IT 08 020
Examensarbete 30 hp
Maj 2008

Storing and Searching Scientific Data with a Relational Database System

Luis Urea

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Storing and Searching Scientific Data with a Relational Database System

Luis Urea

Working with scientific data involves analyzing large amounts of data. For High Energy Physics research, one is interested in analyzing series of different events that occur when particles collide and filter those that fulfill predefined conditions. With the proposed approach the conditions are expressed using numerical formulas expressed as SQL queries in a relational database (RDBMS). Different kinds of database representations were designed, studied, and compared in order to get the most efficient and scalable way to store and access the data for the application.

Handledare: Ruslan Fomkin
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 08 020
Tryckt av: Reprocentralen ITC

Contents

1. Introduction 2

2. Background 3

2.1 Relational Databases. 3

2.2 High Energy Physics Application 8

3. Representing High Energy Physics Data in Relational Databases 9

3.1 Implementation of Analysis Queries 10

3.1.1 Schema Dimension 11

3.1.1.1 Tables Using Flags 11

3.1.1.2 Replicated attributes 13

3.1.1.3 All Particle Data in One 15

3.1.2 Implementation Dimension 17

3.1.2.1 Views 18

3.1.2.2 Functions 24

3.2 Scalar Functions for Numerical Formulas 31

4. Performance Evaluation 33

4.1 Setup Process 33

4.2 Import Data Times 33

4.3 Execution Times 41

4.4 Discussion 52

5. Summary and Future Work 53

REFERENCES 54

1. Introduction

The application area for this work is High Energy Physics (HEP), where large quantities of data to be analyzed are generated. A particular case for these data is the description of the effects from collisions of *particles* pairs. A description of a collision is called an *event*. The analyzed data are sets of *events*, where each *event* has properties that describe sets of *particles* of various types produced by a collision. Scientists define the analyses in terms of these *event* properties. As every collision is simulated independently of other collisions, the *events* are also independent. The analyses are expressed as selections for *events* satisfying certain conditions, called *cuts*. The query results are sets of interesting *events* satisfying the *cuts*. A typical query is a conjunction of a number of *cuts*.

The purpose of the developed relational database, called Storing and Searching Scientific Data with a Relational Database System (S3RDB), is to store and query the data generated by simulation software from the Large Hadron Collider (LHC) experiment ATLAS in a relational database. The scientist method specifies the *cuts* as database queries, using the standard SQL query language. Query optimization by the relational database management system (RDBMS) provides scalability and high performance without any need for the scientist to spend time on low-level programming. Furthermore, as queries are easily specified and changed, new theories, e.g. implemented as filters, can be tested quickly [1].

Queries over *events* are complex since the cuts themselves are complex, containing many predicates. The query conditions involve selections, arithmetic operators, aggregations, projections, and joins**.** The aggregations compute complex calculations derived *event* properties. This complexity makes queries extremely expensive on time and recourses. For our application, this problem was previously solved using an object-oriented database [1]; in the present we implement it using Microsoft's SQL Server RDBMS.

2. Background

2.1 Relational Databases

Relational database technology is based on the relational model developed by Edgar Frank Codd [2]. A relational database allows the definition of data structures, storage and retrieval operations and integrity constraints. In these databases the data and relations between them are organized in *tables*. A *table* is a collection of records and each record in a *table* contains the same fields.

Properties of relational *tables* [3]:
- Values are atomic.
- Each row is unique.
- Column values are of the same kind.
- The sequence of columns is insignificant.
- The sequence of rows is insignificant.
- Each column as a unique name.

A relational database conforms to the relational model where data is represented as a set of *tables*. A *table* is a set of data elements (values) that is organized using horizontal rows, called *tuples*, and vertical columns, called *attributes*. The *attributes* are identified by names, and *tuples* by the value of a particular attribute (or set of attributes) called *key*. A *unique key* or *primary key* is a *candidate key* to uniquely identify each tuple in a *table*. Depending on its design; a *table* may have arbitrarily many *unique keys* but at most one *primary key*. A *foreign key* is a reference from a *tuple* attribute to a *key* in another *table* inside the same database.

The *cardinality* of one *table* with respect to another *table* is a critical aspect of database design. For example, in a database designed to keep track of hospital records there may be separate data *tables* keeping track of doctors and patients,

with a *many-to-one* relationship between the records in the doctor table and records in the patient table. Whether data *tables* are related as *many-to-many* (M,N), *many-to-one* (M,1), or *one-to-one* (1,1) is said to be the *cardinality* of a given relationship between *tables* [2].

The *database schema* represents the description of the structure of the database. In a relational database, the *schema* defines the *tables,* the fields in each *table*, and the relationships between fields and *tables*. In the common *architecture of three schemas* the following *schema* levels are defined:

> The *internal schema* describes the physical structure of how data is stored in the database. This *schema* uses a model of physical data and gives details for its storage, and also the access paths to the database.
>
> The *conceptual schema* describes the structure of the complete database. It hides the details of the storing physical structures and concentrates on describing entities, data types, links, user operations, and restrictions.
>
> The *external level* or *user view* includes various *external schemas* or *user views*. Each *external schema* describes the parts of the databases that are of the interest of a group of users and hide the rest of the database [2].

An *entity-relationship model* [2] is an abstract conceptual representation of structured data; *entity-relationship modeling* is the process of generating these models. The end product of the modeling process is an *entity-relationship diagram* or *ER diagram*, a type of conceptual data model. An *ER-diagram* is a high-level graphical notation used when designing relational databases. Database design includes translating these *ER-diagrams* to relational *database schemas*. For a given *ER-diagram* there are many possible *relational database schemas* and the designer should choose the most suitable one. In the *ER model*, *entities* are represented by squares, *attributes* by circles, relationship

between *entities* by rhombus, the *primary keys* underlining the *attributes* and the *cardinalities* expressing their respective values.

Extended entity-relationship diagrams (EER-diagram) extends basic *ER-diagrams* with inheritance by mean of *class hierarchies*.

Class hierarchies consist of *superclasses* and *subclasses*, in which each *subclass* has a relationship with its superclass. *Subclasses* inherit the attributes and methods of their *superclasses*, and they may have additional attributes and methods of their own. Based on that, the concept of *specialization* appears; this defines a set of *subclasses* to one *superclass*.

With the concept of *specialization* appears two new concepts, the first one is the *disjoining* or *overlaping constraint*. *Disjoining* define that a *tuple* in a *superclass* can belong at most to one of their *subclasses,* and *overlapping,* that allows one *tuple* in a *superclass* to belong to more than one *subclass*. And the second one is the *total* or *partial specialization*. *Total specialization* specifies that all *tuples* in the *superclass* must belong to at least one of the *subclasses*, and *partial specialization* permits that *tuples* in the *superclass* to do not belong to one of the *subclasses*

Queries specify how information is extracted from the relational database. The term *query* is also used for SQL commands that update the database. *Relational queries* are expressed using the query language SQL [2]. In this project, the *queries* are implemented by T-SQL, which is the SQL dialect used in SQL Server DBMS; it is based on the SQL-2003 standard

A *selection* is a mechanism to specify which data is needed from the database. In SQL, *selections* have the structure SELECT, WHERE, FROM, e.g.:

```
SELECT LastName
FROM Members
WHERE Age>30
```

SELECT specifies which attributes of the *tuples* are going to be taken; FROM, from which *table* are the *tuples* taken; and WHERE, which conditions have to be fulfilled.

A *join* is an operation performed on *tables* of data in a relational database in which the data from two tables is combined in a larger, more detailed joined table. A *join* clause in SQL combines records from two *tables* in a relational database and presents the results as a *table*. *Queries* uses this *joins* in order to navigate and combine different *table* to search for data that was asked for. [2]

Aggregation operators compute values based on sets of database values, e.g. summing the incomes of employees in some department. [2]

A *projection operation* picks out listed columns from a relation and creates a new relation consisting of these columns. It is mostly used to take atributes necessaries from a *query*. [2]

A *view* is a virtual or logical table defined as a *query.* A *view* can be used in *queries* and in other *view* definitions [4].

A *stored procedure* is a user program written in a query language running inside the database server.

A database index is a data structure that improves the speed of operations in a *table*. *Indexes* can be created using one or more atributes, providing the basis for both rapid random lookups and efficient ordering of access to records [5].

In SQL-2003 [6] *functions* were introduced into SQL. This version supports three kinds of SQL functions:

1. Scalar functions.

   Scalar functions return a single data value (not a *table*) with a RETURNS clause. *Scalar functions* can use all scalar data types, with exception of timestamp and user-defined data types [6]. For example:

```
Create function pt
           (@px real, @py real)
Returns Real
AS
BEGIN
          Return ( select sqrt(@px*@px + @py*@py))
END
```

2. Inline table-valued functions.

   In-line table-valued functions return a result *table* defined by a single SELECT statement [6]. For example:

```
Create function oppositeLeptons
          (@idevent INT)
Returns TABLE
AS
Return  select   l1.px as l1px, l1.py as l1py,
                 l1.pz as l1pz, l1.ee as l1ee,
                 l2.px as l2px, l2.py as l2py,
                 l2.pz as l2pz, l2.ee as l2ee, l1.eventid
        from Leptons as l1, Leptons as l2
        where   l1.kf = -l2.kf
                and l1.eventid = @idevent
                and l1.eventid=l2.eventid;
```

3. Multistatement table-valued functions.

Multistatement table-valued functions return a *table*, which was built with many SQL-2003 statements [6]. For example:

```
Create function dbo.f_LotsOfPeople(@lastNameA as nvarchar(50), @lastNameB as
nvarchar(50))
        returns @ManyPeople table
        (PersonID int, FullName nvarchar(101), PhoneNumber nvarchar(25))
        as
        begin
              insert @ManyPeople (PersonID, FullName, PhoneNumber)
              select ContactID. FirstName + ' ' + LastName, Phone
              from Person.Contact
              where LastName like (@lastNameA + '%');
         return
end
```

2.2. High Energy Physics Application.

The data consist of *events*, which are collisions of different *particles* in High Energy Physics (HEP), and all the *particles* that are involved in those *events*. These *events* include three kinds of *particles* (*electrons, muons* and *jets*). The SQL queries determine if the collisions fulfill certain conditions called *cuts.*

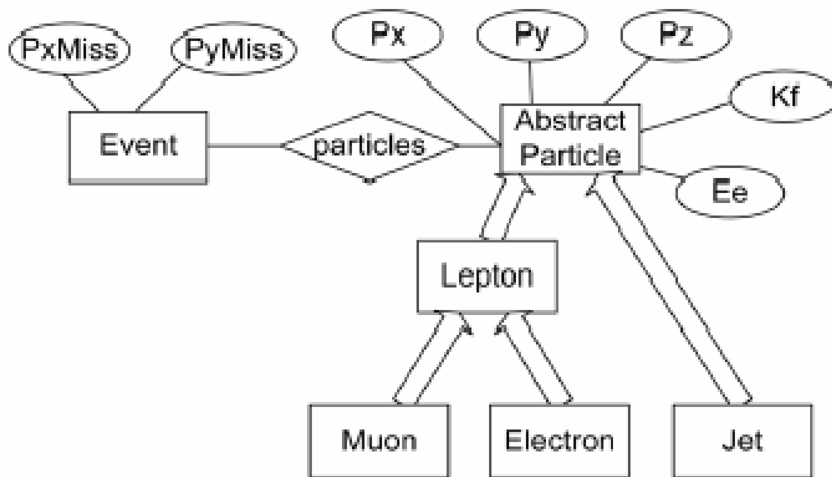

Figure 1 EER schema for Atlas Experiment [1]

The conceptual schema of the database storing LHC *events* is illustrated by the EER-diagram in Figure 1. *Events* represent collisions in which a certain number

of *particles* are involved. These *events* are represented in an entity called *Events*, which have the attributes *PxMiss* and *PyMiss*. Every *particle* that belongs to an *event* is represented by entity called *Particles* with the attributes *Kf, Px, Py, PZ* and *Ee*. *Particles* are related to *Events;* also they are subdivided in the subtypes *Muons*, *Electron*, and *Jets*. *Muon* and *Electron* are represented as subclasses of the entity *Leptons*. *Leptons* and *Jets* are subclasses of entity *Particles*. On the schema in Figure 1 arrows represents inheritance from the superclasses, which mean that all attributes and keys from the superclasses are inherited by the subclasses.

The *cuts* are the conditions that an *event* has to fulfill in order to produce a *Higgs Boson*. The *Higgs boson* is a hypothetical massive scalar elementary particle predicted to exist by the Standard Model of particle physics [7]. There are six kinds of *cuts*, which are called *JetVetoCut, zVetoCut, TopCut, MissEeCuts, LeptonCuts*, and *threeLeptonsCut*. In order to specify the *cuts*, several numerical queries are defined; in other to be use to calculate *cuts*, E.g. *Pt* and *ETA*.

The numerical formulas of *Pt* and *Eta* are:

$\sqrt{x^2 + y^2}$ and $0.5 \bullet \ln\left(\frac{\sqrt{x^2 + y^2 + z^2} \ + z}{\sqrt{x^2 + y^2 + z^2} \ - z}\right)$ respectively [1].

A search for the Higgs Boson according to one possible theory can be formulated as *{ev I jevVetoCut(ev) ^ zVetoCut(ev) ^ TopCut(ev) ^ MissEeCuts(ev) ^ LeptonCuts(ev) ^ ThreeLeptonCut(ev)}.*

3. Representing HEP in a Relational DBMS

S3RDB stores all HEP *events* in a relational database and the *cuts* are implemented as SQL queries. The purpose of the project is to evaluate the

performance of the use of relational DBMS of for this kind of scientific applications.

It was decided to use a Microsoft SQL Server 2005™ (MSSQL2005) as a relational DBMS, using SQL and SQL-2003 as query languages. MSSQL2005 includes SQL-2003 facilities such as *stored procedures* and *functions*.

Several solutions were implemented and tested, to finally conclude which one of them is the best solution for the chosen DBMS platform.

3.1 Implementation of Analysis Queries

The solutions are based on two kinds of dimensions: the *database schema* dimension and the *query implementation* dimension. These dimensions are explained in this section. Also the scalar functions used in the numerical computations needed for the queries will be explained.

3.1.1 Schema Dimension

This dimension deals with how data is stored, which tables are used, which attributes each table has, and how the tables are related.

Three different database schemas were created in order to investigate different approaches to solve the problem and to determine the preferred one.

All schemas use identification of *events* and *particles*, where *Events* have their own id called *eventid* and the attribute *filename* that indicates from which file the *event* or a group of *events* were taken. *Particles* have the attributes *id* and *idap*, id is the same id that comes from the source file. Due two different *Particles* tuples could have the same *id*, the attribute *idap* is created to provide them a

unique identification. Also they have the *eventid* of the *event* to which they belong.

3.1.1.1 Tables Using Flags (*RepeatID*)

The first solution is based on ID flags. This means that the inheritance of the subclasses is made by creating the tables of the subclasses and giving them the same ID of the main superclass. All attribute values are given to the superclass *Particles*. In this case, a tuple in the table *Particles* will have all the values of *Px, Py, Pz, Ee,* and *Kz*. To indicate that a *muon, electron, lepton* or *jetB* are the same *particle* on *Particles*, they must have the same ID. This attribute id on the subclasses is used only in this schema to identify that a tuple in subclass in the same instance in the superclass that it belongs.

In this schema *Events* are related to *Particles* by the *eventid* and *Particles* are related to *Leptons, Muons Electron* and *Jets* by it *idap*. The inheritance is represented by repeating *idap* in every subclass of *Particles.* This *idap* is a unique key that identifies each *particle*, and allows identifying which tuple of *Lepton, Electron, Muon* or *Jet* is related to which tuple in the table *Particles,* in the subclasses this attribute is just called *id*. Also every tuple of *Particle* has an *eventid* that represents the *event* that they belong to. The table *Particles* is directly related to the table *Events*, and then is divided in two tables, the *Leptons* table and the *Jets* table, at the same time the *Leptons* table is divided into an *Electrons* table and a *Muons* table. The attributes *Id* and *filename* are together the primary key on the *Events* table, and the attribute *idevent* is unique key for every tuple in the same table. *Idap* is a primary key on the table *Particles, and Leptons*, *Muons*, *Electrons* and Jets receive id*a*p as a foreign key from *Particles.*

To query specific kind of *particles* on *RepeatID* the implementation could be done using views that join specific particles tables with the main *Particles* table by their ids

The following is the SQL schema definition code used for defining the *RepeatID* schema:

```
CREATE Table Events (
idevent INT IDENTITY(1,1) unique,
id INT not null,
PxMiss Real not null,
PyMiss Real not null,
filenames Varchar(50) not null);
Constraint pk_event Primary key(id,filenames);

CREATE Table Particles(
idap INT IDENTITY(0,1) primary key,
id INT not null,
eventid INT not null,
Px Real not null,
Py Real not null,
Pz Real not null,
Kf Real not null,
Ee Real not null);
Constraint ParticlesId FOREIGN KEY (eventid)
REFERENCES Events (idevent) ON DELETE CASCADE;

Create Table Leptonaux(
id INT not null);
CONSTRAINT pk_leptonaux  PRIMARY KEY (id);
Constraint leptonId FOREIGN KEY (id)
REFERENCES Particles (idap) ON DELETE CASCADE;

create view Leptons AS
Select Particles.*
From leptonaux
Inner JOIN Particles
ON leptonaux.id = Particles.idap;

Create Table Muonaux(
id INT not null);
CONSTRAINT pk_muonaux  PRIMARY KEY (id);
Constraint muonId FOREIGN KEY (id)
REFERENCES Particles (idap) ON DELETE CASCADE;

create view Muons AS
Select Particles.*
From muonaux
Inner JOIN Particles
ON muonaux.id = Particles.idap;
```

```
Create Table electronaux(
id INT not null);
CONSTRAINT pk_electronaux PRIMARY KEY (id);
Constraint electronId FOREIGN KEY (id)
REFERENCES Particles (idap) ON DELETE CASCADE;

create view Electrons AS
Select Particles.*
From electronaux
Inner JOIN Particles
ON electronaux.id = Particles.idap;

Create Table jetaux(
id INT not null);
CONSTRAINT pk_jetaux PRIMARY KEY (id);
Constraint jetId FOREIGN KEY (id)
REFERENCES Particles (idap) ON DELETE CASCADE;

create view Jets AS
Select Particles.*
From jetaux
Inner JOIN Particles
ON jetaux.id = Particles.idap;
```

3.1.1.2 Replicated attributes (*DuplicateData*)

This schema is similar to the *RepeatID* schema, with the difference that here every tuple repeats all the information that the superclass has in every subclass to have a faster access to all the attributes of a *particle*. This means that, for example, a tuple in *Muons* with the value *idap* 1, will have all it data values stored in it intances in *Muons*, *Leptons*, and *Particles* with the same idap 1. The same way works for *Electrons* and *Jets*.

For *DuplicateData, particles* data could be selected directly from the specific particles tables, because all particle data are physically stored inside them.

The following is the SQL schema definition code to define *DuplicateData* schema:

```
CREATE Table Events (
idevent INT IDENTITY(1,1) primary key,
id INT not null,
```

```
PxMiss Real not null,
PyMiss Real not null,
filenames Varchar(50) not null);

CREATE Table Particles(
idap INT IDENTITY(0,1) primary key,
id INT not null,
eventid INT not null,
Px Real not null,
Py Real not null,
Pz Real not null,
Kf Real not null,
Ee Real not null);
Constraint particleId FOREIGN KEY (eventid)
REFERENCES Events (idevent) ON DELETE CASCADE;


CREATE Table Leptons (
idap INT  primary key,
id INT not null,
eventid INT not null,
Px Real not null,
Py Real not null,
Pz Real not null,
Kf Real not null,
Ee Real not null);
Constraint leptonId FOREIGN KEY (idap)
REFERENCES Particles (idap) ON DELETE CASCADE;


CREATE Table Muons (
idap INT primary key,
id INT not null,
eventid INT not null,
Px Real not null,
Py Real not null,
Pz Real not null,
Kf Real not null,
Ee Real not null);
Constraint muonId FOREIGN KEY (idap)
REFERENCES Leptons (idap) ON DELETE CASCADE;

CREATE Table Electrons(
idap INT primary key,
id INT not null,
eventid INT not null,
Px Real not null,
Py Real not null,
Pz Real not null,
Kf Real not null,
Ee Real not null);
Constraint electronId FOREIGN KEY (idap)
REFERENCES Leptons (idap) ON DELETE CASCADE;
```

```
CREATE Table Jets (
idap INT primary key,
id INT not null,
eventid INT not null,
Px Real not null,
Py Real not null,
Pz Real not null,
Kf Real not null,
Ee Real not null);
Constraint jetId FOREIGN KEY (idap)
REFERENCES Particles (idap) ON DELETE CASCADE;
```

3.1.1.3 All Particle Data in One Table (*BigTable*)

The third and last solution presents only a single large table *Particles* with all the *particles* and its attributes in it. As a difference with the other schemas, this has only one table and extra special attribute that indicate which kind of *particle* is stored.

This schema presents all *particles* data in only one table, which includes all the information about these *particles* and an extra attribute called *type* that identifies if the *particle* is a *muon*, an *electron* or a *jet*. *Particles* inherit the *eventid* from the table *Events*, which are unique keys from *Events*, and *Particles* receive it as a foreign key. *Particles* has *idap* as primary key.

For *BigTable* views could be used that select the *particles* depending of their type.

The following is the SQL schema definition code to define a *BigTable* schema:

```
CREATE Table Events (
idevent INT IDENTITY(1,1) primary key,
id INT not null,
PxMiss Real not null,
PyMiss Real not null,
filenames Varchar(50) not null);
```

```
CREATE Table Particles(
idap INT IDENTITY(0,1) primary key,
id INT not null,
Eventid INT not null,
Px Real not null,
Py Real not null,
Pz Real not null,
Kf Real not null,
Ee Real not null,
typ int not null);
constraint chk_typ check (typ in (1,2,3))
Constraint particleId FOREIGN KEY (eventid)
REFERENCES Events (idevent) ON DELETE CASCADE;


create view Jets
As
select idap,id,eventid,px,py,pz,kf,ee
from Particles
where typ=1;

create view Leptons
As
select idap,id,Eventid,px,py,pz,kf,ee
from Particles
where typ=2 or typ=3;

create view Muons
As
select idap,id,eventid,px,py,pz,kf,ee
from Particles
where typ=2;

create view Electrons
As
select idap,id,eventid,px,py,pz,kf,ee
from Particles
where typ=3;
```

For testing, the number of *events* and *particles* that are going to be introduced in every schema will be the same. Physically all three schemas will present the same quantity of tuples in *Events* and *Particles* tables, but *BigTable* will present an extra attribute in each one of the tuples. Also for the *RepeatID* schema the id attribute will be repeated twice for lepton (*muons* and *electrons*) and once for *jetbs*. For the *DuplicateData* schema all *particles* data will be repeated twice for *leptons* (*muons* and *electrons*) and once for *jetbs*.

..

For the moment physical schemas only have the default indexing provided by SQL-Server. It is recommended for future work to study the impact of further to indexing.

3.1.2 Implementation Dimension.

This dimension deals with how data is accessed, searched and evaluated.

The *query implementation* dimension has two kinds of solutions. One of them has the queries implemented as views and the other one has them implemented in functions. In both schemas, the numerical formulas like *Pt* or *Eta* and others are expressed in scalar functions.

The advantage of functions is that it is more natural and simple to express numerical formulas. MSSQL2005 provides the possibility to create Inline table-valued functions, which allow queries to return a table as result. Using views is more complicated to write because it is not possible to parameterize them. The ability for a function to act as a table gives developers the posibility to break out complex logic into short code blocks, this will generally give the additional benefit of making the code less complex and easier to write and maintain. In the case of a *Scalar User-Defined Function*, the ability to use this function anywhere helps to use a scalar of the same data type, which is also a very powerful tool [7].

On other hand, Complex queries can be stored in the form of a view, and data from the view can be extracted using simple queries [8]. Views are opened by the optimizer to optimize the entire query including views. This is different from functions, which are kept closed.

3.1.2.1 Views

With this implementation, a better optimization is expected for faster execution times, because views can encapsulate very complex calculations and commonly used joins. [9].

The following is the code implemented to define the views:

```
/*******************************************************************/
/**
 * Event should have exactly three isolated leptons with pt above
 * minPtOfAllThreeLeptons (7 GeV), one of them should have pt above
 * minPtOfTheHardestLepton (20 GeV), at the same time all of them
 * should have eta within etaRangeForAllThreeLeptons (2.4).
 */

/*
 * TTreeCut::ThreeLeptonCut, m_isolatedLeptons, allLeptonsWithinEtaRange
 * m_minPtOfAllThreeLeptons: minPtL
 * m_etaRangeForAllThreeLeptons: etaL
 */

create view isolatedLeptons
AS
select l.*
from Leptons as l
where dbo.pt(l.px,l.py) > 7.0  and
abs(dbo.eta(l.px,l.py,l.pz))<2.4;

/**
 * minPtOfAllThreeLeptons: minPtL
 * minPtOfTheHardestLepton: hardPtL
 * etaRangeForAllThreeLeptons: etaL
 */

create view ThreeLeptonCut
AS
select  e.*
from Events e
where exists (   select i.*
                 from isolatedleptons i
                 where   i.eventid = e.idevent  and
                         dbo.pt(i.px,i.py)>20.0 and
                         e.idevent in(     select l.eventid
                                           from isolatedleptons l
                                           group by l.eventid
                                           having count(l.id)=3));

/*******************************************************************/
```

```
/**
 * the event which has two opposite charged leptons with invariant
 * mass closed to the Z mass should be cutted away.
 * Differences between invariant mass of any two opposite charged
 * leptons and m_zMass should be bigger or equal to m_minimumZMassDiff.
 * we should look to pairs electron - positron and muon - antimuon.
 */

create view oppositeLeptons
AS
select   distinct l1.px as l1px, l1.py as l1py,
         l1.pz as l1pz, l1.ee as l1ee,
         l2.px as l2px, l2.py as l2py,
         l2.pz as l2pz, l2.ee as l2ee,
          l1.eventid
from Leptons as l1, Leptons as l2
where l1.kf = -l2.kf and l1.eventid = l2.eventid;

/*
 * m_zMass: zMass
 * m_minimumZMassDiff: minZMass
 */

create view EvInvMass
As
select j.eventid
from oppositeleptons j
where dbo.invmass(     j.l1Ee + j.l2Ee,j.l1px + j.l2px,
                       j.l1py + j.l2py,j.l1pz + j.l2pz,
                       91.1882)<10;

create view zVetoCut
AS
select *
from Events
where idevent not in (select eventid from evInvMass);


/************ HadronicTopCut *******************************************/
/**
 * Events must have at least three jets with pt > 20 GeV and eta within 4.5.
 * Three of them most likely to form the three-jet system and to come
 * from the top quark, which means that invariant mass of the three-jet
 * system is close to 174.3 within 35. Two jets from the three-jet system
 * most likely to come from the W boson, which means that invariant mass
 * of the two jets is close to 80.419 within 15. The third jet from the
 * three-jet system has to be tagged as a b-jet.
 */

/*
 * TTreeCut::SelectOkJets, m_okJets
 * Selects jets (with AtlfastB to) which are ok
 * m_etaRangeForJets: etaJ
 * m_minPtForJets: minPtJ
```

```
 */

create view okJets
AS
select *
from Jets as j1
where ( select count(j2.id)
          from Jets as j2
          where   abs(dbo.eta(j2.px,j2.py,j2.px)) < 4.5
                  and dbo.pt(j2.px,j2.py) > 20.0
                  and j1.eventid=j2.eventid
          )>= 3
          and abs(dbo.eta(j1.px,j1.py,j1.px))<4.5
          and dbo.pt(j1.px,j1.py) > 20.0;

/*
 * TTreeCut::SeperateBJets, m_okBJets
 * Select b jets from jets (with AtlfastB to) of event
 * function getPdg is Kfjetb from TTreeClass here
 * m_theIntegerForBTaggedJet: forBJet
 */

create view bjets
as
select j.*
from okjets as j
where j.kf = 5

/*
 * TTreeCut::SeperateBJets, m_okWJets
 * Select wJets from jets (with AtlfastB to) of event.
 * They are ok and not bJets.
 */

create view wjets
AS
select j.*
from okjets as j
where j.kf != 5

/*
 * TTreeCut::Select2WCombinations, m_okWComb
 * select 2W combinations
 * returns vectors of two wJets which satisfy invariant mass condition
 * m_wMass: wMass
 * m_allowedWMassDiff: allowedWMass
 */

create view wPairs
as
select   j1.eventid as jid, j1.idap as j1idap, j1.id as j1id,
         j1.Ee as j1Ee, j1.Px as j1Px, j1.Py as j1Py,
         j1.pz as j1pz, j2.idap as j2idap, j2.id as j2id,
         j2.Ee as j2Ee, j2.Px as j2Px, j2.Py as j2Py,
```

```
          j2.pz as j2pz
from wJets as j1, wJets as j2
where   dbo.invmass(    j1.Ee + j2.Ee, j1.px + j2.px,
                        j1.py + j2.py, j1.pz + j2.pz,
                        80.419)<15.0
        and j1.eventid = j2.eventid
        and j1.id > j2.id;

/*
 * TTreeCut::SelectTopCombination, m_okTopComb
 * m_topMass: tMass
 * m_allowedTopMassDiff: allowedTMass
 */

create view topComb
As
select  j.*, b.*
from wPairs as j, bJets as b
where   dbo.invmass(    j.j1Ee + j.j2Ee + b.Ee,
                        j.j1px + j.j2px + b.px,
                        j.j1py + j.j2py + b.py,
                        j.j1pz + j.j2pz + b.pz,174.3)<35.0
        and j.jid=b.eventid;

/**
 * Hardronic Top Cut 2 (see management file)
 *** OBS!do not forget that it should be at least 3 ok jets
 */

create view TopCut
AS
select distinct e.*
from topComb t, Events e
where e.idevent=t.eventid;


/*********************************************************************/
/* Jet Veto Cut 2
 * leftJets jetbs should have Pt not bigger then maxAllowedPtForOtherJets
 * see Hadronic Top Cut 2
 * m_maxAllowedPtForOtherJets: ptOJets
 */
/*
 * TTreeCut::SelectTopCombination, m_theTopComb
 * min of m_okTopComb
 */

create view mTopComb
As
select j.*
from topComb as j
where ( abs(sqrt(abs(    (j.j1Ee+j.j2Ee + j.Ee)*(j.j1Ee+j.j2Ee +j.Ee) -
                         ((j.j1px +j.j2px + j.px)*(j.j1px +j.j2px + j.px) +
                         (j.j1py +j.j2py + j.py)*(j.j1py +j.j2py + j.py) +
```

```
                                   (j.j1pz +j.j2pz + j.pz)*(j.j1pz +j.j2pz + j.pz))))
                                   - 174.3))
                                   =
                                   (select min(abs(sqrt(abs((t.j1Ee+t.j2Ee +
                                               t.Ee)*(t.j1Ee+t.j2Ee +t.Ee) -
                                              ((t.j1px +t.j2px + t.px)*(t.j1px +t.j2px + t.px) +
                                              (t.j1py +t.j2py + t.py)*(t.j1py +t.j2py + t.py) +
                                              (t.j1pz +t.j2pz + t.pz)*(t.j1pz +t.j2pz + t.pz))))
                                              - 174.3))
                                   from topComb as t
                                   where t.eventid=j.eventid)

/*
 * TTreeCut::SelectTopCombination, m_theLeftOverJets
 * select m_okJets which are not contained in m_theTopComb
 */

create view leftjets
As
select distinct o.*
from okJets as o
where not exists (select o.idap
                  from mtopcomb as j
                  where   j.idap=o.idap or
                          j.j1idap=o.idap or
                          j.j2idap=o.idap);

create view JetVetoCut
AS
select distinct e.*
from Events e
where not exists(select *
                  from leftjets j
                  where e.idevent=j.eventid and
                  dbo.pt(j.px,j.py)>70);


/*
 * Other cuts
 * 1. All isolated leptons should has Pt not bigger then maxPtAll
 * 2. Isolated lepton which has smallest Pt should have Pt not bigger
 *    then maxPtSoft
 * m_isolatedLeptons: isolatedLeptons(event,parameters)->leptons
 * m_maxPtForAllThreeIsolatedLeptons: maxPtAll
 * m_maxPtForTheSoftestIsolatedLepton: maxPtSoft
 */

create view LeptonCuts
AS
select q.*
from Events q
where ( not exists(        select j.eventid
                           from isolatedLeptons as j
                           where   dbo.pt(j.px,j.py)>150.0 and
```

```
                                        q.idevent=j.eventid
                                )
                        and
                        exists ( select i.eventid
                                from isolatedLeptons as i
                                where   dbo.pt(i.px,i.py)<=40 and
                                        q.idevent=i.eventid)
        );

/********************************************************************/
* Other cuts, continue*
 * 1. Missing traverse energy (mod(PtMiss)) should be not smaller
 *    then minTransEe
 * 2. Effective mass should be not bigger then maxEfMass
 * m_minMissingTransverseEnergy: minTransEe
 * m_maxAllowedEffectiveMass: maxEfMass
 * ptMiss={PxMiss,PyMiss}
 * pt31=sum(Px(isolated lepton),Py(isolated lepton))
 */

create view MissEeCuts
as
select distinct e.*
from Events e
where exists (
        select l.eventid
        from isolatedLeptons l
         where e.idevent=l.eventid
        group by l.eventid
         having
        dbo.module(e.PxMiss,e.PyMiss)>=40 AND
        dbo.effectiveMass(e.PxMiss,e.PyMiss,sum(l.px),sum(l.py)) <= 150.0);


/*****************************************************************/
/**
 * All cuts together!
 */

create view allcuts
AS
select  th.idevent, th.filenames, th.id
from    ThreeLeptonCut th, zVetoCut z, TopCut tp,
        JetVetoCut j, LeptonCuts l, MissEeCuts m
where   th.idevent=z.idevent and
        z.idevent=tp.idevent and
        tp.idevent=j.idevent and
        j.idevent=l.idevent and
        l.idevent=m.idevent;

create view optallcuts
AS
select  th.idevent,th.filenames,th.id
```

```
from     ThreeLeptonCut th,LeptonCuts l,MissEeCuts m,
         zVetoCut z, TopCut tp,JetVetoCut j
where    th.idevent=l.idevent and
         l.idevent=m.idevent and
         m.idevent=z.idevent and
         z.idevent=tp.idevent and
         tp.idevent=j.idevent;

create  view expcuts
AS
select   tp.idevent,tp.filenames,tp.id
from     TopCut tp, JetVetoCut j, MissEeCuts m,
         zVetoCut z, ThreeLeptonCut th,LeptonCuts l
where    tp.idevent=j.idevent and
         j.idevent=m.idevent and
         m.idevent=z.idevent and
         z.idevent=th.idevent and
         th.idevent=l.idevent;
```

3.1.2.2 Functions

With this implementation, a more natural way is used to write the queries, which at the same time, is easier to manipulate, and also permits directly managing the data needed; but, on the other hand, the query optimizer treats functions as black boxes, which reduce efficiency of the query optimization.

The following is the code implemented to define the functions:

```
/*******************************************************************/
/**
 * Event should have exactly three isolated leptons with pt above
 * minPtOfAllThreeLeptons (7 GeV), one of them should have pt above
 * minPtOfTheHardestLepton (20 GeV), at the same time all of them
 * should have eta within etaRangeForAllThreeLeptons (2.4).
 */

/*
 * TTreeCut::ThreeLeptonCut, m_isolatedLeptons, allLeptonsWithinEtaRange
 * m_minPtOfAllThreeLeptons: minPtL
 * m_etaRangeForAllThreeLeptons: etaL
 */

create function isolatedLeptons
         (@idevent INT)
Returns TABLE
AS
Return  select l.*
         from Lepton as l
```

```
            where   @idevent = l.eventid
                    and dbo.pt(l.px,l.py) > 7.0
                    and abs(dbo.eta(l.px,l.py,l.pz))<2.4;

/**
 * minPtOfAllThreeLeptons: minPtL
 * minPtOfTheHardestLepton: hardPtL
 * etaRangeForAllThreeLeptons: etaL
 */

create function ThreeLeptonCut
            (@idevent INT)
Returns bit
AS
BEGIN
if(         exists  (select  a.*
                    from isolatedleptons(@idevent) as a
                    where dbo.pt(a.px,a.py)>20.0)
            and (    (select count(i.id)
                    from isolatedleptons(@idevent) as i)=3)
            )
return 1

return 0
END


/**
 * the event which has two opposite charged leptons with invariant
 * mass closed to the Z mass should be cutted away.
 * Differences between invariant mass of any two opposite charged
 * leptons and m_zMass should be bigger or equal to m_minimumZMassDiff.
 * we should look to pairs electron - positron and muon - antimuon.
 */

create function oppositeLeptons
            (@idevent INT)
Returns TABLE
AS
Return  select  l1.px as l1px, l1.py as l1py,
                l1.pz as l1pz, l1.ee as l1ee,
                l2.px as l2px, l2.py as l2py,
                l2.pz as l2pz, l2.ee as l2ee, l1.eventid
        from Leptons as l1, Leptons as l2
        where   l1.kf = -l2.kf
                and l1.eventid = @idevent
                and l1.eventid=l2.eventid;

/*
 * m_zMass: zMass
 * m_minimumZMassDiff: minZMass
 */

create function zVetoCut
```

```
        (@idevent INT)
Returns bit
As
Begin
if ( not exists(
        select *
        from oppositeleptons(@idevent) j
        where dbo.invmass(      j.l1Ee + j.l2Ee,j.l1px + j.l2px,
                                j.l1py + j.l2py,j.l1pz + j.l2pz,
                                91.1882)<10))
return 1
return 0
END


/**********************************************************************/
/************ HadronicTopCut ********************************************/
/**
 * Events must have at least three jets with pt > 20 GeV and eta within 4.5.
 * Three of them most likely to form the three-jet system and to come
 * from the top quark, which means that invariant mass of the three-jet
 * system is close to 174.3 within 35. Two jets from the three-jet system
 * most likely to come from the W boson, which means that invariant mass
 * of the two jets is close to 80.419 within 15. The third jet from the
 * three-jet system has to be tagged as a b-jet.
 */

/*
 * TTreeCut::SelectOkJets, m_okJets
 * Selects jets (with AtlfastB to) which are ok
 * m_etaRangeForJets: etaJ
 * m_minPtForJets: minPtJ
 */

create function okJets
        (@idevent INT)
Returns Table
AS RETURN(
select *
from Jets
where ( select count(id)
        from Jets
        where   eventid = @idevent
                and abs(dbo.eta(px,py,pz)) < 4.5
                and dbo.pt(px,py) > 20.0) >= 3
        and eventid = @idevent
        and abs(dbo.eta(px,py,pz))<4.5
        and dbo.pt(px,py) > 20.0)

/*
 * TTreeCut::SeperateBJets, m_okBJets
 * Select b jets from jets (with AtlfastB to) of event
 * function getPdg is Kfjetb from TTreeClass here
```

```
 * m_theIntegerForBTaggedJet: forBJet
 */

create function bjets
          (@idevent INT)
Returns Table
as
return   select j.*
          from okjets(@idevent) as j
          where   j.kf = 5
                  and j.eventid = @idevent

/*
 * TTreeCut::SeperateBJets, m_okWJets
 * Select wJets from jets (with AtlfastB to) of event.
 * They are ok and not bJets.
 */

create function wjets
          (@idevent INT)
Returns Table
as
return   select j.*
          from okjets(@idevent) as j
          where   j.kf != 5
                  and j.eventid = @idevent

/*
 * TTreeCut::Select2WCombinations, m_okWComb
 * select 2W combinations
 * returns vectors of two wJets which satisfy invariant mass condition
 * m_wMass: wMass
 * m_allowedWMassDiff: allowedWMass
 */

create function wPairs
          (@idevent INT)
Returns Table
as
Return
Select   j1.eventid as jid, j1.idap as j1idap, j1.id as j1id,
          j1.Ee as j1Ee, j1.Px as j1Px, j1.Py as j1Py,
          j1.pz as j1pz, j2.idap as j2idap, j2.id as j2id,
          j2.Ee as j2Ee, j2.Px as j2Px, j2.Py as j2Py,
          j2.pz as j2pz
from wJets(@idevent) as j1, wJets(@idevent) as j2
where   dbo.invmass(    j1.Ee + j2.Ee, j1.px + j2.px,
                        j1.py + j2.py, j1.pz + j2.pz,
                        80.419)<15.0
          and j1.id > j2.id;

/*
 * TTreeCut::SelectTopCombination, m_okTopComb
 * m_topMass: tMass
```

```
 * m_allowedTopMassDiff: allowedTMass
 */

create function topComb
        (@idevent INT)
returns table
As
Return
select  j.*, b.*
from wPairs(@idevent) as j, bJets(@idevent) as b
where dbo.invmass(      j.j1Ee + j.j2Ee + b.Ee,
                        j.j1px + j.j2px + b.px,
                        j.j1py + j.j2py + b.py,
                        j.j1pz + j.j2pz + b.pz,174.3)<35.0

/**
 * Hardronic Top Cut 2 (see management file)
 *** OBS!do not forget that it should be at least 3 ok jets
 */

create function TopCut
        (@idevent INT)
Returns bit
AS
BEGIN
if(exists(      select *
                from topComb(@idevent))
return 1
return 0
END


/*********************************************************************/
/* Jet Veto Cut 2
 * leftJets jetbs should have Pt not bigger then maxAllowedPtForOtherJets
 * see Hadronic Top Cut 2
 * m_maxAllowedPtForOtherJets: ptOJets
 */
/*
 * TTreeCut::SelectTopCombination, m_theTopComb
 * min of m_okTopComb
 */

create function mTopComb
        (@idevent INT)
returns table
As
Return  select j.*
        from topComb(@idevent) as j
        where (abs(sqrt(abs(    (j.j1Ee+j.j2Ee + j.Ee)*(j.j1Ee+j.j2Ee +j.Ee) -
                                ((j.j1px +j.j2px + j.px)*(j.j1px +j.j2px + j.px) +
                                (j.j1py +j.j2py + j.py)*(j.j1py +j.j2py + j.py) +
                                (j.j1pz +j.j2pz + j.pz)*(j.j1pz +j.j2pz + j.pz))))
                                - 174.3))
```

```
                                        =
                                (select min(abs(sqrt(abs((t.j1Ee+t.j2Ee +
                                t.Ee)*(t.j1Ee+t.j2Ee +t.Ee) -
                                        ((t.j1px +t.j2px + t.px)*(t.j1px +t.j2px + t.px) +
                                        (t.j1py +t.j2py + t.py)*(t.j1py +t.j2py + t.py) +
                                        (t.j1pz +t.j2pz + t.pz)*(t.j1pz +t.j2pz + t.pz))))
                                        - 174.3))
                                        from topComb(@idevent) as t)

/*
 * TTreeCut::SelectTopCombination, m_theLeftOverJets
 * select m_okJets which are not contained in m_theTopComb
 */

create function leftjets
        (@idevent INT)
returns table
As
return
select distinct o.*
from okJets(@idevent) as o
where not exists (select o.idap
                from mtopcomb(@idevent) as j
                where j.idap=o.idap
                or j.j1idap=o.idap
                or j.j2idap=o.idap);

create function JetVetoCut
        (@idevent INT)
Returns bit
AS
BEGIN
if(not exists(  select *
                from leftjets(@idevent) j
                where dbo.pt(j.px,j.py)>70))
return 1
return 0
END


/*********************************************************************/
/*
 * Other cuts
 * 1. All isolated leptons should has Pt not bigger then maxPtAll
 * 2. Isolated lepton which has smallest Pt should have Pt not bigger
 *    then maxPtSoft
 * m_isolatedLeptons: isolatedLeptons(event,parameters)->leptons
 * m_maxPtForAllThreeIsolatedLeptons: maxPtAll
 * m_maxPtForTheSoftestIsolatedLepton: maxPtSoft
 */

create function LeptonCuts
        (@idevent INT)
Returns bit
```

```
AS
BEGIN
if(        not exists(        select j.*
                              from isolatedLeptons(@idevent) as j
                              where dbo.pt(j.px,j.py)>150.0)
           and exists (       select *
                              from isolatedLeptons(@idevent) as i
                              where dbo.pt(i.px,i.py)<=40))
return 1
return 0
END


/*
 * Other cuts, continue*
 * 1. Missing traverse energy (mod(PtMiss)) should be not smaller
 *    than minTransEe
 * 2. Effective mass should be not bigger then maxEfMass
 * m_minMissingTransverseEnergy: minTransEe
 * m_maxAllowedEffectiveMass: maxEfMass
 * ptMiss={PxMiss,PyMiss}
 * pt31=sum(Px(isolated lepton),Py(isolated lepton))
 */

create function MissEeCuts
          (@idevent real,@pxm real,@pym real)
Returns bit
AS
BEGIN
if exists (select l.eventid
          from isolatedLeptons(@idevent) l
          group by l.eventid
          having  dbo.module(@PxM,@PyM)>=40
                  and  dbo.effectiveMass(@PxM,@pyM,sum(l.px),  sum(l.py))<=
150.0
          )
return 1
return 0
END


 /*******************************************************************/
 /**
  * All cuts together!
  */

 create view allcuts
 AS
 select ev.*
 from Events ev
 where   dbo.ThreeLeptonCut(ev.idevent)=1
         and dbo.zVetoCut(ev.idevent)=1
         and dbo.TopCut(ev.idevent)=1 and dbo.JetVetoCut(ev.idevent) = 1
         and dbo.LeptonCuts(ev.idevent)=1
```

```
        and dbo.MissEeCuts(ev.idevent,ev.pxmiss,ev.pymiss)=1;

create view optallcuts
AS
select ev.*
from Events ev
where   dbo.ThreeLeptonCut(ev.idevent)=1
        and dbo.LeptonCuts(ev.idevent)=1
        and dbo.MissEeCuts(ev.idevent,ev.pxmiss,ev.pymiss)=1
        and dbo.zVetoCut(ev.idevent)=1
        and dbo.TopCut(ev.idevent)=1 and dbo.JetVetoCut(ev.idevent) = 1;

create view expcuts
AS
select ev.*
from Events ev
where   dbo.TopCut(ev.idevent)=1
        and dbo.JetVetoCut(ev.idevent) = 1
        and dbo.MissEeCuts(ev.idevent,ev.pxmiss,ev.pymiss)=1
        and dbo.zVetoCut(ev.idevent)=1
        and dbo.ThreeLeptonCut(ev.idevent)=1
        and dbo.LeptonCuts(ev.idevent)=1;
```

3.2 Scalar Functions for Numerical Formulas

Additionally, some scalar functions were defined to calculate numerical results from formulas that are needed for the *cuts* in both implementations. These functions are called from the cuts with the parameters needed for the formulas and return a numerical scalar result that will be needed in the cut from they was called.

The functions and the code used to define them are the following:

```
/* Pt */
create function pt
        (@px real, @py real)
Returns Real
AS
BEGIN
        Return ( select sqrt(@px*@px + @py*@py))
END

/* ETA */
create function ETA
        (@px real,@py real, @pz real)
```

```
Returns Real
AS
BEGIN
Return  (select 0.5*log(((sqrt(@px*@px + @py*@py + @pz*@pz)) + @pz) /
        ((sqrt(@px*@px + @py*@py + @pz*@pz)) - @pz)))
END


/* phi */
create function phi
        (@fx Real, @fy real)
returns Real
As
begin
return atn2(-@fx,-@fy) + pi();
END


/*phi_mpi_pi*/
create function phi_mpi_pi
        (@x real)
returns real
AS
begin
return @x + ceiling((-1.0/2.0)-@x/(2.0*pi()))*2*pi()
END


/*effectiveMass*/
create function effectiveMass
(@xMiss Real,@yMiss Real, @x31 Real,@y31 Real)
returns Real
AS
begin
return    sqrt(abs(2.0*((@xMiss*@x31)+(@yMiss*@y31))*
        (1-cos(dbo.phi_mpi_pi(dbo.phi(@x31,@y31)-
dbo.phi(@xMiss,@yMiss))))))
END


/*Mod Of Vector*/
create function module
        (@v1 Real,@v2 Real)
returns real
As
begin
return sqrt(@v1*@v1+@v2*@v2)
END


create function invmass
        (@ee real, @px real, @py real, @pz real, @r real)
returns real
AS
```

```
begin
return abs(sqrt(abs(      (@ee)*(@ee) - ((@px)*(@px) +
                          (@py)*(@py) + (@pz)*(@pz)))) - @r)
END
```

4. Performance Evaluation

The three schema dimensions with the two implementation dimensions were combined, in other to get six different scenarios and take the one that is best for the task that we want to solve. These six experimental scenarios are:

1- RepeatID functions.
2- RepeatID views.
3- DuplicateData functions.
4- DuplicateData views.
5- BigTable functions.
6- BigTable views.

4.1 Setup Process

All three schemas were configured with using MSSQL2005 with some SQL-2003 features. Then, HEP data were uploaded to them.

First a sample of 101 *events* was loaded in order to test S3RDB for every schema. Once knowing that S3RDB worked correctly with the small sample, the rest of the data was loaded to the application. All data had a total of 25000 *events*.

4.2 Import data times:

Here are the different executions times to import data to the application, then they are showed. Due to the differences of the constructions of the data base schemas made for S3RDB application, the import data times were also

compared. *Events*, *jets*, *muons* and *electrons* are imported by different queries; they were called *FillEvent*, *FillJetb*, *FillMuon* and *FillElectron* respectively. *FillEvent* is the same for the three schemas, but *FillJetb*, *FillMuon* and *FillElectron* have differences in the code in order to fix the data correctly.

For every scenario the data is loaded separately. In order to compare how long it takes for data to be loaded in every case.

***FillEvent*:** Query used to import all *events* and their attributes.

For every escenario the same *FillEvent* query is implemented, so mostly the differences between loading times in the different schemas will be due to how data is stored in the physical database schema.

```
/*Query to import events to sql server
*****************************************************************/

Insert into Events (PxMiss,PYMiss,filenames,Id)
Values( ?,?,?,?);
```

The following table presents loading times in seconds to import *events* with *FillEvent* to the different scenarios:

Data Representation	Number of events	Implementation	Load Time (sec)
RepeatID	101	Functions	1.112
RepeatID	101	Views	0.421
DuplicateData	101	Functions	0.36
DuplicateData	101	Views	0.34
BigTable	101	Functions	0.32
BigTable	101	Views	0.37
RepeatID	25000	Functions	107.014
RepeatID	25000	Views	97.08
DuplicateData	25000	Functions	130.327
DuplicateData	25000	Views	88.787
BigTable	25000	Functions	112.852
BigTable	25000	Views	196.622

Table 1. Import times to import *events* to SQL Server

With small quantities of data (101 *events*), there are no large differences between *DuplicateData* and *BigTable*, but *RepeatID* is quite slower for *views* and three times slower for *functions*.

For large quantities of data (25000 *events*) there are some significant differences between the implementations, but *BigTable views* takes nearly double the time.

FillJetb: Query to import *jetb* and their attributes.

Code for *FillJetb* is different depending on into which schema the data will be loaded. On *RepeatID*, all *jetbs* data need to be loaded once in *Particles* table, and then the *ids* are repeated for the *jetaux* table. For *DuplicateData* all *jetb* data needs to be loaded twice, once for *Particles* table and once for *Jets* table. For *BigTable* data is loaded just once in the *Particles* table, but adding the value 1 in the attribute *type* to indicate that a *particle* is a *jetb*.

```
/*Query to import jetbs  to sql server RepeatID
*************************************************************/

declare @q as int;

set @q = (select max(idevent) from Events where id =
?);

Insert into Particles(id,eventid,px,py,pz,kf,ee)
Values(?,@q,?,?,?,?,?);

Insert into jetaux(id)
Values (SCOPE_IDENTITY());
```

```
/*Query to import jetbs to sql server DuplicateData
*************************************************************/

declare @q as int;
set @q = (select max(idevent) from Events where id =
?);

Insert into Particles(id,eventid,px,py,pz,kf,ee)
Values(?,@q,?,?,?,?,?);

Insert into Jets(idap,id,eventid,px,py,pz,kf,ee)
Values (SCOPE_IDENTITY(),?,@q,?,?,?,?,?);
```

```
/*Query to import jetbs to sql server BigTable
****************************************************************/
declare @q as int;

set @q = (select max(idevent) from Events where id =
?);

Insert into Particles(id,eventid,px,py,pz,kf,ee,typ)
Values(?,@q,?,?,?,?,?,1);
```

The following table presents loading times in seconds to import *jetbs* with *FillJetb* to the different schemas:

Data Representation	Number of events	Implementation	Load Time (sec)
RepeatID	101	Functions	5.758
RepeatID	101	Views	4.957
DuplicateData	101	Functions	4.816
DuplicateData	101	Views	4.687
BigTable	101	Functions	3.375
BigTable	101	Views	3.545
RepeatID	25000	Functions	1495.33
RepeatID	25000	Views	1108.67
DuplicateData	25000	Functions	4288.82
DuplicateData	25000	Views	6684.37
BigTable	25000	Functions	4647.12
BigTable	25000	Views	4614.12

Table 2. Times to import *jetbs* to SQL Server

Loading times for small data quantities (101 events) have no big differences, but for *BigTable* the loading is a little bit faster than for *DuplicateData*. Furthermore, *RepeatID* is a little bit slower than *DuplicateData.*

For larger amounts of data (25000 events) *DuplicateData* and *BigTable* are three times slower than *RepeatID,* but for *DuplicateData* the difference is more significant (six times slower); however,, results need probably to be revised.

FillJetb is the function that takes more time to execute because *jetB's* are the more predominant *particles* in the data given.

FillMuon is a query to import all *muons* and their attributes.

Code for *FillMuon* is different depending on into which schema the data will be uploaded. On *RepeatID*, all *muons* data needs to be loaded once in the *Particles* table, and then the *ids* repeated once for *Leptons* table and once for *Muons* table. For *DuplicateData* all *muons* data needs to be loaded three times, once for *Particles* table, once for *Leptons* table, and once for *Muons* table. And for *BigTable* data is loaded just once in the *Particles* table adding the value 2 in the attribute *type* to indicate that *particle* is a *muon*.

```
/*Query to import muons to sql server RepeatID
*************************************************************/
declare @q as int;

set @q = (select max(idevent) from Events where id = ?);

Insert into Particles (id,eventid,px,py,pz,kf,ee)
Values(?,@q,?,?,?,?,?);

Declare @ID as int; Set @ID=SCOPE_IDENTITY();

Insert into leptonaux(id)
VALUES (@ID);

Insert into muonaux(id)
VALUES (@ID);


/*Query to import muons to sql server DuplicateData
*************************************************************/

declare @q as int;

set @q = (select max(idevent) from Events where id = ?);

Insert into Particles (id,eventid,px,py,pz,kf,ee)
VALUES(?,@q,?,?,?,?,?);

Declare @ID as int; Set @ID=SCOPE_IDENTITY();

Insert into Leptons(idap,id,eventid,px,py,pz,kf,ee)
VALUES (@ID,?,@q,?,?,?,?,?);

Insert into Muons(idap,id,eventid,px,py,pz,kf,ee)
VALUES (@ID,?,@q,?,?,?,?,?);


/*Query to import muons to sql server BigTable
*************************************************************/
declare @q as int;
```

```
set @q = (select max(idevent) from Events where id = ?);

Insert into Particles(id,eventid,px,py,pz,kf,ee,typ)
Values(?,@q,?,?,?,?,?,2);
```

The following table presents loading times in seconds to import *muons* with *FillMuon* to the different schemas:

Data Representation	Number of events	Implementation	Load Time (sec)
RepeatID	101	Functions	0.251
RepeatID	101	Views	0.36
DuplicateData	101	Functions	0.351
DuplicateData	101	Views	0.27
BigTable	101	Functions	0.151
BigTable	101	Views	0.16
RepeatID	25000	Functions	42.241
RepeatID	25000	Views	39.326
DuplicateData	25000	Functions	129.246
DuplicateData	25000	Views	202.842
BigTable	25000	Functions	139.491
BigTable	25000	Views	123.017

Table 3. Times to import *muons* to SQL Server

In the case of small quantities of data (101 events) differences are not really significant, but *BigTable* is a little bit faster than the other two schemas.

For big quantities of data (25000 *events*) *RepeatID* shows the fastest times. *DuplicateData* shows incongruent results between *views* and *functions,* because they should be similar in time, since both of them use exactly the same function to load the *muon*'s data to them.

FillElectron: Query used to import all *electrons* and their attributes.

Code for *FillElectron* is different depending on into which schema the data will be uploaded. For *RepeatID* all *electrons* data needs to be loaded once in the *Particles* table, and then the *ids* repeated once for *Leptons* table and once for *Electrons* table. For *DuplicateData* all *electrons* data needs to be loaded three times, once for *Particles* table, once for *Leptons* table, and once for *Electrons*

table. For *BigTable* data is loaded just once in the *Particles* table but adding the value 3 in the attribute type to indicate that *particle* is an *electron*.

```
/*Query to import electrons to sql server DuplicateData
************************************************************/
declare @q as int;

set @q = (select max(idevent) from Events where id =
?);

Insert into Particles (id,eventid,px,py,pz,kf,ee)
Values (?,@q,?,?,?,?,?);

Declare @ID as int; Set @ID=SCOPE_IDENTITY();

Insert into Leptons(idap,id,eventid,px,py,pz,kf,ee)
Values (@ID,?,@q,?,?,?,?,?);

Insert into Electrons(idap,id,eventid,px,py,pz,kf,ee)
Values (@ID,?,@q,?,?,?,?,?);


/* Query to import electrons to sql server RepeatID
************************************************************/
declare @q as int;

set @q = (select max(idevent) from Events where id =
?);

Insert into Particles (id,eventid,px,py,pz,kf,ee)
VALUES(?,@q,?,?,?,?,?);

Declare @ID as int; Set @ID=SCOPE_IDENTITY();

Insert into leptonaux(id)
VALUES (@ID);

Insert into electronaux(id)
VALUES (@ID);


/* Query to import electrons to sql server BigTable
************************************************************/
declare @q as int;

set @q = (select max(idevent) from Events where id =
?);

Insert into Particles(id,eventid,px,py,pz,kf,ee,typ)
Values(?,@q,?,?,?,?,?,3);
```

The following table presents loading times in seconds to import *electrons* with *FillElectron* to the different schemas:

Data Representation	Number of events	Implementation	Load Time (sec)
RepeatID	101	Functions	0.28
RepeatID	101	Views	0.3
DuplicateData	101	Functions	0.32
DuplicateData	101	Views	0.311
BigTable	101	Functions	0.17
BigTable	102	Views	0.16
RepeatID	25000	Functions	69.556
RepeatID	25000	Views	63.219
DuplicateData	25000	Functions	165.779
DuplicateData	25000	Views	163.795
BigTable	25000	Functions	214.118
BigTable	25000	Views	125.811

Table 4. Times to import *electrons* to SQL Server

For small quantities of data (101 events), differences are not really significant, but *BigTable* is a little bit faster than the other two schemas.

For big quantities of data (25000 events) *RepeatID* shows the fastest times. *BigTable* shows incongruent results between *views* and *functions,* because they should be similar in time, since both of them use exactly the same function to load the *electron*'s data to them.

Total time: This is the sum of the times that every schema took to be imported.

Data Representation	Data Quantity	Implementation	Load Time (sec)
RepeatID	101	Functions	7.401
RepeatID	101	Views	6.038
DuplicateData	101	Functions	5.847
DuplicateData	101	Views	5.608
BigTable	101	Functions	4.016
BigTable	101	Views	4.235
RepeatID	25000	Functions	1,654.141
RepeatID	25000	Views	1,308.295
DuplicateData	25000	Functions	4,714.172
DuplicateData	25000	Views	7,139.794
BigTable	25000	Functions	5,113.581
BigTable	25000	Views	5,059.570

Table 5. Times to import all data to SQL Server

For small data quantities (101 *events*), *BigTable* shows faster times than the other two scenarios, then *DuplicateData* goes in the second place, and the worst times are showed by *RepeatID*.

For large quantities of data (25000 *events*), best times are showed by *RepeatID*, followed by *DuplicateData* (except for *DuplicateData* views), and finally *BigTable* times are a little more than 3.5 times slower than *RepeatID* times.

As we said before, results need to be revised, but at least these ones can give an idea of how loading data times behave for every escenario.

4.3 Execution times

Here, the execution time for every *cut* in every scenario is presented in order to perform comparisons and conclude which scenario is the best choice to use.

Because of the bad time results of the *RepeatID* scheme, this one had not been tested completely, and more focus was given to the *DuplicateData* and *BigTable* results.

ThreeLeptonCut:
This a condition that is fulfilled by *events* that have exactly three *isolated leptons* with *transverse momentum* (the momentum that is transverse to the beamline of a *particle* detector, it is also called *pt*) above 7 gigaelectronvolt (GeV), one of them have *pt* above 20 GeV and at the same time all of them have *eta* range within 2.4 GeV. This *cut* search in all *lepton* data and returns the *events* that fulfill the condition described above.

The following table and graphics shows the times that the *cut* need, depending on the schema used, the number of *events* evaluated and query applied.

Data	Results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	6	4.035	365.190	0.582	0.226	1.045	4.511
1000	17	40.200	871.984	1.947	2.330	5.369	8.078
5000	31	189.655	1446.760	8.636	5.858	48.954	10.291
10000	87	1064.516	2072.010	48.471	9.719	274.774	12.320
15000	153	2808.120	2761.279	127.863	13.929	724.834	16.654
20000	330	8075.641	3224.798	216.301	18.563	2084.491	18.626
25000	475	14530.035	3750.505	389.178	23.696	3750.505	20.456

Table 6 *ThreeLeptonCut* execution time results table

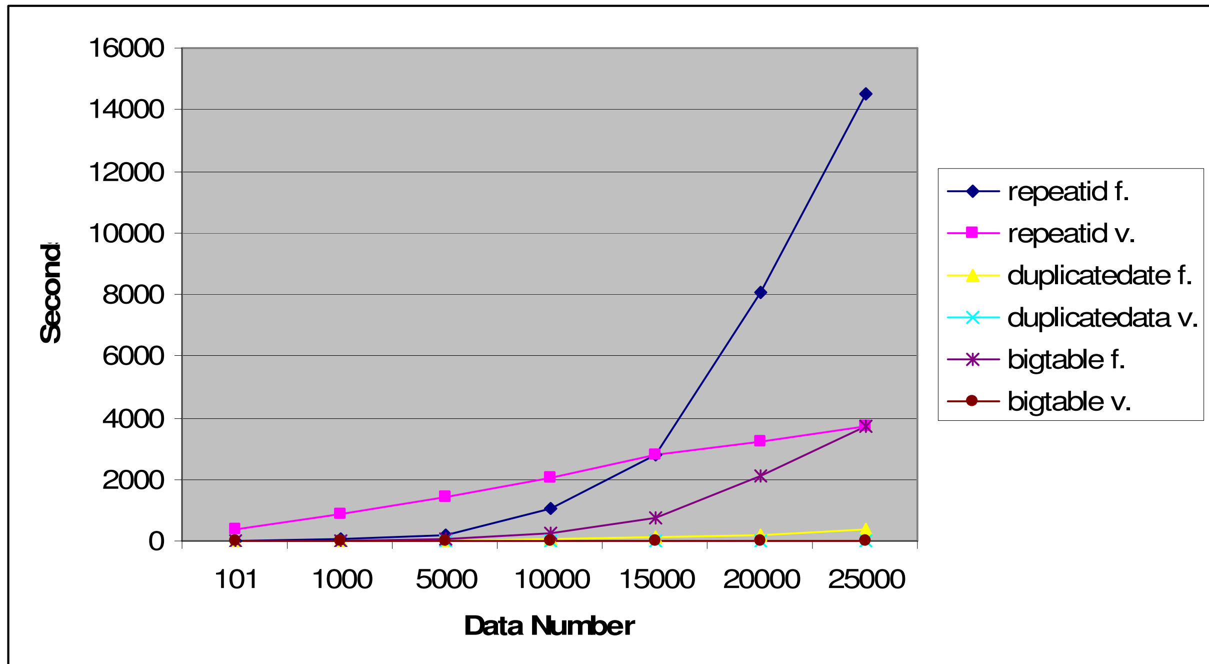

Figure 2: *ThreeLeptonCut* execution time results graphic

Here we can observe that *DuplicateData views* and *BigTable views* have the best performance times, also having a linear growth when more data is evaluated. In general the times for schemas with functions are slow and scale badly. Both evaluations with *RepeatID* schemas show worst performance than the rest.

zVetoCut:

This is a condition that is fulfilled by *events* that have two opposite charged *leptons* with *invariant mass* closed to the *Z mass* should be cut away. Differences between *invariant mass* of any two opposite charged *leptons* and Z mass should

be bigger or equal to *minimum Z mass allowed*. We should look to *electron - positron* and *muon - antimuon* pairs. This cut searches in all *lepton* data and returns the *events* that fulfill the condition described above.

The following table and graphics shows the times that the *cut* need, depending on the schema used, the number of *events* evaluated, and query applied.

Data	Results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	94	5.359	7.366	0.227	0.015	0.997	0.08
1000	931	31.010		1.681	0.075	54.515	0.32
5000	4653	774.908		17.007	0.368	1362.287	0.625
10000	9307	3099.966		68.034	0.914	5449.733	1.305
15000	13961	6975.174		153.082	1.856	12262.339	3.979
20000	19060	12696.972		278.657	4.079	22321.245	5.172
25000	23825	19839.018		435.402	6.354	34876.946	5.451

Table 7 *zVetoCut* execution time results table

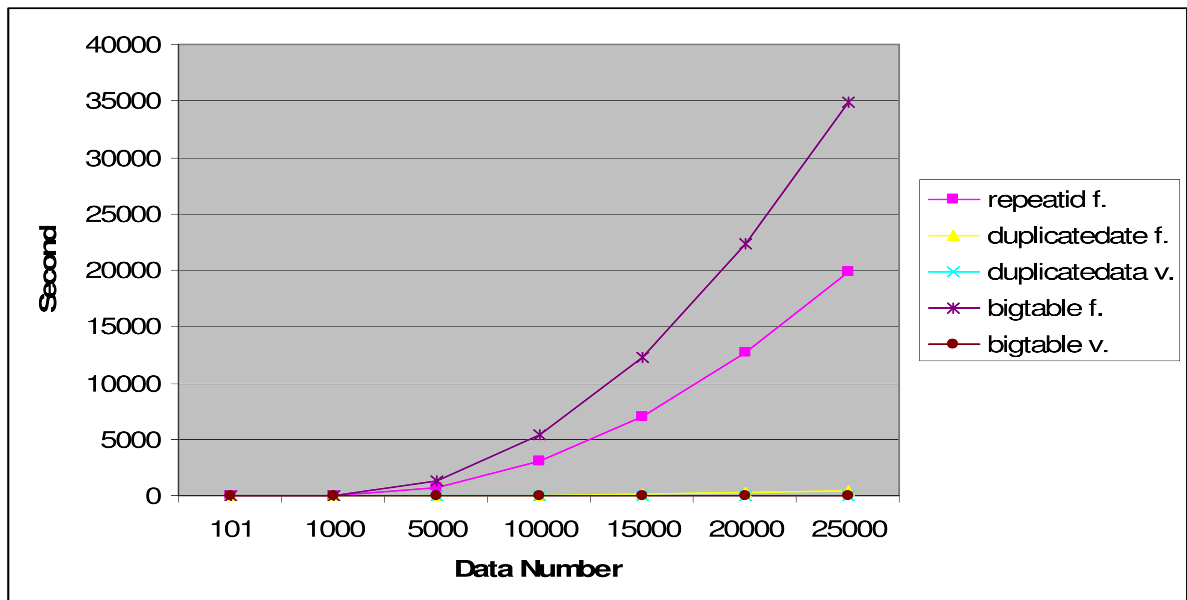

Figure 3: *zVetoCut* execution time results graphic

BigTable views, and *DuplicateData views* shows better scalability. *DuplicateData functions* curve is not as inneficient as the two order functions evaluation, but is still slower in comparison with views evaluations.

TopCut:

HadronicTopCut is fulfilled when an *event* has at least three *jets* with *pt* greater than 20 GeV and *eta* range within 4.5. Three of them most likely to form the three-jet system and to come from the *top quark*, which means that *invariant* mass of the triplet of *jets* is close to 174.3 within 35. Two *jets* from the triplet system most likely to come from the *W boson*, which means that *invariant mass* of the two *jets* is close to 80.419 within 15. The third *jet* from the triplet system has to be tagged as a *b-jet*. This cut search in all *jets* data and returns the *events* that fulfill the condition described above.

The following table and graphics show the times that the *cut* need, depending on the schema used, the number of *events* evaluated and query applied.

Data	results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	60	10.067	4,701.578	11.154	4.875	7.926	6.548
1000	594	160.116		93.570	60.980		42.935
5000	2907	1471.026		527.462	116.546		204.168
10000	5594	5661.451		1260.144	294.352		340.074
15000	8891	13497.309		3004.274	597.288		755.560
20000	11424	23123.496		5146.902	1,112.527		1,056.475
25000	14280	36,130.463		8,042.034	1,420.292		1,303.052

Table 8 *TopCut* execution time results table

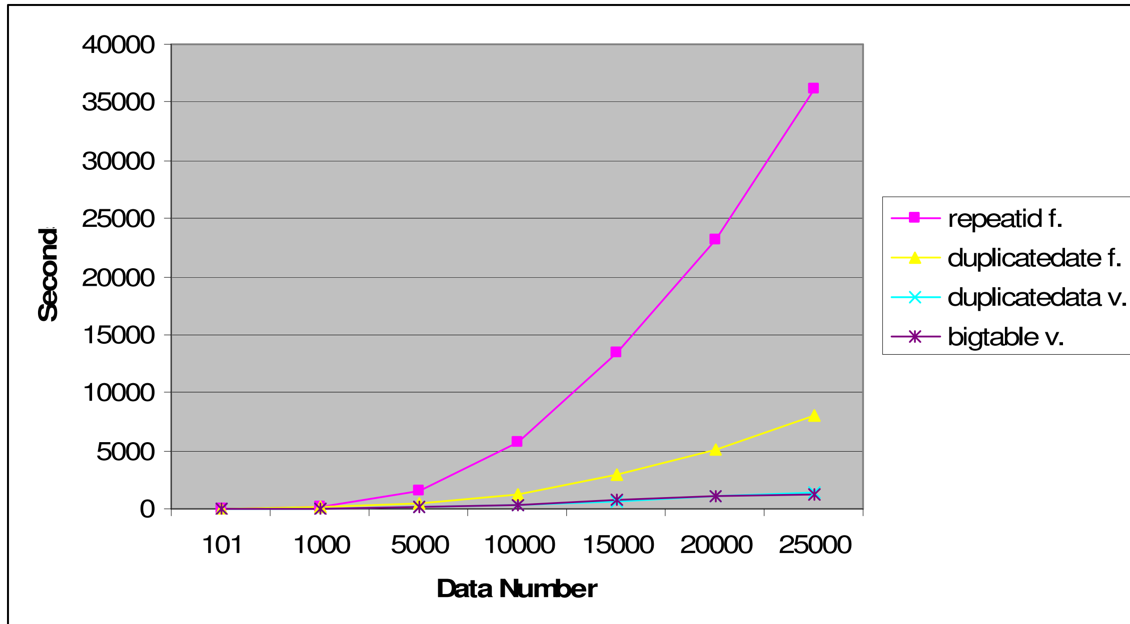

Figure 4: *TopCut* execution time results graphic

RepeatID views and *BigTable* functions were not tested due to their slow performance; *RepeatID functions* scale badly, *DuplicateData functions* times are not as inefficient as *RepeatID* functions, but *DuplicateData views* and *BigTable views* times have the best scale in comparison with the rest. Times are slower in comparison with the previews queries due the complexity of the query.

JetVetoCut:

This cut is a variation of *HardtronicTopCut*. This one takes *events* with *jets* that belong to the three *jet* system described in the *HardtronicTopCut* and those *jets* should have *pt* not bigger then maximum *pt* allowed for the rest of the *jets*. This cut search in all *jets* data and returns the *events* that fulfill the condition described above.

The following table and graphics shows the times that the *cut* need, depending on the schema used, the number of *events* evaluated and query applied.

Data	results	RepeatID f	RepeatID v	DuplicateData f	DuplicateData v	BigTable f	BigTable v
101	15	34.847	7,330.310	20.868	10.289	31.289	10.762
1000	107	207.231		161.510	128.710		134.627
5000	504	4880.578		1441.333	409.718		428.553
10000	1098	21265.376		6280.095	645.457		656.399
15000	1514	43983.305		12989.159	1,921.642		953.312
20000	2132	82582.479		24388.275	2,398.913		1,339.509
25000	2637	127,679.408		37,706.310	2,997.617		2,141.638

Table 9 *JetVetoCut* execution time results table

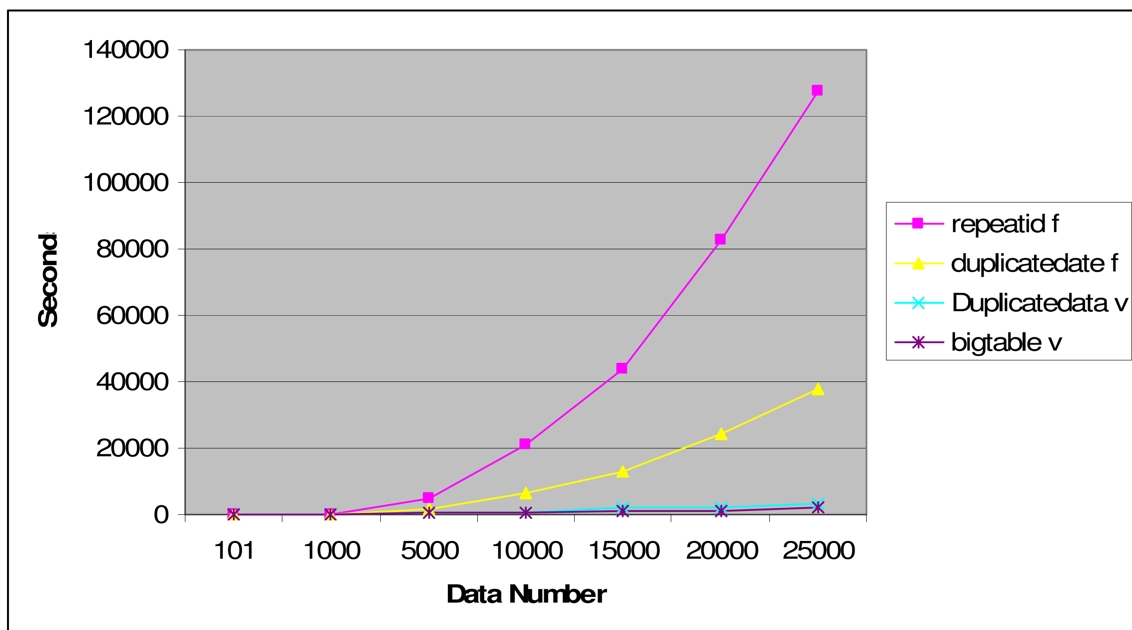

Figure 5: *JetVetoCut* execution time results graphic

Same behavior than previous *cuts* was observed, but this *cut* is even slower than the previous ones. *JetVetoCut* are the most complex queries of all six created.

leptonCut:

This cut takes *events* that have not *isolated leptons* with *pt* bigger than 150 GeV and at the same time have at least one *isolated lepton* with *pt* smaller than 40 Gev. This cut searches in all *lepton* data and returns the *events* that fulfill the condition described above.

The following table and graphics shows the times that the *cut* need, depending on the schema used, the number of *events* evaluated and query applied.

Data	results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	14	1.569	60.643	0.239	0.401	1.126	0.428
1000	95	14.595		3.432	1.443	9.959	1.244
5000	474	103.963		17.252	2.225	64.741	2.400
10000	937	411.028		28.672	3.096	255.961	3.961
15000	1559	1025.817		71.557	5.917	638.810	6.625
20000	2894	2538.989		177.110	9.856	1581.112	9.001
25000	5121	5,616.000		391.751	15.181	4,053.600	14.043

Table 10 *leptonCut* execution time results table

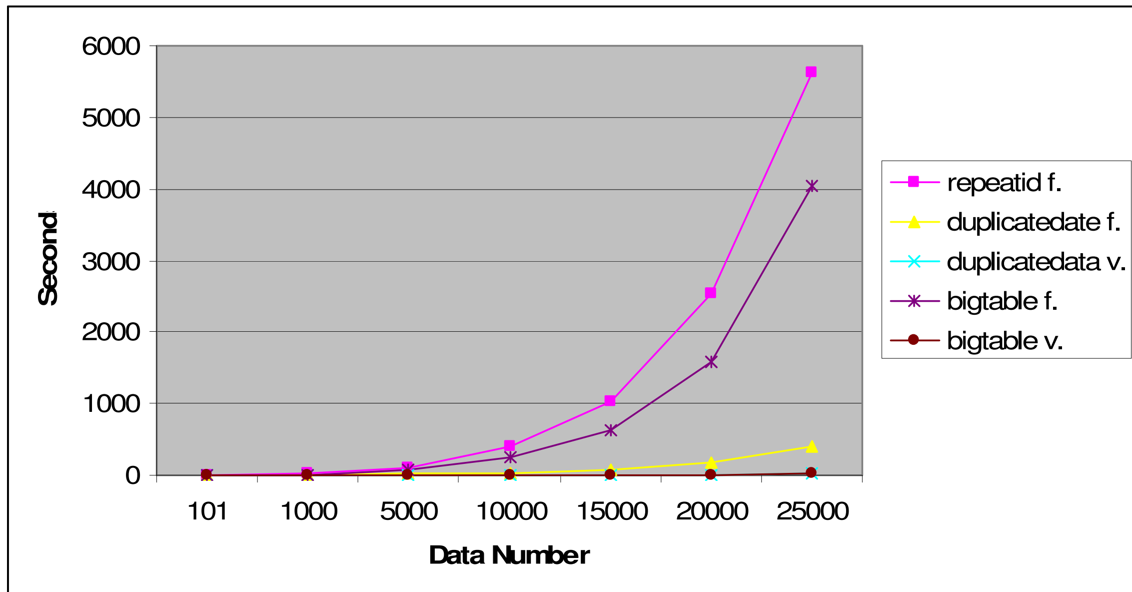

Figure 6: *leptonCut* execution time results graphic

DuplicateData views and *BigTable views* show the best scalability compared to the other scenarios.

MissEeCuts:

This *cut* is fulfilled by *events* that have *missing transverse energy* (mod(PtMiss)) not smaller than minimum *missing transverse energy allowed* (40 GeV) and its *effective mass* should be not bigger then *maximum missing transverse energy*

allowed (150 GeV). This cut search in all *lepton* data and returns the *events* that fulfill the condition described above.

The following table and graphics shows the times that the *cut* need, depending on the schema used, the number of *events* evaluated, and query applied.

Data	results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	28	0.384	54.126	0.406	0.117	0.529	0.153
1000	548	23.260		3.543	2.483	7.437	3.0961
5000	2738	581.074		13.578	5.745	185.789	5.917
10000	5877	2494.501		58.287	9.59	797.576	9.855
15000	8212	5228.393		122.168	16.809	1671.694	15.096
20000	10956	9300.581		217.320	23.392	2973.710	21.986
25000	13693	14,530.035		339.513	34.087	3497.268	30.301

Table 11 *MissEeCut* execution time results table

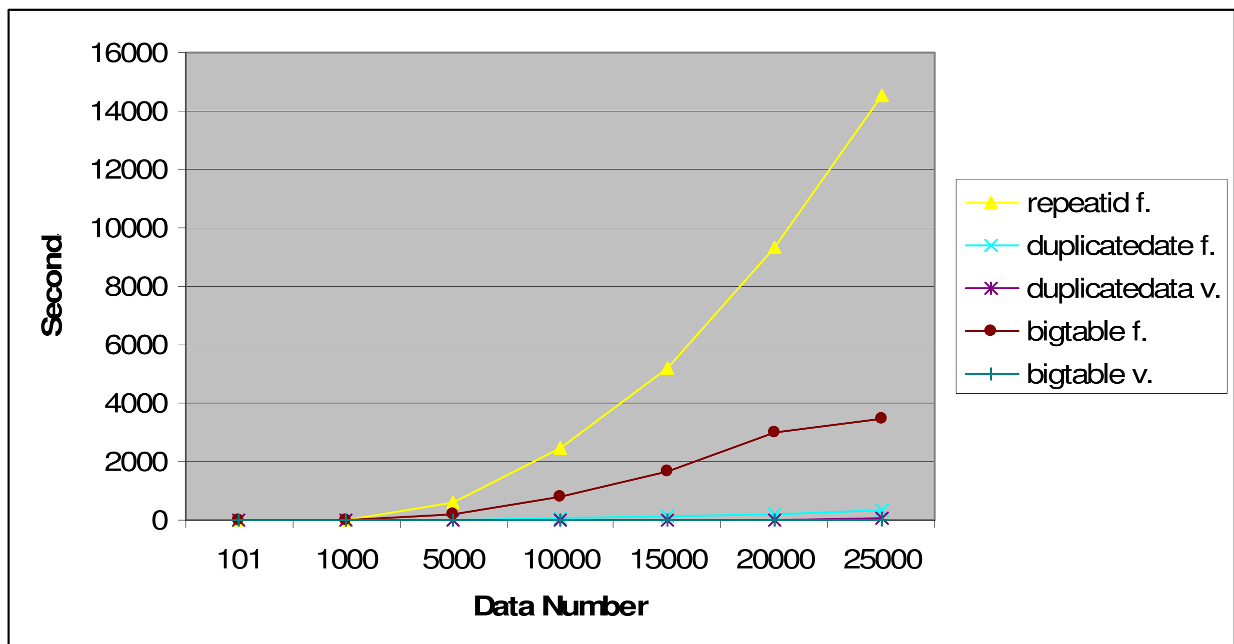

Figure 7: *MissEeCut* execution time results graphic

DuplicateData views and *BigTable* views have a good scaled showing the fastest times.

allCuts:

AllCuts looks all *events* that fulfill all six *cuts* conditions, in the following order *ThreeLeptonCut, zVetoCut, topCut, JetVetoCut, leptonCuts* and *MissEeCuts.* This query searches in all *events* data and returns the ones that complies all cuts developed.

The following table and graphics show the times that the cut needs, depending on the schema used, the number of *events* evaluated and query applied.

Data	results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	1	3.810	14,476.119	0.711	12.868	1.283	16.233
1000	1			22.129	163.036	158.684	133.648
5000	1			103.454	462.923	914.186	484.079
10000	1			211.207	804.524	1,843.749	788.846
15000	2			315.144	2,343.539	2,765.623	2,071.179
20000	2			432.185	3,516.787	3,687.498	2,422.274
25000	2	13746.48		529.971	5,031.388	4,645.743	3,262.833

Table 10 : *allCuts* execution time results table

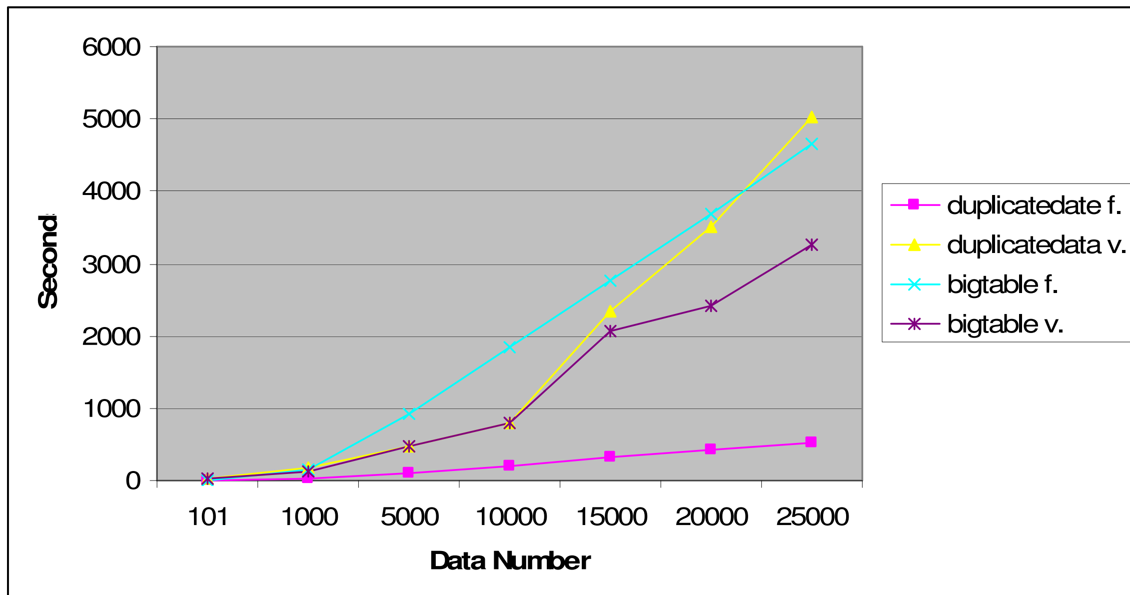

Figure 8: *allCuts* execution time results graphic

Contrary to the single *cuts* operations, *allCuts* shows faster times with *functions* with small quantities of data. *DuplicateData functions* shows the better scalability curve grown, compare with the scenarios tested.

optAllCuts:

This query also looks for *events* that fulfill all six cuts, but in a different order, that order is *threeLeptonCut, leptonCuts, missEECuts, zVetoCut, topCut, and JetVetoCut*. optAllCuts searches in all *events* data and returns the ones that complies all cuts developed.

The following table and graphics shows the times that the cuts need, depending on the schema used, the number of *events* evaluated and query applied.

Data	results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	1	3.711	11776.884	0.757	12,388	1.372	19.195
1000	1			26.339	160.77	174.243	158.051
5000	1			112.647	463.118	871.213	572.468
10000	1			220.833	794.209	1743.312	932.883
15000	2			336.031	2224.775	2613.626	2449.36
20000	2			448.041	3516.075	3646.964	2864.563
25000	2	13244.559		554.899	4942.812	4939.201	3743.025

Table 11 *optAllCuts* execution time results graphic

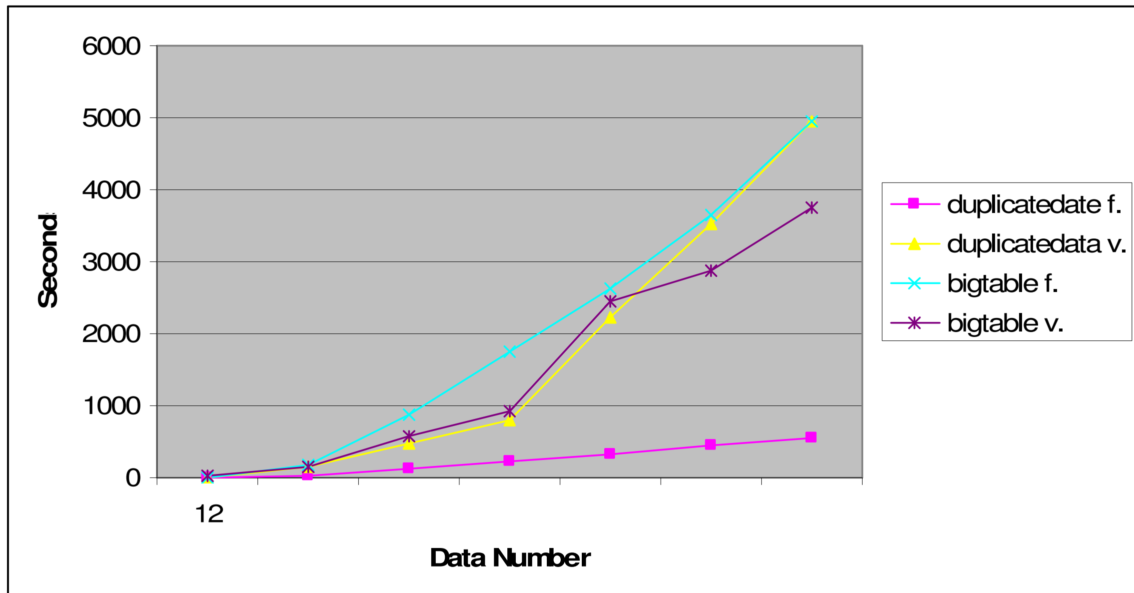

Figure 9: *optAllCuts* execution time results graphic

Here the performance is similar to *allCuts*, but times in the majority of the tests are all little bit faster. This shows that one can gain some better performance by optimizing the query formulation of this *cut*. *DuplicateData* with *functions* shows the fastest times and the best scalability.

expCuts:

This query is the last of all cuts order tested, which is *topCut, JetVetoCut, MissEeCuts, zVetoCut, threeLeptonCut, leptonCuts.*

The following table and graphics show the times that the cuts need, depending on the schema used, the number of *events* evaluated and query applied.

Data	results	RepeatID f.	RepeatID v.	DuplicateData f.	DuplicateData v.	BigTable f.	BigTable v.
101	1	38.191	13361.258	37.725	16,461	32.33	22.609
1000	1			1036.634	209.021	4321.882	190.423
5000	1			5335.985	601.453	22919.467	689.721
10000	1			10743.489	1031.441	45399.48	1123.955
15000	2			16148.99	2876.332	67605.705	2950.796
20000	2			21558.485	4566.332	89701.414	3450.767
25000	2	136303.679		26067.975	6403.329	116,355.670	4499.191

Table 12 *expCuts* execution time results table

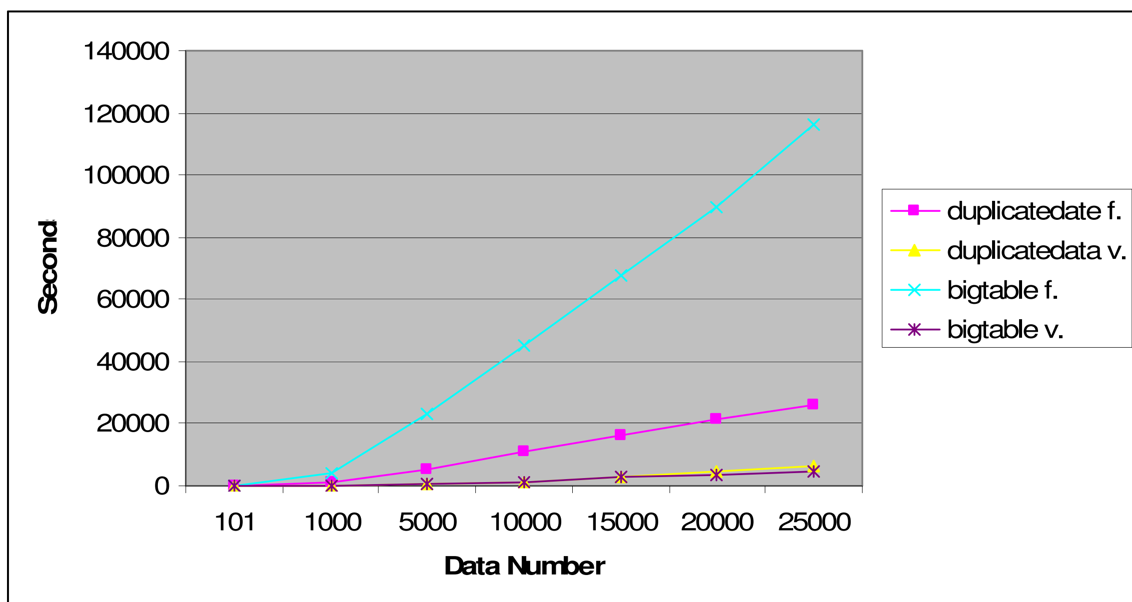

Figure 10: *expCuts* execution time results graphic

Times for *expCuts* are slower than the other two *allCuts* queries, but with this one times for *views* are considerable faster than times with *functions*, but comparing with times with the other two queries, they still seems slower.

4.4 Discussion

The curves of the single cuts with *functions* scale badly. In all single cuts, *DuplicateData views* and *BigTable views* shows the best scale curves and the fastest times. A possible explanation is that *functions* are treated as black boxes by the optimizer, while *views* are expanded with the rest of a query and query optimizer is able to do a better work. On the other hand, *Higgs Boson* queries (*allcuts, optallcuts, expcuts*) do not behave in the same way; best option with a large distance seems to be *DuplicateData* with *functions* as we can observe in the time tables and graphics in the performance evaluation. A possible explanation of why *functions* have faster times than *views* in these cases could be because *functions*, in the moment that they were implemented, were easier to parameterize and obtain the values or the specific tuples that were needed in the moment of the execution directly, making the queries simpler and efficient [9].

Times for Higgs Boson queries with views are close to the total times that every single cut takes; for example, the sum of the times of all single cuts with 25000 *events* and *BigTable* schema with views is 3514.941 seconds and the time of execution for the queries *allCuts*, *optAllCuts* and *expCuts* are 3262.833, 3743.025, and 4499.191 seconds respectively. With 25000 *events* and *DuplicateData* schema, the total times of all single cuts is 4,497.227, and the times the same multiple cuts queries are 5,031.388, 4942.812, and 6403.329 seconds respectively. In general, with times with *expcuts* the times are considerable slower.

5. Summary and Future Work

After check implementation times, it can be concluded that for single cuts with small quantities of data it is faster to work with all attributes replicated in all subclases, and implementing the queries using views; for single cuts with large quantities of data it is better to use a single big table schema and implementing views as queries.

For Higgs Boson queries, replication of data schema with functions implementación is definitely the best option. All these statements can be justified by comparing the measured times on performance evaluation. Quicker times and good scalability being the most favored condition looked for, considering that the main objective of this research is to work with large quantities of data that are generated by HEP *events*, and to reduce processing time.

Due to limitations in time allowed for performing this work, evaluation, in spite of we could find concrete conclusions, it could not be done as detailed and reliable as we wanted. It would be interesting to take some more time for this, in order to confirm the results founded or to rectify wrong conclusions.

It is still possible to improve the performance in SQL-2003 with the use of more indexes. It will be interesting to apply indexes in the schemas for faster performance, in particular for a big table with views for single cuts, and data replicate with functions for *Higgs Boson*.

References

[1] Ruslan Fomkin and Tore Risch. "Cost-based Optimization of Complex Scientific Queries". Department of Information Technology, Uppsala University. Available at http://user.it.uu.se/~ruslan/FomkinSSDBM07.pdf

[2] Fundamentals of Database Systems, 5th Edition, Elmastri and Navathe, Pearson, ISBN 0-321-41506-X, 2007

[3] Database models. http://www.unixspace.com/context/databases.html

[4] Relational Database. Wikipedia http://en.wikipedia.org/wiki/Relational_database

[5] Index (database) Wikipedia. http://en.wikipedia.org/wiki/Index_%28database%29

[6] User-defined_function. Wikipedia. http://en.wikipedia.org/wiki/ User-defined_function

[7] Higgs Boson. Wikipedia. http://en.wikipedia.org/wiki/Higgs_boson

[8] Views In SQL Server. http://www.sql-server-performance.com/articles/dev/views_in_sql_server_p1.aspx

[9]¨Choice between store procedures, store functions, views, triggers, inlineSql. http://www.paragoncorporation.com/ArticleDetail.aspx?ArticleID=28