Increasing productivity in High Energy Physics data mining with a Domain Specific Visual Query Language

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Licenciado em Engenharia Informática e de Computadores
Instituto Superior Técnico, Universidade Técnica de Lisboa
Vasco Miguel Moreira do Amaral
aus Cascais, Portugal

Mannheim, 2004

Dekan:	**Professor Dr. Jürgen Potthoff, Universität Mannheim**
Referent:	**Professor Dr. Guido Moerkotte, Universität Mannheim**
Korreferent:	**Professor Dr. Reinhard Männer, Universität Mannheim**

Tag der mündlichen Prüfung: 17. Februar 2005

Acknowledgments

I thank my mother, father, brother and Patricia for all the care, love and moral support.

My gratitude goes also:

- To Prof. Dr. Guido Moerkotte and to Dr. Sven Helmer, for believing in my work and for all the technical support while advising my thesis.
- To the Pi3 department for the friendly atmosphere that makes a pleasant working environment.
- To Simone Seeger for helping me with the English corrections of my text.
- To the Portuguese Governmental Foundation of Science and Technology FCT for the Phd. scholarship (ref. SFRH / BD / 8918 / 2002).
- To Prof. Dr. João Carvalho project manager of the Portuguese LIP Hera-b group, and to Prof. Dr. António Amorim. Due to them and to this project I got introduced to the HEP world and contacted with a real running HEP experiment.
- To some of my good friends that contributed in a way or in the other to this beautiful phase of my life: Levi Lúcio, Andrej Gorišek, António Rendas, Hugo Castelo Branco, Steven Aplin, Lawrence Jones, Carl-Christian, Rita Torres, Daniel Peralta.
- To Science, the noble and beautiful Mankind's greatest invention!

Zusammenfassung

Diese Arbeit entwickelt die erste anwendungsspezifische visuelle Anfragesprache für Hochenergiephysik. Nach dem aktuellen Stand der Technik ist Analyse von experimentellen Ergebnissen in der Hochenergiephysik ein sehr aufwendiger Vorgang. Die Verwendung allgemeiner höherer Programmiersprachen und komplexer Bibliotheken für die Erstellung und Wartung der Auswertungssoftware lenkt die Wissenschaftler von den Kernfragen ihres Gebiets ab. Unser Ansatz führt eine neue Abstraktionsebene in Form einer visuellen Programmiersprache ein, in der die Physiker die gewünschten Ergebnisse in einer ihrem Anwendungsgebiet nahen Notation formulieren können.

Die Validierung der Hypothese erfolgte durch die Entwicklung einer Sprache und eines Software-Prototyps. Neben einer formalen Syntax wird die Sprache durch eine translationale Semantik definiert. Die Semantik wird dabei mittels einer Übersetzung in eine durch spezielle Gruppierungsoperatoren erweiterte NF2-Algebra spezifiziert. Die vom Benutzer erstellten visuellen Anfragen werden durch einen Compiler in Code für eine Zielplattform übersetzt. Die Benutzbarkeit der Sprache wurde durch eine Benutzerstudie validiert, deren qualitative und quantitative Ergebnisse vorgestellt werden.

Abstract

We propose the first Domain Specific Visual Query language for High Energy Physics in order to tackle the problem of the physicist's reduced productivity in the High Energy Physics data mining phase. This approach comes in contrast to the current one where the user is distracted from Physics by having to code his queries using a general purpose language and complex frameworks. Our new language introduces an abstraction layer where the physicists describe their queries using a notation from their domain of speech. We validated our approach by designing the language and implementing a prototype. The language is defined by a formal syntax together with a semantics defined translationally into a intermediate language, an NF2-Algebra extended by us with special grouping operators. A visual language compiler generates a target source code that deals with the particular existing frameworks. The usability of this proposed language is also evaluated in this report by performing a study with real users. We discuss in this report quantitative and qualitative measurements concerning the user's productivity, by comparing the former traditional approach with our new one.

Table of Contents

List of Figures

Chapter 1

Introduction

For the physicist, the analysis phase of High Energy Physics (HEP) is the culmination of years of work on an experiment. In this phase, physics experimentalists look at the sheer volume of data collected by detector machines and try to infer statistical physics results.

The software systems in these areas have been growing in line with the complexity of the experiments. Unfortunately, for the software of mining the data stored, the growth took an unstructured way. This is reflected negatively in the whole process performance, meaning both user's productivity (in terms of man hour) and the query systems' efficiency (in terms of speed, space and cpu usage).

The work described in this thesis wants to intervene by mitigating the problems on the users productivity. We achieve this by pioneering a new approach for doing physics analysis by making use of a Domain Specific Visual Query Language.

1.1 Motivation

The study of coherent techniques for the development of proper flexible query systems has been neglected by the physics community. To some extent, this situation is explained by the fact that until now, programming with General Purpose Languages (GPL) and some hacking solutions were enough to deal with the problem for the very few people that used to control the whole process of the small experiments. This situation gave the community the erroneous feeling that little investment would be necessary to develop a proper software solution. However, the complexity of the analysis frameworks has grown considerably, due to the enormous size of the data. The next generation of experiments like ATLAS[34], LHCb[32] and CMS[1] [28] in LHC[2] [31] require structured and performant software systems, which means efficient query algorithms and high productivity.

This is calling for experts, both from the fields of physics and of computer science, to work together on the development of a robust analysis framework. To continue the approach used until now would result in strong lack of performance (at all levels: inefficiency, non-productivity). Physicists are motivated to investigate physics and want to decouple their responsibility from the details of the system, but in reality, they are forced by these systems to behave like end users and application developers. Their productivity decreases greatly with time. Yet on the other hand, since the existing frameworks do not provide clear levels of abstraction, the computer scientists are forced to have a proper background in physics in order to have room for improving the efficiency of the system by developing proper optimization techniques. As a consequence, this situation calls for a new strategy to introduce the required productivity, modularity and efficiency in a controlled way.

To find a properly structured solution is very important for this branch of science, since for the coming generation of physics detector machines with their dimension and complexity, the traditional techniques are not sufficient.

This situation is an interesting challenge for computer science, since a new application area for this science is opened. This domain of research has, for instance, requirements which are very different from those in business and industry. A full investigation must be done to find the core technology best suited to develop a query framework for this particularly complex domain.

1.2 Objectives

The aim of this work is to to increase the user productivity and introduce a framework that allows computer experts to investigate efficient ways to

[1]Compact Muon Solenoid
[2]Large Hadron Collider

optimize the High Energy Physics analysis process without requiring to be physics experts.

We achieve this by introducing an engineering methodology and making use of a declarative Domain Specific Visual Query Language (DSVQL) to raise the abstraction level in the existing query systems and to give room to new optimizations of different levels. In order to corroborate our argument, we have implemented a prototype framework, called PHEASANT, and a visual language named PHEASANT QL. This framework was developed in the context of the last big experiment, HERA-B in DESY[3], running in Europe before the LHC era. It is an interesting case study, since it has real data to study and users to interact with.

1.3 Scientific Domain of the Thesis

In order to investigate the solution for this domain, we crossed several scientific domains, basing our solution on their techniques. The most important among them are Physics Computing (PC), Domain Specific Modeling (DSM), Database Computing (DC) with Flexible Query Systems (FQS), Human Centric Interfaces (HCI) and Visual Query Languages (VQL).

1.4 Thesis Outline

This thesis is divided into four major parts:

- The first part deals with the problem definition. Here, we introduce the reader to the context of High-Energy Physics experiments. It is followed by a description of the data mining phase, also called data analysis, and finishes with the problem specification and the motivation for our work.
- In the second part, we introduce some concepts that are useful for the argumentation of our proposed hypothesis in the next part. Query systems taxonomy and domain specific modeling are described.
- The third part approaches the hypothesis. Using domain modeling, a language and a corresponding framework are designed. The core technologies are detailed.

[3]Deutsches Elektronen Synchroton in Hamburg, Germany

- The fourth and last part is dedicated to the evaluation results and conclusions.

Part I

Problem Definition

Chapter 2

Context of the Work

In this chapter, we familiarize the reader with the environment of High Energy Physics (HEP) and the computing activities involved. This way, we lay the foundations for the problem definition presented in the next chapters, concerning the physicist data mining phase (commonly known as analysis in the HEP community).

In Section 2.1 we start by giving a quick overview over the physics goals. We can only roughly sketch the basics of HEP, due to the complexity of the subject and space constraints. A good introduction to the subject can be found in [61]. Then we explain the common architecture of the HEP experiments and give a historical perspective of the analysis frameworks and the analysis tools evolution.

2.1 Overview of High Energy Physics

Generally speaking, physicists try to discover new short-lived particles and their properties or the properties of their interactions, in order to develop a model of the real world at a subatomic level. For this, they use e.g. large accelerators in which particles collide with others, and detector machines composed of sub-detectors to measure the results. The accelerator supplies the particles, which are grouped into *bunches*, with energy taking them close to the speed of light (large kinetic energy), making them collide with other particles, fixed targets or other beams of particles.

When masses slam together at huge kinetic energies, their energy converts to new particles and their kinetic energies. The bigger the initial

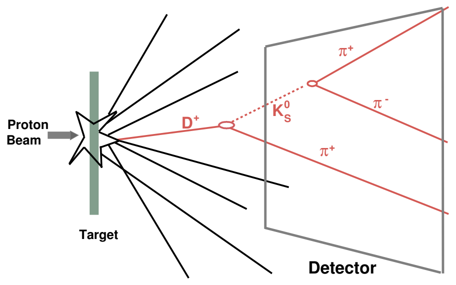


Figure 2.1: Colliding a beam of particles against a target.

kinetic energy is, the bigger the masses of the products potentially are. The new mass resulting from the very high energy collisions will appear as different, unusual, and interesting particles. Some of them have a short life, so they decay into other particles before they can be detected. Those particles that live longer, due to their more stable nature, will be detected by sub-detectors which track them in space, identify their type and determine their energy. As an example, we have in Fig.2.1 the collision of a proton beam with a target. This produces a particle called D^+. D^+ decays to other particles before reaching the detector. Nevertheless, its decay products live long enough to cross the detector machine and be detected.

When the experiments are running, a period of data acquisition, a so-called *run* begins as soon as the system stabilizes. The time-span during which two bunches collide is called an *event*. From now on, we will see an event as an abstract granular entity that refers to the data taken by the detector machine immediately after the collision during this referred time span. In the main detector machine, large sub-detectors, which are independent of each other, record the results of an event. Unfortunately, it is technically infeasible to gather all information of all collisions (due to the sheer volume of data), so the physicists filter the data with several levels of *triggers*. The resulting data is initially stored on tape. After having examined the data, the accelerator and the detectors are reconfigured (if necessary), and another run can be started.

The reconstruction and investigation of decays and decay chains of short-lived particles are the main computationally demanding tasks of the data analysis, which starts after the data acquisition. Roughly speaking, in this phase, physicists have to select those kind of decays and particles they are interested in. For this selection, it is usually necessary to reconstruct parts of the particles' trajectories (also called *segments*), to match them with other segments in order to reproduce the full particle trajectories (called *tracks*), to extract further properties, and to deduce the complete decay chain.

2.2 The Detector

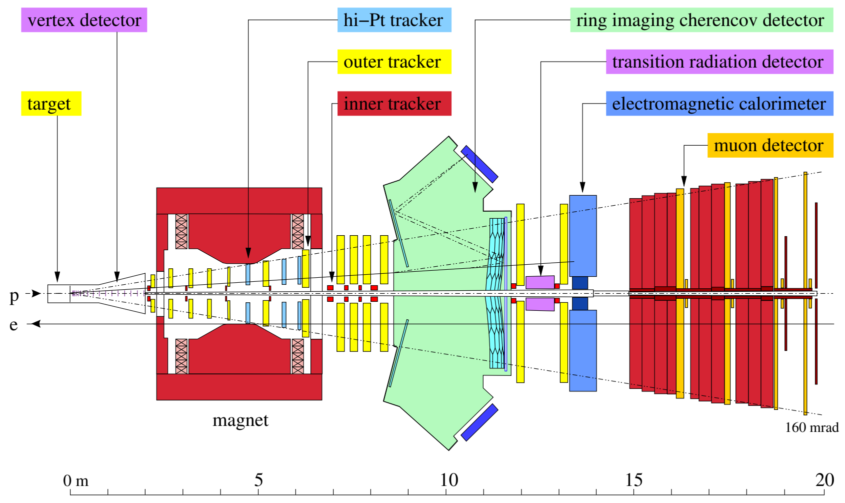


Figure 2.2: The HERA-B detector machine

The work described in this thesis was developed in the context of the HERA-B experiment, based in Germany at the DESY[1] Laboratory.

HERA-B is the biggest working experiment in Europe, before the next generation of probing machines comes around 2007 at CERN[2]. For Hera-B, 32 institutes and about 250 collaborators from 13 countries are working together. The machine built, depicted in Fig.2.2, was meant to search for CP violation in decays of B mesons into the "gold plated" decay mode $B \rightarrow J/\Psi K_S^0$, the details of this concepts is out of the scope of this thesis but can be consulted in [35].

[1]Deutsches Elektronen-Synchroton in Hamburg, Germany
[2]European Laboratory for Particle Physics, Geneva, Switzerland

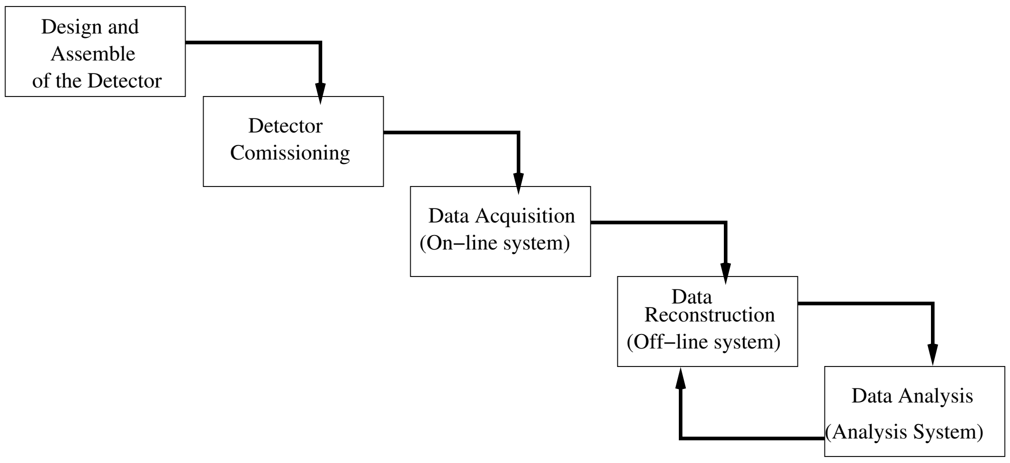

Design and Assemble of the Detector
Detector Comissioning
Data Acquisition (On−line system)
Data Reconstruction (Off−line system)
Data Analysis (Analysis System)

Figure 2.3: The life phases of a typical HEP experiment.

In Fig.2.3 we sketch the five typical life-phases of a HEP experiment . It starts by the design and detector assemble, followed by its commissioning. As soon as the detector is ready the data acquisition takes place making use of the on-line systems technology (hardware and software). Once the data is collected the off-line systems will take the role of looking at the signal data and construct interesting physics quantities. Finally, with this quantities the end-user (the physicist) will proceed to analyze (or mine) this generated data.

In the next section , we have a look at the detector machine and its components. Together with this description, we also explain the last three referred phases of the hep experiment. We describe the on-line systems which are used in the acquisition phase, also called data production. Following that we describe the off-line systems and at last the analysis (the final computing intensive phase of the experiment).

2.2.1 The Machinery

Everything starts with the accelerator, which produces a beam of particles. As the name indicates, the accelerator is the piece of hardware which provides energy for the particles, accelerating them close to the speed of light. Basically, there are two kinds of accelerators: linear accelerators, where the long tunnels have the shape of a straight line, and circular accelerators.

In DESY, this machine is called HERA. It is a large underground ring tunnel, 10 meters below the surface, with a circumference of 3 km. The tunnel is 5 meters in diameter[3].

In HERA, the beam consists of a lined up sequence of several groups of particles of the same type separated by a given distance, called *bunches* in the physics jargon. These bunches come with a frequency of 96 η seconds, i.e. this figure denotes the distance between the bunches.

After the particles are accelerated, the collision takes place. There are two approaches to provoke the collisions: either fixed target or colliding beams. In the first approach, particles are made to crash into a solid block or gas of some type. The second approach is based on the concept of making two bunches of particles which travel into opposite directions meet at a certain point in space. In HERA-B, wires of different materials, called the target, are approached to the beam while the bunches are passing by, provoking collisions with the particles on the wires at a very fast rate, which is called the *interaction rate*.

After the collision, it is necessary to detect and measure the results. Big detector machines are built around the interaction region, extending from a point very near the collision to dozens of meters away. They measure the particles that survive longer, like electrons or muons, and their properties (charge, invariant mass of the particles that are generated, direction and momentum, etc.). With this information, it is possible to reconstruct the original particles from which they decayed, proving their existence, and to measure the desired properties of the interactions between them.

The typical HEP experiment apparatus consists of:

- **Interaction region/Target:** Where the collision takes place.
- **The detector:**

  Vertex track detector: Measures coordinates of the hits provoked by vertex particles very near to the interaction region.

  Magnet: Deflects the passing particles with an angle which is proportional to its momentum.

[3]The future circular accelerator under construction at CERN, called LHC, will have a perimeter of 28 km. The planned TESLA will be a linear collider with a length of 33 km.

Ring Imaging Čerenkov Detectors (RICH): Identifies the kind of particles.

Tracker detector: Measures several coordinate hits along the particles' trajectories.

Electronic/Hadronic calorimeter: Measures the energy of electrons or hadrons and identifies (separates) them.

Muon chamber: Identifies and measures muon particles.

The general structure of the machine is described in Fig.2.2.

The main idea of these different layers on the detector is to generate enough combined information to explain which particles crossed the detector and provide this information to the people doing analysis, who will try to reconstruct what happened during the collision.

2.2.2 On-line System - the Triggers and Data Acquisition System

Due to the sheer volume of data, it is technically infeasible to gather all information of all collisions. Moreover, in many cases the probability of producing the interesting reactions the physicists are searching for is very low, compared to other kinds of reactions. As an example, HERA-B was searching for interactions that have certain particles in the final state. With the rate of 10 million of collisions per second, in other terms, a frequency of 10 MHz, that kind of event will be produced only once every 10^{11} interactions (collisions). Therefore, complex filtering trigger systems have to be designed in order to separate the few interesting interactions from the large background of uninteresting events. In HERA-B, a three-level trigger system was built.

During the processes of data acquisition and reconstruction, the large data sets of these experiments are stored onto robotic tape systems.

The schematics of the data flow during the data acquisition are described in the Fig.2.4. We dedicate the rest of this section to explain the components depicted.

The data is pipelined in the data acquisition system (DAQ) and waits for the different trigger decisions. During this phase, about 1500 software processes are running on several Linux clusters.

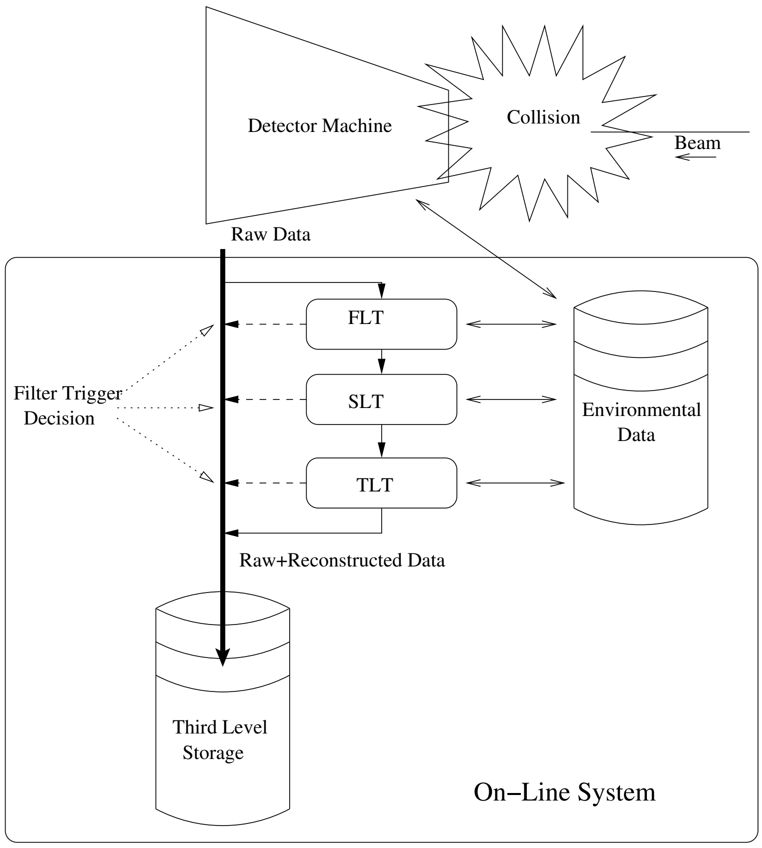


Figure 2.4: The Triggers and the Data Acquisition System

- **First Level Trigger (FLT)-** As soon as the analog signal comes out of the detector chambers, it is amplified, discriminated (by making the difference between what is a valid signal and what is just noise), and digitized in an electronic board named Front End Driver (FED). All this data is pipelined and waits for a decision. This decision is taken by the first level trigger hardware within a time frame of $10\mu s$. It consists in looking at the hits in the several sub-detectors and identify trajectory patterns that justify acception or rejection of the information. This level is dealing with a data rate of 5×10^{12} bytes/s.

- **Second Level Trigger (SLT)-** The data resulting from the FLT is distributed among 1000 Sharc[15] Digital Signal Processors (DSPs)[54]. These DSPs are installed in VME crates[4] with a very efficient data bus, which transports the data to the third/fourth level trigger. The SLT is a programmable layer that allows to run algorithms real time constrained for selections based on additional information coming from the detector. It executes pattern recognition algorithms to reconstruct the trajectories of the particles inside the detector and can determine the invariant mass of the particles in order to decide. These algorithms have to take the decisions in 1 ms, this level is dealing with a data rate of 25×10^9 bytes/s.

- **Third/Fourth Level Trigger (TLT)-** The information is then pipelined to a cluster [46, 47] composed of 240 microprocessors nodes. These nodes run a program for the full reconstruction of the *events*, make some looser selection of the interesting ones, and, based on some computed likelihoods, classify them in the different categories that may be interesting for the different kinds of physics analysis. This reconstruction is more time consuming, therefore the trigger decision has to be taken in 10 ms. This level is dealing with a data rate of 250×10^6 byte/s.

Finally, the **raw data** (signal information) and **on-line reprocessed data** (physical quantities extracted by the recognition algorithms which were run on the raw data) are stored on tape with a data rate of 2.4×10^6 bytes/s.

2.2.3 Off-line System - Data Reconstruction

The real-time system described previously provides a first level of analysis and selection of physics data, which has to be processed automatically.

When the data storage/production is finished, we can categorize the kinds of data[12, 13] produced and/or used in an experiment detector machine in the following way (see Fig.2.5):

- **Basic HEP constants:** for example constants like masses of different particles.

[4]VERSAmodule Eurocard. Systems for mission-critical and real-time applications.

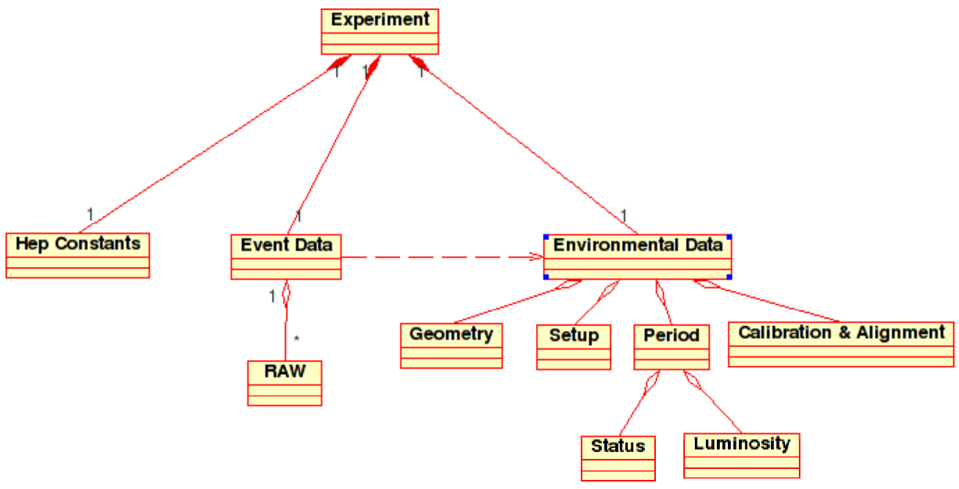


Figure 2.5: Data produced by the HEP machine before the reconstruction

- **Environmental Data:**

  Setup - Cabling connections and software configurations.

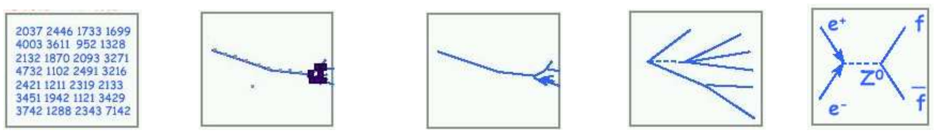
  Geometry - There is a nominal geometry that describes the shape of the detectors, their positions, etc. It is obtained mechanically (by automatic reading of the measuring instruments) during the data acquisition and afterwards with the calibration and alignment data.

  Calibration and Alignment - The geometry of the detectors is obtained mechanically, by reading the measuring instruments during the acquisition. However, the accuracy of the detector position is obtained only in the order of millimeters, which is less precision than needed later. Therefore, it is necessary to apply alignment algorithms in order to determine with a precision of μmeters where the detector and its sub-detectors are located, compared to the beam position (and, in the case of the sub-detectors, between each other). The alignment algorithms make use of calibration and alignment data. This data is stored during the data acquisition and corrected afterwards by the sub-detector experts.

Period (or Slow Control): Status, Luminosity - Conditions like atmospheric pressure and temperature can influence the precision of the machine. This information, which does not change very fast, is stored in a database system[6, 14].

- **Event Data (raw):**

  Event, sub-detector FED bit pattern - Signal data is characterized by being written once and never modified.



(a) Raw data (b) Physics quantities (c) Detector interaction (d) Decay (e) Basic physics

Figure 2.6: Informal description of the results of the major transformation phases.

Data cannot be used directly as it comes from the detector (it consists only of electronic signals). Therefore, it needs to be transformed into some quantities the physicist is able to handle and to understand. As a consequence, the raw data is transformed in several phases. These transformation phases are shown in Fig.2.6 from a) to e). Briefly, we can describe the process at the conceptual level as follows.

The raw data in (a), composed by read-out addresses of the detector, bit patterns, etc., is first converted to the description of the hits (i.e. points of interaction of the particles) in each layer of the detector (b). To pass from (a) to (b), several problems like noise, detector inefficiency, ambiguity, resolution, alignment, and variations in temperature must be solved by calibration, noise reduction, and alignment algorithms. As a result, information about the interaction of the particles with the detector material is obtained. This is the starting point for the next phase, which is the pattern recognition of physical segments, clusters, and rings. This way, particles crossing the detector are identified (c). This last computed information is used by the scientists for the decay studies(d). The results will provide physical statistics and probability figures to support the theoretical model of particle interaction under investigation (e).

2.2.4 Analysis System

One of the main tasks of the data analysis software in experiments on high energy physics is the reconstruction and investigation of decays and decay chains of short living particles. A lot of information from different reconstruction algorithms (e.g. reconstructed primary vertices, particle identification, momentum determination etc.) must be combined in order to identify trajectory combinations which have a common origin and belong to the decay of another particle. The reconstructed decayed particle can be itself a decay product in a complex decay chain and used as an input for further decay reconstruction.

Roughly speaking, the analysis systems are composed of a visualization tool, a set of scientific calculation libraries, and a storage manager (a detailed description with the history of its evolution is contained in Section 2.3). Traditionally, in a first step of his analysis, the user selects a subset of data from the storage manager. Then, several reconstruction algorithms with scientific calculations filter out data and compute new values that are stored in private collections. Finally, the new data collection is visualized in the visualization tools (for instance by histograms). In Chapter 4, we will explain the analysis phase in more detail.

2.3 Historic Perspective of the Analysis Systems

In this section, we are going to describe the structure of the analysis systems, i.e. the storage and visualization tools, and their historic evolution. With this description, we already start approaching the nature of the problem in the analysis phase. In fact, understanding the evolution of the architectures will help us to understand both how the legacy systems dictated the system architecture of the present experiments and what the main reasons for the growing dissatisfaction of their end users are.

We will describe the evolution from the early stages, dominated by an unstructured approach, till the time these frameworks adopted the object-oriented design. We will end with a description of the current trends and future tendencies.

Since we will use the concepts of the levels of abstraction in a DBMS, we will shortly define what we understand by the three levels: conceptual (or external), logical, and physical. In summary, the conceptual model is concerned with the real world view and understanding of data; the logical model is a generalized formal structure according to the rules of information science; the physical model specifies how this will be executed in a particular DBMS instance.

2.3.1 Unstructured Approach

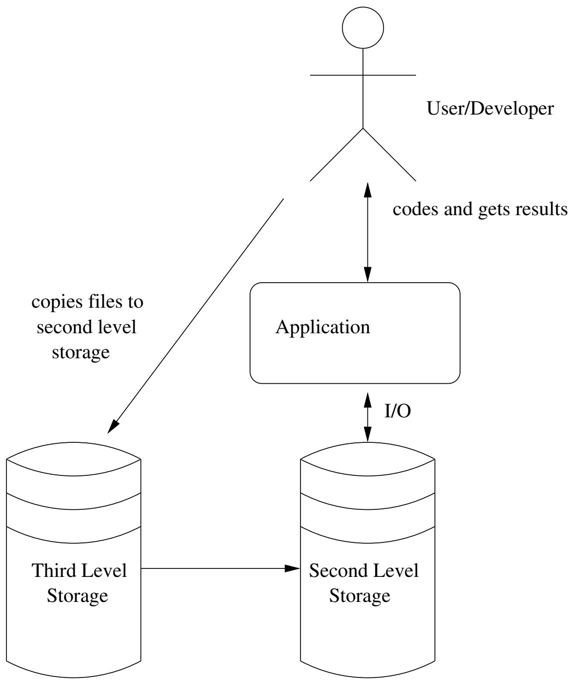


Figure 2.7: User builds his own query system from scratch

In the early and small experiments, data was usually organized in compressed, self-describing data formats stored in flat files. The user was responsible for fully coding the complete query, including loading the data from files into main memory, query computation and result analysis code (see Fig.2.7). A deep knowledge of programming, especially in the FORTRAN language, was necessary. The data schema and the storage formats were unstructured and changed very often, which made the code difficult to reuse and maintenance a nightmare.

In the early experiments, this approach was easy to handle. However, when both the data volume and the schema grew, the community soon realized that a lot of main memory was needed. The code for executing I/O tasks had to be redone continuously. The user also had to worry about things like how to minimize the number of accesses to the tertiary storage, which was possible only by knowing beforehand which files contained relevant events and where they were physically located in the system. In addition to all that, the user did not always implement the most efficient code. In consequence, to build up a library that brings the relevant parts into main memory for processing became the next design concern.

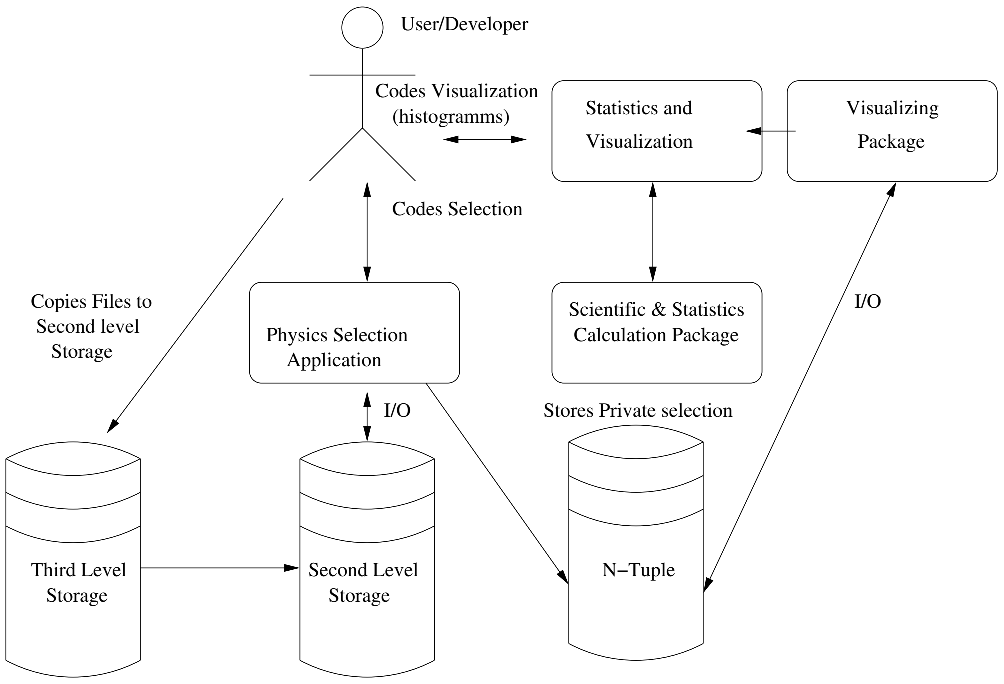

User/Developer
Codes Visualization (histogramms)
Statistics and Visualization
Visualizing Package
Codes Selection
Copies Files to Second level Storage
Physics Selection Application
Scientific & Statistics Calculation Package
I/O
I/O
Stores Private selection
Third Level Storage
Second Level Storage
N−Tuple

Figure 2.8: User codes with different sets of available libraries

Specialized packages (see Fig.2.8) were developed to provide a set of independent libraries providing specialized I/O together with algorithms for physical analysis and mathematical calculation, and functions for statistics, histogramming and visualization. The user still had to deal with the growing complexity of the data's physical layout, but had access to a widely dispersed set of packages to re-use ("glue") instead of doing everything by himself.

A standard storage format for event data called ZEBRA[21] was defined for the data stored in files in HEP. It was quickly adopted in most of the HEP experiments, since it simplified the code for looping inside the files. The event is seen as the granularity of the data, the contents of them as black boxes, which had to be interpreted by the user code.

Typically, the use case would start by a pre-selection of files that might contain interesting events. The user had to program many lines of code using imperative languages like FORTRAN to specify the application code representing the query. The query program would loop over the event data stored in each file, compute new values and determine if they were interesting enough to store in a specific flat file on a local workstation. In the physicists jargon, the result was an n-tuple table. This table was then used for the final statistical calculations.

At this point, these analysis frameworks made no distinction between the physical and the logical levels and, obviously, the conceptual model was not completely covered. The user had to know the specific layout and particular storage location of the data. In order to extract the data from different complex sources and to deal with the complexity of the data, including the transfer to main memory, it was necessary to write code. These frameworks became too complex to use, and the practical reusability of the produced code was limited.

The volume of the data (that had risen to magnitudes of terabytes) and its storage, together with the need for expensive computer resources, forced that hundreds (sometimes thousands) of users at different levels (physicists doing analysis, component experts extracting and generating physics analysis data, system administrators) had to access the data in central data repositories. At the same time, all of them were expecting a highly efficient system.

Visualization Tools

By then, tools like PAW[33] had appeared, which provided the end user with subroutines that would integrate the I/O package for accessing the referred n-tuple, and with data visualization packages (mostly histogramming). While this was very convenient for generating histograms and statistical calculus, it was extremely difficult to use complex data structures that required references among data objects (this problem will be more clear in the next chapter when we describe the conceptual model and the logical schema). Besides, a method to identify all data items was necessary. Queries at this level were obviously limited and totally dependent on the structure of the tuples defined by the user.

2.3.2 Analysis Frameworks

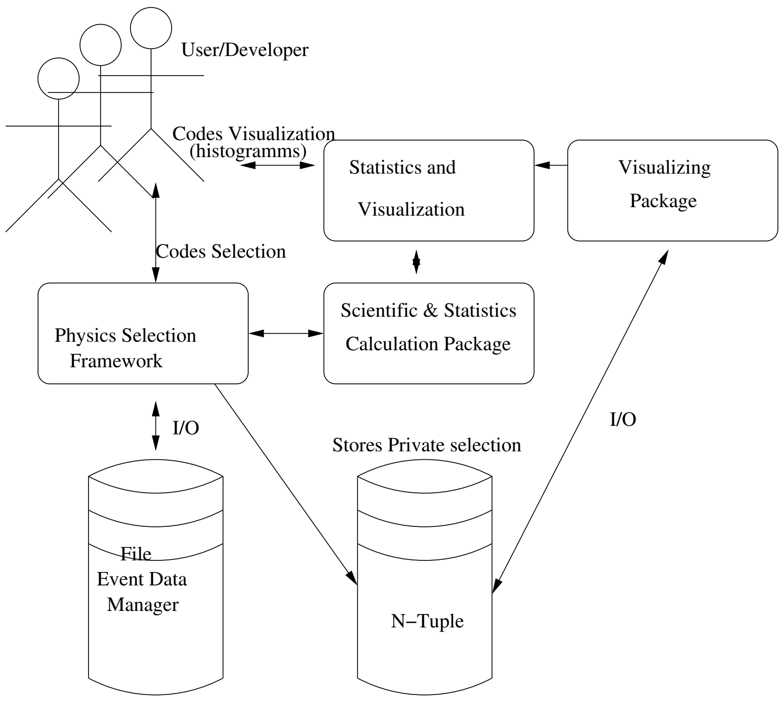

User/Developer
Codes Visualization (histogramms)
Statistics and Visualization
Visualizing Package
Codes Selection
Physics Selection Framework
Scientific & Statistics Calculation Package
I/O
Stores Private selection
I/O
File Event Data Manager
N–Tuple

Figure 2.9: Multi-users'/programmers' query over a framework

A second generation of approaches to this problem, (see Fig.2.9), still used by many experiments world-wide, started to implement hybrid object-oriented frameworks, like ARTE[4], where the other paradigms were inherited. The idea was to cope with the growing user demand for query applications which use an object-oriented design.

These frameworks were meant to centralize all the packages for supporting the tasks emerging during the life phases of the experiment, including data production, simulation, transformation (in physics jargon re-processing) of the raw data into physics data, and finally the analysis phase. Cluster solutions were chosen in order to increase computational power with cheap commercial hardware.

Although not worrying about the specific requirements for the analysis phase, the great achievement of these systems was to provide transparent access to every file with persistent event data. It did not matter whether it resided in secondary or tertiary storage. It gave rise to several projects whose mission was to optimize the I/O performance.

This second generation approach provided the end user with the "main event loop abstraction", where the program loops over the event data elements stored in the file and hands them over to the physics algorithms that are responsible of knowing the proper schema and extracting the required information into main memory. Typically, the user had to know in which file he would find the event data elements he was interested in before running the physics algorithms on them.

These frameworks were also developed with the goal of distributing the queries and data on computer clusters in a multi-user environment. Some of them have primitive load-balancing capabilities. They did not exploit parallelism.

Although they still do not hide the complexity of the data structure of the events from the end user, these frameworks introduced more modularity and integrated the dispersed packages. Nevertheless, they were strongly bound to a particular physics experiment's implementations, which meant that the user had to re-learn them in every new experiment, and was dependent on legacy code. This way, algorithms were coded in several general purpose languages and paradigms, with steep learning curves and with a high risk of being inefficient when badly written by inexperienced users.

These systems also do not present different views for the different users involved as they do not hide the unnecessarily complex data structures from the end users. As a result, users had to map domain concepts into design concepts and then to implementation concepts, without any abstraction involved. Partially, this confusion was already generated by the physicist's dual role of developer and end user.

2.3.3 Object-oriented Frameworks

Attracted by the advantages of a DBMS - for example, concurrency control, indexing support and query capabilities - some recent HEP experiments, like BaBar[5] and AMS[6], implemented solutions based on a commercial OODBMS. Unfortunately, this approach has shown to be problematic - in part due to some inexperience of the community in OO design, and greatly due to the non-scalability of the available OODBMS commercial solutions.

The introduction of the OODBMS technology allowed a clear separation between the physical and the logical levels and allowed some optimization approaches like the introduction of vertical partitioning[87] of event data or bitmap indexes[48].

Visualization Tools

JAS[30](Java), ROOT[81](C++) and other visualization tools became the object-oriented evolution of PAW. Since the relative failure of OODBMS usage for HEP purposes, the tendency now is to change this situation by developing a more adequate storage layer underneath the visualization tool. This way, both visualization and storage are combined in the same tool, and the user programs the complete query in the same programming paradigm. In order to make an HEP object persistent, special machine independent I/O mechanisms are being developed (an example is the package ROOT I/O[81]).

As such tools were originally designed to deal with a local storage of the selected n-tuple data in the physicist's computer, the mission of turning the tool into a distributed very large database within a distributed heterogeneous multi-user environment is necessarily very difficult to accomplish.

Although these tools are meant to support the user during the query programming phase, they have a confusing logical schema that is unrelated to the conceptual one. The physicist has to twist the way he conceptualizes the data into the unnatural object model these tools support. They still imply object-oriented programming activities using a growing number of complex library of functions which are difficult to learn by the end user.

[5]SLAC, USA
[6]NASA, USA

2.3.4 Current and Future Trends

In 1997, in a vision paper at the VLDB conference, the community presented the requirements for their domain [43] with the idea of pushing the limits of technology. Some goals for the next generation of systems were set:

- Deal with petabytes of data.
- Support hundreds of simultaneous queries.
- Return partial results of queries in progress (with time estimates for their completion) and provide interactive query refinements.
- Deal with data on secondary and tertiary storage access for simultaneous queries.
- Provide an environment for data analysis that is identical on desktop workstations and centralized data repositories.
- Support statistical selection mechanisms (uniform random sampling).
- Provide a flexible schema which supports versioning.

In part motivated by these requirements, some future experiments, especially in CERN (Atlas/CMS/LHC-B,etc.), are embracing the development of a new system called GRID. The leading role of the CERN institution worldwide normally has a strong influence on the technology chosen for the other physics experiments in the future.

The main mission of GRID computing is to coordinate distributed heterogeneous hardware and storage resources among a dynamic set of individuals and organizations in order to achieve a common goal. It involves the studies of peer-to-peer solutions applied to this domain's requirements with development and implementation in different areas such as data replication, migration, security, processing, load balancing and networking philosophies. Still at its starting phase, it aims to be the next big revolution on networking for scientific computation in 2007, when the next big experiments at CERN (ATLAS, CMS) start to run.

In spite of the very complex, but promising technology, to our knowledge no serious studies exist about the conceptual model for the analysis, logical schemas and analysis query patterns. We predict that this will become a serious gap in the future when it starts to be necessary to tackle the problem of user's productivity. The other problem directly related to the lack of abstraction layers will be when the experts try to tune the computational performance of the query systems.

2.4 Summary

In this chapter, we have introduced the physicist's HEP analysis phase and described the structure of the typical HEP systems. We have also given an overview of the physics activities involved, trying to avoid unnecessary complex descriptions that are beyond the scope of the thesis.

From this chapter, we would like to highlight some key ideas that will be handy for the discussion in the following chapters:

- Only part of the total experiment's stored data is actually used for physics analysis.
- In order to mine the data, the end user adopts a dual role of application programmer and user.
- The developed tools do not allow for data independence.
- The complexity of the data structures and the representation of the data is not hidden from the user.

In the next chapter, we are going to describe in detail the physics analysis process and the problem we propose to solve.

Chapter 3

The Physics Analysis Process

This chapter is dedicated to describe the physics analysis process in more detail. The documentation in this area under the perspective of computer science is typically very poor, inconclusive and sometimes contradictory. Therefore, we expect to bring some light into this subject with our own interpretation resulting from the experience we have with real users running analysis systems in a running experiment (HERA-B).

We start by explaining the difference between low-level and high-level analysis. Then, we proceed with an overview of the schema. We finalize by explaining what major steps are involved while querying the physics data, and which query patterns we might expect.

3.1 Defining Physics Analysis- Low versus High Level

There is still some controversy about the concepts of low-level analysis in the physics community. Therefore, we are going to define our understanding of them in the rest of this section.

Traditionally, analysis used to involve writing code in a General Purpose Language (GPL), like C++, Fortran, etc. This code was responsible for performing the whole data transformation chain in the same user application program, which includes the reconstruction of physics data from the raw data and the analysis. This implied the use of specific data from the detector machinery such as information on geometry, calibration and alignment, to reprocess the raw data in order to produce the physics data and to run the query algorithms. This kind of analysis, now called low-level analysis, is a legacy from old experiments. It resulted to some extent from the need that the users had to start with their analysis while the detectors were still at the phase of confirming that the system function matches the operational needs, also called commissioning phase[1]. The main reason for this situation is that, as in any complex research experiment, when using cutting edge technology, the behavior of the machines is not always completely understood.

Low-level analysis was only possible thanks to the relatively small detectors. They were characterized by having both small data sets and relatively simple data structures. The sub-detectors' description data was very reduced.

At present, due to the very large data set and the complex queries required for the new generation of experiments, this analysis is no longer possible. Re-processing the whole data set takes several months. In consequence, the control of such data re-processing activities should no longer be on the users' side, but shift to some other actors like the experiment's management. Then the low-level analysis is left to the machine experts that will only perform machine tests over small data sets. On the other hand, high-level analysis, which queries simply physics data and ignores the details of the machinery involved, is the new way of analysis.

High-level analysis involves generating queries only on physics data, with a specific data model in order to return the interesting statistical results.

3.2 Monte Carlo Simulation

As we explain in the next section, while describing the analysis data model, it is very common in Physics to use the so called Monte Carlo simulation technique (for more details consult any statistics book, for instance [83]). It consists in the random generation of values for certain variables according to a model. It is generally used when there is the requirement to automatically analyze the effect of varying inputs on outputs of the modeled system. This simulation technique was named for Monte Carlo, Monaco, where the primary attractions are casinos containing games of

[1]The four main phases of the experiment are: design, construction, commissioning and operation (or data production).

chance such as roulette wheels, dice, and slot machines, that exhibit random behavior.

This statistics technique is very often used for the generation of simulated physics data. It follows a complex model to simulate all the particles that cross the detector, their interactions between them and with the detector, in order to simulate the data that comes out of the detector ("hits").

3.3 Analysis Schema

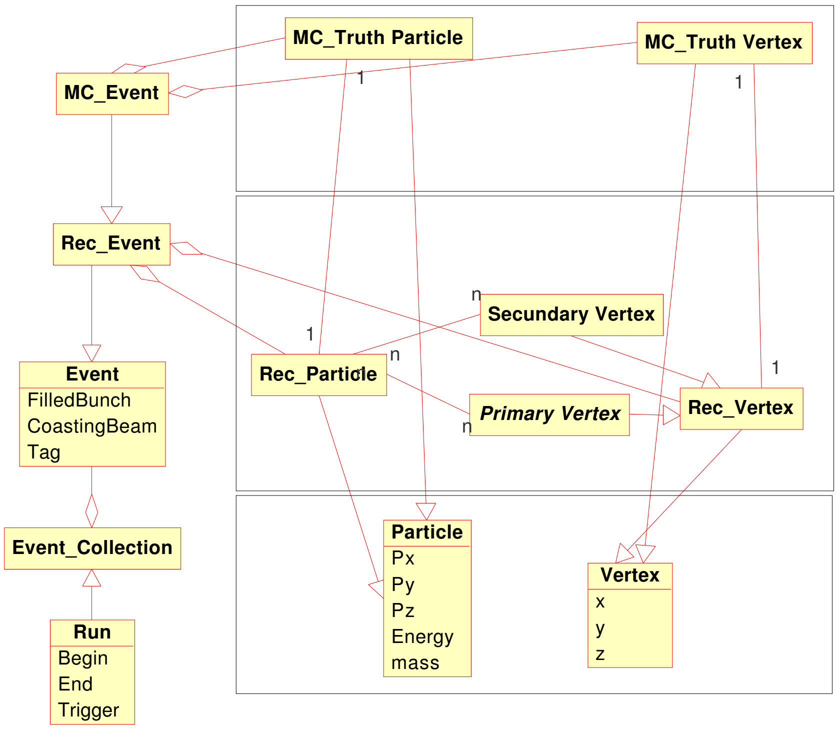


Figure 3.1: Detailed UML model of the analysis of the relevant event data

Based on our case study, the Hera-B experiment, and physicists' descriptions of other experiments taking place worldwide, we derived the conceptual model of the analysis data in a UML diagram that is depicted in Fig.3.1. It consists of the following entities: the generalized

`Collection`, from which other `Run` or even others are inherited; `Rec_Event` that aggregates `Rec_Particle`, `Rec_Vertex` (that can be of two types: primary or secondary), and the simulation events `MC_Event` that inherit from `Rec_Event` and store extra information `MC_Particle`, `MC_Vertex`.

The attributes of the entity `Run`, a typical specialization of the entity `Event_Collection`, define meta-data information for the `Event` data that is being collected, such as the parameters of the experiment, e.g. the setup of the detectors, the time span of the data acquisition and general quality issues.

Event attributes describe properties of the set of particles involved in an event. This entity can have up to ≈ 10 different attributes of mostly Boolean and up to 80 enumerated types with a list of enumerated values comprising up to 80 values. These attributes are mainly referring to the usage of certain algorithms for the re-processing of the particular event. Some, but few, double precision attributes might be used. Finally, an attribute of type integer might be used to specify the version of the same raw event that was reprocessed.

Events can be simulated or real. A simulation means that the production of random collisions is simulated, by using the Monte Carlo method, and that the particles and vertexes are reconstructed using the same software algorithms as if they had really crossed the detector. These particles and vertexes are generated with exactly the same attributes as in a real reconstruction. The difference of this simulated data to the real one, concerns only the so-called Monte Carlo truth, (in Fig.3.1 `MC_Truth Particle` and `MC_Truth Vertex`), which are a one-to-one association of the exact information about the simulated particles and vertexes. This MC_Truth gives the information about the particles and vertexes as if they were crossing the detector before the reconstruction algorithms track and identify them. Mostly, given the complexity of the pattern recognition algorithms, the path of the particles identified is the nearest possible approximation to reality. This kind of information is used extensively for the determination of error rates and efficiency figures. Attributes for this entity can be Boolean values, for instance, or a list of enumerated values in an attribute tag.

Fig.3.2 consists of various entities for the description of particles, vertexes and their corresponding simulation. Almost all of these attributes are double precision numbers.

A particle is, in general, described by its momentum, its mass and the

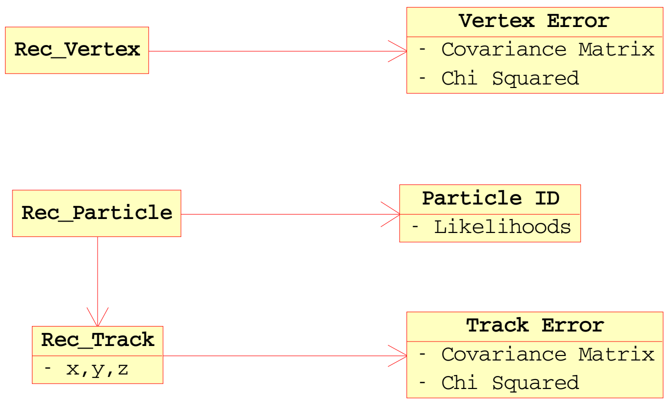


Figure 3.2: UML details of the reconstructed Vertex and Particle.

coordinates of the first measured point of its trajectory. In the case of a decaying particle, the point could be the decaying vertex. In the case of neutral particles they can be described by the energy deposited in the calorimeter detector and the position of this energy deposition, since they do not have a measured trajectory.

In Fig.3.3, we present an informal object graph of the described data model. It represents the physics analysis' database at the instance level as it was defined in Hera-B. Along with it, we show some figures to give an idea of the proportions and number of objects taken for the analysis phase. These figures concern data taken over a period of 6 months.

3.4 The Query Patterns

The physics data for the analysis can be described as WORM (Write Once and Read Many). Typically, the analysis queries are issued only once. This means that every new physics query requires a new application code. Nevertheless, almost all have one sequence in common (see Fig.3.4), where the first three major steps are selecting the available data (first filtering out pre-defined collections, like Runs, and then retrieving sets of the contained Events), reconstruction of the decay for each event, and the last one is visualization of statistics data (usually using histograms).

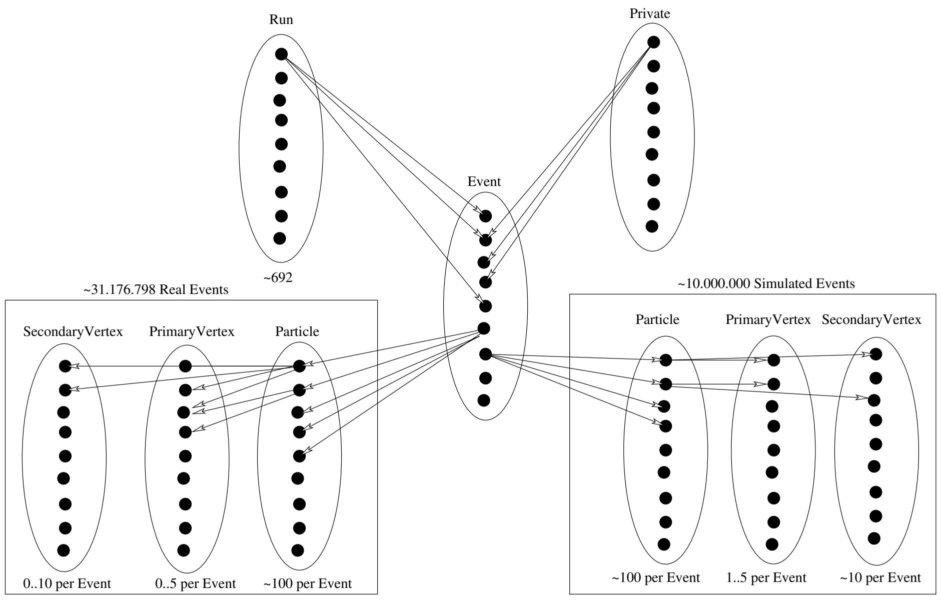


Figure 3.3: Object graph representing the physics' analysis database at the instance level

In this section, we are going to describe the different patterns and give the pseudo-code of a real life example.

3.4.1 Query Steps

In order to sieve out interesting subsets of events, the analysis starts by selecting the `Collections`. This involves partial match queries over some `Collection` attributes. Usually, this makes use of up to 5 dimensions.

The second phase implies range queries over a small subset of the `Event` properties. While the events can have as much as ≈ 10 different attributes, i.e. ≈ 10 dimensions, the number of properties restricted by mostly range and partial match queries is usually much smaller, typically 1 to 4.

With the filtered event data collected in the first and second step, the physicists now try to reconstruct decay chains as a third step. At this level, retrieval techniques must deal with many difficult problems: enormous quantities of data, high data dimensionality, low-dimensional

- 1 - Run/tag selection:
  - **Trigger selection**
  - **Run period**
- 2 -*event* Selection:
  - **Filled bunch**
  - **No coasting beam**
  - **No empty** *events*
  - **Refined confirmation of the trigger**
  - **...**
- 3 - Reconstruction:
  - **Track selection**
  - **Particle ID filter condition**
  - **Combination of tracks**
  - **Vertexing**
  - **Kinematic or geometric filter conditions**
  - **...**
- 4 - Histogramming and/or comparison with Monte Carlo Simulation

Figure 3.4: Query steps for general analysis

region queries, and highly skewed data distributions. However, they are not interested in all decays that took place in an event, but want to sort out data that is irrelevant to their current investigations. This also involves computing and caching of intermediate results.

The third step starts by selecting the different particles, "leaves", of the decay tree. This involves selection predicates with range queries over typically up to 8 dimensions. Sometimes, with the simulated data, the user might be interested in the Monte Carlo truth. This will imply navigational queries, which in the object oriented databases corresponds to the use of path expressions with implicit joins to single-valued attributes (where the particle or vertex objects refer to the corresponding MC_Truth_Particle and MC_Truth_Vertex). This phase is followed by explicit join queries with few range predicates, together with more or less complex mathematical functions to derive properties (which creates new intermediate results). The next operation might imply nearest-neighbor queries of the newly computed results with some other stored data (like vertices).

The fourth and last step consists of the visualization of the results (in form of histograms, tree-like structures, n-tuples, etc.). Group-by and aggregate queries can be used at this level.

Querying does not necessarily stop here: users can go back to the previous steps and reformulate their query.

3.4.2 Example Query

In order to give a more concrete idea of the typical user's query code, we are going to present a query of medium complexity (see Figure 3.5) in pseudocode. In this example, we are going to abstract the physics details and only present the algorithm that performs the data manipulation.

The conditions that are presented in the algorithm are mostly conjunctive expressions. These might make use of user or system defined functions (like geometrical distances etc.).

The query starts with selecting a collection of runs. From these, a second step will retrieve a sub-selection of events according to new predicate conditions.

With this filtered selection, the user starts with the selection of the constructed decay he is interested in. Usually, the algorithm starts by selecting the particles and combining them to form the vertexing. This vertexing can have 2,3 or more particles combined, or even several vertexes can be generated, depending on the type of physics the user is interested in. A system or user defined vertexing function computes the values of the decaying particle. Sometimes, we make use of an operation like determining which vertex stored in the event list of vertexes has a minimal distance from another one. Finally, some of the values, newly computed or not, are stored to be visualized.

As a last step, the visualization tool is fed with the results and displays the information, typically making use of histograms.

```
1)  Declare: List Runs , List Events, List Results
2)  Result is a list of particle1, particle2, Computed Vertex and Vertex
    # Step number 1
3)  while(run=nextRun()) {
4)      if(conditions) Runs.append(run)
5)  }
    # Step number 2
6)  foreach run in Runs {
7)      while(event=run.nextEvent()) {
8)          if(conditions) Events.append(event)
9)      }
    # Step number 3
10) Declare: List Particles and List Vertexes
11) foreach event in Events {
12)     Particles= event.GetParticle(conditions)
13)     Vertexes= event.GetVertex(conditions)
14)     particles=Particles
15)     While (particles.notempty()) {
16)         headParticle=particles.head()
17)         particles=particles.tail()
18)         foreach auxParticle in particles {
19)             if (condition(auxParticle) and condition(head, auxParticle)) {
20)                 Declare: distance=$\infty$ and MinVertex={}
21)                 computedVertex=ComputeVertex(headParticle,auxParticle)
22)                 foreach vertex in Vertexes {
23)                     if(distant(vertex,ComputedVertex)<distance) {
24)                         distance=distant(vertex,Vertex)
25)                         MinVertex=Vertex
26)                     }
27)                 }
28)                 if(MinVertex.notNULL)
29)                     Result.append(headParticle,
30)                         auxParticle, computedVertex, MinVertex)
31)         }
32)     }
33) }
    # Step number 4
34) Histogram(userSetup, Results)
```

Figure 3.5: Example of a user's query, (pseudocode based on a real query).

3.5 Summary

In this section, we have described the analysis phase by introducing the physics data model and describing the query patterns.

Since the queries depend mainly on the kind of physics the researcher is looking for, they were usually considered unpredictable and complex. However, as we have shown, they tend to show a common pattern. We are going to use this characteristic pattern as part of the solution of the problem defined in the following chapter.

Chapter 4

Problem statement

"The three most important factors that determine the success or failure of a database system are performance, performance, performance!"..."at least one of these three references to performance implies that of end-users when interacting with the system to access data, i.e., user productivity." (for researchers) "twisting their way of thinking so that it fits that of the available systems is simply something they are not willing to spend time." Yannis E. Ioannidis.[62]

In the previous chapters, we have described the context of HEP experiments and explained in detail the area of physics data analysis where we want to make our intervention.

In this chapter we concisely explain the problem between the end-user, the physicist, and the present HEP query systems, and therefore introduce our resulting motivation for this thesis.

4.1 The Problem

As we have explained in the last chapter, the analysis queries, which are complex and apparently issued only once, show some common patterns in reality. This situation justifies the usage of flexible query systems that explore these patterns to query the physics data stored in order to improve user productivity.

As already mentioned in the context description, the development process of the analysis frameworks was very unstructured. Forced by the circumstances, the users became developers, partially because of legacy systems and partially because no software engineering solution existed that tackled the problem from its roots in a structured way. In fact, the result was the development of frameworks that do not provide data independence, showing complex data structures and schemas without hiding the physical layer details. Typically, these frameworks make use of several GPLs and a multiplicity of complex entry points. In other words, they have complex interfaces.

The current systems are disadvantageous for the three types of system actors involved: normal users, system expert users and system developers.

Normal users, or non-experts, are usually physicists willing to do physics analysis without any background on the analysis systems implemented by the experiment's experts. Generally, they are dissatisfied since they are usually not very good at programming. They do not have the necessary background for performance optimization at any level. Hence, they spend too much time with learning, coding, producing both semantic (algorithms) and syntactic errors and waiting for the results. Thus, they are distracted from physics.

Experts are characterized by having a deep knowledge of the experiment-specific schema. They are experienced in the framework internals and (usually) master the programming language and paradigm. They expect from the system flexibility and expressiveness. Generally, they cope with the current situation, but with the growing complexity of the systems, coding for analysis is getting more and more time consuming .

Generally, for developers or system maintainers the work is complex because there are no abstraction levels. This means that any produced changes affect the whole chain, and implies that the users tend to reject changes. Like in other engineering projects where it is difficult to design modular software, efficiency and performance problems are not easy to solve. For these system actors, which are no experts in physics, it is very difficult to change the situation since the documentation about the domain is very poor and sometimes contradictory. Use cases are not clear without a profound understanding of the physics involved, with the negative consequence that only few serious studies on query patterns can be found (see [56]).

The consequence of the described situation is a lack of efficiency in the analysis process. As we are going to describe in the next section, there are important tasks that are time consuming and which depend directly on the user's skills.

As a summary, we can say that scientists analyzing HEP data are often distracted from their real work because they have to learn many details on computer science that are completely unrelated to physics. Thus, the analysis of data generated by detectors in High Energy Physics (HEP) experiments can be a tedious, inefficient and cumbersome chore. This problem is very well known and mentioned by the experts in the field, so we want to tackle it in this thesis.

4.2 Time Consuming Querying Process

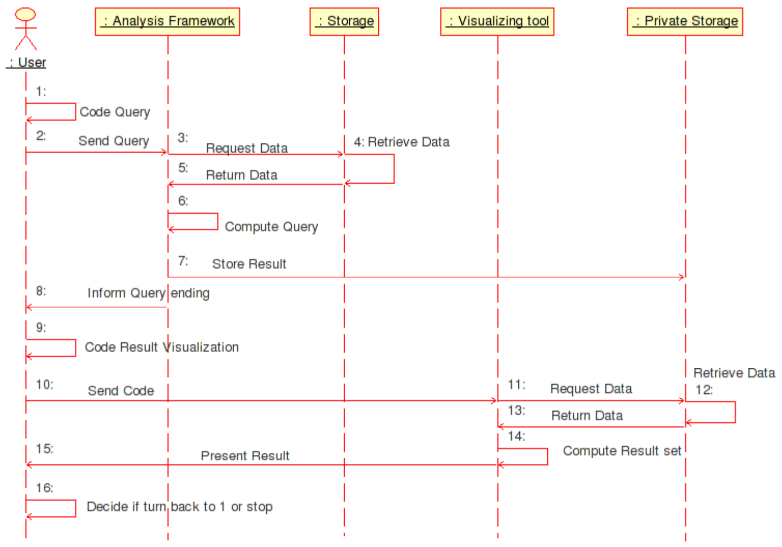


Figure 4.1: Sequences of query steps

If we try to track productivity bottlenecks and, consequently, propose changes to improve the situation, we should look at the whole analysis process and understand which parts are more time consuming. This way the weaknesses can be more easily pinpointed.

We use the term end-user for both normal and expert users, which are referred to in the last section.

Giving an overview of the analysis steps, we can roughly depict a sequence process like in Fig.4.1. We can observe that the time spent on this activity changes widely because there are so many different steps involved. The total time spent on analysis depends on the complexity of the query, the experience of the end user in programming, the programming environment and the execution of the analysis frameworks.

From the end user's perspective the total time spent consists of:

- time spent learning the programming language - ranges widely, depending on the user, but our experience shows that this tends to take between 1 to 3 months;
- time spent with the analysis framework - usually, a month is necessary;
- time spent programming the complete query - ranges from taking three days to two weeks;
- time for debugging syntax errors and semantic errors (bad algorithms) - some hours or some days.

The first and second estimation can only be applied to the normal physicist, since the expert should spend time close to 0 . The third and fourth estimation should be close to the lower bound for the expert users, and closer to the upper bound for the normal physicist.

On the other hand, the system spends time with storage and query computation, depending on the size of the data set. In Hera-B, this used to take from 3 hours up to three days. Additional time is spent with the communication network, the data replication and the visualizing tool, but, as this is of no importance compared to the size of the rest, we can simply ignore it. Although the user bears the responsibility for this, we also consider the time for the execution of inefficient algorithms. Here, time from three hours up to three days is lost because the result of the query is always given at the end of the execution, and the user does not have access to intermediate results to realize the problem.

The direct conclusion from this is that there is too much responsibility for the performance of the system on the user's side. With the state-of-the art technologies and methodologies, it is a very difficult and speculative task to estimate how long it will take to run a query.

4.3 Motivation for the Thesis

The present situation is not satisfactory, especially with the growing complexity of the HEP systems and data storage requirements.

Given the list of problems described, this motivates the introduction of a solid design methodology. It provides the HEP community with a way to develop a robust solution where a flexible query system for this specific domain is produced. There was no previous attempt to tackle this problem, which constitutes a challenging motivation for this thesis.

The solution we search should solve the major problem of the user, lack of productivity, by simplifying the way he writes his queries. In other words, the new approach should increase performance by reducing the burden of the user of being responsible for the optimization, it should improve the learning curve,it should reduce the error generation rate without losing flexibility and expressiveness, and, finally, it should reduce the query production time.

We can expect immediate benefits from the required solution. The framework that will be proposed will serve as a guideline for future systematic studies on how to optimize efficiency of the system and reduce bottlenecks in the analysis process. This way, developers should have a well-designed framework, where they are able to increase the software performance (with better efficiency), without interfering with the user's activities.

4.4 Summary

Scientists analyzing HEP data are often distracted from their real work because they have to learn many details that are completely unrelated to physics. Thus, the analysis of data generated by detectors in High Energy Physics (HEP) experiments can be a tedious, inefficient and cumbersome chore.

This means that they have the main responsibility for producing optimized code for the analysis tasks.

This problem is very well-known in the area. To our knowledge, until now no real attempt has been made to tackle the problem in a comprehensive and methodical manner.

The main highlights from this chapter can be summarized as follows:

- State of the art: Analysis too cumbersome and inefficient
- Motivation for our work: Build a solution that introduces a methodology to increase productivity and performance in HEP data analysis.

In the next part, we introduce some software engineering concepts and computer science tools that is used throughout the rest of the thesis.

Part II

Preliminary Concepts

Chapter 5

Query Systems

5.1 Introduction to Query Systems

In the previous part of this thesis we have explained the context of HEP experiments and the problem it is facing with the current solutions for analyzing/mining their data. The need to increase the user's productivity motivates our intervention in the traditional HEP query systems.

In order to understand what are the common approaches, from Computer Science and Software Engineer, that best fit into our requirements we decided to proceed with a survey of the area. This helps us to decide on what concepts we can reuse for our solution.

Generally, we can define query systems as facilities to process requests for information from a database. There are two ways to access the data: programming languages to write application programs, and query languages.

In many modern database systems, the user has to make requests for information in the form of a stylized query that must be written in a special query language. This language can be used to interactively interrogate the database and retrieve useful information.

The user interaction with the database includes four main tasks: schema definition, query formulation, data update and data visualization. In this chapter, we concentrate on surveying the different general approaches to the formulation of queries, and we will discuss the benefits and drawbacks of each solution. We finish this chapter by detailing some topics which have to be taken into account when developing Visual Query Systems.

5.2 Query Systems Taxonomy

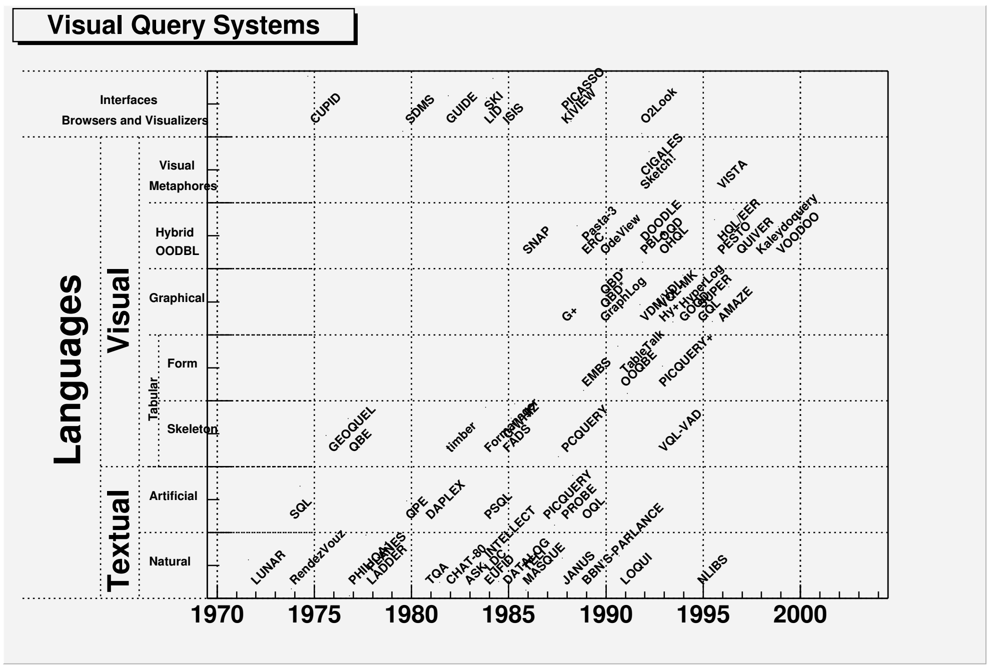


Figure 5.1: Categorization of existing query systems since 1970

Various types of query languages have been developed to interact with storage bases. In order to better understand the work already done in the area, we have investigated a representative set of query languages and categorized them by their paradigm.

Our proposed categories that we will further detail in the next sections are:

- Textual languages:
  - Natural Languages (known by everybody)
  - Artificial languages(learnt and known by specialists):

    Pure textual languages.

    Textual languages with graphical result
- Non-textual languages:

  - Tabular languages:
    * Skeleton
    * Form
  - Graphical languages
  - Visual based languages (metaphor based)
  - Hybrid

- Visual database interfaces

In Fig.5.1 we categorize some examples of query systems. This chart is not exhaustive, for instance, we do not specify XML languages and we also do not mention languages developed from 1998 on. Nevertheless, we consider these languages to be already representative of the different categories proposed.

In order to be able to compare the different alternatives of query systems, when analyzing them, we will take particular attention to the following base comparison criterias:

- Expressiveness - Is the able to produce complex queries?
- Easy to learn - How fast is the user able to start using the language in its plenitude?
- Syntax error free - How easy is it to produce syntax errors? (e.g. misspelling)
- Semantics error free - How easy is it to produce queries that do not do what the user thinks it does?
- Small conceptual distance- How close are the representation of the data entities, language primitives and their manipulation to the way the user conceptualizes them? Does the language force the user to think about this aspects in a different way that he had conceptualized them?
- Memorizable - Can the user easily remember the language syntax?
- Easy to use - Does the user gets confused with the language while using it?

- Non-ambiguous - Can a query have multiple interpretations for the user?
- Formalizable - Can we formally express the language's semantics?

A summary of all this characteristics according to the different query systems can be found by the end of this chapter in table 5.5.

5.2.1 Textual Query Languages

Textual query languages can be either natural or artificial.

Natural Query Languages

The natural query system allows the user to access information stored in a database by requests in some natural language (text through keyboard input and/or voice recognition). An interesting description can be found in [16].

Advantages:

- There are no artificial languages to learn (because queries are formulated in user's native language).
- These languages are better for some questions (negations and quantifications).
- The context of the dialogue is supported.

Disadvantages:

- Linguistic coverage is not obvious and is hard to remember.
- Linguistic has often conceptual failures, meaning that there are a lot of ambiguities still to resolve.
- The user assumes intelligence.
- These languages usually imply a tedious configuration.
- The computer is an inappropriate medium for this kind of languages.

The state of the art in this field is a great deal of *R&D* in some areas (e.g. dictionaries, parsing, etc.). But scientists still do not agree on a common theory or technique for this area.

Artificial Query Languages

Relational calculus can be considered as a formal query language based on mathematical logic, and queries in this language have an intuitive and precise meaning. Relational algebra is another formal query language, based on a collection of operators for manipulating relations, which is as powerful as the calculus [27].

Based on the relational algebra, there exist textual languages like the declarative languages SQL, or QUEL, and the functional language DAPLEX[86], etc.

Although OODBs exist already since 1986 [70], they got the first query language, OQL[3], not until 1994. Deductive databases, on the other hand, are a combination of a conventional database containing facts, a knowledge base containing rules, and an inference engine which allows the derivation of information implied by the facts and rules. Commonly, the knowledge base is expressed in a subset of first-order logic languages like Datalog.

In the category of artificial query languages, we will also consider the extensions to textual query languages with visualization techniques (very often used for geographic applications). The query is described textually and the system retrieves its result set in visual format. Examples of these are: GEO-QUEL, Query-by-Picture Example (QPE)[26], PSQL, PROBE, PICQUERY[64].

We do not consider textual query languages with visualization techniques included into the interfaces category (described later). The main reason is that while with the former languages the user can express an elaborated query, in the later category, queries have a fixed simple pattern (therefore are not very expressive).

Advantages:

- Besides the formalization, one of the main advantages of textual languages is the reduced ambiguity.
- Existing query languages typically allow to work on the logical level, but not on the conceptual level.

Disadvantages:

- These languages rely on the user's memorization of their syntax.

5.2.2 Non-Textual Query Languages

Non-textual query languages, also called direct manipulation languages, usually imply visual query systems (VQS). Those systems make use of visual query languages (VQL) which express the request visually with a set of defined operators. These languages make use of the visibility of the objects of interest and their direct manipulation.

VQSs try to make it easy to deal with the logical model. They make use of VQLs to get closer to the mental model, which is a difficult task, since at two dimensions the ambiguity increases greatly, compared to the one dimension of text queries. In order to pursue this goal, the language should be characterized as follows: it should be clear (without many visual objects), easily readable, simple, and unambiguous (from the point of view of the interpretation by a computer program).

Presently, there is a very active research on visual query databases for formalization, user-interaction techniques and expressiveness. In [25], there is an exhaustive and systematic study of VQSs for querying traditional databases that deal with alphanumeric data.

General advantages:

- There is less distance between the user's mental model of reality and the representation of reality proposed by the computer.
- The basic functionality of the interaction is easy to learn.
- Highly efficient also for expert users, mainly because of the possibility of defining new functions and features.
- The rate of semantic and syntactic errors is significantly reduced.

General disadvantages:

- These languages are more difficult to design. A visual query might not have a unique translation into a textual query.
- They are more difficult to implement.
- Some types of query languages show a lack of formalization, in contrast to textual query languages.

- As for textual query languages, but specifically for visual languages, systems dealing with image data, non-structured text data, geographical data and physics data have different characteristics. The system must deal with the different kinds of data, and its data model, in distinct ways.

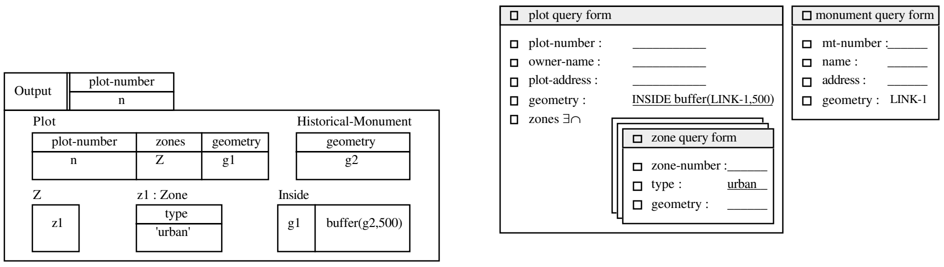
A wide variety of visual query languages have been studied over the years, each designed for a particular data model. For instance, we can find visual languages for specific applications like temporal databases, hypertext systems, statistical databases, geographic databases, video databases, etc. Usually, the data these systems have to deal with ranges from image data, unstructured text data or geographical data to alphanumerical data, each having different characteristics.

In order to find interesting properties we have categorized different VQLs according to their similarities into: tabular languages, graphical-based languages, metaphors and hybrid languages. We will specify each of them in the rest of this section.

Very shortly, we can say that the first successful visual query languages were the tabular ones, based on the relational data model and ER modeling tools. The next generation of visual languages to appear were the graph-based ones, characterized by their great expressive power and their formalization strength. They were, however, awkward to use since they were strictly bound to the logical model and did not try to deal with the conceptual model. Graph-based languages used both relational and object-oriented models. Other general visual languages were the metaphor-based that dealt with the conceptual model but lacked the flexibility and formalization of graph-based ones, especially because of the fact that they tend to be ambiguous. Finally, hybrid-based languages, which use the object-oriented data model, tried to pick the best qualities from the different approaches and are nowadays the most promising languages.

Tabular Languages

They are considered to be the first visual query languages that brought the concept of user-friendliness and flexible querying into the evolution of artificial textual languages.



(a) VQL[90], Skeleton-based (b) OOQBE[88], Form-based

Figure 5.2: Example of tabular languages taken from [18].

- *Skeleton-based* - Each relation is represented by a two-dimensional skeleton in which the column headings show the names of the relations and the names of the attributes. The query is expressed by filling the skeletons with a combination of variables, constants and keywords that give an example of the possible answer. An example can be found in Fig.5.2(a).

  Query-by-example [94] was one of the first attempts which analyzed querying in a non-textual way (used as a basis for many commercial database systems). It is very convenient for simple queries, but awkward for complex ones. It supports transitive closure, which is an extension of relational query languages. It has been extended to deal with aggregate queries.

  Another formally defined language of this kind, VQL[90], makes use of different data models: relational, extended relational and object-oriented.

- *Form-based* - Seen as an evolution of skeleton-based languages making use of the multi-windowing technology. Each object type has its own dedicated window. In this window, the user can see menus of commands, lists of predefined constants or menus of operations just by clicking the mouse on buttons or icons. For a query, the user fills out the forms of the related object types. An example can be found in Fig.5.2(b).Examples of this languages are: G-WHIZ[77] for the functional data model where recursive queries are allowed, OOQBE[88] (Object oriented query by example), and PICQUERY+ [64].

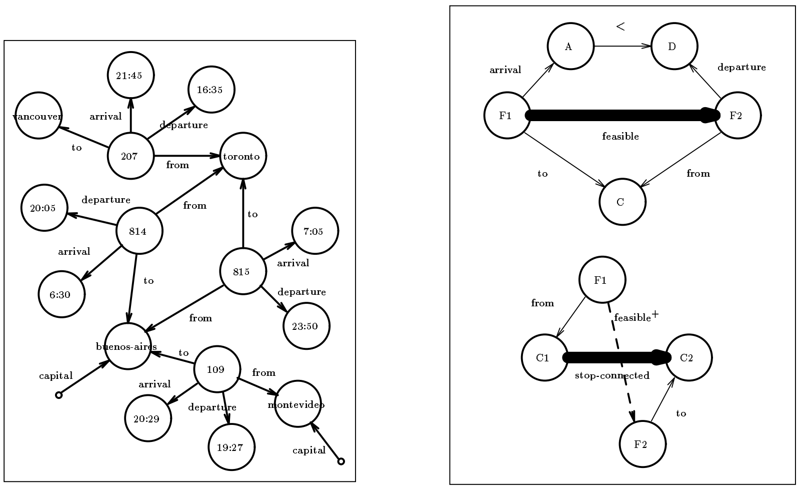
Advantages:

- Generally, these languages are user-friendly (form-based more than skeleton-based). It is more convenient than just typing on the keyboard.
- These languages have less things to learn. It is not necessary to remember the database schema, and the user is aware at any point how to navigate through it.

Disadvantages:

- Complex queries have an awkward representation. Some join operations must be expressed by means of variables, which is a source of mistakes.
- These languages have very poor visual representation of the data model concepts.

Graph Query Languages



(a) Graph representation of a flights schedule database (b) Query of feasible flight connections

Figure 5.3: Example of GraphLog [37].

Graphical query languages correspond to queries that are actually graphs (graph-theoretic perspective). It is based on the use of symbols which represent the data model concepts. These symbols, such as rectangles, circles and arrows, are pure graphical conventions without any metaphorical power. As a consequence, they need to be explained and memorized.

GQLs are more suited to be formalized, given their precise mathematical structure (i.e., graph). This formalization makes it possible to compare them with other query languages and to precisely evaluate their expressive power.

The database schema is usually visualized by a graph where nodes represent the objects and arrows the relations between them. With their knowledge acquired by schema browsing, end-users express their query following a mode which varies with the considered language:

- The user builds a query graph in a separate window. This graph uses the symbols of the database schema. The user can also use some new conventions in order to visualize a selection predicate, for instance, or to mark the elements which must be printed in the query result.
- The user directly marks on the database schema graph which elements are relevant for the query, and then he also uses different menus to to specify the selection criteria.

The majority of these languages is based in the context of visualization in deductive databases. The semantics of the graphical primitives is given as a translation to Datalog. Mostly, they were declarative and meant to query graphs. Examples of these are GOOD[58], GraphLog[37], Hyperlog, VDM/VDL, VQL[72], G2QL[53], etc.

Other kinds of graph-based languages make use of graphs purely for specifying primitives that can be mapped to textual language commands. The use of graphs is mostly related to the formalization power and, consequently, to the unambiguity that it provides. The semantics of the graphical primitives is given as a translation of statements of an object-oriented programming language supported by the underlying database. There are several examples of this last type of graph-based languages querying both entity-relationship and object-oriented models. Listing them we find: SNAP[22], QBD^*[17], QBD^*, VQL-MK[72], ERC[42], SNAP[22], GQL[76], that use the functional querying paradigm.

Hygraphs are an extension to the graph theory, incorporating blobs in addition to edges. A blob relates a containing node with a set of contained nodes. It is possible to assign semantics to the relationships represented by blobs. Some other concepts were added through colored graphs (G-Log[78]), where the body of the rule is colored red and the head green. The same directed labeled graphs are used to represent database schema and instances. The nodes of the instance graphs stand for objects, and the edges indicate relationships between values. Examples of this type of languages are Hy+ [36] and G+[41].

Main advantages:

- These languages are more formalizable.
- They make better use of the visual medium than tabular languages.
- These languages are powerful enough to express more complex queries (transitive closure, recursion and computation of paths in directed graphs).
- It is a natural way of querying schema intensive domains, where we find a large number of classes and many interrelationships between them.

Main disadvantages:

- Requires experienced programmers to exploit its power, since it uses of a lot of symbols that are only graphical conventions.
- The visual notation does not have a direct meaning, (a triangle means something that is defined by the person that designed the language). Instead, they have underlying concepts that are not perceived in a metaphorical way.
- They are costly to design and implement.
- Complex queries very easily become unreadable.
- These languages need to be explained and memorized.
- The semantic distance between the real world and the database universe is still too big for the normal end-user/non-programmer.

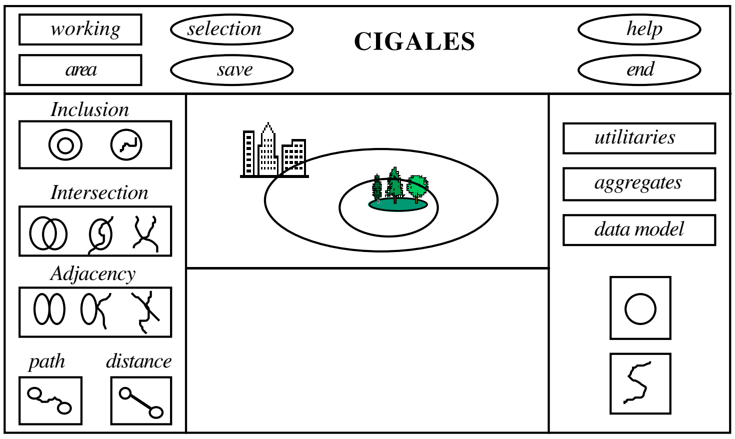


Figure 5.4: Cigales [79] Metaphor-based. Uses the map metaphor. Example taken from [18].

Metaphor-based Visual Languages

This kind of visual languages uses metaphors to show the concepts. Metaphors take the mental model of the end-user into account. An example of these languages is VISTA[20], where the metaphor is a room with objects to manipulate inside, or Cigales[79] (see Fig.5.4), representing a map.

The way the user expresses his query varies widely and mainly depends on the metaphor chosen by the language developer.

Advantages:

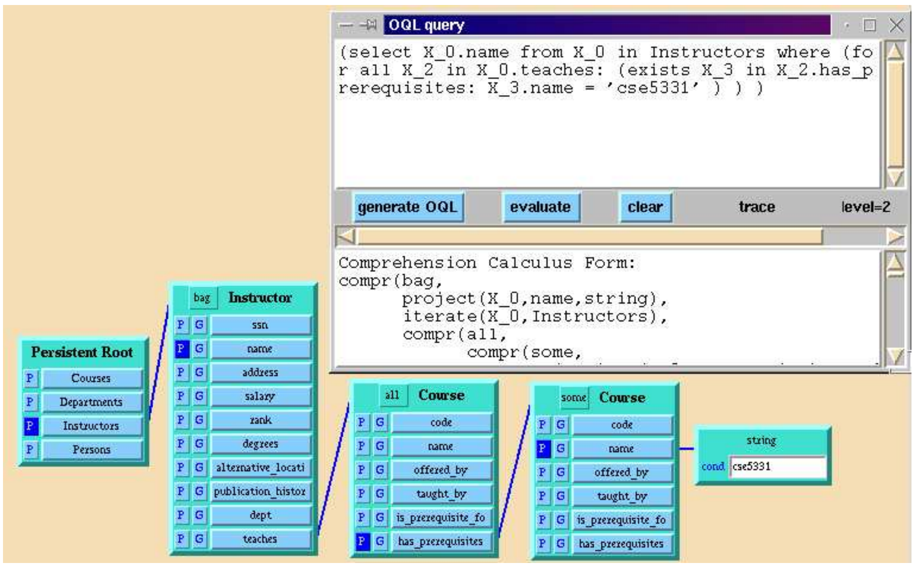
- These languages offer an intuitive and incremental view of the queries.

Disadvantages:

- It is very difficult to find an adequate metaphor for a problem in a given context.
- There is no proper software engineering methodology to design such a language.
- The risk of failing as a query language is very high.
- Very often, a multidisciplinary development team (computer scientists, psychologists, designers, etc.) is required.

- Usually these languages have poor expressive power.
- Very often these languages suffer from execution inefficiency.
- The system might have multiple interpretations for a query.
- These languages have difficulties to handle objects that do not necessarily have a visual representation (like arrays, lists, stacks, and application-oriented data types like forms and documents).

The Hybrid Approach



```
(select X_0.name from X_0 in Instructors where (fo
r all X_2 in X_0.teaches: (exists X_3 in X_2.has_p
rerequisites: X_3.name = 'cse5331' ) ) )
```

```
Comprehension Calculus Form:
compr(bag,
        project(X_0,name,string),
        iterate(X_0,Instructors),
        compr(all,
              compr(some,
```

Figure 5.5: Hybrid language VOODOO[50] (based on OQL).

This category of languages uses the power of formalization of graphs (defining the abstract syntax with them) and the concrete syntax (making use of combined menu-based and simple metaphor-based solutions) to reduce the mental gap.

The underlying principle of these systems is to provide a visual representation of the data residing within objects, and to offer visual operators for navigating through related objects. In other words, there is a direct correspondence between each window and an object in the underlying database. Two kinds of interactions are usually supported by these object browsers: navigation within a collection of objects, and navigation between objects by the way of their relationships.

In these systems, it is also very common to use the filter flow metaphor proposed in [84], where the water flows through a series of pipes and filters and each filter lets through only the appropriate items. The layout of the pipes indicates the relationships of $\vee$ and $\wedge$.

Examples of these languages are DOODLE [40], Kaleidoquery [74], OdeView [2], VQL-VAD [90], SNAP [22], PASTA-3 [67], PESTO [23], QUIVER [71], VOODOO [50].

Advantages:

- The structure of the database classes, attributes and relationships is readily available for the users. Usually, it is just "one click away" from the layout.
- For non-programmers, it is easy to memorize the language and to learn the schema.
- The way to deal with filter predicates in the flow metaphor is close to intuitive.
- Designed to deal with a general purpose query language, usually can be mapped into object-oriented query languages.

5.2.3 Visual Database Interfaces

The main task of these systems is to perform schema browsing, or result visualization. They are inflexible and are mostly tools for visualizing a database, but do not contain a formally defined query language.

With this kind of system, the users can access the information easily and quickly without having to give an exact description of it or where it is stored in the database. There are four standard operations common to these applications: structuring, filtering, panning, and zooming. This means a fixed query pattern of selected project queries.

We can find examples of visual interfaces implemented on top of the relational model to browse the schema: CUPID[69], SDMS[60], GUIDE[93], LID[52], ISIS[57], SKI[65], etc. As far as interfaces for object-relational models are concerned,we have: PBL+, DAA+ (on top of SUPER)[45], DGJSA (on top of ODEVIEW[2]), PESTO[23], KIVIEW[73], LID[52], etc.

Advantages:

- These languages are easy to use and very good for occasional, unexperienced users, with simple, repetitive requests.
- The fixed set of queries, with a very well-known query pattern, makes the system easily optimizable.

Disadvantages:

- Do not have a properly formulated query language. As a consequence, it is not possible to formulate complex, elaborated queries.

5.2.4 Summary of Features

A comparison of all the mentioned query systems is summarized in Fig.5.5. From that, we can conclude that hybrid systems manage to gather benefits from other visual languages. They are potentially the best approach for developing a new language for non-experts on programming. They can can be learned quickly and have reduced error rates. As we will see in the following chapters, we have taken these considerations into account when developing our own solution.

5.3 Building a Visual Query System

After we have decided the type of query system that is more appropriate to our goals, we now have to consider the implications on its design and development.

A VQS has the same goal as any user-interface application: it is meant to simplify the user-system interaction. A VQS includes a VQL and a variety of functionalities to facilitate man-machine interactions. When building such a system, three major topics must be covered: schema display and navigation, query creation and result visualization, query optimization and evaluation.

	Textual		Visual				
	Natural	Artificial	Tabular	Graphical	Metaphor	Hybrid	Interfaces
Expressive	√	√	√	√		√	
Easy to learn					√	√	√
Syntax error Free			√	√	√	√	√
Semantics error Free					√	√	√
Small Conceptual distance					√	√	√
Memorizable					√	√	√
Easy to use					√	√	√
Non-Ambigous		√	√	√		√	√
Formalizable	√	√	√	√		√	

Table 5.1: Query languages comparison

5.3.1 The Visual Language

For the development of an effective language for visual interaction with a complex knowledge base, there are four major requirements:

- There should be given a set of visual language primitives, i.e. a set of graphical icons that constitute the alphabet of the language.
- With this language, it must be possible for the user to easily combine the primitives in different ways to create valid queries. This means that a syntax and grammar for combining various visual primitives has to be specified.
- Special symbols have to be designed which represent query targets, database variables and logical constraints.
- For ease of conceptual visualization, it is necessary that the visual query language developed for a particular data model consists of primitives that conceptually (and visually) parallel the schema representation mechanism.

5.3.2 Human Factors

VQSs are part of a special subset of user interfaces. This means that an human-centric development of the software must be used while developing them. The emphasis should be on the user comfort, by providing an accessible interface, and on its usability.

The language designer should always design the language with a strong user's feedback, trying to understand how the tool is going to be perceived, learned, and mastered. In order to achieve a successful system, the future users must be properly classified into the different kinds of possible categories, and their specific requirements identified. The engineering life cycle must include a proper validation of the language through usability evaluation tests. This topic will be deeply discussed in chapter 10, which is dedicated to the evaluation of our proposed language.

5.4 Summary

VQLs exist to make it easier for the end-user to deal with the database systems.

The main ideas we want to take from this section is that hybrid visual query languages are beneficial compared to others:

- They reduce the need to previously know the database schema, attributes and relationship structure before writing the query. This means a short learning phase.
- They reduce the problem of semantic and syntactic errors, meaning better productivity.
- They get much closer to the mental model than other languages.

Chapter 6

Domain Specific Modeling

In chapters 2 and 4 we have observed that one of the reasons for the problem we have in hands, with the physics data analysis, is the lack of abstraction layers. In addition to that, the solution of providing the end user with a general purpose language is problematic for several already discussed reasons. In Software Engineering, one solution for this kinds of problems, when the user has to develop his own software products (and probably is not skilled enough), but we want to increase his productivity, is to make use of the concept of Domain Engineering and develop a Domain Specific Language.

In section 6.1 we start by giving an introduction to the general idea of Domain Engineering. Then, in section 6.2, we proceed by giving an overview of the modeling strategy required. Following that, in section 6.3, we shortly discuss the engineering process and in section 6.4 we highlight the advantages and disadvantages of domain specific languages. We finalize with section 6.5 by observing solutions in HEP that, although unstructured, could be considered to be remotely related to domain modeling but that did not lead to any learned lessons.

6.1 Introduction to Domain Specificity

In order to cope with markets that evolve at a rapid pace, where it is necessary to bring solutions to market quickly and to constantly develop new software products, a procedure different from the conventional software engineering methods is necessary.

Domain engineering approaches the problem by increasing the accessibility of the information systems, giving the end-users the opportunity to develop programs. This can only be achieved by raising the level of abstraction, making common parts explicit, and, at the same time, limiting the possible design space to a single range of products. In other words, it defines a family of applications (instead of developing products individually) and a production facility (Domain Specific Language DSL, generators, tools). The models generated are made up of elements representing things that are part of the domain world, not the code world.

The direct consequence of this is that less training is required to use a process, which speeds up the software development process considerably.

Family-specific modeling languages make product families explicit, shift the abstraction level from designs to the product concept level, and allow for a fast and automated variant generation. The language follows the domain abstractions and semantics, allowing developers to work with concepts in their particular domain.

DSLs stand in contrast to general purpose languages (GPL). While the first are dedicated to a particular domain or problem, being small and usually declarative, the second can be used generally for a wide field of solutions, using imperative, functional or object-oriented styles. A GPL is usually suboptimal for specific applications, especially where it is used by people not trained as software engineers.

We can find implementations of domain-specific languages in areas such as robot control [39], VLSI design [19], CASE tools [85], and GIS [80]. To our knowledge, no DSVL exists for the analysis of data collected in physics experiments, or other HEP purposes. An interesting summary and inventory of references to DSLs can be found in [91].

6.2 Modeling Strategy

Domain-specific modeling works on the problem level instead of the solution level. This means that models are made of elements representing things that are part of the domain world and not the code world. It is meant to automate a large portion of software production.

As defined by OMG[75], Domain-modeling engineering exploits a four-layer meta-data architecture (see Fig.6.1):

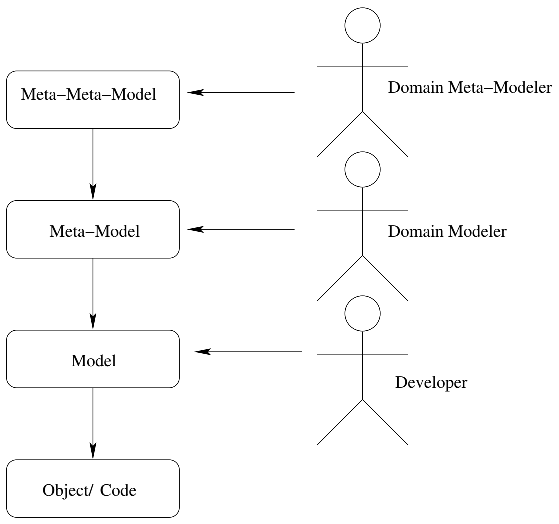


Figure 6.1: Domain-specific development

- **Meta-meta-modeling layer** - Is the definition of a tool that supports the domain-specific language modeling.

- **Meta-modeling layer** - This features the implementation of the domain-specific modeling language, for instance a language for robot control or for the generation of software for mobile phones. The design tool for product families reduces the cost of creating domain-specific tools by allowing the domain expert, the domain modeler, to specify the syntax and semantics of a language in the form of a meta-model and by creating the supported family members automatically.

- **Model layer** - The domain user, developer, uses the domain model language to specify his application using concept structures. This means that the developer (or user of the DSL) is able to model the family member. For example, the user models the new robot control software for an automobile painting procedure in a new production line, or the new control software for the new mobile hardware.

- **Object layer (or Instance layer)** - This layer represents the automatic code generation. This implies the existence of a domain-specific component library, and, of course, the automatic code generator. The model specified is mapped to code that calls the components.

Domain modeling should not be confused with modeling languages like UML, since those are based on code structures and make use of semantic concepts of programming languages. The users usually have to make error prone mapping of domain concepts into UML and then to program code, which requires a good knowledge of software engineering.

6.3 DSL Engineering Process

During the analysis phase of the development of a domain specific language we must identify the problem domain, gather all relevant knowledge in this domain and cluster this knowledge.

After this preliminary analysis, we must proceed with the family- oriented software development. This entails defining the family with its terminology, commonalities and variabilities. A good introduction to the family analysis and the definition process can be found in [38]. Basically, we identify and use the abstractions that are common to all known, or predicted, family members, and we structure the design to allow changes. Sources of abstraction are the terminology used to describe the family and assumptions that are true for all family members. To identify the scope of the family, the analysis must include predictions of how family members will vary.

Implementation usually involves constructing a library that implements the semantic notions. In the following, we build a compiler that translates DSL into a sequence of library calls.

6.4 Advantages and Disadvantages

From [89], comparing the benefits of DSLs over GPLs, we have:

- Familiar program notation - DSL use domain notations, which makes the language more readable, and its specification more accessible to the domain users (normally non-programmers).

- Design reuse - The user has a well-defined path to develop his application. This is convenient since the code needs to be tested only once.
- High-level abstraction - The users deals with constructs at a higher level of abstraction. This way the user does not have to deal with error-prone and low-level implementation details. As always, more levels of abstraction reduce complexity, shortening the development and the testing phase.
- Clear concise program specification - program specifications can be by big factors smaller than the corresponding specification in the GPL.
- Program checking - As a result of using a restricted language, it is possible to catch some semantic errors which cannot be caught by with a GPL compiler.
- Efficient execution - DSL programs can have at least the same performance as in common general purpose languages.
- Reduces time and effort drastically - There is a payoff at the development and production of family members. DSLs enhance productivity, reliability, maintainability and portability.

In addition to this list, we can say that the target code, as it is automatically generated, does not contain syntax and logic errors. This is determined by the semantic and modeling rules captured in the metamodel.

The obvious drawbacks of this approach are mainly related to the fact that DSLs are difficult and costly to build, since each requires its own significant design and development effort, and each domain supported by a tool is specific to a certain type of problem (limited marketing).

DSLs can only be developed with the involvement of experts in the specific field that they were developed to, since in most cases, the domain is very complex. This development is only justified if it can be expected to generate a number of family products. Thus, the developer must evaluate and balance the costs of designing each tool from scratch or using a DSL.

As stated in [38], the success depends on how well the software engineers can predict which family members will be needed. The concept of family member is not well formalized, there are no rules that enable engineers to identify families easily, the prediction of variations is difficult and implies spending time for family analysis during the development process.

6.5 DSL "Attempts" in HEP

In order to avoid to "redo the wheel", we had to determine if any domain specific approach has been taken before in HEP analysis.

We have observed that Analysis frameworks like ROOT[81], specifically designed for this domain, do not hide the internal complexity of the library of functions from the query code programmed making use of a GPL object-oriented programming language (C++). With time, the libraries become larger and more generic. The consequence is that the usability decreases because of the multiplicity of entry points, parameters and options offered.

From our research we also found out that some experiments which tried to reduce the problem of the general purpose approach, have used rudimental textual domain- specific commands. We have KAL in the experiment ARGUS/DESY[5] (from the early 90s), or εZ in ZEUS/DESY[29] (from the late 90s and early 00), or even ATGEN in ATLAS/CERN[34] (still being built). This shows that the question of how to improve the user's productivity in HEP is already standing for long, and developers have been trying to answer it. Unfortunately, almost no documentation has been written about them, and we can not proceed with a thorough evaluation of them. Since they had no methodological approach like the definition of the objects and their operators with the help of an alphabet and a grammar, we cannot call them languages. They are just collections of some common commands, with no formal specification. They were very inflexible and confined to the scope of the experiments where they were developed, and did not lead to their standardization. The main reason of this is that the abstraction was weak. The positive contribution of these tools was to help gathering domain-specific functions in component libraries.

We conclude that introducing a structured domain specific language in this HEP can be considered to be a pioneer idea.

6.6 Summary

Domain-specific engineering methodology comes into play when a family of applications has to be developed by users that are not necessarily software engineers, in a specific domain. It focuses on generating a language that gives the user the possibility to center on what to compute in opposition to how to compute, so that he does not need to be a skilled programmer.

The development of a DSL reduces time and cost involved in the development and modification of a family of tools in a certain domain.

Part III

Tackling the Problem

Chapter 7

The Solution

In this chapter we present our proposal for solving the problem of lack of productivity in HEP analysis, already described in chapter 4.

Our hypothesis, explained in section 7.1, is that we can solve the problem by developing a Domain Specific Visual Query Language. In section 7.2 we give arguments to support this idea. In section 7.3 we define what we expect to obtain as result of a developed solution. Finally, we sketch the services overview of the required system 7.3.1.

7.1 Proposed Approach

The usual way how we can simplify a user's interaction with a system and make it more flexible for incorporating changes is to introduce different layers of abstraction. In the ideal case, we want to be able to abstract the user's point of view (conceptual layer) from the data representation (logical layer) and this, in turn, from the actual data storage (physical layer).

In order to raise the abstraction levels, increase productivity and give experts a clear architecture where it is more easy increase efficiency by tracking new points of optimization in the analysis query system, we propose to introduce a properly defined declarative visual query language (and system) specific to the HEP domain, by means of an adequate development process.

We propose a *unifying framework* for analysis, called PHEASANT (PHysicist's EAsy ANalysis Tool), that distinguishes between the concep-

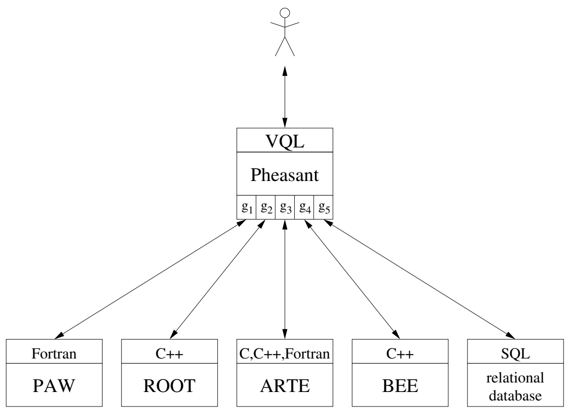


Figure 7.1: *Unifying* framework - The user views his particular analysis framework in the same way as others.

tual, logical, and the physical layer of the data and presents the same view to the user for each analysis framework he is working with. At the conceptual level, this framework features the first declarative *domain-specific visual query language* (DSVQL) for HEP analysis called PHEASANT QL in which physicists are enabled to construct queries using familiar concepts, opening up a new application area. If it is not necessary to know implementation details or certain programming skills, it is much easier for a user to become acquainted with the framework.

At the logical level, we provide a more detailed representation of the data in form of a logical schema. However, this representation still hides implementation details. The visual language queries are mapped onto an algebra which operates on the logical schema.

At the physical level, different (existing) tools can be plugged into our framework via code generation modules (represented by g_i in Fig.7.1). A code generation module translates the algebraic form of the query into the appropriate syntax of the corresponding tool. This way, if the query primitives do not change, developers may introduce changes in the core technology, storage layer, physical model and physical algorithms without affecting the user. With a proper abstraction design the framework can be extended wrapping around new tools (like histograms generators) in a quite elegant way.

7.2 Why a DSVQL?

The concept of a domain-specific visual query language (DSVQL) gathers several other concepts and their qualities into one: visual query languages, declarative languages and domain specificity. The several benefits discussed in the previous second part of this thesis justify the combination of them to derive the solution.

In fact, the language should be domain-specific because in this complex domain, where users have to code their queries hundreds of times to complete an investigation, it is justified to develop a solution that gathers the patterns and data objects into the user's conceptual notions of the domain, and automate the generation of query code. Obviously, a family of products has commonalities that can be explored to perform code reuse. Moreover, the physics community is usually not trained in software engineering. Automating the generation of code releases the burden of writing it. In fact, general purpose languages (GPL) have shown to be difficult for the user in this domain.

We suggest that the language should be declarative, since the user benefits from the fact that no programming logic is involved. As we have seen before in chapter 5, it should also be visual since it is easier to use and learn, and reduces the error rate.

7.3 Expected Results

By introducing the DSVQL, we expect to improve the user's productivity. This is the immediate result of introducing clear abstraction layers. Furthermore, it helps hiding details of storage and efficiency.

An commonly accepted query language for the domain will be beneficial to the end-user. The physicists non-experts in programming will no longer have to cope with different languages in different experiments. They will get no more error-prone mappings to other languages. As a consequence, it can be expected that they will learn the system and design their queries more quickly.

On the other hand, expert users, being used to hack their systems, will have an extra tool to speed up their analysis, without necessarily loosing expressive power.

Finally, developers of analysis frameworks will have a system with properly isolated modular levels at their disposal. This improved architecture will give them plenty of room for system efficiency and design improvements with the extra benefit that produced changes do not affect the rest of the modules and, in consequence, the rest of the analysis chain. The users, hopefully, will not realize the changes, except for the increased efficiency of the system. This means that they do not need to change their query in order to cope with a new system interface.

7.3.1 System Overview

In Fig.7.2 we sketch the general services of the system we want to develop.

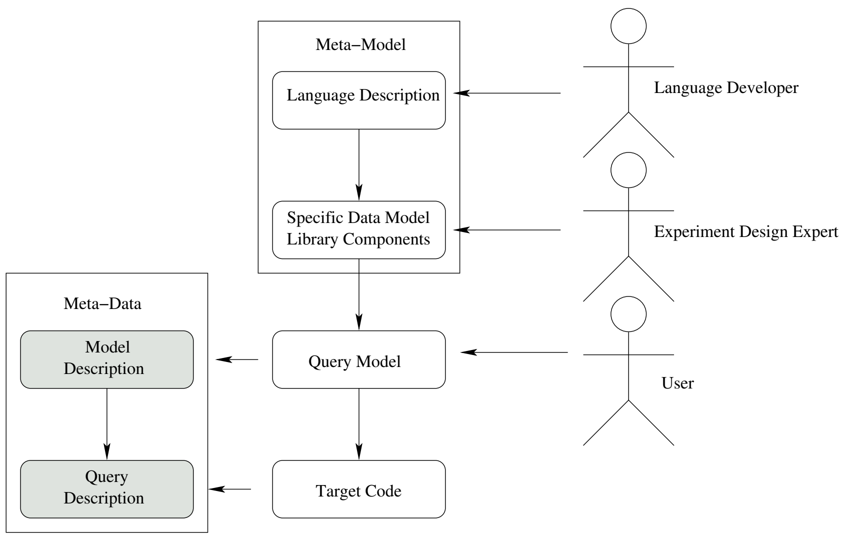


Figure 7.2: System services

The developed system provides facilities for specifying the visual language meta-model. This modeling is done by two actors: the language developer, responsible for describing the language (and that might want to extend the language), and the Experiment Design expert, responsible for specifying the data model and the library of functions available to the physicist.

At the physicist (user) modeling level, the system provides the hybrid visual query language, whose operators were defined at the meta-model level, with characteristics similar to what was already described in chapter

5. This language raises the level of abstraction in such a way that the end users can ignore the implementation of the frameworks and can share their queries (i.e. have a way to talk about the specification of their queries without having to go deeply into the details of the programming environment).

Once the query is modeled by the physicist, the system will generate the target source code, that runs on the target analysis framework.

In order to cope with the domain adaptability and evolution of both the data schema and the library of components, we propose to use a metadata system (in Fig.7.2 represented at the bottom left side containing grey boxes) that deals with the versions of the different query models (keeping track of what versions make a given query valid), the user history, and with the data and component library elements. This concept will not be studied in this thesis, but we propose it as future work instead.

In the following chapters, we will describe how we have developed a language (PHEASANT QL) and a prototype of a framework (PHEASANT) that meets the requirements.

7.4 Summary

In order to answer the question of how to develop a systematic approach to improve the analysis' framework performance by increasing the user's productivity? We propose the introduction of a declarative domain-specific visual query language. This should be implemented by a unifying framework.

- We propose a DSVQL as a way to:
  - Raise the abstraction level
  - Modularize the architecture
  - Structure the points of optimization
  - Have a more usable interface, since it is close to the user's concepts
- Why declarative?
  - The user states the problem and not the solution

- Why visual?
  - More intuitive and easy to use and learn
  - Helps reducing the error rate
- Why domain specific?
  - The query language deals with the physicist's concepts.
  - GPLs are difficult for the user in this domain

The next chapters, are dedicated to describe the design and development of the solution. In chapter 8 we formally define the new language, called PHEASANT QL, and in chapter 9 we describe the prototype framework that implements it.

Chapter 8

Query Language - PHEASANT QL

We dedicate this chapter to the complete description of the new PHEASANT Query Language. The syntax and the semantics of the language are detailed.

8.1 Introduction

Any query language should be specified by means of a formal syntax and semantics. This approach is beneficial since then we are forced to develop both major concepts of the language and the details, leading to a correct implementation. Additionally, the user has a unique and clearly determined semantics for any sentence in the language.

The syntax of a language is a set of rules that define the ways symbols may be combined to create well-formed sentences in that language. The semantics, on the other hand, deals with the meaning of programs, i.e. how they behave when executed on computers.

In this chapter we describe both syntax and semantics of the newly proposed query language, PHEASANT QL[10, 9]. We start by summarizing some concepts of language specification. Then, we introduce the syntax with the notation and alphabet of our proposed language, motivating them with the user's conceptual layer of this specific domain. After that, we specify the semantics of the language, making use of translational semantics. In other words, we define the semantics of our language by mapping it into our own Algebra (some of the operators were based on the work of [51]).

8.2 Syntax

Commonly, the syntax definition of a language is a formalization of its internal structure, called grammar, that lists the symbols for building words, the word structure, the structure of well-formed phrases and the sentence structures. This structure is often formally defined by using a notation known as Backus-Naur Form (BNF). This BNF definition is a set of rules where the left-hand side is a non-terminal, also called structural type. The right-hand side is composed using both terminal symbols and non-terminals that define the structure of the non-terminal symbol at the left-hand side. When describing the grammar of PHEASANT QL in 8.2.4, we will have the chance of detailing this subject more deeply.

8.2.1 Concrete versus Abstract Syntax

Concrete syntax establishes the concrete visual representation of language elements, defining that a certain entity should be represented by a specific geometric shape, defining the layout and spatial relationships. In visual query languages, this is a subject important for the field visual parsing[92], since it studies the recognition of concrete syntax elements. The result of the interpretation of these rules is usually a spatial relationship graph (SRG). This graph will be mapped into an Abstract Syntax Graph (ASG), which contains only the logical structure, abstracting concrete details like distances, shapes, sizes, etc. In this chapter, our language will be defined by means of the Abstract Notation.

8.2.2 Overview of PHEASANT QL

The user's conceptual view of PHEASANT is based on the stream of objects flowing through four major steps. This view, as we are explaining in this section, motivates the design of a specific language's visual syntax for this domain. The underlying logical schema and manipulation of the data is detailed in the section 8.3.1, where we describe the semantic mapping of this language.

When specifying a query, the user has to go through four sequential steps, where one feeds the next. Although they are not linked visually, the user mentally connects the steps' flow. It starts with the operators for data collection, meaning filtering specified sets of *Event* objects, which will "feed" the rest of the query operators and, consequently, the rest of the query steps. The omission of these operators will assume that all *Event* objects from the universe of the stored events will be chosen.

As a second step, the set of *Events* selected will be filtered out by the user's filter predicates on the *Event* attributes. This reduced set of *Events* will serve as input to the third query step of reconstruction and filtering of specified decays. If the Event filter operators are omitted, all the event objects are selected from the previous step.

The query described in the third step looks at the data objects (Particles and Vertexes) associated with each Event and extracts a set of *Decays* for each of them. For the user, a *Decay* is a set of related particles, vertexes and objects newly generated as the result of the description of the declarative query.

The result of this step, the set of *Decays*, will flow to the target operators of the fourth and last step where the result operators are specified. The user will get as a result from his query a *Histogram*, a value of a *Basic_type* (meaning Float or Integer), or, if a result operator is missing, a set of *Decays* to "feed" other analysis tools.

In the next informal description of the user perspective of the framework, we use the notation $\{Event\}$ to mean a set of *Events*, and $\{Decay\}$ to mean a set of *Decays*.

8.2.3 PHEASANT QL Alphabet - Symbolic Notation

In this section, we introduce the basic building blocks or visual operators of our language with the help of a running example. We base it on the query presented in Fig.2.1. In some of the operators we introduce, we have associated with them a second level (indirectly visual) of textual description of parameters like a list of attributes and filter predicates (in a loose approximation this means a projection of a set of attributes and a selection based on a filter predicate in the relational approach). For the full understanding of these operators at the logical level and how they

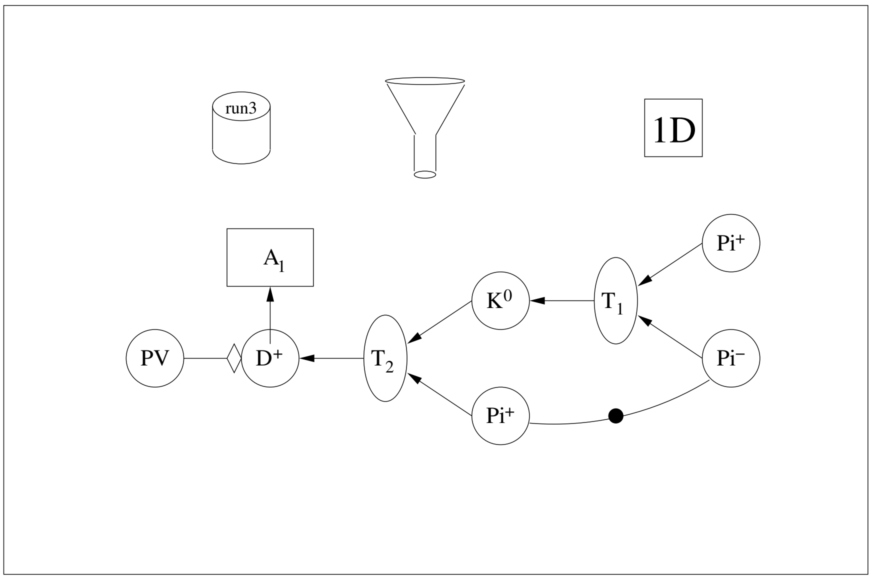


Figure 8.1: Example of a complete query: the D^+ decay

interact with each other, we are going to describe the grammar and the formal semantics in the following sections.

Fig.8.1 shows the complete query, where four major steps are integrated in the visual query sentence.

Selecting Collections

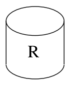


Figure 8.2: Collecting the data in step 1

First of all, we have to decide which collection or collections of event data to use (e.g. Runs, private event collections etc.). This task of selecting the collection objects according to a predicate criteria over the properties of the referred collections, is performed by the collection operator, which is represented by a small disk symbol (see Fig.8.2). Let us assume for a moment that we are only interested in the data from the third run. So, in a first step, we have a collection operator that selects this data for us. This symbol reflects the user's perspective on the `Collection` class entity that are interpreted as collection objects (like in the object-oriented approach). Collections' filter sentences can be composed using a combination of these operators with standard set operators $\cap, \cup$, and $\setminus$. Fig.8.3 shows the signatures of the different collection operators.

The query described in this step selects a subset of the specified collections. This is done by using a filter predicate over the collection. Afterwards, the set of events to which the selected collections refer to are united and passed to the next phase.

In our running example, we have the left operator in the upper part of Fig.8.1 that tells the system that we are interested in the data from the third run. The list of attributes (hidden in the schema) is a set of properties of the `run` like $\{runid, quality, itr, otr, \ldots\}$, and the filter predicates would be for instance $\{runid = 3 \wedge itr = true \wedge otr = false\}$.

Collection ⛁ $\substack{collection \\ pred}$	$\rightarrow \{Event\}$
Union $\textcircled{\scriptsize$\cup$}$	$\{Event\} \times \{Event\} \rightarrow \{Event\}$
Intersection $\textcircled{\scriptsize$\cap$}$	$\{Event\} \times \{Event\} \rightarrow \{Event\}$
Difference $\textcircled{\scriptsize$\setminus$}$	$\{Event\} \times \{Event\} \rightarrow \{Event\}$

Figure 8.3: Signature of the Collection PHEASANT Operators

Selecting Events

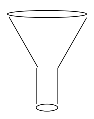


Figure 8.4: Collecting the data in step 2

The second step involves dealing with the set of Events resulting from the first step. Those that are collected in the first step will be filtered out by predicates like 'coasting_beam=true' , or other arithmetic inequalities with physics formulas that make use of the Event attributes (algebraic expressions joined by inequality symbols like $>, <, >=, <=, \ldots$). This way, a smaller subset of events is selected to feed the following steps. Fig.8.5 shows the Event operator signature.

Event	
$\triangledown_{pred}$	$\{Event\} \rightarrow \{Event\}$

Figure 8.5: Signature of PHEASANT Operators for the Event filtering

Selecting the Decay

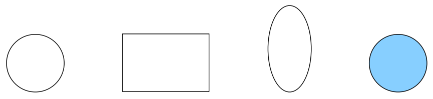


Figure 8.6: Selection, Aggregation, Transformation, Transformation Result

For the third step, that is going to deal with the multivalued data referenced by the Event objects, we need four more operators: Selection, Aggregation, Transformation, and Transformation Result (see Fig.8.6 for their symbols). At this step, the query deals with the input data of one event at a time, dealing with the objects it is composed of. The resulting sets of related objects, the decay, are handed over to the fourth query step.

From the perspective of the user, the Selection operator selects actual particles detected during these events to be added to the decay that is going to be the input of the Result step. The operator filters them according to predicates that refer to special particles' attributes, like having $'mass > 0.4'$. In this step, the origin of the object flow starts at the Selection operators that are leaves of the tree.

The Transformation and Aggregation operators work only on the results of Selection operators. Again, from the user's perspective, Transformation combines the results of two (or more) selections according to user-defined filter predicates. Usually, this results in the construction of a particle higher up in the decay chain (added to the *decay* or *decays* of the particular *event*). So the transformation operator creates new particle objects with the data from previous selections. These new objects are represented with the Transformation Result operator, which we use a symbol similar to the Selection operator, because both of them describe the objects to be added to the decay. From the computational point of view, this corresponds to a join of the input object streams, followed by an aggregation that generates a new object element in the decay through some special user-defined functions called vertexing (that compute the attributes for the new particles).

An aggregation sums up information on particles per event, i.e. we get one result for each event. It is a grouping of the decays by event and a subsequent aggregation (using a user-defined aggregate function like $D^+.max(mass)$).

Now we need a way to connect the objects. For this, we use a simple line with an arrow that describes the data flow from one operator to another.

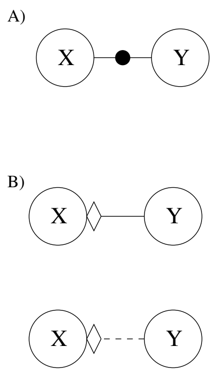


Figure 8.7: A) Comparison B) Minimal distance

Our language supports two more primitives to relate the result of Selection operators: the Comparison and the Minimal Distance operators (see Fig.8.7). Both of them relate the two different input streams and apply a selection predicate.

The first one, the comparison operator, compares a particular attribute value of some object from each decay (X) to those of the decay (Y) within the same Event. In doing so, it filters out particles that do not satisfy the condition of the comparison operator. It represents an algebraic join under a condition predicate.

The second case is the Minimal Distance operator. In contrast to the comparison operator, the minimum distance operator is directed. It operates in two modes: mandatory (computationally a join) and non-mandatory (left-outer join), which are symbolized by a solid and a broken line, respectively. In both cases, the result is a pair of particles (X,Y). The user can define a distance threshold for all particles in X that are further away from Y to be filtered out. In this threshold, the user defines the limits within which the result of the distance function is valid, and the result is not filtered out. The first mode (mandatory) means that all particles in A are matched to the nearest particle in B, and the pairs of particles are returned. All particles in A which do not find a matching partner in B are filtered out. The second mode (non-mandatory) is the same as the first except that particles from A not finding a partner in B are retained, i.e. these particles are paired with a empty value.

Finally, our running example of Fig.8.1 summarizes the description of our language operators for this step. We begin on the right-hand side with extracting all Π^+ and Π^- particles from the events of the third run. With the help of a transformation operator (T_1), we reconstruct $\overline{K}^0$ particles. Another transformation operator (T_2) helps us to find D^+ particles. One condition operator was inserted which contains the condition expression that guarantees that Π^+ and Π^- have the same mass. A minimal distance operator is used to select the PV (primary vertex in physics jargon), that is closer to the computed D^+ particle. If none exists, the decay chain is discarded. Finally, an aggregation operation filters out the particles with the maximal energy level for each event.

For the analysis, it might be interesting to get objects that are referenced by some particles, or vertexes, selected in the decay (e.g. $Particle \rightarrow MCParticle$). It is even possible that the selection of a given particle, or vertex, is conditioned to the existence of the object it is referring to.

We will use: ○•–○, to mean that the particle, or vertex, is selected and also the referenced object if this last one exists. The other possibility is to use ○•. . .○, to mean that the particle, or vertex, will be selected if and only if the corresponding referenced object exists.

The different operators' signature can be consulted in Fig.8.8.

Selection $\bigcirc^{head/path}_{pred}$	$\{Event\} \rightarrow \{Decay\}$
Transformation $⬯^{head}_{pred}$	$\{Decay\} \times \cdots \times \{Decay\} \rightarrow \{Decay\}$
Transformation Result $●^{head}_{pred}$	$\{Decay\} \rightarrow \{Decay\}$
Aggregation $\square^{func}_{pred}$	$\{Decay\} \rightarrow \{Decay\}$ func is the aggregator function
Comparison $\bullet_{pred}$	$\{Decay\} \times \{Decay\} \rightarrow \{Decay\}$
Minimal distance $\diamond^{func}_{pred}$	$\{Decay\} \times \{Decay\} \rightarrow \{Decay\}$ func is the minimal distance function

Figure 8.8: Signature of PHEASANT Operators in the decay description step.

Selecting the Result

$$\boxed{\text{1D}} \quad \boxed{\text{2D}} \quad \boxed{\text{3D}} \quad \boxed{\#} \quad \perp$$

Figure 8.9: Specification of the result set:1D, 2D, 3D, Value result and operator omission

Last but not least, we have to describe how to visualize the result of the query as the fourth step. We provide four different operators for the description of the result (see Fig.8.9 for the notation, and Fig.8.10 for the corresponding signatures): three operators to create one-, two-, and three-dimensional histograms, and one operator to output numeric values. These operators will basically apply a reduction on a certain user-specified list of attributes over the decays that resulted from the previous step.

In case of the histograms they simply represent a resulting set of tuples to which the framework should visually present its result in the shape of a histogram. A grouping criteria, also user-defined, can be used.

In the case of the numeric value operator, a user-defined aggregation function is specified to get a single result value. In case of absence of a result operator(in this case, we will represent it textually by $\perp$), the result can be used to feed some other analysis frameworks, external to our own one. In our running example, a 1D histogram is requested as output from the query result with the list of attributes $\{D^+.mass\}$.

Result 1D $\boxed{\text{1D}}head$	$\{Decay\} \rightarrow Histogram$
Result 2D $\boxed{\text{2D}}head$	$\{Decay\} \rightarrow Histogram$
Result 3D $\boxed{\text{3D}}head$	$\{Decay\} \rightarrow Histogram$
Result Number $\boxed{\#}head$	$\{Decay\} \rightarrow Basic\_Type$
Omission $\perp$	$\{Decay\} \rightarrow \{Decay\}$

Figure 8.10: Signature of PHEASANT's Result Operators

8.2.4 Grammar

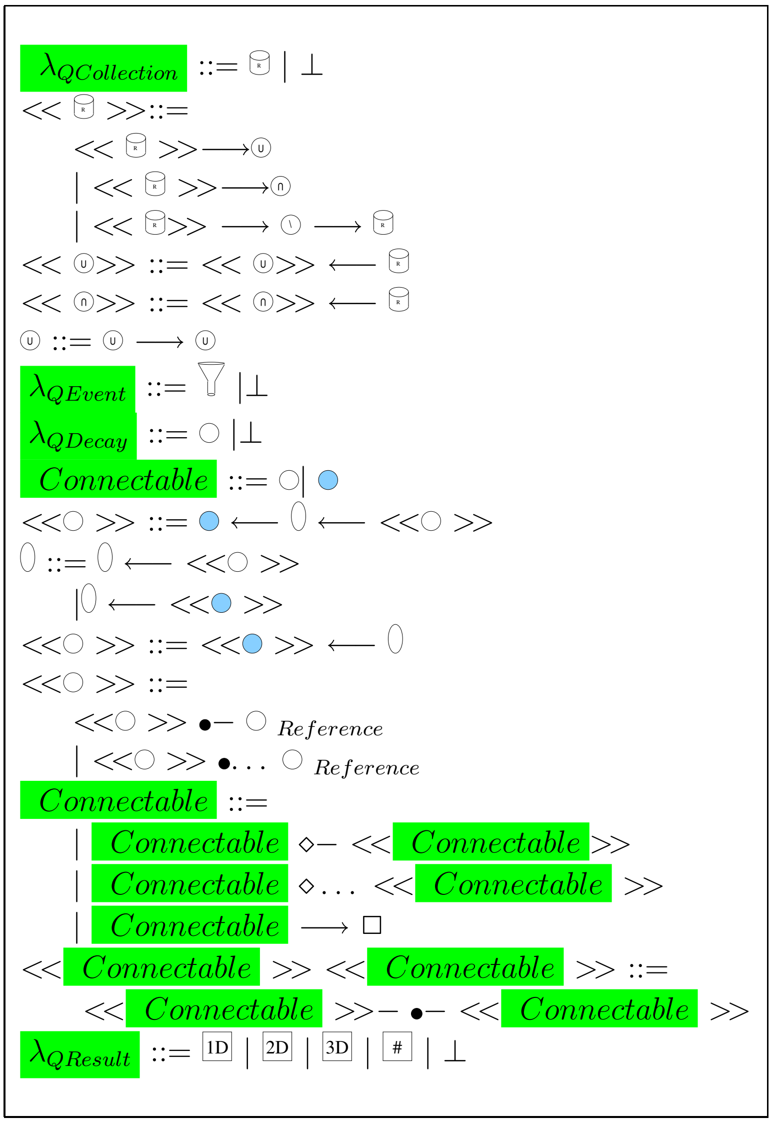
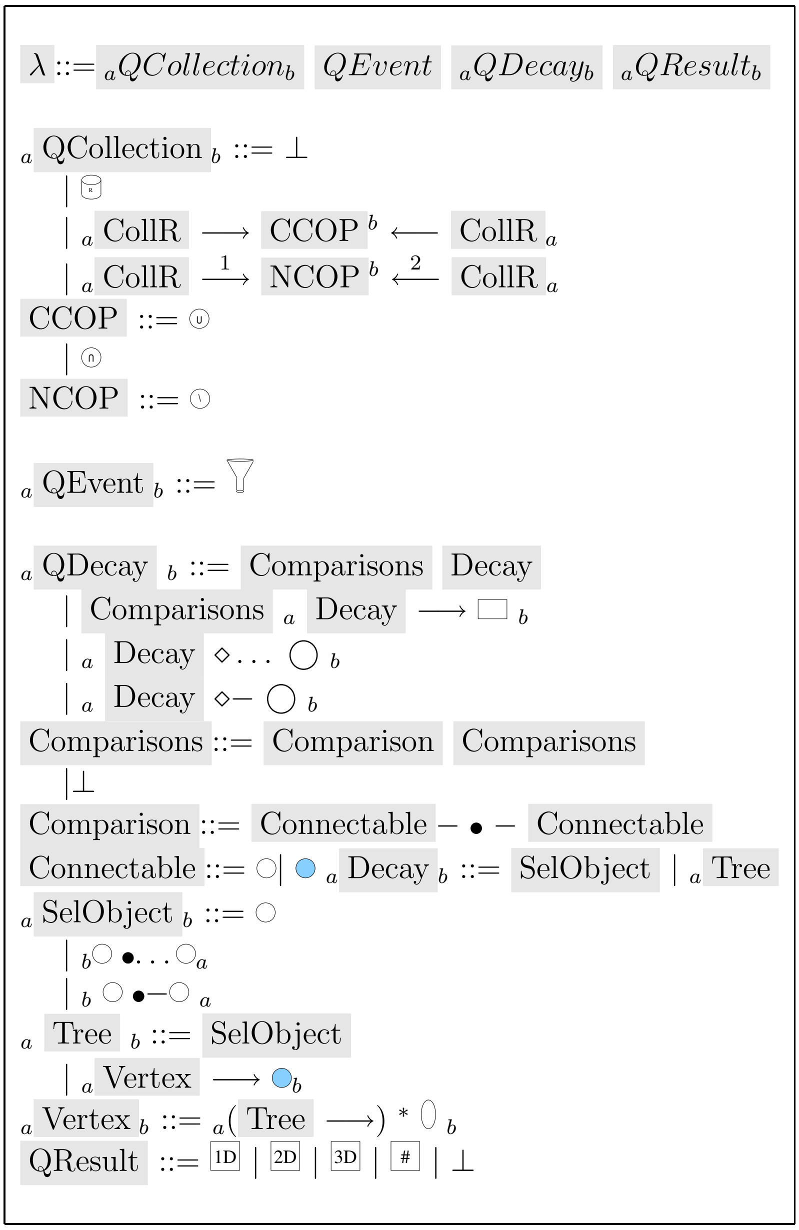


Figure 8.11: Context-sensitive graph grammar

In order to proceed with the definition of the syntax of our language, we have to describe how symbols may be formed into valid phrases of the language.

Comparing our diagrammatic language operators with graphs and edges, we make use of a graph grammar to define our visual query language (see Fig.8.11). This grammar is context-sensitive since it allows the usage of terminals and non-terminals in the left-hand side, leading to left and right graphs of a production to have an arbitrary number of nodes and edges. The left-hand side represents part of the graph structure that is going to be extended in the right-hand side.

This grammar notation, however being useful for the implementation of the graphical parser, is not convenient for semantic description purposes. The graph structure leads to complex algorithms to interpret them. This way, we must describe the syntax notation making use of a BNF like grammar, which is represented inherently by a tree structure, the Abstract Syntax Tree (AST). This means that we have to describe the syntax at a higher abstraction level. In practice, this implies to deal with the concept of comparison operators that are the elements that close the DAGs. We break the structure by decoupling them from the DAG, (which becomes a tree). These comparisons (predicates) are going to be interpreted later by the semantics mechanism.



λ ::= aQCollectionb QEvent aQDecayb aQResultb
a QCollection b ::= ⊥
a CollR ⟶ CCOP b ⟵ CollR a
a CollR 1⟶ NCOP b 2⟵ CollR a
CCOP ::=
NCOP ::=
a QEvent b ::=
a QDecay b ::= Comparisons Decay
| Comparisons a Decay ⟶ b
| a Decay ◇. . . b
| a Decay ◇− b
Comparisons ::= Comparison Comparisons
|⊥
Comparison ::= Connectable − ● − Connectable
Connectable ::= ○| a Decay b ::= SelObject | a Tree
a SelObject b ::= ○
| b○ ●. . .○a
| b ○ ●−○ a
a Tree b ::= SelObject
| a Vertex ⟶ b
a Vertex b ::= a(Tree ⟶) * b
QResult ::= 1D | 2D | 3D | # | ⊥

Figure 8.12: PHEASANT’s BNF-like grammar

PHEASANT QL's grammar consists of four parts $< \Sigma, N, P, S >$ where:

- Σ is the finite set of terminal symbols, the alphabet of the language, that are assembled to make up the sentences of the language. We decided to use the symbols of the language itself as terminals in the grammar, so there is no problem to recognize the components introduced in the last section.

- N is a finite set of nonterminal symbols or syntactic categories, each of which represents some collection of subphrases of the sentences. In our description, non-terminals have a grayish background, while for the terminals the regular background is used.

- P are the production rules stated in Fig.8.12. They are represented as $LHS ::= RHS$ where productions with the same LHS (left hand side) separate the different RHSs (right-hand sides) by |. Both left and right sides are defined in terms of terminal symbols and nonterminals.

- S is the start symbol λ or null graph.

Let us give some extra explanatory notes. In our production rules, we define a and b as connection points to the rest of the graph, and they are used to keep the graph orientation after applying the rule (which means that the data flow goes from `a` to `b`). Whenever the orientation is obvious, we will not use these characters for readability purposes.

Associated with each operator is some additional data, like attribute lists and condition lists. During query construction, when using the user interface, this information is hidden most of the time. Therefore, we describe this hidden data associated with each operator with the symbol $::\propto$ (see Fig.8.13).

Furthermore, we distinguish between two different collection types: **run** collections and **event** collections. When no collection operators are given in a query, it considers all available data. If a run collection operator is given without an event, only data from the **runs** that match the selected filter conditions specified in that operator will be considered. We can further restrict this by additionally supplying a description of an event collection operator. Then only a subset of the **events** of the chosen

runs selected by the run collection operator will be taken into account. When specifying an event collection operator without specifying any run collection operator, we regard the relevant **events** from all **runs**.

When connecting collection operators via set operators, the grammar differentiates between commutative operators, CCOP ($\cup$, $\cap$), and non-commutative operators, NCOP ($\setminus$).

The language has been designed considering the need of the user to extend the expressions, conditions and transformation functions with his own ones (otherwise, it would be very restrictive). We will make use of the terms UDF[68], which stands for set of user-defined functions, the corresponding subsets are: UDSFs (user defined scalar functions with the signature: $Float \times \cdots \times Float \rightarrow Float$); UDAFs (user-defined aggregate functions with the signature: $\{Float\} \rightarrow Float$), and UDTFs (user defined transform functions $Decay \times \cdots \times Decay \rightarrow Decay$). Some expressions and conditions are composed using UDSFs. Users can integrate their own aggregation functions into the system (it currently provides a max- and min-function UDAF) into an aggregation operator. To connect selection objects via a transformation operator, the user can also supply his or her own transformation function (usually a function to reconstruct vertices UDTF).

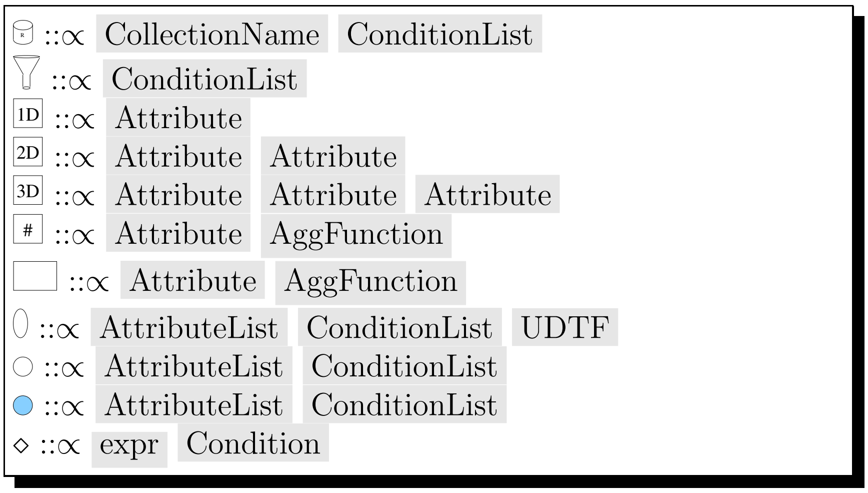

::∝ CollectionName ConditionList
::∝ ConditionList
1D ::∝ Attribute
2D ::∝ Attribute Attribute
3D ::∝ Attribute Attribute Attribute
# ::∝ Attribute AggFunction
::∝ Attribute AggFunction
::∝ AttributeList ConditionList UDTF
::∝ AttributeList ConditionList
::∝ AttributeList ConditionList
◇ ::∝ expr Condition

Figure 8.13: Terminal definitions

AttributeList ::= Attribute *
Attribute ::= (Label, Type)
ConditionList ::= Condition *
Condition ::= expr CompOP expr
expr ::= expr ArithOp expr | Ar | UDSF (ArList)
ArList ::= Ar *
Ar ::= Constant | Attribute
Constant ::= IntegerConstant | RealConstant | StringConstant
CompOP ::= > | < | >= | <= | = | <>
UDTF ::= StringConstant
UDSF ::= StringConstant
UDAF ::= StringConstant
ArithOP ::= $+ | - | * | \backslash$
AggFunction ::= UDAF | Max | Min
IntegerConstant ::= [sign][digit]$^+$
RealConstant ::= [sign][digit]*.[digit]*
StringConstant ::= ValueReference | MemberReference
ValueReference ::= letter [letter | digit]*
MemberReference ::= letter [letter | digit]*.[letter | digit]*
Digit ::= 0|1|2|3|4|5|6|7|8|9
sign ::= $+|-$
letter ::= Lowercase | Uppercase
Lowercase ::= $a|b|c|...|z$
Uppercase ::= $A|B|C|...|Z$

Figure 8.14: Grammar of the textual elements of PHEASANT QL

8.3 Semantics

The next step after defining the abstract syntax is the definition of the formal semantics. The normal approaches are:

- Translational Semantics - The semantics is given by defining a mapping to models of a simpler language, which is better understood.

- Operational Semantics - Expresses the semantics of a modelling technique by giving a mechanism that allows to determine the effect of any model specified in the technique. An operational semantics for a particular programming language describes how any particular valid program in the language is interpreted as sequences of computational steps. These sequences then are the meaning of the program.
- Denotational Semantics - The syntactic constructs of a language are mapped onto constructs in another language with a well-defined meaning. The target is a mathematical domain and not another modelling technique.
- Axiomatic Semantics - Treats a model like a logical theory, does not center on what the model means, but on what can be proven about it.

In our case, we want to define our language by means of algebraic operators that are very well understood and deeply studied in the field of database research. This way, we want to take advantage of the accumulated knowledge in this area. In consequence, we find it adequate to make use of the translational semantics, by making a syntax-to-syntax mapping of the language into the algebraic operators of the target object.

8.3.1 The Target Language - Intermediate Algebra Operators

We have designed our language making use of a syntax mapping to the algebra where the semantics is described here.

Type System

Before we start with the explanation of the algebraic operators, we introduce the definitions and precise notations which are useful for a clear and unambiguous interpretation of these operators.

Definition 1 (Basic types).
The primitive types are:

- $\mathcal{F}loat$ (floating point number)
- $\mathcal{B}ool$ (value "true" or "false")
- $\mathcal{I}nteger$
- $\mathcal{S}tring$ (sequence of characters)

Definition 2 (Type constructor).
For the bulk type *set*, (unordered collection of elements of type τ), we write: $\{\tau\}$

Definition 3 (Type variable).
We define the notation for a type variable to be: $\tau_1, ..., \tau_n$

Definition 4 (Tuple type constructor).
A tuple is a mapping from a set of attributes to values of a certain type. We can define tuple types as $[a_1 : \tau_1, ..., a_n : \tau_n]$ where for $1 \leq i \leq n$:

- τ_i are types
- a_i are attribute names
- $a_i \neq a_j$

The set of attributes defined for a tuple t is written as $\mathcal{A}(\tau)$. All the tuples of type τ have the same attributes $\mathcal{A}(\tau)$.

Nested tuples are possible. A value of an attribute may be a set of tuples.

In order to represent a tuple of type τ' that contains the same attributes as $\tau = [a_1 : \tau_1, ...a_n : \tau_n]$ except for the attribute $a_j, 1 \leq j \leq n$ we use τ / a_j.

The concatenation of tuples and functions is denoted by $\circ$.

Definition 5 (Relation types).

A relation is a set of tuples which are all of the same type $[a_1 : \tau_1, ..., a_n : \tau_n]$, and we represent the type of the relation by $\{[a_1 : \tau_1, ..., a_n : \tau_n]\}$.

Definition 6 (Structural sub-typing).

Sub-typing is the notion of inclusion between types. It is represented by $A \leq B$, A is a subtype of B.

$\tau \leq \tau' \Rightarrow \{\tau\} \leq \{\tau'\}$, meaning that if τ is a subtype of τ' then the set type $\{\tau\}$ is subtype of $\{\tau\}'$. Further: $[a_1 : \tau_1, ..., a_n : \tau_n] \leq [a_1 : \tau'_1, ..., a_k : \tau'_k]$ if for all $0 \leq k \leq n$:$\tau_1 \leq \tau'_1$

Definition 6 (Free variables).

$\mathcal{F}(e)$ is defined as the set of free variables of an expression e.

Definition 7 (Predicates).

For an expression *pred* possibly containing free variables, and a tuple t, we denote by $pred(\tau)$ the result of evaluating *pred* where bindings of free variables are taken from attribute bindings provided by τ. $\mathcal{F}(pred) \subseteq \mathcal{A}(\tau)$.

Definition 8 (Elements of a tuple).
If b is a tuple of type $[a_1 : \tau_1 ... a_n : \tau_n]$ then the type of the attribute $b.a_i$ is τ_i, with $0 < i \leq n$.

Definition 9 (Mapping function).

A function mapping a tuple to a new tuple, possibly of a different type, is also denoted by the symbols *head*.

Definition 9 (Unique attribute names generator).
$\zeta : \rightarrow String$ is a function that generates a unique string, different from all others generated before. These strings are used as labels in some of the algebraic operators defined in the following section.

Definition 10 (Type histogram).

We define the type histogram to be:

- $\tau_H 1 = \{[r_1 : \mathcal{F}loat]\}$
- $\tau_H 2 = \{[r_1 : \mathcal{F}loat, r_2 : \mathcal{F}loat]\}$

- $\tau_H 3 = \{[r_1 : \mathcal{F}loat, r_2 : \mathcal{F}loat, r_3 : \mathcal{F}loat]\}$

Sometimes we make use of the notation $< a_1, ..., a_n >$, which means $\mathcal{A}([a_1 : \tau_1, ..., a_n : \tau_n])$.

Schema

For the examples used to explain our operators, we are going to make use of the following schema:

- **expcol** =

  $\{[id : Integer,$
  $event : \{\textbf{Event}\},$
  $eventsType : Integer,$
  $responsible : string]\}$

- **runcol** =

  $\{[id : Integer,$
  $event : \{\textbf{Event}\},$
  $start : Integer,$
  $end : Integer,$
  $sequence : Integer]\}$

- **myprivatecol** =

  $\{[id : Integer,$
  $event : \{\textbf{Event}\},$
  $Date : String,$
  $queryNumber : Integer]\}$

- **Event** =

  $[id : Integer,$
  $bx : integer,$
  $particle : \{\textbf{Particle}\},$
  $vertex : \{\textbf{Vertex}\}]$

- **Particle** : $[id : Integer,$
  $mass : Float,$
  $Px : Float,$
  $Py : Float,$
  $Pz : Float,$
  $Energy : Float,$
  $MCParticle$: {**MCParticle**},
  ...]
- **Vertex** :
  $[id : Integer,$
  $x : FLoat,$
  $y : Float,$
  $z : Float,$
  $MCVertex$: {**MCVertex**},
  $outgoingParticle$: **Particle**,
  $ingoingParticle$: {**Particle**}]
- **MCParticle** : $[id : Integer,$
  $mass : Float,$
  $Px : Float,$
  $Py : Float,$
  $Pz : Float,$
  $Energy : Float,$
  ...]
- **MCVertex** :
  $[id : Integer,$
  $x : FLoat,$
  $y : Float,$
  $z : Float,$
  ...]

As stated before in chapter 3, some of the entity attributes might change slightly from experiment to experiment.

Algebraic Operators

Having the type system defined in the previous section, we are now ready to define our algebraic operators. In Fig.8.15, we give the type signature for each operator. The semantic of the operators is summarized in Fig.8.16. Informally, we can define our operators as follows:

- **selection** $\sigma_{pred}(X)$ - Selects all elements of X that satisfy the predicate pred.

X	id	mass	energy
	1	1.5	4
	2	1.8	5

$[\sigma_{mass>1.5}(X)]$	id	mass	energy
	2	1.8	5

- **join** $X \bowtie_{pred} Y$ - Joins the collection X and Y using the join predicate pred.

X	id	mass	energy
	1	1.5	4
	2	1.8	5
	3	1.0	6

Y	id	mass	energy
	5	1.3	4
	6	1.4	5

$[X \bowtie_{pred} Y]$	$Tuple_1$			$Tuple_2$		
	id	mass	energy	id	mass	energy
	1	1.5	4	5	1.3	4
	1	1.5	4	6	1.4	5
	2	1.8	5	5	1.3	4
	2	1.8	5	6	1.4	5

Where $pred = tuple_1.mass > 1.0$ *and* $tuple_2.mass > 1.0$.

- **unnest** $\mu_{pred}^{path}(X)$ - returns the collection of all pairs (x,y) for each $x \in X$ and for each $y \in x.path$ that satisfy the predicate pred(x,y)

X	Event				
	id	Particle			
		x	y	z	...
	1	1	1	1	...
		2	2	2	...
		3	3	3	...
	2	2	2	2	...
		1	1	1	...

$[\mu_{true}^{``Particle1:Event.Particle''}(X)]$	Event		Particle1			
	id	...	x	y	z	...
	1	...	1	1	1	...
	1	...	2	2	2	...
	1	...	3	3	3	...
	2	...	2	2	2	...
	2	...	1	1	1	...

- **reduce** $\mathbf{\Delta}_{pred}^{\oplus/head}(X)$ - generalizes the relational projection operator, collects the values head(x) for all $x \in X$ that satisfy pred(x) using the accumulator $\oplus$, which can be a set, $(\bigcup)$, or an aggregate function like $\{max, min, sum, avg\}$.

X	id	mass	energy
	1	1.5	4
	2	1.8	5

$[\Delta^{\cup/<id>}_{true}(X)]$	id
	1
	2

- **outer-join** $X ⟕_{pred} Y$ - is the left outer join between X and Y using the join predicate pred. If the range variable y of Y is empty or there are no elements that can be joined with the range variable x of X, then y becomes a null and the result is the pair $< x, null >$.

X	id	mass	energy
	1	1.5	4
	2	1.8	5
	3	1.0	6

Y	id	mass	energy
	5	1.3	4
	6	1.4	5

$[X ⟕_{pred} Y]$	$Tuple_1$			$Tuple_2$		
	id	mass	energy	id	mass	energy
	1	1.5	4	5	1.3	4
	2	1.8	6	5	1.4	5
	3	1.8	5			

Where $pred = “tuple_1.energy = tuple_2.energy''$.

- **outer-unnest** $=\!\!\mu^{path}_{pred}$ - Similar to the unnest, but if x.path is empty for $x \in X$ or pred(x,y) is false for all $y \in x.path$, then the pair (x, NULL) is given as output.

X	Event				
	id	Particle			
		x	y	z	...
	1	1	1	1	...
		2	2	2	...
		3	3	3	...
	2	2	2	2	...
		1	1	1	...

$[=\mu^{“Particle1:Event.Particle''}_{“Event.Particle.x>2''}(X)]$	Event		Particle1			
	id	...	x	y	z	...
	1	...	3	3	3	...
	2	...				

- **nest** $\mathbf{\Gamma}^{\oplus/head/group}_{pred}(X)$ - Images of elements x and y of a given collection X, (head(x) and head(y)), are grouped together in the same group if their evaluation value of the group-by-function group is equal, ($group(x) = group(y)$). After grouping, the accumulator $\oplus$, where either $\oplus \in \{Max, Min\}$ or $\oplus \in \{max, min, sum, count, ...\}$, will reduce each group. The next section will describe thoroughly these aggregator functions.

  The result of evaluating the accumulator function can be divided into two groups:

  For $\oplus \in \{max, min, sum, count, ...\}$, in order to feed directly the result operators. An example could be:

X	Event		Particle1				Particle2			
	id	...	x	y	z	...	x	y	z	...
	1	...	0	0	0	...	1	1	1	...
	1	...	1	1	1	...	2	2	2	...
	1	...	0	0	0	...	3	3	3	...
	2	...	0	0	0	...	2	2	2	...
	2	...	0	0	0	...	1	1	1	...

$[\mathbf{\Gamma}^{max/[value:''Particle1.x+Particle2.x'']>/Event.id}_{true}(X)]$	max
	3
	2

  For $\oplus \in \{Max, Min\}$. An example could be:

$[\mathbf{\Gamma}^{Max/<vale,tuple>/Event.id}_{true}(X)]$	Event		Particle1				Particle2			
	id	...	x	y	z	...	x	y	z	...
	1	...	1	1	1	...	2	2	2	...
	2	...	0	0	0	...	2	2	2	...

Where $< value, tuple >= [value :'' Particle1.x+Particle2.x'', tuple : X]$.

- **union**$= \cup(X, Y)$ - Returns all tuples that occur in either X and Y, if $\mathcal{A}(\tau_X) = \mathcal{A}(\tau_Y)$, with τ_X and τ_Y being type variables of respectively X and Y.
- **intersection** $= \cap(X, Y)$ - Returns all tuples that occur both in X and in Y, if $\mathcal{A}(\tau_X) = \mathcal{A}(\tau_Y)$, with τ_X and τ_Y being type variables of respectively X and Y.
- **difference**$= \backslash(X, Y)$ - Returns all tuples that occur in X but do not occur in Y, if $\mathcal{A}(\tau_X) = \mathcal{A}(\tau_Y)$, with τ_X and τ_Y being type variables of respectively X and Y.

Selection σ_{pred}	$\{\tau\} \rightarrow \{\tau\}$ $pred : \tau \rightarrow \mathcal{B}ool, \mathcal{F}(pred) \leq \mathcal{A}(\tau)$ $\tau \leq []$
Join $\bowtie_{pred}$	$\{\tau_1\} \times \{\tau_2\} \rightarrow \{[tuple_1 : \tau_1, tuple_2 : \tau_2]\}$ $pred : \tau_1, \tau_2 \rightarrow \mathcal{B}ool, \mathcal{F}(pred) \leq \mathcal{A}(\tau_1) \cup \mathcal{A}(\tau_2)$ $\tau_i \leq []$
Unnesting $\mu^{name:path}_{pred}$	$\{\tau\} \rightarrow \{\tau'\}$ $pred : \tau, \tau' \rightarrow \{\mathcal{B}ool\}$ $if\ \tau = [a_1 : \tau_1, ..., a_n : \tau_n, path : \tau_0], 0 < n, \tau_0 \leq$ $\tau' = [a_1 : \tau_1, ..., a_n : \tau_n] \circ [name : \tau_0]$ $name = \zeta()$

Reduce
$\mathbf{\Delta}^{\oplus/head}_{pred}$
	if $\oplus = \cup$: $\{\tau_1\} \rightarrow \{\tau_2\}$
	if $\oplus = max, min, sum, ...$: $\{\tau_1\} \rightarrow \tau_2$
	$head : \tau_1 \rightarrow \tau_2$
	$pred : \tau_1 \rightarrow \mathcal{B}ool, \mathcal{F}(pred) \leq \mathcal{A}(\tau_1) \cup \mathcal{A}(\tau_2)$

Outer-Join $⟕_{pred}$	$\{\tau_1\} \times \{\tau_2\} \rightarrow \{[tuple_1 : \tau_1, tuple_2 : \tau_2]\}$ $pred : \tau_1, \tau_2 \rightarrow \mathcal{B}ool, \mathcal{F}(pred) \leq \mathcal{A}(\tau_1) \cup \mathcal{A}(\tau_2)$ $\tau_i \leq []$
Outer-Unnest $=\!\mu^{name:path}_{pred}$	$\{\tau\} \rightarrow \{\tau'\}$ $pred : \tau, \tau' \rightarrow \mathcal{B}ool$ $if\ \tau = [a_1 : \tau_1, ..., a_n : \tau_n, path : \tau_0], 0 < n, \neg(\tau_0 \leq [])$ $\tau' = [a_1 : \tau_1, ..., a_n : \tau_n] \circ [name : \tau_0]$ $name = \zeta()$
nest $\mathbf{\Gamma}^{\oplus/head/group}_{pred}$	if $\oplus \in \{max, min, sum, avg...\}$ $\{\tau\} \rightarrow \{\mathcal{F}loat\}$ $head = \lambda\tau.[value : \mathcal{F}loat]$ if $\oplus \in \{Max, Min\}$ $\{\tau\} \rightarrow \{\tau\}$ $head = \lambda\tau.[value : \mathcal{F}loat, tuple : \tau]$ $pred : \tau \rightarrow \mathcal{B}ool, \mathcal{F}(pred) \leq \mathcal{A}(\tau)$
Union $\bigcup$	$\{\tau\} \times \{\tau\} \rightarrow \{\tau\}$
Intersection $\bigcap$	$\{\tau\} \times \{\tau\} \rightarrow \{\tau\}$
Difference $\setminus$	$\{\tau\} \times \{\tau\} \rightarrow \{\tau\}$

Figure 8.15: Type signature of our algebraic operators

Selection σ	Selects qualifying tuples according to predicate *pred*: $\sigma_{pred}(e) := \{t \mid t \leftarrow e, pred(t)\}$
Join $\bowtie$	Connects all tuples in e_1 to all in e_2 and selects tuples according to *pred*: $e_1 \bowtie_{pred} e_2 := \{[tuple_1 : t_1, tuple_2 : t_2] \mid t_1 \leftarrow e_1, t_2 \leftarrow e_2, pred(t_1, t_2)\}$
Unnesting μ	Selects a tuple and its nested attribute defined in path, according to to predicate *pred*: $\mu^{name:path}_{pred}(e) := \{(t_1, t_2) \mid t_1 \leftarrow e_1, t_2 \leftarrow path(t_1), pred(t_1, t_2)\}$
Reduce $\boldsymbol{\Delta}$	Collects values defined in head(t), according to *pred*, in the aggregator $\oplus$: $\boldsymbol{\Delta}^{\oplus/head}_{pred}(e) := \oplus\{head(t) \mid t \leftarrow e, pred(t)\}$
Outer-Join ⟕	Same as Join, but returns the tuple $[tuple1_1 : t, tuple_2 : NULL]$ if e_2 is empty or there are no elements to join to $t \in e1$: e_1 ⟕$_{pred}$ $e_2 := \{[tuple1 : t_1, tuple_2 : t_2] \mid$ $t_1 \leftarrow e_1,$ $t_2 \leftarrow$ **if** $\wedge\{\ \neg pred(t_1, x)\ \mid x \leftarrow e_2\}$ **then** *null* **else** $\{x \mid x \leftarrow e_2, pred(t_1, x)\}\}$
Outer-Unnest $=\!\!\mu$	Same as Unnest, but returns the tuple $t \circ [name = NULL]$ if $t.path$ is empty: $=\!\!\mu^{path}_{pred}(e) := \{(t_1, t_2) \mid$ $t_1 \leftarrow e,$ $t_2 \leftarrow$ **if** $\wedge\{\neg pred(t_1, x) \mid x \leftarrow path(t_1)\}$ **then** *null* **else** $\{x \mid x \leftarrow path(e), pred(t_1, x)\}\}$
nest $\boldsymbol{\Gamma}$	$\boldsymbol{\Gamma}^{\oplus/head/group}_{pred}(X) := \{\oplus\{head(w)\} \mid w \leftarrow e, pred(w), v = group(w)\} \mid$ $v \leftarrow \pi_{group}(X)\}$ $\pi_{group}(X) = \{group(t) \mid t \leftarrow X\}$, with duplicate elements removed.
Union $\bigcup$	returns the set of tuples that occur in both sets: $\cup(e_1, e_2) := \{x \mid x \in e_1 \vee x \in e_2\}$
Intersection $\bigcap$	Returns the set of common tuples: $\cap(e_1, e_2) := \{x \mid x \in e_1 \wedge x \in e_2\}$
Difference $\setminus$	Returns the set of tuples that return just in the first set: $\setminus(e_1, e_2) := \{x \mid x \in e_1 \wedge \neg(x \in e_2)\}$

Figure 8.16: Operators of the target algebra

Aggregation Functions

Each of the functions in the set $aggfunc \in \{max, min, sum, count, ...\}$ has the signature: $aggfunc : \{\mathcal{F}loat\} \rightarrow \mathcal{F}loat$. The user is supposed to write more user-defined aggregate functions if necessary, using the same signature. The definition of the $aggfuncs$ can be specified as follows:

- $count(x) = +\{1|e \leftarrow x\}$
- $sum(x) = +\{e|e \leftarrow x\}$
- $max : \{[value : \mathcal{F}loat]\} \rightarrow \mathcal{F}loat$
- $max(e) := \{x|x \leftarrow e : \forall y \in e, mx(x, y) = x.value\}$

  where:

  $mx : \mathcal{F}loat \times \mathcal{F}loat \rightarrow \mathcal{F}loat$

  $$mx(a, b) = \begin{cases} a & \textbf{if } a >= b \\ b & \textbf{else} \end{cases}$$

- $min : \{[value : \mathcal{F}loat\}] \rightarrow \mathcal{F}loat$
- $min(e) := \{x|x \leftarrow e : \forall y \in e, mn(x.value, y.value) = x.value\}$

  where:

  $mn : \mathcal{F}loat \times \mathcal{F}loat \rightarrow \mathcal{F}loat$

  $$mn(a, b) = \begin{cases} a & \textbf{if } a <= b \\ b & \textbf{else} \end{cases}$$

An example of the usage of max is:

X	id	mass	energy
	1	1.5	4
	2	1.8	5

$\Delta_{true}^{max/<X.id>}(X)$	X.id
	2

Certain PHEASANT operators like the Aggregator and Minimal distance need to define a special set of aggregate functions. Given a set of tuples $[value : \mathcal{F}loat, tuple : \tau]$, we need two operators to return the *tuple* with maximum/minimum value. We define these functions as $agg \in \{Max, Min\}$. Max is specified as follows:

- $Mx : \tau \times \tau \rightarrow \tau$
- $\tau = [value : \mathcal{F}loat, tuple : \tau']$
- $Mx([value : f_1, tuple : t_1], [value : f_2, tuple : t_2]) :=$
  $\begin{cases} [value : f_1, tuple : t_1] & \textbf{if } f_1 >= f_2 \\ [value : f_2, tuple : t_2] & \textbf{else} \end{cases}$
- $Max : \{\tau_1\} \rightarrow \tau_2$
- $\tau_1 = [value : \mathcal{F}loat, tuple : \tau]$ *and* $\tau_2 = \tau$
- $Max(e) := \{t_1| < v_1, t_1 >\leftarrow e, \forall < v_2, t_2 >\in e : Mx(< v_1, t_1 >, < v_2, t_2 >) =< v_1, t_1 >\}$

For example:

X	Event		Particle1				Particle2			
	id	...	x	y	z	...	x	y	z	...
	1	...	0	0	0	...	1	1	1	...
	1	...	1	1	1	...	2	2	2	...
	1	...	0	0	0	...	3	3	3	...
	2	...	0	0	0	...	2	2	2	...
	2	...	0	0	0	...	1	1	1	...

$\Gamma_{true}^{\begin{cases} Max/ \\ <'' Particle1.x + Particle2.x'', < X >> / \\ X.Event.id \end{cases}}$ (X)	value	tuple									
		Event		Particle1				Particle2			
		id	...	x	y	z	...	x	y	z	...
	3	1	...	1	1	1	...	2	2	2	...
	2	2	...	0	0	0	...	2	2	2	...

The opposite operation Min returns the corresponding *tuple* value that pairs with the minimal *value* compared against the whole set. It is defined in a similar way as Max:

- $Mn : < \mathcal{F}loat \times \tau > \rightarrow < \mathcal{F}loat \times \tau >$
- $Mn(< f_1, t_1 >, < f_2, t_2 >) := \begin{cases} < f_1, t_1 > & \textbf{if } f_1 <= f_2 \\ < f_2, t_2 > & \textbf{else} \end{cases}$
- $Min : \{< \mathcal{F}loat \times tuple >\} \rightarrow tuple$
- $Min(e) := \{t| < v_1, t_1 > \leftarrow e, \forall < v_2, t_2 > \in e : Mn(< v_1, t_1 >, < v_2, t_2 >) = < v_1, t_1 >\}$

We can now exercise a formal denotation of these algebraic operators as it can be seen in Fig.8.16.

Operator Trees

Textual algebraic forms using the operators just described tend to be better understood if we represent them as operator trees. This concept is easy to grasp if we make use of the concept of a stream of tuples from the leaves to the root of the tree.

As a helpful visual feature, we represent on the right side of the root of the tree the $\mathcal{A}(\tau)$, surrounded by <>, of the tuples that are resulting from the data stream. On the right side of the Unnest operator and the Collection, (leaf), we represent the new unnested attributes.

In the first case, the variable represent the new unnested attribute extracted, containing its type. In the second case, the variable ranges over the collection, meaning that the variable is of the type of the collection instances.

A simple example can be visualized in Fig.8.17. At the leaf, we are generating a stream of tuples of type $[student : Student]$. The Student collection is being ranged by the variable *student*. The stream is accepted by the Unnest operator that will output the stream of tuples of type $[student : Student, sup : Supervisor]$, meaning that in a tuple we match each student with each of his supervisors. Finally, with the stream resulting from the Unnest operator, the Reduce operator accepts each tuple and evaluates the expression $[phd : stu.name, prof : sup.name]$, building the set of tuples with the structure $< phd, prof >$.

We will make use of this visual concept in the following section to help explaining the semantics of our language.

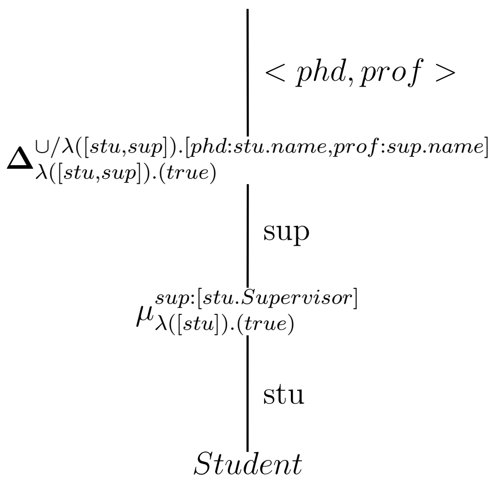

$< phd, prof >$
$\Delta^{\cup/\lambda([stu,sup]).[phd:stu.name,prof:sup.name]}_{\lambda([stu,sup]).(true)}$
sup
$\mu^{sup:[stu.Supervisor]}_{\lambda([stu]).(true)}$
stu
$Student$

Figure 8.17: Example of an algebraic form represented as a tree.

8.3.2 Language Description

As defined by the grammar, the user's query is described by four main components:

$$Query = \; Q_{Result} \; Q_{Decay} \; Q_{Event} \; Q_{Collection}$$

We are going to detail the mapping for each component and its operators as well as the necessary symbols for the formalization. We structure our explanation for each operator in the following way: first the conversion rules, then an informal explanation followed by an example and a depiction of the corresponding plan tree.

Mapping

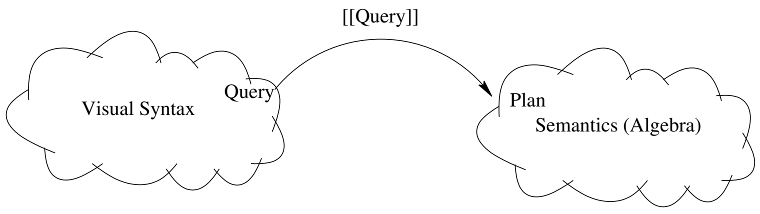


Figure 8.18: Map operator - Translates the visual query into our algebra.

We define the map operator $[\![Q]\!]$ as the translation of the query Q (a statement composed by abstract syntax notation operators) into the corresponding algebraic notation.

Definition 1 (PHEASANT QL mapping). A map operator $[\![q]\!]$ is a function that maps the query q specified in PHEASANT QL syntax into a corresponding expression of the intermediate Algebra.

Definition 2 (PHEASANT QL sub-mapping). The map operator is a composition of four sub-map operators.

- $[\![Q_{Collection}]\!]_C$, maps the PHEASANT collection visual query, $Q_{Collection}$, into the corresponding Algebra.

- $[\![Q_{Event}]\!]_E$, maps the PHEASANT Event filter visual query Q_{Event}. The resulting algebraic form depends on the result of $[\![Q_{Collection}]\!]_C$ evaluation.
- $[\![Q_{Decay}]\!]_D$, maps the PHEASANT Decay visual query Q_{Decay}. The resulting algebraic form is dependent on the result of $[\![Q_{Event}]\!]_E$ evaluation.
- $[\![Q_{Result}]\!]_R$, maps the PHEASANT Result query Q_{Result}. The resulting algebraic form is dependent on the result of $[\![Q_{Decay}]\!]_D$ evaluation.

A query in PHEASANT QL can be interpreted (as given by the grammar) as four subqueries that correspond to the four major query steps: Result, Decay, Event and Collection. This means that the first step of the mapping operation will be described by the following rule:

$$\begin{aligned} &[\![Query]\!] = [\![Q_{Result}\ Q_{Decay}\ Q_{Event}\ Q_{Collection}]\!] = \\ &\qquad = [\![Q_{Result}]\!]_R\ ([\![Q_{Decay}]\!]_D([\![Q_{Event}]\!]_E([\![Q_{Collection}]\!]_C))) \quad (Q1) \end{aligned}$$

Figure 8.19: Translation rules from the AST to query Plan - Collection-Event materialization

The sub-query $[\![Q_{Collection}]\!]_C$ is a sub-plan of $[\![Q_{Event}]\!]_E$ which in turn is a sub-query of $[\![Q_{Decay}]\!]_D$. Finally, $[\![Q_{Decay}]\!]_D$ is a sub-plan of $[\![Q_{Result}]\!]_R$.

Symbol	*Definition*
$\langle\langle\tau\rangle\rangle^w$	Notation used when needed, to make explicit the attributes of the tuples $w = \mathcal{A}(\tau)$
$[H \mid T]$	list with a head H and tail T
$[H \mid \emptyset]$	list with one element
$\emptyset$	empty list
$\perp$	empty or absence of operator
lattrib	list of attributes, $[a_1 : \tau_1, ..., a_n : \tau_n]$
name	label for the attribute, $name = \zeta()$
collect	Collection name
$fdist()$	arithmetic distance calculation function
$expr(\tau)$	arithmetic expression where the free variables $\in \mathcal{A}(\tau)$ we may use the notation expr(fdist) to mean that the expression uses an fdist function
$udtf$	function to generate a tuple according to a pre-defined type structure (useful) for the definition of the Transformer operator

Figure 8.20: Used symbols

The Collection Operators

$$[\![\perp]\!]_C = Event \tag{C_0}$$

$$[\![\boxed{\scriptstyle R}^{collect}_{pred}]\!]_C = \langle\!\langle \mathbf{\Delta}^{\cup/\lambda(<c,evt>).<evt>}_{\lambda(<c,evt>).(true)}(\mu^{evt:c.event}_{pred}(collect))\rangle\!\rangle^{[evt]} \tag{C_1}$$

$$[\![X \cup Y]\!]_C = \cup([\![X]\!]_C, [\![Y]\!]_C) \tag{C_2}$$

$$[\![X \cap Y]\!]_C = \cap([\![X]\!]_C, [\![Y]\!]_C) \tag{C_3}$$

$$[\![X \setminus Y]\!]_C = \setminus([\![X]\!]_C, [\![Y]\!]_C) \tag{C_4}$$

Figure 8.21: Translation rules from the AST to query Plan Collection-Event materialization

The first rule C_0 says that in the case of any collection operator in the description of the query, the considered collection source of events will be the Event collection.

As it is explicit in the C_1, the collection selection symbol can be expressed by intermediate algebraic operators. The stream of tuples c $\{[c : collect]\}$, where collect is the name of the collection of requested tuples, and existing in the Collection Catalog. The unnest operator, in turn, accepts the stream of tuples and constructs a stream $\{[c : collect, evt : Event]\}$, connecting each collect with one of its events. The reduce operator will evaluate the expression head $\lambda(< c, evt >).< evt >$ for every input element and constructs a set.

In order to make the manipulation of these collections more flexible by means of set union(C_2), intersection (C_3) and difference (C_4) we have set mapping rules for these operators. Since the operators are the same and the mapping is direct, we will refrain ourselves from explaining them further.

We will now present an example that combines some of these operators and their corresponding mapping into to the algebra.

A short example of a possible query mapping could be:

Example 1:

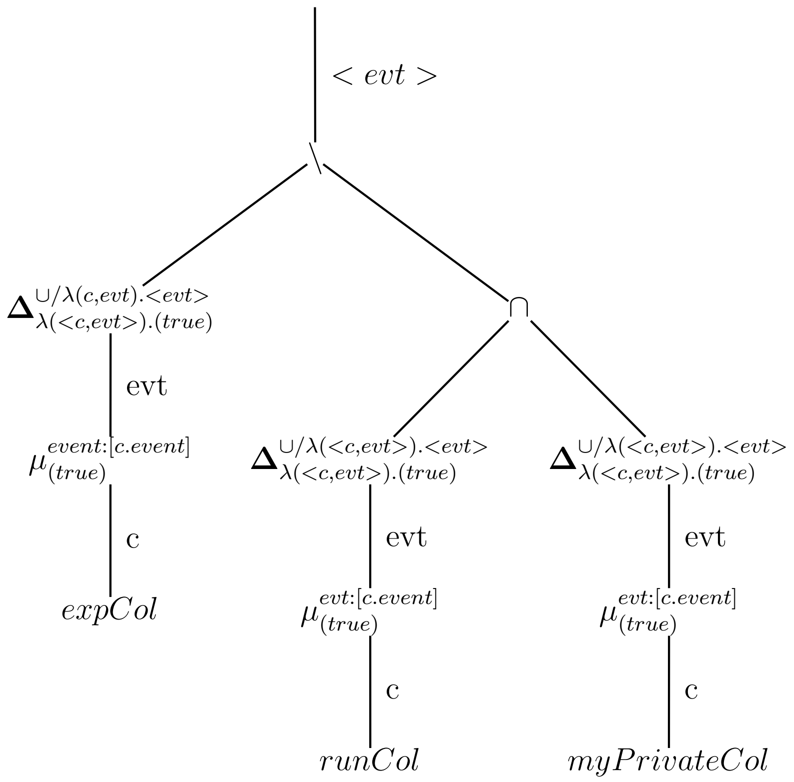
$$[\![\text{⛁}^{expCol}_{true} \setminus (\ \text{⛁}^{runCol}_{true} \cap \text{⛁}^{myPrivateCol}_{true}\ )]\!]_C =$$

$$(C_4) \qquad = \setminus([\![\text{⛁}^{expCol}_{true}]\!]_C,\ ([\![\text{⛁}^{runCol}_{true} \cap \text{⛁}^{myPrivateCol}_{true}]\!]_C))$$

$$(C_2) \qquad = \setminus([\![\text{⛁}^{expCol}_{true}]\!]_C, (\cap(\ [\![\text{⛁}^{runCol}_{true}]\!]_C,\ [\![\text{⛁}^{myPrivateCol}_{true}]\!]_C))$$

$$\begin{aligned}(C_1) \qquad = \setminus(&\langle\!\langle \mathbf{\Delta}^{\cup/\lambda(<c,evt>).<evt>}_{\lambda(<c,evt>).(true)}(\mu^{evt:[c.event]}_{(true)}(expCol))\rangle\!\rangle^{[evt]},\\ &(\cap(\ \langle\!\langle \mathbf{\Delta}^{\cup/\lambda(<c,evt>).<evt>}_{\lambda(<c,evt>).(true)}(\mu^{evt:[c.event]})_{(true)}(runCol))\rangle\!\rangle^{[evt]},\\ &\quad \langle\!\langle \mathbf{\Delta}^{\cup/\lambda(<c,evt>).<evt>}_{\lambda(<c,evt>).(true)}(\mu^{evt:[c.event]})_{(true)}(\\ &\qquad myPrivateCol))\rangle\!\rangle^{[evt]}\end{aligned}$$

We assume that expCol are collections with some special purposes defined by the system experts, runCol are collections of events organized in runs and finally myPrivate are the user's personal collections of selected events (likely during previous analysis phases). In this example, we want to select the collection of events contained by expcol collection tuples, with the exception of the set of events that are part of the intersection between runCol collection tuples and myPrivate collection tuples.



$< evt >$
$\backslash$
$\mathbf{\Delta}^{\cup/\lambda(c,evt).<evt>}_{\lambda(<c,evt>).(true)}$
$\cap$
evt
$\mu^{event:[c.event]}_{(true)}$
c
$expCol$
$\mathbf{\Delta}^{\cup/\lambda(<c,evt>).<evt>}_{\lambda(<c,evt>).(true)}$
evt
$\mu^{evt:[c.event]}_{(true)}$
c
$runCol$
$\mathbf{\Delta}^{\cup/\lambda(<c,evt>).<evt>}_{\lambda(<c,evt>).(true)}$
evt
$\mu^{evt:[c.event]}_{(true)}$
c
$myPrivateCol$

Figure 8.22: Mapping result of collection query example

The Event Specification Operator

$$[\![\perp]\!]_E = \langle\!\langle [\![Q_{Collection}]\!]_C \rangle\!\rangle^{[evt]} \tag{E_0}$$

$$[\![\bigtriangledown_{pred}]\!]_E = \langle\!\langle \sigma_{\lambda(<evt>).(pred)}([\![Q_{Collection}]\!]_C) \rangle\!\rangle^{[evt]} \tag{E_1}$$

Figure 8.23: Mapping the Event specification operator

In case the Event specification operator is omitted, the result of this operator will be the resulting set of the evaluation $[\![Q_{Collection}]\!]_C$.

We can explain the semantics for the rule E_1 in the following manner: The resulting set of tuples $< evt >$ of the query plan mapped by $[\![Q_{Collection}]\!]_C$ are fed into the Selecion operator in order to discard the the tuples that do not validate the predicate specified in `pred`. Basically, for each variable evt the σ operator constructs a stream of tuples $\{< evt : Event >\}$, where each tuple satisfies the condition *pred*.

Let us assume we want to filter out all the events coming from the query plan that is the result of the first mapping step ($[\![Q_{Collection}]\!]_C$), with the filter predicate "$evt.bx = 3$", where bx is an attribute of event. We would make use of our mapping rules like in example 2. The result can be better visualized in Fig.8.25.

Example 2:

$[\![\bigtriangledown_{'evt.bx=3'}]\!]_E =$

$(E1) \quad = \sigma_{\lambda(evt).(pred('evt.bx=3'))}([\![Q_{Collection}]\!]_C)$

$$\begin{tikzcd} {} \arrow[d, dash, "< evt >"] \\ \sigma_{\lambda(<evt>).('evt.bx=3')} \arrow[d, dash, "\text{evt}"] \\ [\![Q_{Collection}]\!]_C \end{tikzcd}$$

Figure 8.24: Mapping result of an example of Event Specification

The Decay Specification Operators

$$[\![\bot]\!]_D = [\![Q_{Event}]\!]_E \qquad (D_{0_a})$$

$$[\![\emptyset]\!]_D = \bot \qquad (D_{0_b})$$

$$[\![\mathbf{H}|\mathbf{T}]\!]_D = \Delta^{\cup/\lambda(<tuple_1,tuple_2>).([evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))}_{\lambda(<tuple_1,tuple_2>).(true)}([\![\mathbf{H}]\!]_D \bowtie_{tuple_1.evt.id=tuple_2.evt.id} [\![\mathbf{T}]\!]_D) \qquad (D_1)$$

Figure 8.25: Translation rules for the selection operator

The mapping rules D_0 presented in 8.25 exist especially to deal with empty sets. If there is no Decay operator, the considered result set will be the complete input set returned by $[\![Q_{Event}]\!]_E$, using the query rule D_{0_a}. Rule D_{0_b} deals with an empty list of operators.

The rule D_1 is important to deal with the result of several isolated decays drawn by the user. In fact, the semantics of different unconnected decays is specified by this rule to be a stream of tuples resulting from the join operation over the streams of their individual results.

The Selection Operator

$$\begin{aligned}
&[\![\bigcirc^{name:path}_{pred/lattrib}]\!]_D = \\
&\quad = \langle\!\langle \mathbf{\Delta}^{\cup/\lambda(evt,name).<evt,name,lattrib>}_{\lambda(evt,name).(true)} (\\
&\qquad \mu^{name:evt.path}_{\lambda(<evt>).(pred)}([\![Q_{Event}]\!]_E)) \\
&\quad \rangle\!\rangle^{[\mathbf{evt},\mathbf{name},\mathbf{lattrib}]} \qquad (D_2)
\end{aligned}$$

Figure 8.26: Translation rules for the selection operator

This operator will be defined in the algebraic notation in the following way: the input will be the set of events (evt) result of the query subplan $[\![Q_{Event}]\!]_E$ and it will start with a generation of a stream $\{< evt >\}$.

Suppose, as in our example 3, that the user simply wants to retrieve from the system all the particles with positive energy and existing mass. It is required that the result stream has for each tuple the values of the computation of the square root of the sum of the squared px, py and pz. The result of the mapping of this query by using our just defined rules would look like in Fig.8.27.

Example 3:

$$\begin{aligned}
&[\![\bigcirc^{myparticle:Particle}_{'Energy>0\ and\ mass>0'/\{p=sqrt(px^2+py^2+pz^2),b=mass-0.1\}}]\!]_D = \\
(D_2)\quad &= \langle\!\langle \mathbf{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle,p=sqrt(px^2+py^2+pz^2),b=mass-0.1>}_{\lambda(evt,myparticle).(true)} (\\
&\qquad \mu^{myparticle:[evt.particle]}_{\lambda(evt,myparticle).(pred)}([\![Q_{Event}]\!]_E))\rangle\!\rangle^{[\mathbf{evt},\mathbf{name},\mathbf{p},\mathbf{b}]}
\end{aligned}$$

$$\begin{tikzcd}
{} \\
\mathbf{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle,p=sqrt(px^2+py^2+pz^2),b=mass-0.1>}_{\lambda(evt,myparticle).(true)} \arrow[u, dash, "< evt, myparticle, p, b >"'] \\
\mu^{myparticle:[evt.Particle]}_{\lambda(evt,myparticle).(pred('Energy>0\ and\ mass>0'))} \arrow[u, dash, "\text{myparticle}"'] \\
[\![Q_{Event}]\!]_E \arrow[u, dash, "\text{evt}"']
\end{tikzcd}$$

Figure 8.27: Simple Selection example

Transformer

$$\llbracket \text{●}^{\,name:\perp}_{\,pred/lattrib} \rightarrow \bigcirc \rightarrow [\mathbf{H}|\mathbf{T}]\rrbracket_D = \\ \langle\!\langle \boldsymbol{\Delta}^{\cup/\lambda(w').w'\circ lattrib}_{\lambda(w').(pred)}(\llbracket \bigcirc \rightarrow [\mathbf{H}|\mathbf{T}]\rrbracket^{\mathbf{w'}}_D)\rangle\!\rangle^{[\mathbf{w'},\mathbf{lattrib}]} \tag{D_3}$$

$$\llbracket \bigcirc \rightarrow [\mathbf{H}|\mathbf{T}]\rrbracket_D = \\ \langle\!\langle \Delta^{\cup/\lambda(<tuple_1,tuple_2>).([evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))}_{\lambda(<tuple_1,tuple_2>).(true)}(\\ \llbracket \bigcirc \rightarrow \mathbf{T}\rrbracket^{\mathbf{tuple_1}}_D \bowtie_{\text{“}tuple_1.evt.id=tuple_2.evt.id''} \llbracket \mathbf{H}\rrbracket^{\mathbf{tuple_2}}_D) \\ \rangle\!\rangle^{[\mathbf{evt},\mathcal{A}(\mathbf{tuple_1})/\mathbf{evt},\mathcal{A}(\mathbf{tuple_2})/\mathbf{evt}]} \tag{D_4}$$

$$\llbracket \bigcirc^{name:udtf}_{pred/lattrib} \rightarrow [\mathbf{H}|\emptyset]\rrbracket_D = \\ \langle\!\langle \boldsymbol{\Delta}^{\cup/\lambda(w).(w\circ[name:udtf(w)]\circ lattrib)}_{\lambda(w).pred}(\llbracket \mathbf{H}\rrbracket^{w}_D)\rangle\!\rangle^{[\mathcal{A}(\mathbf{w}),\mathbf{name},\mathcal{A}(\mathbf{lattrib})]} \tag{D_5}$$

Figure 8.28: Translation rules from the Transformer operator

The mapping of the description of the transformation operation is defined mainly by three translation rules. The first, D_3, is responsible for starting to map the chain that links the resulting tuples to the transformer operator and the rest of the decay tuples. The first thing is to interpret the type structure of the tuple that will be the result and leave the rest of the mapping to the rules D_4 and D_5. This means that a reduction to the result of the mapping of the pair composed by the transformer operator and the list of decayed particles is set. The rule D_4 is responsible for recursively mapping the operators in the several branches of the decay into joins and D_5 stops the recursion in the last element and maps the transformer operator itself into a reduction.

In Example 4, we will transform a decay query. Here it is described by two particles (one with positive mass and the other negative), that decay from a vertex (myvertex) the values of which are computed by using the transformation function Transform, if the sum of both masses is greater than 0.5. The result of applying the transformation rules can be observed in the query plan of Fig.8.29.

Example 4:

$$\llbracket \bullet^{myvertex:\perp}_{true/\{\}} \rightarrow \bigcirc^{mytrans:Transform}_{'\mu^+.mass+\mu^-.mass>0.5'/\{\}} \rightarrow$$
$$\{\bigcirc^{\mu^+:Particle}_{'energy>0'/\{\}}, \bigcirc^{\mu^-:Particle}_{'energy<0'/\{\}}\}\rrbracket_D =$$

$$(D_3) \quad = \langle\langle \mathbf{\Delta}^{\cup/\lambda(w').<w'>}_{pred(w')(true)}(\llbracket \bigcirc^{mytrans:Transform}_{'\mu^+.mass+\mu^-.mass>0.5'/\{\}} \rightarrow$$
$$\{\bigcirc^{\mu^+:Particle}_{'energy>0'/\{\}}, \bigcirc^{\mu^-:Particle}_{'energy<0'/\{\}}\}\rrbracket^{\mathbf{w'}}_D)$$
$$\rangle\rangle^{[\mathbf{w'}]}$$

$$(D_4) \quad = \langle\langle \mathbf{\Delta}^{\cup/\lambda(w_1,w_2).<w_1,w_2>}_{pred(w_1,w_2)(true)}($$
$$\langle\langle \mathbf{\Delta}^{[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt)}_{\lambda(<tuple_1,tuple_2>).true}($$
$$\llbracket \bigcirc^{mytrans:Transform}_{'\mu^+.mass+\mu^-.mass>0.5'/\{\}} \rightarrow \bigcirc^{\mu^-:Particle}_{'energy<0'/\{\}}\rrbracket^{\mathbf{tuple_1}}_D$$
$$\bowtie_{tuple_1.evt.id=tuple_2.evt.id}\llbracket \bigcirc^{\mu^+:Particle}_{'energy>0'/\{\}}\rrbracket^{\mathbf{tuple_2}}_D$$
$$)\rangle\rangle^{[\mathbf{evt},\mathcal{A}(\mathbf{tuple_1/evt}),\mathcal{A}(\mathbf{tuple_2/evt})]})$$
$$\rangle\rangle^{[\mathbf{w'}=[\mathbf{evt},\mathcal{A}(\mathbf{tuple_1/evt}),\mathcal{A}(\mathbf{tuple_2/evt})]]}$$

$$(D_5) \quad = \langle\langle \mathbf{\Delta}^{\cup/\lambda(w,mytrans,evt,\mu^+).<w,mytrans,evt,\mu^+>}_{pred(w,mytrans,evt,\mu^+)(true)}(\langle\langle$$
$$\mathbf{\Delta}^{[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt)}_{\lambda(<tuple_1,tuple_2>).true}($$
$$\langle\langle \mathbf{\Delta}^{\cup/\lambda(w).<w,mytrans=Transform>}_{\lambda(w).pred('\mu^+.mass+\mu^-.mass>0.5')}(\llbracket \bigcirc^{\mu^-:Particle}_{'energy<0'/\{\}}\rrbracket^{w}_{D})\rangle\rangle^{\mathbf{tuple_1}=[\mathcal{A}(\mathbf{w}),\mathbf{mytrans}]}$$
$$\bowtie_{tuple_1.evt.id=tuple_2.evt.id}\llbracket \bigcirc^{\mu^+:Particle}_{'energy>0'/\{\}}\rrbracket^{\mathbf{tuple_2}=[\mathbf{evt},\mu^+]}_D$$
$$\rangle\rangle^{[\mathbf{evt},\mathcal{A}(\mathbf{w/evt}),\mathbf{mytrans},\mu^+]})$$
$$\rangle\rangle^{[\mathbf{evt},\mathcal{A}(\mathbf{w/evt}),\mathbf{mytrans},\mu^+]}$$

$$(D_2) \quad = \langle\langle \mathbf{\Delta}^{\cup/\lambda(evt,\mu^-,mytrans,evt,\mu^+).<evt,\mu^-,mytrans,evt,\mu^+>}_{pred(evt,\mu^-,mytrans,evt,\mu^+)(true)}(\langle\langle$$
$$\mathbf{\Delta}^{[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt)}_{\lambda(<tuple_1,tuple_2>).true}($$
$$\langle\langle \mathbf{\Delta}^{\cup/\lambda(evt,\mu^-,mytrans,evt,\mu^+).<evt,\mu^-,mytrans,evt,\mu^+,mytrans=Transform]>}_{\lambda(evt,\mu^-,mytrans,evt,\mu^+).pred('\mu^+.mass+\mu^-.mass>0.5')}($$
$$\langle\langle \mathbf{\Delta}^{\cup/\lambda(evt,\mu^-).<evt,\mu^->}_{\lambda(evt,\mu^-).(true)}($$
$$\mu^{\lambda(evt).[\mu^-:evt.Particle]}_{\lambda(evt,\mu^-).(pred('energy<0'))}(\llbracket Q_{Event}\rrbracket_E))\rangle\rangle^{\mathbf{w}=[\mathbf{evt},\mu^-]})\rangle\rangle^{[\mathbf{evt},\mu^-,\mathbf{mytrans}]}$$
$$\bowtie_{tuple_1.evt.id=tuple_2.evt.id}\langle\langle \mathbf{\Delta}^{\cup/\lambda(evt,\mu^+).<evt,\mu^+>}_{\lambda(evt,\mu^+).(true)}($$
$$\mu^{\lambda(evt).[\mu^+:evt.Particle]}_{\lambda(evt,\mu^+).(pred('energy>0'))}(\llbracket Q_{Event}\rrbracket_E))\rangle\rangle^{[\mathbf{evt},\mu^+]}$$
$$\rangle\rangle^{[\mathbf{evt},\mu^-,\mathbf{mytrans},\mu^+]})$$
$$\rangle\rangle^{[\mathbf{evt},\mu^-,\mathbf{mytrans},\mu^+]}$$

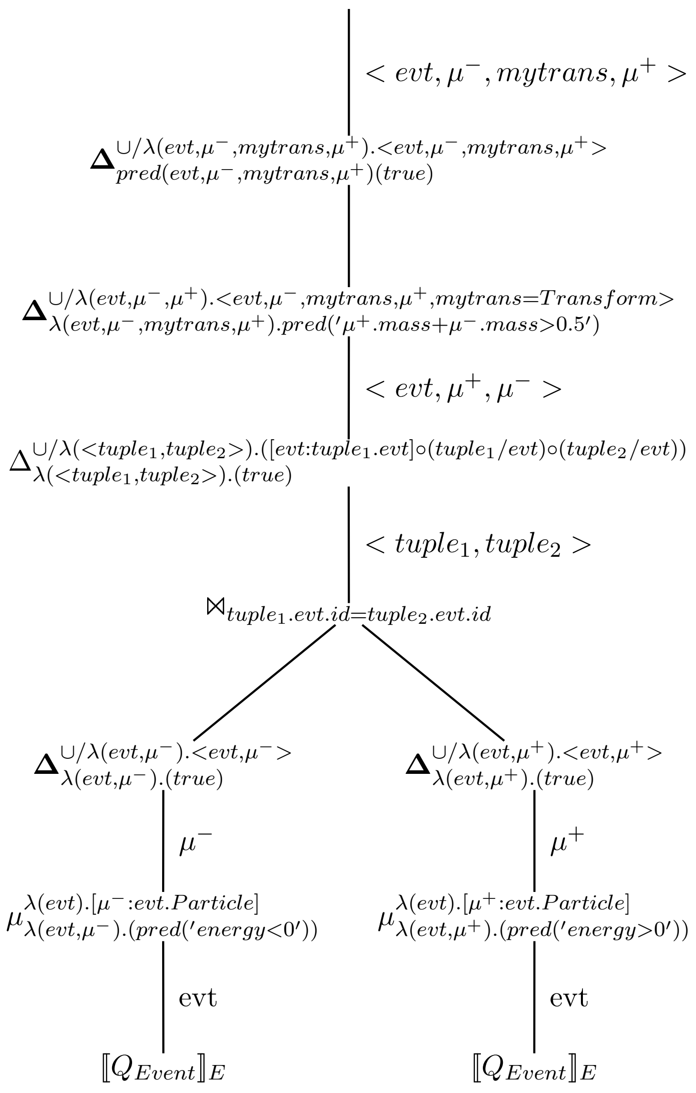

$< evt, \mu^-, mytrans, \mu^+ >$
$\Delta^{\cup/\lambda(evt,\mu^-,mytrans,\mu^+).<evt,\mu^-,mytrans,\mu^+>}_{pred(evt,\mu^-,mytrans,\mu^+)(true)}$
$\Delta^{\cup/\lambda(evt,\mu^-,\mu^+).<evt,\mu^-,mytrans,\mu^+,mytrans=Transform>}_{\lambda(evt,\mu^-,mytrans,\mu^+).pred('\mu^+.mass+\mu^-.mass>0.5')}$
$< evt, \mu^+, \mu^- >$
$\Delta^{\cup/\lambda(<tuple_1,tuple_2>).([evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))}_{\lambda(<tuple_1,tuple_2>).(true)}$
$< tuple_1, tuple_2 >$
$\bowtie_{tuple_1.evt.id=tuple_2.evt.id}$
$\Delta^{\cup/\lambda(evt,\mu^-).<evt,\mu^->}_{\lambda(evt,\mu^-).(true)}$
$\Delta^{\cup/\lambda(evt,\mu^+).<evt,\mu^+>}_{\lambda(evt,\mu^+).(true)}$
μ^-
μ^+
$\mu^{\lambda(evt).[\mu^-:evt.Particle]}_{\lambda(evt,\mu^-).(pred('energy<0'))}$
$\mu^{\lambda(evt).[\mu^+:evt.Particle]}_{\lambda(evt,\mu^+).(pred('energy>0'))}$
evt
evt
$[\![Q_{Event}]\!]_E$
$[\![Q_{Event}]\!]_E$

Figure 8.29: Example of the mapping of the transformer operator

Reference Operator - Path Expressions

$$[\![\bigcirc^{name:path}_{pred/lattrib}[\mathbf{H}|\mathbf{T}]]\!]^{path}_{D}(Q) = [\![[\mathbf{H}|\mathbf{T}]]\!]^{name}_{D}([\![\bigcirc]^{name:path}_{pred/lattrib}]_{D}(Q)) \qquad (D_6)$$

$$[\![[\mathbf{H}|\mathbf{T}]]\!]^{path}_{D}(Q) = [\![\mathbf{T}]\!]^{path}_{D}([\![\mathbf{H}]\!]^{path}_{D}Q) \qquad (D_7)$$

$$\begin{aligned}&[\![\bullet\rightarrow\bigcirc^{name:\perp}_{pred/lattrib}[H|T]]\!]^{name'}_{D}(Q) = \\ &\qquad [\![H|T]\!]^{name}[\![\bullet\rightarrow\bigcirc^{name:\perp}_{pred/lattrib}]\!]^{name'}_{D}(Q)\end{aligned} \qquad (D_8)$$

$$[\![\bullet\rightarrow\bigcirc^{name:\perp}_{pred/lattrib}]\!]^{name'}_{D}(Q) = \mu^{name:[name'.path])}_{pred}(Q) \qquad (D_9)$$

$$\begin{aligned}&[\![\bullet\hookrightarrow\bigcirc^{name:\perp}_{pred/lattrib}[H|T]]\!]^{name'}_{D}(Q) = \\ &\qquad [\![H|T]\!]^{name}[\![\bullet\hookrightarrow\bigcirc^{name:\perp}_{pred/lattrib}]\!]^{name'}_{D}(Q)\end{aligned} \qquad (D_{10})$$

$$[\![\bullet\hookrightarrow\bigcirc^{name:\perp}_{pred/lattrib}]\!]^{name'}_{D}(Q) = =\!\mu^{name:[name'.path])}_{pred}(Q) \qquad (D_{11})$$

Figure 8.30: Translation rules for the references operators

In Example 5, we show a mapping which uses both mandatory and non-mandatory path expressions. We want to select all the particles with mandatory path expressions to Vertex and a corresponding MonteCarlo simulation particle. We also want to return the simulated MonteCarlo Vertex if there is any reference to it as well. The mapping is somewhat more dense than our other examples. The result is a sequence of unnesting operations, which can be better visualized with the execution plan of 8.31.

Example 5:

$$\begin{aligned}&[\![\bigcirc^{myparticle:Particle}_{true/\{\}}(\bullet\rightarrow\bigcirc^{primvertex:Vertex}_{true/\{\}}, \\ &\qquad (\bigcirc^{simparticle:MCParticle}_{true/\{\}}(\bullet\hookrightarrow\bigcirc^{simprimvertex:MCVertex}_{true/\{\}})))]\!]_{D} =\end{aligned}$$

(D_6) $=[\![\perp\bullet\rightarrow(\bigcirc^{primvertex:Vertex}_{true/\{\}}, (\bigcirc^{simparticle:MCParticle}_{true/\{\}}\bullet\hookrightarrow$ $\bigcirc^{simprimvertex:MCVertex}_{true/\{\}}))]\!]^{myparticle}_{D}([\![\bigcirc^{myparticle:Particle}_{true/\{\}}]\!]_{D})$

$$(D_7) \quad = \llbracket(\bullet \rightarrow \bigcirc^{simparticle:MCParticle}_{true/\{\}}(\bullet \hookrightarrow \bigcirc^{simprimvertex:MCVertex}_{true/\{\}})\rrbracket^{myparticle}_{D}$$
$$(\llbracket\bigcirc^{primvertex:Vertex}_{true/\{\}}\rrbracket^{myparticle}_{D}(\llbracket\bigcirc^{myparticle:Particle}_{true/\{\}}\rrbracket_{D})$$

$$(D_7) \quad = \llbracket\bullet \hookrightarrow \bigcirc^{simvertex:MCVertex}_{true/\{\}}\rrbracket^{simparticle}$$
$$(\llbracket\bullet \rightarrow \bigcirc^{simparticle:MCParticle}_{true/\{\}}\rrbracket^{myparticle}$$
$$(\llbracket\bullet \rightarrow \bigcirc^{primvertex:Vertex}_{true/\{\}}\rrbracket^{myparticle}_{D}$$
$$(\llbracket\bigcirc^{myparticle:Particle}_{true/\{\}}\rrbracket_{D})))$$

$$(D_2) \quad = \llbracket\bullet \hookrightarrow \bigcirc^{simvertex:MCVertex}_{true/\{\}}\rrbracket^{simparticle}$$
$$(\llbracket\bullet \rightarrow \bigcirc^{simparticle:MCParticle}_{true/\{\}}\rrbracket^{myparticle}$$
$$(\llbracket\bullet \rightarrow \bigcirc^{primvertex:Vertex}_{true/\{\}}\rrbracket^{myparticle}_{D}$$
$$(\langle\langle\boldsymbol{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{(true)}(\mu^{myparticle:[evt.Particle]}_{(true)}($$
$$\llbracket Q_{Event}\rrbracket_{E}))\rangle\rangle^{[\mathbf{evt},\mathbf{myparticle}]})))$$

$$(D_9) \quad = \llbracket\bullet \hookrightarrow \bigcirc^{simvertex:MCVertex}_{true/\{\}}\rrbracket^{simparticle}$$
$$(\llbracket\bullet \rightarrow \bigcirc^{simparticle:MCParticle}_{true/\{\}}\rrbracket^{myparticle}$$
$$(\langle\langle\mu^{primvertex:[myparticle.Vertex]}_{(true)}$$
$$(\boldsymbol{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{(true)}(\mu^{myparticle:[evt.Particle]}_{(true)}($$
$$\llbracket Q_{Event}\rrbracket_{E})))\rangle\rangle^{[\mathbf{evt},\mathbf{myparticle},\mathbf{primvertex}]}))$$

$$(D_9) \quad = \llbracket\bullet \hookrightarrow \bigcirc^{simvertex:MCVertex}_{true/\{\}}\rrbracket^{simparticle}$$
$$(\langle\langle\mu^{simparticle:[myparticle.MCParticle]}_{(true)}$$
$$(\mu^{primvertex:[myparticle.Vertex]}_{(true)}$$
$$(\boldsymbol{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{(true)}(\mu^{myparticle:[evt.Particle]}_{(true)}($$
$$\llbracket Q_{Event}\rrbracket_{E}))))\rangle\rangle^{[\mathbf{evt},\mathbf{myparticle},\mathbf{primvertex},\mathbf{simparticle}]})$$

$$(D_{11})$$
$$= \langle\langle = \mu^{simprimvertex:[simparticle.MCVertex]}_{(true)}($$
$$\mu^{simparticle:[myparticle.MCParticle]>}_{(true)}($$
$$\mu^{primvertex:[myparticle.Vertex]>}_{(true)}($$
$$\boldsymbol{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{(true)}(\mu^{myparticle:[evt.Particle]}_{(true)}($$
$$\llbracket Q_{Event}\rrbracket_{E})))))\rangle\rangle^{[\mathbf{evt},\mathbf{myparticle},\mathbf{primvertex},\mathbf{simparticle},\mathbf{simprimvertex}]}$$

$$\begin{tikzcd}
{} \arrow[d, dash, "{< evt, myparticle, primvertex, simparticle, simprimvertex >}"] \\
{=\mu^{simprimvertex:[simparticle.MCVertex]}_{(true)}} \arrow[d, dash, "\text{simparticle}"] \\
{\mu^{simparticle:[myparticle.MCParticle]}_{(true)}} \arrow[d, dash, "\text{primvertex}"] \\
{\mu^{primvertex:[myparticle.Vertex]}_{(true)}} \arrow[d, dash, "{< evt, myparticle >}"] \\
{\Delta^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{(true)}} \arrow[d, dash, "\text{myparticle}"] \\
{\mu^{\lambda myparticle:[evt.Particle]}_{(true)}} \arrow[d, dash, "\text{evt}"] \\
{[\![Q_{Event}]\!]_E}
\end{tikzcd}$$

Figure 8.31: Example of the mapping of the reference operator

Aggregator

$$[\![\square^{aggfunc(expr)}_{pred}(Q)]\!]_D = \langle\!\langle \mathbf{\Gamma}^{aggfunc/\lambda(w).([value:expr(w)])/evt.id}_{\lambda(w).(pred)}([\![Q]\!]_D)\rangle\!\rangle^{[\mathbf{value}:\mathbf{expr(w)}]} \qquad (D_{13})$$
$$aggfunc \in \{max, min, avg\}$$

$$[\![\square^{agg(expr)}_{pred}(Q)]\!]_D = \langle\!\langle \mathbf{\Gamma}^{agg/\lambda(w).([value:expr(w),tuple:w])/evt.id}_{\lambda(w).(pred)}([\![Q]\!]_D)\rangle\!\rangle^{[\mathbf{value}:\mathbf{expr(w)},\mathbf{tuple}:\mathbf{w}]} \qquad (D_{14})$$
$$agg \in \{Max, Min\}$$

Figure 8.32: Translation rules for the aggregator operator.

A nest operator groups the tuples by evt. Each group will then be reduced by the aggregator function Max or Min, and will produce as result a set of tuples (of type $\{\tau\}$), where each tuple is the maximum or minimum expr value.

In order to show the usage of these rules we have Example 6. Here, we have a selection of all the particles with positive energy, and we want to group them by event and determine the one that has the maximum value for the mass attribute with the restriction that it should be greater than 0.65. The result of the mapping can be better visualized in the execution plan of Fig.8.33.

Example 6:

$$[\![\square^{Max(myparticle.mass)}_{'myparticle.mass>0.65'}(\bigcirc^{myparticle:Particle}_{'Energy>0'/\{\}})]\!]_D =$$

$$(D_{14}) \quad = \mathbf{\Gamma}^{Max/\lambda(w).([value:myparticle.mass,tuple:w])/evt.id}_{\lambda(w).('myparticle.mass>0.65')}(\square^{Max(myparticle.mass)}_{'myparticle.mass>0.65'}([\![\bigcirc^{myparticle:Particle}_{'Energy>0'/\{\}}]\!]_D))$$

$$(D_2) \quad = \langle\!\langle \mathbf{\Gamma}^{Max/\lambda(w).([value:myparticle.mass,tuple:w])/evt.id}_{\lambda(w).('myparticle.mass>0.65')}(\langle\!\langle \mathbf{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{\lambda(evt,myparticle).(true)}($$

$$\mu^{\lambda(evt).[myparticle:evt.Particle]}_{\lambda(evt,myparticle).(pred('Energy>0'))}(\llbracket Q_{Event}\rrbracket_E))$$
$$\rangle\!\rangle^{[\mathbf{evt},\mathbf{myparticle}]})\rangle\!\rangle^{[\mathbf{w}=[\mathbf{evt},\mathbf{myparticle}]]}$$

$$=\langle\!\langle$$
$$\mathbf{\Gamma}^{Max/[value:myparticle.mass,tuple:<evt,myparticle>]/evt.id}_{\lambda(evt,myparticle).('myparticle.mass>0.65')}($$
$$\mathbf{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{\lambda(evt,myparticle).(true)}($$
$$\mu^{\lambda(evt).[myparticle:evt.Particle]}_{\lambda(evt,myparticle).('Energy>0')}(\llbracket Q_{Event}\rrbracket_E)$$
$$)$$
$$\rangle\!\rangle^{[\mathbf{evt},\mathbf{myparticle}]}$$

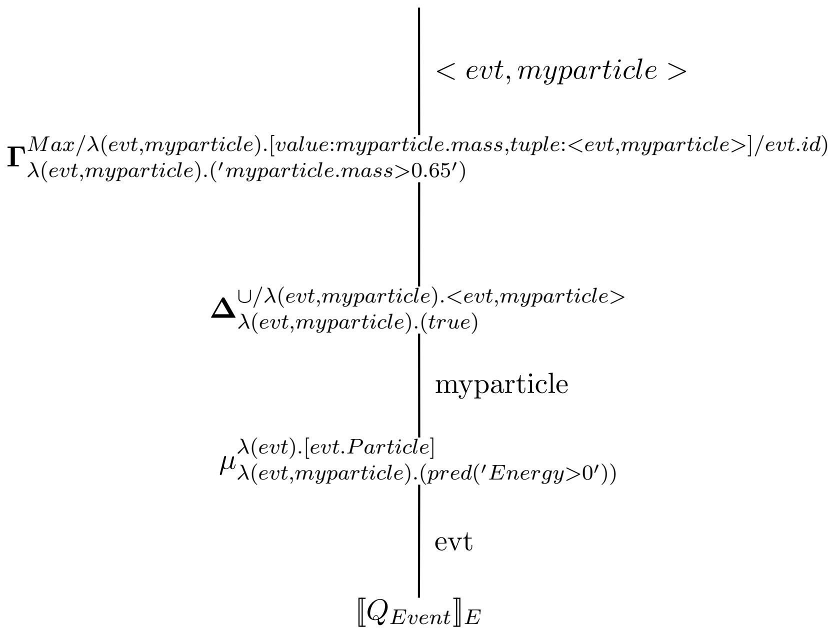

$< evt, myparticle >$
$\mathbf{\Gamma}^{Max/\lambda(evt,myparticle).[value:myparticle.mass,tuple:<evt,myparticle>]/evt.id)}_{\lambda(evt,myparticle).('myparticle.mass>0.65')}$
$\mathbf{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{\lambda(evt,myparticle).(true)}$
myparticle
$\mu^{\lambda(evt).[evt.Particle]}_{\lambda(evt,myparticle).(pred('Energy>0'))}$
evt
$\llbracket Q_{Event}\rrbracket_E$

Figure 8.33: Result of a simple aggregator operator example

Minimal Distance

$$
\begin{aligned}
&[\![Q_1 \diamond^{fdist}_{expr/agg} \rightarrow Q_2]\!]_D = \\
&\quad = \langle\!\langle \mathbf{\Gamma}^{agg/<value=fdist,tuple=([evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))>/[evt.id])}_{true}(\\
&\quad \Delta^{\cup/\lambda(<tuple_1,tuple_2>).([evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))}_{\lambda(<tuple_1,tuple_2>).(true)}(\\
&\quad\ \ [\![Q_2]\!]_D \bowtie_{[tuple_1.evt.id]=[tuple_2.evt.id]\wedge expr(fdist)} [\![Q_1]\!]_D)) \\
&\quad \rangle\!\rangle^{[\mathbf{evt},\mathcal{A}(\mathbf{tuple_1/evt}),\mathcal{A}(\mathbf{tuple_2/evt})]} \qquad (D_{15}) \\
&agg \in \{Max, Min\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![Q_1 \diamond^{fdist}_{expr/agg} \hookrightarrow Q_2]\!]_D = \\
&\quad = \langle\!\langle \mathbf{\Gamma}^{agg/<value=fdist,tuple=[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))>/[evt.id]))}_{true}(\\
&\quad \Delta^{\cup/\lambda(<tuple_1,tuple_2>).([evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))}_{\lambda(<tuple_1,tuple_2>).(true)}(\\
&\quad\ \ [\![Q_2]\!]_D ⟕_{[tuple_1.evt.id]=[tuple_2.evt.id]\wedge expr(fdist)} [\![Q_1]\!]_D)) \\
&\quad \rangle\!\rangle^{[\mathbf{evt},\mathcal{A}(\mathbf{tuple_1/evt}),\mathcal{A}(\mathbf{tuple_2/evt})]} \qquad (D_{16}) \\
&agg \in \{Max, Min\}
\end{aligned}
$$

Figure 8.34: Translation rules for the minimal distance operators.

Both rules D_{15},(mandatory), and D_{16}, (non-mandatory), are very similar. The difference between them is only the join operator. Tn the case of the non-mandatory version we will use a left-outer join.

The join constructs a set of pairs of all tuples which result from the query plan mapped from Q_1 that are associated with the ones of the query plan mapped from Q_2 through the validation of the logical expression $expr$. In other words, for each event all resulting tuples $tuple_1$ and $tuple_2$ are combined. In the case of the left-outer join of the non-mandatory case, the result is a set of pairs that combines all values of $tuple_1$ with all values of $tuple_2$, if their condition predicate is true. In the case that no $tuple_2$ values exist (or are not valid) for every $tuple_1$ a pair $< tuple_1, null >$ is returned.

The nest operator groups the tuple input,$[evt : tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt)$ and their evaluated distance function, $fdist$, by the attribute value $[evt.id]$ of each tuple. Each group is reduced by the aggregator function, Max or Min, and the result is the first tuple found that verifies the aggregation.

In our Example 7, we want to determine for the Particles and Vertexes the pairs per event with the result of the minimal distance greater than 0.12 (the minimal function is defined by the expression fdist). As we have used the non-mandatory version of the minimal distance operator in this example, the resulting tuple pair can have empty Vertex elements. The result of using our described rules is visualized in Fig.8.35.

Example 7:

$$fdist = \sqrt{(w_1.x - w_2.x)^2 + (w_1.y - w_2.y)^2 + (w_1.z - w_2.z)^2}$$

$$\llbracket \bigcirc^{myparticle:Particle}_{\lambda(w_1).true/\{\}} \diamond^{fdist}_{fdist>0.12/Min} \hookrightarrow \bigcirc^{myvertex:Vertex}_{\lambda(w).true/\{\}} \rrbracket_D =$$

$$(D_{16}) \quad = \mathbf{\Gamma}^{agg/<value=fdist,tuple=[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))>/[evt.id]))}_{true}(\\ \llbracket \bigcirc^{myparticle:Particle}_{\lambda(w_1).true/\{\}} \rrbracket_D \diamond^{fdist}_{fdist>0.12/Min} \hookrightarrow \llbracket \bigcirc^{myvertex:Vertex}_{\lambda(w).true/\{\}} \rrbracket_D)$$

$$(D_2) \quad = \langle\langle \mathbf{\Gamma}^{agg/<value=fdist,tuple=[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt))>/[evt.id]))}_{true}(\\ \mathbf{\Delta}^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{\lambda(evt,myparticle).(true)}(\mu^{myparticle:[evt.Particle]}_{\lambda(evt,myparticle).(true)}(\llbracket Q_{Event} \rrbracket_E)) \\ \rangle\rangle^{[\mathbf{evt},\mathbf{myparticle}]} \\ \diamond^{fdist}_{fdist>0.12/Min} \hookrightarrow \\ \langle\langle \\ \mathbf{\Delta}^{\cup/\lambda(evt,myvertex).<evt,myvertex>}_{\lambda(evt,myvertex).(true)}(\mu^{\lambda(evt).[evt.Vertex]}_{\lambda(evt,myvertex).(true)}(\llbracket Q_{Event} \rrbracket_E)) \\ \rangle\rangle^{[\mathbf{evt},\mathbf{myvertex}]} \\ )\rangle\rangle^{[\mathbf{evt},\mathbf{tuple_1}=\mathbf{myparticle},\mathbf{tuple_2}=\mathbf{myvertex}]}$$

$$=\langle\langle\ \mathbf{\Gamma}^{Min/<value=fdist,tuple=[evt,myparticle,myvertex]>/[evt.id]}_{(true)} \\ (\Delta^{\cup/[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt)}_{\lambda(<tuple_1,tuple_2>).(true)}(\\ \langle\langle \mathbf{\Delta}^{\cup/<evt,myvertex>}_{\lambda(evt,myvertex).(true)}(\\ \mu^{\lambda(evt).[evt.Vertex]}_{\lambda(evt,myvertex).(true)}(\llbracket Q_{Event} \rrbracket_E))\rangle\rangle^{[\mathbf{evt},\mathbf{myvertex}]} \\ ⟕_{[tuple_1.evt.id]=[tuple_2.evt.id]\wedge fdist>0.12} \\ \langle\langle \mathbf{\Delta}^{\cup/<evt,myparticle>}_{\lambda(evt,myparticle).(true)}(\\ \mu^{\lambda(evt).[evt.Particle]}_{\lambda(evt,myparticle).(true)}(\llbracket Q_{Event} \rrbracket_E))\rangle\rangle^{[\mathbf{evt},\mathbf{myparticle}]} \\ ))\rangle\rangle^{[\mathbf{evt},\mathbf{myparticle},\mathbf{myvertex}]}$$

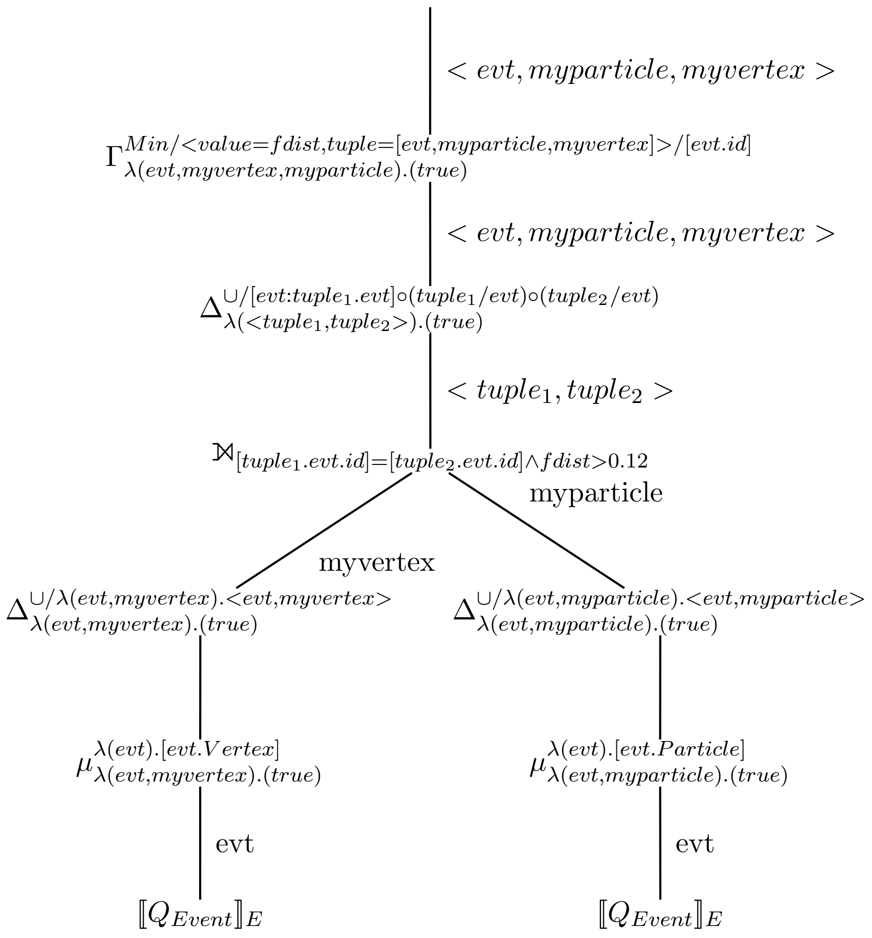

$< evt, myparticle, myvertex >$
$\Gamma^{Min/<value=fdist,tuple=[evt,myparticle,myvertex]>/[evt.id]}_{\lambda(evt,myvertex,myparticle).(true)}$
$< evt, myparticle, myvertex >$
$\Delta^{\cup/[evt:tuple_1.evt]\circ(tuple_1/evt)\circ(tuple_2/evt)}_{\lambda(<tuple_1,tuple_2>).(true)}$
$< tuple_1, tuple_2 >$
$\bowtie_{[tuple_1.evt.id]=[tuple_2.evt.id]\wedge fdist>0.12}$
myparticle
myvertex
$\Delta^{\cup/\lambda(evt,myvertex).<evt,myvertex>}_{\lambda(evt,myvertex).(true)}$
$\Delta^{\cup/\lambda(evt,myparticle).<evt,myparticle>}_{\lambda(evt,myparticle).(true)}$
$\mu^{\lambda(evt).[evt.Vertex]}_{\lambda(evt,myvertex).(true)}$
$\mu^{\lambda(evt).[evt.Particle]}_{\lambda(evt,myparticle).(true)}$
evt
evt
$[\![Q_{Event}]\!]_E$
$[\![Q_{Event}]\!]_E$

Figure 8.35: Result of the minimal distance example

Comparison

$$[\![\odot_{pred}(Q)]\!]_D = \langle\!\langle \sigma_{\lambda(w).pred}([\![Q]\!]_D)\rangle\!\rangle^{\mathbf{w}} \qquad (D_{17})$$

$$[\![\odot_{pred_1}(\odot_{pred_2}(Q))]\!]_D = [\![\odot_{pred_1\ and\ pred_2}([\![Q]\!]_D)]\!]_D \qquad (D_{18})$$

Figure 8.36: Translation rule of the comparison operator

By rule D_{17}, the selection operator σ filters out the tuples resulting from the rest of the query plan, generated by mapping the query Q. The rule D_{18} is a simplification that composes a conjunctive single predicate out of two.

A possible example of the usage of these rules can be like the one described in example 8. It is required that the mass of the vertex should be greater than the particle it decays from. The corresponding query plan generated by using our rules can be visualized in Fig.8.38.

Example 8:

$[\![\odot_{'myvertex.mass>myparticle.mass'}(Q)]\!]_D =$

$(D_{17})\quad \langle\!\langle \sigma_{\lambda(w).pred(w)('myvertex.mass>myparticle.mass')}([\![Q]\!]_D)\rangle\!\rangle^{\mathbf{w}}$

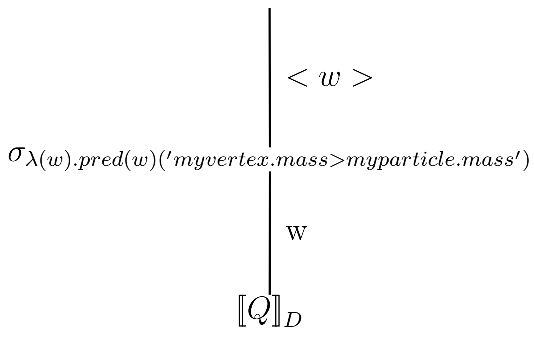

$< w >$
$\sigma_{\lambda(w).pred(w)('myvertex.mass>myparticle.mass')}$
w
$[\![Q]\!]_D$

Figure 8.37: Result of a comparison simple example

The Result Operators

$$[\![\bot]\!]_R = \langle\!\langle [\![Q_{Decay}]\!]_D \rangle\!\rangle^w \qquad (R_0)$$

$$[\![\boxed{\text{1D}}_{lattrib}]\!]_R = \boldsymbol{\Delta}^{H1D/\lambda(w).lattrib}_{\lambda(w).pred}(\langle\!\langle [\![Q_{Decay}]\!]_D \rangle\!\rangle^w) \qquad (R_1)$$

$$[\![\boxed{\text{2D}}_{lattrib}]\!]_R = \boldsymbol{\Delta}^{H2D/\lambda(w).lattrib}_{\lambda(w).pred}(\langle\!\langle [\![Q_{Decay}]\!]_D \rangle\!\rangle^w) \qquad (R_2)$$

$$[\![\boxed{\text{3D}}_{lattrib}]\!]_R = \boldsymbol{\Delta}^{H3D/\lambda(w).lattrib}_{\lambda(w).pred}(\langle\!\langle [\![Q_{Decay}]\!]_D \rangle\!\rangle^w) \qquad (R_3)$$

$$[\![\boxed{\#}^{aggfunc}_{head}]\!]_R = \boldsymbol{\Delta}^{aggfunc/\lambda(w).(head)}_{\lambda(w).(true)}(\langle\!\langle [\![Q_{Decay}]\!]_D \rangle\!\rangle^w) \qquad (R_4)$$

$$aggfunc \in \{max, min, sum, count, avg, ...\}$$

Figure 8.38: Result set transformation rules.

The rule R_0 states that in case of omission of result operators the result set will be the collection retrieved by the evaluation of $[\![Q_{Decay}]\!]_D$. The usefulness of this rule depends on the implementation of it, but could be considered as a way to feed some other tools on top of PHEASANT meant to manipulate the result in some different ways.

The aggregate functions to generate histograms can be defined as described in Fig.8.39.

$$\text{HiD},\ i \in \{1,2,3\}: \quad \{\tau\} \rightarrow \tau_H i$$

Figure 8.39: Signature of the histogram aggregate functions

In Fig.8.40, we exemplify the mapping of the value result operator. In this case, the user is interested in summing up all the energy values in the tuples that result from the algebra mapped by: $[\![Q_{Decay}]\!]_D$. Basically, this will represent a reduction on the tuple stream, with the aggregation function Sum.

Example 9:

$$[\![\boxed{\#}^{Sum}_{'myparticle.energy'}]\!]_R =$$

R_4 $\quad \mathbf{\Delta}^{Sum/\lambda(w).([r:myparticle.energy])}_{\lambda(w).(true)}(\langle\!\langle[\![Q_{Decay}]\!]_D\rangle\!\rangle^w)$

$$\begin{tikzcd} {} \arrow[d, dash] \\ \mathbf{\Delta}^{Sum/\lambda(w).([r:myparticle.energy])}_{\lambda(w).(true)} \arrow[d, dash, "< w >"] \\ {[\![Q_{Decay}]\!]_D} \end{tikzcd}$$

Figure 8.40: Transformation result of a result operator using the aggregation function Sum.

8.4 Summary

In this section, we have described the design approach for the HEP analysis query language.

A carefully selected alphabet notation was introduced. It aims to deal with the domain-specific concepts at all the stages of the query patterns. This is part of our global strategy to approach the optimization of the HEP analysis process.

We described the syntax of PHEASANT QL. We started with the grammar of the abstract syntax graph, which is more close to the visual language parsing requirements, and proceeded to a more abstract approach, although more easy to deal, which was the syntax tree grammar. Together with those definitions, rules to describe valid or false sentences in our language were set.

Finally, the semantics of the language were defined by making use of the translational semantics mapping. Syntax-to-syntax mapping from PHEASANT QL into an algebra was used.

Further optimization at this phase can be done either by proposing an alternative language and/or addition of new operators, or by tightening controlling the translation to the algebraic notation by adding more semantic rules.

Chapter 9

Prototype Framework - PHEASANT

In this chapter, we present the architecture of the framework of our implemented prototype.

We start with a general overview of this framework in Section 9.1. Afterwards, the several query transformation modules are detailed: First, we describe the user interface in section 9.2; then, in section 9.3, we explain the plan generator; and finally, in section 9.4, we present the code generator.

9.1 General Overview

It is not the purpose of this thesis to discuss a full-fledged implementation, but to present a proposed architecture of a feasible prototype. Therefore, we describe the architectural design and implementation in a very compact way.

In order to jump into the description of the three main modules, we start by introducing the reader to the general design. In this overview section, we first give a system engineering perspective of the framework, remembering the roles and use cases from the requirements. Then, we give an insight into the architectural design and, finally, we explain the technology used for implementing PHEASANT.

9.1.1 Roles and Use Cases

As reasoned in Chapter 7, inspired by the Domain Modeling approach we proposed to tackle the problem in HEP analysis by specifying a framework that supports different modeling levels [7] (see Fig.7.1). We reserve to ourselves the role of meta-meta-modelers. This means that we have set the data meta-meta model the framework has to deal with. The instances of this meta-meta model (see 9.1) are defined by the meta-modeler.

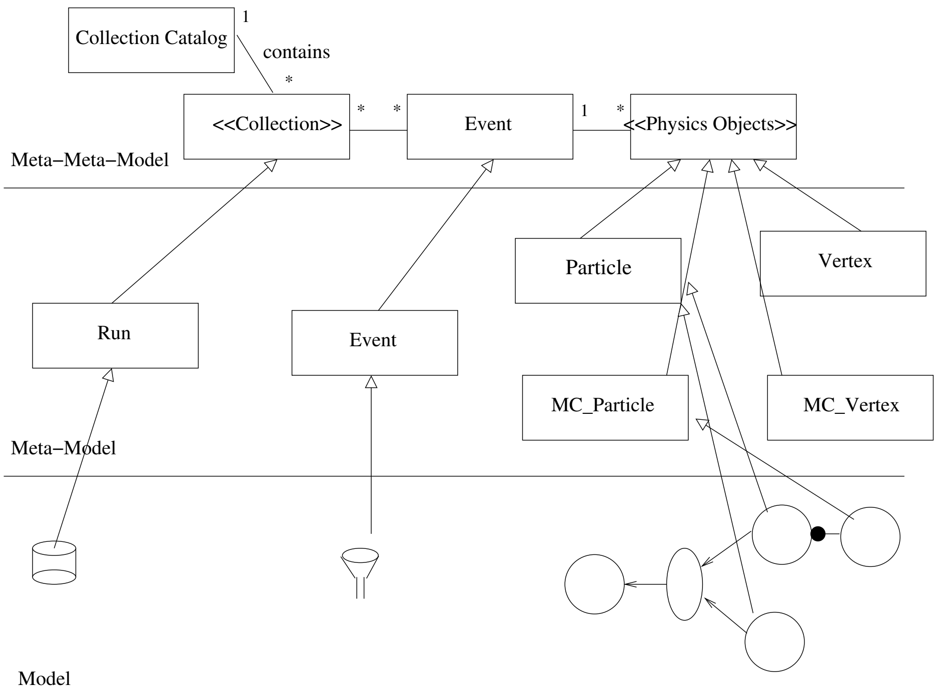


Figure 9.1: Model levels. The domain experts will deal with the meta-modeling of physics objects.

In this case, we assign the responsibility of meta-modeling to the team of developers in a specific experiment. The framework copes with the description of the specificities of the schema that have some variations in different experiments.

The physicist, who takes the role of the query modeler, is immediately aware of the changes in the instances of the meta-meta-model just by using the visual operators when modeling his query.

As shown in Fig.9.2, we have defined four main system actors. The final user interacts with the system by editing the query statement and requests it to be run.

The domain expert is responsible for defining the data schema, the user-defined functions and constants for the specific experiment.

The query storage base is the main repository for the queries and their results. It is responsible for dealing with the query history for each user and analysis. It can be seen as the repository of the meta-information concerning the query itself (dealing, for instance, with details like query graph, query result, version of the query, dates, author, time spent running, etc.).

Finally, the Physics Storage base is the experiment's specific storage framework that deals with the data to be analyzed.

In the first implementation of our prototype, we have concentrated on the dark grey use cases of Fig.9.2, leaving the rest for future work.

9.1.2 Architecture

The system was designed to cope with several query transformation phases required to deliver a target query source code that should be compiled and run against a specific physics storage base.

We have devised three main modules, as seen in Fig.9.3: user interface, plan generator and code generator. Each of these modules is described more deeply in the next sections.

The user interface deals with the user's query edition and interactively notifies the user of incorrect syntax. Internally, a Concrete Components Graph is maintained and simultaneously mapped, using the observers pattern, to a corresponding Abstract Syntax Graph, as described in the last chapter. We describe this module in more detail in Section 9.2.

The Plan Generator starts by interpreting the Abstract Syntax Graph and transforming it into an Abstract Syntax Tree (AST). Then, it runs an algorithm that walks down the AST and generates the corresponding algebraic query plan (QP). Details are described in Section 9.3.

Finally, the query Code Generator looks at the algebraic query plan, optimizes it at the algebraic logical level and generates the physical operators. In the sequence of that, a new algorithm generates the required source code to be compiled and run against the storage base. This module

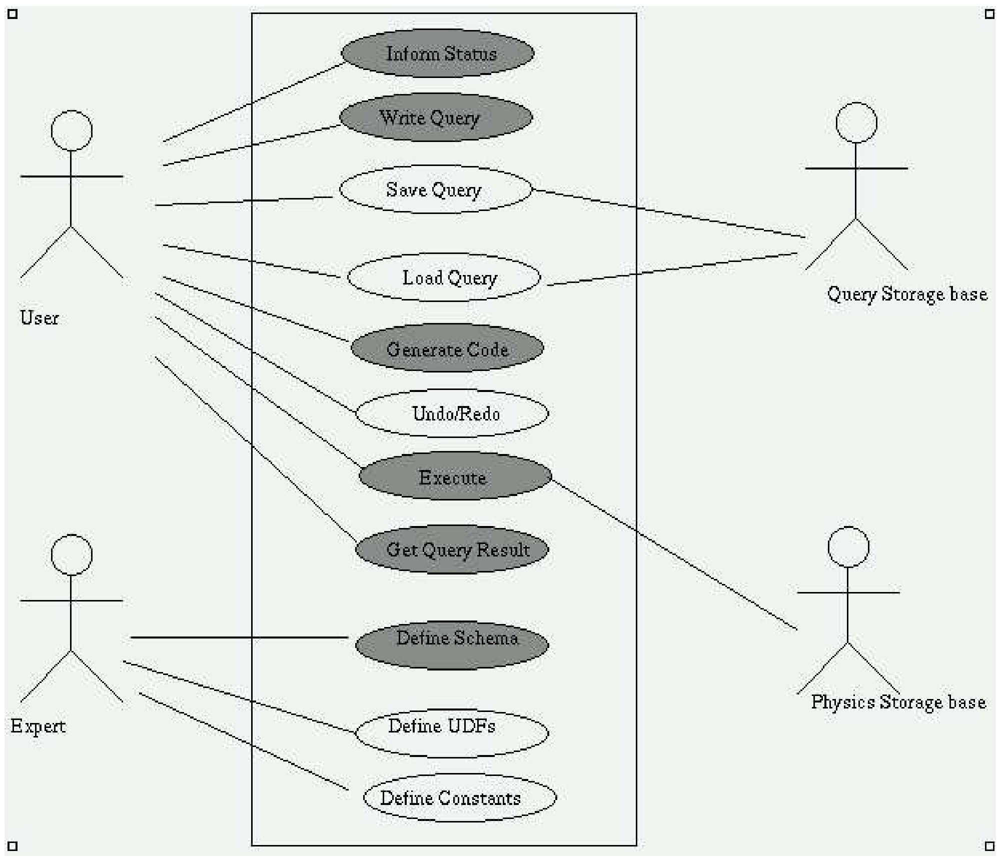


Figure 9.2: Use cases - the use cases in dark grey are covered by the prototype implementation.

is strictly bound to the specific target framework. The query code generator is implemented as a plug-in to our PHEASANT framework specific to BEE (see Appendix B). Other plug-ins can be added to deal with different target physics frameworks without necessarily imposing changes to the rest of the query generation modules. The Code Generator module is described in Section 9.4.

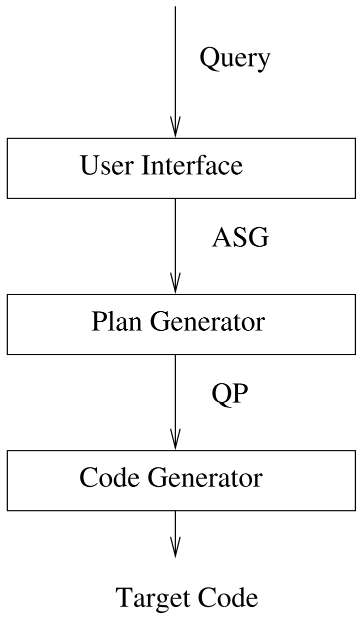


Figure 9.3: General structure

9.1.3 Technology Used for the Implementation

In order to test our hypothesis, we delivered a first prototype implemented in the TCL/TK scripting language on a Unix platform. We use extra packages for dealing with graphs, trees and visual widgets. The advantage of a scripting language like TCL/TK over other structured languages is the fast implementation due to the simplicity of the language itself and their visual manipulation packages. It allows also easy portability, which implies that we can use the same product on other platforms. In our case, the only changes will take place at the code generator plug-ins. Nevertheless, the unstructured nature of the language makes it difficult to produce elegant and clean code. The larger it gets, the more difficult it is to maintain. Therefore, we suggest that the next phase of the engineering life cycle delivers a product using a structured language like C++ or Java.

9.2 User Interface - The Visual Editor

This section explains the architecture of the visual editor, which provides the user with an environment for editing visual queries. This software deals with the concrete components and the concrete syntax graph, delivering the corresponding ASG into the next architectural layers.

First, we justify the general design decisions taken, based on related work done in Human-Computer Interface (HCI) and Visual Languages (VL). Then we describe the different modules required for our solution: Graphical User Interface (GUI) and Abstract Syntax Graph (ASG) generator. We conclude this section by discussing some proposals for future work.

9.2.1 Related Work and Design Decisions

A good introduction to the implementation of Visual Language editors can be found in [44]. There are several different ways to implement them. One option are so-called free hand editors, in which users can draw whatever they want on a virtual canvas and produce a graphical bitmap as output. This entails image processing and pattern recognition to understand the programmer's intentions by identifying the graphical objects and their relationships. (With text scanned from paper, the problem would be the same.) This editor is the most flexible solution, but, on the other hand, it is very difficult to interpret the input of the user.

As HEP analysis implies a clear symbology and query pattern, we do not allow this flexibility, but provide a predefined set of graphical objects that can be used. Instead of allowing the specification of text,lines, rectangles, or circles which are then scanned and parsed into graphical objects, we go further than that and we allow the user to specify already the graphical objects/symbols of the query language. The meaning of the pictorial elements is either already known or easy to learn, so we apply an incremental parsing approach where the query elements are easily parsed while being built. An internal spatial relation graph is generated.

At this stage, editors can be classified as:

- Syntax-free - They are merely used to enter visual queries, without any syntax concerned.

- Syntax-directed - They only allow the user to enter syntactically correct queries.

- Syntax-assisted - They prompt the users to write syntactically correct queries.

Syntax-free editors were not under consideration because we need to syntactically validate the query construction. We could leave the responsibility to a syntax checker module that feeds the syntax problems back to the editor after the query construction. But the problem of this approach is that because of all the syntax errors returned back, the user has to memorize the syntax if he wants to have a less time-consuming query production phase. We want the user to realize when he is producing an invalid query sentence.

Having declined the first option, we are left either with a syntax-directed or a syntax-assisted editor.

Syntax-directed editors enforce correct user inputs. Inputs that conflict with the given syntax are immediately rejected. These editors are fine for situations where we do not have to deal with intermediate syntactic states. It is much more complex to implement these editors if intermediate states occur that are syntactically incorrect but potentially correct. If, for instance, the Transformer operator in PHEASANT is left alone, we can say that the question is incorrect. However, it is potentially correct because we can connect this operator to the selection operators and the result particle operator. In our opinion, these editors are not the right approach for PHEASANT, as users are going to have the same problems as with the existing textual interfaces, only on a graphical level. As a consequence, we have decided to implement a syntax-assisted editor, although knowing that it is harder to develop than a syntax-directed one, as it is more interactive. This means that it helps the user to arrive at a correctly formed syntax by giving hints. Incorrect constellations of objects are not rejected outright but are highlighted to indicate problems with the query. Fig.9.6 shows the query from Fig.8.1 in the layout of our editor. Note the pop-up menu in the lower right corner of the figure. This is used to input the before-mentioned attribute and condition lists.

When dealing with languages that imply the use of the keyboard for the input of text (for labeling, the specification of strings, or mathematical equations) and using the mouse to draw the graphics, this implies that the programmer is constantly distracted. Together with that, the repeated movement of the mouse pointer between the “palette” to select the graphical elements and the virtual canvas can cause user complaints. Because of these two reasons, we implement pop-up menus on the drawing site.

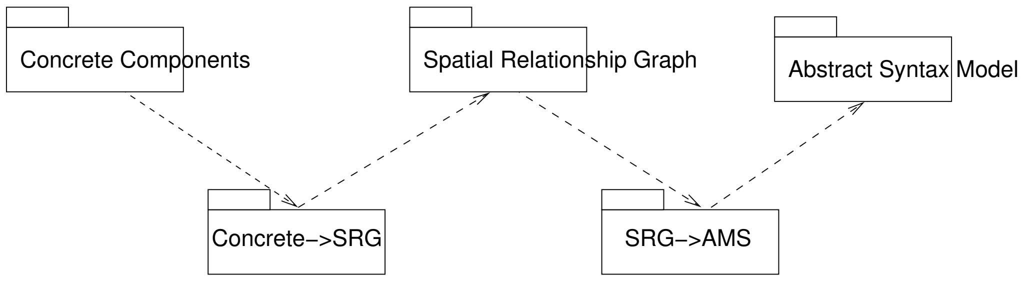


Figure 9.4: Transformation from CSG to ASG

We must identify two levels of syntax: Concrete and Abstract Syntax.

The concrete syntax must include every detail concerning visual aspects of the language, whereas the abstract syntax can safely ignore all aspects that are not needed within the semantics definition. This means that the abstract syntax abstracts concrete symbols and geometric details, like size and position of objects.

In parallel to the textual language parsing, the visual language has a sequence of steps that involves a three-stage process:

- **Scanning or lexical analysis** - Some intermediate data structure is necessary to represent the pictorial structure of the diagram. The physical layout is then scanned to produce a spatial relationships graph (SRG), indicating the components of the diagram and their relationships. It contains all graphical objects, but instead of containing all individual properties, it represents the higher-level spatial relations which hold between its objects.

- **Parsing or syntax analysis** - The SRG is mapped to an abstract syntax graph (ASG) to reflect the internal (logical) structure of the diagram according to its visual language. Nodes and edges in this graph should correspond to language constructs, but do not determine what these constructs look like.

- **Generation or semantic analysis** - This phase implies the interpretation of the logical structures according to the rules for semantic description of the language and the corresponding generation of a target code and/or error report. We will concentrate on this topic in the following sections.

9.2.2 The Architecture of the Visual Editor

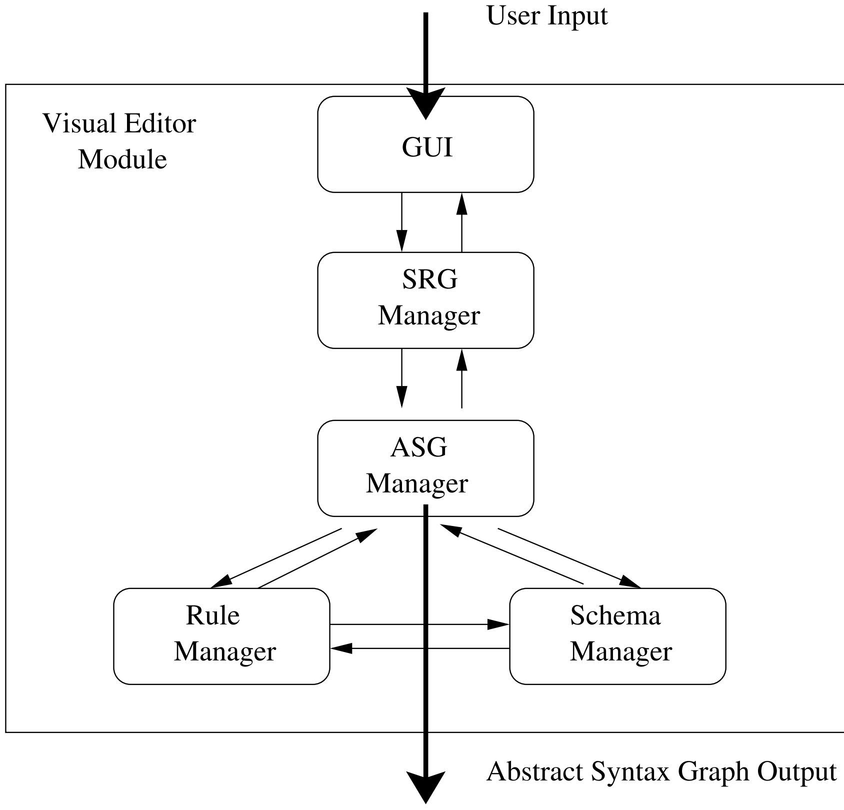


Figure 9.5: Components of the Visual Editor

The Visual Editor of our implemented prototype consists of five main components: GUI, Spatial Relationship Graph (SRG) manager, Abstract Syntax Graph manager, Rule manager and Schema manager.

GUI

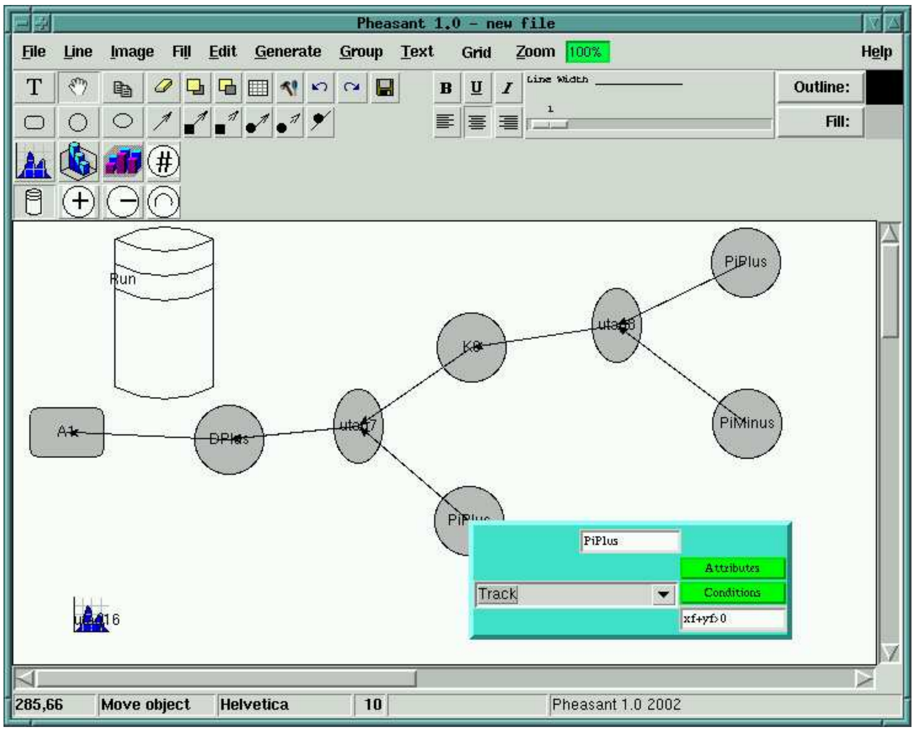

Pheasant 1.0 - new file
File Line Image Fill Edit Generate Group Text Grid Zoom 100% Help
B U I Line Width 1 Outline: Fill:
Run
PiPlus
K0
PiMinus
A1
DPlus
PiPlus
Track
Attributes
Conditions
xf+yf>0
285,66 Move object Helvetica 10 Pheasant 1.0 2002

Figure 9.6: PHEASANT query layout

GUI is a typical vector graphics editor, as it can be seen on Fig.9.6. It notifies the SRG manager every time a concrete component is inserted or modified in order to update both the SRG and the ASG. These packages verify the corresponding syntax. In order to provide feedback to the user, they return the status through the message window or by using colors to indicate the incomplete state of the query sentence drawn so far.

Some of the visual elements have a second level of detail through the use of pop-up menus. With them, the user is able to specify condition predicates or extend the list of attributes represented by the element, or even select the inherent type (for example Particle or Vertex).

The layout is composed by a typical menu bar with pull-down options; a toolbar, where it is possible to select the several query components; a canvas, where the query is edited; and a status bar with a corresponding message box.

SRG manager

Spatial Relationship Graph manager is a graph of geometric objects like described in Fig.9.7. It simply deals with insertions, updates and removals from the graph, leaving all the syntax interpretation for the ASG manager to which it communicates the changes.

ASG manager

This is a graph manager that deals with the syntax graph generation derived from the graph grammar described in Chapter 8. This manager is responsible for calling the rule manager and the schema manager to verify the syntactical validity of the sentence.

It is possible to implement a type-safe or a non-type-safe data model. Our framework was designed to support the first one. This way, it is able to check and reject queries that will generate run-time errors due to type inconsistencies.

Rule manager

Although very primitive in our first prototype implementation, this manager collects a set of grammar rules (based on our graph grammar specified in Chapter 8) to deal syntactically with the components. This manager is also fed by a script during the initialization phase, that provides a list of arithmetic constants and user defined functions (UDFs) that are considered to be valid.

In order to check the well-formedness of the text strings where the user describes the query predicates (or conditions attributes) and the new attributes description for each visual component, a special parser was implemented. It has to deal with arithmetic inequality expressions that understand the constants and user-defined functions.

Schema manager

To support the slight variations in the data schema of each different experiment framework, this module accepts the schema description (in the

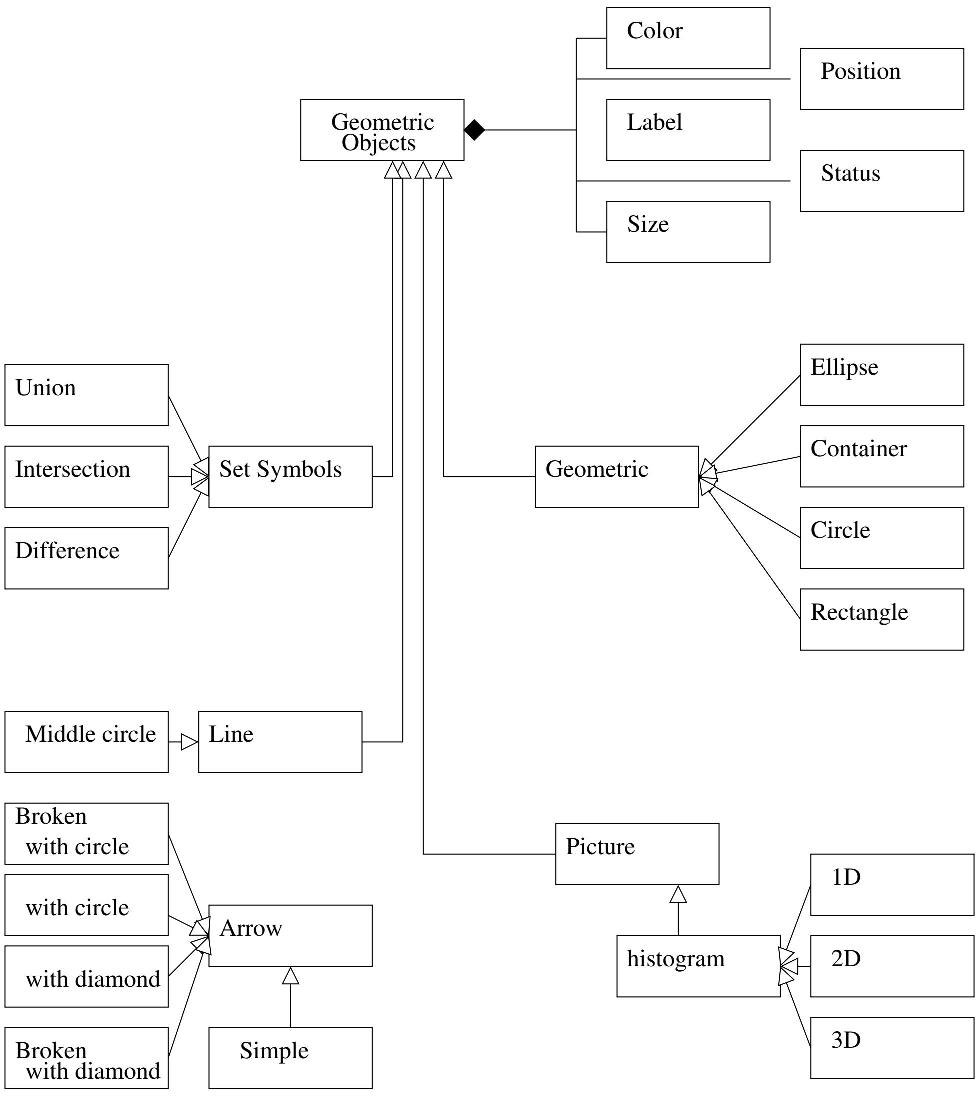


Figure 9.7: Meta-description of the concrete symbols

shape of a graph). The schema objects must obey to the base structure presented in Fig.9.1.

$\# \ InsertingNodes\#$

$Node: \ Run \ X \ AttList \ X \ Type$
$AttList = < \{id \ BIGINT\} \ \{flag \ BIGINT\}... >$
$Type = Container$

$Node: \ Event \ X \ AttList \ X \ Type$
$AttList = < \{id \ BIGINT\} \ \{exp \ SMALLINT\} \ \{flagBIGINT\} \cdots >$
$Type = Event$

$Node: \ Particle \ X \ Attlist \ X \ Type$
$AttList = < \{xf \ DOUBLE\} \ \{yf \ DOUBLE\} \cdots >$
$Type = Physics \ Objects$
$\ldots$

$\#InsertingArcs$

$Arc: \ Run \ contains \ Event$
$Arc: \ Event \ contains \ Particle$
$Arc: \ Event \ contains \ Vertex$
$Arc: \ Particle \ refers \ Vertex \ using \ vertex$
$Arc: \ Vertex \ refers \ Particle \ using \ ingoing$
$Arc: \ Vertex \ refers \ Particle \ using \ outgoing$
$\ldots$

Figure 9.8: Specifying the schema in PHEASANT

9.2.3 Future Work

This language editor meets many of the requirements for effective graphical user interfaces. Human Computer Interaction here takes the main role in increasing the usability. Our first prototype can and should be improved on both the output of the visual language and data visualization by taking what this research area has to offer.

For instance, it is very well known that providing copying protocols to produce duplicate fragments like cut-&-paste mechanisms reduces the time spent to reproduce similar query fragments. A typical example would be when the user specifies the list of predicates of a selection object and does not want to input them again in another similar selection object part of the decay being specified. Another interesting idea would be to support a list of template queries, where the user would get a skeleton of a query that he could fill in and/or expand.

9.3 The Generation of a Logical Query Plan

In this section, we describe how we have implemented the transformation of the Abstract Syntax Graph (ASG) into a valid logical query plan. The ASG provided by the User Interface tool will be first translated using an intermediate step into an Abstract Syntax Tree (AST), and then into a logical query plan.

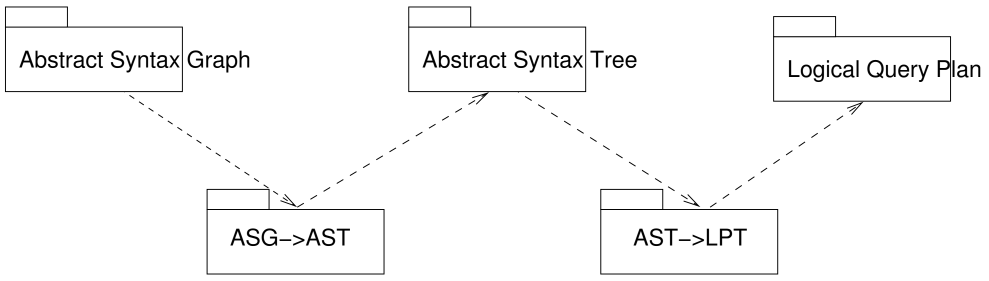


Figure 9.9: Generation of a logical query plan

The process of generating a Logical Plan Tree (LPT) from the query represented in the ASG internal structure derived from the GUI application, described in the previous chapter, is split into two major steps. In the first one, the possibly cyclic graph is transformed into a tree, which is easier to deal with. In fact, the algorithms to interpret the graph get very complex and error-prone. In contrast, handling trees is much simpler. Our second step deals with the translation of the tree into the corresponding algebraic operators, as predicted in the semantics description of our language PHEASANT QL.

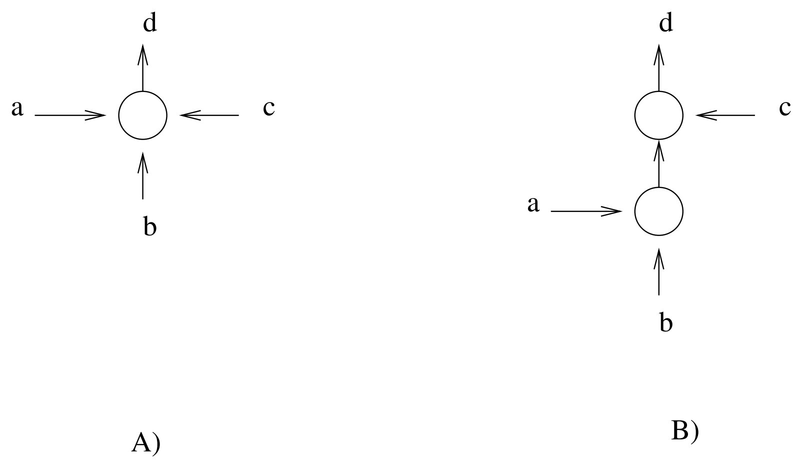
9.3.1 AST Generator

We now introduce intermediate transformation steps applied directly to the Abstract Syntax Graph (ASG). At the moment, the ASG has the form of a DAG. After re-writing it, we will have translated it into an Abstract Syntax Tree (AST).

First, we present our first naive approach, as described in [9], presenting its drawbacks, then we present an improved approach.

Rewriting the Visual Query with the Naive approach

The intersection and union operators - At the first step of the query - the collection selection, when more than two containers are linked by the intersection operator - it is necessary to unfold the graph structure in order to be able to run the semantic rules for intersection that just deal with two operators. The rule is visually explained in Fig.9.10.



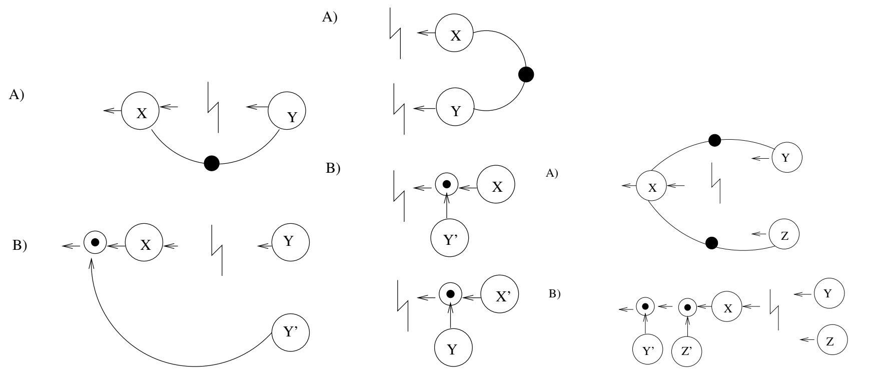
Unfolding from A) to B) the QCollection graph with intersections or unions (represented by a circle) with more then two operators.

Figure 9.10: Unfolding the QCollection

Comparison operators - As described before, the Comparison is a binary operator that is linked with non-directed edges to Selection operators. This operator necessarily closes the DAG, which is more easy to deal with if we break it up into a tree.

While dealing with this operator, the break-up rule depends on whether both selection objects have to be dependent on each other or not. In the first case, we operate the decomposition by using the transformation rules depicted in Fig.9.11 on the left and middle side. If they are dependent, we use the transformation depicted on the right side.

As stated by the rule on the left of Fig.9.11, the closed DAG is broken into a tree-like structure by making a copy Y' of the Y selection object and connecting it as input to the Comparison operator.



in the three situations A) is transformed into B):

Left side: Rewrite rule of the Compare operator with objects part of the same decay tree.

Middle: Rule for rewriting the compare operator between selection objects of different decay chain trees.

Right side: describes the rewriting rule for the compare operator when a *"collision"* occurs.

Figure 9.11: Naive rewriting of the comparison operators

When comparison operations are used between selection objects that are not part of the same direct decay tree (have non-dependency), we copy both selection objects, and two operator nodes are inserted accordingly, as it is depicted in Fig.9.11 (middle transformation).

In case of *"collision"*, which means that the same selection object is being used by two or more different comparison operators, it should be decomposed one by one (see Fig.9.11, right transformation). The criteria for the order of nesting is set in the implementation phase.

Sample rewriting - A complete example of the rules defined in this section can be found in Figure 9.12, which is a rewrite of our first example in Figure 8.1 and also uses an object reference from the particle Pi^{+} to its Monte Carlo counterpart.

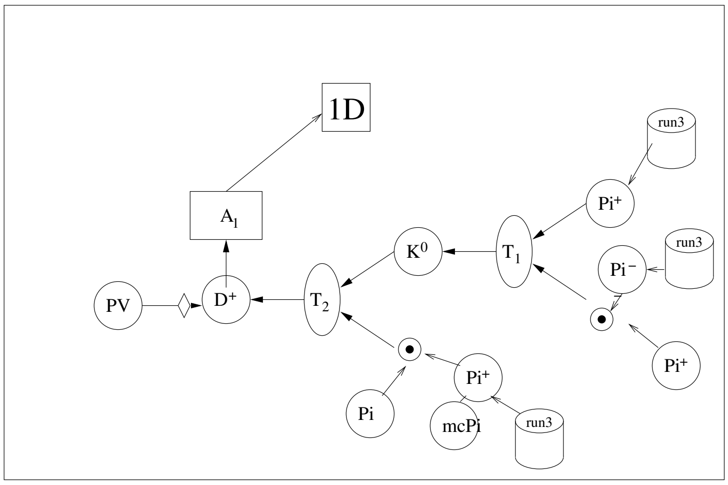


Figure 9.12: The D^+ decay example rewritten with a naive approach

Rewriting the Visual Query with a Non-defactorization approach

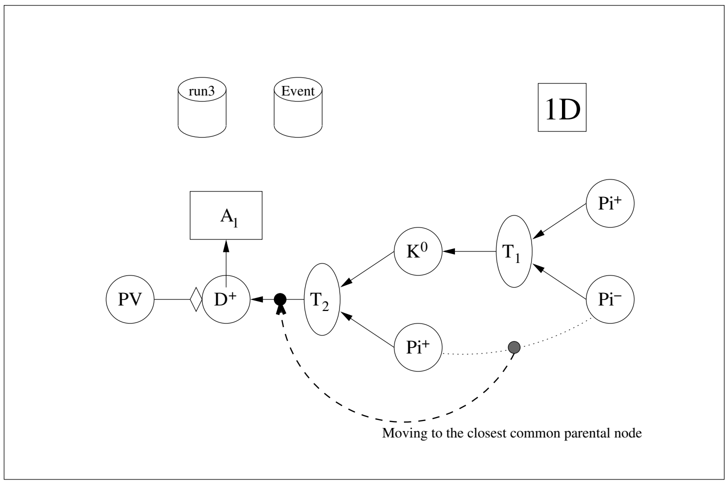


Figure 9.13: Rewriting the graph into a tree by restructuring the comparison nodes

The naive splitting approach is satisfactory for the collection queries, but as far as the decay description queries are concerned, this approach has the problem of causing an exponential query plan explosion. The de-factorization produces redundancy in the semantics mapping, which is more difficult to deal with for optimization purposes. The resulting query plan, after the translation of the semantic translational rules, would be extremely inefficient. The natural visual factorization of the user is being wasted this way.

In accordance with the semantics description of the language in the last chapter, we can observe that comparison operators represent purely a selection predicate. Abstracting the syntax as proposed by our BNF like grammar, although being designed for formal purposes, still helps us in the implementation phase.

In fact, if we remove these comparison operators from their place and add their predicate to a conjunctive list to be applied to the root node, the meaning will keep the same. This is another form of de-factorization. The price to pay in this case is that when translating the semantics we will get a query execution plan with late selections. As a result, longer object streams are used in main memory, data are kept unnecessarily data, and computational resources are wasted with unnecessary data manipulations.

In consequence of this line of thought, we can do even better than that. We want to keep the factorization. If we set a rule that moves the predicate to the closest parental connection node in the tree, it is more efficient since the comparison always implies a join of two branches in the query tree (see Fig.9.13).

9.3.2 Logical Query Plan Generator

Based on the description of our mapping generator, we were able to derive a recursive algorithm that reads the query tree.

This algorithm descends from the query tree root and walks down its branches until it reaches the leaves and transforms the syntax tree into a corresponding algebraic query plan.

The following pseudo-code roughly represents the algorithm for mapping the decay tree (for simplicity reasons, we omit the part that translates the collections):

MapDecay
Input: Node in the AST as *node*
Output: Query plan sub-Tree

```
switch $node$ :

 case $\bigcirc$
    if $(deriveType(\bigcirc) = \longrightarrow\!\circ)$
        $childs = getChilds(\longrightarrow\!\circ)$
        $tree = \{\}$
        foreach $child$ in $childs$
            $mapchild = MapDecay(child)$
            if $(tree = \{\})$ $tree = mapchild$
            else $tree = setTree(\bowtie, \{tree, mapchild\})$
        $setTree(\Delta, childs)$
        return $setTree(\Delta, tree)$

    if $(deriveType(\bigcirc) = \bullet \longrightarrow)$
        $childs = getChilds(\longrightarrow\!\circ)$
        if $(mandatory(deriveType(\bigcirc)) =$ True$)$
            $tree = setTree(\Delta, setTree(\mu, QEvent))$
            foreach $child$ in $childs$
              $tree = setTree(\mu, tree)$
        else
            $tree = setTree(\Delta, setTree(=\mu, QEvent))$
            foreach $child$ in $childs$
              $tree = setTree(=\mu, tree)$
        return $tree$

    if $(deriveType(\bigcirc) = \diamond \longrightarrow)$
        if $(mandatory(deriveType(\bigcirc)) = True)$
            return $setTree(\Gamma,$
               $setTree(\bowtie, MapDecay(\bigcirc), MapDecay(getChild(\bigcirc))))$
        else
            return $setTree(\Gamma,$
               $setTree(⟕, MapDecay(\bigcirc), MapDecay(getChild(\bigcirc))))$

    else if $(deriveType(\bigcirc) = \{\})$
        return $setTree(\boldsymbol{\Delta}, setTree(\mu, QEvent))$
```

MapDecay (cont.)

```
case $ROOT$
  $childs = getChilds(ROOT)$
  $tree = \{\}$
  foreach $child$ in $childs$
    $mapchild = MapDecay(child)$
    if $(tree = \{\})$ $tree = mapchild$
    else $tree = setTree(\bowtie, \{tree, mapchild\})$
  $setTree(\Delta, childs)$
  return $setTree(\Delta, tree)$

case $\square$
   return $setTree(\mathbf{\Delta}, setTree(\mathbf{\Gamma}), MapDecay(getChild(node))$

case $\odot$
   $parent = NewNode(\sigma)$
   $child = MapDecay(getChild(node))$
   return $setTree(parent, children)$
```

As said before, this approach was used for a prototype implementation. Therefore, we believe there is still plenty of room for improvement in this phase. Deriving more elegant or improved algorithms is the possible evolution of this work.

9.4 Code Generation

In this section, we briefly describe how we have implemented the generation of code starting with a logical query plan and then optimizing it and generating the physical query plan that is finally mapped into the target code.

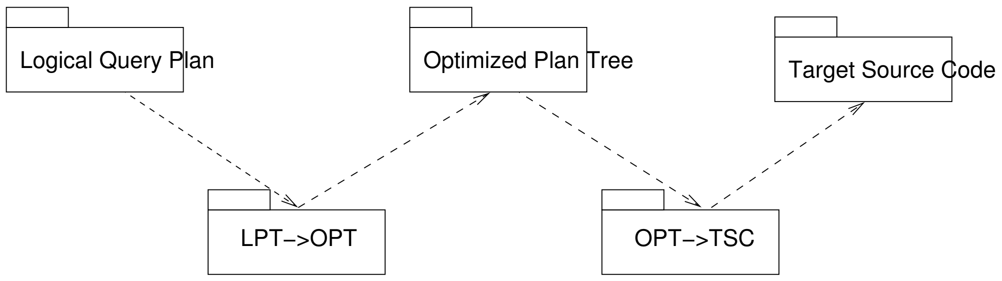


Figure 9.14: Source code generation

9.4.1 Query Plan Optimization

In this phase, the initial logical query plan tree is re-written into an equivalent, but more efficient expression using transformation rules. This means that it will be optimized into a new algebraic query plan and then mapped into a corresponding physical execution plan that uses physical operators. We have left out the logical optimization from our prototype, since this is an active research area that is beyond the scope of our thesis. It should, nevertheless, be considered in the next phase of the implementation. A good introduction to the topic can be found in [59].

Usually, in other database engines, to generate the physical query plan, there should be an optimizer to decide on the mapping: which selection method to use, which join method, and where to materialize or pipeline. In our case we have skipped the optimization and have mapped the query plan tree directly to physical operators. This gives room for future improvements that will have strong impact on the performance (but are not relevant for the purposes of this thesis).

9.4.2 Target Code Generation

For each of the algebraic operators we have built a corresponding physical operator, but in the future we need to add more operators to broaden the possibilities for physical optimization. Furthermore, as a typical characteristic of a domain language, rather than interpreting our query plan, we compile the plans into an executable code.

In order to implement this phase, we have been inspired by the elegant approach of Fegaras in the LDB database system [51, 49]. He makes use of a stream-based execution engine (also called pipelining or iterator-based processing) in a purely functional fashion. As described in his paper, this technique borrows concepts from the area of lazy functional languages avoiding to use threads to implement the pipelining.

The concept is simple to grasp. A Stream is an entity that contains a stream of tuples. It operates with the main services: Open, Close and Next. It can be of the type materialized, when stored into second storage, or suspended, while kept in memory.

The operator algorithms return a tuple as soon as it is constructed. In order to retrieve all the tuples, the algorithm is called several times.

The pipelining is then guaranteed by a structure of an embedded function, which is named suspended stream, that for each time it is invoked without arguments, it calls the algorithm to construct the tuple. The following pseudo-code, generated when running a map algorithm that walks down the physical plan tree, makes this strategy evident:

Example of query code

```
##############
#  Query Predicates #
##############
 Bool pred1(tuple t)
     {  if( Some defined predicate )
            return new bool(true);
        else
        return new bool(false); }

 Bool pred2(tuple t)
    {...}

 Bool pred3(tuple t1, tuple t2)
    {...}

##############
#  Query Functions  #
##############
 tuple function1()
 {return table_scan(((Stream*) stream[0]), &pred; }

 tuple function2()
 {return table_scan(((Stream*) stream[1]), &pred2); }

 tuple function3()
 {return Nested_Loop((*Stream) query[1], (*Stream) query[2], pred3); }

##############
#   Query Streams  #
##############
 query[1] = (void*)suspended_stream(&function1);
 query[2] = (void*)suspended_stream(&function2);
 query[3] = (void*)suspended_stream(&function3);
```

A complete description of the library of physical operator algorithms that we have implemented based on Fegaras' work is described in Appendix B. The signature of these algorithms is defined in Fig.9.15. As we can see there, the implementations are a simplification of the algebraic operators. In fact, the Nest algorithm is prepared to group-by the tuples by event identifiers and they already include the evaluation of the Reduction operator usually associated with it in our query pattern (as described in Chapter 8).

$table - scan: \ Stream \ X \ Predicate \longrightarrow Tuple$

$Union: \ Stream \ X \ Stream \longrightarrow Tuple$
$Intersection: \ Stream \ X \ Stream \longrightarrow Tuple$
$Difference: \ Stream \ X \ Stream \longrightarrow Tuple$

$Unnest: \ Stream \ X \ Path \ XPredicateXOuter \longrightarrow Tuple$
$Nested\_Loop: \ Stream \ X \ Stream \ X \ Predicate \ X \ Outer \longrightarrow Tuple$
$Nest: \ Stream \ X \ Head \ XAggregateFunction \longrightarrow Tuple$

Figure 9.15: Physical operators: Signature of the Table-scan

Interfacing with the Storage Engine

The experiment's framework developers are responsible for the design of the storage engine. The function of this layer is to deal with persistent data and their transfer between main and secondary memory. We will abstract from the way HEP storage engines do buffer management and how they deal with some indexing to retrieve the data.

In order to be able to couple our operator's Streams with the BEE (see Appendix B) storage engine, we have designed a "quick and dirty" solution which is to use an API on top of BEE and maintain an Event buffer with the FIFO (First In First Out) rule. The idea is then to make use of a hybrid solution of similar concepts to Eager and Lazy pointer swizzling

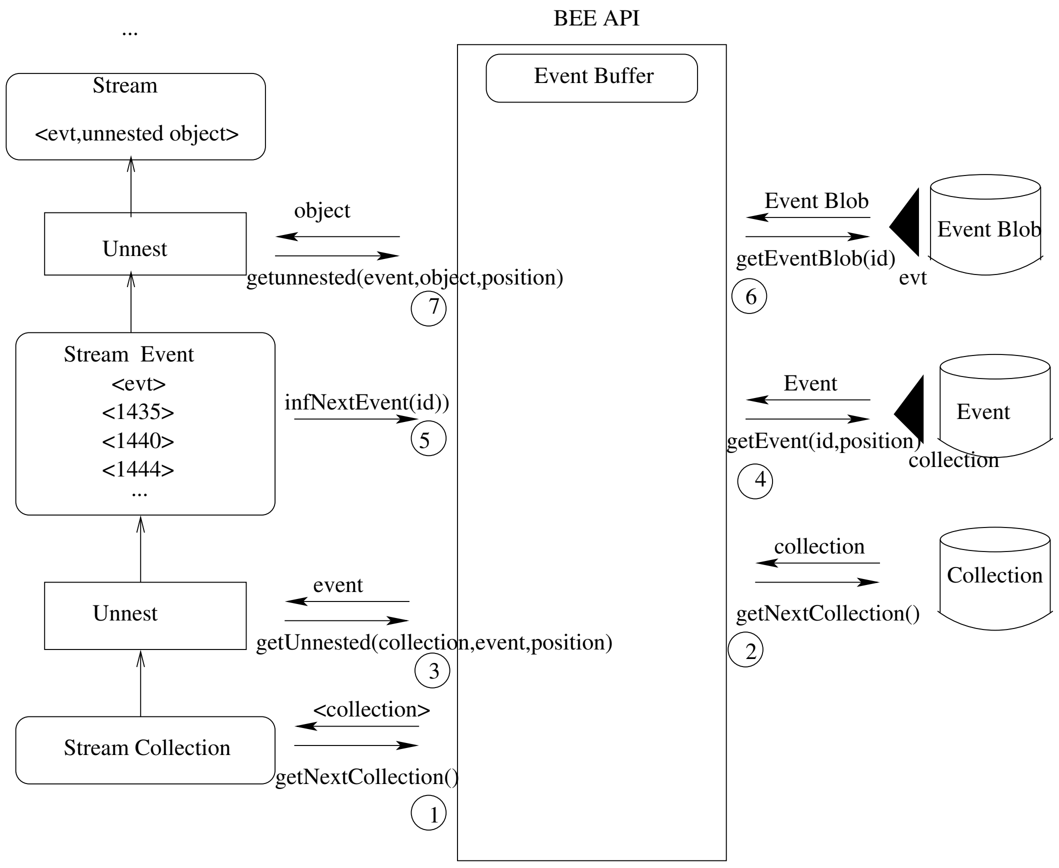


Figure 9.16: Interacting with the storage engine

[1]. This means that before unnesting an object contained by an event for the first time, the stream evokes a method to load the specific event object from disk into main memory. All the persistent pointers this object contains in the shape of OIDs (referencing particles, vertexes, and others) are transformed (swizzled) immediately into main memory pointers. At the same time, the referenced objects are all copied into the Event buffer. The way data is clustered makes this phase very easy, since all the related objects of the event are usually stored together as a blob not only in BEE, but also in several other analysis frameworks.

The rest of the data, like the Collections or the Events, are dealt with by lazy swizzling, meaning that no pointer is swizzled unnecessarily, only upon request.

In order to better understand our API's function, let us follow an example like the one described in Fig.9.16. It starts with the stream of collection objects that is called by the unnest operator. This activates the method getNextCollection() in our API(1), which, in turn, calls the getNextCollection (2) method on the BEE side. This is going to retrieve from the class extent collection (materialized in second storage) the next collection object which is then returned to the stream.

The unnest operator (3) will now request (getUnnested) for the next Event object contained in the currently selected collection object. This is achieved by giving the collection ID and the position in the set of Events. The API will retrieve (4) the event from the Event class extent from the BEE framework (which is indexed by the collection ID to which this Event belongs).

The stream of Events will inform the API (5) that the Event is requested every time its method Next is called. This way, the API verifies whether it already exists in the event buffer. If this is not the case, the API calls the getEventBlob (6) that requests the corresponding Event blob from the BEE framework. It should be stored on secondary storage and possibly indexed by the event ID (if the framework developers designed it correspondingly). The objects are eagerly swizzled and are ready in memory to be accessed by an unnest operator in (7) or by the rest of the pipelined branches of the query plan.

9.5 Summary

In this chapter, we focused on giving a top-down view of the several architectural modules required for our prototype framework.

- **Visual Editor -** Deals with the interface to the user and generates an ASG as the input to the following module. HCI considerations were taken into account while designing the first PHEASANT prototype. We have presented our reasoning and decisions that took previous research in this area into account. At this level, optimization can be reached by exploring user interface techniques with a strong feedback from the users. It is an evolving process, like any HCI software engineering project. Therefore, we suggest more iterations in the software engineering life-cycle for further developments in this area.

- **Plan Generator -** responsible for transforming the Abstract Syntax Graph into an execution plan. We have presented several algorithms as solutions. First, we have introduced a naive approach, and then an improved version. As future work, we expect to derive better algorithms.

- **Code Generator -** We detailed the transformation of the query plan into a valid source code that can be compiled and run against the physics storage base.

Part IV

Evaluation of the Research

Chapter 10

Evaluation

In previous chapters, we have detailed the new methodology proposed by this thesis for improving the user's performance at the analysis phase. It is necessary to evaluate the usability of the proposed DSVQL. In this chapter, we are going to describe how we have structured our assessment to do so.

To support our claims that with our methods we manage to improve the efficiency, reduce the error rate and have a steep learning curve, we have to perform a complete and unbiased evaluation of our language, comparing it to a real-life programming analysis framework. During the development of our prototype, two frameworks developed by the Hera-B collaboration were considered: ARTE [4] and BEE [55]. The first option had multiple portability problems and lacked technical support like a documentation, which forced us to adopt the second one. BEE makes use of C++ as a query language.

Next, we present how we have systematized the evaluation process to provide quantitative and qualitative information on the usability of our language.

In section 10.1, we discuss the concepts and the related work on interface evaluation that has deeply influenced ours. In section 10.2, we present the formal definition of usability according to ISO 9241-11. Section 10.3 is dedicated to detail the tasks we have programmed to lead the experiment to its end. In session 10.4, we present our interpretation of the results obtained in the assessment and, finally, our conclusions.

10.1 Related Work

An interesting work on human factors in the evaluation of query languages can be found in [82]. A survey in recent visual query experiences is contained in [24]. A complete evaluation of a comparison between a visual query language named Kaleidoquery and OQL can be found in [74]. We have made use of these papers as major guidelines to our experiment.

10.1.1 Human Factors

Together with physical and perceptual activities, visual query languages involve cognitive activities like learning, understanding and remembering. Experimenters in human factors have developed a list of tasks to capture particular aspects. In [82], the authors propose the following list:

- Query writing - users are given a question stated in natural language and have to write a query in the given query language.
- Query reading - users are given a query written in the query language and asked to write a translation into a natural language.
- Query interpretation - users are given a query in the query language and a printed database with the data filled in. They are asked to find the result of the query.
- Question comprehension - users are given a question in a natural language and a printed database and are asked to find the data asked for.
- Memorization - users are asked to memorize and reproduce a database.
- Problem solving - users are given a problem and a database and are asked to generate questions in English that would solve the problem. The questions should be answerable with the database.

To evaluate these tasks, we can use different kinds of tests:

- Final exams - Test how easily a query language can be learned. These exams take place at the end of teaching the language under evaluation.

- Immediate comprehension - Help identify why particular learning problems occur. They are given during teaching, immediately after some function has been taught, to determine whether the participants can use the function, given that they know it is the one to use.

- Reviews - Help identify why particular learning problems occur. They are given during teaching and cover functions taught up to that time. The participants are required to know which function to use.

- Productivity - Tests of the query language use by "skilled" users. They test how well the language can be used after some predetermined level of learning has been attained.

- Retention - Tests how easy a query language is to remember: how well it can be used by people who have been away from it for a period of time.

- Re-learning - Tests how easy a query language is to relearn by users who have been away from it for a period of time and have forgotten some of it.

Testing different tasks in the language usage is interesting, but to perform an exhaustive evaluation of them would be very expensive. Therefore, we concentrate on the critical activities. In the case of Pheasant's evaluation, we are interested in the task of query writing and problem solving. This is justified by the fact that the main function of our language is to provide the users with a tool that speeds up code generation. We want to know how easily it is to learn and use. Therefore, we will restrict our evaluation to the first three tests.

10.2 A Definition of Usability

As specified by ISO 9241-11[1], usability is:

[1]ISO economic requirements for office work with visual display terminals (VDTs), guidance on usability 1998

"The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use".

Measuring **effectiveness** means to determine the accuracy and completion when performing queries.

Efficiency measurement is related to the level of effectiveness achieved at the expense of various resources, such as mental and physical effort, time, financial cost, etc. Efficiency is more commonly measured in terms of time spent to complete a query.

When measuring **satisfaction in use** it means freedom from inconveniences and positive attitude towards the use of the product. How comfortable does the user feel while using the system?

10.3 The Evaluation

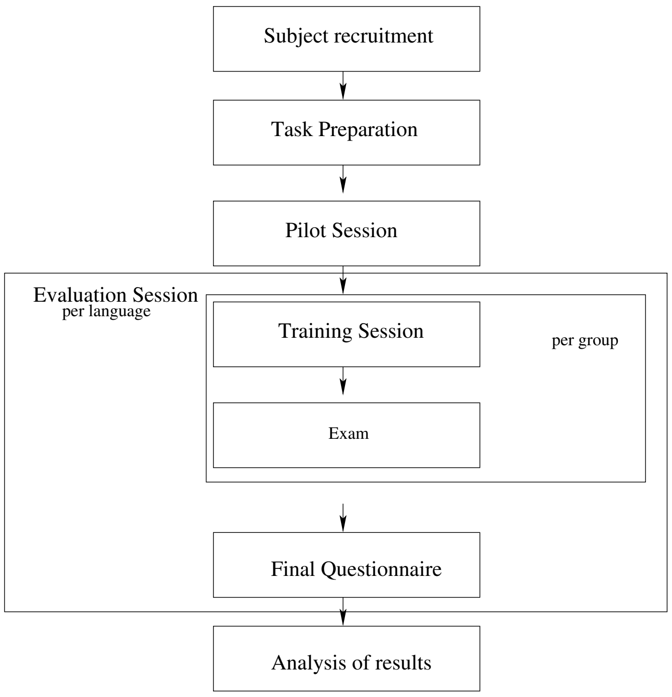


Figure 10.1: The evaluation process steps

As already mentioned, our goal is to assert the usability by evaluating the error rate, user satisfaction, and time to write a query.

The steps of the evaluation process are summarized in Fig.10.1. The whole process starts with the **Participant Recruitment**, where the users are analyzed and grouped into clear categories. This way,the variables concerning the user profile that will lead to different results for different groups are controlled. This step is followed by the **Task Preparation**. The aim here is to organize the evaluation by determining which tasks have to be done and which tests are elaborated in order to provide the proper results. This will generate the information required to be analyzed afterwards. The next step is the **Pilot Session**, which is meant to simulate the exam and test that the material for the training and the evaluation procedures is well organized. The main advantage of this rehearsal is to check that the time constraints and other possible external variables like proper equipment are controlled and do not interfere with the results.

Once everything is tested, we proceed to the assertion, which we called **Evaluation Session**, for each group and language being compared. First, the **Training Session** will introduce a language. At this stage, the Immediate Comprehension and Review tests will already take place while introducing the features of the language.

The final exams of learning, in the **Exam Session**, will involve query writing activities. This session implies observation and recording of the participants' activities like completion times and error rates and a questionnaire. The goal is to determine the easiness of learning.

After each group has been evaluated in the different languages, the participants are asked for a debriefing in the form of a questionnaire. The goal is to obtain the user's perspective of the comparison between the languages.

From [24] we are advised that in order to evaluate unbiasedly, the users should test the same environment and as realistically as possible.

10.3.1 Recruitment of Participants

According to the context of HEP experiments we devise three types of persons involved: informed programmers (Inf-P), uninformed programmers (non-I-P) and non-programmers (non-P). Programmers are those familiar with computers and regular users of programming languages (C, C++, Java or Fortran). This group can be subdivided in informed (if they have already programmed with the present analysis framework) or uninformed (if they have not). Non-programmers are familiar with computers and operating systems, but have little experience in programming languages and have not used any form of physics database interface before[2].

We suggest the usage of two different groups of programmers because the informed ones may introduce a bias on the learning phase of the compared query methodologies. This assumption is taken into account even if informed programmers and non-programmers are the majority of the population in the Hera-B experiment (although this proportion is not necessarily the same in other experiments).

In order to place every participant in the proper group, they were all interviewed and their previous experience was analyzed, avoiding this way the bias of a self-evaluation.

We will now detail the steps taken for the comparison of both Pheasant and BEE/C++ framework.

10.3.2 Task Preparation

Johnson[63] suggests that six individuals per subset of the population is the minimum required for a controlled experiment. Of course it is sensible to take a larger number, but the costs should be kept to a minimum. The task of gathering three groups of six persons in a HEP research lab is already nontrivial. All the participants should have a degree in physics or be near its completion at least, and they should be skilled in experimental analysis. A basic knowledge of programming concepts is mandatory, since this subject is taught in the first years of the physics courses.

Introducing one query system to the whole group of participants and only afterwards the other query system can lead to the situation of knowing the first system to influence the results on evaluating the other. In order to reduce this bias in the results we have to split the group in two. This way, we reduce the influence of the first language while presenting the second. Mixing the three groups might lead to new variables in the

[2]Usually, these are found among students newly introduced to the environment like summer students or students starting their thesis. It is also common to find senior physicists with very little experience in programming languages and the new generation of analysis frameworks. A possible description of the system actors and their role in High Energy Physics can be found in [66].

evaluation that are hard to track. Therefore, we have to organize at least six sessions, with each group taking part in two sessions.

The features we want to have evaluated are:

- Query steps in Pheasant vs the object-oriented coding
- Expressing a decay
- Specification of filtering conditions
- Vertexing and the usage of user-defined functions
- Aggregation
- Path expressions (navigational queries)
- Expressing the result set
- The expressiveness of user-defined functions

10.3.3 Pilot Session

Our evaluation technique was tested with two individuals (two physics experts) in order to verify it and to test the teaching materials and questionnaires. This also helped to avoid (or to reduce the risk) that the evaluation had to be redone from scratch because of unforeseen problems.

10.3.4 Training Session

Obviously, due to the complexity and the time constraints, we cannot teach the complete C++ query language plus the interface of the analysis frameworks' libraries. We have to focus on presenting examples (6 examples), and on the corresponding explanation of the code that represents each of the features to be evaluated. The individuals should try the features by designing a similar query. The last query should make use of all the features taught in the session.

Murray[74] suggests that the participants should give themselves a mark for their feeling of correctness of their trial. This introduces them to the system of auto-marking. Besides, it helps the trainer to infer if there are difficulties experienced and an extra explanation is required. This session should take the time required for each group to understand the six examples.

10.3.5 Evaluation Session

a) Evaluation Queries

In this phase, we evaluate the participants' performance in query writing. Every participant has four queries in English to be rewritten in the previously learned language. The subject makes a self-assessment of his reply rating his feeling of the correctness of the answer. The rates are totally correct (TC), almost correct (AC), totally incorrect (TI), not attempted (NA). The conditions are equal for every individual in the experiment. For each of the queries, we measure the time taken to reply them.

b) Questionnaires

After each session, the participants are asked to judge the intuitiveness, suitability and effectiveness of the query language. The goal is to evaluate:

- Overall reactions - to obtain an overall reaction to one of the query languages through queries.
- Query language constructs - with the participants rating how easily specific aspects of the query language are to use.

After the tests are completed, the participants are asked to compare the two query languages. It is rated which query language they prefer, and to what extent.

- Query language comparisons - the participants are asked to compare specific aspects of both query languages and rate the preferences they have.
- Participants' comments - allows the participants to comment freely on the query language.

Since with the evaluation questionnaire we can only identify problems but not infer how to solve them, we ask the participants to contribute creative comments. Sometimes improvements are obvious and the comments can be fruitful. Therefore, after the evaluation session the participants are asked to write down informal comments and suggestions for improving the language.

10.4 Results

In this section, we summarize the relevant results of our evaluation tests. First, we deal with effectiveness by having a look at the test results with regard to the errors produced by the user while interacting with both evaluated approaches. Then, we will describe the results related to efficiency, which are mainly concerned with time measurements. Finally, we will describe the results concerning the user satisfaction.

Unfortunately, due to the fact that the Hera-B experiment was over before our assertion, we did not manage to gather the expected number of scientists for our assertion (two Non-P, one Non-I-P and two Inf-P). A greater number of subjects would mean a higher certainty on the conclusions and a lower error rate. Nevertheless, it is still a strong indicator.

In order to reduce the variables that could influence the results, the queries were explained orally by an expert. This reduces the required interpretation time (which has a significant impact, especially in the first group, since it is less experienced). Code re-usage was not allowed, although they could use all the necessary documentation and especially the notes from the training session.

10.4.1 Effectiveness - Errors

Analogous to Reisner's [82] proposal, we grade the queries by:

5	Correct
4	Minor data error, will not retrieve the complete result. (e.g some results missing)
3	Minor language error, e.g. misspelling and punctuation
2	Error of substance; valid queries that produce wrong answer
1	Error of form, invalid query
0	Not attempted

As it can be observed in the histograms of Figs.10.2 and in more detail in Fig.10.4, while using C++ as a query language, the error rate was tremendous. We must state that the user did not have any sort of feedback from the system execution in order to spot the mistake and correct it before it came to the hands of the evaluator. In his daily life, the user tries to execute the algorithm and watches the result data after the execution.

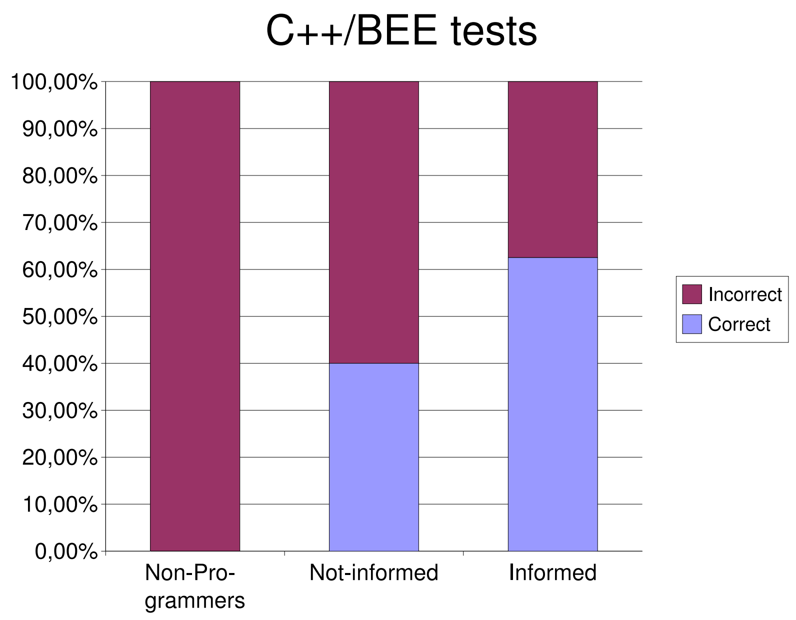


Figure 10.2: Effectiveness C++/BEE: Huge error rate.

Then, in a cyclic way, he corrects himself and runs the query against the storage base. As we have claimed in chapter 4, this is one of the main reasons why the query generation in the physics analysis phase is so time-consuming.

In Fig.10.3 and Fig.10.4, we can also observe that different groups of users get different results. As expected, their quality is directly proportional to the user's experience. Some of the most complex queries were not even tried due to the fact that they were too difficult for users unexperienced in C++, which had just 2 hours of training (obviously not enough).

As far as the Pheasant Query language is concerned, the results are much more promising. As the query mechanisms are much simpler and controlled, we do not observe invalid queries, and only a few wrong answers (which can be explained by some inexperience of the users in doing analysis itself).

Generally, the results show that the user did not have to essentially change the way he thinks about the query generation, which means that

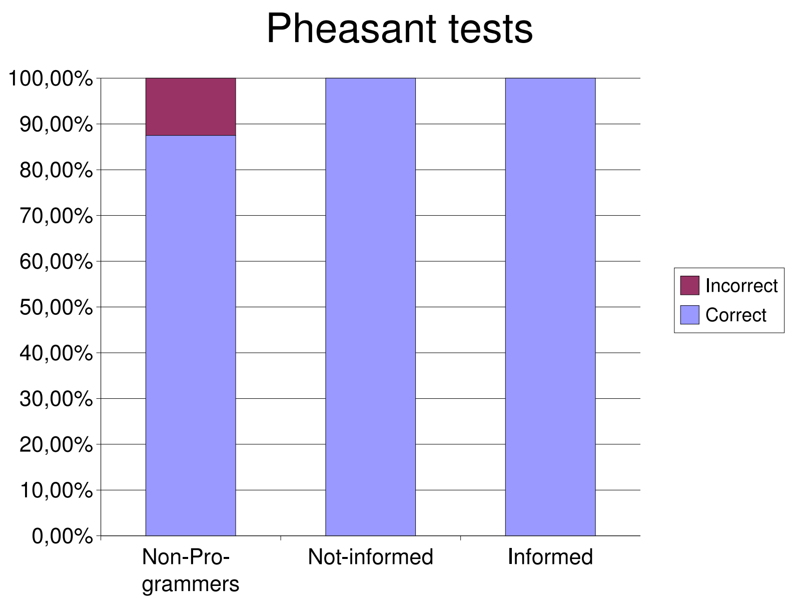


Figure 10.3: Effectiveness Pheasant: Huge correct rate.

we have reached the goal of introducing a query language closer to the physicist's conceptual level of analysis.

BEE/C++	N-P	N-I-P	Inf-P
Correct			
Minor data error			12.5
Minor language error		20	50
Essentially correct	0	20	62.5
Wrong answer	37.5	20	25
Invalid	25	20	12.5
Not attempted	37.5	20	
Totally incorrect	100	60	37.5

Figure 10.4: Error analysis in BEE framework (percent values)

Pheasant	N-P	N-I-P	Inf-P
Correct	87.5	80	87.5
Minor data error			
Minor language error		20	12.5
Essentially correct	87.5	100	100
Wrong answer	12.5		
Invalid			
Not attempted			
Totally incorrect	12.5	0	0

Figure 10.5: Error analysis in Pheasant (percent values)

Pheasant/ BEE	Non-P	Non-I-P	Inf-P	Mean
Structuring the query	5/1	5/1	4/4	4.7/2
Different data schema feature	3.5/1	3/1	3.5/3	3.3/1.7
Expressing a decay	5/1	5/2	4.5/2	4.8/1.7
Expressing filter conditions	5/1	5/2	5/4.5	5/2.5
Expressing and using vertexing	5/1	5/2	5/4	5/2.3
Expressing and using UDFs	4.5/1	3/3	3.5/5	3.7/3
Path expressions	5/3.5	5/2	3/5	4.3/3.5
Expressing the result set	5/1	5/ 2	5/3.5	5/2.2
Mean	4.8/1.3	4.5/1.9	4.2/3.9	

Figure 10.6: Language constructs analysis: Subject evaluation. Scale from 1(worst) to 5(best)

10.4.2 Efficiency - Resulting Times

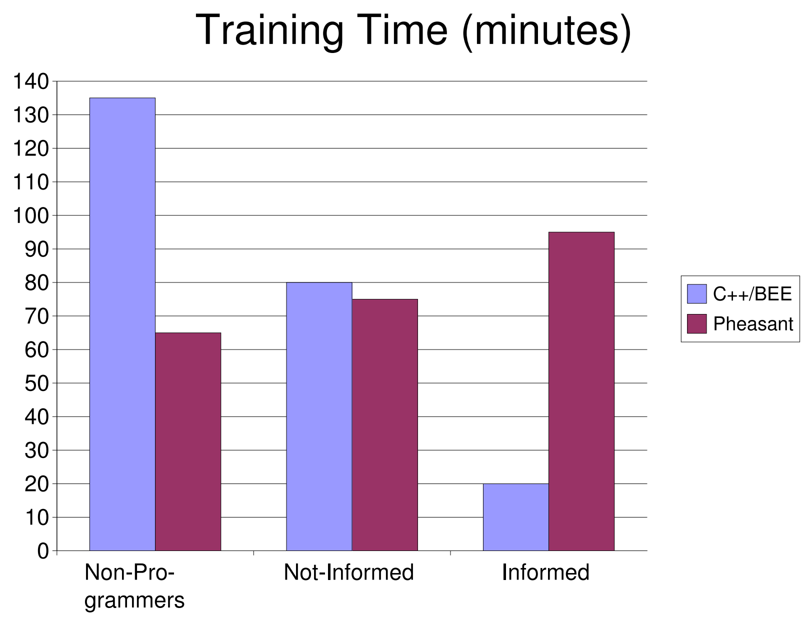


Figure 10.7: Efficiency of C++/BEE vs. Pheasant: Less training time required.

From our time analysis (Figs.10.7, 10.7 and detailed in Fig.10.9), it becomes clear that more time has to be spent learning and using C++ and BEE than with Pheasant. This can be justified by the complexity of C++ and the BEE library. At the same time, the test participant had less confidence in the quality of his/her query (see also Fig.10.10 and Fig.10.9). This subjective impression is confirmed, as we have seen, by the huge error rate when using BEE, Figs.10.4 and 10.5.

In Fig.10.6 we have given an excerpt of a list of features needed in HEP analysis. The test participants were supposed to rate how they were satisfied with the realization of each feature in the corresponding framework. Our goal was to identify potential weaknesses of each framework.

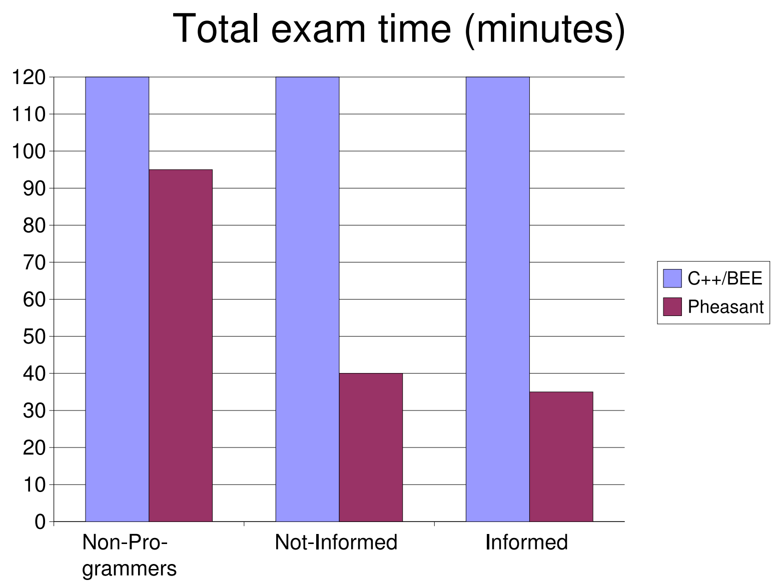


Figure 10.8: Efficiency C++/BEE vs. Pheasant: Much less time to complete the task.

10.4.3 User Satisfaction

The enthusiasm towards the language was significant. The several comments focused more on implementation issues to improve interactivity and did not criticize the language itself. This is a typical situation in user interfaces when dealing with prototypes. It is explained by the fact that the prototype needs to evolve into the next engineering life cycle phase to result in a properly engineered software product. Only this way the product is able to provide a real analysis environment and the user can compare it in his daily life with the other alternative solutions.

Although the system experts (a minority in a typical HEP experiment analysis) recognize that the solution is a more comfortable approach for analysis, they still worry that the query tool might be less expressive. In order to confirm or reduce these fears, we propose to carry out further tests of `Productivity`, as described in 10.2.

Some of the most relevant comments are listed in the following:

- “Pheasant should reuse my previous queries, with C++ I just re-

Non-P	C++ BEE	Pheasant
Training time (hours:minutes)	2:15	1:05
Mean total exam time (hours:minutes)	$> 2 : 00$	1:35
Mean confidence/query (5 very/0 not)	1	3,5

Non-I-P	C++ BEE	Pheasant
Training time (hours:minutes)	1:20	1:15
Mean total exam time (hours:minutes)	$> 2 : 00$	$0 : 40$
Mean confidence/query (5 very/0 not)	2	4

Inf-P	C++ BEE	Pheasant
Training time (hours:minutes)	0:20	1:35
Mean total exam time (hours:minutes)	$2 : 00$	$0 : 35$
Mean confidence/query (5 very/0 not)	3.5	4,5

Figure 10.9: Time analysis - The result times were rounded to multiples of 5 minute units.

edit"

- "The user interface could be similar to a Wizard of a Microsoft product"

- "I think the tests should be done with full execution environment"

- "Is there a way to script my query? I have the feeling sometimes it would speed up...With complex repetitive things the mouse tires me."

From these comments we can infer, for instance, that a query reuse mechanism should be provided in a final implementation solution. Also, a query history mechanism where the user can browse on past queries and respective solutions, is an extra feature which might have a great impact on user satisfaction.

10.5 Summary

In this chapter, we have detailed the procedure to validate our initial usability claims.

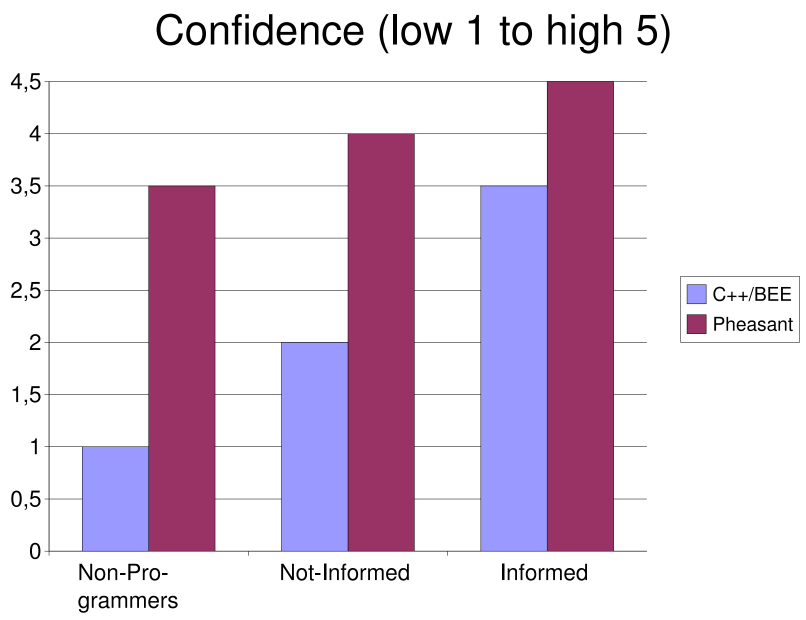


Figure 10.10: Effectiveness C++/BEE vs. Pheasant: More confidence from the user.

Let us now summarize our evaluation. In terms of **Effectiveness**, Pheasant provides the user with a tool that is more accurate and complete than the other solutions. When looking at **Efficiency** the present running approaches, we have a clear evidence that less time is used to achieve the same goal. Generally, the **Satisfaction in use** was higher with Pheasant. The only exception to this were expert users, who feel very familiar with their day-to-day tool and request further tests on the expressive power of the language by trying it out in different experiment contexts (leading it to its extremes).

Chapter 11

Conclusions

In this section we rest the case. First we will give a quick summary and conclusions of the thesis. Then we state what were our major contributions and we end with suggestions for future work.

11.1 Thesis Summary

This thesis had the main goal of presenting a solution for improving the user productivity in HEP analysis/data-mining phase. In order to do that, we have started by carefully understanding the context and the traditional procedure of the physicist while analyzing the data. This implied gathering widely dispersed information, justified by the fact that no serious studies have been done so far, and by the fact that any documentation concerning this phase is typically scarce and inconclusive.

Two key concepts from other research areas were taken into account when proposing the solution. From a survey in the area of Visual Query Systems (VQL), we concluded that hybrid VQLs were the ones that suited best our requirements. From the area of Domain Engineering, we derived a procedure to design and develop our language that was suited to this specific domain where the general purpose approach is traditionally problematic.

By combining these two concepts, we have proposed to approach our problem by developing a declarative Domain Specific Visual Query Language for HEP analysis.

With our language, which we named PHEASANT QL, we introduce an abstraction level in the system. As a consequence, the user is no longer responsible for the performance, and the computer scientist is able to optimize it without interfering with the user activities.

The work did not stop here. Proposing a DSVQL necessarily entailed proving that it was a feasible and usable approach.

From the point of view of feasibility, we have proposed a notation that uses objects from the conceptual model, not from the logical model (as it is usual in other languages). We have formally defined the language notation by mapping it to our defined algebra (based on relational algebra). The next step was the implementation of a prototype that is able to deal with this language and to generate the queries in a target language that will run against the physicist's database when compiled and executed. The architecture of this system and some design options were described in this thesis.

The final step was to prove that our approach improves productivity. In Human Computer Interfaces, this is known as evaluation of the language's usability. In order to do that, we have organized a complete evaluation session and determined efficiency, effectiveness and satisfaction in use. The evaluation corroborates our hypothesis.

As future work, we propose to use our framework to improve the efficiency of the system by deriving better algorithms (in order to be faster and to use less bandwidth and memory).

11.2 Contributions from This Thesis

The problem is very well-known in the area. To our knowledge, before this thesis was written there was no real attempt to tackle the problem in such a global and methodical manner. Therefore, during our thesis argumentation, we believe to have introduced tools to solve a long-standing question: How to improve performance in the analysis phase?

As a major contribution for both computer science and High Energy Physics Computing, we have opened up a new application area, a domain-specific visual query language for HEP, and thoroughly explored it. This can also be interpreted as a practical application of computer science tools to solve a problem in High Energy Physics.

Instead of adopting a bottom-up approach for designing the solution, where tuning and hacking legacy systems would be the only way to pro-

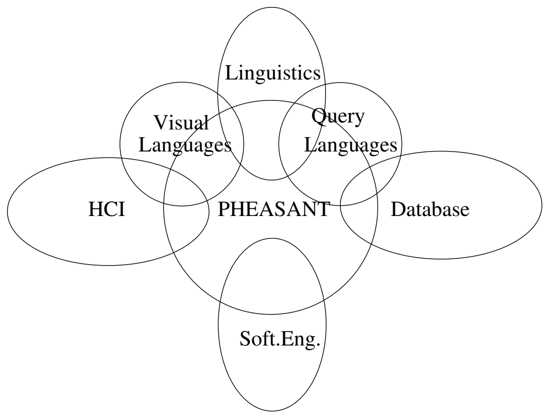


Figure 11.1: Research areas

ceed, we decided for a top-down design. With our approach, we have provided the HEP community with the concept of a unifying framework. This framework combines several areas of knowledge in computer science research (see Fig. 11.1): physics computing, databases, human centric interfaces, linguistics, software engineering (with domain-specific modeling), and the intersecting areas of query languages and visual languages. The main design strategy was to propose a way to raise abstraction, modularizing the analysis frameworks, designing a new language as a flexible query tool, and introducing possibilities for optimization. Thus, we have designed an ambitious framework by using a new software engineering methodology[8], and we have validated most of the ideas through the implementation of a prototype.

11.3 Suggestions for Future Work

In our opinion, our work can and should be extended and evolved. Given the methodology and the different phases in the query processing that were introduced by the proposed framework, we have established the foundations and opened the doors to the next phase, i.e. to study, explore and derive better or superior algorithms at the different stages. We have already proposed several directions in the thesis, in the different topics approached, as future research. We then summarize these proposals:

- **Language Design**

  At the level of the query language design, which is always an interesting subject, the potentials for optimization are promising. Starting with the language notation, passing by the definition of semantic rules that optimize before translating into the algebraic notation, and finally the syntax translation itself. For instance, semantic errors can be already filtered out, releasing the burden of the query plan optimizer to do it. This entails moving from a simple syntax translational approach to a more operational approach, exposing the semantic content to a more clever virtual machine for semantics optimization.

- **Framework Design and Implementation**

  At the framework level, there is also a huge potential for optimization. More work can be done at the user interface level (human interfaces area), introducing more techniques like undo-redo, query history mechanism, etc. We can also proceed with the algebraic optimization and invest on deriving new physical operators together with the physical plan optimization. Finally, we can invest on evaluating the storage engines performance and design when integrated to Pheasant. Eventually, given the query pattern that we have already studied, it will help us on determine the best approach.

Part V

Bibliography

Bibliography

[1] G. M. A. Kemper. *Object-Oriented Database Management: Applications in engineering and computer science.* Prentice Hall International Editions, Englewood Cliffs, New Jersey, 1994.

[2] R. Agrawal, N. Gehani, and J. Srinivasan. Odeview: The graphical interface to ode. *Proc. ACM SIGMOD conf.*, May 1990.

[3] A. Alashqur, S. Su, and H. Lam. Oql - a query language for manipulating object-oriented databases. *Proc. 15th VLDB Conf. Amsterdam*, pages 434–442, 1989.

[4] H. Albrecht. The computing model for hera-b. *Proc. CHEP'97, Berlin, edited by DESY Hamburg*, 1997.

[5] H. Albrecht and et al. Argus: A universal detector at doris ii. *NIM*, pages A275:1–48, 1989.

[6] V. Amaral, A. Amorim, and et.al. Operational experience running the herab-b database system. In H. Chen, editor, *Proceedings of CHEP 2001, International Conference on Computing in High Energy and Nuclear Physics, Beijing, P. R. China*, pages 396–397. Science Press, September 2001.

[7] V. Amaral, S. Helmer, and G. Moerkotte. Designing and implementing a new abstraction layer to optimize the hep analysis process. *IEEE Conf. Record of Nuclear Science Symposium NSS, Portland, OR, USA*, pages N26–104, October 2003.

[8] V. Amaral, S. Helmer, and G. Moerkotte. A domain specific visual query language for the high energy physics environment. In J.-P.

Tolvanen, J. Gray, and M. Rossi, editors, *3rd Workshop on Domain-Specific Modeling, An OOPSLA 2003 Workshop, Anaheim, CA, USA*, pages 9–16. Jyväskylä University Printing House, Finland, October 2003.

[9] V. Amaral, S. Helmer, and G. Moerkotte. Pheasant: A physicist's easy analysis tool. *Technical Report of the University of Mannheim: 8/03*, 2003.

[10] V. Amaral, S. Helmer, and G. Moerkotte. A visual query language for hep analysis. *IEEE Conf. Record of Nuclear Science Symposium NSS, Portland, OR, USA*, pages N26–105, October 2003.

[11] V. Amaral, S. Helmer, and G. Moerkotte. Pheasant: A physicist's easy analysis tool. In J. Carbonell and J. Siekmann, editors, *LNAI Lecture Notes in Artificial Inteligence*, pages 3055:229–242. Springer Verlag, June 2004.

[12] V. Amaral, G. Moerkotte, A. Amorim, and S. Helmer. Studies for optimization of data analysis queries for hep using hera-b commissioning data. In H. Chen, editor, *Proceedings of CHEP 2001, International Conference on Computing in High Energy and Nuclear Physics, Beijing, P. R. China*, pages 154–155. Science Press, September 2001.

[13] A. Amorim, V. Amaral, and et. al. The hera-b database management for detector configuration, calibration, alignment, slow control and data classification. In I. P. Mirco Mazzucato, editor, *Proceeding of CHEP 2000, international conference on Computing in High Energy and Nuclear Physics, 7-11 February, Padova-Italy*, pages 469–472. Imprimenda, Padova, Italy, February 2000.

[14] A. Amorim, V. Amaral, and et. al. The hera-b database services for detector configuration, calibration, alignment, slow control and data classification. *Elsevier Science, Computer Physics Communications*, 140(15):172–178, October 2001.

[15] I. Analog Devices. Adsp-2106x sharc^{TM}. *User's Manual*, 1997.

[16] I. Androustsopoulos, G. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *Journal of Natural Language Engineering*, Mars 1995.

[17] M. Angelaccio, T. Catarci, and G. Santucci. *qbd**: A graphical query language with recursion. *IEEE Transactions on Software Engineering*, 16:1150–1163, 1990.

[18] M. aude, A. Portier, and C. Trépied. A survey of query languages for geographic information systems. *Proc. 3rd International Workshop on Interfaces to Databases*, July 1996.

[19] T. Bapty, S. Neema, J. Scott, J. Sztipanovits, and S. Asaad. Model-integrated tools for the design of dynamically reconfigurable systems. *VLSI Design*, 10(3):281–306, 2000.

[20] B. Belieres and C. Trepied. New metaphors for a visual query language. *7th International Workshop on Database and Expert Systems*, 1996.

[21] R. Brun. Zebra - reference manual - rz random access package. *Program Library Q100, CERN.*

[22] D. Bryce and R. Hull. Snap: A graphics-based schema manager. *Proc. IEEE Data Eng. Conf.*, 1986.

[23] M. Carey, L. Haas, V. Maganty, and J. Williams. Pesto: An integrated query/browser for object databases. *Proc. ACM conf. VLDB*, 1996.

[24] T. Catarci. What happened when database researchers met usability. *Information Systems*, 3(25):177–212, October 2000.

[25] T. Catarci, M. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8:2:215–260, April 1997.

[26] N. Chang and K. Fu. Query-by-pictorial example. *IEEE, Tran. on Software Eng.*, 6(6):519–524, 1980.

[27] E. F. Codd A Data Base Sublanguage Founded on the Relational Calculus. Proceedings of 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, 1972.

[28] Collaboration. CMS(Compact Muon Solenoid). *http://cmsinfo.cern.ch.*

[29] Collaboration. ϵz. *http://www-zeus.desy.de/~odeppe/ez/ez.html*.

[30] Collaboration. JAS:. *http://jas.freehep.org/*.

[31] Collaboration. LHC:. *http://public.web.cern.ch/public/about/future/whatisLHC/whatisLHC.html*.

[32] Collaboration. LHCb:. *http://lhcb-public.web.cern.ch*.

[33] Collaboration. PAW. *http://www.cern.ch/paw/*, 1988.

[34] Collaboration. Atlas high-level trigger data acquisition and controls. *Technical Design Report* *http://atlasexperiment.org/*, ATLAS TDR CERN/LHCC(016), June 2003.

[35] Collaboration. Hera-B, design report. *DESY-PRC 95/01,URL:http://www-hera-b.desy.de*, January 1995.

[36] M. Consens and A. Mendelzon. Hy+: A hygraph-based query language and visualization system. *SIGMOD,Washington,DC,USA*, 93(5).

[37] M. Consens and A. Mendelzon. Expressing structural hypertex queries in graphlog. *Proceedings of the 2nd ACM Hypertext Conference*, pages 269–292, 1989.

[38] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, pages 37–45, November/December 1998.

[39] P. Cox and T. Smedley. Experiences with visual programming languages for end-users and specific domains. In J.-P. Tolvanen, J. Gray, and M. Rossi, editors, *Proc. 1st. OOPSLA Workshop on Domain-Specific Visual Languages, Tampa Bay FL*, pages 87–96. Jyväskylä University Printing House, October 2001.

[40] I. Cruz. Doodle: A visual language for object-oriented databases. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 71–80, June 1992.

[41] I. Cruz, A. Mendelzon, and P. Wood. G+: Recursive queries without recursion. *Proceedings of the 2nd Int. Conf. on Expert Database Systems*, pages 355–368, 1988.

[42] B. Czejdo and R. Elmasri. A graphical data manipulation language for an extended entity-relationship model. *IEEE Computer journal*, 23:26–36, 1990.

[43] E. M. D. Malon. Critical database technologies for high energy physics. *Proc. of the 23rd VLDB Conference Athens, Greece*, 1997.

[44] R. Davies. A metatool for visual language. *Master Thesis*, October 1997.

[45] Y. Dennebouy, M. Anderson, A. Auddino, Y. Dupont, E. Fontana, M. Gentile, and S. Spaccapietra. Super: visual interfaces for object + relationship data models. *Journal of visual languages and computing*, 1(6):27–52, 1995.

[46] J. et al. Farming in hera-b. *Proc. of the DAQ 2000 workshop at the IEEE NPSS conference, Lyon*, October 2000.

[47] J. et al. Pc farms for triggering and online reconstruction at hera-b. *Proc. of the CHEP 2001 conference, Beijing, China*, September 2001.

[48] S. K. et. al. Improving the performance of high-energy physics analysis through bitmap indices. *Proc. 11th International Conference on Database and Expert Systems Applications DEXA 2000, London, Greenwich,UK*, 2000.

[49] L. Fegaras. Ldb database:. *http://lambda.uta.edu/lambda-DB.html*.

[50] L. Fegaras. Voodoo: A visual object-oriented database language for odmg oql. *Proc. ECOOP Workshop on Object-Oriented Databases*, pages 61–72, 1999.

[51] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database systems*, 25(4):457–516, December 2000.

[52] D. Fogg. Lessons from 'living in a database' graphical query interface. *Proc. ACM SIGMOD Int. Conf. management of Data*, pages 100–106, June 1984.

[53] A. Franzke. Querying graph structures with g^2ql. *Fachbericht Informatik 10/96, Universitt Koblenz-Landau*, 1996.

[54] F.Sánchez. Digital signal processor software for the hera-b second level trigger. *Proc. CHEP 98 conference, CHICAGO*, September 1998.

[55] T. Glebe. Clue - the bee event model library. *HERA-B Note 01-138, Software 01-019*, 2001.

[56] T. Glebe. Pattern - high level tools for data analysis. *Internal report HERA-B Note 02-002, Software 02-002, DESY*, 2002.

[57] K. Goldman, P. Kanellakis, S. Goldman, and S. Zdonik. Isis: Interface for semantic information system. *Proc. ACMSIGMOD Int. Conf. Management of Data*, pages 328–342, May 1985.

[58] M. Gyssens, J. Paredaens, J. Bussche, and D. Gucht. A graph-orientedd object database model. *POODS*, pages 417–424, 1990.

[59] J. W. Hector Garcia-Molina, Jeffrey Ullman. *Database System Implementation.* Prentice Hall, 2000.

[60] S. Herot. Spatial management of data. *ACM Trans. Database Systems*, 5:493–514, December 1980.

[61] H.Perkins and Donald. *Introduction to High Energy Physics.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[62] Y. Ionnadis. Advanced user interfaces for database systems. *SIGMOD RECORD*, 21(1), March 1992.

[63] P. Johnson. *Human Computer Interaction.* McGraw-Hill, London, 1992.

[64] T. Joseph and A. Cardenas. Picquery: A high level query language for pictorial database management. *IEEE Trans. Software Eng.*, 14:630–638, May 1988.

[65] R. King. A database management system based on an object model. *Expert Database Systems*, pages 443–467, 1986.

[66] B. Knuteson. Quaero: Motivation, summary, status. *Proc. CHEP 2003, UC San Diego, USA*, 2003.

[67] M. Kuntz and R. Melchert. Pasta-3's graphical query language: direct manipulation, cooperative queries, full expressive power. *Proc. 15th VLDB Conf.*, August 1989.

[68] B. M. M. Jaedicke. User-defined table operators: Enhancing extensibility for ordbms. *25th VLDB Conference, Edinburg, Scotland*, 1999.

[69] N. Macdonald and M.Stonebraker. Cupid: A user friendly graphics query language. *Proc. ACM pacific*, pages 127–131, 1975.

[70] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object oriented dbms. *Report no. CS/E-86-005, Oregon Graduate Center*, 1986.

[71] C. Manoj. Towards an odmg-compliant visual object query language. *Proc. of the 23rd VLDB Conference Athens, Greece*, 1997.

[72] L. Mohan and R. Kashyap. A visual query language for graphical interaction with schema-intensive databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5), October 1993.

[73] A. Motro, A. D'Atri, and L. Tarantino. The design of kiview: An object-oriented browser'. *Proc. 2nd Int'l. Conf. on Expert Database Sys.*, April 1988.

[74] N. Murray, N. Paton, and C. Goble. Kaleidoquery: A visual query language for object databases. *Int. working Conference in Advanced Visual interfaces*, May 1998.

[75] OMG, (Object Management Group, Inc.) Meta Object Facility (MOF) Specification Version 1.4 (April 2002),*http://www.omg.org*.

[76] A. Papantonakis and P. King. Syntax and semantics of gql, a graphical query language. *Journal of Visual Languages and Computing*, 6:3–25, 1995.

[77] J. Paredaens, P. Peelman, and L. Tanca. G-whiz, a visual interface for the functional model with recursion. *Proc. 11th Int. Conference on Very Large Databases,Stockolm*, pages 209–218, 1985.

[78] J. Paredaens, P. Peelman, and L. Tanca. G-log: A graph-based query language. *IEEE Transactions on Knowledge and Data Engineering*, 3(7):436–453, 1995.

[79] A. Portier. Grasp: A graphical system for statistical databases. In *CIGALES: un langage graphicque d'interrogation de Systémes d'Information Géographiques*, Ph.d Thesis. University of Paris, 1992.

[80] E. Powabbas and M. Rafenelli. A pictorical query language for querying geographic databases using positional and olap operators. *Sigmod Record*, 31(2):22–27, June 2002.

[81] F. Rademakers and R. Brun. Root: An object-oriented data analysis framework. *Proc. AIHENP'96 Workshop, Nucl.Inst. Meth. in Phys. Res. A 389 (1997),Lausanne . See also `http://root.cern.ch`*, pages 81–86, September 1996.

[82] P. Reisner. Query languages. In M. Helander, editor, *Handbook of Human-Computer interaction*, volume 420, pages 257–280. Elsevier Science publishers B. V., North-Holland, 1988.

[83] S. Ross. Introduction to probability and statistics for engineers and scientists. Wiley series in probability and mathematical statistics, John Wiley and Sons, 1987.

[84] B. Schneiderman. Visual user interfaces for information exploration. *Proc. of the 54th Annual Meeting of the American Society for Information Science,Medford. NJ. Learned Information Inc.*, pages 379–384, 1991.

[85] T. Shih, Y. Tsai, J. Hung, and D. Jiang. A case tool supports the software life cycle of participator dependent multimedia presentations. *ICMS*, pages 200–203, 1998.

[86] D. Shipman. The functional data model and the data language daplex. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[87] A. Shoshani, L. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional indexing and query coordination for tertiary storage management. *IEEE, 11th International Conference on Scientific and Statistical Database Management, Cleveland, Ohio*, page 214, July 28-30 1999.

[88] F. Staes, L. Tarantino, and A. Tiems. A graphical query language for object oriented databases. *Proc. IEEE Workshop on Visual Languages,Kobe,Japan*, pages 205–210, 1991.

[89] S. Thibault. Domain-specific languages: Conception, implementation and application. *Phd. Thesis*, October 1998.

[90] K. Vadaparty, Y. Aslandogan, and G. Ozsoyoglu. Towards a unified visual database access. *ACM SIGMOD*, pages 357–366, 1993.

[91] A. van Deursen, P. Klint, and J. Visser. Domainspecific languages. *Tech. Report SEN-R0032,http://www.cwi.nl/arie, paulk,jvisser/*, November 2000.

[92] K. Wittenburg. Early-style parsing for relational grammars. *Proc. IEEE Workshop Visual Languages*, pages 192–199, 1992.

[93] H. Wong and I. Kuo. Guide: A graphical user interface for database exploration. *Proc. 8th VLDBD Conf.*, pages 22–32, 1982.

[94] M. Zloof. Query by example. *IBM Systems Journal*, 4:324–343, December 1977.

Glossary

Abstraction - supression of irrelevant details.

AOD - Physics analysis object data, information used in final analysis.

ASG - Abstract Syntax Graph.

AST - Abstract Syntax Tree.

CERN - European Laboratory for Particle Physics, Geneva, Switzerland.

DESY - In Hamburg / Germany.

Detector data - Data that describes and qualifies the detecting apparatus, and are used to interpret the event data (structure, geometry, calibration, alignment, environmental parameters). Statistical data - resulting data from processing a set of events (histograms, n-tuples).

DSP - Digital Signal Processor.

DSVL - Domain Specific Visual Language.

DSVQL - Domain Specific Visual Query Language.

ESD - Event summary data. Information required for detayled analysis and high level reconstruction.

Event data - Data obtained from particle collisions, and their subsequent refinements (raw data, reconstructed data, analysis data, etc...).

GPL - General Purpose Language.

GUI - Graphical User Interface.

HEP - High Energy Physics.

HERA-B - Experiment in DESY.

IDL - Interface Definition Language.

LEP - Large Electron Positron collider.

LHC - Large Hadron Collider.

Meta-Data - That describes other data, like the statistics and event catalog (example Run).

Model - formal specification of a function,structure and/or behaviour of a system.

Monte Carlo Simulation - Random generation of values for certain variables according to a model. Used when there is the requirement to automatically analyze the effect of varying inputs on outputs of the modeled system. This simulation technique was named for Monte Carlo, Monaco, where the primary attractions are casinos containing games of chance such as roulette wheels, dice, and slot machines, that exhibit random behavior. This statistics technique is very often used for the generation of simulated physics data. It follows a complex model to simulate all the particles that cross the detector, their interactions between them and with the detector, in order to simulate the data that comes out of the detector (hits).

n-tuple - The flat (or table) model that consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another.

NASA - National Aeronautics and Space Administration.

ODL - Object Description Language.

Particle Accelerator - uses electric/magnetic fields to propel charged particles to great energies. Quadrupole magnets are used to focus the particles into a beam and prevent their mutual electrostatic repulsion from causing them to spread out.

Particle Collider - the purpose of an accelerator is to generate high energy particles for interaction with matter. This entails provoking a collision using usually a fixed target. The other way is to make these particles collide with particles accelerated in oposite directions.

Platform - general term to unify technological and engineering details that are irrelevant to the fundamental functionality of a software component.

QL - Query Language.

RAW - Real raw data. data read directly from the detector and eventually processed on-line. Defined as being WORM data (write once read many),must be securely stored and never modified.

RTTI - Run Time Type Identification.

Run - Meta-data information for the `Event` data that is being collected, such as the parameters of the experiment, e.g. the setup of the detectors, the time span during which data acquisition took place and general quality issues.

SIM - Simulated raw data.

SLAC - Stanford Linear Accelerator Center.

TAG - Event tag data, summaries the main feature of an Event in order to be used for fast event selection.

TESLA - The Superconducting Electron-Positron Linear Collider with an Integrated X-Ray Laser Laboratory. To be built in the future.

VL - Visual Language.

VME - VERSAmodule Eurocard. Systems for mission-critical and real-time applications.

Index

Part VI

Appendix

Appendix A

The BEE framework

BEE is a layer(wrapper) on top of ROOT[81] that introduces the schema of the analysis data of the Hera-B experiment. ROOT, by its turn, is meant to deal with large amounts of event data. It's primary goal is to support the Particle Physics analysis, under the assumption that physicists doing analysis, are mostly concerned with the manipulation of the computed results in histograms and n-tuples.

The framework integrates several functionalities:

- **CINT C/C++ Interpreter** [1] - It allows the interactive ROOT command line with the C/C++ scripting language. Large scripts can be compiled and dynamically linked, making the extension of the framework very easy.
- **The ROOT Dictionary** - Functions, global variables and classes are stored in its memory resident dictionary. This dictionary is much more extensive than the RTTI[2] facility as proposed in the C++ standard. The dictionary is generated by the CINT Dictionary Generator using the C++ header files without requiring ODL[3] or IDL [4].
- **Automatic Document Generation** - Using the dictionary and the comments stated in the source files ROOT can automatically

[1]by Masaharu Goto of Hewlett Packard Japan
[2]Run Time Type Identification
[3]Object Description Language
[4]Interface Definition Language

generate a source code documentation both in HTML and PostScript format.

- **GUI Classes and Object Browser** - Embedded in the ROOT system is an extensive set of GUI classes. The GUI classes provide a full OO-GUI framework as opposed to a simple wrapper around a GUI such as Motif. All GUI classes are fully scriptable and accessible via the interpreter,(which makes it very easy to do fast prototyping of widgets layout). A very complete library of histogramming with fitting methods was included and is the main point of attraction to physicists and mathematicians since this scientists are able to get advanced statistical analysis (multi dimensional histogramming, fitting and minimization algorithms) together with visualization tools. The facility of being able to deal with histograms and n-tuples as persistent objects in the ROOT database files format is another important feature.

- **ROOT object I/O System and Class/Schema evolution** - The framework was developed to include general purpose language's functionality such as distribution and object persistency. Despite the fact that ROOT is not a DBMS, the persistency mechanism is being tuned to match physics' data storage retrieval and analysis requirements.

- **Distributed system** - Using the PROOF (Parallel ROOT Facility) extension, large databases can be analyzed in parallel on Massively Parallel Processing (MPP) and Symmetric Multiprocessing (SMP) systems or loosely coupled workstation/PC clusters.

- **ROOT and ODBC** - This library is a set of classes that provides an interface to ODBC. It is implemented as ROOT wrappers of libodbc++. As usual with this packages it allows: establishing a connection from ROOT session to any database for which an ODBC driver is available; send SQL statements and process the results.

Appendix B

Physical operators' algorithms

B.1 Stream Class

Stream

Attributes:
kept tuple
last path
found tuple

Methods:
Open
Close
Next tuple
memorize found tuple
forget found tuple
memorize last path for unnest
forget last path
memorize kept tuple
forget kept tuple
is stream opened?
is stream closed?

B.2 Table-scan/Selection

Table-Scan/Selection
Input: Stream s, Predicate pred
Output: Tuple

```
get next tuple from stream s
$while$(tuple exists)
 {
  $if$( predicate is true)
  return tuple
   get next tuple from stream s
 };
 return No Tuple
```

B.3 Table-scan/Selection

Reduce
Input: Stream s, Predicate pred, Head head
Output: Tuple

```
get next tuple from stream s
$while$(tuple exists)
 {
  $if$( predicate is true)
  return head(tuple)
   get next tuple from stream s
 };
 return No Tuple
```

B.4 Operators for sets

Union
Input: Stream sx, Stream sy
Output: Tuple

```
*if* (sx is closed)
   return next tuple from sy stream
get next tuple x from stream sx
*if* (tuple x exists)
   return tuple x
close stream sx
return next tuple from sy
```

Intersection
Input: Stream sx, Stream sy
Output: Tuple

```
get next tuple x from sx
*if* (tuple x exists)
   open sy
   get next tuple y from sy
   *while* (evt id of x is different from evt id of y)
       get next tuple y from sy
   close stream sy
   *if* (tuple y exists) return tuple x
return No Tuple
```

Difference
Input: Stream sx, Stream sy
Output: Tuple

```
get next tuple x from sx
$if$ (tuple x exists)
    open sy
    get next tuple y from sy
    $while$ (evt id of x is different from evt id of y)
        get next tuple y from sy
    close stream sy
    $if$ (tuple y does not exist) return tuple x
return No Tuple
```

B.5 Operator for Unnesting

Unnest
Input: Stream s, Predicate pred, Bool outer, Path path
Output: Tuple

```
get next tuple x from s
while (tuple x exists)
  {
     tuple y= next path of x
     s memorizes last path
     while (tuple y exists)
         tuple z= y appended to x
        if (predicate of z is true)
          s memorizes x
          s memorizes it was found a tuple
          return z
        tuple y= next path of x
     we have reached the end of inner part
     if ( outer is true and s does not remember found tuple)
        s keeps memory of x
     s forgets memory of x
     s forgets if it was found a tuple
     s forgets last path
     get next tuple x from s
     if (tuple x exists )
        if (outer is true and s kept memory of old x)
          s memorizes tuple x
          set tuple result= kept appended by <>
          s forgets kept tuple
          return result tuple
     else if (outer is true and s kept memory of old x)
          reached the end of the outer stream
          return the kept old x appended with <>
  }
  return No Tuple
```

B.6 Operators for Join

Nested_Loop
Input: Stream sx and sy, Predicate pred, Boolean outer
Output: Tuple

```
get next tuple x from sx
*while* (tuple x exists)
  {
    get next tuple y from sy
    *while* (tupple y exists)
    {
       *if* (predicate is true)
          sx memorizes x for next tuple
          return tuple $< x, y >$
       get next tuple y from sy
    }
    we have reached end of inner stream
    *if* ( outer is true and it does not remember x)
       outer was not joined
       keep x as left
    sx forgets x
    get next tuple x from sx
    *if* ( tuple x exists)
     open again sy
    *else if* (outer is true and exists left)
       we have reached the end of outer stream
       $return < left, null >$;
  }
  return No Tuple
```

B.7 Operators for Nest

Nest
Input: Stream s, Predicate pred, Head Function head,
Aggregate Function agg
Output: Tuple

Assumes that all the tuples with the same evt id are consecutive
head function inputs tuple and returns $< value >$ or $< value, tuple >$

```
get next tuple x from stream s
s forgets any tuple it might remember
while(tuple exists)
 {
     tuple result_tuple initiates with No Tuple
     tuple y=x
     get evt.id to keepid from x
     if (exists evt.id)
        while (exists tuple y and evt.id of y=keepid)
           result_tuple=agg(head(y),result_tuple)
           get next tuple y from s
     else get nest tuple y from stream s
     if ( predicate of result_tuple is true)
         s memorizes y
         return result_tuple
     tuple x = y
 }
 return No Tuple
```