrogate objects for every user-defined type. Values of surrogate objects are related to the objects by *functions*. Functions also define relationships between objects of different types. Therefore, both attributes and relationships are modeled by functions, which are called *stored functions*.

The functional data model of Amos II is well suited to model scientific data. For example, the EER model of the application data presented on Figure 2.1 is mapped into the particle schema in the functional data model as presented on Figure 2.3 and defined in Amos II. All presented entities are directly mapped to types, which are organized in a type hierarchy. Attributes

of the type *Event* are implemented as stored functions named *EventId*, *FileName*, *PxMiss*, and *PyMiss*. These functions take objects of type *Event* as argument and return literal objects of types *Integer*, *Charstring*, *Real*, and *Real*, respectively, as results. Analogously attributes of entity *Particle* are implemented as functions over type *Particle* and return numbers. Types *Lepton* and *Jet* are implemented as subtypes of type *Particle*, and therefore, inherits all functions defined for the type *Particle*. Type *Lepton* is supertype for types *Electron* and *Muon*. The relationship between *Event* and *Particle* is implemented by the function *event*, which takes an object of type *Particle* as argument and returns an object of type *Event* as result, and by the functions from type *Event* to each particle type, which return all particles of the kind belonging to an input event.

2.3.1 Functions in Amos II

A function in Amos II can be a *stored function* implementing attributes or relationships, a *derived function* implementing parameterized views, or a *foreign function* implemented in a procedural sub-language of Amos II or some external programming language. Basic operators such as less, equality, plus, absolute value are implemented as foreign functions in C. Queries and functions return a single value or bags of values.

Functions can be defined as *multidirectional* to represent different implementations for a function for each of its inverses. A multidirectional function has different implementations for different *binding patterns* [44], i.e. which argument or result parameters are bound in a query. Multidirectional functions can be defined explicitly by providing different implementations for different binding patterns. Multidirectional functions provide flexibility for the query optimizer to implement access to external data structures. For example, a function *vref* returning an element of a vector is defined as multidirectional foreign function for two binding patterns *bbf* and *bff*:

```
create function vref(Vector v, Integer i) -> Object o
  as multidirectional
    ("bbf" foreign 'vrefbbf')
    ("bff" foreign 'vrefbff');
```

The first binding pattern *bbf* means that both the vector *v* and the position *i* of the element in the vector are known. Therefore, the implementation *vrefbbf* is going to be called to access the element in the vector directly. With the second binding pattern *bff* only the vector *v* is known. Therefore, the implementation *vrefbff* is used to iterate over all elements of the vector *v* and emit values for both the index *i* and element *o*.

2.3.2 Query Language and Query Processing in Amos II

The query language of Amos II is called *AmosQL*. In AmosQL queries are specified in SELECT-FROM-WHERE statements. The FROM clause specifies type extents to access, the WHERE clause specifies selection conditions, and the SELECT clause specifies the values to return. SELECT and WHERE clauses can contain calls to any kind of functions.

AmosQL queries are processed in four phases as presented in Figure 2.2. First, a query is parsed and translated in a logical calculus representation called *ObjectLog* [66], which is a dialect of Datalog [44]. Then various rewriting rules are applied to the query. *View expansion* is performed by substituting derived functions with their definitions. Another rewriting rule applied to the query is *partial evaluation*, which reduces query fragments by evaluating them during the rewriting phase [77]. After rewriting the query represented in ObjectLog is optimized by a cost-based query optimizer, which produces an execution plan represented in an object algebra.

In Amos II each function is associated with cost models consisting of execution *costs* and *fanouts*. Costs of functions indicate if one function is more expensive in terms of its execution time than another one. The fanout of a function estimates how many tuples are produced by the function per one input tuple. The fanout of selection predicates (called *selectivity*) are less than one since they filter their inputs. Numerical functions usually transform an input value into some output tuple; thus their fanouts are equal to one. The fanout of a function returning a bag (called its *cardinality*) is equal to the size of the bag. Default statistics are defined for different groups of common functions, e.g., bag valued functions have fanout 100, selective predicates have fanout 0.4, and other foreign functions have fanout one. More specific cost models can be defined for functions by providing either cost hints, which are constant numbers, or cost functions, which dynamically calculate operator costs and fanouts on the query optimizer's requests. Different cost models can be used for different binding patterns of a function.

The query optimizer chooses the operators that implement the functions for one of their binding patterns, and places operators in a sequential execution plan in certain order. The choice and place of operators depends on two factors: each operator should be *executable*, i.e., the operator's arguments should all be bound, and the total cost of the execution plan should be minimized. Three optimization methods are available in Amos II. They are dynamic programming, greedy optimization, and randomized optimization. The optimization method based on dynamic programming [87] finds the optimal execution plan according to the cost model, i.e. the optimal plan has the smallest total cost among all possible execution plans for the query. The total cost for nested loop joins is calculated by formula [66]:

$$\sum_{k=1}^{n}\left(cost(p_i)\cdot\prod_{l=1}^{k-1} fo(p_k)\right) \tag{2.4}$$

, where p_k is an operator placed at position k in the sequential execution plan consisting of n operators. The cost of operator p_k is $cost(p_k)$ and its fanout is $fo(p_k)$. Calculation of the total cost assumes that all n operators are independent from each other.

Dynamic programming can handle only queries with few operators, since, e.g., the worst case complexity of System R algorithm [87] is $O(2^N)$ for a query with N joins. The other optimization methods, greedy optimization and randomized optimization, are able to handle queries of any size, but they do not guarantee to find the optimal plan.

Greedy optimization [66] is assigning ranks to operators and sorting the operators according their ranks. An execution plan is constructed by chosing an executable operator with smallest rank among all operators, which are not yet in the plan. The rank for an operator p_k is calculated by formula:

$$\frac{fo(p_k)-1}{cost(p_k)} \tag{2.5}$$

The idea behind the rank formula is that selective operators are placed as earlier as possible and operators with fanouts bigger than one are placed as late as possible. Among selective operators the cheapest is placed first. Among operators with fanouts bigger than one the most expensive is placed first. To be able to compare operators with fanouts equal to one, their fanouts are replaced with 0.99 while calculating of their ranks.

This greedy optimization finds suboptimal plans in complex cases, but it is very fast.

The randomized optimization [71] is a two-phase algorithm based on random walk. It minimizes plan cost calculated by formula (2.4). The first phase is called *Iterative Improvement* (*II*), which randomly generates an executable query plan and searches for local minimum in its each iteration. The cheapest plan among of all iterations is returned as result of the iterative improvement. On the result plan of the iterative improvement *Sequence Heuristic* (*SH*) is applied. Each iteration of the sequence heuristic randomly chooses a neighbor plan to the best known plan and searches for local minimum from the neighbor by random walks. The result plan of sequence heuristic is provided as the final execution plan. The number of iterations for iterative improvement and sequence heuristic phases can be tuned. For large and complex queries the randomized optimization needs to run for a long time to obtain a good plan. Randomized optimization is able to find much better plans than greedy optimization, but it can take a lot of time for the randomized optimization to find a good plan.

The execution engine interprets an execution plan obtained by one of the optimization methods. Operators in a query plan are executed iteratively in a stream fashion in the same order as in the plan by a nested loop join.

This Thesis implements the ATLAS application, a DSMS SQISLE, and parallel query management system POQSEC as extensions of Amos II. The query language of Amos II is extended with numerical and aggregate functions to define analyses queries for the ATLAS application. The data model of Amos II is extended with data type *Sobject* for efficient processing events with complex structures streamed from files or other sources. The query processing of Amos II is extended with *runtime query optimization*, which collect data statistics and optimizes queries at runtime, and *profiled grouping*, which fragments queries in groups, measures execution time and fanout of each group, and optimizes join-order of groups. Operators cost models of Amos II are extended with *aggregate cost model* for aggregate functions over nested subqueries. These extensions are important contribution of the Thesis.

2.4 Grid Technologies

Grid technologies are being developed to establish infrastructures for coordinating and sharing distributed heterogeneous resources between multiple users and across organizations [35]. Grid infrastructures emerged first within scientific communities. The goal of Grid there is to provide uniform access to heterogeneous computational resources, e.g., clusters, through Grid infrastructures. Most of Grid infrastructures are based on kernel software developed and provided by the Globus Alliance [41]. The standardization of Grid is managed by the Open Grid Forum (OGF) [75].

In Sweden most commonly used Grid infrastructure is the Advanced Resource Connector (ARC) [32]. The Thesis utilizes resources of Swedish National Grid, Swegrid [90]. Swegrid consists of six computational clusters, which are accessible through ARC. Section 2.4.1 describes ARC based on its state at the beginning of 2005.

2.4.1 ARC Grid Middleware

The Advanced Resource Connector (ARC) [73] is a middleware between Grid users and computational resources that are managed by local batch systems. Thus ARC does not control computational resources; instead it submits user tasks to local batch systems on clusters. Each local batch system allocates cluster nodes according to its policy and the current load of the cluster.

The *Computing Elements* (*CE*) are clusters where Grid jobs are executed while *Storage Elements* (*SE*) are file servers where the data to be queried are

stored. The CEs and SEs are managed by ARC and are accessible by submitting Grid jobs to an *ARC Client*. The ARC client is a set of command line tools to submit, monitor, and manage jobs on the Grid. It also has commands to move data between storage elements and clients, and to query Grid resource information such as loads on different CEs and job statistics. Users of ARC always first initiate communication with an ARC client.

The ARC client includes a resource brokering service [33] to find suitable resources for jobs. Jobs are described in a resource specification language, *xRSL* [86], which includes specification of, e.g.:

- A user executable and its arguments to be run on some suitable computing element.
- Files to be transported to and from the chosen computing element before and after the execution.
- Maximal CPU time for the execution.
- *Runtime environments* for the execution. A runtime environment is an additional software package, e.g., an application library such as ROOT [18].
- Standard input, output, and error files for the execution.
- Optional names of the computing elements where the executable can run.
- The number of parallel sub-jobs to be run on the computing element.

In summary ARC requires detailed user specifications to describe computation tasks as xRSL scripts.

POQSEC simplifies this considerably by automatically generating ARC interactions and job scripts to execute a task specified as a declarative query over contents of data files. To manage jobs generated by POQSEC, to track their executions, and to download results we provide a *babysitter* integrated with the POQSEC framework.

3. The Loading Approach

We implemented the ATLAS application (Section 2.1) in Amos II [79] and its analyses as AmosQL queries. The 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 independently 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, aggregate functions, foreign functions, and joins. The aggregate functions compute complex derived event properties. A complex query used as the test example in this chapter is implementation of *Six Cuts Analysis* (Appendix A), which searches for the events likely producing Higgs bosons by applying scientific theories.

The implementation is called ALEH (query system for Analysis of LHC Events for containing charged Higgs bosons). Naïve execution of a complex query described in ALEH performs much worse than the C++ implementation of the corresponding analysis (Example 2.2). This chapter investigates how execution of the query can perform better by improving optimization of the query. To analyze optimization of the queries, the events are loaded into the main-memory database of ALEH. Then query optimization and execution is analyzed for the loaded database. The architecture of the *loading approach* is presented in Figure 3.1. The loading phase is presented in Figure 3.1(a). To load data ALEH accesses an Amos II meta-database called *File DB*, which contains information about files storing events. Then ALEH calls the ROOT library [18] for each file and materializes every event in the ALEH database (*DB*). After data is loaded a user can issue queries to analyze the stored data. During query execution data are accessed from DB and processed by ALEH as presented in Figure 3.1(b).

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

- The queries contain many joins of event properties within each event.

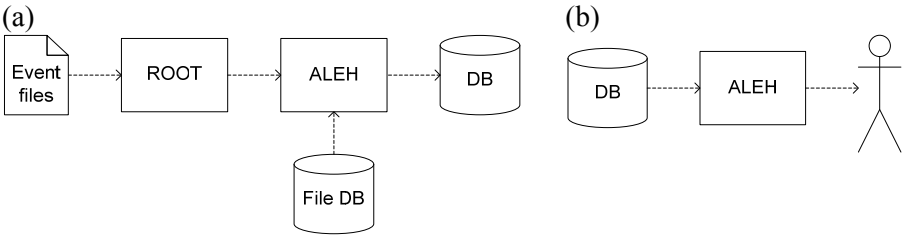

Figure 3.1. Architecture of ALEH with data flow. (a) Modules participating in loading phase; (b) modules participating in query execution.

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

We first investigated whether cost-based optimization improves query execution compared to no optimization. To enable effective cost-based optimization over our kind of scientific queries, we developed an *aggregate cost model* [38] for the operators occurring in the 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 [87] could not be used. Instead randomized optimization [56][71][82] running for a long time and greedy heuristic optimization [60][66] were used. Performance measurements showed that cost-based optimization with the aggregate cost model 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 aggregate cost model becomes unreliable for large plans [54] 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 [38] 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 fanouts and costs per group called *profiled group cost model*. 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 computing estimates based on a cost model. 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 the 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 aggregate cost model, profiled grouping, and the application query in Amos II and evaluated the effectiveness of both ungrouped strategies and profiled grouping in combination with different optimization strategies: dynamic programming, randomized optimization, and greedy heuristic optimization. As references we also compared with a best effort manual optimization. The measurements were made with the two data sets in Section 2.1.1. One data set is with high selectivities of the cuts, and the other one is 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 chapter is organized as follows. Section 3.1 describes implementation of the application analysis in ALEH and a test query used in the rest of the chapter. The aggregate cost model is presented in Section 3.2. Profiled grouping is described in Section 3.3. It is followed by performance measurements for the query execution strategies in Section 3.4. Section 3.5 concludes the chapter.

3.1 High Energy Physics Queries

The data are events and their properties were described in Section 2.1.1. They are loaded into the DBMS from the ROOT files. The ROOT files are associated with *meta-data conditions* for each file, which describe, e.g., experiment settings and what kinds of events were produced. Events and particles from the schema in Figure 2.1 are defined by the *particle schema* in our functional data model as presented in Figure 2.3.

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 assumed 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 *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.

The events are delivered in binary files managed by the ROOT library [18]. A *ROOT loader* is implemented to load events from ROOT files into the Amos II database. The particle schema of the collision events is implemented in the query language AmosQL [79] (see Appendix B) and presented in Figure 2.3. Events are represented by entities of type *Event* with two attributes *PxMiss* and *PyMiss*. Particles are represented by objects with attributes *Kf*, *Px*, *Py*, *Pz*, and *Ee* and several relationships to entities of type *Event*. Particles of different types are represented by different entity subtypes *Muon*, *Electron*, and *Jet*. *Muon* and *Electron* are generalized by an abstract entity type *Lepton*, which is used in definitions of some cuts.

A number of basic numerical foreign functions, e.g. *Pt* and *Eta*, are defined in the database in order to make the analyses. The cuts are expressed as derived functions in terms of these basic functions. 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 predicate to scan the events.

For example, a general query, which implements *Six Cuts Analysis* (Appendix A), is:

```
select e
from Event e
where  hadrtopcut(e) and jetvetocut(e) and
       misseecuts(e) and zvetocut(e) and
       threeleptoncut(e) and leptoncuts(e);
```
(3.1)

Here the functions *jetVetoCut*, *zVetoCut*, *hadrTopCut*, *missEeCuts*, *leptonCuts*, and *threeLeptonCut* are examples of cuts that provide necessary conditions for the collision event *e* to produce a Higgs boson according the theory described in Appendix A. *Other Cuts* is split into *missEeCuts* and *leptonCuts* here. The implementation of these cut functions is presented in Appendix C. The predicate *d(e)*, which accesses events of type *Event*, is

generated by the *from* clause. This general query is a reference query for the rest of the chapter.

The definition of the *Three Lepton Cut* is:

```
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* has the definition:

```
create function isolatedLeptons (Event e) -> Lepton as
select l
from Lepton l
where l=leptons(e) and
      abs(Eta(l))<2.4 and
      Pt(l)>7.0;
```

The other cuts are defined as functions in a similar way.

The *Pt* and *Eta* functions call foreign functions *Pt* and *Eta* over a momentum triple for a given particle *l*. The formulas of the functions are presented in (2.1) and (2.2), respectively.

Before query optimization, derived functions are expanded as views and the query is represented in ObjectLog (see Section 2.3). The plan for the query (3.1) is a conjunction of 51 operators. The predicates are comparisons, numerical operations, aggregate functions, foreign function calls, and joins. The large size of the query makes it difficult to optimize, and dynamic programming [87] cannot be used. We were able to optimize it using randomized optimization [56][71][82], 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:

```
em = Eta(m) AND
aem = Abs(em) AND
aem < 2.4 AND
pm = Pt(m) AND
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 foreign functions *Eta*,

Abs, and *Pt*. It is difficult to estimate selectivities for such predicates defined in terms of foreign functions. 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* 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 fanout estimates and therefore suboptimal execution plans are chosen [54].

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

3.2 The Aggregate Cost Model

We developed a cost model for aggregate functions and numerical functions used in our application, assuming data independence between predicates. Table 3.1 and Table 3.2 define the aggregate cost model. The aggregate 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 costs of different functions are comparable. For example, the cost of aggregate functions *SOME* and *NOTANY* should be complementary and *SOME* is a special case of *ATLEAST*.

The costs of complex numerical functions are approximated according their measured execution time. The costs of basic numerical functions, such as *plus*, *minus*, and *times*, are set to one. The costs for the numerical functions that are used in ALEH queries are presented in Table 3.1. The fanouts of the numerical functions are always one.

The costs and fanouts of aggregate functions are based on the estimated costs and fanouts of subqueries they are applied on.

The cost of an aggregate function depends on the estimated number of tuples produced by its subquery *sq*. For aggregate function *SUM*(*sq*) all tuples emitted by *sq* have to be processed, while for other aggregate functions, such as *SOME(sq)* and *COUNT(sq)=N,* only a limited number of tuples emitted by *sq* are processed. Therefore the cost of an aggregate function is the cost of producing the required tuples by *sq* plus the cost of processing the emitted tuples by the aggregate function. The cost per

Table 3.1. Costs of numerical functions, 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 operator	Cost	Description
PLUS(x,y)=z	1	$z = x + y$
TIMES(x,y)=z	1	$z = x \cdot y$
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 vector of size three
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 vector of size three
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*

produced tuple by subquery *sq* is the estimated total cost of executing the subquery, *cost*(*sq*), divided by its estimated fanout, *fo* (*sq*), i.e. $\frac{cost(sq)}{fo(sq)}$. The cost for the aggregation function to process one received tuple from *sq* is set to one. For example, *SUM*(*sq*) has the cost *cost*(*sq*)+*fo*(*sq*). The cost of *SOME*(*sq*) when *fo*(*sq*)<1 is *cost*(*sq*)+*fo*(*sq*). If *sq* emits at least one tuple the cost becomes $\frac{cost(sq)}{fo(sq)} + 1$ since only the first tuple is processed by SOME. Analogous cost model formulas are developed for other aggregate functions.

The fanout of *SUM*(*sq*) is always one. The fanouts of *SOME*(*sq*) and *NOTANY*(*sq*) depend on the estimated fanout of *sq*. If *sq* emits less than one result tuple the fanout of *SOME*(*sq*) is set proportional to *fo*(*sq*), $\frac{fo(sq)}{2}$. Otherwise it is set to $1 - \frac{1}{2 \cdot fo(sq)}$. Basically, the model converges to one as *fo*(*sq*) increases since it becomes more and more likely that *SOME* is true.

Table 3.2. Cost model for aggregate functions over subquery *sq*, where *cost(sq)* is the estimated total cost of executing *sq*, and *fo(sq)* is the estimated fanout of *sq*.

Operator	Cost	Fanout
SOME(sq)	if $fo(sq)<1$ then $cost(sq)+fo(sq)$ else $\frac{cost(sq)}{fo(sq)}+1$	if $fo(sq)<1$ then $\frac{fo(sq)}{2}$ else $1-\frac{1}{2\cdot fo(sq)}$
NOTANY(sq)	if $fo(sq)<1$ then $cost(sq)+fo(sq)$ else $\frac{cost(sq)}{fo(sq)}+1$	if $fo(sq)<1$ then $1-\frac{fo(sq)}{2}$ else $\frac{1}{2\cdot fo(sq)}$
ATLEAST(sq)=N	if $fo(sq)<N$ then $cost(sq)+fo(sq)$ else $N\frac{cost(sq)}{fo(sq)}+N$	if $fo(sq)<N$ then $\frac{fo(sq)}{2\cdot N}$ else $1-\frac{N}{2\cdot fo(sq)}$
COUNT(sq)=N	if $fo(sq)<N+1$ then $cost(sq)+fo(sq)$ else $(N+1)\frac{cost(sq)}{fo(sq)}+N+1$	if $fo(sq)<N$ then $\frac{fo(sq)}{10\cdot N}$ else $\frac{N}{10\cdot fo(sq)}$
SUM(sq), COUNT(sq)	$cost(sq)+fo(sq)$	1
MINAGG(sq), MAXAGG(sq)	$cost(sq)+fo(sq)$	if $fo(sq)<1$ then $\frac{fo(sq)}{2}$ else $1-\frac{1}{2\cdot fo(sq)}$

The factor two allows *NOTANY* to have a complementary model (see Table 3.2).

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

3.3 Profiled Grouping

The profiled grouping fragments a conjunctive 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 $g_j(e)$ are groups and $g_j(e)$ cannot be further fragmented, i.e. $\neg\exists(g_{j1}(e), g_{j2}(e)) : g_j(e) = g_{j1}(e) \wedge g_{j2}(e)$. 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 minimal groups. One group is:

```
count(isolatedLeptons(e))=3
```

Another group is

```
some(select r
     from Real r
     where r=Pt(isolatedLeptons(e)) 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 aggregate cost model and assuming that *e* is bound by the domain predicate *d*(*e*). Both randomized and greedy optimization were 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 an aggregate cost model may not produce good estimates [54]. 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 loader. The profiler executes each group on the same sample set and calculates fanouts 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.

In our grouping algorithm (Algorithm 3.1) 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

Algorithm 3.1. The grouping algorithm.

```
 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
```

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

3.4 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 functions have the same cost and all foreign functions also have the same cost. Thus the query optimizer does not change the order of aggregate functions and foreign functions 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 e
from Event e
where  threeleptoncut(e) and leptoncuts(e) and
       misseecuts(e) and zvetocut(e) and
       hadrtopcut(e) and jetvetocut(e);
```
(3.2)

Ungrouped strategies (UR and UG). Query (3.1) was optimized without grouping using the aggregate 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 randomized optimization (UR) and greedy optimization (UG) (see Section 2.3.2) were used. We first made extensive experiments to determine the minimal number of iterations in the randomized optimization to get a converged plan. For comparing optimization time of UR with other strategies we used the time to find the converged plan. This optimization time is regarded as the 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.

3.4.1 Experimental Setup

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

The same large query (3.1) 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 from experiment *bkg2* with high query selectivity, where only 0.008% of the events satisfy the query. Each data set contains 25000 events. As comparison, the performance was also measured for a data set from experiment *signal* with low query selectivity where 16% of the events passed the query. It contained 8623 events.

Before each experiment the main memory database is loaded only with those events participating in the experiment. It takes about 15 seconds to load one file containing 25000 events with the high selectivity. The loading of events scale linearly.

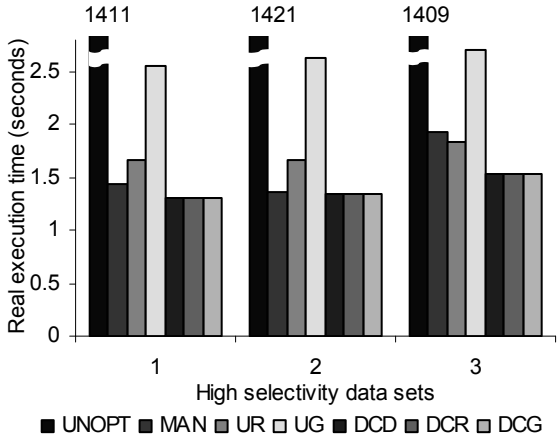

Figure 3.2. Comparing execution times for three data sets with high selectivity.

3.4.2 Experimental Results

Figure 3.2 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 3.3 measures the time to do the query optimization. All profiled grouping strategies (DCD, DCR, and DCG) are significantly faster than

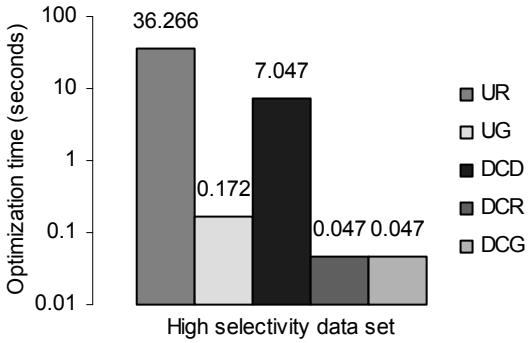

Figure 3.3. Comparing optimization time (logarithmic scale).

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 (Figure 3.2).

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 3.4. 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 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.

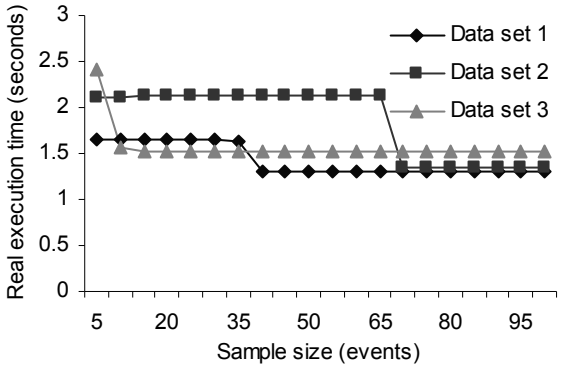

Figure 3.4. Execution performance for different sample sizes.

In Figure 3.5 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 execution emeasurements 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

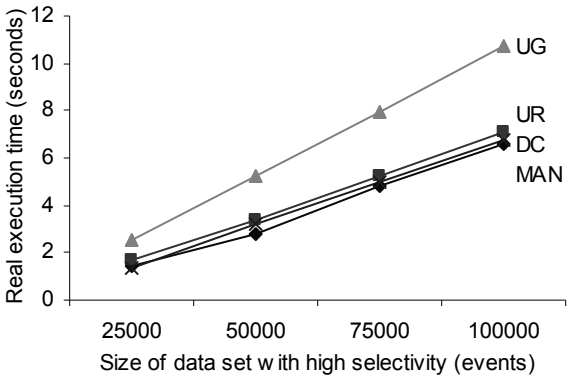

Figure 3.5. Scaling the data size with high selectivity queries.

aggregate 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 from the same experiment have the same selectivity.

Finally, Figure 3.6 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 manual 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 using manual

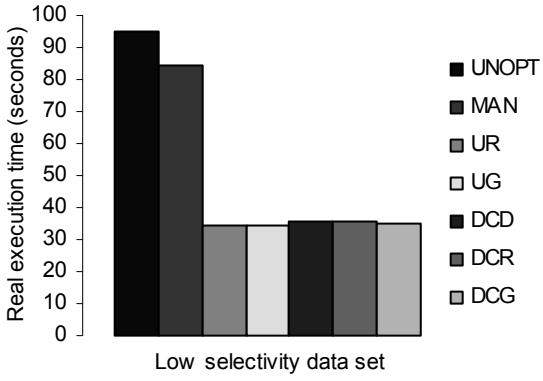

Figure 3.6. Comparing optimization strategies for low selectivity data.

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.

3.5 Summary

We implemented the ATLAS application as an extension of the main-memory DBMS Amos II. Scientific queries performing analyses are complex and naïve query processing of them is slow. Therefore, we developed a cost model for aggregate functions and other functions used in scientific queries from the ATLAS application. It was 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++ (Section 2.1.2) 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. 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, the 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 aggregate 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 aggregate cost model also has fast optimization, but the plan is around twice slower than the other plans. Still, it is shown to be substantially better than no optimization at all.

In this chapter the evaluation of the optimization approaches was performed on pre-loading events into the DBMS. This loading approach has two main drawbacks: it takes significant amount of time to load data, and the data normally cannot fit the main-memory requiring slow disk access. To

alleviate these drawbacks the next chapter investigates the streaming approach.

4. The Streaming Approach, SQISLE

A time consuming part of the loading approach used in the ALEH implementation is loading complex objects describing events from ROOT files into the indexed database of surrogate objects. For example, it takes about 15 seconds to load the ROOT file *bkg2Events_000.root*, which contains 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 queried events as surrogate objects with indexes.

Instead of preloading the data into a DBMS we therefore investigate a *streaming approach*, where data stays in their sources, e.g. ROOT files, and are streamed through the system. The system accesses complex objects from sources through a *wrapper interface* where each independent complex object is analyzed one-by-one as they are streamed. Thus the streaming approach accesses and analyzes the complex objects in a stream fashion by reading, e.g., ROOT files sequentially without populating the database and therefore the streaming approach requires limited memory and should be efficient. We implemented a DSMS called *SQISLE (Scientific Queries over Independent Streamed Large Events)* as a stream extension of Amos II [79] with facilities for processing streams of scientific events with complex structures as required for applications selecting events satisfying a number of complex conditions.

The architecture of SQISLE is illustrated by Figure 4.1, where the arrows show the data flow during query execution. A scientist specifies the analysis as a query over a stream of events from *event sources* processed by SQISLE through a wrapper interface. The scientists write their analysis queries in terms of a high level *application schema* (*App. schema*), such as the particle schema (Figure 2.3), that defines events and objects of different types

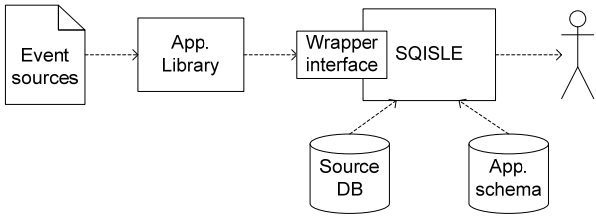

Figure 4.1. Architecture of SQISLE with data flow.

derived from events emitted by the wrapper interface. The *source database* (*Source DB*) contains meta-data about stream sources. It is accessed in queries to locate sources containing data for the analyses. The wrapper interface is defined in terms of an *application data management library* (*App. Library*), e.g., ROOT.

In the loading approach sources were specified by the names of files loaded into the database. Therefore, source specifications could be omitted in user queries over preloaded events. In contrast, in the streaming approach data are not preloaded into the database. Thus analysis queries in SQISLE always include the specifications of stream sources.

We made a streamed implementation of ALEH using SQISLE, called *SALEH* (*Streamed ALEH*). The SALEH implementation provides the same particle schema as in the loading approach with ALEH (Figure 2.3), while queries are slightly different, since they must specify also the ROOT files to access as sources.

For example, a SALEH query formulating the *Six Cuts Analysis* is an extension of the corresponding ALEH query (3.1) with specification of the source files:

```
1:  select e
2:  from Event e, EventFile f
3:  where  name(experiment(f)) = "bkg2" and
4:         fileid(f) < 15 and                                    (4.1)
5:         e = saleh_events(filename(f)) and
6:         hadrtopcut(e) and jetvetocut(e) and
7:         misseecuts(e) and zvetocut(e) and
8:         threeleptoncut(e) and leptoncuts(e);
```

The query selects the events satisfying all cuts constituting the *Six Cuts Analysis*. Each cut is a complex condition on properties of event *e* involving joins, aggregate functions, and complex numerical computations. 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 *saleh_events* calls the wrapper interface to read events from the selected ROOT files. The rest of the query specifies the *Six Cuts Analysis* as in the loading approach.

As with the loading approach, naïve execution of analysis queries in SQISLE without query optimization strategies is slow. Therefore the optimization strategies from the loading approach are utilized here too. However, since events are not stored in a database the cost-based query optimizer has no information of data statistics. Therefore *runtime query optimization* mechanisms are implemented that dynamically, at query execution time, collect statistics on a subset of a stream. The query is automatically reoptimized when enough statistics is collected. Once

reoptimized, the query execution is immediately continued using the reoptimized query execution plan for the rest of the stream.

The query processing is further improved by *query transformations*, use of *materialized views*, and *compile time evaluation* of query fragments. The query transformations reduce the number of predicates in queries. Materialized views are executed only once per event and then materialized view results are accessed during processing the same event. Partial evaluation [59][77] 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 SALEH queries over ROOT files with different selectivities it is shown that these SQISLE query processing techniques improve performance of queries very significantly. The query performance is compared with the performance of a manually coded C++ program provided by the physicists doing the same analysis as the queries. The SALEH implementation is shown to have performance close to or better than the C++ implementation.

The rest of this chapter is organized as follows. Section 4.1 presents what is needed to implement a new application with SQISLE. Section 4.2 describes SQISLE stream objects, which represent the data streamed through the system. Section 4.3 gives an overview of the different query processing techniques we have developed in SQISLE. Section 4.4 describes query optimization using dynamic query optimization at runtime. Further improvements by query transformations and materializations are presented in Section 4.5. Section 4.6 describes the experimental setup for the performance evaluation. Section 4.7 compares the performance of SALEH queries using the different implemented approaches, including a comparison with a manually coded C++ program. The chapter is concluded with summary in Section 4.8.

4.1 Defining a SQISLE Application

Before SQISLE can be used for querying data from files for a new application the application schema and the wrapper interface are defined by a SQISLE administrator.

First, *wrapper interface functions* are implemented to access data from the sources using the application data management library. The same wrapper interface can be used for every application that stores data in the same format.

The objects representing events are constructed by the wrapper interface functions. Such *stream objects* must be efficiently streamed and processed by complex scientific queries, and SQISLE provides a special datatype, *Sobject*, for that.

For example, the SALEH application includes wrapper interface functions to access data from the ATLAS experiment stored in files managed by ROOT as collections of tuples of simple C values. The wrapper interface is called the *ROOT wrapper interface*, and described in details in Appendix E. The ROOT wrapper interface functions allocate stream objects to represent tuples read from ROOT files preserving the original structure of the data. The ROOT wrapper interface includes also functions to read meta-information about the structure of data stored in ROOT files. The ROOT wrapper interface can be used for any application accessing data stored in collections of tuples in ROOT files.

For each new application the SQISLE administrator defines an application schema in terms of the wrapper interface functions. The schema definition provides a set of types and functions used in user queries.

Since an application schema is usually different from the structure of the data emitted by the wrapper interface functions, mappings between the original structure and the application schema are defined as *transformation views*. The objects derived from the events by transformation views are also represented as stream objects.

For example in SALEH, ROOT events in a source are mapped to different kinds of particle objects by transformation views, described in details in Appendix F. The transformation views map the representation of particle data on each event to the different representation of particle objects according to the particle schema.

Meta-information about data sources is stored in the source database. This meta-information is used to select sources of stream objects.

For example, SALEH stores meta-properties of files (Figure 4.2) associated withthe particle schema. The files are described by the attribute *Filename*, which is set to the name of a ROOT file along a path to the file. In addition to the name, the path, and the size of the ROOT file a *file identifier* is stored for each event file. The file identifier is used by the ATLAS software to partition an event set over several files. Since event files are produced by experiments, they are related to the type *Experiment*, which describes[1] each experiment producing event files.

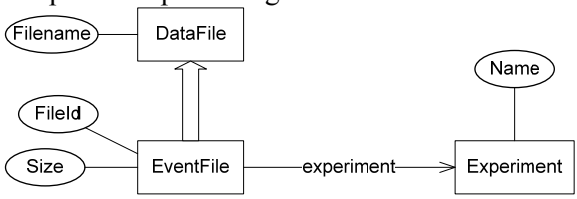

Figure 4.2. Schema of the Source Database in SALEH.

[1] Currently the description contains only the name of the experiment.

The particle schema is defined in SALEH in terms of the ROOT wrapper interface functions, the transformation views, the stream objects, and the source database. The definition is presented in Appendix G.

An example of a query in SALEH is (4.1). The query is defined in terms of cuts analogous the cuts defined in the loading approach. To support scientific applications more elegantly and efficiently, SQISLE includes some new utility functions compared to the loading approach, as presented in Appendix H. The implementations of cuts in SALEH using these functions are presented in Appendix I.

4.2 Stream Objects

Stream objects are implemented in SQISLE by a system data type *Sobject* and the types of the applications are all subtypes of *Sobject*.

For example, in SALEH the particle schema is defined in terms of the type hierarchy in Figure 4.3.

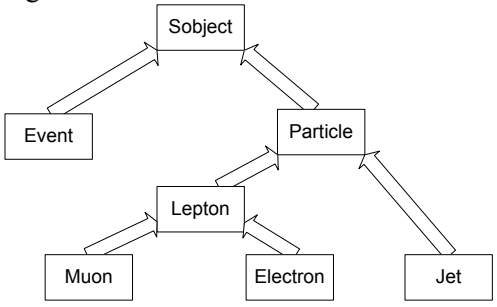

Figure 4.3. Type hierarchy in SALEH.

Events are emitted by wrapper interface functions, and objects derived from these events are materialized as stream objects. Each stream object belongs to a type defined by the application schema. This enables specification of queries in terms of stream objects analogous to the queries over surrogate objects stored in the database in the loading approach. The difference between stream objects and surrogate objects is that the stream objects are automatically deallocated by a garbage collector rather then explicitly removed, and the extents of stream object types need not be maintained by the system. By contrast, surrogate objects are more heavy-weight, have system-maintained extents, and are allocated and deallocated explicitly by the user.

Since the extents of stream object are not maintained, a stream object requires an explicit key consisting of its type, source, and identifier within a source. This allows duplicate stream objects to be eliminated and the same stream objects represented by different instances of *Sobject* to be considered equal.

Implementation details of stream objects are presented in Appendix D.

4.3 Query Processing in SQISLE

The query processing steps in SQISLE are illustrated by Figure 4.4. The *query pre-processor* expands views and applies a number of rewrite rules on the user query. The cost-based *query optimizer* produces an execution plan 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*) to access the *event sources*.

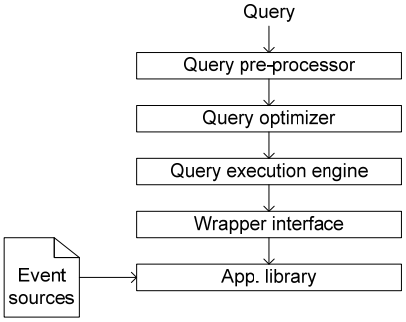

Figure 4.4. Query processing steps in SQISLE.

A SQISLE query consists of fragments accessing sources, implementing transformation views, and doing the analysis. Figure 4.5 presents the general structure of a query execution plan with the data dependencies between the different kinds of operators grouped in three blocks. The *wrapper argument*

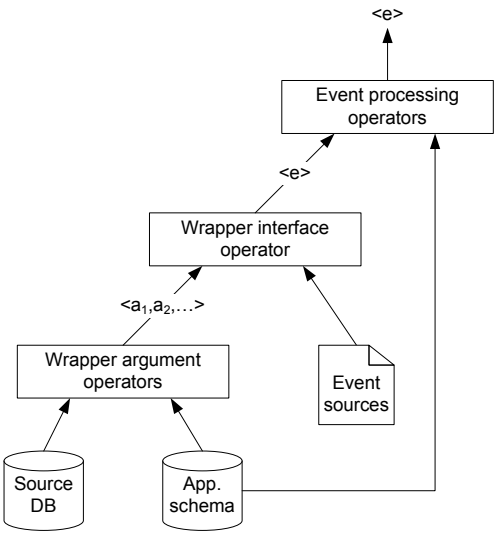

Figure 4.5. Structure of naïve query plan in SQISLE.

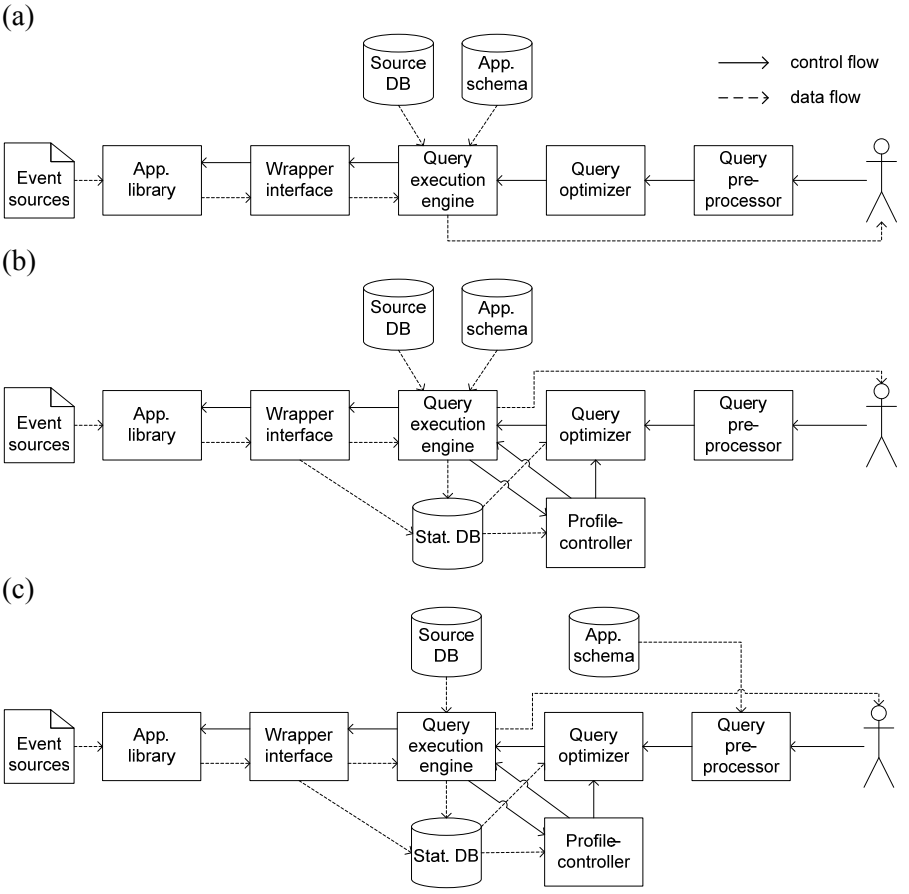

Figure 4.6. Query processing in SQISLE: (a) naïve query processing; (b) dynamic query processing with aggregate cost model; (c) rewritten query processing using all proposed rewrites in addition to runtime query optimization.

operators are placed first in the plan. They access both the source database (*Source DB*) and the application schema meta-data (*App. schema*) to bind parameters a_1, a_2 for a *wrapper interface operator*. The wrapper interface operator accesses the event sources and creates stream objects representing source events *e*. The query execution engine then executes the *event processing operators* doing both event transformations and analyses. Those event processing operators that perform transformations access the application schema meta-data to map event objects to other objects in the application schema. Finally, the result of the query execution, e.g., those events *e* that passed all cuts, is streamed to the user.

Figure 4.6 illustrates the different query processing alternatives investigated for SQISLE. *Naïve query processing* (Figure 4.6(a)) optimizes stream queries using a simple cost model without runtime query optimization. The dashed arrows indicate how data flows when answering a

query. The execution plans have the structure as in Figure 4.5 containing operators that access the wrapper interface, the source database, and the application schema meta-data.

Naïve query execution performs badly because the static cost model does not contain any statistics about the contents of the streams. Figure 4.6(b) illustrates *dynamic query processing* using runtime query optimization to collect data statistics at runtime. The optimizer uses the data statistics and the aggregate cost model (Table 3.2) to reoptimize the query while it is running. Similar to the loading approach profiled grouping is used to fragment the query into *groups* joined only of the event variable *e*. Each group is profiled and their join-order is optimized using the statistics on the groups collected at runtime.

When runtime profiling is enabled, for each emitted event object the wrapper interface operator collects statistics about the created event object and stores it in the *statistics database* (*Stat. DB*). Next the *profile-controller* operator is executed to encapsulate the event processing operators. It first calls the execution engine to execute the event processing operators. Then the profile-controller checks if enough statistics has been collected. If so, profiling is disabled and the query fragment is reoptimized to obtain a more efficient query subplan. After the profile-controller is ready the evaluation is immediately continued with the next event using the new query subplan. The query optimizer uses collected statistics from the statistics database for the query reoptimization.

Dynamic query processing for selective queries produces very efficient execution plans, even faster than a manually coded unoptimized C++ program. However, performance of queries with low selectivities is still substantially slower than C++. The reason is that query optimization will not significantly improve performance of queries with low selectivities, where most operators are always executed for each event. Figure 4.6(c) illustrates *rewritten query processing* in SQISLE where a number of query transformation techniques described in this chapter are used, including view materializations, simplifications, and compile time evaluation. On the architecture level, the difference from Figure 4.6(b) is that with rewritten query processing the application schema meta-data is accessed already during the pre-processing phase by evaluating at compile time all predicates accessing the application schema meta-data. In the rewritten query, these meta-data predicates are replaced with their results by the query preprocessor. Therefore there is no access to the application schema meta-data from the query execution plan. This makes the rewritten query simpler, which improves the performance of both query optimization and execution.

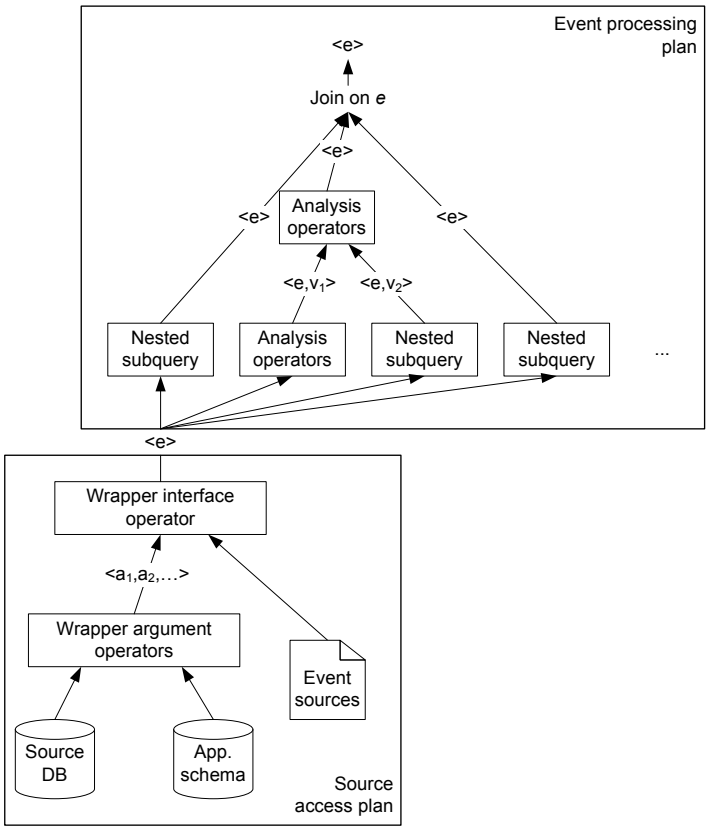

Figure 4.7. Detailed structure of a query plan.

4.4 Optimization of Stream Queries

As illustrated by query (4.1) scientific analysis queries are often large and complex as each event processing filter is a complex view. A detailed structure of query plans analyzing an event stream is illustrated in Figure 4.7. The query plans can be split into two subplans: the *source access plan* and the *event processing plan*. The source access plan contains a *wrapper interface operator* and *wrapper argument operators*. The number of operators in the source access plan for query (4.1) is ten. The source access plan produces a stream of events, which are analyzed by the event processing plan. The event processing plan contains many operators and several calls to *nested subqueries*. A general structure of the execution plans for the nested subqueries is presented in Figure 4.8. They perform the analyses in terms of objects derived by the transformation views.

The number of operators in the event processing plan for query (4.1) is 22, and 8 of them are nested subqueries. The number of operators in the

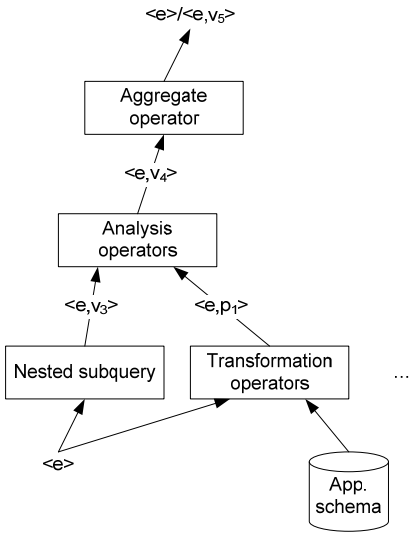

Figure 4.8. Structure of a nested subquery plan.

nested subquery plans is between 9 and 59, including *transformation operators* performing transformation views and *analysis operators* implementing the selections. The nested subqueries may also contain calls to further nested subqueries having the same structure as in Figure 4.8.

There are many possible orders of the operators in the event processing plan. Thus query optimization is difficult, and the query plans obtained with naïve query processing perform very slowly. To improve query processing, the runtime query optimization approach (Figure 4.6(b)) collects data statistics for the query optimizer and reoptimizes the query at runtime using the collected statistics.

Runtime query optimization was investigated together with three profiling strategies:

1. *Event statistics profiling* maintains statistics on the sizes of event attribute vectors as the events are read. The collected statistics is used in operator cost models for optimizing the query.
2. *Group statistics profiling* decomposes the queries into fragments, called *groups*, joined only on the event variable and then maintains runtime statistics of executing each group. The collected statistics per group is used for optimizing the join order between the groups.
3. *Two-phase statistics profiling* combines the two strategies above by in a first phase collecting statistics of event attribute vector sizes to optimize the group definitions, and in a second phase switching to group statistics profiling for ordering the groups.

These strategies are evaluated by the SALEH application. It is shown that the performance of SALEH queries with dynamic query processing is significantly improved compare to the naïve query processing with a static

cost model. Furthermore, the performance of SALEH with the different variants of runtime query optimization is compared with the loading approach (ALEH) and the total processing time for SALEH is shown to be significantly faster for the different queries.

4.4.1 The Profile-Controller Operator

The goal of the profile-controller operator is to monitor statistics collected during query execution in order to dynamically reoptimize a query fragment according to some runtime query optimization strategy, and then switch into another runtime query optimization strategy or non-profiled execution. Once the switch is made into non-profiled execution there is no profiling overhead.

To enable runtime query optimization, the query pre-processor modifies the view expanded query to include the profile-controller operator. The query is thereby split into the *source access query fragment* that generates the events and the *processing query fragment* that filters the events by complex conditions. To optimize these complex conditions at runtime, the event processing fragment is controlled by the profile-controller operator.

The source access query fragment contains calls to a wrapper interface function and functions that compute parameters of the wrapper interface function. The wrapper interface function has a single result variable the *event variable* holding the currently processed event. The source access query fragment is constructed by joining the wrapper interface function with all functions having other variables in common except the event variable. The rest of the query forms the processing query fragment that needs to be optimized carefully, since it is defined as a complex condition over each event.

For example, the source access query fragment for query (4.1) will be:

```
name(experiment(f)) = "bkg2" and
fileid(f) < 15 and
e = saleh_events(path(f))
```

The source access fragment generates each event *e*, which is the output variable from the wrapper interface function called inside the derived function *saleh_events*. The processing query fragment will be:

```
hadrtopcut(e) and jetvetocut(e) and
misseecuts(e) and zvetocut(e)and
threeleptoncut(e) and leptoncuts(e)
```

The predicates of the event processing fragments are defined as complex conditions over each event *e*.

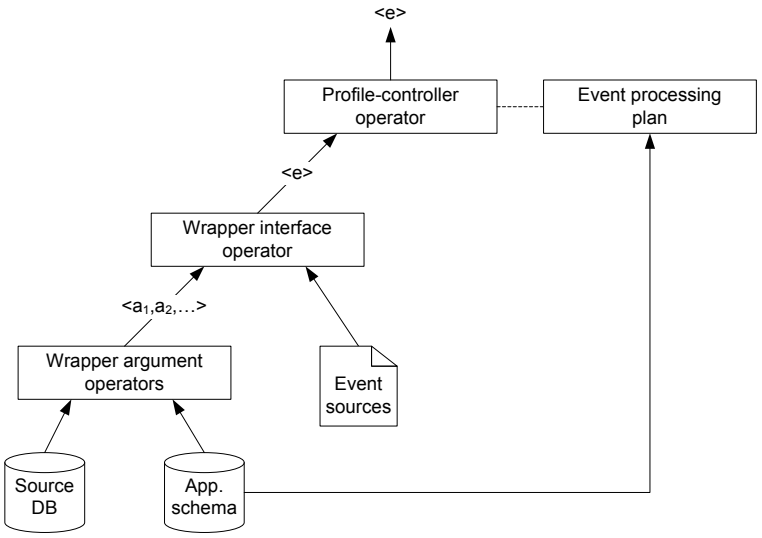

Figure 4.9. Structure of query rewritten with profile-controller.

After the query is split into the two fragments, calls to the profile-controller operator are inserted to encapsulate the event processing fragment as a subquery. The structure of a query plan with the profile-controller is illustrated in Figure 4.9. The profile-controller operator takes an event variable *e* generated by the wrapper interface operator as its input and applies the event processing plan on this event. It returns the result of the event processing plan. The structure of an event processing plan is presented in Figure 4.10.

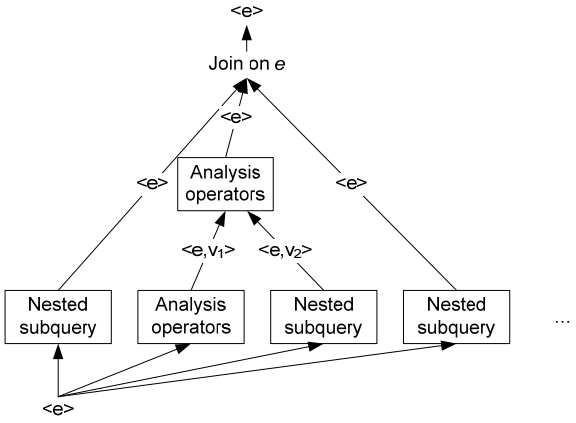

Figure 4.10. Structure of an event processing plan.

The profile-controller performs the following operations for each event:

1. It executes the event processing plan for the event.

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 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 executed in item one is always emitted as result of the profile-controller operator.

4.4.2 Event Statistics Profiling

With event statistics profiling enabled statistics on event attribute sizes is collected when each new event is constructed by the wrapper interface operator. Statistics to maintain means and variances of each event is stored for each event attribute vector in an internal table.

The event statistics profiling assumes that data statistics over the stream is stable so that the estimated average of the statistics collected in the beginning of the stream is expected to be close to the mean of the entire data stream. The switch condition monitor here maintains the statistics to check whether the following confidence interval is satisfied for every tenth read event:

$$\Pr(-\delta \cdot \bar{x} \le \bar{x} - \mu \le \delta \cdot \bar{x}) = 1 - \alpha \qquad (4.2)$$

This formula checks if an estimate $\bar{x}$ of the mean size of an attribute value (μ) is close enough to μ with probability *1-α*. The closeness is defined by δ. The estimate of the mean size $\bar{x}$ is calculated by $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, where x_i is the size of an attribute value (e.g., *Kfele*) for the i^{th} event, and n is the number of events read so far. The confidence interval (4.2) is checked by following inequality:

$$z_{\alpha/2} \cdot S_E \le \delta \cdot \bar{x} \qquad (4.3)$$

Where S_E is an estimate of $\sigma/\sqrt{n}$ and calculated by $S_E = \sqrt{\frac{1}{n^2}\sum_{i=1}^{n} x_i^2 - \frac{\bar{x}^2}{n}}$. The inequality (4.3) is obtained by normalizing the confidence interval and applying the central limit theorem [67].

The switch condition monitor the test condition (4.3), for which α and δ are provided as tuning parameters, and if the condition is satisfied for every event attribute, the switch procedure is called. It reoptimizes the processing

query fragment and disables collecting statistics and profiling. After this, when the wrapper interface operator constructs a new event, it does not collect statistics any more. For new events the profile-controller executes only the event processing plan and does not call the switch condition monitor or the switch procedure.

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

4.4.3 Group Statistics Profiling

With *group statistics profiling*, first, a stream fragmenting algorithm (Appendix J) is applied to a query. The algorithm splits the query into source access and processing query fragments and decomposes the processing query fragment into groups. The groups have only the event variable *e* in common and thus the groups are equi-joined only on *e*. The event 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 an event as arguments and returns the result of applying the subplan on the event. If profiling is enabled, it measures execution time and fanout of the subplan.

Figure 4.11 illustrates the structure of an event processing plan after grouping. In the figure three monitored group subplans are formed and noted by *Group plan 1*, *Group plan 2*, and *Group plan 3*. The query optimizer orders the executions of the monitored subplans based on available statistics on the groups. An internal table keeps track of the groups and their statistics. When a query is initially optimized, before any query execution, no group statistics have been collected and therefore the first ordering of the groups will be based on heuristic default estimates of data sizes and the aggregate cost model.

The profile-controller operator encapsulates the entire event processing plan containing the joined groups. It invokes the event processing plan at runtime. If some join fails, the entire event processing plan fails. Thus, to answer the query the event processing plan only executes those first group subplans up to the first subplan that fails. No group subplans joined after the failed one are executed. However, statistics need to be collected for all groups, even those not executed by the event processing plan. Thus, if profiling is enabled, the switch condition monitor executes those groups that were not executed by the event processing plan in order to collect statistics on real execution time and fanout by their group monitor operators. The switch condition monitor checks the current group statistics after invoking the remaining group subplans. Rather than testing for stable statistics as with event statistics profiling, the check here determines if the new statistics does

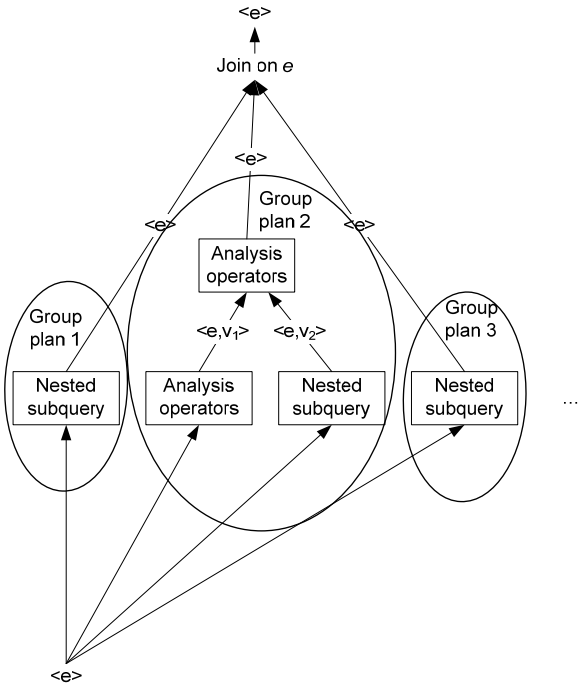

Figure 4.11. Structure of the event processing plan with formed groups.

not affect the optimized join order of the groups. This is done by greedily reordering the groups for every new event based on the measured estimates of the group costs and fanouts. To minimize overhead the event processing plan is reoptimized once per event, there is no dynamic reordering per operator as with Eddies [4]. The profiling is disabled if the order of the groups in the new event processing plan is the same as earlier for a number of events in a row, called the *stable reoptimization interval* (*SI*), which is provided as a tuning parameter. Together with disabling the profiling the group monitor operators are removed from the final event processing plan by substituting them with their group subplans. This removes overhead of invoking the group monitor operators.

4.4.4 Two-Phase Statistics Profiling

As with group statistics profiling, with the *two-phase statistics profiling* queries are first fragmented into groups before executing them. Initially during query execution event statistics profiling is enabled. When the profiling condition (4.3) is satisfied, the entire query is optimized, including the group fragments, and event statistics profiling is disabled. Then the switch condition monitor and switch procedure are changed to perform group statistics profiling and 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 initially collected event statistics. With group statistics profiling alone, where the events are not monitored, the groups themselves are optimized based on heuristic estimates of costs and fanouts. Two-phase statistics profiling could potentially be faster since the optimization of groups is based on monitored statistics rather than heuristics.

4.5 Query Rewrite Strategies

A comparison of query performance with runtime query optimization with a manually code C++ program shows that the query plans of selective queries may perform better than a C++ implementation, while queries with low selectivities are still around 28 times slower.

In order to improve the performance of queries with low selectivities, 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 event. To remove this bottleneck, the use of query transformation rules to simplify and speed up the transformation views were investigated. One kind of rewriting is based on observing that in SALEH the transformations can be regarded as a two-dimensional matrix transposition. Different variants of operators for the transpositions were implemented and evaluated. The most efficient matrix transpose operator creates new particle stream objects as the result of the transposition and caches them as an attribute on the event object. This strategy is called *transformation view materialization*. It improves performance of queries with low selectivities about 1.5 – 2.5 times compared with only runtime query optimization, which is still around 13 times slower than the C++ program.

Queries are further simplified in SALEH by removing unnecessary vector constructions appearing 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 rewritings* improve performance of queries with low selectivity with factor 1.5 – 2, i.e. around 7 times slower than C++.

In addition *computational view materialization* is implemented to improve query performance. Computational views perform complex numerical calculations for computing properties of derived stream objects used in analysis queries, e.g. in cut definitions. Their materializations pay off when a query does the same complex numerical calculations several times. The materialization of the computational views improves the queries with low selectivities with at least another 32% in SALEH, i.e. about six times

faster than only using runtime query optimization, but still around 5 times slower than the C++ program.

Finally, the performance of queries is further improved by partial evaluation [59][77], which is a general technique to evaluate predicates at query compilation time and replace predicates with computed values. The partial evaluation is used to remove accesses to application schema meta-data, which simplify the queries. The partial evaluation improves performance of queries with low selectivities an additional 20%, i.e. seven times faster than only using runtime query optimization and about 4 times slower than C++.

Notice that the execution plan is interpreted in SQISLE. Further performance improvements can be made by making an execution plan compiler, which is expected to make the plan as fast as C++ also for non-selective queries.

4.5.1 Rewritten and Materialized Transformation Views

In SALEH queries the cuts are defined in terms of particle properties according to the particle schema. Thus every time events are analyzed the transformation views deriving particles are used. Therefore, it is investigated how performance of scientific queries in SQISLE can be improved by applying a number of rewriting rules to simplify transformation views and to materialize the transformation views.

The number of operators performing transformations is first reduced by defining rewriting rules that transform a conjunction of ObjectLog predicates (Section 2.3.2) before query optimization. First of all, the transformation views defining particles in the particle schema can be seen as matrix transposition of the event attribute vectors. A *matrix transposition rewriting rule* recognizes query fragments where new vectors are constructed by transposing original vectors. Such query fragments are replaced with a *matrix transposition function*. The matrix transposition function takes as argument a matrix of size *m*x*n*, which is represented as vector of size *m* containing *m* vectors of size *n*. The function returns as result the transposition as a new matrix of size *n*x*m*, which is represented as vector of size *n* containing *n* vectors of size *m*.

For example, values of electron properties can be represented by a matrix, where rows contain values for each electron and columns contain values for each event attribute. Originally values of electron properties are stored in the attribute vectors of an event object, represented by columns in Figure 4.12(a). The indexes of the vectors identify the particles. The result of the electron transformation view is a set of electron stream objects where the value of each electron attribute corresponds to an element value of an attribute vector in the event. Each electron is represented as a row as in Figure 4.12(b).

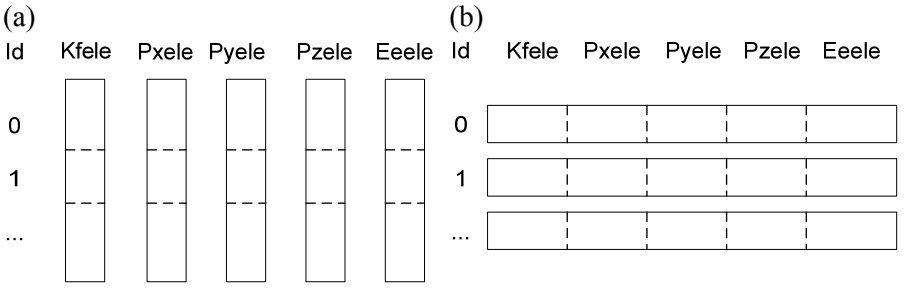

Figure 4.12. Transformation view for electrons. (a) Electron properties originally stored in event objects as attribute vectors; (b) After transposition the values are stored as attributes per electron object.

Thus the transformation view can be seen as a matrix transposition, and the matrix transposition rewriting rule defines it. The transformation view *new_electrons* has the following definition (Appendix F) in terms of basic functions after view expansion:

V_1 = get_slot(e,1) AND	Get event attribute vector *Kfele* of event object stored in position one
E_2 = V_1[e_i] AND	Iterate over all values e_i of attribute vector *Kfele*
V_2 = get_slot(e,2) AND	Get event attribute vector *Pxele*
E_2 = V_2[e_i] AND	Get each value of attribute vector *Pxele*
…	etc.
V = {E_1,E_2,E_3,E_4,E_5} AND	Construct vector of attribute values for each electron
T = typenamed("ELECTRON") AND	Obtain type object
el = new_sobject(T,e,e_i,V)	Create a steam object for each electron

The view *new_electrons* first constructs a vector in variable *V* and calls the basic function *new_sobject* to construct the new stream object. The event attribute vectors *Kfele*, *Pxele*, *Pyele*, *Pzele*, and *Eeele* are assigned to variables V_1, V_2, V_3, V_4, and V_5.

After rewrite the definition of *new_electrons* to use matrix transposition the query defining the view becomes:

V_1 = get_slot(e,1) AND	Get event attribute vector *Kfele*
V_2 = get_slot(e,2) AND	Get event attribute vector *Pxele*
…	
V = {V_1,V_2,V_3,V_4,V_5} AND	Form an input vector of the event attribute vectors
V_T = transpose(V) AND	Transpose the vector of the event attribute vectors
T = typenamed("ELECTRON") AND	Obtain type object
A = V_T[e_i] AND	Iterate over all elements of the transposed result vector
el = new_sobject(T,e,e_i,A)	Create a stream object for each electron

To enable matrix transposition, a vector V of the event attribute vectors is formed by the vector construction operator noted by "*{}*", and then the matrix transposition function is applied on the constructed vector. After the call to the matrix transposition function, the elements (rows in Figure 4.12) of the transposed result vector V_T are accessed to create each stream object *el* representing the electrons of the event.

The rewritten definition is smaller and it does not access individual elements of the event attribute vectors, as in the original definition. Before the rewriting the number of operators to call in the execution plan is $1+m\cdot(2\cdot(i-1)+4)$, where i is the number accessed attributes (here 5) and m is the size of the attribute vectors, i.e., the number of electrons. After the rewriting the number of operator calls is $i+3+2\cdot m$, thus the rewritten query scales better for in the number of called operators when many event attributes are accessed and there are many particles in each event, There is thus less overhead during the interpretation of the execution plan. The number of predicates in the view definition is reduced from $2\cdot i+3$ to $i+5$, which makes query optimization faster.

The new definition can be further rewritten to reduce the number of functions in the rewritten expanded view. Since the attribute vectors are originally stored in a stream object *e* and the result of the transformation view is a stream object *el*, a specialized version of the matrix transposition *sobject_transpose* is implemented to operate directly on stream objects rather than first accessing event attributes. The stream object transposition function takes as input a stream object *e*, a vector of slots containing attribute vector positions for the accessed attributes, and the type of the result stream object. The result of *sobject_transpose* is a vector of new stream objects, representing, e.g., a set of electrons.

The definition of *new_electrons* in terms of *sobject_transpose* is:

`I = {1,2,3,4,5} AND`	Form vector of indexes
`T = typenamed("ELECTRON") AND`	Obtain type object
`V = sobject_transpose(e,I,T) AND`	Transpose the event attribute vectors of *e* specified in *I* and create stream objects for all electrons
`el = V[e_i]`	Iterate over all electrons in the vector

In this definition the number of called operators is further reduced to $3+m$ calls and the total number of predicates to *4*. Therefore query optimization is much faster since the predicate to optimize is of fixed size.

To avoid recomputing the transformation views, materialization of transformation view results is implemented by a stream object transposition function *mat_sobject_transpose.* It stores transposed vectors of stream objects directly in the event object. This materialization is made as soon as

possible in the query plan. Therefore, it is guaranteed that all objects derived from an event by transformation views are materialized before they are analyzed. During analyses the materialized objects are accessed directly from the event objects.

The original definitions of the analysis queries and the transformation views were manually rewritten to investigate impact on query performance of different rewriting rules and materialization of the transformation views.

A structure of a subplan containing the *mat_sobject_transpose* operator is shown in Figure 4.13. In the subplan the first two operators bind arguments for the *mat_sobject_transpose* operator. As for *sobject_transpose*, *I* is bound to the vector of attribute vector positions, and *T* is bound to the type *Electron*. Then the *mat_sobject_transpose* operator creates the new stream objects representing electrons and stores them in the event stream object *e* at position 17. The last operator, *get_slot_bag*, accesses the event stream object for the stored electrons and returns the electrons into *el* one by one.

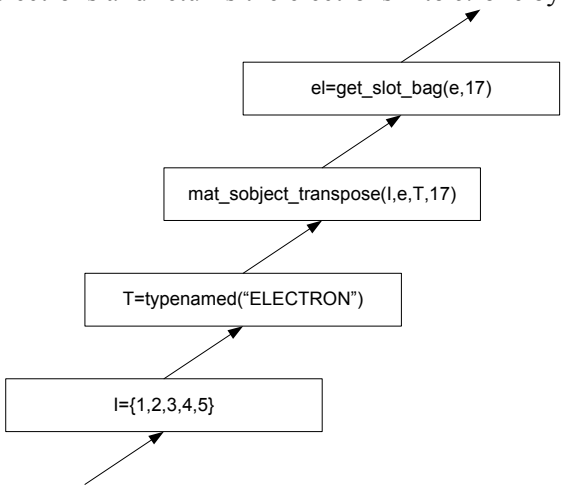

Figure 4.13. Structure of a subplan with materialization of a transformation view.

4.5.2 Materialized Computational Views

In analysis queries different computational views, defined as derived functions, are often applied more than once on the same event. For example, a derived function defining isolated leptons is applied several times over the same event in different cuts in query (4.1) and the function defining *Ok jets* is applied several times. To investigate if materialization of such computational views can improve performance of less selective queries a *materialize operator* was implemented to perform lazy materialization of derived functions over stream objects, e.g., events. The materialize operator encapsulates a call to a derived function for a stream object and materializes its results in the stream object.

The query is rewritten to call the materialize operator for specific derived functions. When the materialize operator is called for a function and a stream object, it first checks whether a materialized result is already stored in the stream object. If so, the materialized result is emitted. If the result is not already materialized, the function is applied on the stream object and the result of the execution is materialized and emitted.

Materialization of different derived functions used in SALEH queries was investigated by manually rewriting the queries to call the materialize operator. The measurements demonstrate that materialization of the view *OkJets* gives 32% improvement in execution performance of the queries with low selectivities on top of pervious optimizations, while materializations of other functions did not give improvements in performance.

An approximate cost model for the materialize operator is defined as the cost of executing the view definition. This is a conservative estimate for the case when the materialize operator is called the first time for a given argument. Later, when a call to the materialize operator accesses a materialized value, this gives too high cost, while the fanout is correct. Future work would implement a cost model that estimates cost of the materialize operator correctly by recognizing if the call to the materialized view is first in a query plan or if it is previously called in the plan.

4.5.3 Vector Rewritings

Scientific analysis queries are large and complex and are defined using many views. This can lead to unnecessary vector constructions and vector accesses that can be avoided. In particular, rewritings are investigated to remove unnecessary vector constructions where the constructed vectors are not needed, since only the vector elements are accessed. For example, the following query fragment can be replaced with $f(x_2)$ if v is not used anywhere else:

```
v={x1,x2,x3) AND
f(v[2])
```

Four rewriting rules are proposed to rewrite queries with vector constructions and accesses after view expansion:

1. The *element replacement rule* applies when both the vector construction and an access to an element of the vector are presented in the query. The rewriting rule replaces the vector accesses with a corresponding variable used when constructing the vector.
   Thus

a query fragment	is transformed into
`v={…,xi,…} AND` `f(v[i])`	`v={…,xi,…} AND` `f(xi)`

   where *i* is an integer constant.

2. The *argument spreading rule* applies when a vector constructed in the query is used as the argument of a function that has an equivalent 'spread' definition with separate arguments for each element. The original function call is then replaced with a call to the spread function applied on of the variables used to construct the vector. This rewriting rule requires that the system either automatically builds the spread functions for the original functions or maintains pairs of equivalent functions, where one function is applied on a vector and another function is applied on separate arguments.

   Thus given a function *f(v)*, where *v* is a vector of size *n*, has equivalent spread definition $f'(x_1,\ldots,x_n)$ in terms of vector elements, the argument spreading rule makes the following rewrite:

a query fragment	is transformed into
$v=\{x_1,\ldots,x_n\}$ AND a=f(v)	$v=\{x_1,\ldots,x_n\}$ AND $a=f'(x_1,\ldots,x_n)$

3. The *result spreading rule* applies when a function returns a vector and has an equivalent definition with spread results for each element. The original function is replaced with the spread function and the result of the spread function is assigned to spread variables. Then the vector is formed by the spread variables. This rewriting rule requires that the system either builds equivalent functions with spread results for original functions or maintains such pairs of equivalent functions. For example, if an original function is an aggregate function over a nested subquery, the equivalent function with spread result can be automatically constructed.

   Given that a function *f(a)*, which returns a vector, has an equivalent spread definition *f'(a)* that returns vector elements, the result spreading rule makes the following rewrite:

a query fragment	is transformed into
v=f(a)	$\{x_1,\ldots,x_n\}=f'(a)$ AND $v=\{x_1,\ldots,x_n\}$

4. The *constructor removal rule* removes a vector construction if the constructed vector is not used anywhere else.

   For example, if constructed vector *v* is not used anywhere as in the following example, the vector construction is removed:

a query fragment	is transformed into
$v=\{x_1,\ldots,x_n\}$ AND $f(x_2)$	$f(x_2)$

First the result spreading rule is applied to a query. Then the element replacement rule and the argument spreading rule are applied to remove vector accesses. Finally, the constructor removal rule removes all constructions of vectors that are not used in the queries.

To measure impact of the proposed rewriting rules the scientific queries are manually rewritten. Experiments demonstrate that the vector rewritings improve performance of queries with low selectivities about twice.

4.5.4 Applying Partial Evaluation

A rewriting technique for reducing the size of a query is partial evaluation [59][77]. Partial evaluation evaluates some predicates at query compilation time and replaces them with the evaluated result if possible. It is applied to queries by the query pre-processor together with other rewriting rules. Using partial evaluation, the size of a query can be reduced before query optimization and execution. In SQISLE partial evaluation is used to evaluate at compile time all predicates accessing meta-data about the application schema.

For example, several of the views in SALEH call the function *typenamed("Event")*. This call is replaced by partial evaluation with the object representing the type named Event, e.g. *#[OID 1242 "EVENT"]*.

Experiments demonstrate that the partial evaluation improve performance on queries with low selectivities by 20%.

4.6 Performance Measurements

Performance experiments were done for scientific analyses expressed as queries to SALEH for the ATLAS application. The experiments were performed on a cluster node having 2.8 GHz Intel P4 CPU with 2GB RAM, and running Linux OS.

The SALEH queries implement the *Four Cuts Analysis* (Example 2.1) and the *Six Cuts Analysis* (Example 2.2). The *Four Cuts Analysis* is defined in terms of particle properties by four cuts. The *Six Cuts Analysis* is more complex and is defined in terms of both event properties (attributes *PxMiss* and *PyMiss*) and particle properties by six cuts.

The performance was evaluated for the different query processing strategies and queries implementing both scientific analyses. The performance of the C++ implementation was demonstrated only for the *Six Cuts Analysis*, since this implementation was the only one provided by the physicists.

The query performance was measured by evaluating SALEH queries over events from two different experiments. The events were stored in ROOT files accessed as streams. The first experiment *bkg2* simulates background

events, which unlikely produce the Higgs bosons, so the selectivities of both kinds of analysis queries are very high (<0.2%). The other experiment *signal* simulates signal events that are likely to produce the Higgs bosons, and the selectivities of the two kinds of queries are low (16% and 58%).

Events from the *bkg2* experiment are stored in 41 ROOT files, where each file contains 25000 events, i.e., a stream with 1025000 events in total. Events from the *signal* experiment are stored in a single file, which contains 8623 events.

The evaluations were performed by scaling the size of the event streams by reading subsets of these streams. For this four queries were defined for both scientific analyses and both experiments as a number of functions where a parameter is used to specify the number of events to read and analyze, i.e. the stream size.

The query implementing the *Six Cuts Analysis* (Example 2.2) over events from experiment *bkg2* is defined as a derived function *bkgsixcuts*. The stream size is specified as the number of files to analyze:

```
create function bkgsixcuts(Integer nrFiles) -> Event e
select e
from EventFile f
where  name(experiment(f)) = "bkg2" and
       fileid(f) < nrFiles and
       e = saleh_events(filename(f)) and
       hadrtopcut(e) and jetvetocut(e) and
       misseecuts(e) and zvetocut(e) and
       threeleptoncut(e) and leptoncuts(e);
```

The query implementing the *Six Cuts Analysis* over events from experiment *signal* is defined by a derived function *signalsixcuts*, which processes only the single file produced in experiment *signal* having identity zero. The upper limit on the event identity to read from the file is specified as parameter:

```
create function signalsixcuts(Integer idEvent) ->
  Event e
select e
from EventFile f
where  name(experiment(f)) = "signal" and
       fileid(f) = 0 and
       e = saleh_events(filename(f),0,idEvent) and
       hadrtopcut(e) and jetvetocut(e) and
       misseecuts(e) and zvetocut(e) and
       threeleptoncut(e) and leptoncuts(e);
```

Analogously, the query implementing the *Four Cuts Analysis* (Example 2.1) over events from experiment *bkg2* is defined as:

```
create function bkgfourcuts(Integer nrFiles) -> Event e
select e
from EventFile f
where  name(experiment(f)) = "bkg2" and
       fileid(f) < nrFiles and
       e = saleh_events(filename(f)) and
       jetCut(e) and topcut(e) and
       threeLeptonCut(e) and twoLeptonCut(e);
```

Finally, the query that implements the *Four Cuts Analysis* over events from experiment *signal* is defined analogous to the function *signalsixcuts*:

```
create function signalfourcuts(Integer idEvent) ->
  Event e
select e
from EventFile f
where  name(experiment(f)) = "signal" and
       fileid(f) = 0 and
       e = saleh_events(filename(f),0,idEvent) and
       jetCut(e) and topcut(e) and
       threeLeptonCut(e) and twoLeptonCut(e);
```

The sizes of input streams in the evaluations were scaled over six points for each experiment. The sizes of the event streams from the ROOT files produced in experiment *bkg2* are presented in Table 4.1. The difference between successive sizes is eight files (200000 events). The table also demonstrates for each stream size how many events pass the *Six Cuts Analysis* and the *Four Cuts Analysis,* respectively, along with their selectivities. Table 4.2 presents the sizes of measured streams of events from experiment *signal* (the difference between neighbor sizes is 1437 events) along with the numbers of events that pass each the scientific analysis and their selectivities. For both the data sets the *Six Cuts Analysis* is more selective than the *Four Cuts Analysis*. Both analysis queries are much more selective for substreams of events from experiment *bkg2* than from *signal*.

Table 4.1. The sizes of the event substreams from files produced in experiment *bkg2* (lines 1 and 2), the number of events selected by the scientific analysis queries (lines 3 and 5) and their query selectivities (lines 4 and 6).

Number of files	1	9	17	25	33	41
Number of events	25000	225000	425000	625000	825000	1025000
Six Cuts Analysis	2	47	72	103	139	187
%	0.008%	0.021%	0.017%	0.016%	0.017%	0.018%
Four Cuts Analysis	32	424	826	1240	1607	2002
%	0.12%	0.18%	0.19%	0.19%	0.19%	0.19%

The queries over events from *bkg2* experiment (*bkgsixcuts* and *bkgfourcuts*) selects less than 0.2% of the events. The query *bkgsixcuts* is

Table 4.2. The sizes of the streams of events from experiment *signal* (line 1), the numbers of events selected by the scientific analyses (lines 2 and 4) and their query selectivities (lines 3 and 5).

Number of events	1437	2874	4311	5748	7185	8622
Six Cuts Analysis	234	476	705	932	1154	1387
%	16%	16%	16%	16%	16%	16%
Four Cuts Analysis	835	1691	2524	3363	4226	5083
%	58%	58%	58%	58%	58%	58%

more selective than the query *bkgfourcuts*, and is also larger and more complex, since the *Six Cuts Analysis* (Appendix A) is more complex than the *Four Cuts Analysis* (Example 2.1).

The query *signalsixcuts* is less selective and has low selectivity around 16%. It is more selective than query *signalfourcuts*, which has selectivity of 58%.

4.6.1 Evaluated Strategies

First, the impact of runtime query optimization strategies is investigated, without the query rewrite strategies. The following strategies were evaluated:

Naïve query processing (*NaiveQP*). As a reference point, this strategy demonstrates performance of naive query optimization without reordering aggregated subqueries. The aggregate cost model and runtime query optimization are not enabled. Since the aggregate cost model is disabled the costs of different nested subqueries with aggregate functions are the same, and the query optimizer will not reorder them. Thus 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 strategy demonstrates the impact of static cost-based optimization based on the aggregate cost model. The aggregate cost model is enabled, but not runtime query optimization strategies. Therefore, unlike a loaded database, no data statistics is available when the query is optimized and default statistics are used. Since queries are very large, they were optimized using randomized optimization (Section 2.3.2), which is able to find a good plan in terms of estimated cost. The strategy is compared with *NaiveQP* to demonstrate impact of the aggregate cost model.

Event statistics profiling (*EventSP*). This strategy demonstrates the impact of event statistics profiling (Section 4.4.2) compared with *StatQP*. The query is initially optimized with the aggregate cost model and default

statistics. During execution of the query the statistics on sizes of the event attribute vectors is collected and query reoptimization is performed using collected statistics. The initial optimization uses the fast greedy optimization method (Section 2.3.2) and default statistics. The query reoptimization uses the randomized optimization.

Group statistics profiling (*GroupSP*). This strategy demonstrates the impact of group statistics profiling (Section 4.4.3) compared with *StatQP* and *EventSP*. After query fragmentation into groups, the created groups and their order are initially optimized by the greedy optimization method using default statistics. Fast greedy optimization is used to reoptimize the group order since dynamic programming produced the same execution plans.

Two-phase statistics profiling (*2PhaseSP*). The impact of two-phase statistics profiling (Section 0) is compared with the other strategies. The initial optimization uses greedy optimization and default statistics. In the first reoptimization both groups and group orders are reoptimized using greedy optimization, the aggregate cost model, and collected event statistics. In the final reoptimization, group join order is reoptimized again using greedy optimization and collected group statistics.

The differences between the strategies used to investigate query optimization approaches are summarized in Table 4.3.

Table 4.3. Query optimization strategies and features used in them.

Strategy	The aggregate cost model	Event statistics profiling	Group statistics profiling
NaiveQP	–	–	–
StatQP	+	–	–
EventSP	+	+	–
GroupSP	+	–	+
2PhaseSP	+	+	+

In the next set of experiments, the impact of different rewriting and materialization strategies (Section 4.5) is investigated. All queries are optimized using group statistics profiling and they are compared with *GroupSP* alone in which no materialization or rewriting is implemented.

Rewritten and materialized transformation views (*Trans*). This strategy, described in Section 4.5.1, is compared with *GroupSP*.

***Trans* with vector rewritings (*TransVect*).** This strategy extends the previous strategy with vector rewritings (Section 4.5.3) to evaluate the impact of the vector rewritings.

***TransVect* with caching the computational view *OkJets* (*TransVectCache*).** The impact of caching the computational view *OkJets* (Section 0) is measured and compared with the previous strategy *TransVect*.

Full query processing (*FullQP*). This strategy extends *TransVectCache* with partial evaluation of predicates that access the particle schema meta-data (Section 4.5.4). This strategy implements the all proposed query processing methods.

The difference between these strategies is summarized in Table 4.4.

Table 4.4. Query rewriting and materialization strategies and features used in them.

Strategy	Rewritten and materialized transformation views	Vector rewritings	Caching computational views	Partial evaluation
GroupSP	–	–	–	–
Trans	+	–	–	–
TransVect	+	+	–	–
TransVectCache	+	+	+	–
FullQP	+	+	+	+

As reference points *FullQP* is also compared with manually coded strategies:

Best effort manual plan (*MAN*). This strategy demonstrates the performance of a manually optimized query plan of a query being simplified by the rewritings and materializations in Section 4.5. The order of cuts was manually optimized by experimenting with different orders. The orders were optimized only for the selective queries from experiment *bkg2*, because query reordering has most impact on selective queries and the manual effort to do the optimization is substantial (many hours). The optimal cut order for the definition of the *Six Cuts Analysis* in query *bkgsixcuts* was found to be *Three Lepton Cut*, *Lepton Cuts*, *Miss EE Cuts*, *Z Veto Cut*, *Hadr Top Cut*, and *Jet Veto Cut*. For query *bkgfourcuts* the optimal order was *Three Lepton Cut*, *Two Lepton Cut*, *Top Cut*, and *Jet Cut*.

Unoptimized C++ implementation (*ExpCPP*). This strategy demonstrates the performance of a manual C++ implementation of the *Six Cuts Analysis* (Example 2.2) executed in the same order as in query *bkgsixcuts*. Thus the cuts are executed in the following order: *Hadr Top Cut*, *Jet Veto Cut*, *Z Veto Cut*, *Three Lepton Cut*, and *Other Cuts*.

Optimized C++ implementation (*OptCPP*). This strategy demonstrates the performance of the *Six Cuts Analysis* (Example 2.2) implemented in C++, where the order of the cuts is optimized by a researcher manually. The

optimized order of cuts is: *Three Lepton Cut*, *Z Veto Cut*, *Hadr Top Cut*, *Jet Veto Cut*, and *Other Cuts*.

All evaluated strategies are summarized in Table 4.5.

Table 4.5. Evaluated strategies. Abbreviations: *AgCM* – aggregate cost model, *Event*– event statistics profiling, *Group* – group statistics profiling, *Trans* – rewritten and materialized transformation views, *Vect* – vector rewriting rules, *ViewMat* – the function *OkJets* is materialized, *Parteval* – partial evaluation of schema access predicates, *Man* – manually optimized cuts ordering.

Strategy	AgCM	Event	Group	Trans	Vect	ViewMat	Parteval	Man
NaiveQP	–	–	–	–	–	–	–	–
StatQP	+	–	–	–	–	–	–	–
EventSP	+	+	–	–	–	–	–	–
GroupSP	+	–	+	–	–	–	–	–
2PhaseSP	+	+	+	–	–	–	–	–
Trans	+	–	+	+	–	–	–	–
TransVect	+	–	+	+	+	–	–	–
TransVectCache	+	–	+	+	+	+	–	–
FullQP	+	–	+	+	+	+	+	–
MAN	–	–	–	–	–	–	–	+
ExpCPP	C++ implementation with the expensive order of cuts							
OptCPP	C++ implementation with the cuts ordered by a researcher							

4.6.2 Measured Variables

In the measurement 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 just execute the optimized plan. The measures for the C++ strategies (*ExpCPP* and *OptCPP*) and manual query optimization (*UNOPT* and *MAN*) do not include any optimization times and therefore both times are the same. The measurements of the total query processing time for static query processing (*StatQP*) consist of time to optimize a query and time to execute the query.

Figure 4.14 illustrates what is included in the total query processing time for one-phase runtime query optimization strategies, i.e. *EventSP* and *GroupSP*. After initial optimization, the profiling is enabled for the first *k*

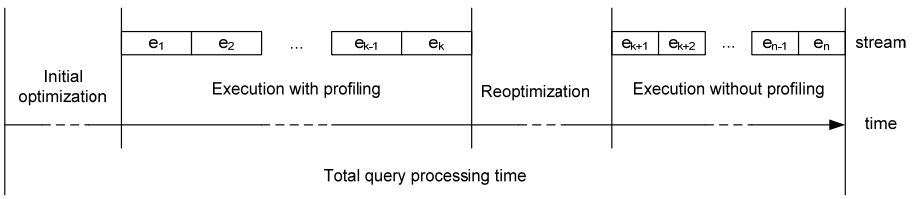

Figure 4.14. Total query processing time with one-phase runtime query optimization.

events, e_1, ..., e_k. Then, after reoptimizing the query execution continues without profiling for the remaining events. In the two-phase statistics profiling strategy (*2PhaseSP*) execution with profiling and reoptimization are performed two times and then execution is continued without profiling overhead.

The final plan execution time measures how well the different strategies improve the execution plan. For runtime query optimization strategies this is measured by executing the query again using the reoptimized plan for the entire event stream.

4.6.3 Setting Optimization and Profiling Parameters

The strategies that rely on runtime query optimization require setting different tuning parameters. For event statistics profiling the confidence interval parameters δ (closeness of sampled mean) and α (probability of the closeness) have to be chosen. As result of tuning experiments, δ is chosen to be equal to 15% and α is 90%, which requires that estimated values should be within 15% from the mean values with probability 90%. For formula (4.3) it corresponds to $\delta = 0.15$ and $z_{\alpha/2} = 1.65$. In reality the difference between the estimated values and the actual values were measured to be less than 10%.

Randomized optimization finds better plans in terms of estimated costs than the greedy optimization method. However, it takes a lot of time for randomized optimization to find a converged plan for large queries. Therefore, fast greedy optimization is always used for the initial optimization of queries in all runtime query optimization strategies. Randomized optimization is used for the final runtime reoptimization.

For randomized optimization (Section 2.3.2), the number of iterative improvement (*II*) steps is chosen to *II = 25* and sequence heuristic (*SH*) steps to *SH = 650*. The aggregated subqueries are larger and therefore *II = 60* and *SH = 300* when optimizing these. These setting were found to produce the cheapest query plan in terms of estimated cost for query *bkgsixcuts* with full query processing and event statistics profiling (*EventFullQP*) in approximately 20 seconds. In general, each query requires different settings of *II* and *SH* and extensive experiments have to be made to find the optimal settings. Furthermore, with cost-based optimization a cheaper plan in terms of estimated costs does not necessarily perform better in practice than a more expensive plan due to the large errors in the estimates of plan costs. Therefore, careful tuning of randomized optimization was not performed per query for all strategies. Instead the same settings for randomized optimization as in *EventFullQP* were used in all experiments.

For group statistics profiling, the stable reoptimization interval (*SI*) (Section 4.4.3) was tuned experimentally, *SI = 4* for both group statistics profiling (Section 4.4.3) and two-phase statistics profiling (Section 0).

Larger values of *SI* were not found to significantly improve performance, while smaller values were unstable.

Join orders of groups are always optimized using greedy optimization, since the fast greedy optimization was found to obtain the same order of groups as randomized optimization and dynamic programming strategies. Predicates inside the groups were always optimized using the greedy optimization method, because the slow randomized optimization inside the groups did not significantly improve overall performance of the final query execution plans.

Table 4.6 illustrates the choices of optimization methods used for the different strategies. Separate strategies and optimization methods were used for initial optimization (*Initial opt.*) and the reoptimizations (*Reopt.*). Optimization of group definitions (*Inside groups*) is different from optimization of group join orders (*Group join*). The cost-based optimization methods used are greedy and randomized. Statistics is collected either on event attribute vector sizes (*evattr.*) or on group execution times and selectivities (*groups*). If reoptimization or collecting statistics are not performed in a strategy it is denoted by *N/A*.

Table 4.6. Optimization methods and statistics collection methods used.

Strategy	Initial opt.		Collect stat.	Reopt.		Collect stat.	Reopt.	
	Inside groups	Group Join		Inside groups	Group Join		Inside groups	Group Join
StatQP	randomized[2]		N/A	N/A		N/A	N/A	
EventSP	greedy[2]		evattr.	randomized[2]		N/A	N/A	
GroupSP	greedy	greedy	groups	N/A	greedy	N/A	N/A	
2PhaseSP	greedy	greedy	evattr.	greedy	greedy	groups	N/A	greedy
Trans	greedy	greedy	groups	N/A	greedy	N/A	N/A	
TransVect	greedy	greedy	groups	N/A	greedy	N/A	N/A	
TransVectCache	greedy	greedy	groups	N/A	greedy	N/A	N/A	
FullQP	greedy	greedy	groups	N/A	greedy	N/A	N/A	

4.7 Evaluation Results

The performance of different optimization approaches without query rewrites is investigated first. Then the additional impact of the query rewritings is investigated. Finally, the best strategy is compared with manually coded strategies.

[2] On entire query without grouping.

4.7.1 Impact of Query Optimization

Figure 4.15 presents performance of the query plans that are obtained by the different optimization approaches for the high selectivity query *bkgsixcuts*.

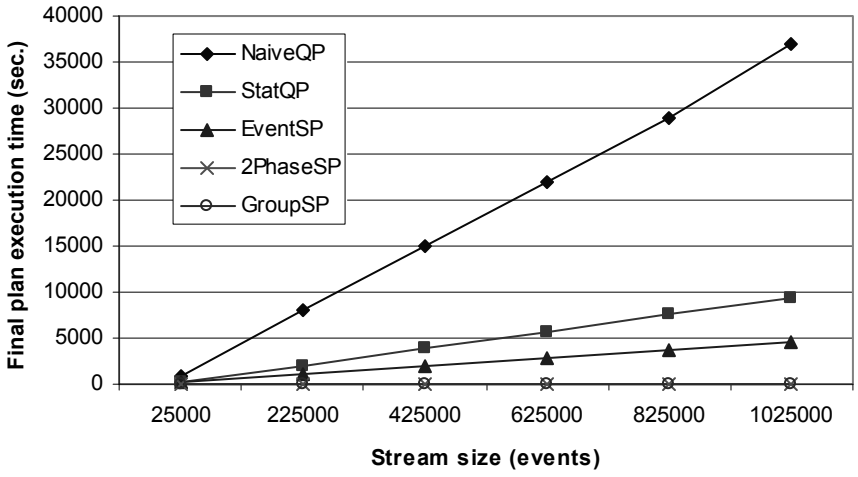

Figure 4.15. Performance of different strategies for query *bkgsixcuts*.

The query plan of the naïve query 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 event statistics profiling (*EventSP*) 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 and two-phase statistics profiling strategies (*GroupSP* and *2PhaseSP*) perform the best and substantially better than the strategies without grouping. They outperform naïve query processing (*NaiveQP*) with a factor 450 and event statistics profiling without grouping (*EventSP*) with a factor 50. This demonstrates that the grouped strategies (*GroupSP* and *2PhaseSP*) alleviate the problem of errors in the estimates [54] by measuring real execution time and fanout for each group. The difference between *GroupSP* and *2PhaseSP* is insignificant.

The optimization strategies are also compared by measuring the total query processing times, including the times to obtain the query plans. Figure 4.16 shows the *optimization overheads* obtained by subtracting the final plan execution time from the total query processing time for query *bkgsixcuts*.

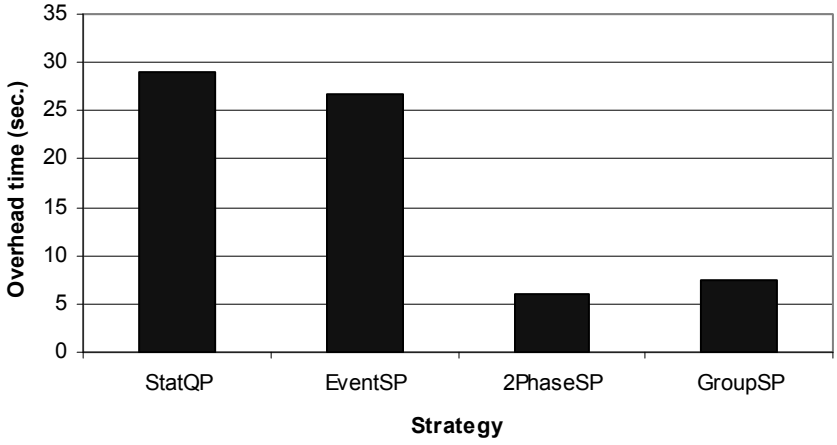

Figure 4.16. Optimization overhead for query *bkgsixcuts*.

These overheads are independent of the stream size so the impact is negligible in practice for large streams.

The optimization overhead of the ungrouped strategy *StatQP* is the time to perform randomized optimization. Also the overhead of *EventSP* is dominated by the randomized optimization (80%). The remaining time is there spent on collecting and monitoring statistics. The overheads of the grouped strategies (*GroupSP* and *2PhaseSP*) are dominated (75%) by performing group profiling. To obtain the final execution plan *GroupSP* profiled only around first 20 events of the stream. So the overhead of profiling all groups for a single event (0.25s) is substantial. The reason is that statistics is 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 is 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 and, therefore, the slow randomized optimization is used.

The query performance for the other selective query *bkgfourcuts* is similar to query *bkgsixcuts*, but with lower overheads since the queries are simpler (Figure 4.17).

Figure 4.18 presents performance of the optimization strategies for the query *signalsixcuts* with the low selectivity 16%. 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 cost model (*StatQP*) gives a query plan that performs 28% better than *NaiveQP*. Using the event statistics profiling

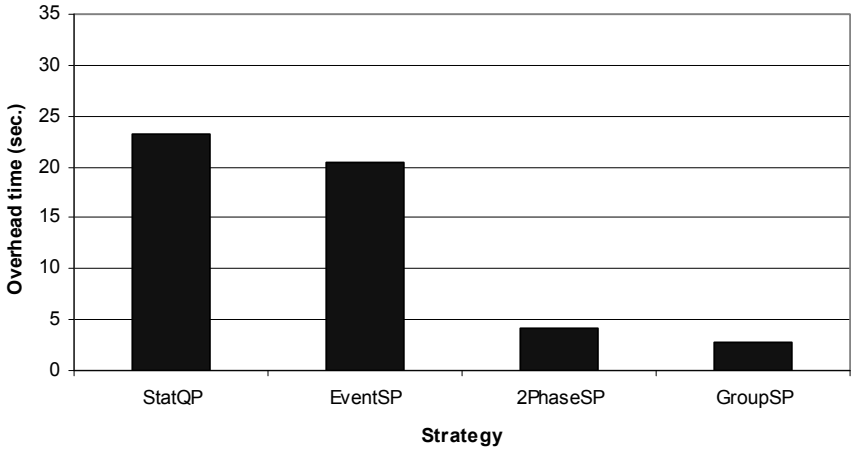

Figure 4.17. Optimization overhead for query *bkgfourcuts*.

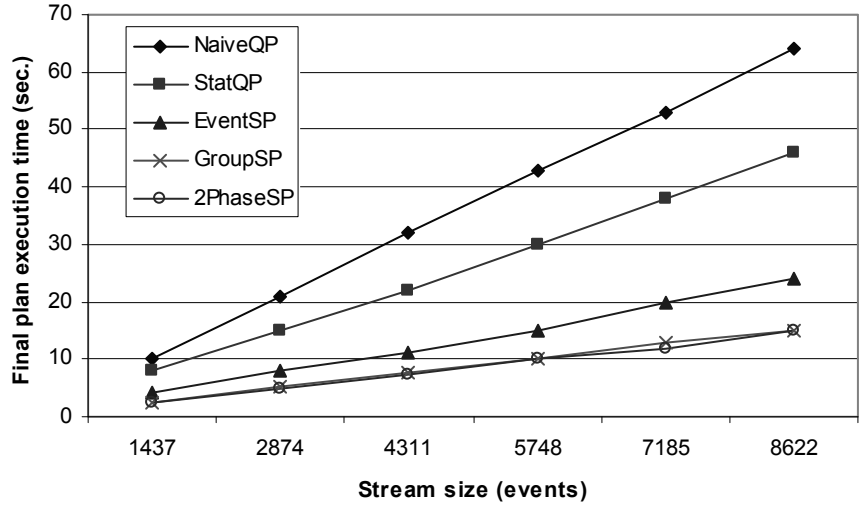

Figure 4.18. Performance for different strategies for query *signalsixcuts*.

(*EventSP*) 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 *EventSP*. The difference between *GroupSP* and *2PhaseSP* is again insignificant. Thus query optimization has substantially larger impact on queries with high selectivities.

The total query processing times of the optimization strategies for the low selectivity query *signalsixcuts* are presented in Figure 4.19. For small stream sizes the overhead of randomized optimization makes the performance of the

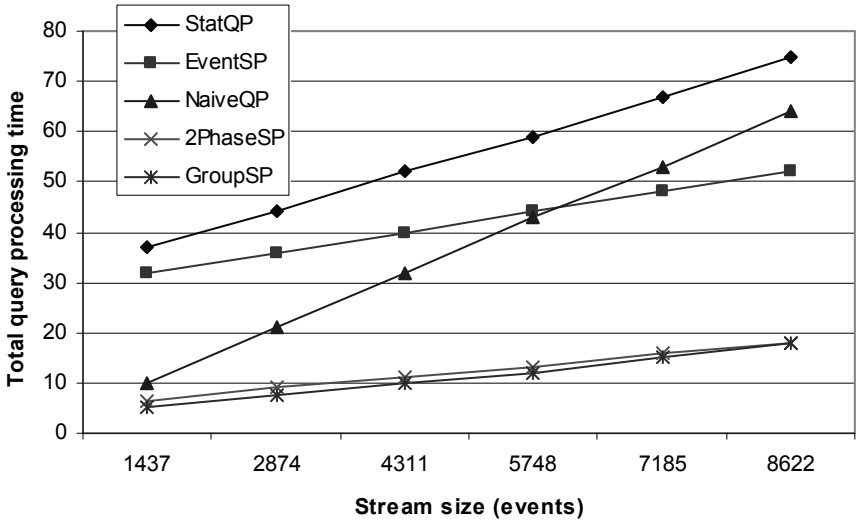

Figure 4.19. Performance of different strategies for query *signalsixcuts*.

ungrouped strategies (*StatQP* and *EventSP*) worse than naïve query processing (*NaiveQP*). However, since the overhead does not depend on stream size, runtime query optimization and static query processing pays off for large streams. Again the overheads of the grouped strategies (*GroupSP* and *2PhaseSP*) are the smallest. The query *signalfourcuts* with very low selectivity (58%) performs similar to *signalsixcuts*.

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 grouped strategies (*GroupSP* and *2PhaseSP*) perform the best.

4.7.2 Impact of Query Rewrites

The performance of query rewritings and materializations is measured by applying different kinds of rewritings on the best runtime query optimization method (*GroupSP*).

The performance of query plans for the selective (<0.2%) query *bk2sixcuts* simplified by the different rewritings and materializations (Section 4.5) is presented in Figure 4.20. The strategy without any rewrites (*GroupSP*) performs the worst. The rewritten transformation views strategy (*Trans*) improves performance by approximately 10%. The vector rewritings (*TransVect*) improves performance further by 5%. Materialization of the computational view *OkJets* view does not give any improvements. Finally, partial evaluation (*FullQP*) improves performance by 3%, giving 17% total improvement compare to *GroupSP*. The conclusion is that the impact of the

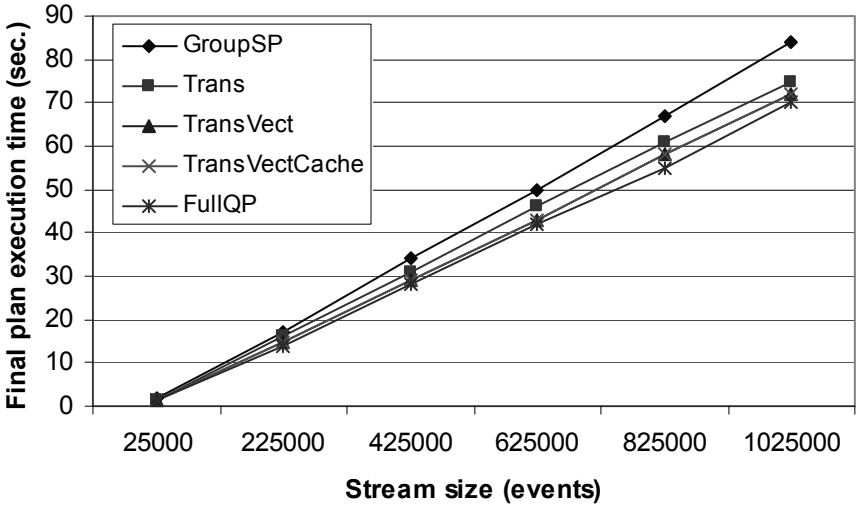

Figure 4.20. Performance of the query rewrite strategies for query *bkgsixcuts*.

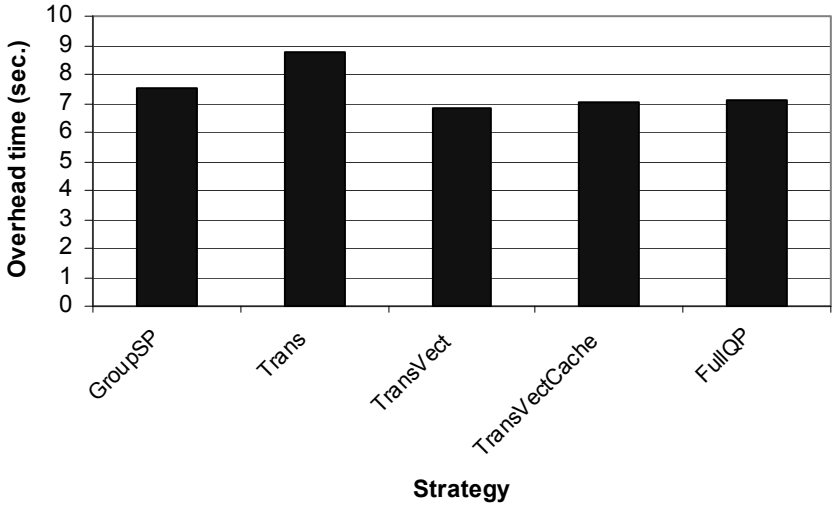

Figure 4.21. Optimization overhead of the group statistics profiling approach combined with rewritings and materializations for query bkgsixcuts.

simplifications is insignificant compared to query optimization for selective queries.

The optimization overheads for the query rewrite strategies are presented in Figure 4.21, where *GroupSP* is not applying query rewrites. The graph

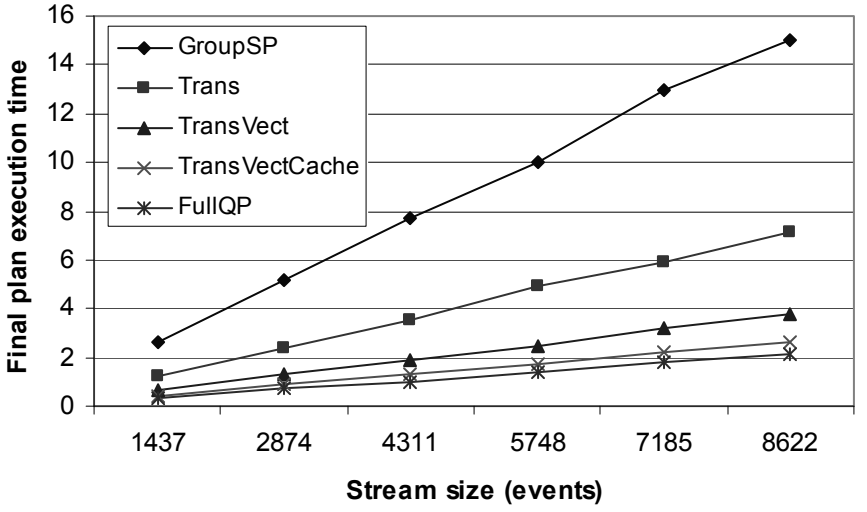

Figure 4.22. Performance of the query rewrite strategies for query *signalsixcuts*.

shows that query rewrites do not change significantly the time spent in query optimization and profiling.

Similar results were obtained for the other selective query *bkgfourcuts*, where improvement of all query rewrites (*FullQP*) was 28% compared to *GroupSP*.

Figure 4.22 demonstrates performance of rewritings and materializations for the query *signalsixcuts* with low selectivity (16%). Rewritten transformation views (*Trans*) improve performance by a factor two. Applying vector rewritings (*TransVect*) gives another factor two in improvement. Materialization of the computational view *OkJets* (*TransVectCache*) further improves performance by 30%. Finally, partial evaluation (*FullQP*) improves performance another 20%. The total improvement between the non rewritten strategy (*GroupSP*) and the strategy with the all query rewrites (*FullQP*) is a factor seven. The impact of query rewrites for the query *signalfourcuts* with very low selectivity (58%) is similar to query *signalsixcuts*. For query *signalfourcuts* the strategy with all query rewrites (*FullQP*) is five times better than *GroupSP*. Query *signalfourcuts* is simpler than *signalsixcuts*, which explains the difference.

In conclusion, for queries with low selectivities the combination of query optimization and query rewrite techniques significantly improve performance.

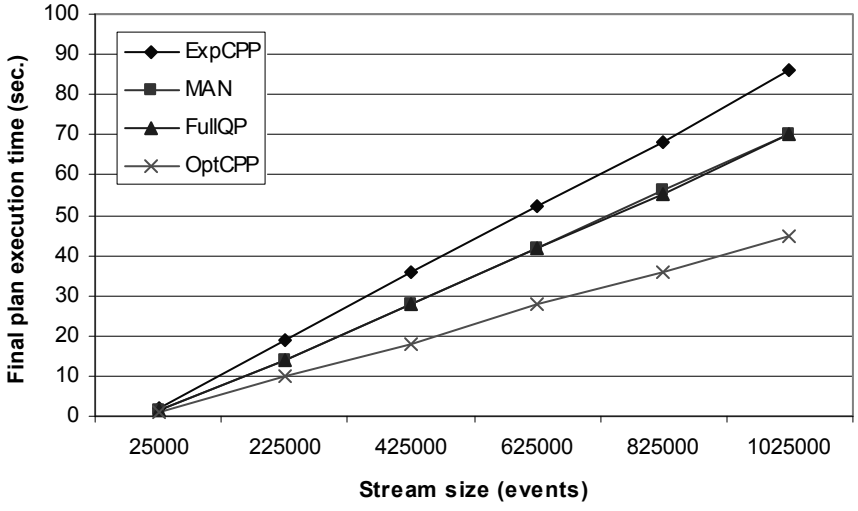

Figure 4.23. Comparing manually coded strategies with full query processing for the selective query *bkgsixcuts*.

4.7.3 Manually Coded Strategies

To measure the impact of the proposed query processing techniques, the best performing strategy *FullQP* is compared with a manually ordered query plan *MAN* and two manually coded C++ programs, *ExpCPP*, and *OptCPP*. Their performances for query *bkgsixcuts* are shown in Figure 4.23. The manually ordered query plan (*MAN*) and the query plan from strategy *FullQP* perform almost the same. They both perform 20% better than the C++ implementation of *Six Cuts Analysis*. The C++ implementation where the order of cuts is optimized manually by the physicist, *OptCPP*, performs 34% better than the query plans from *FullQP* and *MAN*.

This demonstrates that for selective queries the database approach performs as good as a manual C++ implementation of the analysis. Notice that performance can be significantly improved further by conventional code generation, since SQISLE interprets the query plans.

Performance of the strategies for the query *signalsixcuts* with low selectivity (16%) is illustrated by Figure 4.24. For *MAN* the same query plan as used as for the query with high selectivity (*bkgsixcuts*), because it is extremely cumbersome to find the best order manually. *MAN* therefore performs 40% worse than *FullQP*. However, *FullQP* still performs four times worse than the C++ implementations. The reason is that since the selectivities are low 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++. Again, implementing a compiler for query plans will reduce the interpretation overhead significantly.

The evaluation demonstrates that query optimization techniques proposed in this Thesis can achieve performance for large and complex scientific queries close to or better than a manually optimized C++ program.

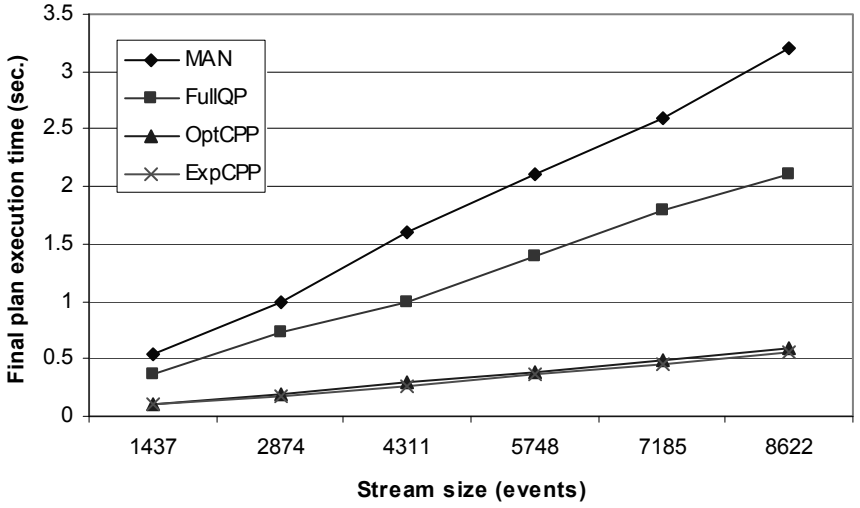

Figure 4.24. Comparing manually coded strategies with full query processing for the query *signalsixcuts*.

4.8 Summary

This chapter presented implementation of a complex scientific application in a data stream management system (SQISLE). It shows that the streaming approach allows to process large volumes of data efficiently. The application specific assumption that makes streamed processing feasible is that every event is analyzed independently from each other so there is no joins between different events in queries. The chapter shows that query optimization techniques enable scalable and efficient execution of large and complex queries implementing scientific analyses. In summary the contributions of the chapter are:

5. The stream object data type implements efficiently complex stream objects representing application events.
6. The profile-controller operator monitors query execution and dynamically reoptimizes the query while it is running. The profile-controller was used to implement and evaluate several dynamic

optimization methods: event statistics profiling, group statistics profiling, and two-phase statistics profiling.

7. Various query transformation techniques were proposed and evaluated. They were shown to improve performance.
8. The performances of the presented contributions were evaluated on queries with different selectivities, to understand their impacts for different kinds of queries.

Combining the all contributions together gives query performances close to or better than with hard-coded C++ implementations of scientific analyses.

Performance of executing queries can be further improved by implementing an algebra compiler. The algebra compiler will remove overhead of interpreting algebra query plans and eliminate the difference in speed compared to C++ for queries with low selectivities. With the algebra compiler SALEH is expected to perform better than C++ for most queries.

The profile-controller operator, which enables runtime query optimization, can be used also for continuous adaptive query execution, during which profiling and monitoring is performed for the entire stream. However, since in our application the distribution of the event properties is constant over entire stream of events, the expensive profiling and monitoring is performed only on small part of a stream and then disabled to eliminate its overhead.

To demonstrate the impact of the rewritings presented in Section 4.5, they were implemented manually and evaluated. Automatic rewritings remains to be implemented.

5. Managing Long-Running Queries in a Grid Environment

Data for the ATLAS application is usually stored on distributed storage resources available through a Grid infrastructure. The amount of data is huge and requires utilizing external computational resources. Therefore, we investigated execution of ALEH queries in the Grid. We developed a framework *POQSEC* [36] (Parallel Object Query System for Expensive Computations) that processes scientific analyses specified as declarative SQL-like queries over data distributed in the Grid. The goal of the POQSEC project is to provide a transparent and scalable way to specify and execute scientific queries in a Grid environment. A user should be able to specify his/her query transparently in a client database without respect to where it will be executed and how data will be accessed.

A high-level layered architecture of running POQSEC is presented in Figure 5.1. POQSEC utilizes computational resources of Swegrid [90], which are clusters, and storage resources of NorduGrid [31], which store event data files, through the middleware Grid infrastructure Advanced Resource Connector (ARC) [32]. ARC manages the computational resources to run POQSEC jobs and transfers event data from storage resources to the clusters. Query execution is performed on the cluster nodes by ALEH, which accesses event data from files through ROOT library [18].

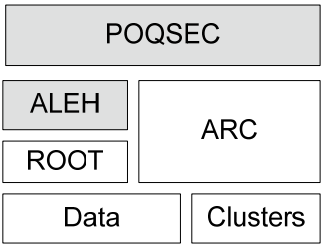

Figure 5.1. High-level architecture of running POQSEC.

POQSEC provides an interface for submitting user queries for execution in the Grid. The system then creates jobs executing the queries, submits the jobs to ARC, monitors execution of the jobs by ARC, downloads results of the jobs, and delivers results of the queries to the user. The user states queries to POQSEC in terms of a database schema available in the client database. The schema contains both an application-oriented part and Grid meta-data. The application schema describes data stored inside files in Grid storage resources, for example events produced by the ATLAS experiment. Wrappers are defined for accessing the contents of these files, e.g. in our application we use a loader to load event data from files by calling the

ROOT library. The Grid meta-data contains information about the files. Thus user queries can restrict data both in terms of application data contents and meta-data about files. The latter is very important since there is a huge amount of Grid data files and queries are normally over a small percentage of them. User queries are parallelized to a number of jobs for execution. The parallelization is done by partitioning data between jobs. Our preliminary results show that the parallelization gives significant performance improvements.

The rest of the chapter is organized as follows. Section 5.1 presents the POQSEC architecture and describes interaction between the DBMS and the Grid infrastructure. It is followed by a brief description of an application query in Section 5.2. The implementation of the framework is presented in details in Section 5.3. Section 5.4 concludes the chapter.

5.1 POQSEC Architecture

The architecture presented in Figure 5.2 illustrates implementation of POQSEC. The POQSEC architecture considers limitations of the Advanced Resource Connector (ARC). ARC and its limitations were described in Section 2.4.1. Here POQSEC components and its interaction with ARC are described.

The *Query Coordinator* of POQSEC manages user queries submitted to POQSEC for execution in the Grid. It communicates with an ARC client directly through a command line interface. Both the query coordinator and the ARC client are running on the same node, the *Grid Client Node*, which is a user accessible computer node. On it the user must first initialize his/her Grid credentials required for using ARC client services according to the *Grid Secure Infrastructure* (*GSI*) [93] mechanism.

The *POQSEC Client* component is a personal POQSEC database running on the Grid client node and communicating with the query coordinator. It could also run on a separate node from the Grid client node, e.g. on a user's desktop computer, if GSI is used for the communication with the query coordinator. Queries are submitted through the POQSEC client to the query coordinator for further execution on Grid resources.

The components of the query coordinator are the *Coordinator Server* and the *Babysitter*. The coordinator server contains a *Grid Meta-Database*, a *Submission Database*, and a *Job Queue*. The Grid meta-database stores information about data files and computational elements accessible trough POQSEC. It is needed since Grid resources are heterogeneous and require Grid users to know the computational elements that are able to execute their jobs and properties of the computational elements required for job executions, e.g. runtime environments. POQSEC users need not specify this

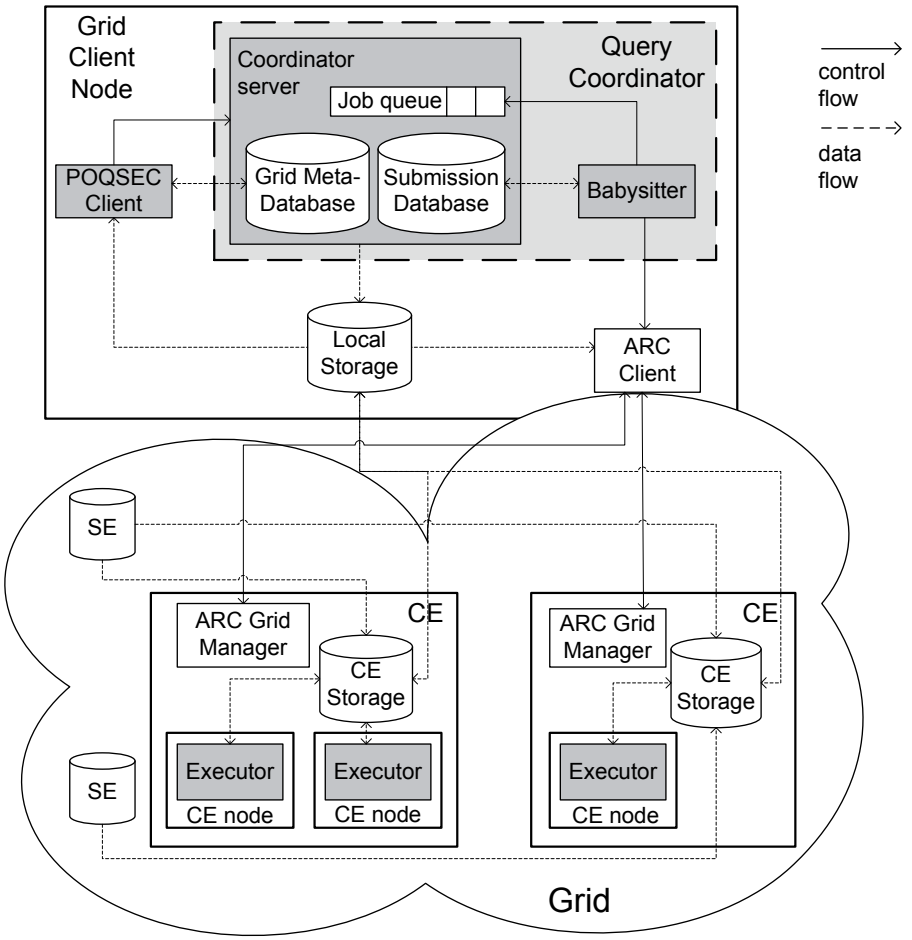

Figure 5.2. Architecture of POQSEC implementation.

information when submitting queries since it is stored in the Grid meta-database.

The submission database contains descriptions of queries submitted from the POQSEC client and job descriptions generated by POQSEC to execute the queries. The job queue contains jobs that are created but not yet submitted to ARC for execution.

The process of submitting and evaluating a query is presented in Figure 5.3. When a query is received (1) from the POQSEC client the coordinator server first registers the query in the submission database and stores there a number of job descriptions to parallelize the query execution. The number of jobs to create is currently provided by the user as part of the query submission. Information about computational resources and data files from the Grid meta-database is used to generate these job descriptions. xRSL scripts (Section 2.4.1) are generated from the job descriptions and are stored

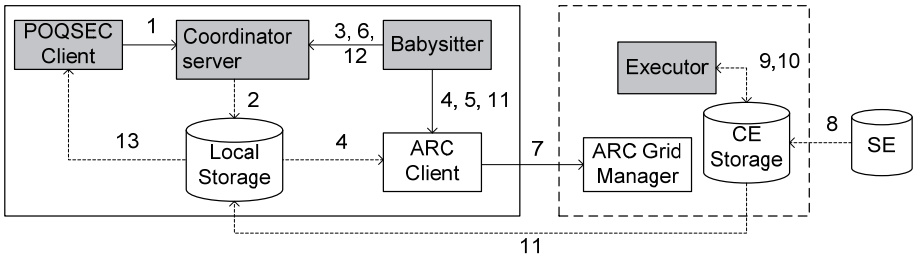

Figure 5.3. Interactions between POQSEC components and ARC.

(2) in the local storage. Then the jobs are registered in the job queue. The babysitter picks (3) jobs from the job queue and submits (4) them as xRSL scripts to the ARC client for execution on Grid resources. Once a job has been submitted the babysitter regularly polls (5) the ARC client for its job status and reports (6) the status to the coordinator server to update the submission database. When a job is finished the babysitter downloads (11) the result to the *Local Storage*, which is the file system of the Grid client node, and notifies (12) the coordinator server. The result can be retrieved (13) to the POQSEC client after the query is finished.

On each CE ARC maintains an *ARC Grid Manager*. It receives (7) job descriptions from ARC clients. In our case these jobs are executing POQSEC subqueries. The ARC Grid manager uploads (8) input files from SEs to the local *CE Storage* before each job is submitted to the local batch system. The local batch system allocates *CE nodes* for each job according its policies and current load, and then starts the job executions. For POQSEC these jobs contain *Executors*, e.g. ALEH, that evaluate (9) subqueries over uploaded data and store (10) the results in local CE storage files. The babysitter polls (5) the ARC client regularly for finished executions. After a job has finished the babysitter requests (11) the ARC client to download (11) the result to the local storage of the Grid client node and notifies (12) the coordinator server that the job is ready. Since a given POQSEC query often generates many jobs a query is ready only when all its jobs are finished. However, partial results can be obtained once some jobs are finished.

5.2 HEP Queries

POQSEC is evaluated on the ATLAS application (Section 2.1) implemented in ALEH (Chapter 3). Evaluation experiments are performed on the naïve query processing of a scientific query, which implements the *Six Cuts Analysis* (Appendix A) and is defined in terms of the particle schema (Figure 2.3) over data loaded from ROOT files. The query definition is:

```
select e
from Event e
where  jetvetocut(e) and zvetocut(e) and                    (5.1)
       hadrtopcut(e) and misseecuts(e) and leptoncuts(e)
       and threeleptoncut(e);
```

The query is expressed in terms of derived functions that define the cuts. Definitions of the cuts are different from the ones presented in Chapter 3 (Appendix C), since the POQSEC experiments were done earlier (published in [37]) than the experiments presented in Chapter 3 (published in [38]) and queries were specified without the restriction to be conjunctive. The definition of one of the cuts is:

```
create function zvetocut (Event e) -> Boolean as
select TRUE
where  notany(oppositeleptons(e)) or
       (abs(invMass(oppositeLeptons(e)) - zMass) >= minZMass);
```

Where *invMass* calculates the invariant mass of a pair of two given leptons, *zMass* is the mass of a Z particle, *minZMass* is range of closeness, and *oppositeLeptons* is a derived function defined as another query:

```
create function oppositeLeptons (Event e) -> <Lepton, Lepton>
                as
select l1, l2
from Lepton l1, Lepton l2
where  l1 = particles(e) and
       l2 = particles(e) and
       Kf(l1) = -Kf(l2);
```

5.3 Implementation

A POQSEC client running the ALEH application has an interface to a coordinator server through which a user can submit queries for execution in the Grid. It can monitor the status of submitted queries, and can retrieve results of finished queries. To submit a query the user invokes a system interface function named *submit* and specifies there the query defined in terms of the application schema, set of file names which should be processed by the query, number of jobs for parallelization the query, CPU time required for processing one job, and optionally a computing element where the query's jobs should be executed. If no computing element is specified the jobs will be submitted to an ARC client along with a list of possible

computing elements for execution. The result of the *submit* function is an object used to monitor the status and to retrieve the result.

The test data are events produced by ATLAS simulation software and stored on storage recourses accessible through ARC. Paths to the data files are stored in the Grid meta-database of the coordinator server in a format according to xRSL specification [86]. Thus the user provides file names without paths during submission.

For example, the user wants to execute on any of available computational resources of Swegrid the analysis query (5.1) over eight specific files, with parallelization in four jobs, where each job will process two files, where the CPU time of executing the query over the two files is 20 minutes,. The user submits the query and assigns the result of the submission to a variable *:s*:

```
set :s = submit( "select e
                  from Event e
                  where  jetvetocut(e) and
                   zvetocut(e) and
                   hadrtopcut(e) and
                   misseecuts(e) and
                   leptoncuts(e) and
                   threeleptoncut(e)",                    (5.2)
                 {"bkg2Events_000.root",
                  "bkg2Events_001.root",
                  "bkg2Events_002.root",
                  "bkg2Events_003.root",
                  "bkg2Events_004.root",
                  "bkg2Events_005.root",
                  "bkg2Events_006.root",
                  "bkg2Events_007.root"},4,20);
```

The submission is then translated into four xRSL scripts, which are submitted to an ARC client for execution. One of the scripts is presented in Figure 5.4. The executable there is the ALEH application, which contains the loader of ROOT files.

It is necessary for the user to specify which files to analyze to restrict amount of data for processing. In the example the user specifies file names explicitly. Alternatively the user can define a query over the meta-database of the coordinator server to retrieve the file names. The local batch systems of all computational elements available through ARC require specification of CPU time and thus the user needs to provide this[3].

[3] With profiled grouping approach it can be estimated automatically using the measured execution time for each group.

```
& (executable=aleh)
(arguments="aleh.dmp")
(inputfiles= (aleh "/home/udbl/ruslan/Amox/bin/aleh")
   (aleh.dmp "/home/udbl/ruslan/Amox/bin/aleh.dmp")
   (query2005420103329443.osql "query2005420103329443.osql")
   (bkg2Events_ruslan_001.root "gsiftp://se1.hpc2n.umu.se:2811/
se3/ruslan_poqsec/bkg2Events_ruslan_001.root")
   (bkg2Events_ruslan_000.root "gsiftp://se1.hpc2n.umu.se:2811/
se3/ruslan_poqsec/bkg2Events_ruslan_000.root"))
(outputfiles=(result.out ""))
(cputime=20)
(|  (runtimeenvironment=ROOT-3.10.02)
   (runtimeenvironment=APPS/HEP/ATLAS-8.0.8)
   (runtimeenvironment=APPS/PHYSICS/HEP/ROOT-3.10.02)
   (runtimeenvironment=ATLAS-8.0.8)
   (runtimeenvironment=APPS/HEP/ATLAS-9.0.3))
(stdin="query2005420103329443.osql")
(stdout="outGen.out")
(stderr="errGen.err")
(gmlog="grid.debug")
(middleware>="nordugrid")
(|  (cluster=sg-access.pdc.kth.se) (cluster=bluesmoke.nsc.liu.se)
   (cluster=hagrid.it.uu.se) (cluster=hive.unicc.chalmers.se)
   (cluster=ingrid.hpc2n.umu.se) (cluster=sigrid.lunarc.lu.se))
(jobName="POQSEC: swegrid2005420103329444.xrsl")
```

Figure 5.4. Example of the xRSL file with name swegrid2005420103329444.xrsl.

The performance of many queries can be significantly improved by parallelization into several jobs. Our experience shows that parallelization of executing a query gives dramatic improvements. For example, the above submission took 24 minutes. The time was calculated as the elapsed time between when the query was submitted until all job results were downloaded from the Grid. A submission of the same query without parallelization as one job was much slower and took 3 hours and 45 minutes, where 3 hours and 10 minutes were spent for the query evaluation.

During execution of a query submitted to POQSEC the user can monitor its status of a submission *:s* by calling *status(:s)*. The status of the query is computed from its batch jobs statuses. The status *"DOWNLOADED"* will be returned only if results of all jobs of the query were downloaded. Then the user can retrieve the result data by executing *retrieve(:s)*. The result of the query can be retrieved also by using the function *wait(:s)*. The difference is that if *wait* is invoked before the result of the jobs is available the system waits until the coordinator server notifies it that all jobs are downloaded.

After that it retrieves the result, while *retrieve* will just print a message if the query is not finished. The user can cancel his/her query submission by executing *cancel(:s)*.

The coordinator server, the babysitter, and the ARC client are running on the same Grid client node as the POQSEC client. The coordinator server contains the Grid meta-database and the submission database. The user is able to query the coordinator server for data from the Grid meta-database and to request updates of the Grid meta-database through the POQSEC client. The babysitter polls the coordinator server to pick up jobs from the job queue and to request updates of the submission database.

A schema of the Grid meta-database and the submission database is presented in Figure 5.5. The Grid meta-database is defined by the type *Cluster* and the source database (similar to Figure 4.2) with the type *DataFile* and its subtype *EventData*. The submission database is presented by the types *Submission* and *Job*.

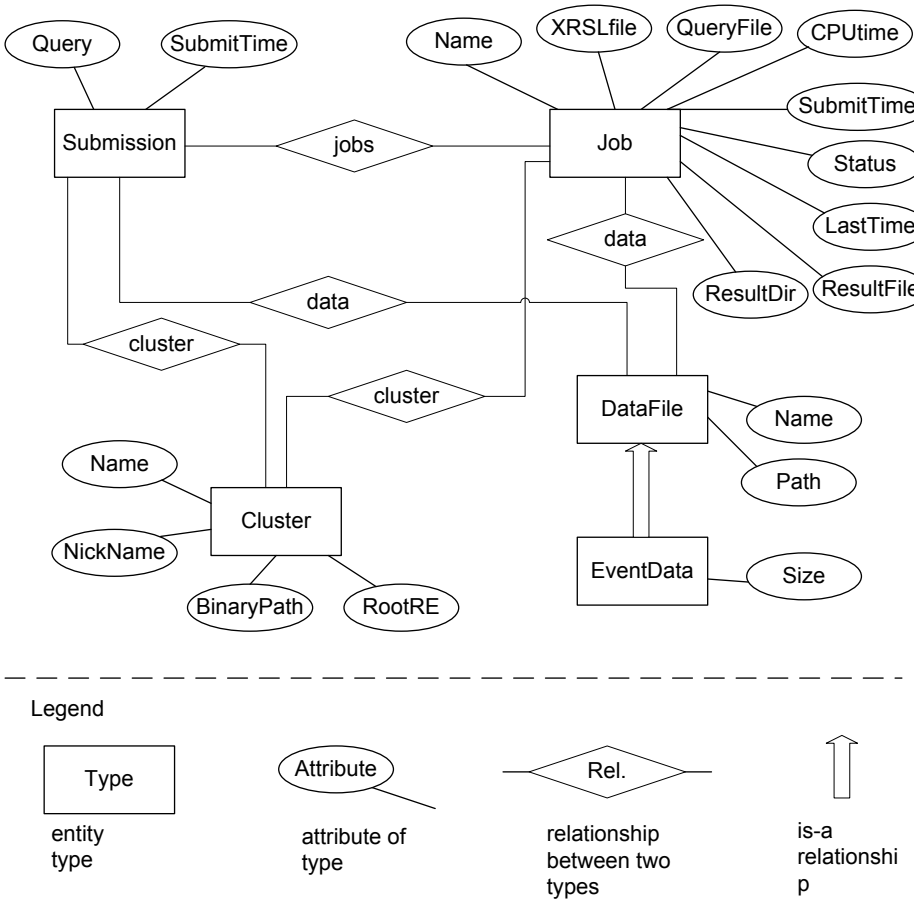

Figure 5.5. Schema of the Grid meta-database and the submission database.

When the coordinator server receives query submissions from the POQSEC client it generates job descriptions and creates xRSL files for ARC and script files for POQSEC executors. For example, for the submission given above the coordinator server generates four xRSL files and four script files. Example of one of the xRSL file is given in Figure 5.4. The POQSEC script files contain commands for executors to load the input data from the data files through the ROOT loader and to execute the user query. In our example one of the script files contains:

```
load_root_file("bkg2Events_ruslan_001.root");
load_root_file("bkg2Events_ruslan_000.root");
save("result.out",
      select e
      from Event e
      where  jetvetocut(e) and zvetocut(e) and
             hadrtopcut(e) and misseecuts(e) and
             leptoncuts(e) and threeleptoncut(e));
```

The results of the query executions are saved by the executors in files (here in *result.out*) in a way that they can be read by the POQSEC client. Objects, in our case events, which originally were the same, will be treated by the POQSEC client as the same object regardless of that they came from different sources.

The other three xRSL files and three script files are similar except that they have different input data files. Automatic generation of the files by POQSEC exempts the user from manually creating such files for each job.

The main tasks for the babysitter are to interact with the ARC client to submit jobs, to monitor status of executing jobs, and to download finished jobs. Each interaction with the ARC client can take from several seconds to a minute; thus the coordinator server does not contact the babysitter immediately when a job is created. Instead the babysitter polls the coordinator server regularly when it is not interacting with the ARC client.

5.4 Summary

We implemented a framework that provides basic tools for executing long running batch queries on Grid resources over scientific data distributed in the Grid. With the framework a user specifies files to analyze by queries to the Grid meta-database and analysis of the data from the files in queries. The framework interacts with the Grid and executes queries there.

With use of the framework the Grid can be utilized to scale analysis queries over big volumes of data, since the queries are parallelized and executed on non-dedicated distributed Grid nodes in parallel.

6. Related Work

This chapter presents research related to the Thesis contributions. In summary, the major contributions are:

- The use of a query language to implement scientific application selecting events with complex structure.
- Techniques for efficient processing of queries over streams of events with complex structure.
- To enable efficient execution plans for streamed queries, the profile-controller operator manages different runtime query optimization strategies.
- The cost model for aggregate functions over nested subqueries enables optimization of complex queries having selection conditions with many aggregate functions.
- The profiled grouping approach fragments queries into groups, measures execution time and fanouts for each group, and optimizes the join orders of groups using the measured statistics.
- The performance of streamed queries with low selectivities is shown to be improved significantly by using query transformation techniques, view materialization, and partial evaluation.
- An infrastructure for managing queries executed in a batch-oriented grid infrastructure enables scalable parallel execution of queries on external computational resources over data stored in grids.

In this chapter different kinds of technologies related to these contributions are discussed. First, high level interfaces for specifying analysis in HEP applications are studied. Second, related work in DSMS is studied. Third, related techniques on adaptive query processing are discussed. Fourth, related database techniques to process complex queries are presented. Fifth, related work on databases utilizing computational resources through Grid infrastructures and other computational distributed infrastructures is presented. Finally, systems that apply database technologies for scientific applications in general are presented.

6.1 High-Level Analysis Tools for HEP Applications

It is recognized that physicists need to perform their analyses of ATLAS data by systems that are simple to use. Several systems for ATLAS experiment (for example, ADA [3] and DIAL [28]) are developed. They provide job submission systems oriented for executing ATLAS analyses on distributed computational resources and hide details of different underlying Grid infrastructures, batch systems, and ATLAS environment installations. Physicists specify their analyses as C++ and Python programs and provide descriptions of their jobs in some job description language to perform the analyses on external resources. Then the high level analysis tools take care of distributing and executing the analysis on computational resources and return merged results to the scientists. By contrast, in our system scientists specify analyses in a query language, which is more high-level and requires less time to specify the analyses than writing C++ or Python programs. As the other systems, our system also takes care of executing the analyses in parallel on external resources managed by a Grid infrastructure.

A visual query language for specifying HEP analyses is provided by the system PHEASANT [5]. HEP analyses are there defined in queries, which then are compiled into a general purpose language [80] without performing any query optimization or query simplification. By contrast, our system rewrites and optimizes queries, which is shown to give significant improvement in performance, approaching that of hard-coded C++ programs.

6.2 Data Stream Management Systems

Most developed DSMSs (e.g., Aurora [2], Gigascope [21], STREAM [1], TelegraphCQ [61], and XStream [42]) focus on infinite streams of rather simple objects and efficient processing of time-series operations including aggregates and joins of the streams. Such DSMSs are data driven and process the streams by continuous queries. In contrast, in SQISLE elements of streams are complex objects (each event can be seen as a small database) and large and complex queries are applied on each streamed object independently from other objects. Therefore, the queries in SQISLE do not contain time-series operations and no join between streams is performed. Furthermore, SQISLE is demand driven, since it has full control of the stream flow.

Aurora [2] processes rather simple continuous queries over dynamic streams of rather simple tuples. Queries are defined on algebra level and views are not supported in the query definitions. The performance of stream queries is improved by rewriting and optimizing algebra execution plans of the stream queries. The rewriting combines several algebra operators into

one operator to reduce operator execution overhead. During execution of stream queries Aurora continuously measures costs and fanouts for each operator. Then operators in query plans are greedily reoptimized using the measured statistics by Aurora's cost-based optimizer. In contrast, SQISLE processes large complex queries over stable streams of complex objects. The queries are specified in a declarative SQL-like query language with use of many views. Therefore, SQISLE implements different query processing techniques. Complex large queries are optimized using group statistics profiling, which is shown to produce better performing plans than greedily optimization of the ungrouped query. In contrast to continuously adapt query execution plans for dynamic streams as in Aurora, SQISLE adapts a query execution only for a small part of a stream until group statistics are stabilized.

In SQISLE the rewriting of transformation views combines calculus predicates, which simplifies and speeds up cost-based query optimization and reduces the interpretation overhead. In contrast to Aurora SQISLE in addition applies rewrites to remove unnecessary predicates, e.g. the vector rewriting rules that remove unnecessary vector constructions and vector element accesses.

TelegraphCQ [61] and its extensions CACQ [70] and CBR [11] process simple and small queries, which do not contain nested subqueries, over streams with dynamic properties. By contrast SQISLE processes large and complex scientific queries that contain many nested subqueries. SQISLE aims to efficiently process large complex queries with many aggregates.

STREAM [1] has a cost-based query optimizer that optimizes query plans for runtime memory minimization [10]. The system periodically measures execution times and fanouts of each operator and reoptimizes the execution order of the operators. By contrast SQISLE does not collect statistics on each operator of a large query execution plan and, therefore, minimizes the profiling overhead. Furthermore, in contrast to memory minimization being the focus in STREAM, the query optimization in SQISLE minimizes processing time of each single complex object for complex and large queries. SQISLE does not need to minimize memory consumption, since only one complex object is materialized in memory at a time. Another difference is that SQISLE rewrites queries to simplify them while STREAM does not.

MIT develops a data stream management system, XStream [42], to process high rate scientific streams of isochronous temporal data with application specific analysis. A high processing rate of streamed data is enabled by implementation of a new data type, *SigSeg* [42], for representing large windows of streamed tuples, and providing efficient operators executed over these windows. By contrast each streamed event in SQISLE is a complex object and analyzed separately from other events. Furthermore, the analyses first derive new objects from each streamed complex object and

then objects are selected in terms of properties of the derived objects. Therefore, SQISLE focuses on efficient processing of each complex object separately, while XStream concentrates on efficient processing of large windows of rather simple tuples.

6.3 Adaptive Query Processing

In DBMSs and 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, which utilize runtime feedback and modify query processing [29]. AQP systems (e.g. [4][14][65][68]) usually continuously adapt the execution plan of a query to reflect significant changes in data statistics. By contrast SQISLE profiles a query until no significant changes in stream properties are noticed, i.e. statistics on the stream is stabilized, and then reoptimizes the query using the stable statistics. Therefore, after the statistics is stabilized, the rest of a stream is efficiently processed without profiling overhead.

Many AQP systems (e.g. [4][11][14][55][65][68]) collect statistics only on cardinalities. Some of them [68] inject monitoring operators in a query execution plan to measure throughput between pairs of operators. Other [14][55] uses processing operators that also monitor their fanouts. Similarly the event statistics profiling in SQISLE collects statistics on cardinalities of event properties by a wrapper interface function. By contrast the group statistics profiling in SQISLE first rewrites the processing query fragment into groups and wraps each group with the group monitor operator. The group monitor operator measures the execution time and fanout for each wrapped group during query execution. In both the event statistics profiling and group statistics profiling the measured statistics are used to optimize a query in terms of both fanouts and costs.

Different AQP systems implement different mechanisms for changing running query execution plans to more efficient ones during query execution. Some AQP systems (e.g. [14][46][55][65]) generate several query execution plans for the entire query or for query fragments during initial optimization and switch between the plans during query execution. By contrast SQISLE generates only a single query execution plan during the initial optimization. Then during query execution the controlled query fragment, i.e. the processing query fragment, is reoptimized using collected statistics to obtain a more efficient execution plan. Generating many execution plans during initial optimization is not feasible for large and complex queries.

A number of AQP systems [1][2][68] initially generate a single plan as SQISLE and then reoptimize the entire query during query execution. This is a common strategy for data driven DSMSs [1][2]. However, for demand driven AQP systems [68], this requires implementing a mechanism to exploit

already computed intermediate results. In contrast to reoptimizing the entire query SQISLE reoptimizes only the processing query fragment. Execution of the source access plan, which controls stream generation, is not affected by the reoptimization.

Usually DSMSs (e.g. Aurora [2], STREAM [1], and TelegraphCQ [61]) implement scheduling operators that route tuples through processing operators. A scheduling operator decides for each tuple which operator is going to process the tuple. After the tuple is processed by the processing operator it is returned back to the scheduling operator to be send to a next operator. The scheduling operators in STREAM and Aurora route tuples according an execution plan produced by a query optimizer, which reoptimizes the execution plan when it notices significant changes in the monitored stream statistics. TelegraphCQ uses the eddy operator [4], which makes dynamic decisions for every tuple and every operator. Invoking a scheduling operator for every tuple and every operator is important to deal with high-rate and burst streams. However, this strategy adds an overhead to each processing operator, thus the overall overhead in query processing is going to be larger for larger queries. Furthermore, the overhead for the eddy operators is much more significant than the overheads of the other scheduling operators. By contrast SQISLE does not deal with high-rate and bursty streams and focuses on efficient processing of queries over each independent event. Thus the profile-controller operator is executed once per input event before the event is filtered by the complete event processing plan. This minimizes overhead of invoking the profile-controller operator during execution of large queries.

6.4 Processing of Complex Queries

Modern database applications perform complex queries over stored data. For example, on-line analytical processing (OLAP) and data mining queries are often complex [19]. Complex queries are usually defined in terms of views and consist of many joins, aggregate functions, nested subqueries, selections, and user-defined functions. Various query processing techniques are developed to improve performance of such complex queries.

OLAP and data mining queries are usually performed over data objects loaded into a data warehouse, similar to the loading approach. Our queries process each independent complex object separately and the performance evaluation demonstrated that therefore the streaming approach performs significantly better than the loading approach.

Rewriting calculus representation of queries into equivalent representations during a pre-processing phase before cost-based query optimization [58][49] is demonstrated to improve query performance for different kinds of applications in, e.g., engineering [88], image processing

[69], and business processing [92]. In this Thesis several novel rewrite rules are developed for SQISLE and evaluated for a scientific test application. The performance experiments confirm the importance of query rewrite rules for query processing.

In SQISLE view results are temporarily materialized in the stream object representing the event. The system provides efficient access and immediate removal of materialized view results as the stream is progressed. This is different from methods to cache expensive computations in a regular DBMS where results are materialized permanently in, e.g., a hash table [51].

Cost-based query optimization is important to obtain an efficient execution plan for a complex query. Most cost-based optimizers are based on the System R approach [87]. Such optimizers are limited to optimize only join-orders of relations. Therefore processing of complex queries with expensive predicates requires either optimizing the predicates separately from the relation join-order [48][26] or transforming queries to regard the expensive predicates as joins of relations [6]. For example, [48] and [26] optimize queries in the presence of expensive functions by optimizing the order of the expensive predicates for each relation join-order. An example of the transformation approach is the optimization of queries containing aggregate functions over nested subqueries. In relational databases such queries are rewritten to either regard the nested subqueries as joins of relations or to unnest the nested subqueries into a flattened query [6][25] . Thus, relational database optimizers are based on the fact that, for a large disk-based database, the join is very expensive compared to selection operators. In SQISLE all operators have similar costs. Therefore, SQISLE needs to optimize all operators in the complex query together and does not differentiate between various types of predicates. To deal with queries containing aggregate functions, the aggregate cost model for aggregate functions over nested subqueries is developed in the Thesis. To our knowledge there is no published work presenting a cost model for aggregate functions over nested subqueries, since such queries are usually regarded as joins [6][25][45][63][94].

Cost-based query optimizers are often based on approximate statistics and simplified cost models, e.g. based on the independence assumption. Thus estimates of query plan costs are inaccurate, and the errors in the cost estimates are propagated in large queries [54]. Therefore, query optimization of large and complex queries is often unreliable. Many works [13][17][34][50][78][84] focus on collecting and maintaining statistics for different operators and their combinations. They are limited to simple SPJ queries [13][34][78][84] or they do not cover statistical dependencies [17][50]. In this Thesis instead of developing a cost model for large and complex scientific queries which is reliable and covers dependences, we develop the profiled grouping approach that fragments queries into groups, measures costs and fanouts for the groups, and optimizes join-order of

groups using the measured cost model. The performance evaluation demonstrates that the profiled grouping approach finds better performing plans than the cost-based query optimization alone without the query fragmentation.

In [12] and [39] statistics on query fragments or views are used to handle data dependencies during optimization of simple SPJ queries, without collecting statistics for large numerical queries. They do not automatically fragment the queries. In contrast SQISLE automatically fragments queries using the stream fragmenting algorithm developed for our kinds of applications.

6.5 Databases and Distributed Computational Infrastructures

Database queries that are computation and data intensive require using external computational resources to scale query execution. Large amounts of computational resources are shared within communities across the Internet, e.g. through Grid infrastructures. Therefore, different projects (e.g., DQP [64], CODIMS-G [81], STORM [72], and LeSelect [22]) develop frameworks to execute expensive database queries on distributed heterogeneous external computational resources.

The distributed query processing system (DQP) [83] (its web service version is called OGSA-DQP [64]) is a system that utilizes a Grid infrastructure and provides a high-level declarative query language for data access and analyses. The DQP scheduler requires that dedicated resources for the distributed query execution are preallocated before a query is parallelized and optimized [43]. Any of the preallocated resources can be utilized by DQP dynamically. DQP is different from POQSEC that utilizes ARC as a middleware above autonomous local batch systems on computational resources. Unlike DQP, neither POQSEC nor ARC have full control of computational resources and POQSEC therefore needs to consider the ARC limitation that jobs are not guaranteed to start immediately. Furthermore, as part of a job description ARC requires to specify all descriptions of resources in advance. This includes, for example, estimating execution time and number of computational nodes for jobs.

CODIMS-G [81] is an adaptive parallel query processing middleware on top of a Grid infrastructure that allows dynamic allocation of dedicated computational resources. CODIMS-G dynamically allocates computational resources, deploys query engines on the allocated nodes, and performs query execution on the deployed query engines. During query execution resources can be adaptively re-allocated. In contrast POQSEC runs on top of Grids

where resources are managed by batch systems, which do not allow immediate allocation of resources.

STORM [72] is a framework for processing on distributed and parallel resources very large multi-dimensional scientific datasets stored in distributed files. During execution of the query, selections are performed locally on distributed storage resources and the results of selections are shipped to a computational cluster. Then on the computational cluster the selected subsets are joined and the rest of the query is executed. By contrast grid based query management in POQSEC does not assume pre-installed database management capabilities on storage resources, and therefore queries are executed only on computational resources to where both files and the lightweight data management system are shipped.

LeSelect [22] is a distributed mediator system, which aims to support scientific collaborations. Through LeSelect scientists working at different locations share data, which are binary large objects (blobs), and analysis code, which is represented as external expensive functions. LeSelect assumes that functions can be executed only on LeSelect servers. Therefore data are transferred to LeSelect servers storing the expensive functions for processing. By contrast POQSEC transfers both data and analysis code. Furthermore, in contrast to pre-installed LeSelect servers, POQSEC does not require to install its components (e.g., SALEH) on computational Grid nodes in advance.

6.6 Scientific Databases

Database technologies have been extended for scientific applications to provide high-level easy-to-use interfaces for scientists (e.g., MauveDB [30], LeSelect [22], SDSS [91], XStream [42], STORM [72]). Using such scientific data management systems the scientists can analyze their scientific data in terms of views in efficient and transparent ways without studying details of wrapped complex underlying systems used to store or generate scientific data.

The system MauveDB [30] focuses on supporting views, called *model-based views*, which are defined in terms of statistical models over base tables containing scientific data. MauveDB allows writing SQL queries in terms of model-based views that transform base data into the data representing the views. MauveDB utilizes a relational query optimizer during query processing. For this, either the queries are rewritten in terms of base tables by application specific rewriting rules if the rules are available [62], or, otherwise, cost models for model-based views are defined [30]. Then the join order of the base tables and views is optimized. In contrast to implementing views in a foreign language and defining specific transformation rules for expanding the views, views in SQISLE are defined

as queries and, therefore, are automatically expanded by general view expansion rules. Furthermore, the queries are optimized in order to reduce time to analyze independent events, while in MauveDB query optimization focuses on join ordering. Thus MauveDB reduces data accesses from disk, while SQISLE optimizes operators processing streams of complex objects.

An example of implementing a complex scientific application in a relational DBMS is the Sloan Digital Sky Survey (SDSS) project [91]. In the project huge amounts of astronomical data from the SDSS telescope are loaded into a cluster of SQL Server databases, which corresponds to data warehousing or loading approach. The SQL queries submitted for execution by the parallel DBMS contain application specific computations implemented in SQL and external languages. In SQISLE the queries are also specified in a declarative query language similar to SQL and include application specific computations. In contrast to SDSS, in SQISLE the application data is not loaded into the database. Instead the data is stored in original files and accessed in a stream fashion that demonstrates significantly better performance for our type of applications than the loading approach (ALEH).

7. Summary and Future Work

This Thesis presented implementation of a query processing system targeted to scientific applications where data are independent events with complex structures selected by complex large queries. The queries process large volumes of data stored in files distributed in Grids. The new system POQSEC for managing scientific queries in Grids was developed. POQSEC parallelizes queries by data partitioning and executes them in a Grid through the Grid infrastructure Advanced Resource Connector [32].

Processing of the queries on computational nodes of a Grid is performed by new data stream management system SQISLE, which is an extension of the functional main-memory DBMS Amos II [79]. It accesses events from files through a wrapper interface and process them efficiently by utilizing novel query processing techniques. SQISLE implements runtime query optimization methods to collect runtime stream statistics and reoptimize queries during execution. For this the profile-controller operator was implemented. During query execution it monitors collected statistics, reoptimizes a query fragment that processes events, and switches to another strategy, e.g. into non-profiled execution. To alleviate large errors in estimates of execution plan costs in large queries, group statistics profiling was implemented in SQISLE that fragments queries into groups, measures statistics for each group, and reoptimizes the join-order of groups at runtime. Performance of queries with low selectivities was further improved by transformation rules that simplify the queries.

To verify the approach, a scientific application from the ATLAS experiment [15][47] was implemented in SQISLE. The implementation demonstrated that performance of the application analysis queries in SQISLE is close to or better than a hard-coded and manually optimized C++ implementation of the same analysis which requires a significant effort to develop.

The system currently interprets the generated query execution plans. By making a compiler of the executions plans into C or machine code, the performance will be significantly better than the current implementation.

The demonstrated performance results inspire us to implement other scientific applications in the future. We are looking for applications where analyses can be written as searching for objects using conjunctive analysis queries over streams of complex objects. Each new application requires

implementing high-level interface functions in SQISLE, which can be simplified by providing a set of high-level tools to developers.

Currently parallelization of queries in POQSEC requires specifying the degree of parallelization and the expected execution time. Future work would include estimating expected execution times using measured statistics on groups obtained by group statistics profiling. Furthermore, heuristics or learning algorithms could calculate the degree of parallelization for efficient query execution in a Grid. This could make query processing in POQSEC fully transparent for a user.

The impact of rewriting rules in SQISLE was investigated by manually rewriting application queries. Since performance evaluation demonstrated the importance of using rewriting rules, the pre-processor of SQISLE should transform the queries automatically.

Another future work is to investigate if group statistics profiling can be improved by using another fragmenting algorithm and more sophisticated cost models for groups. Currently the cost model for groups assumes that groups are uncorrelated. This assumption can lead to suboptimal join-order of groups. The impact of measuring correlations between groups could be investigated, e.g. by implementing algorithms proposed in [16] for adaptive ordering pipelined stream filters.

A challenge is to support complex queries joining several streams of time stamped complex objects.

Finally, implementing new applications in SQISLE can give more issues for future work, e.g. cases for adaptive query processing.

Summary in Swedish

Optimerad sökning bland stora mängder vetenskapliga data

Vetenskapliga instrument producerar stora volymer mätvärden. Dessa data analyseras av forskare som testar och utvärderar olika vetenskapliga teorier. Ofta är analyserna utformade med hjälp av konventionella program i ett programmeringsspråk, t.ex. C++. Sådan programmering hämmar forskningsproduktiviteten därför att det krävs mycken specialistkunskap för att skriva effektiva och bra C++-program. Dessutom är det svårt att förstå och modifiera sådana program. Programutvecklingen blir extra komplicerad eftersom programmen måste vara *skalbara*, d.v.s. de ska vara effektiva när mängden data är mycket stor.

Inom databasområdet har man sedan länge utvecklat ett flertal tekniker och system för att snabbt kunna göra avancerade frågor över stora mängder data. I denna avhandling undersöks hur tekniker som används i stora databaser också kan tillämpas för sökning och analys av vetenskapliga mätdata. Avhandlingen visar att nya databassökmetoder krävs för denna utvidgade tillämpning av databasteknik.

Följande frågeställningar tas upp:

1. Kan ett databashanteringsystem (DBHS) användas för att implementera vetenskapliga analyser? Speciellt undersöks vilka nya tekniker som behövs i ett DBHS för att möjliggöra effektiv sökning bland stora mängder vetenskapliga mätvärden.
2. Kan sökteknologi förbättra prestanda och skalbarhet för komplexa vetenskapliga analyser? Vilka nya tekniker för frågebearbetning och optimering behövs för att uppnå detta?
3. Hur kan grid-teknik användas för att utföra sådana storskaliga vetenskapliga sökningar?

Tillämpningsområdet för den föreslagna ansatsen är data och frågor från ATLAS- experimentet vid den nya LHC-acceleratorn hos CERN. I ATLAS-experimentet mäts olika fenomen producerade av mycket stora mängder kollisioner mellan partiklar. Kollisionerna kallas *händelser*. Forskare testar olika teorier för att identifiera de partiklar som producerats vid kollisionerna, t.ex. Higgs-bosoner.

I tillämpningar som ATLAS-experimentet utgörs data av stora mängder mätvärden av egenskaper hos händelser. Varje händelse är komplex, dvs. det finns mycket data om varje händelse. Man kan därför se varje händelse som en egen databas. En sådan databas kallas ett *komplext objekt*.

Avhandlingen visar att vetenskapliga teorier kan uttryckas som databasfrågor. Frågorna testar hypoteser genom att söka efter de komplexa objekt som beskriver händelser där t.ex. Higgs-bosoner skapats. Sökkriterierna uttrycks som numeriska filter i termer av olika egenskaper uppmätta vid varje händelse. Många numeriska filter kombineras i en och samma fråga när en sammansatt hypotes formuleras. Det gör att sökkriterierna sammantaget blir mycket komplexa. I tillämpningsområdet filtreras varje händelse för sig, oberoende av andra händelser. Därför behöver man aldrig göra filter som kombinerar egenskaper hos olika händelser. Detta oberoende kan utnyttjas i frågeoptimeringen.

För att undersöka hur sådana avancerade databassökningar kan hanteras generellt och effektivt har ett nytt s.k. dataströmhanteringssystem (DSHS) utvecklats som heter SQISLE (eng. Scientific Queries over Independent Streamed Large Events). Till skillnad från konventionella DBHS, som är inriktade mot effektiv sökning bland data som ligger på disk, kan man med ett DSHS som SQISLE uttrycka databasfrågor som söker direkt i stora strömmar av händelser utan att först ladda in dem i ett DBHS. Ett speciellt krav för SQISLE är att objekten i strömmarna är komplexa och att frågorna väljer ut komplexa objekt ur strömmarna m.h.a. avancerade sökkriterier.

Nya metoder har utvecklas för effektiv sökning med komplicerade sökkriterier bland strömmar av stora mängder oberoende komplexa objekt. Metoderna har utvärderats genom att tillämpa SQISLE på strömmar producerade av ATLAS-experimentet. Sökkriterierna i ATLAS-experimentet är komplicerade. Dessa sökkriterier kräver nya tekniker för effektiv och skalbar sökning.

I likhet med DBHS skapar SQISLE automatiskt en exekveringsstrategi för varje given fråga. En sådan strategi, en *frågeplan*, utför sökningen på effektivast möjliga sätt. I traditionella DBHS genererar en *frågeoptimerare* en statisk frågeplan för varje fråga, baserat på statistiska egenskaper hos de data som lagras i databasen. I avhandlingen visas att en sådan statisk frågeplan förbättrar prestanda avsevärt också för avancerade frågor över strömmande komplexa objekt. Emellertid blir databasstatistiken otillförlitlig när den används för att optimera frågor med komplicerade sökkriterier. Detta gäller för traditionella DBHS, men speciellt för DSHS eftersom statistik om data i strömmarna inte finns tillgänglig i förväg. Därför innehåller de frågeplaner som SQISLE genererar en operator som kallas profilövervakaren (eng. profile-controller). Profilövervakaren övervakar de ingående delsökkriterierna genom att samla statistik medan de körs, och anropar regelbundet frågeoptimeraren. Frågeoptimeraren genererar dynamiskt en ny frågeplan när tillräckligt mycket statistik finns tillgänglig från

profilövervakaren. Resultatet av en sådan omoptimering är att delar av frågeplanen dynamiskt byts ut under det att strömmen genomsöks.

Liksom för DBHS baseras optimeringen på en s.k. *kostnadsmodell* som uppskattar kostnaden att utföra de funktioner som används i sökkriterierna. För att optimeringen skall kunna skapa en effektiv frågeplan är det viktigt att kostnadsmodellen någorlunda korrekt uppskattar tidsåtgången för att utföra olika sökkriterier. En viktig del av sökkriterierna utgörs av aggregeringar av data, d.v.s. summeringar och andra sammanställningar. Därför utvecklar och utvärderar avhandlingen en ny kostnadsmodell för vanliga typer av aggregeringar över delar av avancerade sökkriterier. Denna *kostnadsmodell för aggregeringar* uppskattar körtid och hur mycket data som sökoperatorerna filtrerar bort. En annan viktig teknik är att automatsikt dela upp frågan i delfrågor, s.k. *grupper*, och samla statistik för varje grupp under körning. Grupper förfinar kostnadsuppskattningen för varje fråga, vilket visar sig förbättra prestanda väsentligt. Ytterligare prestandaförbättringar uppnås genom att tillämpa olika tekniker för att transformera och förenkla frågan.

I avhandlingen visas att de föreslagna frågeoptimeringsstrategierna ger prestanda i närheten av eller bättre än manuellt programmerade C++-program som utför samma analys. Vidare jämförs dataströmansatsen med den konventionella databasansatsen att först ladda upp händelserna i en databas innan man ställer frågorna. Det visas att strömansatsen är betydligt snabbare än den konventionella ansatsen.

Eftersom mängden data för vetenskapliga analyser har oerhört stor volym behövs ny infrastruktur för lagring och bearbetning av dessa data, vilket lett till utvecklandet av grid-teknik. Grid-tekniken tillhandahåller lagrings- och bearbetningsresurser för vetenskapliga analyser. Avhandlingen inkluderar en ansats, POQSEC (eng. Parallel Object Query System for Expensive Computations), som utnyttjar grid-teknik för att utföra skalbara vetenskapliga frågor över stora datavolymer genom att köra dem parallellt på många datorer i en grid-omgivning. POQSEC demonstrerar en systemarkitektur där inte bara frågorna skeppas till beräkningsnoder utan hela databashanteringssystemet.

Sammanfattningsvis visar avhandlingen:

- Att vetenskapliga analyser kan specificeras enkelt och tillämningsnära i termer av högnivåfrågor. I analysen representeras händelser som komplexa objekt modellerade m.h.a. en funktionell datamodell.
- Att med de föreslagna frågeoptimeringsmetoderna kan vetenskapliga frågor uttryckas enkelt och samtidigt utföras lika effektivt som med ett handkodat C++-program, fast betydligt mer anpassningsbart.
- Att en strömmande implementation har betydligt bättre prestanda och skalar bättre än motsvarande traditionella implementering där data laddas in i en databas innan frågorna specificeras.

- Att profilövervakaren väsentligt förbättrar prestanda genom att övervaka och och dynamiskt optimera om strömfrågorna under det att de utförs.
- Att en nyutvecklad kostnadsmodell för aggregeringar förbättrar prestanda väsentligt för komplexa frågor.
- Att uppdelning av sökkriteriet i grupper kombinerat med kostnadsmodellen för aggregeringar förbättrar prestanda väsentligt.
- Att programtransformationer signifikant förbättrar prestanda.
- Att frågor över stora datavolymer kan exekveras effektivt genom parallell körning på icke-dedicerade externa grid-resurser.

Acknowledgments

First and foremost I would like to thank my supervisor professor Tore Risch. With great help from Tore I learned a lot about databases and worked on an exciting project. I really enjoyed implementing my research in our extensible prototype Amos II.

During my time in UDBL it was a pleasure to meet the former and current members of the group: Timour Katchaounov, Kjell Orsborn, Milena Ivanova, Johan Petrini, Erik Zeitler, Sabesan Manivasakan, Silvia Stefanova, and Gyözö Gidófalvi. Our group was moving around Uppsala University. In all places within DIS and IT-department I met very nice people and I would like to thank all of them for the time.

For the study I left my home country and started a new life in Uppsala. My life was not boring, because I met a lot of new friends. The list of friends is huge, thus I do not write all their names, but I would like to thank all of them.

I am grateful to Oleg Peil, who has supported me in different activities, including all sport activities, and helped me with different troubles, e.g., a broken car.

Achieving my doctoral degree would not have been possible without undergraduate studying in LETI and all teachers who have taught me there. Furthermore, meeting Vladislav Valkovsky made my PhD study here possible.

I am grateful to my parents Mikhail and Tatiana for their support and guidance. Finally, I would like to thank my wife Oksana and daughters Katja and Lena for their love and patience.

This work was supported in part by The Swedish Research Council (VR) under contract 343-2003-955 and by Vinnova under contract 2007-02916.

A. Definition of the Six Cuts Analysis in Natural Language

This appendix presents the definition of the five cuts: *Hadr Top Cut*, *Jet Veto Cut*, *Z Veto Cut*, *Three Lepton Cut*, and *Other Cuts*. The definition fully follows the description of the analysis in [15] except that *Hadr Top Cut* is modified and *B Tag* condition was removed.

The modified *Hadr Top Cut* is defined in the following way:

- Events must have at least three *ok jets*, each with $Pt > 20$ GeV in $|Eta| < 4.5$.
- One of these three jets must be *b-tagged*, meaning that *Kf* of the jet must be equal to 5.
- Two other jets (*w jets*) of these three jets should not be b-tagged and they are selected by minimizing $|m_{jj} - m_W|$. Their invariant mass, m_{jj}, must be in the range $m_W \pm 15$ GeV.
- Among these, the three jets that are most likely to come from a top quark decay are selected by minimizing $|m_{jjj} - m_t|$, where m_{jjj} is the invariant mass of the three-jet system. This invariant mass m_{jjj} must be in the range $m_t \pm 35$ GeV.

Jet Veto Cut requires:

- Reject all events containing any jets (other than the three jets selected for the top reconstruction) with $Pt > 70$ GeV and $|Eta| < 4.5$.

Requirement of *Z Veto Cut* is:

- Reject all events with di-lepton pairs with opposite charges and the same flavor that have an invariant mass in the range $m_Z \pm 10$ GeV, where m_Z is equal to 91.1882.

Three Lepton Cut requires:

- Events must have exactly three *isolated leptons* ($l = e, \mu$) with $Pt > 20$, 7, and 7 GeV, respectively, all with $|Eta| < 2.4$.

This cut is the same as *Three Lepton Cut* from Example 2.1.

Requirements of *Other Cuts* are:

- For the three isolated leptons already selected, the *Pt* of the hardest lepton should be below 150 GeV whereas the *Pt* of the softest lepton should be below 40 GeV.
- The missing transverse energy should be large than 40 GeV.
- The effective mass, m_{eff}, constructed from the Pt_{3l} and Pt_{miss} vectors as $m_{eff} = \sqrt{2 \cdot Pt_{3l} \cdot Pt_{miss} \cdot (1 - \cos\Delta\phi)}$, is required to be lower than 150 GeV (here $\Delta\phi$ is the azimuthal angle between Pt_{3l} and Pt_{miss}).

The missing traverse energy is calculated over the missing momentum of an event by formula $\sqrt{Pxmiss^2 + Pymiss^2}$. The Pt_{3l} vector contains two elements, where the first element is the sum of *Px* values of the three isolated leptons and the second element is the sum of *Py* values for the three isolated leptons. The Pt_{miss} vector contains *Pxmiss* as the first element and *Pymiss* as the second element.

B. Definition of the Particle Schema in ALEH

The particle schema for the loading approach is implemented as a database schema representing events and their particles and views on top of the schema. The definition here is from [38] with minor changes. For example, the type *AbstractParticle* is renamed into *Particle*, the type *JetB* is renamed into *Jet*. The names of the functions are also affected by these changes. The schema is defined by:

```
create type Event;

create type Particle;

create type Lepton under Particle;

create type Jet under Particle;

create type Electron under Lepton;

create type Muon under Lepton;

create function PxMiss(Event) -> Real as stored;

create function PyMiss(Event) -> Real as stored;

create function filename(Event) -> Charstring as stored;

create function Eventid(Event) -> Integer as stored;

create function Pid(Particle) -> Integer as stored;

create function event(Particle) -> Event;

create function Kf(Particle) -> Integer as stored;

create function Px(Particle) -> Real as stored;

create function Py(Particle) -> Real as stored;
```

```
create function Pz(Particle) -> Real as stored;

create function Ee(Particle) -> Real as stored;

create function event(Muon)-> Event as stored;

create function event(Electron)-> Event as stored;

create function event(Jet)-> Event as stored;
```

The view over the schema provides functions to retrieve particles of a given event:

```
create function electrons(Event e) -> Electron el as
select el
where event(el)=e;

create function muons(Event e) -> Muon mu as
select mu
where event(mu)=e;

create function jets(Event e) -> Jet jt as
select jt
where event(jt)=e;

create function leptons(Event e) -> Lepton l as
select l
where event(l)=e;

create function particles(Event e) -> Particle p as
select p
where event(p)=e;
```

C. Definition of Analysis Cuts in ALEH

The analysis cuts from Appendix A are implemented as queries over the particle schema definition presented in Appendix B.

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

create function threeleptoncut (Event e-v) -> 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);

create function oppositeLeptons(Event e) ->  Vector of Lepton
       as
select {l1, l2}
from Lepton l1, Lepton l2
where  l1 = particles(e)
       and l2 = particles(e)
       and Kf(l1)=-Kf(l2);

create function EvInvMass(Event e) -> Bag of Real r as
select r
from Vector lept, Real inv
where  lept = oppositeLeptons(e)
       and inv = invMass(lept)
       and r=abs(inv-91.1882)
       and r < 10.0;
```

```
create function zVetoCut(Event e) -> Boolean as
select TRUE
where notany(EvInvMass(e));

create function okJetsHelpfunc(Event e) -> Bag of Jet as
select jt
from Jet jt
where  event(jt)=e
       and abs(eta(jt))<4.5
       and pt(jt)>20.0;

create function okJets(Event e) -> Bag of Jet as
select jt
from Jet jt
where  jt=okJetsHelpfunc(e)
       and atleast(3,okJetsHelpfunc(e));

create function bJets(Event e) -> Bag of Jet as
select jt
from Jet jt
where  jt=okJets(e)
       and kf(jt)=5;

create function wJets(Event e) -> Bag of Jet as
select jt
from Jet jt
where  jt=okJets(e)
       and kf(jt)!=5;

create function wPairs(Event e) -> Bag of Vector v as
select v
from Jet j1, Jet j2
where  j1=wJets(e)
       and j2=wJets(e)
       and j1>j2
       and v={j1,j2}
       and absInvMass(v,80.419)<15.0;
```

```
create function topComb(Event e) -> Bag of Vector v as
select v
from Jet j1, Jet j2, Jet bj
where  v={j1,j2,bj}
       and {j1,j2}=wPairs(e)
       and bj=bJets(e)
       and absInvMass(v,174.3)<35.0;

create function hadrtopCut(Event e) -> Boolean as
select TRUE
where some(topComb(e));

create function mTopComb(Event e) -> Vector v as
select v
where  v=topComb(e)
       and absInvMass(v,174.3) =
           minagg(absInvMass(topComb(e),174.3));

create function leftJets(Event e) -> Bag of Jet as
select jt
from Jet jt
where  jt=okJets(e)
       and notany(select jt
                  where jt=in(mTopComb(e)));

create function jetVetoCut(Event e) -> Boolean as
select  TRUE
where notany(select jt
             from Jet jt
             where   jt=leftJets(e) and
                     Pt(jt)>70.0);

create function leptonCuts(Event e) -> Boolean as
select  TRUE
where notany(select l
             from Lepton l
             where  l=isolatedLeptons(e)
                    and Pt(l)>150.0)
                    and some(select r
                             from Real r
                             where  r=Pt(isolatedLeptons(e))
                                    and r<=40.0);
```

```
create function missEeCuts(Event e) -> Boolean as
select  TRUE
from Vector ptMiss, Vector pt31
where ptMiss={PxMiss(e),PyMiss(e)}
       and mod(ptMiss)>=40.0
       and pt31={sum(select Px(isolatedLeptons(e))),
                 sum(select Py(isolatedLeptons(e)))}
       and effectiveMass(ptMiss,pt31)<=150.0;
```

D. Implementation of Stream Objects

A stream object has the following structure:

Type tag	Source	Sid	Attribute 1	Attribute 2	Attribute 3	…

The *type tag* represents the type of the object, a subtype of *Sobject*. The attribute *source* contains an object representing where the data represented by the stream object originates. The attribute *Sid* is an object uniquely identifying the data within the source. The *type tag*, *source*, and *Sid* attributes are used to uniquely identify stream objects. The system regards two stream objects as equal if this compound key is the same. This allows recognizing duplicates by comparing stream objects without maintaining an index of all currently existing stream objects.

For example, the source of a stream object representing an event in SALEH is the name of the ROOT file from which the event originates. The attribute *Sid* of an event stream object is an integer identifying the event in the source file. Thus each event is uniquely identified by the ROOT file name, the event identifier, and the fact that it is a stream object of type *Event*.

The remaining attributes *Attribute 1, 2, 3, ...* of a stream object are called *non-key attributes*. The attribute values can be objects of any type. Thus stream objects can represent complex objects. Each kind of stream object has a fixed number of attributes.

SQISLE provides internal interface functions for stream objects. They are not used in user queries but to define the application schema by a SQISLE administrator. There are functions to create new stream objects and to access and update of *Sobject* attributes. A new stream object is constructed by calling a function:

```
new_sobject(Type tpo,
            Object source,
            Object sid,
            Vector values) -> Sobject so
```

The result is a stream object *so* belonging to the given type *tpo*, having the given source object *source* and identifying object *sid*. It has the same number of non-key attributes as the size of the vector *values*. The values of the non-

key attributes are set to the values from the vector *values* in the same order as its elements. It is also possible to create a stream object without providing values. In this case an overloaded function is used:

```
new_sobject(Type tpo,
            Object source,
            Object sid,
            Integer n) -> Sobject so
```

In the overloaded function the number of non-key attributes *n* is specified rather than the vector of values. The non-key attribute values are initialized to *nil*. The values can be later updated by a function:

```
set_slot(Sobject so,
         Integer i,
         Object v) -> Boolean b
```

The function takes a stream object *so*, a non-key attribute position *i*, and a value object *v* as arguments, and always return *true*.

Values of non-key attributes are accessed by calling a function:

```
get_slot(Sobject so, Integer i) -> Object v
```

For a given stream object *so* and attribute position *i,* the function returns the stored value *v*. If a slot contains a collection of values the values can alternatively be returned as a bag *b* by the function:

```
get_slot_bag(Sobject so, Integer i) -> Bag of Object b
```

The function has the same arguments as *get_slot*, but if the accessed value is a vector the values of the vector are emitted one by one as a bag of values. The source and identifier of a stream object *so* are retrieved by calling the following operators over the given stream object:

```
source(Sobject so) -> Object source
sid(Sobject so) -> Integer sid
```

Appendix G shows how to use these interface functions to define the particle schema in SALEH.

E. The ROOT Wrapper Interface

A general ROOT wrapper interface is implemented as a set of functions, the *ROOT wrapper interface functions* to retrieve events from ROOT files that store events as ROOT tuples containing simple C values (Section 2.1.1). The interface provides functions to access meta-data about a ROOT file and functions to return from a ROOT file a stream of tuples represented by stream objects. The meta-data about a ROOT file includes internal paths inside the ROOT file and names of ROOT collections stored in the file. Furthermore it provides meta-data that describes the structure of the ROOT tuples in each ROOT collection as lists of the names and types of the tuple attributes. The ROOT wrapper interface is used by a SQISLE administrator for defining the particle schema.

The ROOT wrapper interface functions can stream all tuples of a collection, a subset of the tuples, or a single tuple. To retrieve data from a ROOT collection a ROOT wrapper interface function needs at least the name of the ROOT file, the internal path to a collection in the ROOT file, the collection name, and the type of the result stream objects. The result type must be subtype of the type *Sobject*. The sources of the stream objects are defined by the name of the ROOT file. The *Sid*s of the stream objects are the identifiers of the corresponding ROOT tuples. The path to the collection and the collection name are stored as meta-properties of their type and not in the result stream objects. The attributes of a stream object contains values of tuple elements in the same order as in the corresponding tuple indexed by a *slot* number for each attribute.

Since not all attributes are needed for an analysis, unnecessary attributes of the tuples can be projected away in the corresponding stream objects. These projections are specified as an input vector of projected attribute names, called the *projection vector*. The emitted stream objects contain the same number of attributes as the size of the projection vector and the attributes are ordered in the same way as in the projection vector.

The definition of the particle view uses the following ROOT wrapper interface function:

```
root_scan_project(Charstring file,
                  Charstring path,
                  Charstring collection,
                  Vector projections,
                  Type stype) -> Stream of Sobject so
```

The function returns all tuples of a given *collection* stored under a given *path* in a given ROOT *file*. The result of the ROOT wrapper interface function is a stream of objects *so* of type *stype* containing values of attributes from the projection vector *projections*.

In the SALEH application all events in the application are stored in ROOT files using the internal path */ATLFAST* and the collection name *h51*. All stream objects representing the ROOT tuples belong to the same type *Event*, which is subtype of *Sobject* (Figure 4.3). Therefore a view function *saleh_events* is defined in terms of the ROOT wrapper interface function *root_scan_project*:

```
create function saleh_events (Charstring filename) ->
                  Bag of Event e as
select e
where e in root_scan_project
               (filename,"/ATLFAST","h51",
                {"Pxmiss","Pymiss","Kfele","Pxele","Pyele",
                 "Pzele","Eeele","Kfmuo","Pxmuo","Pymuo",
                 "Pzmuo","Eemuo","Kfjetb","Pxjetb","Pyjetb",
                 "Pzjetb","Eejetb"},
                typenamed("Event"));
```

The function *saleh_events* takes a ROOT file name as the parameter and returns bag of stream objects of type *Event* representing ROOT tuples in the file. Since in the SALEH all events in the ROOT files are stored in collections named *h51* under path */ATLFAST* these values are specified as constants parameters to the function *root_scan_project*. The projection vector of attributes is also given as a constant and it contains only those attributes that are needed for defining the particle schema. The result type is given by calling the function *typenamed("Event")*, which returns the SQISLE object representing the type named *Event*.

To be able to stream only part of a ROOT file the following interval function is defined in terms of corresponding general ROOT wrapper interface function:

```
saleh_events(Charstring filename,
             Integer firstEvent,
             Integer lastEvent)-> Bag of Event e
```

The interval function *saleh_events* retrieves all events from a ROOT file with *filename* within an interval [*firstEvent*,*lastEvent*].

To enable materializations in event objects, the following wrapper interface function is implemented:

```
root_scan_project_addslots(Charstring file,
                           Charstring path,
                           Charstring collection,
                           Vector projections,
                           Type stype,
                           Integer slots) ->
                                        Stream of Sobject so
```

It creates stream objects *so* having *slots* additional slots used for materializing derived particle objects. Thus the actual number of non-key attributes in the stream objects *so* is the size of the projection vector *projections* plus the number of additional arguments *slots*.

F. The Transformation Views in SALEH

The particle schema (Figure 2.3) contains particles of different kinds derived from the tuples representing events in ROOT files. For each tuple a corresponding stream object of type *Event,* called an *event object,* is constructed by a ROOT wrapper interface function. The objects representing particles are not explicitly stored in the event objects. Instead, attribute values of particles are derived from several attributes of an event object as *transformation views* over the event objects. The transformation views create stream objects that represent particles derived from the event objects. The particle schema is defined in terms of the transformation views. Each kind of particle is specified by derived functions constructing stream objects as a separate transformation view for each kind of particle. For example, the set of electrons for a given event is defined by the transformation function *new_electrons* in terms of the function *new_sobject*:

```
create function new_electrons(Event e)-> Bag of Electron el as
select el
from Integer i
where el=new_sobject(typenamed("Electron"), e, i,
                     {Kfele(e)[i],Pxele(e)[i],Pyele(e)[i],
                      Pzele(e)[i],Eeele(e)[i]});
```

Electrons are constructed from the event attribute vectors *Kfele, Pxele, Pyele, Pzele, Eeele,* where each *attribute vector* represents a particular attribute of all the electrons belonging to the given event. For example, energy values of each electron in an event are stored in the event attribute vector *Eeele* indexed by an integer *ei* identifying the electron within the event. The same integer identifier is used in all these electron vector attributes of an event. The function *new_electrons* creates as many electrons as the size of the vectors.

The sets of stream objects representing muons and jets are created analogously:

```
create function new_mouns(Event e)-> Bag of Muon mu as
select mu
from Integer i
where mu=new_sobject(typenamed("Muon"), e, i,
                     {Kfmuo(e)[i],Pxmuo(e)[i],Pymuo(e)[i],
                      Pzmuo(e)[i],Eemuo(e)[i]});

create function new_jets(Event e)-> Bag of Jet jt as
select jt
from Integer i
where jt=new_sobject(typenamed("Jet"), e, i,
                     {Kfjetb(e)[i],Pxjetb(e)[i],
                      Pyjetb(e)[i],Pzjetb(e)[i],
                      Eejetb(e)[i]});
```

G. The Particle Schema Definition in SALEH

The particle schema (Figure 2.3) is defined in terms of stream objects, which are created by calling the ROOT wrapper interface functions and the transformation functions. The types defined in the particle schema are subtypes of the type *Sobject* as in Figure 4.3. The collection of events from a file is defined by the function *saleh_events(Charstring filename)-> Bag of Event e*. Each stream object *e* of type *Event* contains source and identifier for the event, and 17 other attributes representing event values. The particle schema contains four event attributes, which are defined by four public functions having the following signatures:

```
filename(Event e) -> Charstring filename
eventid(Event e) -> Integer eventid
pxmiss(Event e) -> Real pxmiss
pymiss(Event e) -> Real pymiss
```

To access the other event attributes private functions are provided with signatures, for example:

```
Kfele(Event e) -> Vector kfele
Pxele(Event e) -> Vector pxele
Pyele(Event e) -> Vector pyele
Pzele(Event e) -> Vector pzele
Eeele(Event e) -> Vector eeele
```

These functions are used internally in the transformation view definitions.

Particle objects from the particle schema are derived from events by the transformation views. Thus to define a specific particle type (*Electron*, *Muon*, or *Jet*) of the particle schema it is necessary to provide:

1. The attributes of the particle type.
2. The particle objects derived from the event object.
3. The event objects from which the particle objects are derived.

The attributes are implemented by functions of type *Particle* to access slots in the created particle objects:

```
event(Particle) -> Event e
pid(Particle) -> Integer pid
kf(Particle) -> Integer kf
px(Particle) -> Real px
py(Particle) -> Real py
```

```
pz(Particle) -> Real pz
ee(Particle)->Real ee
```

Items 2 and 3 define the relationship between stream objects of type *Event* and stream objects of the different specific kinds of particles. This is implemented as a multi-directional function that accesses the source in one direction and creates all new particles of an event in the other direction. For example, the relationship between electrons and event is defined by:

```
create function electrons(Event e) -> Bag of Electron el as
       multidirectional
("bf" select new_electrons(e))
("fb" select event(el));
```

The function *electrons* is executed either in forward or in inverse directions using different implementations. In the forward direction, noted by binding pattern *bf*, for the given event *e* the function *new_electrons* constructing electrons of the event is called. In the inverse direction, noted by binding pattern *fb*, for the given electron *e* the function event is called, which returns the event object of the electron.

Analogously the relationships between type *Event* and types *Muon* and *Jet* are defined by the functions:

```
create function muons(Event e) -> Bag of Muon mu as
       multidirectional
("bf" select new_muons(e))
("fb" select event(mu));

create function jets(Event e) -> Bag of Jet jt as
       multidirectional
("bf" select new_jets(e))
("fb" select event(jt));
```

The relationships between type *Event* and the types *Lepton* and *Particle* are specified as unions of their subtypes by the functions:

```
create function leptons(Event e)-> Bag of Lepton l as
       multidirectional
 ("bf" select l
       where  l=electrons(e) or l=muons(e);
("fb" select event(l));

create function particles(Event e)-> Bag of Particle p as
       multidirectional
```

```
 ("bf" select p
       where  p=leptons(e) or p=jets(e);
("fb" select event(p));
```

The following function is used for selecting events in user queries:

```
create function saleh_events (Charstring filename) ->
                  Bag of Event e as
select e
where e in root_scan_project
                (filename,"/ATLFAST","h51",
                 {"Pxmiss","Pymiss","Kfele","Pxele","Pyele",
                  "Pzele","Eeele","Kfmuo","Pxmuo","Pymuo",
                  "Pzmuo","Eemuo","Kfjetb","Pxjetb","Pyjetb",
                  "Pzjetb","Eejetb"},
                 typenamed("Event"));
```

The following functions access event attributes of the view in user queries:

```
create function filename(Event e) -> Charstring as
select filename
from Charstring filename
where filename=source(e);

create function eventid(Event e) -> Integer as
select id
from Integer id
where id=id(e);

create function pxmiss(Event e) -> Real as
get_slot(e,0);

create function pymiss(Event e) -> Real as
get_slot(e,1);
```

The following functions access the event attribute vectors in transformation queries:

```
create function Kfele(Event e) -> Vector of Integer as
get_slot(e,2);

create function Pxele(Event e) -> Vector of Real as
get_slot(e,3);
```

```
create function Pyele(Event e) -> Vector of Real as
get_slot(e,4);

create function Pzele(Event e) -> Vector of Real as
get_slot(e,5);

create function Eeele(Event e) -> Vector of Real as
get_slot(e,6);

create function Kfmuo(Event e) -> Vector of Integer as
get_slot(e,7);

create function Pxmuo(Event e) -> Vector of Real as
get_slot(e,8);

create function Pymuo(Event e) -> Vector of Real as
get_slot(e,9);

create function Pzmuo(Event e) -> Vector of Real as
get_slot(e,10);

create function Eemuo(Event e) -> Vector of Real as
get_slot(e,11);

create function Kfjetb(Event e) -> Vector of Integer as
get_slot(e,12);

create function Pxjetb(Event e) -> Vector of Real as
get_slot(e,13);

create function Pyjetb(Event e) -> Vector of Real as
get_slot(e,14);

create function Pzjetb(Event e) -> Vector of Real as
get_slot(e,15);

create function Eejetb(Event e) -> Vector of Real as
get_slot(e,16);
```

The following transformation functions define particles of the different kinds:

```
create function new_electrons(Event e)-> Bag of Electron el as
select el
```

```
from Integer i
where el=new_sobject(typenamed("Electron"), e, i,
                     {Kfele(e)[i],Pxele(e)[i],Pyele(e)[i],
                      Pzele(e)[i],Eeele(e)[i]});

create function new_mouns(Event e)-> Bag of Muon mu as
select mu
from Integer i
where mu=new_sobject(typenamed("Muon"), e, i,
                     {Kfmuo(e)[i],Pxmuo(e)[i],Pymuo(e)[i],
                      Pzmuo(e)[i],Eemuo(e)[i]});

create function new_jets(Event e)-> Bag of Jet jt as
select jt
from Integer i
where jt=new_sobject(typenamed("Jet"), e, i,
                     {Kfjetb(e)[i],Pxjetb(e)[i],
                      Pyjetb(e)[i],Pzjetb(e)[i],
                      Eejetb(e)[i]});
```

The following function access attributes of particles:

```
create function kf(Particle p)-> Integer as
get_slot(p,0);

create function px(Particle p)-> Real as
get_slot(p,1);

create function py(Particle p)-> Real as
get_slot(p,2);

create function pz(Particle p)-> Real as
get_slot(p,3);

create function ee(Particle p)-> Real as
get_slot(p,4);
```

This completes the definition of the particle schema in SQISLE.

H. SQISLE Utility Functions

Some utility functions are provided in SQISLE to provide more elegant and general ways to define scientific queries. For example, some functions in SQISLE return tuples rather than anonymous vectors. These tuples are assigned to single values by converting them to vectors.

For example, the function *oppositeLeptons(event)->vector* was defined in ALEH (Appendix C) as:

```
create function oppositeLeptons(Event e) ->  Vector as
select {l1, l2}
where  l1 = particles(e) and
       l2 = particles(e) and
       kf(l1)=-kf(l2);
```

In SQISLE it is reformulated to return a tuple instead of a vector as:

```
create function oppositeLeptons(Event e) ->
                 <Lepton l1, Lepton l2> as
select l1, l2
where  l1 = particles(e) and
       l2 = particles(e) and
       kf(l1)=-kf(l2);
```

This definition demonstrates the semantic of the function better than the original definition, which returned an anonymous vector.

Another extension is some general aggregate functions to deal with tuples of multiple values. For example, a new general aggregation function *minagg2* over bags of tuples is introduced. It operates on bags of pairs, where the first element of a tuple is expected to be the value that is minimized and the second a corresponding property value. With *minagg2* the definition of the function *mTopComb(event)->Vector* in Appendix C can be simplified. The original definition of *mTopComb* in ALEH is:

```
create function mTopComb(Event e) -> Vector v as
select v
where  v=topComb(e) and
       absInvMass(v,174.3) =
```

```
        minagg(select absInvMass(tc,174.3)
                from Vector tc
                where tc = topComb(e));
```

The function *mTopComb* finds the triple of jets that has smallest invariant mass among all triple combinations of jets and is produced by *topComb*. Without the general *minagg2* the function *topComb* has to be called twice. First it is called to find the smallest value of invariant mass among the triples. Then it is called to choose a triple that has the invariant mass equal to the smallest value found by the nested subquery. With the more general *minagg2* operating on bags of tuples the function *mTopComb* is simplified in SALEH:

```
create function mTopComb(Event e) -> Vector v as
select v
from Real r
where  <r,v> = minagg2(select absInvMass(tc,174.3), tc
                       from Vector tc
                       where tc = topComb(e));
```

In this case the function *topComb*, which generates the triples, is called only once, while in the original implementation of *mTopComb* it is called twice.

I. Definitions of Analysis Cuts in SALEH

Cuts from the analyses described in Appendix A and in Example 2.1 are defined here over the particle schema definition in SQISLE.

First, the cuts from the analyses described in Appendix A are presented:

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

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);

create function oppositeLeptons(Event e) ->
                 Bag of <Lepton l1, Lepton l2> as
select l1, l2
where  l1 = leptons(e)
       and l2 = leptons(e)
       and kf(l1) = -Kf(l2)
       and kf(l1) > 0;

create function EvInvMass(Event e) -> Bag of Real r as
select r
from Vector v
where  v = oppositeLeptons(e)
       and r = absinvMass(v,91.1882)
       and r < 10.0;
```

```
create function zVetoCut(Event e) -> Boolean as
select TRUE
where notany(EvInvMass(e));

create function okJets(Event e) -> Bag of Jet as
select jt
from Jet jt
where  jt = jets(e)
       and abs(eta(jt)) < 4.5
       and pt(jt) > 20.0;

create function bJets(Event e) -> Bag of Jet as
select jt
from Jet jt
where  jt = okJets(e)
       and kf(jt) = 5;

create function wJets(Event e) -> Bag of Jet as
select jt
from Jet jt
where  jt = okJets(e)
       and kf(jt) != 5;

create function wPairs(Event e) -> Baf of <Jet j1, Jet j2> as
select j1, j2
from Vector v
where  j1 = wJets(e)
       and j2 = wJets(e)
       and v = <j1,j2>
       and absInvMass(v, 80.419) < 15.0
       and j1 > j2;

create function topComb(Event e) ->
                Bag of <Jet j1, Jet j2, Jet bj> as
select j1, j2, bj
from Vector v
where  <j1,j2> = wPairs(e)
       and bj = bJets(e)
       and v = <j1,j2,bj>
       and absInvMass(v, 174.3) < 35.0;

create function hadrtopCut(Event e) -> Boolean as
select TRUE
where some(topComb(e));
```

```
create function mTopComb(Event e) -> Vector v as
select v
from Real r
where <r,v> = minagg2(select rt, vt
                      from Real rt, Vector vt
                      where  vt = topComb(e)
                             and rt = absInvMass(vt,174.3));

create function leftJets(Event e) -> Bag of Jet jt as
select jt
where  jt = okJets(e)
       and not_in(jt, mTopComb(e));

create function jetVetoCut(Event e) -> Boolean as
select TRUE
where notany(select jt
             from Jet jt
             where  jt = leftJets(e)
                    and Pt(jt) > 70.0);

create function leptonCuts(Event e) -> Boolean as
select  TRUE
where notany(select l
             from Lepton l
             where  l = isolatedLeptons(e)
                    and Pt(l) > 150.0)
       and some( select r
                 from Real r
                 where  r = Pt(isolatedLeptons(e))
                        and r <= 40.0);

create function missEeCuts(Event e) -> Boolean as
select TRUE
from Real x, Real y
where  modulo(PxMiss(e), PyMiss(e)) >= 40.0
       and <x,y> = sum2( select Px(l),Py(l)
                         from lepton l
                         where  l = isolatedleptons(e))
       and effectiveMass(PxMiss(e),PyMiss(e),x,y) <= 150.0;
```

The definitions of the cuts from Example 2.1 are:

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

create function threeLeptonCut(Event e) -> Boolean as
select TRUE
where  count(select isolatedLeptons(e))=3 and
       some( select l
             from Lepton l
             where  l=isolatedLeptons(e) and
                    pt(l)>20.0);

create function TwoLeptonCut(Event e) -> Boolean as
select TRUE
where some( select l1,l2
            from Real r, Lepton l1, Lepton l2, Vector v
            where  r = invMass(v) and
                   v = <l1,l2> and
                   l1 = isolatedleptons(e) and
                   l2 = isolatedleptons(e) and
                   kf(l1) = -kf(l2) and
                   r > 10 and
                   r < 63 and
                   kf(l1) > 0);

create function okJets(Event e) -> Bag of Jet jt as
select jt
where  pt(jt) > 20.0
       and eta(jt) < 4.5
       and jt = jets(e);

create function wPair(Event e) -> Bag of <Jet j1, Jet j2> as
select j1, j2
from Vector v
where  j1 = okJets(e)
       and j2 = okJets(e)
       and v = <j1,j2>
       and j1 > j2
       and absInvMass(v, 80.419) < 15.0;
```

```
create function threeJets(Event e) ->
                Bag of <Jet j1, Jet j2, Jet bj> as
select j1, j2, bj
from Vector v
where  <j1,j2> = wPair(e)
       and bj != j1
       and bj != j2
       and v = <j1,j2,bj>
       and absInvMass(v, 174.3) < 35.0
       and bj = okJets(e);

create function topCut(Event e) -> Boolean as
select TRUE
where  some(select threeJets(e));

create function jetCut(Event e) -> Boolean as
select TRUE
where 300 >
       sum(select pt(jt)
            from Jet jt, Vector v, Real r
            where not_in(jt,v) and
                  <r,v> = minagg2(select rt,vt
                                  from Real rt, Vector vt
                                  where vt = threeJets(e) and
                                     rt = absInvMass(vt,174.3))
                   and pt(jt)>50
                   and jt=jets(e));
```

J. The Stream Fragmenting Algorithm

The stream fragmenting algorithm fragments an analysis query into groups and assigns every group either to a source access query fragment or to a processing query fragment (Section 4.4). Every group is formed by functions that have variables in common except an event variable, which is the result variable of a wrapper interface function and is bound to a stream of events. The stream fragmenting algorithm is a modification of Algorithm 3.1 with ability to check if a formed group calls a wrapper interface function. The pseudo code of the algorithm is:

```
 1:  Groups = {}
 2:  while (S != {})
 3:    pick a predicate p from S
 4:    S = S \ p
 5:    G = {p}
 6:    if p is a wrapper interface function
 7:    then G.isWrap = true
 8:    else G.isWrap = false
 9:    V = variables(p) \ varE
10:    while (V !={})
11:      pick a variable v from V
12:      V = V \ v
13:      for each q in S
14:        if v ∈ variables(q)
15:        then G = G∪q
16:             S = S \ q
17:             V = V∪variables(q)\{v,varE}
18:             if q is a wrapper interface function
19:             then G.isWrap = true
20:    Groups = Groups ∪ {G}
21:  return Groups
```

In the new algorithm each group has a flag *isWrap* that indicates if the group calls a wrapper interface function or not. On lines (6-8) the flag of a created group is initialized with true or false depending whether or not the function *p* is a wrapper interface function. The rest of the predicates in the

group are added on lines (14-19), and, if a function *q* is a wrapper interface function, then the flag *isWrap* is set to true for the group on lines (18-19).

After grouping by the stream fragmenting algorithm the source access query fragment is constructed by merging predicates from the groups that contain wrapper interface functions. Predicates from groups that do not contain wrapper interface functions form the processing query fragment.

Bibliography

[1] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma and Jennifer Widom. STREAM: The Stanford Stream Data Manager. In *IEEE Data Eng. Bull.*, volume 26, 19-26, 2003.

[2] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. In *VLDB J.*, volume 12, 120-139, 2003.

[3] ADA – ATLAS Distributed Analysis. http://www.usatlas.bnl.gov/ADA/.

[4] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference*, 261-272, 2000.

[5] Vasco Amaral, Sven Helmer and Guido Moerkotte. PHEASANT: A PHysicist's EAsy ANalysis Tool. In *FQAS*, 229-242, 2004.

[6] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zat and Thierry Cruanes. Cost-Based Query Transformation in Oracle. In *VLDB*, 1026-1036, 2006.

[7] The ATLAS Experiment. http://atlasexperiment.org/

[8] The ATLAS TWiki. https://twiki.cern.ch/twiki/bin/view/Atlas/WebHome

[9] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS*, 1-16, 2002.

[10] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani and Dilys Thomas. Operator scheduling in data stream systems. In *VLDB J.*, volume 13, 333-353, 2004.

[11] Pedro Bizarro, Shivnath Babu, David J. DeWitt and Jennifer Widom. Content-Based Routing: Different Plans for Different Data. In *VLDB*, 757-768, 2005.

[12] Nicolas Bruno and Surajit Chaudhuri. Conditional Selectivity for Statistics on Query Expressions. In *SIGMOD Conference*, 311-322, 2004.

[13] Nicolas Bruno, Surajit Chaudhuri and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. In *SIGMOD Conference*, 211-222, 2001.

[14] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa and Jennifer Widom. Adaptive Ordering of Pipelined Stream Filters. In *SIGMOD Conference*, 407-418, 2004.

[15] M. Bisset, F. Moortgat and S. Moretti. Trilepton+top signal from chargino-neutralino decays of MSSM charged Higgs bosons at the LHC. In *European Physical Journal C*, volume 30, 419-434, 2003.

[16] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa and Jennifer Widom. Adaptive Ordering of Pipelined Stream Filters. In *SIGMOD Conference*, 407-418, 2004.

[17] Jihad Boulos and Kinji Ono. Cost Estimation of User-Defined Methods in Object-Relational Database Systems. In *SIGMOD Record*, volume 28, 22-28, 1999.
[18] Rene Brun and Fons Rademakers. ROOT - An Object Oriented Data Analysis Framework. In *AIHENP'96 Workshop, Nucl. Inst. & Meth. in Phys. Res. A 389*, 81-86, 1997. See also http://root.cern.ch.
[19] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. In *SIGMOD Record*, volume 26, 65-74, 1997.
[20] Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. In *ACM Trans. Database Syst.*, volume 1, 9-36, 1976.
[21] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD Conference*, 647-651, 2003.
[22] Maria Cláudia Cavalcanti, Marta Mattoso, Maria Luiza Machado Campos, Eric Simon and François Llirbat. An Architecture for Managing Distributed Scientific Resources. In *SSDBM*, 47-58, 2002.
[23] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. In *Commun. ACM*, volume 13, 377-387, 1970.
[24] E. F. Codd. Relational Database: A Practical Foundation for Productivity. In *Commun. ACM*, volume 25, 109-117, 1982.
[25] Surajit Chaudhuri and Kyuseok Shim. Optimizing Queries with Aggregate Views. In *EDBT*, 167-182, 1996.
[26] Surajit Chaudhuri and Kyuseok Shim. Optimization of Queries with User-Defined Predicates. In *ACM Trans. Database Syst.*, volume 24, 177-228, 1999.
[27] Database Languages - SQL, ISO/IEC 9075-*:2003
[28] DIAL – Distributed Interactive Analysis of Large datasets. http://www.usatlas.bnl.gov/~dladams/dial/.
[29] Amol Deshpande, Zachary G. Ives and Vijayshankar Raman. Adaptive Query Processing. In *Foundations and Trends in Databases*, volume 1, 1-140, 2007.
[30] Amol Deshpande and Samuel Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD Conference*, 73-84, 2006.
[31] Paula Eerola, Tord Ekelöf, Mattias Ellert, John Renner Hansen, Aleksandr Konstantinov, Balázs Kónya, Jakob Langgaard Nielsen, Farid Ould-Saada, Oxana Smirnova and Anders Wäänänen. Science on NorduGrid. In *ECCOMAS*, 2004. See also http://www.nordugrid.org.
[32] Mattias Ellert, Michael Grønager, Aleksandr Konstantinov, Balázs Kónya, J. Lindemann, I. Livenson, Jakob Langgaard Nielsen, Marko Niinimäki, Oxana Smirnova and Anders Wäänänen. Advanced Resource Connector middleware for lightweight computational Grids. In *Future Generation Comp. Syst.*, volume 23, 219-240, 2007. See also http://www.nordugrid.org/.
[33] Mattias Ellert. The NorduGrid Brokering Algorithm. 2004. Available at http://www.nordugrid.org/documents/brokering.pdf.
[34] Cristian Estan and Jeffrey F. Naughton. End-biased Samples for Join Cardinality Estimation. In *ICDE*, 20, 2006.
[35] Ian Foster and Carl Kesselman (ed.). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2003.
[36] Ruslan Fomkin and Tore Risch. Managing Long Running Queries in Grid Environment. In *OTM Workshops*, 99-110, 2004.
[37] Ruslan Fomkin and Tore Risch. Framework for Querying Distributed Objects Managed by a Grid Infrastructure. In *DMG*, 58-70, 2005.

[38] Ruslan Fomkin and Tore Risch. Cost-based Optimization of Complex Scientific Queries. In *SSDBM*, 1, 2007.
[39] César A. Galindo-Legaria, Milind Joshi, Florian Waas and Ming-Chuan Wu. Statistics on Views. In *VLDB*, 952-962, 2003.
[40] Peter M. D. Gray, Larry Kerschberg, Peter J. H. King and Alexandra Poulovassilis. *The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data*. Springer, 2004.
[41] The Globus Alliance. http://www.globus.org/.
[42] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan and Samuel Madden. XStream: a Signal-Oriented Data Stream Management System. In *ICDE*, 1180-1189, 2008.
[43] Anastasios Gounaris, Rizos Sakellariou, Norman W. Paton and Alvaro A. A. Fernandes. A novel approach to resource scheduling for parallel query processing on computational grids. In *Distributed and Parallel Databases*, volume 19, 87-106, 2006.
[44] Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer D. Widom. *Database Systems: The Complete Book, Second Edition*. Prentice Hall, 2008.
[45] Richard A. Ganski and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. In *SIGMOD Conference*, 23-33, 1987.
[46] Goetz Graefe and Karen Ward. Dynamic Query Evaluation Plans. In *SIGMOD Conference*, 358-366, 1989.
[47] 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. In *European Physical Journal C*, volume 44, 1-9, 2005.
[48] Joseph M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. In *ACM Trans. Database Syst.*, volume 23, 113-157, 1998.
[49] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman and Hamid Pirahesh. Extensible Query Processing in Starburst. In *SIGMOD Conference*, 377-388, 1989.
[50] Zhen He, Byung Suk Lee and Robert R. Snapp. Self-tuning cost modeling of user-defined functions in an object-relational DBMS. In *ACM Trans. Database Syst.*, volume 30, 812-853, 2005.
[51] Joseph M. Hellerstein and Jeffrey F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *SIGMOD Conference*, 423-434, 1996.
[52] Joseph M. Hellerstein and Michael Stonebraker. Anatomy of a Database System. In *Joseph M. Hellerstein and Michael Stonebraker (ed.) Readings in Database Systems: Fourth Edition*. MIT Press, 2005.
[53] Joseph M. Hellerstein and Michael Stonebraker (ed.). *Readings in Database Systems: Fourth Edition*. MIT Pres, 2005.
[54] Yannis E. Ioannidis and Stavros Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD Conference*, 268-277, 1991.
[55] Zachary G. Ives, Alon Y. Halevy and Daniel S. Weld. Adapting to Source Properties in Processing Data Integration Queries. In *SIGMOD Conference*, 395-406, 2004.
[56] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized Algorithms for Optimizing Large Join Queries. In *SIGMOD Conference*, 312-321, 1990.
[57] Yannis E. Ioannidis. Query Optimization. In *Allen B. Tucker (ed.) Computer Science Handbook, Second Edition*. CRC Press, 2004.
[58] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. In *ACM Comput. Surv.*, volume 16, 111-152, 1984.

[59] Neil D. Jones. An Introduction to Partial Evaluation. In *ACM Comput. Surv.*, volume 28, 480-503, 1996.
[60] Ravi Krishnamurthy, Haran Boral and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *VLDB*, 128-137, 1986.
[61] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss and Mehul A. Shah. TelegraphCQ: An Architectural Status Report. In *IEEE Data Eng. Bull.*, volume 26, 11-18, 2003.
[62] Bhargav Kanagal and Amol Deshpande. Online Filtering, Smoothing and Probabilistic Modeling of Streaming data. In *ICDE*, 1160-1169, 2008.
[63] Won Kim. On Optimizing an SQL-like Nested Query. In *ACM Trans. Database Syst.*, volume 7, 443-469, 1982.
[64] Steven Lynden, Arijit Mukherjee, Alastair C. Hume, Alvaro A.A. Fernandes, Norman W. Paton, Rizos Sakellariou and Paul Watson. The design and implementation of OGSA-DQP: A service-based distributed query processor. In *Future Generation Computer Systems*, 2008.
[65] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby and Guy M. Lohman. Adaptively Reordering Joins during Query Execution. In *ICDE*, 26-35, 2007.
[66] Witold Litwin and Tore Risch. Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates. In *IEEE Trans. Knowl. Data Eng.*, volume 4, 517-528, 1992.
[67] L. Daniel Massey. *Probability and Statistics*. McGraw-Hill, 1971.
[68] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman and Hamid Pirahesh. Robust Query Processing through Progressive Optimization. In *SIGMOD Conference*, 659-670, 2004.
[69] Arunprasad P. Marathe and Kenneth Salem. Query processing techniques for arrays. In *VLDB J.*, volume 11, 68-91, 2002.
[70] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, 49-60, 2002.
[71] Joakim Näs. Randomized optimization of object oriented queries in a main memory database management system. *Master's Thesis*, LiTH-IDA-Ex-93/25, 1993. Available at http://user.it.uu.se/~udbl/Theses/JoakimNasMSc.pdf.
[72] Sivaramakrishnan Narayanan, Tahsin M. Kurç, Ümit V. Çatalyürek and Joel H. Saltz. Database Support for Data-Driven Scientific Applications in the Grid. In *Parallel Processing Letters*, volume 13, 245-271, 2003.
[73] The NorduGrid/ARC User Guide. 2005. Available at http://www.nordugrid.org/documents/userguide.pdf.
[74] William O'Mullane, Nolan Li, Mara A. Nieto-Santisteban, Alexander S. Szalay and Ani Thakar. Batch is Back: CasJobs, Serving Multi-TB Data on the Web. In *ICWS*, 33-40, 2005.
[75] The Open Grid Forum (OGF). http://www.ogf.org/
[76] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.
[77] Johan Petrini. Querying RDF Schema Views of Relational Databases. In *Uppsala Dissertations from the Faculty of Science and Technology,* 75, 2008.
[78] Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *VLDB*, 486-495, 1997.

[79] Tore Risch, Vanja Josifovski and Timour Katchaounov. Functional data integration in a distributed mediator system. In *The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data*. SpringerVerlag, 2003.

[80] Vasco Sousa, Vasco Amaral and Bruno Barroca. Towards a full implementation of a robust solution of a domain specific visual query language for HEP physics analysis. In *IEEE Nuclear Science Symposium Conference Record*, volume 1, 910-917, 2007.

[81] Vincius F. V. da Silva, Márcio L. Dutra, Fabio Porto, Bruno Schulze, Álvaro Cesar P. Barbosa and Jauvane C. de Oliveira. An adaptive parallel query processing middleware for the Grid. In *Concurrency and Computation: Practice and Experience*, volume 18, 621-634, 2006.

[82] Arun N. Swami and Anoop Gupta. Optimization of Large Join Queries. In *SIGMOD Conference*, 8-17, 1988.

[83] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A. A. Fernandes and Rizos Sakellariou. Distributed Query Processing on the Grid. In *GRID*, 279-290, 2002

[84] Utkarsh Srivastava, Peter J. Haas, Volker Markl, Marcel Kutsch and Tam Minh Tran. ISOMER: Consistent Histogram Construction Using Query Feedback. In *ICDE*, 39, 2006.

[85] Praveen Seshadri, Miron Livny and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *VLDB*, 99-110, 1996.

[86] Oxana Smirnova. Extended Resource Specification Language Reference Manual. 2005. Available at http://www.nordugrid.org/documents/xrsl.pdf.

[87] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD Conference*, 23-34, 1979.

[88] Timos K. Sellis and Leonard D. Shapiro. Query Optimization for Nontraditional Database Applications. In *IEEE Trans. Software Eng.*, volume 17, 77-86, 1991.

[89] Michael Stonebraker, Lawrence A. Rowe, Bruce G. Lindsay, Jim Gray, Michael J. Carey, Michael L. Brodie, Philip A. Bernstein and David Beech. Third-Generation Database System Manifesto - The Committee for Advanced DBMS Function. In *SIGMOD Record*, volume 19, 31-44, 1990.

[90] Swegrid. http://www.swegrid.se/.

[91] Alexander S. Szalay. The Sloan Digital Sky Survey and beyond. In *SIGMOD Record*, volume 37, 61-66, 2008.

[92] Marko Vrhovnik, Holger Schwarz, Oliver Suhre, Bernhard Mitschang, Volker Markl, Albert Maier and Tobias Kraft. An Approach to Optimize Data Processing in Business Processes. In *VLDB*, 615-626, 2007.

[93] Von Welch, Frank Siebenlist, Ian Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Meder, Laura Pearlman and Steven Tuecke. Security for Grid Services. In *HPDC'03*, 48-57, 2003. See also http://www-unix.globus.org/toolkit/docs/3.2/gsi/.

[94] Weipeng P. Yan and Per-Åke Larson. Eager Aggregation and Lazy Aggregation. In *VLDB*, 345-357, 1995.

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science

Editor: The Dean of the Faculty of Science

1–11: 1970–1975

12. *Lars Thofelt:* Studies on leaf temperature recorded by direct measurement and by thermography. 1975.
13. *Monica Henricsson:* Nutritional studies on Chara globularis Thuill., Chara zeylanica Willd., and Chara haitensis Turpin. 1976.
14. *Göran Kloow:* Studies on Regenerated Cellulose by the Fluorescence Depolarization Technique. 1976.
15. *Carl-Magnus Backman:* A High Pressure Study of the Photolytic Decomposition of Azoethane and Propionyl Peroxide. 1976.
16. *Lennart Källströmer:* The significance of biotin and certain monosaccharides for the growth of Aspergillus niger on rhamnose medium at elevated temperature. 1977.
17. *Staffan Renlund:* Identification of Oxytocin and Vasopressin in the Bovine Adenohypophysis. 1978.
18. *Bengt Finnström:* Effects of pH, Ionic Strength and Light Intensity on the Flash Photolysis of L-tryptophan. 1978.
19. *Thomas C. Amu:* Diffusion in Dilute Solutions: An Experimental Study with Special Reference to the Effect of Size and Shape of Solute and Solvent Molecules. 1978.
20. *Lars Tegnér:* A Flash Photolysis Study of the Thermal Cis-Trans Isomerization of Some Aromatic Schiff Bases in Solution. 1979.
21. *Stig Tormod:* A High-Speed Stopped Flow Laser Light Scattering Apparatus and its Application in a Study of Conformational Changes in Bovine Serum Albumin. 1985.
22. *Björn Varnestig:* Coulomb Excitation of Rotational Nuclei. 1987.
23. *Frans Lettenström:* A study of nuclear effects in deep inelastic muon scattering. 1988.
24. *Göran Ericsson:* Production of Heavy Hypernuclei in Antiproton Annihilation. Study of their decay in the fission channel. 1988.
25. *Fang Peng:* The Geopotential: Modelling Techniques and Physical Implications with Case Studies in the South and East China Sea and Fennoscandia. 1989.
26. *Md. Anowar Hossain:* Seismic Refraction Studies in the Baltic Shield along the Fennolora Profile. 1989.
27. *Lars Erik Svensson:* Coulomb Excitation of Vibrational Nuclei. 1989.
28. *Bengt Carlsson:* Digital differentiating filters and model based fault detection. 1989.
29. *Alexander Edgar Kavka:* Coulomb Excitation. Analytical Methods and Experimental Results on even Selenium Nuclei. 1989.
30. *Christopher Juhlin:* Seismic Attenuation, Shear Wave Anisotropy and Some Aspects of Fracturing in the Crystalline Rock of the Siljan Ring Area, Central Sweden. 1990.
31. *Torbjörn Wigren:* Recursive Identification Based on the Nonlinear Wiener Model. 1990.
32. *Kjell Janson:* Experimental investigations of the proton and deuteron structure functions. 1991.
33. *Suzanne W. Harris:* Positive Muons in Crystalline and Amorphous Solids. 1991.
34. *Jan Blomgren:* Experimental Studies of Giant Resonances in Medium-Weight Spherical Nuclei. 1991.
35. *Jonas Lindgren:* Waveform Inversion of Seismic Reflection Data through Local Optimisation Methods. 1992.
36. *Liqi Fang:* Dynamic Light Scattering from Polymer Gels and Semidilute Solutions. 1992.
37. *Raymond Munier:* Segmentation, Fragmentation and Jostling of the Baltic Shield with Time. 1993.