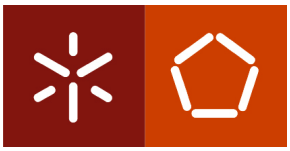

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master's Thesis
Master in Informatics Engineering

Science Data Vaults in MonetDB: A Case Study

João Nuno Araújo Sá

Supervisors:
Prof. Dr. José Orlando Pereira
Departamento de Informatica, Universidade do Minho
Prof. Dr. Martin Kersten
Centrum Wiskunde & Informatica, Amsterdam

July 2011

Declaration

Name: João Nuno Araújo Sá

Email: joao.nuno.a.sa@gmail.com

Telephone: +351964508853

ID Card: 13171868

Thesis Title: Science Data Vaults in MonetDB: A case study

Supervisors:
Prof. Dr. José Orlando Pereira
Prof. Dr. Martin Kersten
Dra. Milena Ivanova

Year of Completion: 2011

Designation of Master: Master in Informatics Engineering

University of Minho, 13th July 2011

João Nuno Araújo Sá

Experience is what you get when you didn't get what you wanted.

Randy Pausch (The Last Lecture)

Acknowledgments

To Dr. José Orlando Pereira for accepting being my supervisor and for giving me this possibility to do my master thesis in Amsterdam. Apart from the distance, all the emails and recommendations were very helpful and I am thankful.

To Dr. Martin Kersten for receiving me in such a recognizable place as CWI and providing me the opportunity to be responsible for this project.

To Bart Scheers for all the patience talking about astronomical concepts and suggesting ideas to build a robust and solid use case.

A special thanks to Milena Ivanova for all the support, advice, inspiration and friendship. During all the meetings her help was essential, and to all the intensive corrections on the text I am extremely grateful.

To all the people of CWI, in particular the INS-1 group, for making me feel one of them through their sympathy and professionalism.

Ao meu pai, José Manuel Araújo Fernandes Sá e à minha mãe, Fernanda da Conceição Pereira de Araújo Sá, por todo o apoio, compreensão e saudades durante este ano que estive fora.

A toda a minha família, aos meus velhos amigos de Viana do Castelo e aos amigos que fiz em Braga.

To all my Amsterdam friends for making this year memorable. I will never forget the moments we had together.

Resumo

Hoje em dia, a quantidade de dados gerada por instrumentos científicos (dados capturados) e por simulações de computador (dados gerados) é muito grande. A quantidade de dados está a tornar-se cada vez maior, quer por melhorias na precisão dos novos intrumentos, quer pelo aumento do número de estações que recolhem os dados. Isto requere novos métodos científicos que permitam analisar e organizar os dados.

No entanto, não é fácil lidar com estes dados, e com todos os passos pelos quais necessitam de passar (capturar, organizar, analisar, visualizar e publicar). Muitos são coleccionados (captura), mas não são selecionados (organização, análise) ou publicados.

Nesta tese focamo-nos nos dados astronómicos, que são geralmente armazenados em ficheiros FITS (Flexible Image Transport System). Vamos investigar o acesso a esses dados, e pesquisar informação neles contida, utilizando para isso uma tecnologia de base de dados. A base de dados alvo é o MonetDB, uma base de dados de armazenamento por colunas, de código livre, que já demonstrou ter sucesso em aplicações que analisam a carga de trabalho e aplicações científicas (SkyServer).

Perante os resultados obtidos durante as experiências, a perceptível superioridade apresentada pelo MonetDB em relação à ferramenta STILTS quando mais computação é exigida, e por último, pelo sucesso na execução do conjunto de testes apresentado pelo astronómo que trabalha no CWI, podemos afirmar que o MonetDB é uma alternativa forte e robusta para manipular e aceder informação contida em ficheiros FITS.

Abstract

Nowadays, the amount of data generated by scientific instruments (data captured) and computer simulations (data generated) is very large. The data volumes are getting bigger, due to the improved precision of the new instruments, or due to the increasing number of collecting stations. This requires new scientific methods to analyse and organize the data.

However, it is not so easy to deal with this data, and with all the steps that the data have to get through (capture, organize, analyze, visualize, and publish). A lot of data is collected (captured), but not curated (organized, analyzed) or published.

In this thesis we focus on the astronomical data, typically they are stored in FITS files (Flexible Image Transport System). We will investigate the access and querying of this data by means of database technology. The target database system is MonetDB, an open-source column-store database with record of successful application to analytical workloads and scientific applications (SkyServer).

Given the results of the experiments, the perceptible superiority presented by MonetDB over STILTS when more computation is required, and the success obtained during the execution of the use case proposed by an astronomer working at the CWI, we can declare that MonetDB is a powerfull and robust alternative to manipulate and access information contained in FITS files.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past, scientific data was collected and stored predominantly in files. Using the file system is easy and practical but it has a number of disadvantages. First, files have no metadata, they do not benefit the evolution of data analysis tools, they do not have a high-level query language and the query methods will not do parallel, associative, temporal, or spatial search.

One strategy used by scientists to overcome the lack of metadata in files, is to provide extra information in the file name, allowing for the data of interest to be filtered efficiently. For example, the file name "January2010Hubble" describes that data was captured in January, 2010, by the space telescope named Hubble. The disadvantages of this approach are: limited number of parameters can be encoded in the file name, specific software needs to be written to understand the names, and we can have very long file names.

However, scientists prefer to use files instead of databases [17]. And when they are confronted about the reason why do not use databases, there is a huge range of answers they give:

- They do not benefit with the utilization of them
- The cost of learning the tools is not worth it
- They do not have a good visualization/plotting of the results or because they use their own programming language
- Because there are incompatible scientific data types such as N-dimensional arrays and spatial text which are very difficult to support
- Because is to slow (loading takes too long, and sometimes it is not the data they need)

- Because once the scientific data is loaded, it cannot be manipulated anymore using standard applications.

In other words, database systems were not initially built to support science's core data types. Therefore, a big evolution is needed before a second look by the scientists. However, things are different now. The datasets are becoming bigger and bigger (peta-scale), file-ftp will not work with such a huge amount of data, and scientists need databases for their data analysis, for non-procedural query analysis, automatic parallelism, and search tools. Another way to access the data is needed. In the book "The 4th Paradigm" [17], Jim Gray describes those problems, emphasizing that better tools that support the whole research cycle need to be produced, from data capturing and data curation to data analysis and visualization. He also affirmed that the science evolution developed in the following four paradigms:

- It belongs to thousands of years ago, when science was only experimental and empirical.
- When it turned into theoretical science (some hundreds of years ago) with its equations, laws and models.
- In the last few decades, the theoretical paradigms got so complex and complicated to solve analytically that some simulation was needed. With this, computational science was born, resulting in a huge amount of data generated by the simulations.
- Data exploration, where the data is either captured by instruments or generated by simulations before being processed by software and finally stored in computers. The scientists only look at the data at the end of this whole process.

1.1 The need to integrate with repositories

It is essential to have good metadata, describing data in standard terms, so people and programs can understand it. In the scientific community, data must be correctly documented and must be published in a way that allows easy access and automated manipulation.

Predominately the reasons why we need to integrate with repositories are as follows. Firstly, they already have their own metadata. Secondly, they have their own way to structure the data. Thirdly, they were built with the purpose to supply answers to a specific community of scientists that already have their own unique demands. Database integration of the repository will extend functionality with the minimum investment.

1.2 Assumptions

The first assumption of this work is that it is limited to a specific scientific community: astronomy, and to a particular format used in astronomy: FITS.

FITS is the standard astronomical data format endorsed by both NASA (National Aeronautics and Space Administration) and the IAU (International Astronomical Union). It also has the data structured in a standard scheme and it is extremely easy to get access to FITS files. They are available in thousands of websites: [6] and [5] are some examples of them. There are also some well known surveys that produce data using FITS files as output: FIRST [26], SDSS [4], and UKIDSS [7] for example.

The last assumption is that we limit ourselves to MonetDB, a powerful column-store database that is being developed in CWI, Amsterdam. It has the advantage of being open-source and built for analytical applications. The fact of being open source brings advantages. The availability of the source code and the right to modify it, enabling the unlimited tuning and improvement of a software product, as it is referenced in [15], is one of them. Applied to our case, allowed the creation of a new module and the development of new functionalities, inside MonetDB code.

1.3 Contributions

Our contributions to this project are a seamless integration of FITS vaults with MonetDB, through the development of a module inside the MonetDB code, that will provide a set of functionalities and it will provide a powerfull alternative to access and manipulate FITS files. We support only FITS Tables.

Once the FITS vault is understood, through the analysis of the metadata and the data model, it can be expressed in a relational database system.

The described alternative is based on a set of experimentations which are handled using MonetDB. It uses a fully new integration of FITS files with the database world that is ready to support an intricate set of astronomical questions that only databases can answer. The databases are enabled to process this data because they are developed and maintined for this purpose.

The experiments will be undertaken by a set of stress tests using MonetDB.

1.4 Approach

We do a bottom-up evaluation of the primitive needs that access and manipulate FITS files. The context is limited to FITS files, but we must keep others in mind. How could we work with other formats that are able to store astronomical data, how can we access their metadata and which properties do we have to understand in order to draw the data model in the relational database system. Another important factor that must be considered is what are the differences between them and the FITS format. All this questions will be answered in the **Related Work** section, where we compare FITS files with others astronomical file formats.

1.5 Project Objectives

The main objective of this thesis is to study what extensions of a modern database architecture are needed, to provide the system with the ability to understand scientific data in external formats. Such ability will provide the users-scientists with:

- View over repository of data files in FITS standard format. Such a view presents metadata of the files and allows users to locate data of interest by posting queries against the metadata;
- Automatic attachment of files and data of interest. Using this feature the user can avoid manual loading of high volumes of data, which is time- and labour-consuming. The system automatically ships the data to the database. Furthermore, only pre-selected data of interest will be touched;
- Declarative SQL queries for analysis;
- Data Integration. The system allows for complex analysis that includes combining, comparing, correlating data from different FITS files and repositories (through SQL join queries, missing in FITS tools).

1.6 Outline of report

In Chapter 2 we will provide some knowledge on the background required to understand the thesis work. We will describe some important features of MonetDB and FITS files. In Chapter 3 we will present our contribution to MonetDB. More precisely, we describe the FITS vault module that was developed and some important decisions made during the design & development process. In Chapter 4 we illustrate the functionality

of the module by conducting an astronomy use case. In Chapter 5 we will conduct experiments, comparing the performance of MonetDB with the existing tools that operate with FITS files. In Chapter 6 we will enumerate some libraries that provide a set of functionalities to manipulate and access FITS files. We will list some programs that use those libraries and investigate what they can and cannot do with the respect to our use case. Finally, in Chapter 7, we will talk about the results and a overview about the entire project. We will end with some suggestions for future work.

Chapter 2

Background

In this chapter we introduce the main concepts that underly our project. Here we will introduce the MonetDB system, focusing on the enhancements that it brings to the database world and the benefit for the astronomy community, as a database engine designed and prepared to answer routine astronomical queries. Thereafter, a presentation about the FITS files will be undertaken, exploring their history, their characteristics and how they are structured.

2.1 Introduction to MonetDB

MonetDB [2] is an open-source column-oriented database management system developed, since 1993 at CWI, Amsterdam. The column-store idea was born as a dependable solution to solve the main bottleneck faced by the majority of database systems: the main-memory access. Vertical fragmentation is identified as the solution for database data structures, which leads to optimal memory cache usage [11].

It is implemented by storing each column of a relational table in a separate binary table, called a Binary Association Table (BAT). A BAT is represented in memory as an array of fixed-size two-field records [OID,value], or Binary UNits (BUN). Their width is typically 8 bytes. MonetDB executes a relational algebra called the BAT Algebra that is programmed with the MonetDB Assembler Language (MAL). With a binary table for each column of the relational table, the query execution model is also different from main stream systems, having one operator at a time over entire columns. All these changes in the database architecture led to a creation of a brand new software stack, innovating all layers of the Database Management System.

The three layers that compose MonetDB software stack are: the query language parser, the set of optimizer modules and the back-end (MAL interpreter). The query language

parser it has an optimizer that reduces the amount of data produced by intermediates and exploits the catalogue on join-indices. The output is a logical plan expressed in MAL. The optimizer module takes decisions based on cost-based optimizers and runs algorithms. This module is crucial for the efficiency of the database. The MAL interpreter maintains properties over the objects accessed to gear the selection of subsequent algorithms. For example, a Select operator can benefit from sorted-ness.

This provides a simplification of the database kernel, an ability to fully materialize intermediate results, efficiency in the query processing speed and high performances when dealing with complex queries on sizable amounts of data.

This database system has already proved to be an asset for real-life astronomy applications (SkyServer project) [19], with the goal to provide public access to the Sloan Digital Sky Survey warehouse for astronomers and the wider public. MonetDB, and the column store approach, demonstrate that they are promising for the scientific domain. We will use MonetDB to store the data imported from external file formats, and provide the data when requests are made.

The MonetDB code can be extended with new functionality. Functions need to be compiled and brought into the MAL level, so they can be used in the SQL level. This extensibility is done through a `create procedure` [21] in the SQL level. This will enable the user to call the functions and have access to the functionalities provided by the module.

2.2 Introduction to FITS

It was in Holland and in the United States of America that the first high quality images of the radio sky were produced. The decade was 1970 and the pioneers were the Westerbork Synthesis Radio Telescope (WSRT) in Westerbork, Holland, and the Very Large Array (VLA) in New Mexico, United States of America.

Their wish was to combine the data derived from both instruments. The main problem was that the two groups were observing at different frequencies. As a consequence, it was complex to exchange information, either because the institutions had their own way to organize the data (internal storage format), or because there were considerable differences in the architecture of their machines, using a distinct internal representation for the same number, for instance.

Lacking a standard format for the transport of images, everytime when an astronomer needed to take data from an observatory to a home institution, some special software had to be created, in order to convert (restructure and perform bit manipulations) the data from the original machine, to the format that was being used by the home institution.

With all this setbacks, creating a single interchange format for transporting digital images between institutions, in order to avoid all this heavy process was needed. The idea was that each institution would need only two software packages: one that would translate the transfer format into the internal format, used by the institution and one that would transform the internal format into the transfer format. The Flexible Image Transport System (FITS) [13], was created with the intention to provide such a transfer format.

2.2.1 Applications of the FITS

The inaugural applications of FITS were the Exchange of radio astronomy images between Westerbork and the VLA and the Exchange of optical image data among Kitt Peak, VLA and Westerbork. Afterwards, the use of FITS has expanded and it is now being explored as a data structure in a diversity of NASA-supported projects:

- X-ray data from the Einstein High Energy Astrophysics Observatory (**HEAO-2**)
- Compton Gamma Ray Observatory
- Roentgen Satellite (**ROSAT**)
- Ultraviolet and visible from the International Ultraviolet Explored (**IUE**) and the **Hubble Space Telescope**
- Infrared data from the Infrared Astronomical Satellite (**IRAS**) and the Cosmic Background Explorer (**COBE**)

It is being used as a standard for gound-based radio and optical observations, for organizations such as National Radio Astronomy Observatory (**NRAO**), National Optical Astronomy Observatories (**NOAO**) and European Southern Observatory (**ESO**)

2.2.2 The structure of a FITS file

A FITS file is composed by a sequence of Header Data Units (HDUs), that can be followed by a set of special records. Each HDU is composed by one header and the data that follows. The header is a sequence of 36 80- byte ASCII card images containing *keyword* = *value* statements. There are three special classes of keywords: required keywords, reserved keywords and the ones that are defined by the user. The data that follows (also called data records) is structured as the header specifies and it is *binary data*. The size of each logical record is 23040 bits, equivalent to 2880 bytes. Each HDU consists of one or more logical records. The last record of the header is filled with ASCII blanks so it can fill

the 23040-bit lenght. The first HDU of a FITS file is called *Primary HDU*. The HDUs that follow the Primary HDU are called *extensions*. When the FITS files contains one or more extensions, it is most likely that the Primary HDU does not contain any data. When the FITS file does not contain extensions it is called a Basic FITS, that is a file containing only the primary header followed by a single primary data array.

The Primary HDU is the first HDU of a FITS file. It is composed of one header (Primary Header) and the data that follows. If the Primary HDU is alone in the FITS file (there are no extensions), so it will be called *Basic FITS*. It is not normal (except for FITS images) that a Primary HDU contains any data, but if it does, it has to be a matrix of data values, in binary format that it is called Primary Array.

The *Extensions* have the same overall organization of all the HDUs (one header and the data that follows) and they come after the Primary HDU, respecting the structure of the FITS file. The extensions brought some new functionalities to the FITS files:

- Transfer new types of data structures: Images, ASCII Tables and Binary Tables
- Transfer collections of related data structures
- The data to be transported do not always fit conveniently into an array format
- Transport of auxiliary Information

The *Tables* are used to store astronomical data that is collected and they contain rows and columns of data. In the FITS files there are two types of tables: the ASCII Tables and the Binary Tables. As the name says, the ASCII tables store the data values in an ASCII representation. The data appear as a character array, in which the rows represent the lines of a table and the columns represent the characters that make up the tabulated items. Each member of the array is one character or digit. Each character string or ASCII representation of a number are in the FORTRAN-77 format. As for the binary tables, they store the data in a binary representation. The binary tables are more efficient, compact (about half of the size for the same information content), support more features and the time spent converting to ASCII tables is eliminated. The display is not as direct as for ASCII tables. The data types that can be stored in the FITS tables are:

- L: Logical value: 1 byte
- B: Unsigned byte: 1 byte
- I: 16-bit integer: 2 bytes
- J: 32-bit integer: 4 bytes

- K: 64-bit integer: 8 bytes
- A: Character: 1 byte
- E: Single precision floating point: 4 bytes
- D: Double precision floating point: 8 bytes
- C: Single precision complex: 8 bytes
- M: Double precision complex: 16 bytes
- P: Array Descriptor (32-bit): 8 bytes
- Q: Array Descriptor (64-bit): 16 bytes

Chapter 3

Contribution to MonetDB

In this chaper we present our contribution to MonetDB, through the development of a vault module that provides a set of functionalities concerning to FITS files. We will do a short overview of the vault concept. Further, we will describe the architecture of a vault. Finally, we will list a set of procedures and functions that were developed to make the integration between FITS files and MonetDB possible.

3.1 Overview of the vaults

A vault can be defined as a safety deposit box or as a repository for valuable information. By conducting an analogy to computer science terms, a data vault can be seen as a folder that contain only images. Inside the same data vault, the objects have one important factor in common: the metadata. It is the metadata that they have in common that allows a possible distinction between different kinds of vaults, and even the ability to create a completely new data vault based on similar parameters of the objects.

What distinguishes the objects in the same vault is the data that they carry. For example, if the object is an image, we know that it will have pixels, height and width. However, the values that are assigned to each one of the attributes differ for each image.

Knowing that, we can create a vault based on a directory of files. We just need to understand their metadata (using appropriate tools that allow us to access it), what metadata they have in common and what is their data model. Comprehending the data model, we can decide how it will be represented in the relational database system.

We will apply the term vault to our case study. Creating a vault directory of FITS files, that share the same metadata but for which each one contains its own information.

The system needs to understand the external formats that contain the scientific data

(FITS). Once this is understood, there will be a distinction between loading data and attaching data. The idea of loading high volumes of data will be abandoned as it is time consuming, and for the most part, it is not what the scientist requires. This concept allows for the attachment of data (automatic attachment of files to the database), providing the metadata to the scientists, giving them the opportunity to decide what is relevant. It will be a selective load, and it will take less time.

3.2 Architecture of the vault

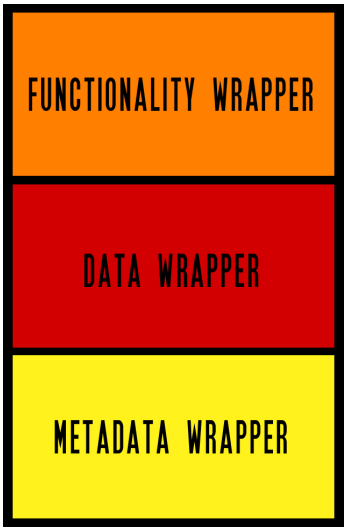

Figure 3.1: Three layers of a vault

In **Figure 3.1** we can identify three distinct layers: the metadata wrapper, the data wrapper and the functionality wrapper. The metadata wrapper reads and understands the metadata of the vault and decides how the data model can be efficiently represented in a relational database system. Looking to our FITS vault example, there are three functionalities that support this demand: **Attach a file**, **Attach all the files in the directory** and **Attach all the files in the directory, giving a pattern**.

The data wrapper loads a fragment of data of interest that was explicitly required into the database. Once again, analyzing our FITS vault case, there are two procedures that support this requirement: **Load a table** and **Export a table**. This will avoid loading huge amounts of data at once, restricting the load to small portions that might be of interest to the user. Once inside the database, the data can be queried, manipulated or even exported as a completely new FITS file.

The functionality wapper is still future work and includes load on demand and syn-

chronization when the repository is updated. For the load on demand procedure, the idea is to enable the user with the ability of typing a query to the system and the functionality wapper will decide which FITS files will be attached and which tables will be loaded, in order to give the right answer to the user.

In the following sections we will describe each one of the functionalities listed before. All of them were implemented by building a module inside MonetDB code, that accesses FITS files and the FITS catalogue, through a library called CFITSIO [16]. The CFITSIO library is written in C and provides a poweful interface for accessing, reading and writting FITS files. However, before using this library the user must have a general knowledge about the structured of a FITS file.

All the requests to the CFITSIO library are made using the C language. If the request demands some data filtering on the tables, a SQL query needs to be translated into MAL statements within MonetDB for execution.

3.3 Attach a file

We start with the attachment procedure, that opens a FITS file given its absolute name. After checking if the file has the FITS format, it scans the metadata that is present in its HDU and inserts descriptions of the table extensions to an internal FITS catalog. The catalog is composed out of the following tables:

fits_files

id: Primary key. Unique number that identifies the file

name: absolute path to the attached file

fits_tables

id: Primary key. Unique number that identifies the table

name: name of the table that coincides with the name of the HDU. If the name is already taken by another table, it creates a new name, concatenating the name of the file, underscore and the number of the HDU which corresponds to the respective table

columns: number of columns present in the table

file: id of the file, that can be identified in the table fits_files

hdu: number of the HDU that the table represents in the FITS file (the number 1 is always reserved for the primary HDU)

date: this information is not always present in the FITS file. However, it stores the date when the FITS file was created

origin: similar to the date column, is data that is not always present. It stores the information about the station responsible for the creation of the FITS file

comment: sometimes FITS files have a field reserved for some additional information or comments that might be appropriate to supply.

fits_columns

id: Primary key. Unique number that identifies the column

name: name that was given to the column within the table

type: the type of column represented in a FITS format. For instance: 8A represents a string with 8 characters.

units: extra information about the units of the stored data: meters, kilometers, etc.

number: number of the column within the table

table_id: id of the table which the column is present. It can be identified in the table fits_tables

fits_table_properties

gives extra information about a table, such as extension, bitpix, stilvers (version of the product generating the file) and stilclas

3.4 Attach all FITS files in the directory

This procedure enables the attachment of all FITS files that are present in a specific directory, that is explicitly given as a parameter. If the directory can not be opened, a proper error message will be transmitted to the user. This pattern is useful because it avoids the attachment of the FITS files whithin a directory one by one, that is time consuming and exhaustive if the directory has thousands of FITS files.

3.5 Attach all FITS files in the directory, using a pattern

This procedure is an extension of the previous one. It adds the possibility of giving a pattern, in conjunction with the name of the directory, in order to diminish and limit the number of FITS files that are attached to the database. It works similar by the **ls** program used by UNIX. The advantage of this pattern follows the same idea as the previous one.

However, it adds the option to give a pattern that will filter the FITS files attached. For example, suppose that there is a directory that has two thousand files, from which we are only interested in one thousand of them. If it is possible to build a pattern that will attach only those files of interest, we should call this procedure.

3.6 Table loading

Another feature designed was the load procedure, that loads the table with a given name. It searches the name in the FITS catalog, opens the corresponding file, moves to the matching HDU, creates an SQL table and loads the data into it. If the table is not described in the catalog, or it is already loaded, an appropriate error message will be returned. It calls a function from the CFITSIO library called *fits_read_col*. This function reads a column of the table and the data read are moved into an internal binary structure of MonetDB. This function will be called the same number of times as the number of columns present in the table.

This is an important and mandatory functionality, because it brings the data into the database system so it can be queried, processed and manipulated. The data type that takes more time to be loaded into MonetDB is the string type, due its unique structure in the database architecture. But more details on that will be studied in the upcoming chapters. This feature is in fact a bottleneck, when compared to existing tools that already query, process and manipulate data present in files. To make it work in the database world, the data needs to be first loaded from the files, so later it can give some answers to the users.

A fast mechanism to load the data into the database is essential. With this, it would be possible for a transition from the old and slow software tools world, to the more powerful and faster database system world.

In the further sections we will analyze the loading of different types of data into MonetDB. We will focus on one type at a time, building a set of FITS files, with only one column, that will contain the target type.

Each FITS file has only two HDUs: the Primary HDU, that is mandatory in each FITS file; and one extension, that contains a binary table with the type that we want to study.

With this group of tests, we can have a clear and succinct idea about how each one of the different types behaves. We will start with the column types that can be used to represent numbers.

- **Short**: any number between 0 and 32767 that occupies 2 bytes in MonetDB representation

- **Integer**: any number between 0 and 15000000 that occupies 4 bytes in MonetDB representation
- **Long**: any number between 0 and 2147483647 that occupies 8 bytes in MonetDB representation
- **Float**: any floating point number between 0 and 5 that occupies 4 bytes in MonetDB representation
- **Double**: any floating point number between 0 and 5 that occupies 8 bytes in MonetDB representation

Finally the column type **String**. Three different sets of FITS files will be created for this type: files that contain strings with the size of **4 bytes**, files that contain strings with the size of **8 bytes** and files that contain strings with the size of **20 bytes**. The idea is to apply distinct benchmarks, with the same implementation, to the strings with different sizes.

Knowing the different types that will be studied, it is now time to enumerate the number of rows that each set of FITS files will contain:

FITS File	Number of Rows
1	30000
2	2942000
3	29420000
4	58840000
5	117680000
6	205940000
7	235360000
8	470720000

Table 3.1: First group of FITS files

During the experiments, it was verified that the strings were the ones that took more time to be loaded into **MonetDB** and the type **Short** was the fastest, as we will see in the further tests. The measurements were done using **GDKms()** calls, in the FITS load routine, to measure the times it takes per column.

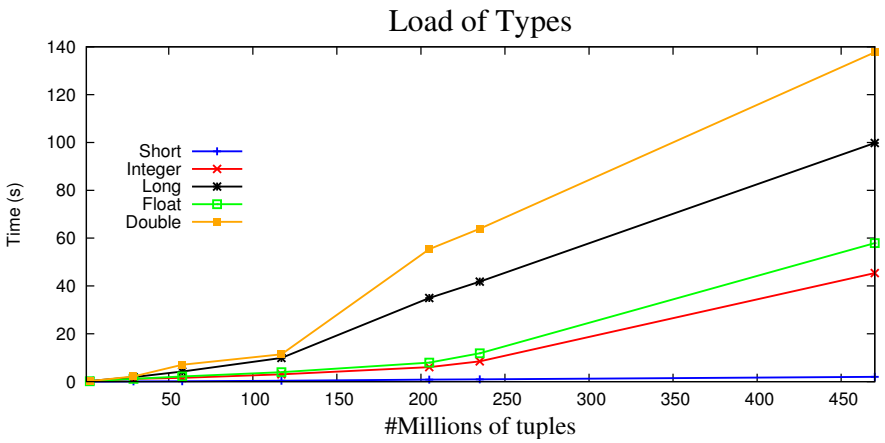

Figure 3.2: Load of all the numerical types

Figure 3.2 shows the time that took to load each one of the types in each one of the files. It can be noticed that the short type is the fastest and the double is the slowest. The type short is the fastest because it has the size of 2 bytes in MonetDB representation. There is a similarity between the loading time behavior of the integers and the loading time of the floats. This happens because both use 4 bytes in their MonetDB internal representation. Nevertheless, floating points are more complex to store, leading to a worse performance in the loading of the tables, when compared to integers. The same scenario takes place for the doubles and the longs. Both have 8 bytes in their MonetDB internal representation and both have a similar behavior in their loading time process, however, the double type is used to store floating point values. Consequently, there is a little additional time due to their complexity.

In order to understand some future behaviors, and taking advantage of the fact that the numerical types are already loaded into MonetDB, we can consult their BAT sizes in MonetDB internal representation and also the sizes of the original FITS files, where the data were in first place.

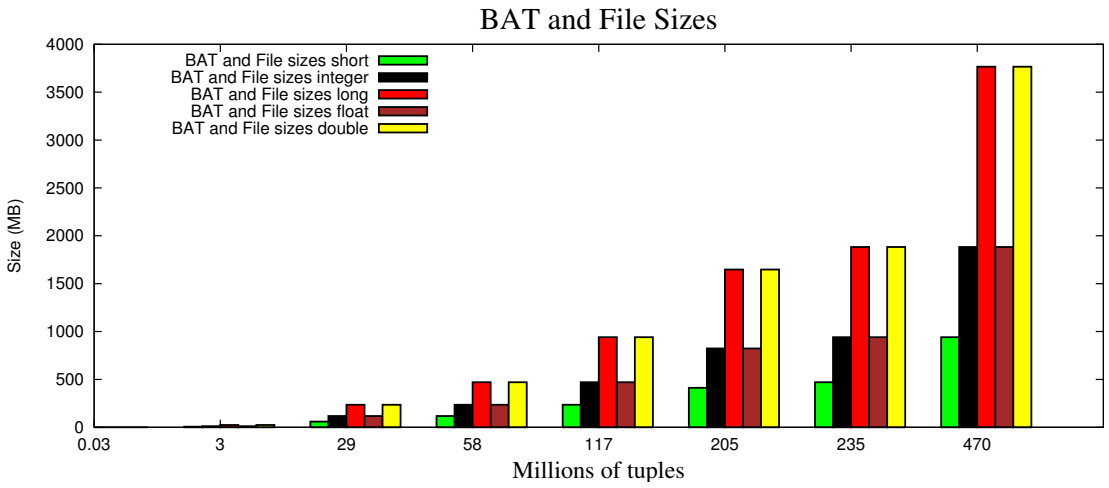

Figure 3.3: BAT and File sizes for each one of the numerical types

Figure 3.3 represents the BAT and file sizes of each one of the numerical types. We realized that the BAT sizes in the internal representation of MonetDB are the same as the sizes of the FITS files that contain the original data, for all the types tested. So, instead of having two bars, representing the BAT size and the file size, for each one of the types, we draw only one bar, that represents both BAT and file size.

These results can be explained as follows:

- the short type takes 2 bytes in its representation. Multiplying 2 bytes for 470 millions of tuples (the last file of the test) we get 940 MB, that is precisely the size of the last file that belongs to the set of files tested to the short type.
- the integer and float types use 4 bytes in their representation. As a consequence, the sizes of the BATs and the sizes of the files increase to the double (1.8 GB in the last file), comparing to the type short type.
- the types long and double occupy 8 bytes, that is twice the size of integers and floats. Fact that can be easily realized if we check the last BAT and file size of the respective types: (3.7 GB).

The next tests evaluate the loading of strings. We separate strings from numerical types due to the fact that strings are stored in a different way in MonetDB. In their internal representation, strings are composed by two different arrays:

- Tail: that contains pointers to the strings (it starts with a 4-byte size pointer)
- Heap: that contains the actual strings

During the experiments we found out that the string type needs longest time to be loaded. This statement can be proven by checking the loading times of the strings in the following set of graphs, comparing to the results of any other numerical type represented in **Figure 3.2**. Even for the string with 4 bytes, the loading time is worse than the loading time of the doubles, that are represented with 8 bytes.

To load the numerical and string types from the FITS files into MonetDB internal structure, two distinct processes need to be done:

- the call of the FITS library;
- the creation of MonetDB temporary BAT (appending time).

In order to improve the time to load the strings, several alternatives were implemented. The first alternative creates a big array with all the strings, the second performs a single call per string and the third is a middle term between the first and second approach, reading a vector of strings at a time with a given size, and then load the data. The best alternative is the third, because it avoids the first case, where a lot of memory is being used and thus swapping to disk and it also prevents so many calls of the fits load function, that occurs in the second approach, which leads to a big overhead. For example, for a vector with the size 20, 19 fits load calls are avoided comparing to the second approach.

The following tests aim to find the optimal size of the batch, measuring the time that it takes to load the strings. Those measurements involve three different components:

- FITS library function calls;
- Append to MonetDB BAT structure;
- Total time to load the strings.

We did not do this tests for the numerical types because the idea here is to optimize the time needed to load the slowest type, the strings.

3.6.1 Search for the ideal batch size

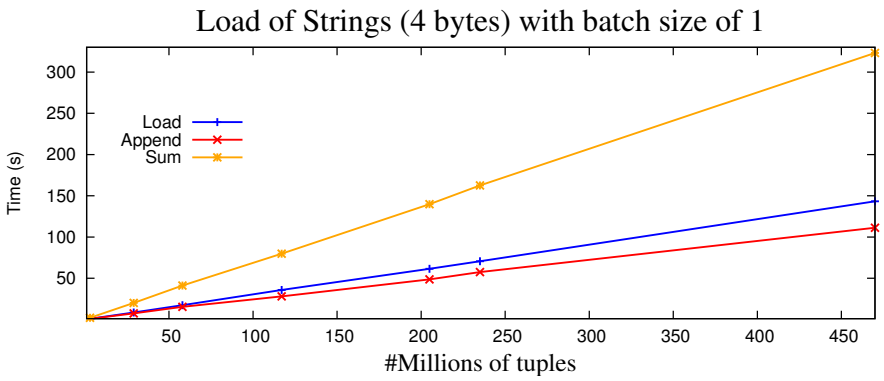

Figure 3.4: Batch 1 for the strings with 4 bytes

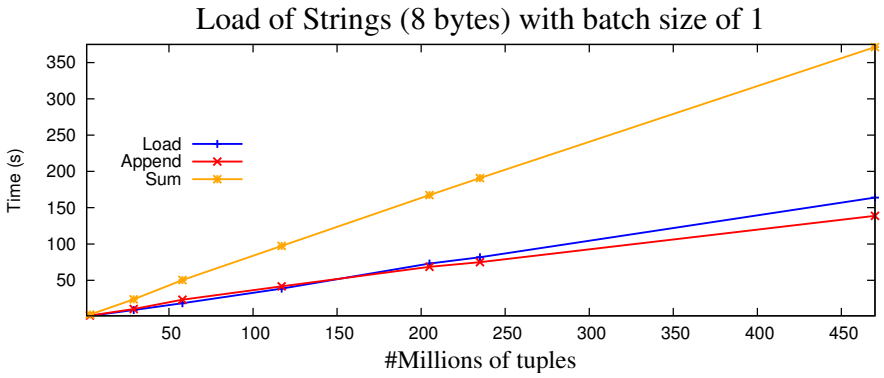

Figure 3.5: Batch 1 for the strings with 8 bytes

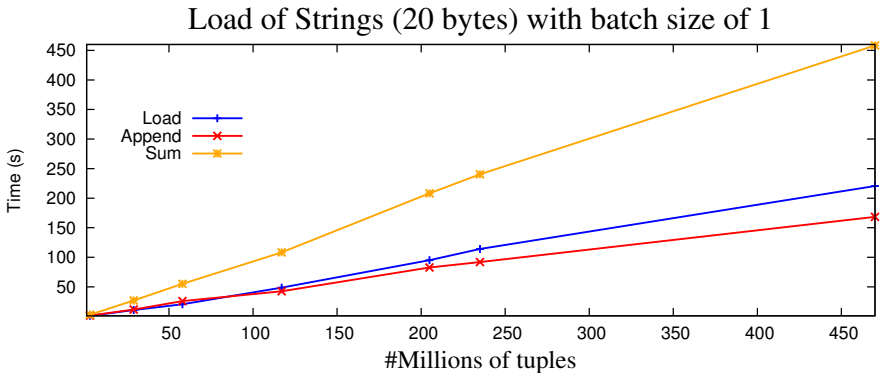

Figure 3.6: Batch 1 for the strings with 20 bytes

We start with the batch size of 1, represented in **Figure 3.4**, **Figure 3.5** and **Figure 3.6**. They are in fact the second approach that was mentioned before, performing a single call, to the fits library, per string. It can be noticed that loading the data, due to the fits library calls, takes always more time than the actual loading into the MonetDB data structure. For example, in **Figure 3.4**, for the last file, with 470 millions of tuples, the loading time is 143.3 seconds and the appending time is 111.2 seconds. The total time to load the strings into MonetDB is given as 323.1 seconds, meaning that a lot more computation is done in the background. As a final remark, the total time needed to load the strings with 4 bytes in the file with 417 millions of tuples is 323.1 seconds, to load the strings with 8 bytes is 371.4 seconds and to load the strings with 20 bytes is 458.3 seconds. The time needed to load through the fits library the strings with 4 bytes in the file with 417 millions of tuples is 143.3 seconds, to load the strings with 8 bytes is 163.9 seconds, and to load the strings with 20 bytes is 220.7 seconds. And finally, the time needed to load into MonetDB the strings with 4 bytes in the file with 417 millions of tuples is 111.2 seconds, to load the strings with 20 bytes is 138.8 seconds and to load the strings with 20 bytes is 168.4 seconds. We can easily perceive that the times are increasing while we augment the number of bytes that represent the strings.

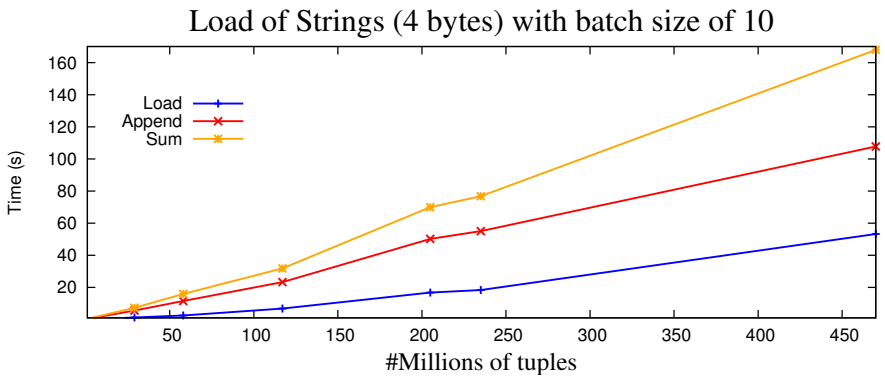

Figure 3.7: Batch 10 for the strings with 4 bytes

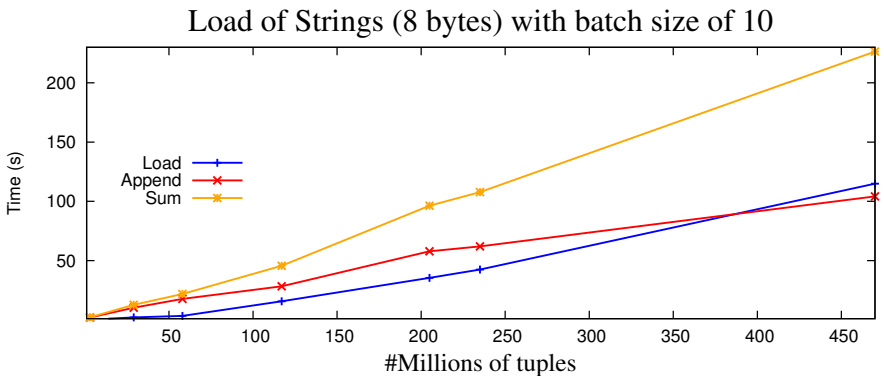

Figure 3.8: Batch 10 for the strings with 8 bytes

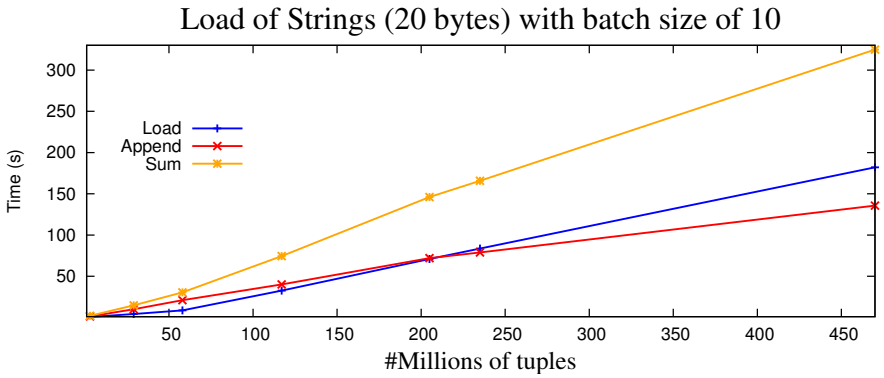

Figure 3.9: Batch 10 for the strings with 20 bytes

For the tests with batch size of 10, represented in **Figure 3.7**, **Figure 3.8** and **Figure 3.9** we got better results. This approach is in fact the first using the third alternative mentioned before, reading a vector of 10 strings at a time. The times are, in fact, faster than the tests performed in the **Figure 3.4**, **Figure 3.5** and **Figure 3.6**, that use the batch size of 1.

In the last file, with 470 millions of tuples, the total time needed to load the strings with 4 bytes is 167.9 seconds. Much faster than the 323.1 seconds used in **Figure 3.4**. The responsible for this decrease in the time is the loading task, performed by the fits library. In the test with the batch size of 1 (**Figure 3.4**), it was 143.3 seconds and in the test with the batch size of 10 (**Figure 3.7**) it was 53.26 seconds. The appending time algo gets faster, being 111.2 seconds in **Figure 3.4** and 107.7 seconds in **Figure 3.7**.

These are significant differences, and they are reflected in the other two tests, for the strings with 8 and 20 bytes (**Figure 3.8** and **Figure 3.9**). Nevertheless, for this last two

tests, an interesting fact occurs.

Note that in the transition from the file with 235 millions of tuples to the file with 470 millions of tuples, represented in **Figure 3.8**, the appending time starts to be faster than the loading time. This happens because MonetDB stops looking if there is a duplicate value in the dictionary of strings when the Heap size, that stores the strings, reaches the maximum size. The maximum size of the Heap corresponds to size of the main memory available in the system. When it reaches the maximum size, it just inserts at the end of the Heap. The insert of values it will be faster, as we can see in the graph, however it will be worse for the lookup operations, that will not be able to use the dictionary as a help.

As for **Figure 3.9**, the same happens, however, the Heap gets full earlier.

In this case in the transition from the file with 205 millions of tuples to the file with 235 millions of tuples. This happens because strings grew from 8 to 20 bytes, filling up the main memory faster.

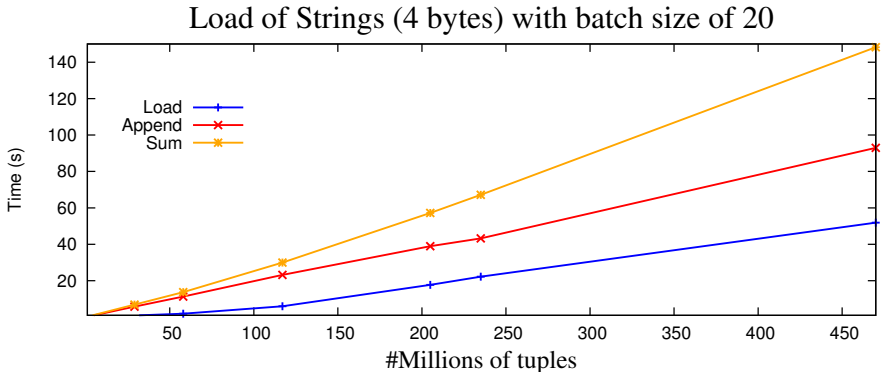

Figure 3.10: Batch 20 for the strings with 4 bytes

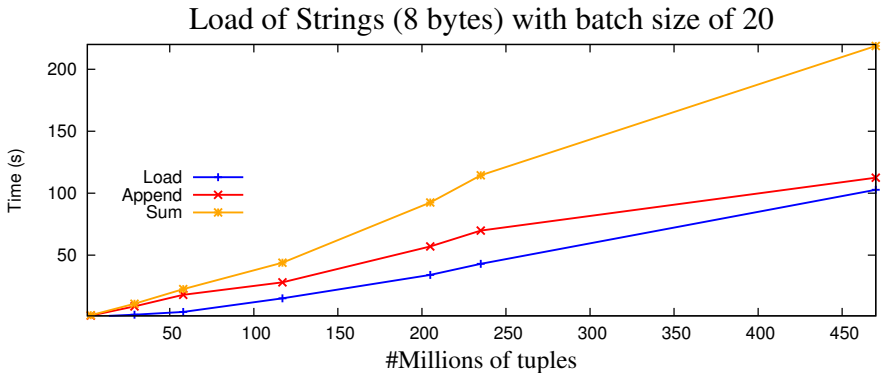

Figure 3.11: Batch 20 for the strings with 8 bytes

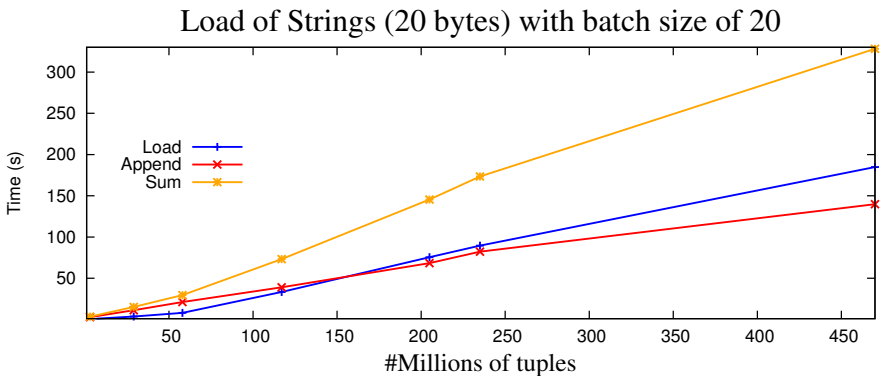

Figure 3.12: Batch 20 for the strings with 20 bytes

The tests with a vector of 20 strings read at a time are represented in **Figure 3.10**, **Figure 3.11** and **Figure 3.12**. For the last file, with 470 millions of tuples, the total time needed to load the strings with 4 bytes is 148.2 seconds. It is an improvement, once the same test, for the batch size of 10, represented in **Figure 3.7**, needed 167.9 seconds. It is also an improvement for the strings with 8 bytes, represented in **Figure 3.11**. For the string with 20 bytes, there is a growth of 4 seconds comparing to the test with the batch size of 10, represented in **Figure 3.9**.

However, the intersection of the appending time with the loading time observed in **Figure 3.8** does not occur in **Figure 3.11**. This happens because the loading time through the fits library gets faster with the batch size of 20 and the appending time into MonetDB has the same behavior. For example, with the batch size of 10, to load the strings of 8 bytes, in the last file with 470 millions of tuples, 114.9 seconds are needed. On the other hand, with the batch size of 20, the same test takes only 102.7 seconds. That difference is enough to avoid the intersection of times.

In **Figure 3.12** the intersection happens again due the loading time, that gets once again slow, being exceeded by the appending time. Nevertheless, it is not enough to be considered a improvement compared to the same test with the batch size of 10, as we said before.

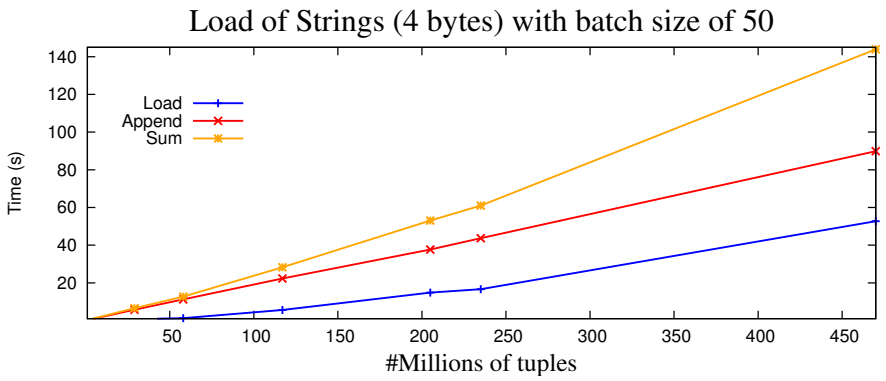

Figure 3.13: Batch 50 for the strings with 4 bytes

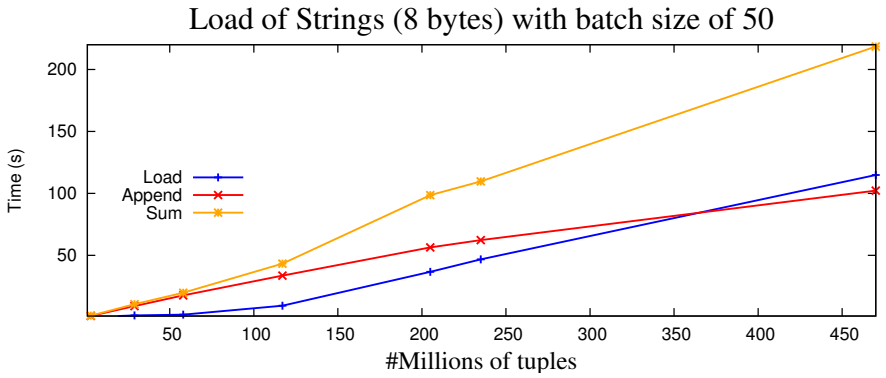

Figure 3.14: Batch 50 for the strings with 8 bytes

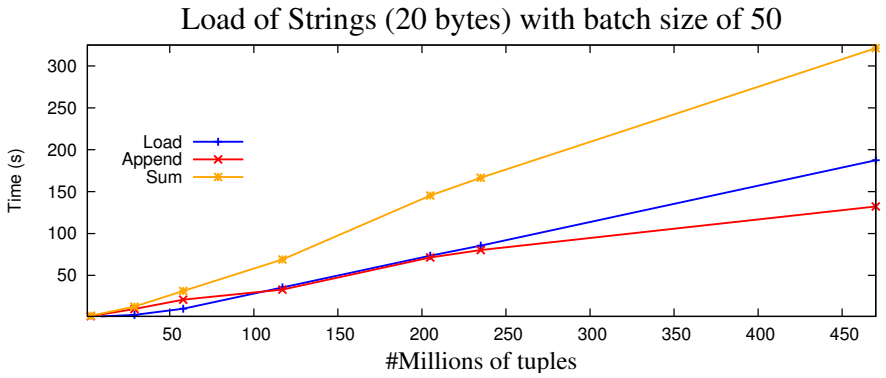

Figure 3.15: Batch 50 for the strings with 20 bytes

In the tests with the batch size of 50, represented in **Figure 3.13**, **Figure 3.14** and **Figure**

3.15, the results improved for all the string sizes. In conclusion, we can claim that this are the best results that we got till now.

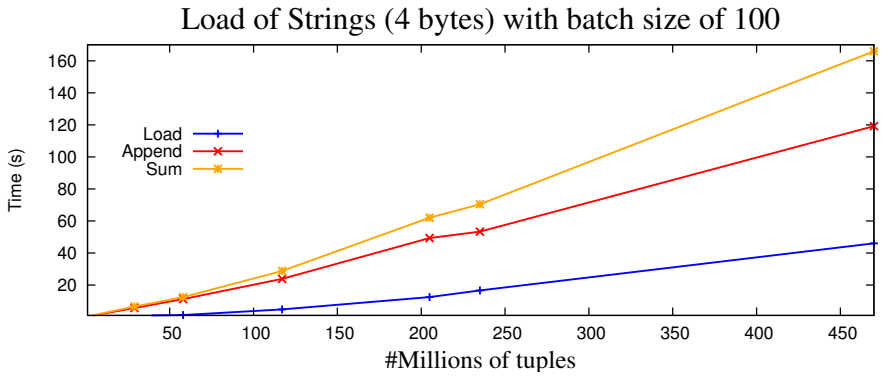

Figure 3.16: Batch 100 for the strings with 4 bytes

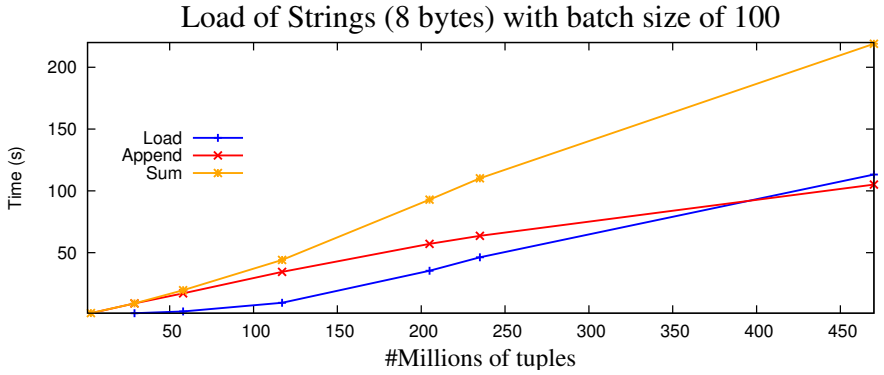

Figure 3.17: Batch 100 for the strings with 8 bytes

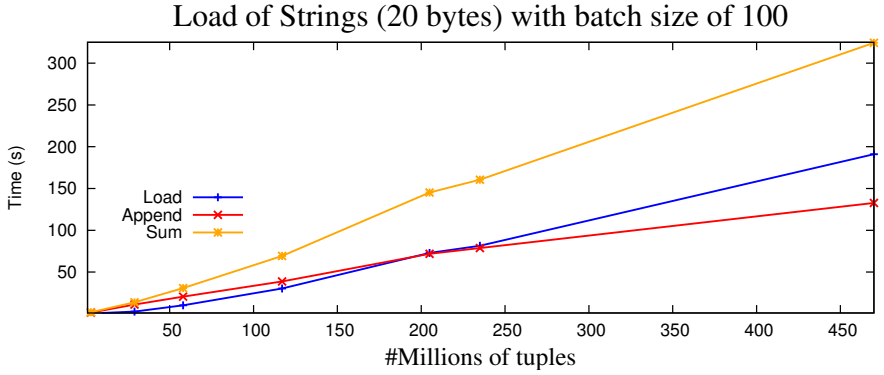

Figure 3.18: Batch 100 for the strings with 20 bytes

For the vector size of 100, represented in **Figure 3.16**, **Figure 3.17** and **Figure 3.18**, the results start to get worse because the size of the vector begins to be too large. With an average of 146 seconds, 218.5 seconds and 323.0 seconds to load the strings of 4, 8 and 20 bytes respectively, with the batch sizes of 20 and 50, as a total time to load the strings for the file with 470 millions of tuples, this tests with a batch size of 100 give us a total time of 165.9 seconds, 219.0 seconds and 324.4 seconds to load the strings of 4, 8 and 20 bytes respectively, in the last test with the file with 470 millions of tuples. As a consequence of this results, it is considered a bad result and it will not be selected.

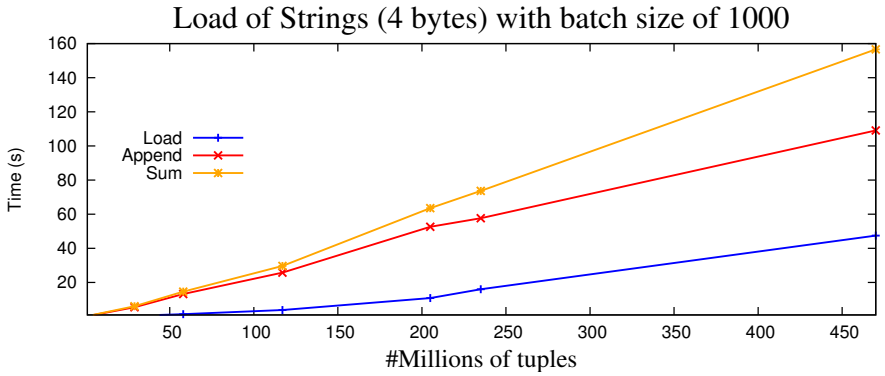

Figure 3.19: Batch 1000 for the strings with 4 bytes

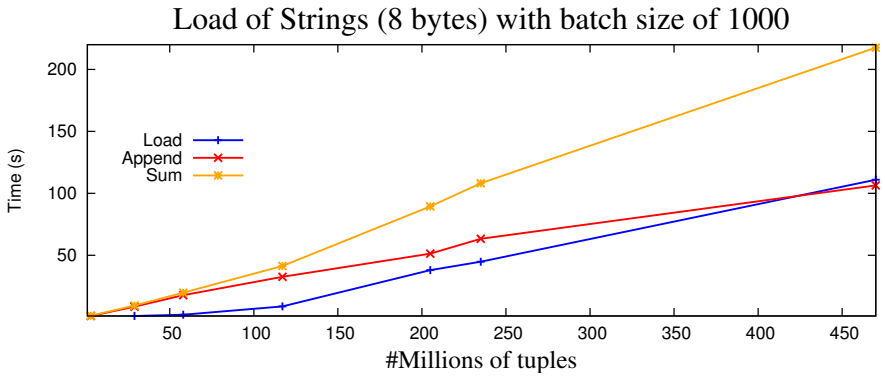

Figure 3.20: Batch 1000 for the strings with 8 bytes

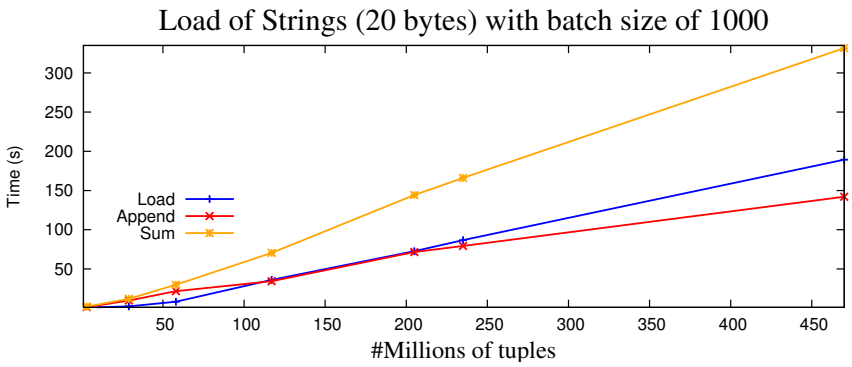

Figure 3.21: Batch 1000 for the strings with 20 bytes

For the last test, with a batch size of 1000, represented in **Figure 3.19**, **Figure 3.20** and **Figure 3.21**, there are no remarkable differences in the times, that make us look for bigger batch sizes, in order to find the perfect one.

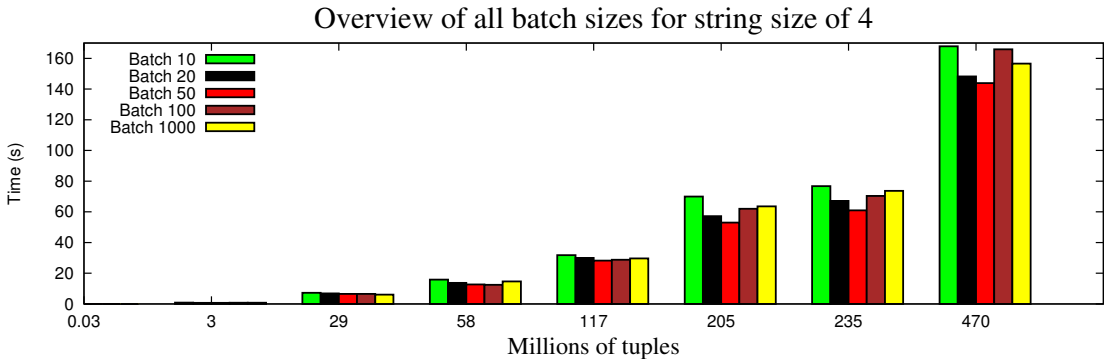

Figure 3.22: Loading strings with 4 bytes

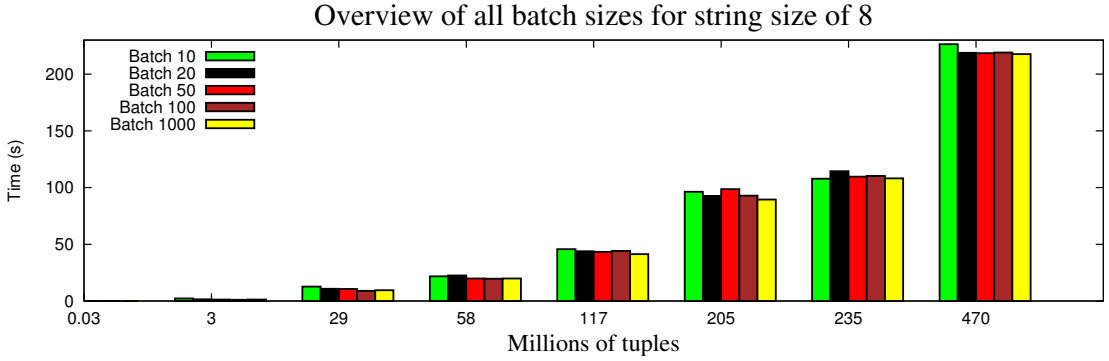

Figure 3.23: Loading strings with 8 bytes

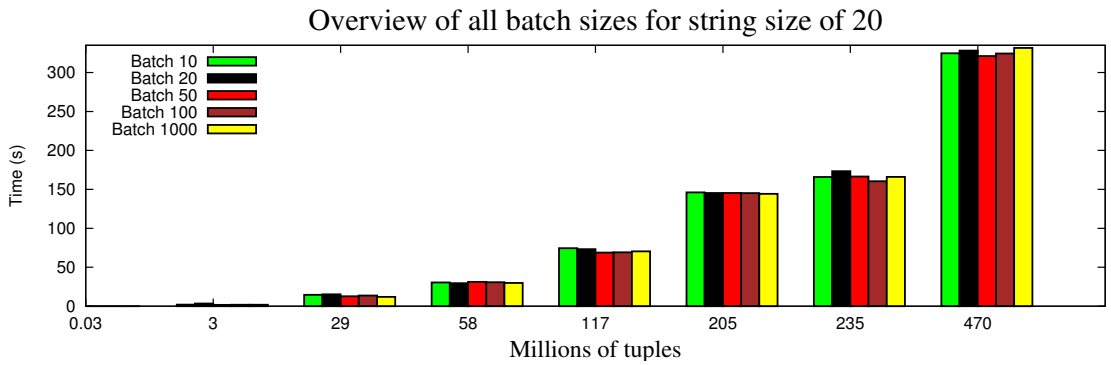

Figure 3.24: Loading strings with 20 bytes

With this experiments we were trying to find a close-to-optimal batch size. **Figure 3.22**, **Figure 3.23** and **Figure 3.24** show that the new approach of different batch sizes improved the loading times in all cases, but the best batch size seems different for diferent file sizes. In **Figure 3.22**, where the strings with 4 bytes were loaded, the batch size of 50 is slightly better than the other batch sizes. As for **Figure 3.23** and **Figure 3.24**, there is no concrete answer about which batch size is actually better. They are all very close to each other and all of them describe the same behavior during the tests.

3.6.2 BAT size representation of Strings in MonetDB

As it was done for the numerical types, also a graph that represents the BAT size in the internal representation of MonetDB it will be done for the string type, with some additional differences. In the following graphs, some additional information will be given:

- Tail size: Total size occupied by the string pointers
- Heap size: Total size occupied by the strings
- Tail + Heap: Sum of the previous two sizes that represent the actual size used to store the string
- Memory: Line thats represents the total memory of the system
- File size: the size of the FITS file where the data was originally

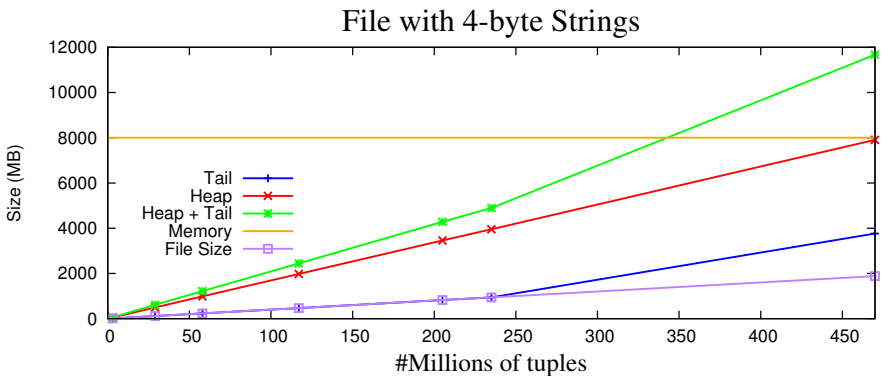

Figure 3.25: Representation of the space occupied by the strings with the size of 4 bytes

Analysing the **Figure 3.25**, we can focus in five main occurrences. First, the linear growth on the FITS file size, obtained by multiplying the number of tuples times the 4 bytes of the string. Second, the transition in the Tail size, when it goes from the file with 235 millions of tuples to the file with 470 millions of tuples. This happens because the 4-bytes used as a pointer to the string are no longer enough. As a consequence, the size of the pointer needs to grow to 8 bytes. With the 8 bytes, there is space to represent all the pointers, however, the space needed to store them its bigger. Third, the Heap size also grows linearly, representing the size used to store the string. It needs 17 bytes to store each string of 4 bytes. Fourth, the memory line is exceeded, passing from the file with 235 millions of tuples to the file with 470 millions of tuples. This leads to future swapping operations, that will increase exponentially the time to execute some queries. The last and fifth fact that needs to be emphasized is the very high overhead of MonetDB for the short strings with 4 bytes: a FITS file with 2 GB takes 12G internally.

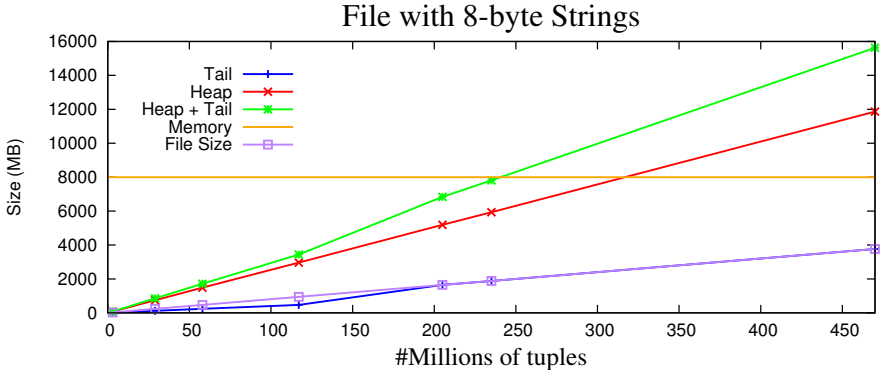

Figure 3.26: Representation of the space occupied by the strings with the size of 8 bytes

For the results presented in **Figure 3.26**, there is one occurrence that must be emphasized, when compared to **Figure 3.25**. The transition in the Tail size that was observed in **Figure 3.25**, when going from the file with 235 millions of tuples to the file with 470 millions of tuples, this time happened in the file with 117 millions of tuples to the file with 205 millions of tuples. This happens because the strings have 8 bytes, instead of 4 bytes like in the previous test, consuming more heap space and also need to increase the address space for the pointers in the tail. As for the Heap size, it requires 25 bytes to store each string of 8 bytes. It uses 8 bytes more per string than in **Figure 3.25**.

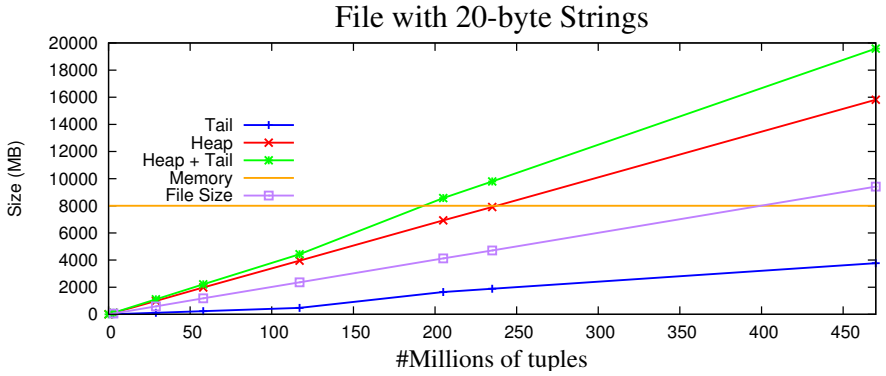

Figure 3.27: Representation of the space occupied by the strings with the size of 20 bytes

In **Figure 3.27**, the transition in the Tail size occurs in the same interval, as the one observed in **Figure 3.26**, however, the bytes used to represent the strings are 20, that leads to a bigger space needed to store the strings. That fact is reflected in the sizes of the FITS files, that is bigger than the memory for the file with 470 millions of tuples. The same happen in the internal representation of MonetDB, that exceeds the memory size in the transition between the file with 117 millions of tuples and the file with 205 millions of tuples. As for the Heap size, it requires 34 bytes to store each string of 20 bytes. It uses 9 bytes more per string than in **Figure 3.26**.

3.7 Export a table

This procedure allows the user to create a completely new FITS file, based on an existing table that is present inside the database. The file will have the name of the table, with the proper ".fits" extension. Once this function is called, the system will check if the table actually exists, giving a proper error message if not.

Assuming that the table exists, once the function is invoked, the properties of the table will be consulted: number of columns, types of the columns and number of tuples. This

information is used to create the header of the HDU containing the metadata about the table. With this information, we then need to collect the BATs that belong to each one of the columns in the table.

The optimal number of rows to write at once in the FITS file depends on the width of the table and the data types of the stored values. There is a routine in *CFITSIO* that will return the optimal number of rows to write for a given table, when given a FITS file as a parameter: *fits_get_rowsize*. We will use that routine to get the value. Each time the block of rows is full, the *fits_write_col* routine is called, and the rows that compose the block are added to the table present in the FITS file. This method of doing the export is much more efficient than calling the function only one time, which generates a big memory consumption. With this method we try to avoid the memory consumption problem, cleaning the block everytime that it is full and repopulating the array with the new data that needs to be added.

As we did before for the loading procedure, some measurements in the time needed to export each one of the types will be performed. We will start with the numerical types and finally we will export to a FITS file the three kinds of strings already studied before. The idea is to have a perspective about the types that take more time to be exported, so it can be manageable a proper future comparison with the existing tools that also provide the same functionality.

The time that will be checked is the one related with the invocation of the fits library function, *fits_write_col*.

The tables that will be exported are the same ones that were loaded into MonetDB in the loading section, with the same number of tuples and with only one column, that represents the target type that is being studied.

In **Figure 3.28** we can realize that this strategy of writting the values with an optimal size, is in fact very efficient. The fastest type to be exported is the short type, due to the 2 bytes in MonetDB internal representation and the slowest is the double type, due to the 8 bytes in MonetDB internal representation. Another important point is the linear grow of the types till they reach the file with 235 millions of tuples. The growth behavior changes in the last file, with 470 millions of tuples, because it requires more memory and swapping operations, which leads to a worse result. As a last remark, the behaviour of the two groups: Integers, Floats and Longs, Doubles. Between them, they have the same behaviour till they reach the 235 million tuples file. For the file with 470 millions tuples, the types that require more computation and complexity (Floats and Doubles), need two or three additional seconds, than the numerical types that share the same number of bytes (Integers and Longs respectively).

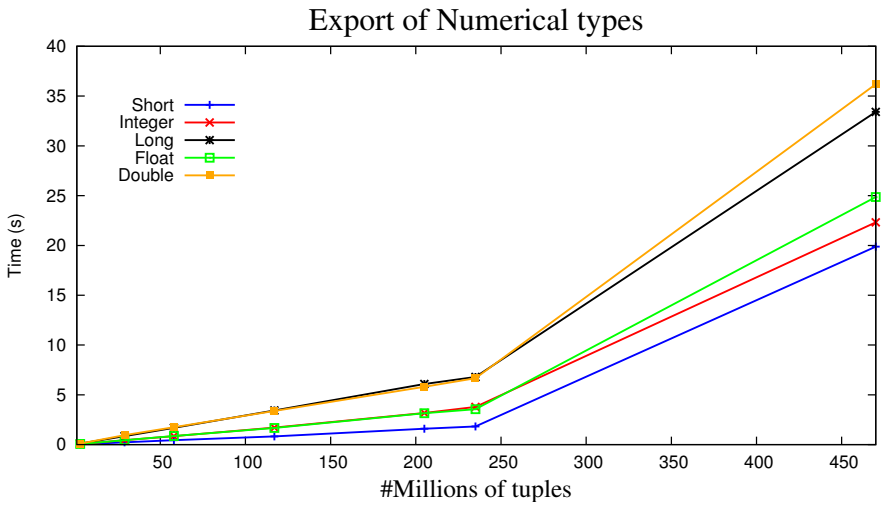

Figure 3.28: Export the numerical types into a FITS file

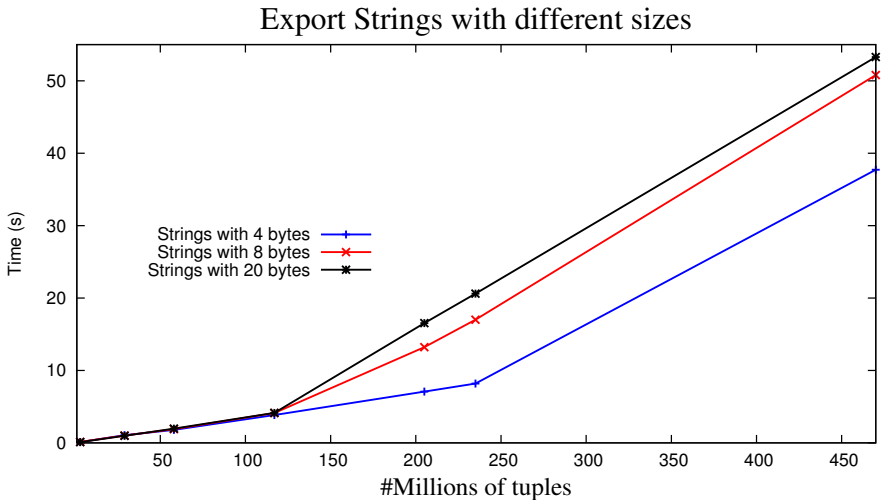

Figure 3.29: Export the strings into a FITS file

In **Figure 3.29** we export the strings with 4, 8 and 20 bytes. For the strings with 4 bytes, the growth is linear till it reaches the file with 235 millions of tuples. After that, it grows faster because in MonetDB internal representation, for the file with 470 millions of tuples, the space needed to store the strings with 4 bytes exceeds the memory capacity, as we can see in **Figure 3.25**. As for the strings with 8 and 20 bytes, that change occurs earlier because to store this strings more space is required, as we can see in **Figure 3.26** and **Figure 3.27**. Consequently, it will also take more time to export this strings.

Chapter 4

Case Study

In this section we will demonstrate, using a concrete example taken from astronomy, that the MonetDB-FITS combination works. That will be done through a step by step tutorial, guiding and explaning in detail to the astronomy user what needs to be done in order to get everything working and ready to be used.

4.1 Overview

We will present to the astronomer how can a specific FITS file be attached, accessing its metadata and spreading the information along several tables present in the SQL catalog. We will also exhibit how can the astronomer attach all the FITS files present in the directory, or even attach all the files present in the directory, giving a specific pattern. Further, we will explain how to load a specific table present in a FITS file that is already attached into the SQL catalog. Also how can we export a table present in our SQL catalog to a brand new FITS file. Finally, once all the astronomical data is present in the database we want to do something with it. As a suggestion proposed by an astronomer working at CWI, we will demonstrate how to execute a set of common astronomical queries in MonetDB. This will be the added value of this section, where we present MonetDB as a capable database engine that is ready to fulfil a set of astronomical demands.

We assume that the user already has MonetDB installed in the system, with its last version, downloaded from the Mercurial repository.

4.2 Attach a file

For this initial task, a FITS file must be present in the file system. Lets assume that the FITS file is called *testattach.fit*, and its directory is '/ufs/fits/'. To attach the file into

MonetDB, the user needs to type:

- `call fitsattach('/ufs/fits/testattach.fit');`

If there are no problems (wrong file name, wrong directory or the attachment procedure was already called for the input file), the user can make sure that all the information about the FITS file is spread along the tables **fits_files**, **fits_tables**, **fits_columns** and **fits_properties**.

4.3 Attach all FITS files in the directory

Let us assume that there is a folder inside the '/ufs/fits/' directory called *astro* that contains the files *testdir1.fit* and *testdir2.fit*. To attach those both files with a single operation, the user needs to type:

- `call fitslist('/ufs/fits/astro/');`

And the user will see that two lines were add to the table called *fits_files*, corresponding to the files that were attached with the call of the function.

4.4 Attach all FITS files in the directory, giving a pattern

The directory is once again called '/ufs/fits/', and it has the folder *astro*, that contains the files *testdir1.fit* and *testdir2.fit*. If the user types:

- `call fitslistpatt('/ufs/fits/astro/','*1.fit');`

With the given pattern, only the file named *testdir1.fit* will be attached. The user can confirm it by checking the table *fits_files*.

4.5 Load a table

Once we have already attached three files in our FITS catalog: *testattach.fit*, *testdir1.fit* and *testdir2.fit*, we are now able to load one of the tables present in the files. By typing:

- `select * from fits_tables;`

We can realize that the *testdir1.fit* has a table called *table1*. This table can be target of our loading operation, through the command:

- `call fitsload('table1');`

The table named ('table1') has become a table of the database, with its metadata, columns and data ready to be queried.

4.6 Export a table

Assuming that during our querying of the table named *table1*, some metadata, columns and data underwent some changes. This led to a creation of a new table called *export1*. That same table can be exported, through the command:

- `call fitsexport('export1');`

This will create a new FITS file called *export1.fit*, with only one table, the one that was aim of our call.

4.7 Cross-matching astronomical surveys

What we have seen so far is a set of mandatory procedures that must be done, to load the data into the database for a further manipulation and querying. Our objective in this section is to continue working with the data that was loaded and use a concrete example of an astronomical use case proposed by an astronomer working at CWI. We want to demonstrate that it is possible to delegate a set of astronomical queries performed by astronomers in their daly activities into MonetDB. The idea is to present an efficient and fast alternative to the actual existing tools. This is needed because this kind of tests requires a lot of computation time, deals with big amounts of data and perform sophisticated algorithms, that only databases are able to provide.

As an example of a complex query we have the *cross-matching* test, that takes two or more surveys as input, and searches for astronomical point-sources that respect a specific set of restrictions, like the distance between sources, calculated using the Cartesian coordinates (x, y, z), that represents a vector in unit length on unit sphere. The idea is to combine results and information of the sources that is reported in the different surveys. With those matching data, information like position and brightness of the sources can be easily comparable. But we have to be aware that, in most cases, we face a substantial number of differences in object detections between surveys and between observations

taken at different times within the same survey or instrument. One difference can be the sensitivity, that is a measure of the minimum signal that a telescope can distinguish above the random background noise. The more sensitive a telescope, the more light it can gather from faint objects. The other distinction than should be considered is the resolution of the survey. The higher the resolution of a telescope, the more details we can see from the images obtained on it [12]. The ability of a telescope to distinguish between, that is, resolve, close objects. For the first tests, we start with data captured by two distinct astronomical radio surveys:

- **FIRST** [26]: stands for Faint Images of the Radio Sky at Twenty-cm and it works at the frequency of 1.4 GHz, has a resolution of 5 arc seconds and a sensitivity of 0.15 mJy
- **SDSS** [4]: stands for Sloan Digital Sky Survey. It is one of the most ambitious and influential surveys in the history of astronomy and it works at the frequency of $4.866x10^5$ Ghz

As we can see, the different surveys work at distinct frequencies. Cross-matching the information between them allows us to compare the values of the brightness (light flux) of the same source, measured at different frequencies. Those values allow the astronomer to study the behavior of the source on the spectrum and classificaty the stars based on their spectral characteristics.

To calculate which is the brightness of the same object at different frequencies, we need to join the two tables derived from FIRST and SDSS, and extract the brightness of the objects measured at different frequencies. The join condition represents a spatial match between two astronomical radio surveys.

The FIRST dataset is a subset that has 18 columns and 285 rows. It was extrated from a vast repository that contains 816331 sources. It was supplied by an astronomer and it covers the region of (RA: 0.048 to 14.80 degrees, DEC: -0.100 to 0.100 degrees). From those columns we are only interested in the following ones:

- *cx*(Double): unit vector of spherical co-ordinates
- *cy*(Double): unit vector of spherical co-ordinates
- *cz*(Double): unit vector of spherical co-ordinates
- *ra*(Double): units: degrees; J2000 Celestial Right Ascension
- *ra_error*(Double): units: degrees; error associated with the Right Ascension
- *dec*(Double): units: degrees; error associated with Declination

- *dec_error*(Double): units: degrees; J2000 Celestial Declination
- *fInt*(Double): represents the integrated brightness of the object. Is an alternative to the peak brightness method, that collects the maximum value of the brightness of the source. The first method, the integrated brightness, calculates the area around the peak value. It is expressed in *mJy* (millijansky). A *Jansky* is a non-SI unit that measures electromagnetic flux density and it is equal to 10^{-26} watts per square meter per hertz. It catches very weak signals.

As for the SDSS dataset, it has 446 columns and 77104 rows. It is a bigger than the FIRST catalog. The idea is to make sure that the entire FIRST dataset is covered, trying to match all the sources of the FIRST catalog with the sources of the STILTS catalog. This catalog covers the area of the sky equivalent to (RA: 1.69e-05 to 15.00 degrees, DEC: -0100 to 0.100 degrees).cFrom the 446 columns we are interested in the following ones:

- *cx*(Double): unit vector of spherical co-ordinates
- *cy*(Double): unit vector of spherical co-ordinates
- *cz*(Double): unit vector of spherical co-ordinates
- *ra*(Double): units: degrees; J2000 Celestial Right Ascension
- *ra_error*(Double): units: degrees; error associated with the Right Ascension
- *dec*(Double): units: degrees; error associated with Declination
- *dec_error*(Double): units: degrees; J2000 Celestial Declination
- *sky_r*(Float): represents the brightness of the object. It is expressed in $maggies/arcsec^2$ and it has to be converted to **mJy**, dividing the value by 3631000

4.7.1 Query 1: Distribution of distances between sources in both surveys

As an initial query, we will join the two tables of both surveys and get all the possible combinations of all the sources. Having that, it is possible to get all the distances between all the sources. Those distances will be given in arcseconds. It is also possible to get some statistical information (distance average and standard deviation). With those two values, once can calculate the Normal Distribution of the distances. That will give an extra information about how the data is distributed.

To calculate the distance between two sources we use the *Euclidean* distance for the three-dimensional space. Assuming that the point a and the point b have the following coordinates: $a = (x1, y1, z1)$ and $b = (x2, y2, z2)$. The distance between them is given by:

$$d(a,b) = \sqrt{(x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2}$$

However, we want the superficial distance, and the formula to calculate it is:

$$sin\frac{1}{2}\theta = \frac{1}{2}d(a,b)$$

That can be simplified to:

$$\theta = 2arcsin(\frac{1}{2}d(a,b))$$

The problem is that the value of θ is given in radians and we want arc seconds. We know that π radians correspond to 180 degrees, so the transition to degrees is easy. To get the value in arc seconds, that is a unit of angular measurement, we need to first calculate the value in arc minutes, that is equal to one sixtieth ($\frac{1}{60}$) of one degree. Knowing that, we just multiply by 60, because 1 arc minute is equal to 60 arc seconds.

Once we have all the distances available, we can calculate the Normal Distribution of the distances between the objects. The formula for the *Normal Distribution* is:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where the parameter μ is the *mean* (location of the peak) and σ^2 is the *variance*. The *mean* is the sum of the values divided by the number of values. The *variance* is used as a measure of how far a set of numbers are spread out from each other and it can be calculated using the formula:

$$\sigma = \sqrt{\frac{\sum_{k=1}^{n} (x_k - \mu)^2}{n}}$$

Where μ is the average and n is the number of values. Information like distance between objects and measurement of the brightness will be required in the further SQL queries. In order to accomplish that goal and give a faster answer to the queries, a SQL table that provides all those parameters will be built:

- CREATE TABLE distances (

  htmid_first BIGINT,

  cx_first FLOAT(51),

  cy_first FLOAT(51),

  cz_first FLOAT(51),

  brightness_first FLOAT(51),

  htmid_sdss BIGINT,

  cx_sdss FLOAT(51),

  cy_sdss FLOAT(51),

  cz_sdss FLOAT(51),

  brightness_sdss FLOAT(51),

  distance FLOAT(51)

  );

Once the table is created, we need to insert all the data of interest, that is spread along both tables and require some calculation and computation.

```
INSERT INTO distances (
    htmid_first, cx_first, cy_first, cz_first, brightness_first,
    htmid_sdss,  cx_sdss,  cy_sdss,  cz_sdss,  brightness_sdss,
    distance
)
SELECT
   tbl1.htmid, tbl1.cx, tbl1.cy, tbl1.cz,
   tbl1.FInt,
   tbl2.htmid, tbl2.cx, tbl2.cy, tbl2.cz,
   (tbl2.sky_r/3631000),
   2*asin(0.5 * sqrt(
                      ((tbl1.cx-tbl2.cx)^2) +
                      ((tbl1.cy-tbl2.cy)^2) +
                      ((tbl1.cz-tbl2.cz)^2)
                    )*180*3600/pi())
FROM first_2 as tbl1, sdss_2 as tbl2;
```

With this query, that took only 34.9 seconds, all the information needed for the further queries is already computed. As a consequence, the answers will be much faster and require less memory.

4.7.2 Distribution of the distances smaller than 45 arc seconds

As a first filter on possible candidates, we will calculate the distribution of distances smaller than 45 arc seconds between the sources of FIRST and SDSS. It is unlikely that two sources are associated, but with this first test we just want to have a general idea about the distribution of the distances between the objects. To keep in mind that the resolution of the survey is an important factor and will impact the number of sources. If the resolution is lower, the number of sources that fulfill the distance will be larger.

The idea is to create a histogram, plotting the number of sources against the distance region in which the sources are apart from each other. For example, the number of sources that are separated from each other with a bin width of 5 arc seconds. To achieve this goal, we will first create and populate a table with the necessary bins to our test:

```
CREATE TABLE bins (min_value INT, max_value INT);
INSERT INTO bins values (0,5);
INSERT INTO bins values (5,10);
INSERT INTO bins values (10,15);
INSERT INTO bins values (15,20);
INSERT INTO bins values (20,25);
INSERT INTO bins values (25,30);
INSERT INTO bins values (30,35);
INSERT INTO bins values (35,40);
INSERT INTO bins values (40,45);
```

Having the table with the bins and the table with the distances available, we can easily create an histogram of the distances, using the following SQL query in MonetDB:

```
SELECT bins.min_value, count(*) as total
FROM bins left outer join distances
ON distances.distance BETWEEN bins.min_value and bins.max_value
GROUP BY bins.min_value;
```

min_value	total
0	110
5	0
10	0
15	0
20	0
25	0
30	0
35	0
40	0

Having this information, the histogram containing the bins and the total number of sources can be plotted.

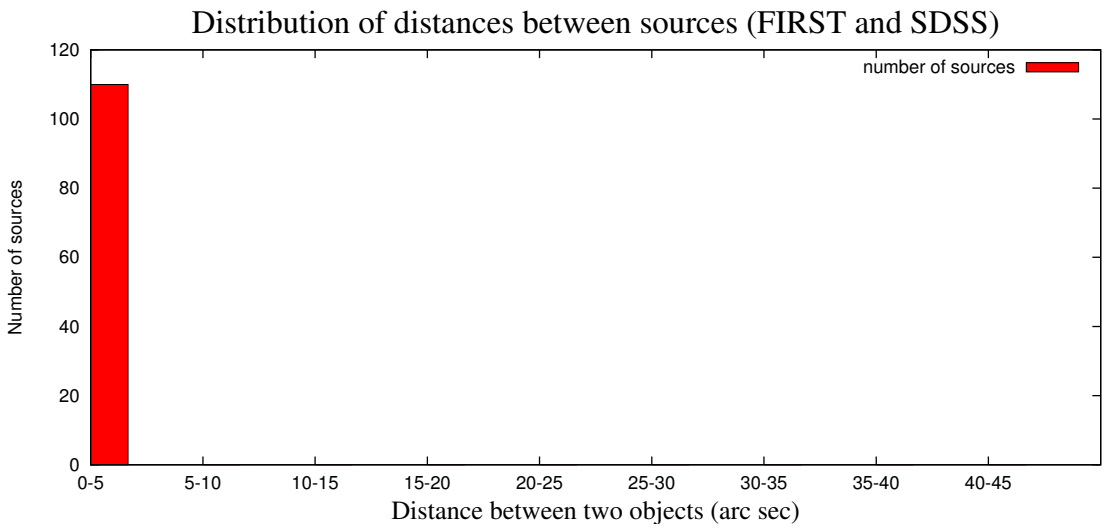

Figure 4.1: Distribution of distances between sources

In **Figure 4.1** we can see the histogram that plots the distribution of the distances between sources. It starts by searching for sources that are apart from each other with a distance between 0 and 5 arc seconds and it finishes by calculating the number of sources that are apart from each other with a distance between 40 and 45. With this information, we have a well-formed idea about how the distances are dispersed. We can easily realize that the distances between the sources are concentrated in the interval 0 and 5 arc seconds.

4.7.3 Normal Distribution of all the data

To obtain the Normal Distribution of the distances between the sources, three steps need to be made: calculate the average of the distances (μ), calculate the total number of distances (n) and the standard deviation (σ^2). To calculate the average in MonetDB we use the following SQL query:

- `SELECT avg(distance) FROM distances WHERE distance <45;`

```
+------------------------+
| average (arc seconds)  |
+========================+
|    0.71174942511540928 |
+------------------------+
```

To calculate the total number of sources that are at a distance smaller then 45, the SQL query in MonetDB is:

- `SELECT count(*) as total FROM distances WHERE distance <45;`

```
+----------+
| total    |
+==========+
|      110 |
+----------+
```

Finally, we need to calculate the standard deviation, with the following SQL query:

```
SELECT
  sqrt( sum((distance-0.71174942511540928)^2) / 110 )  as stdev
FROM distances
WHERE distance < 45;
```

```
+------------------------+
| stdev                  |
+========================+
| 0.57459821453517357    |
+------------------------+
```

Having all the values available is now possible to plot the **Normal Distribution** for the distances.

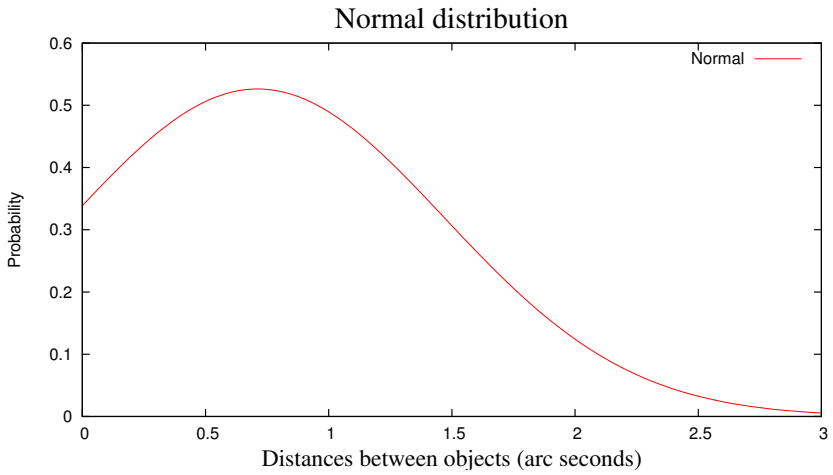

Figure 4.2: Normal Distribution

Now that we have a general idea about the data set we can refine our query for smaller distances. The next section will modify the restriction to 5 arc seconds in the distance between the sources, instead of 45 arc seconds.

4.7.4 Frequency of the distances smaller than 5 arc seconds

The idea is to do the same as we did with the previous data set, but changing the restriction to 5 arc seconds in the distance between the sources. We also need to change the values of the bin width to 0.5, starting in 0 and ending in 5.

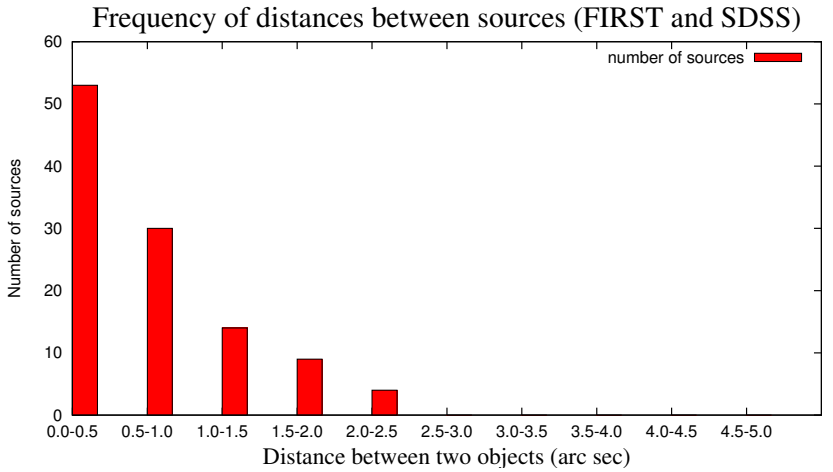

Figure 4.3: Frequency of distances between sources that are less than 5 arc seconds apart from each other

4.7.5 Frequency of the r value between sources in both surveys

On his PhD thesis [22], Bart Scheers presents his idea about source association. He defends that it can be measured through the right ascention (*RA*) and declination (*DEC*), taking into account their respective errors. This values are to the sky what longitude and latitude are to the surface of the Earth. *RA* corresponds to east/west direction (like longitude), while *DEC* measures north/south directions, like latitude. Their values and the respective errors are measured in degrees. The *RA* varies between 0 and 360 degrees. On the other hand, the *DEC* varies between -90 and +90 degrees. There are 60 arcmin in a degree, and 60 arcsec in an arcmin. The formula that Bart uses in this thesis is:

$$r_{i,j} = \sqrt{\frac{(\triangle\alpha)^2_{i,j}}{\sigma^2_{\triangle\alpha,i,j}} + \frac{(\triangle\delta)^2_{i,j}}{\sigma^2_{\triangle\delta,i,j}}}$$

Where α represents the *RA*, δ the *DEC* and σ^2 their respective errors. In order to store the values of interest, we will create a table only with the fields that we are interested in:

- ```
  CREATE TABLE rvalues (
      htmid_first BIGINT,
      ra_first FLOAT(51),
      dec_first FLOAT(51),
      ra_error_first FLOAT(51),
      dec_error_first FLOAT(51),
      htmid_sdss BIGINT,
      ra_sdss FLOAT(51),
      dec_sdss FLOAT(51),
      ra_error_sdss FLOAT(51),
      dec_error_sdss FLOAT(51),
      rvalue FLOAT(51)
      );
  ```

Once we have the table, we can populate it with values.
  ```

```
INSERT INTO rvalues (
    htmid_first,
    ra_first,
    dec_first,
    ra_error_first,
    dec_error_first,
    htmid_sdss,
    ra_sdss,
    dec_sdss,
    ra_error_sdss,
    dec_error_sdss,
    rvalue
)
SELECT
   tbl1.htmid, tbl1.ra, tbl1."dec", tbl1.ra_error, tbl1.dec_error,
   tbl2.htmid, tbl2.ra, tbl2."dec", tbl2.ra_error, tbl2.dec_error,
   sqrt(
    (((tbl1.ra-tbl2.ra)^2) / (tbl1.ra_error^2 + tbl1.dec_error^2))+
    (((tbl1."dec"-tbl2."dec")^2) / (tbl2.ra_error^2 + tbl2.dec_error^2))
   )
FROM first_2 as tbl1, sdss_2 as tbl2;
```

This query only took 23.8 seconds. Once again, all the information needed for further queries is already computed. We will not proceed with further tests. We only want to demonstrate how to compute this values, so they can be used in the future, avoiding extra and unnecessary computation.

4.7.6 Query 2: extract & compare brightness in different frequencies

In the introduction section we talked about the brightness and how it changes with the different frequencies. In this section we will join the two tables from the FIRST survey and the SDSS survey and extract their brightness measurements.

The SQL query that is used by MonetDB, once the table is loaded into the Database Management System is:

```
SELECT
    brightness_first as brightness1, log10(brightness_first) as log1,
    (brightness_sdss) as brightness2, log10(brightness_sdss) as log2
FROM distances
WHERE distance < 1;
```

Where **brightness1** and **brightness2** represent the brightness measurement in each one of the surveys and the values of **log1** and **log2** are the logarithm of **brightness1** and **brightness2** and they are calculated in order to provide both scales in the graphic representation. The distance between the sources was changed for 1 arc second because with this value it is more likely that the object is the same and also the number of lines in the graph is less, making it understandable and readable.

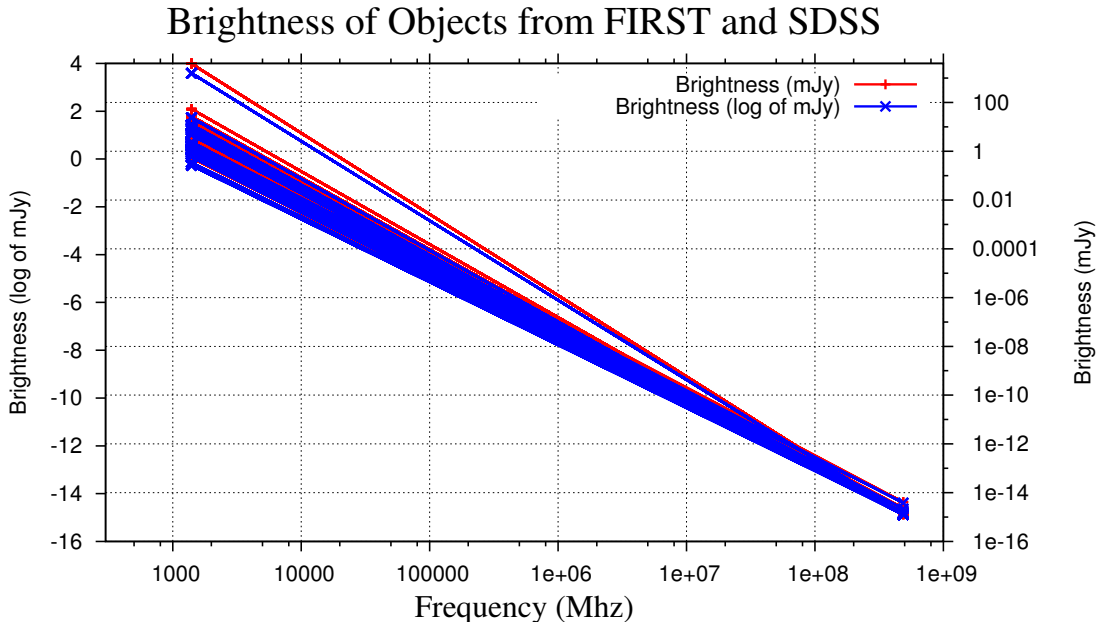

Figure 4.4: Measure of the brightness in different frequencies

Figure 4.4 represents the brightness of each one of the objects, that are separated from each other by 1 arc second, measured at different frequencies. We can easily realize that the brightness of the objects is higher for the frequencies of 1.4 GHz and lower for the frequencies of $4.866x10^5$ GHz.

4.7.7 Query 3: extract the spectral index

Having the values of the brightness of each object, measured at different frequencies, it is now possible to calculate the spectral index. In astronomy, the spectral index of a source is a measure of the dependence of radiative flux density on frequency. Given frequency ν and radiative flux S , the spectral index α can be calculated as:

$$\alpha = -\frac{log(Sf - Ss)}{log(\nu f - \nu s)}$$

Where Sf and Ss represent the brightness of the sources in the FIRST and SDSS surveys respectively and νf and ν **s** the frequencies. As we did for the distances and the brightness, the same procedure of creating a table that calculates the spectral index will be created.

```
CREATE TABLE spectal_indexes (spectal_index FLOAT(51), distance FLOAT(51));
```

Once the table is created, we will insert the values of the spectral index and the distance between the objects into the table.

- ```
  INSERT INTO spectal_indexes (spectal_index, distance)
  SELECT -(
      (log10(brightness_first)/(log10(brightness_sdss))) /
      (log10(1400)/(log10(486281359)))),
      distance
  FROM distances where distance <1;
  ```

4.7.8 Distribution of the spectal index

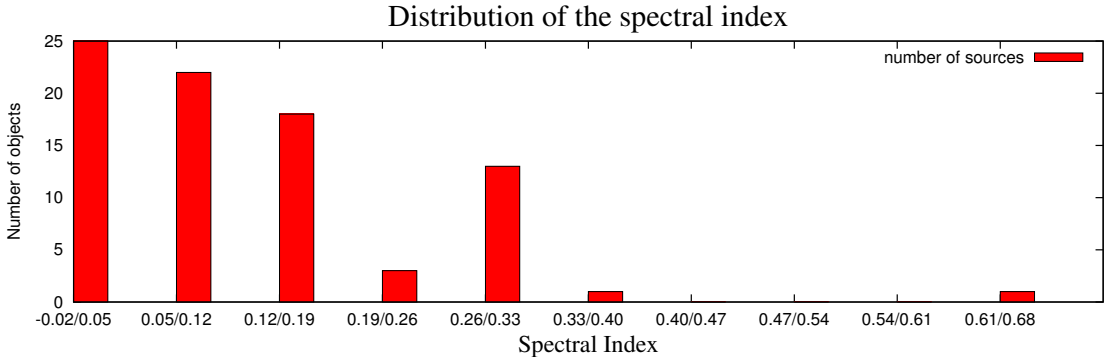

Figure 4.5: Calculation of the spectral index
  ```

Figure 4.5 represents the frequency of spectral indexes calculated. The same procedure was used as for the frequency of the distances, in order to build a histogram. The minimum value of the spectral index is -0.016980525624847961 and the maximum value is 0.66733249647620008. Knowing this two values, the histogram can be created. It will have 10 intervals, adding 0.07 units in each iteration. We can notice that, for this experiment, the most frequent spectral index rounds the values -0.02 and 0.19.

4.7.9 Normal distribution of the spectral indexes

As a next step, we will calculate the Normal Distribution of the spectral index. As in the previous sections, we need the average of the spectral index and how much the objects deviate from it. For that, first we need a SQL query that calculates the average:

```
SELECT avg(spectal_index) as average FROM spectal_indexes WHERE distance < 1;
```

```
+------------------------+
| average                |
+========================+
|    0.13056802784870311 |
+------------------------+
```

Knowing the average is now possible to calculate the standard deviation and the total number of tuples. That is done through the following SQL queries:

```
SELECT count(*) as total FROM spectral_indexes WHERE distance < 1;
```

```
+------+
| total|
+======+
| 83   |
+------+
```

```
SELECT
  sqrt( sum((spectral_index-0.71174942511540928)^2) / 83 ) as stdev
FROM spectral_indexes
WHERE distance < 1;
```

```
+------------------------+
| stdev                  |
+========================+
| 0.57459821453517357    |
+------------------------+
```

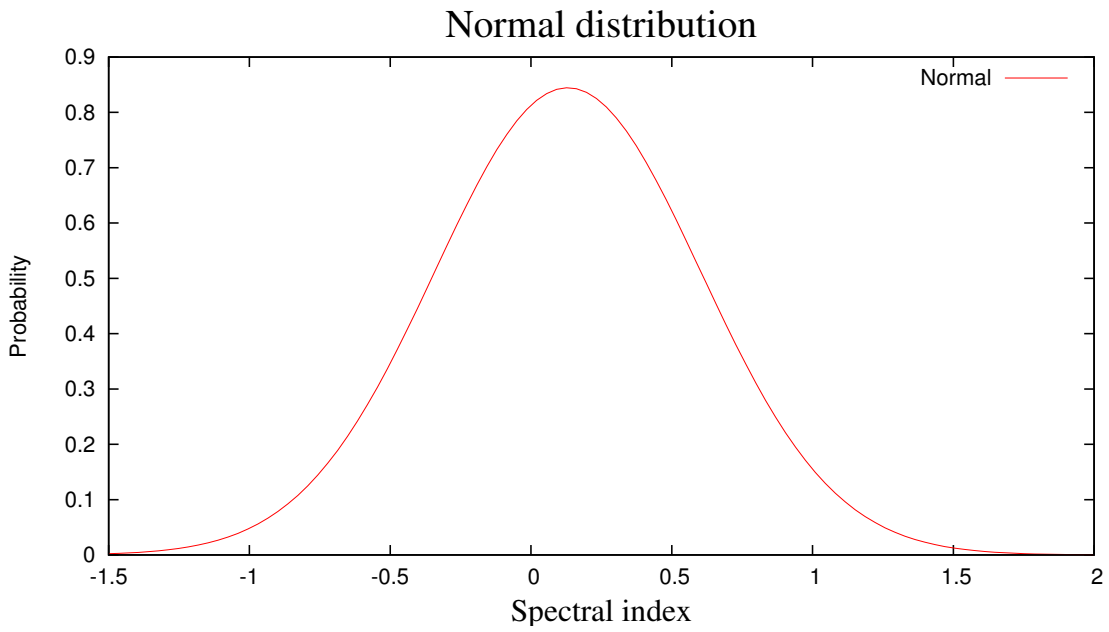

Figure 4.6: Plot of the Normal distribution of the spectral index

Chapter 5

Performance Experiments

For this section of performance experiments, a stress test was done in MonetDB and STILTS to check the performance of both systems when accessing data present in FITS files. *STILTS* stands for **Starlink Tables Infrastructure Library Tool Set** [24] and it is a set of command-line tools for table manipulation. More details about this program will be further explored in the **Related Work** section. For now, we will only present the tool and explore in more detail when appropriated.

5.1 Experimental Setting

The tests were done on a machine with 8GB of memory, 4 CPU 2.83 GHz and 300 GB of free disk space. For each one of the tests, the performance of the tools is checked in hot and cold memory. Hot memory means that data are already loaded in memory. Cold memory means that the memory is clean and all data need first to be brought from disk to memory to provide the final result.

Some relational algebra expressions are applied on the tables present in the FITS files. It consists of a set of operations that take one or two tables as input and produce a new table as output. As some example of those operations we can have:

- The select operation: selects tuples that satisfy a given predicate or condition. In STILTS, they are identified by filtering operations
- The project operation: returns only the columns of interest
- The cartesian product operation: allows combining information from two tables
- The natural join operation: it is a binary operation and a combination of certain selections and a cartesian product into one operation

- The aggregation operation: in SQL is called *group by* and include five aggregate functions: Sum, Count, Average, Maximum and Minimum
- The Sorting operation: in SQL is called *order by* and has the goal to sort the tuples of a input column

STILTS is a powerful tool, that can be used to manipulate and access astronomical data present in a variety of different formats. In this chapter we are only interested in the FITS format and how efficient this tool is with FITS files. In *STILTS*, for the *filtering* operations, the table data is streamed through the pipeline, using the command *tpipe*. One *row at a time* is checked, either using a *sequential mode* or a *random mode*. They can be explicitly set in the command line. We tested both modes and the results were similar, either for small and big files, so we just choose the default mode to run the tests. Using only one method (check a row at a time) for this kind of operations, we expect similar results for STILTS in each one of the tests.

As for *MonetDB*, the purpose of this tests is to load the tables with data, present in the FITS files and perform some main relation algebra operations with them. *MonetDB* needs to invest time to load FITS data into its internal data structures. This operation is done only once, and the time that it takes will appear only in the graphs that measure the cold memory operations, because it involves all the loading of the data to main memory. For the hot memory tests, MonetDB already has the data present into its tables and the processing times will be amortized. The methods that MonetDB uses to perform the relational algebra select operations vary between sequential scan and hash table structures.

For that, four different times were measured:

- the time for MonetDB to load the table
- the time for MonetDB to execute the Query
- the sum of both times (Query + Load)
- the time for STILTS to access the data and execute the Query

5.2 Test Files

Experiments are divided in three groups. The first group was already presented and it is described in **Table 3.1**. We reuse this set of FITS files because we want to study the behavior of an individual specific type. For the types that represent numerical values, the tests that will be performed are: Point Query, Range Selection of data and Statistical information (average, minimum and maximum) tests. For the String type, only the Point

Query test will be performed, since the range and statistical tests cannot be performed in strings.

The second group of tests is based on the creation of a large file, that contains a table with all the column types described in the first group. The set of FITS files that is created is based on the number of rows used in the first group, resulting in the following sizes:

Megabytes	Number of Rows
1	30000
100	2942000
1000	29420000
2000	58840000
4000	117680000
7000	205940000
8000	235360000
16000	470720000

Table 5.1: Second group of FITS files

For this large file, represented in **Table 5.1**, the tests that will be performed are: Point Query, Range Selection, Projection of a specific column and Statistics Information (average, minimum and maximum) of each column.

For the third and last group, comparison of join features of STILTS and MonetDB will be performed. For that, the size and the number of rows of each file is described in the following table:

Megabytes	Number of Rows
1	30000
10	300000
30	900000
50	1500000

Table 5.2: Third group of FITS files

In **Table 5.2** the files are smaller than the files used in group one and two because a lot more computation is needed for join operations and also because it is just impossible to perform such queries in STILTS when the files reach a particular size. Another change comparatively to the previous two groups, is that only hot memory tests were considered. The reason for that is because the files are relatively small, and the time to bring them to memory does not result in a subtantial difference between hot and cold memory.

STILTS comes with a set of routines (tmatch1, tmatch2, tmatchn) for various join operations over tables. The routines are parameterized with the type of the join condition, so called matcher, and can be grouped in the following categories:

- *exact matcher*: corresponds to the equi-join in DB terms.
- *isotropic* (1d, 2d) and *anisotropic* (2d) matcher: considers two rows as matching if the difference between the values of their keys is less than a given *error*. It corresponds to the more general theta-join.
- *sky and skyerr matchers*: apply join conditions customized for the astronomical spatial searches.

In STILTS documentation, joins are mentioned as crossmatching operations, where the goal is to identify different rows, which may be in the same or different tables, that refer to the same item. A crossmatching is performed in three steps. First, define what is the condition that must be satisfied for two rows to be considered matching. The condition can be one of the matches presented before. Second, decide what happens if, for a given row, more than one match can be found. The user can choose between *Best Match Only* and *All matches*. And third, decide what to do with the matching rows. The user can count them, present them as an output table or create a new FITS file with the produced table.

Matching can in general be a computationally intensive process. The algorithm used by the tmatch* tasks in STILTS, scales as **O(N log(N))** (*loglinear* complexity), where **N** is the total number of rows in all the tables being matched. No preparation (such as sorting) is required on the tables prior to invoking the matching operation. It is quite fast for small tables. The same scenario do not occur for bigger tables, where can easily run out of memory.

5.3 Delegation experiments

In this section we illustrate the effect of delegation some of the operations that can be done by some existing tools that work with FITS files into MonetDB.

We will start with selection, filter and statistical delegation, that will be performed by the first and second group of FITS files and then we will move to a theta and equi join delegation, performed by the third group of FITS files.

5.3.1 Selection and Filter delegation for Group number 1

For the first group, the first test is the Point Query. Starting with the types that are used to represent numbers, the general structure of the SQL query in MonetDB is:

- `select count(*) from binary_table where Value=Number;`

and in STILTS:

- ./stilts tpipe in=binary_table.fit cmd='select Value==Number' omode=count

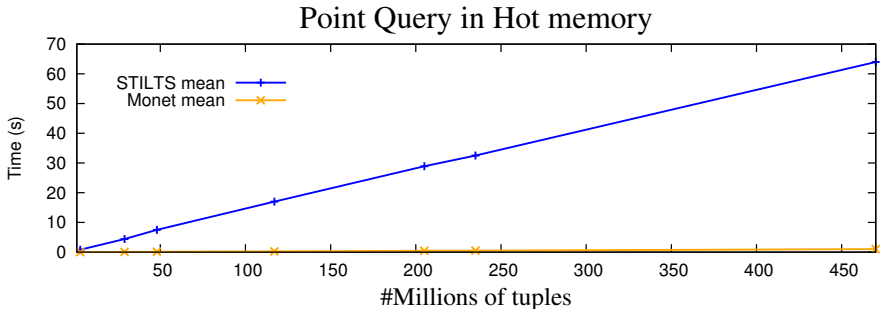

Figure 5.1: Performance of MonetDB and STILTS in Point Query operations with numerical types

Figure 5.1 represents the Point Query test of the numerical types, in hot memory, for MonetDB and STILTS. We grouped all types together, calculating the average behavior of STILTS and MonetDB. We did it because the times needed by the tools to perform the tests were very similar along the different types. Both tools have a linear behavior during the tests. Although, we have to highlight the performance of MonetDB, that is always better than STILTS, reaching the query time of one second only in the last test. As for STILTS, it takes an average of 64 seconds to perform the last test.

As for the cold memory tests, we assign a graph to each one of the types, because it involves loading of the tables and MonetDB behaves differently for each one of the numerical types. The first type is the *Short*.

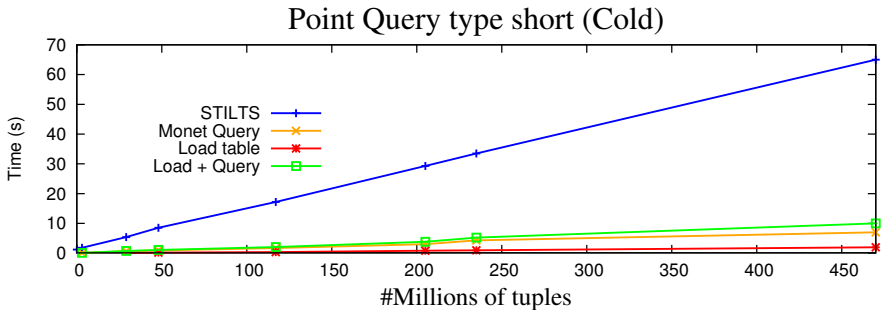

Figure 5.2: Performance of MonetDB and STILTS in Point Query operations for short type

Figure 5.2 represents the Point Query test of the type short, cold memory, for MonetDB and STILTS. Both tools have a linear behavior during the tests. MonetDB is superior in all the tests, even when the loading time is added to the query time. As for the second type that can be used to represent numbers, we will use **Integers**.

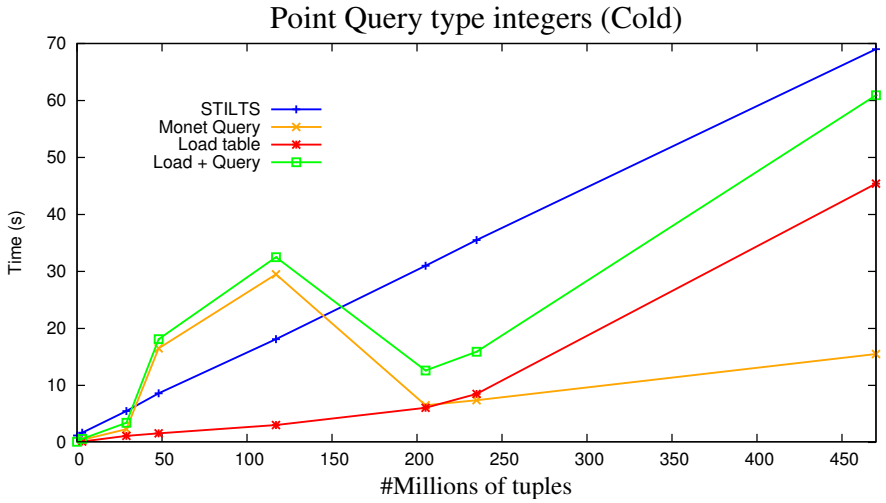

Figure 5.3: Performance of MonetDB and STILTS in Point Query operations for integer type

Analysing **Figure 5.3**, two main points can be identified: the increased loading time when compared to the type short; and the behavior in the performance of MonetDB Query when it deals with the interval of files starting in the one that has 29 millions of tuples and ending in the one that has 205 millions of tuples.

The increment of the loading time is expected, once the size of the data increases from 2 bytes, in the **short** type, to 4 bytes, in the **integer** type. As for the behavior of MonetDB query, a trace was done to check which algorithms were being used. It happens that most of the time is going to the **uselect** statement. The **uselect** statement is the one responsible for building the hash table. Once the hash table is created, the queries will be much faster, because the result is already in memory structures as a hash table. Knowing that, and observing the times in **Figure 5.3**, we can deduce that in the interval of files starting in the one that has 29 millions of tuples and ending in the one that has 205 millions of tuples what happen is that MonetDB tries to build the hash table and it fails, it tries again and it fails. It fails three or four times. We say this because the results are 4-5 times bigger than the expected. As for the other files, it builds the hash table in the first try. The reason for this failure still need some more debugging into MonetDB code. For now is still unclear.

The third type chosen to represent numerical numbers is the type **Long**.

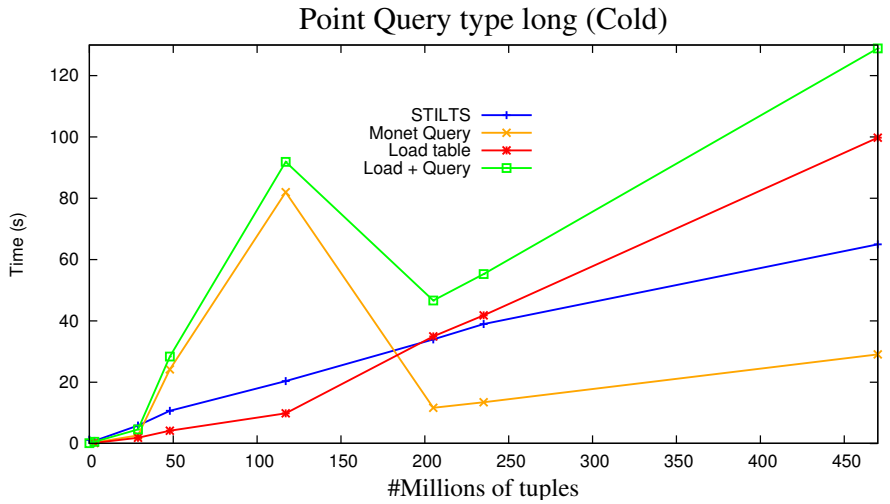

Figure 5.4: Performance of MonetDB and STILTS in Point Query operations for long type

Figure 5.4 represents the Point Query test for the type long, in cold memory. The same problem that was found for the integer type, described in **Figure 5.3**, also occurs for the **long** type tests. The peak on the computation time is even more accentuated in this test and the loading time of the table is also bigger, due to the 8 bytes size that the **long** type requires. The combination of both factors leads to a total time spend by MonetDB (load plus SQL query) worse than the computation time needed by STILTS.

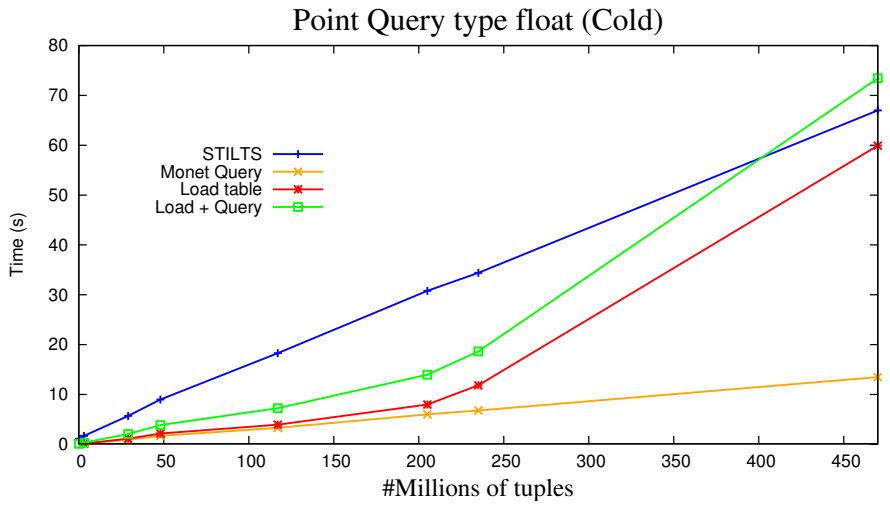

Figure 5.5: Performance of MonetDB and STILTS in Point Query operations for float type

The **Float** type is used to represent floating point numbers and it occupies 4 bytes. **Figure 5.5** shows the performance of MonetDB and STILTS in the Point Query test for the type float, in cold memory. There is a similarity between the loading time behavior of the integers, described in **Figure 5.3**, and the loading time of the floats, described in **Figure 5.5**. This happens because both types use 4 bytes in their MonetDB internal representation. Nevertheless, the floating points are more complex to store, leading to worse performance in the loading of the tables when compared to integers.

As a consequence, adding the loading time to the query time, the performance of MonetDB is worse than the performance of STILTS for the last test, with the 470 million tuples file. For the other files, the performance of MonetDB is always better than STILTS. The last Point Query test takes the type **double** as an input.

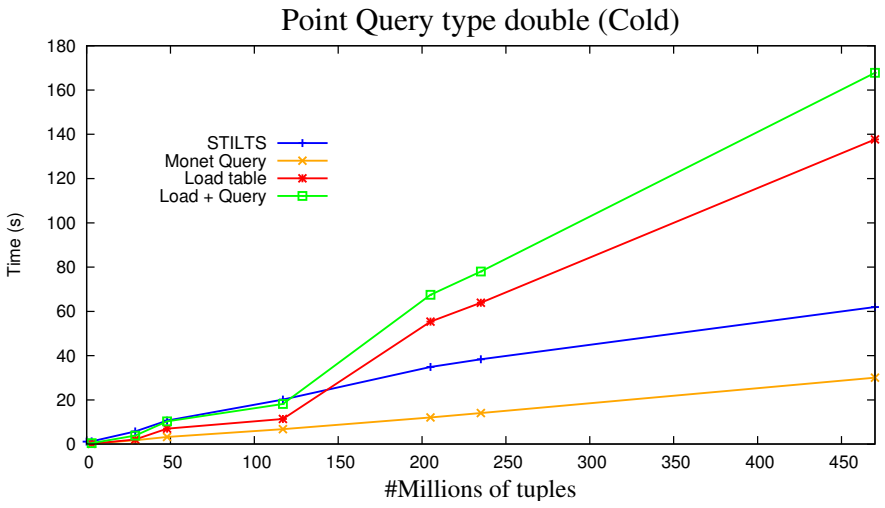

Figure 5.6: Performance of MonetDB and STILTS in Point Query operations for double type

As mentioned before, the similarity between the loading times behavior of the floats and the integers happens because both have a MonetDB internal representation of 4 bytes. Therefore, the same happens for doubles and longs. Both have 8 bytes in their MonetDB internal representation and both have a similar behavior in their loading time process, however, the double type is used to store floating point values. Consequently, there is a little additional time with doubles due to their complexity.

Finished with the numerical types, we will now proceed to the string type. Once again, all the information will be gathered in one graph, for the hot memory tests. This makes it simpler and comparable along the different string sizes. As for the cold memory tests, we will maintain one graph for each of the string sizes. The general structure of the

SQL query in MonetDB is:

- `select count(*) from binary_table where A='string';`

and in STILTS:

- ./stilts tpipe in=binary_table.fit cmd='select A=="string"' omode=count

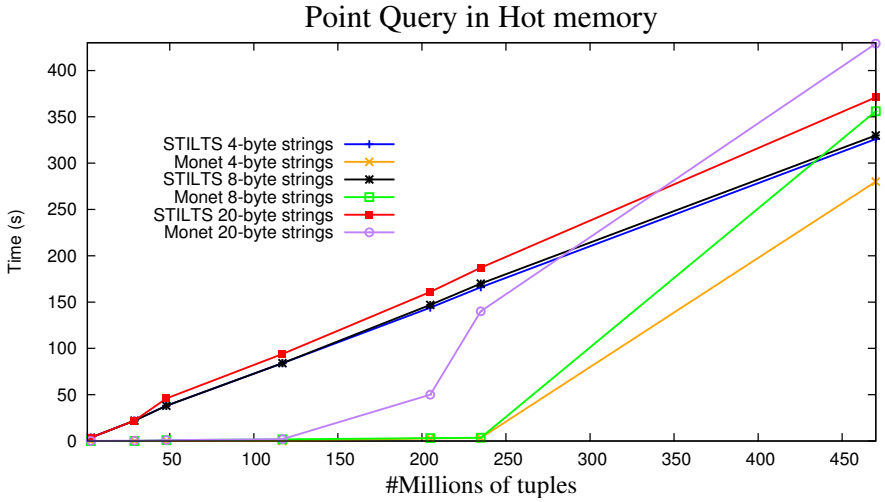

Figure 5.7: Performance of MonetDB and STILTS in Point Query operations with different string sizes

Figure 5.7 represents the Point Query test of the strings with different sizes, in hot memory, for MonetDB and STILTS. For the tests with the 4-byte strings, we conclude that MonetDB is always faster than STILTS, for all the file sizes tests. The behavior of STILTS during the tests is linear. However, the time in MonetDB gets exponentially bigger moving from the 235 million tuples file to the 470 million tuples file. This happens because the system runs out of memory and needs to perform swapping operations.

Another possible explanation for the occurrence can be seen in **Figure 3.25**, where a representation of the BAT size in MonetDB of the strings with 4 bytes is displayed. We can see that for the last file, with 470 million tuples, the total size to represent the strings exceed the memory capacity and the system needs to execute swapping operations, which makes the time of the query grow.

As for the tests with the 8-byte strings, we can easily realize that MonetDB also runs out of memory in the last file. However, the performance of STILTS is similar. This happens because once again, the BAT size of the strings in MonetDB, for the last file, exceed

the memory capacity (**Figure 3.26**). As for STILTS, it takes as input the file present in the file system, that is smaller than the memory capacity. This leads to a worse performance of MonetDB in the last test, comparing to the behavior of STILTS.

Finally, the test with the 20-byte strings. As we can see in the BAT size representation of the strings with the size of 20 bytes, **Figure 3.27**, memory capacity is exceeded in the interval of the files with 205 and 235 millions of tuples. It is precisely in that interval that the time grows exponentially.

Completed the tests in hot memory, we will now proceed for the string tests in cold memory.

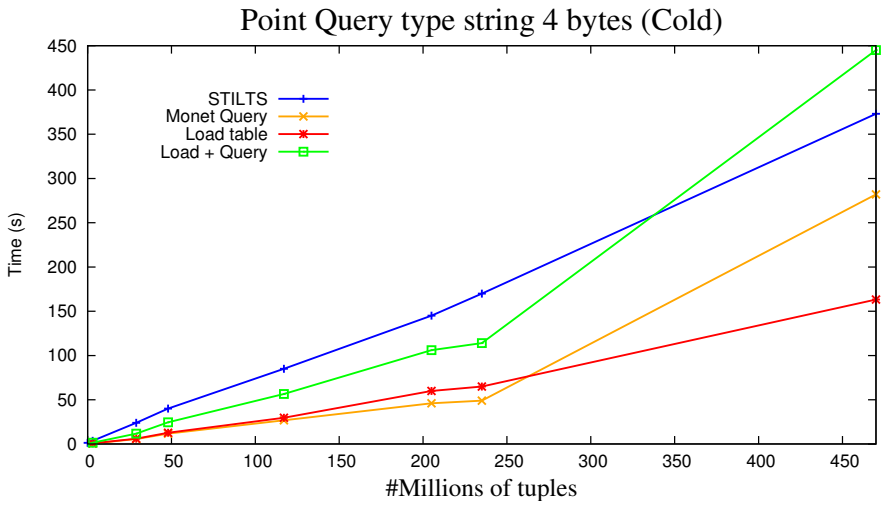

Figure 5.8: Performance of MonetDB and STILTS in Point Query operations with single 4-byte string column

Figure 5.8 represents the Point Query test for the strings with the size of 4 bytes, in cold memory for MonetDB and STILTS. We conclude that MonetDB is always better than STILTS, even when the loading time of the table is added to the query time. Nevertheless, due to the exponential increase of query time in the last test, already observed in the hot memory test for the 470 million tuples file, added to the loading time of the table, the total time of MonetDB gets worse than STILTS. The next Point Query test it will involve strings with the size of 8 bytes. **Figure 5.9** depicts the Point Query test in cold memory for the 8-byte size string. The performance of STILTS is linear and similar to the previous tests. As for the performance of MonetDB, it is worse comparing to **Figure 5.9** because both loading and query times increased, leading to a bigger total time, and, consequently, a worse result comparing to STILTS. The same for the tests with the string size of 20 bytes, represented in **Figure 5.10**.

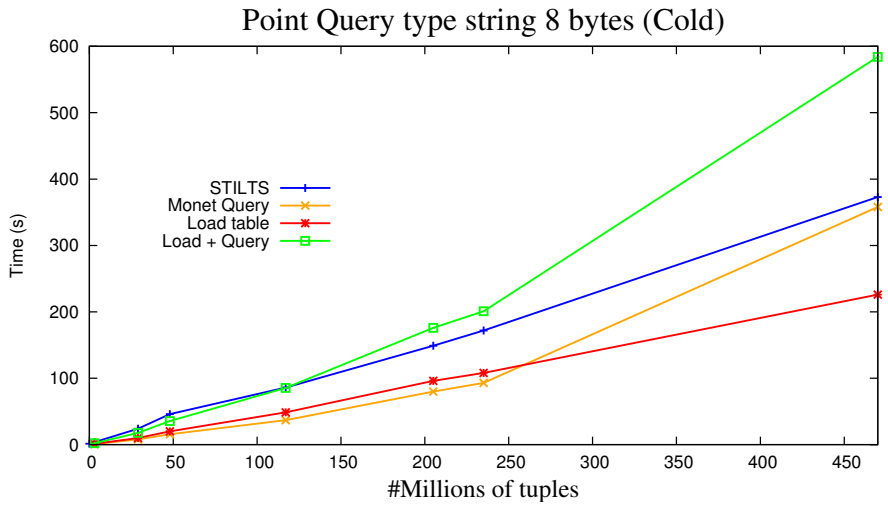

Figure 5.9: Performance of MonetDB and STILTS in Point Query operations with single 8-byte string column

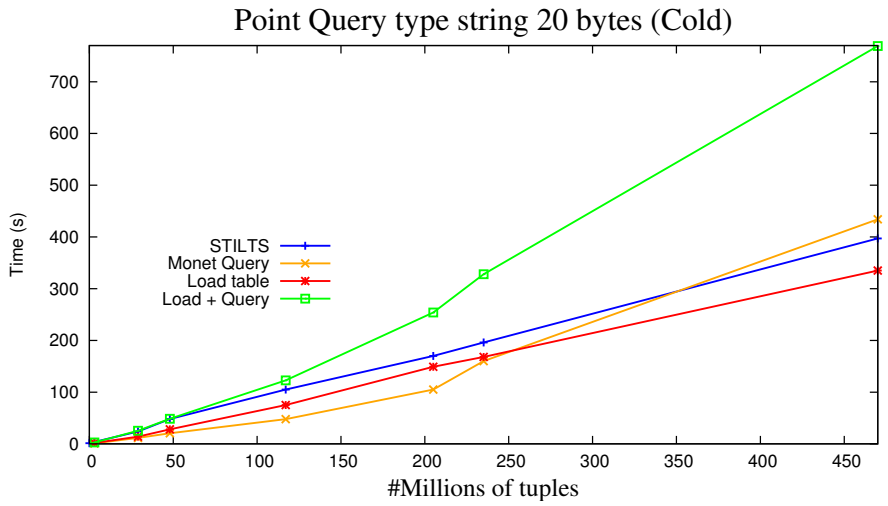

Figure 5.10: Performance of MonetDB and STILTS in Point Query operations with single 20-byte string column

5.3.2 Range Delegation for Group number 1

For the range delegation, only the types that represent numerical values will be target of the **Range** test. They are the **Short**, **Integer**, **Long**, **Float** and **Double** types. For each table, the query collects 5% of all the data. For example, the numbers that represent the short type vary between 0 and 32760. If we apply a range that collects numbers between

0 and 1638, approximately 5% of all data is gathered. The queries differ according to the type. The SQL query used in MonetDB for the type Short is:

- `select count(*) from binary_table where i>=0 and i<=1638;`

and in STILTS:

- ./stilts tpipe in=binary_table.fit cmd='select (i>=0); select (i<=1638);' omode=count

For the type long, the SQL query is:

- `select count(*) from binary_table where j>=0 and i<=750000;`

and in STILTS:

- ./stilts tpipe in=binary_table.fit cmd='select (j>=0); select (i<=750000);' omode=count

For the Long type, the SQL expression in MonetDB is:

- `select count(*) from binary_table where k>=0 and k<=107374182;`

and in STILTS:

- ./stilts tpipe in=binary_table.fit cmd='select (k>=0); select (k<=107374182);' omode=count

For the Float type, the SQL expression in MonetDB is:

- `select count(*) from binary_table where e>=0 and e<=0.25;`

and in STILTS:

- ./stilts tpipe in=binary_table.fit cmd='select (e>=0); select (e<=0.25);' omode=count

The last type to be tested is the Double. The SQL expression in MonetDB is:

- `select count(*) from binary_table where d>=0 and d<=0.25;`

and in STILTS:

- ./stilts tpipe in=binary_table.fit cmd='select (d>=0); select (d<=0.25);' omode=count

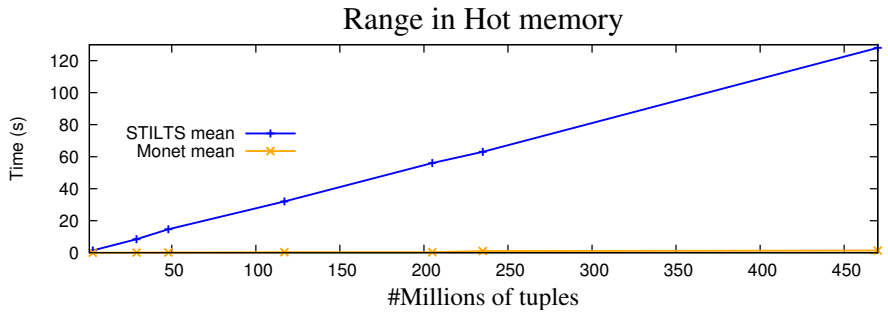

Figure 5.11: Performance of MonetDB and STILTS in Range operations with numerical types

Figure 5.11 represents the Range test for the numerical types, in hot memory, for MonetDB and STILTS. Once again, we grouped all types together for the same reason as we did in **Figure 5.1**. Both tools have a linear behavior during the tests and STILTS takes an average of 128 seconds to perform the last test. As for MonetDB, it reaches one second in the last test.

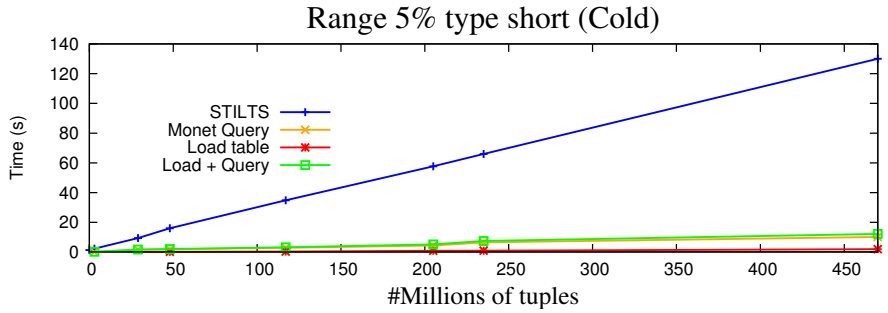

Figure 5.12: Performance of MonetDB and STILTS in Range operations for short type

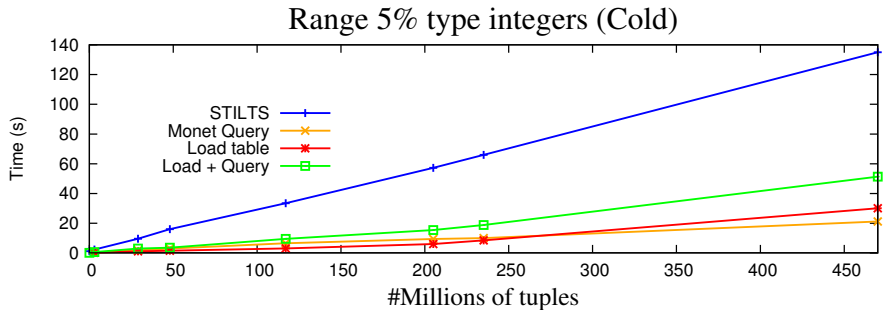

Figure 5.13: Performance of MonetDB and STILTS in Range operations for integer type

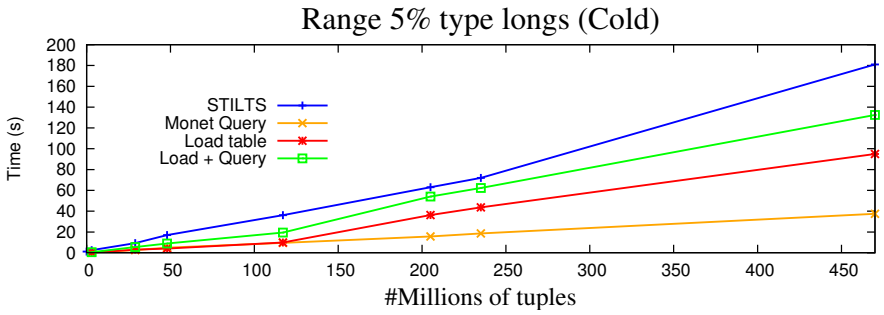

Figure 5.14: Performance of MonetDB and STILTS in Range operations for long type

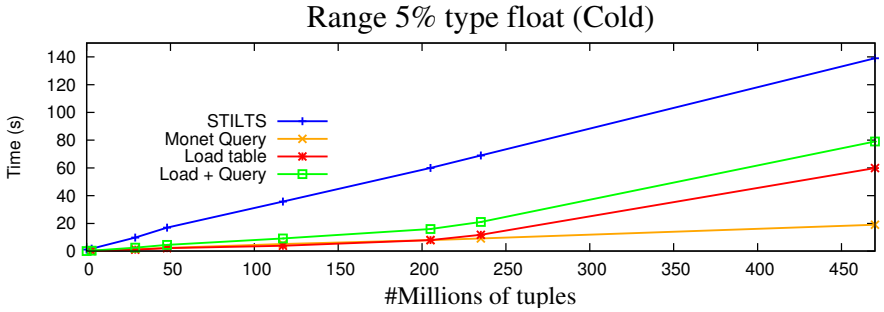

Figure 5.15: Performance of MonetDB and STILTS in Range operations for float type

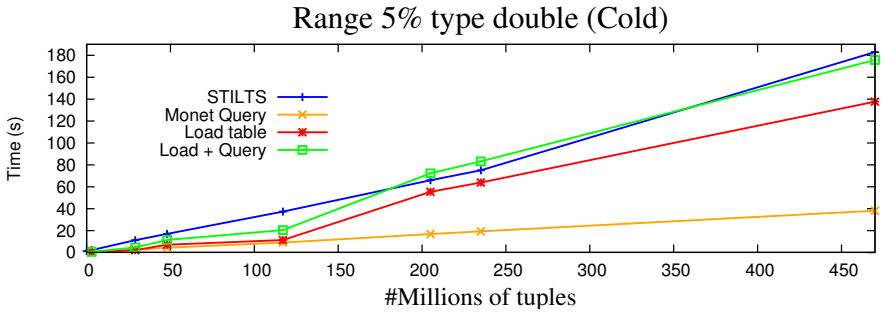

Figure 5.16: Performance of MonetDB and STILTS in Range operations for double type

As for the cold memory tests, we assign a graph to each one of the types. They are represented in **Figure 5.12**, **Figure 5.13**, **Figure 5.14**, **Figure 5.15** and **Figure 5.16**. For all of them the performance of MonetDB is always better, even when the loading time of the table is added to the total time.

5.3.3 Statistics Delegation for Group number 1

For the statistical tests, the generic SQL Query in MonetDB is:

- `select min(J),max(J),avg(J) from binary_table;`

As for STILTS:

- ./stilts tpipe binary_table.fit omode=stats;

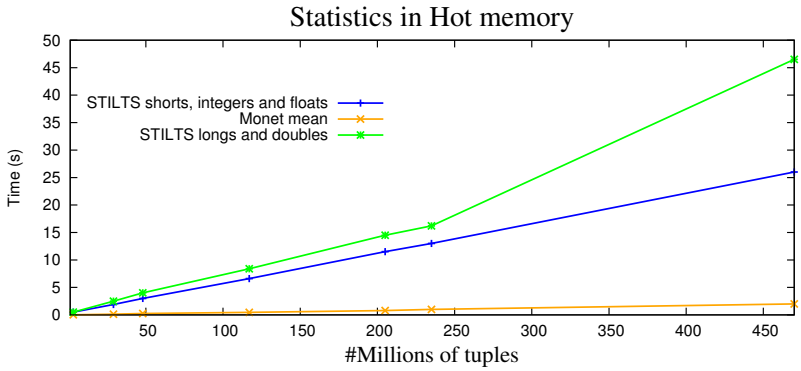

Figure 5.17: Performance of MonetDB and STILTS in statistical operations for short type

Figure 5.17 presents the statistical tests in hot memory, for MonetDB and STILTS. We observe that STILTS has two different behaviors. The first one is when dealing with shorts, integers and floats. It is the fastest one, reaching 26 seconds in the file with 470 millions of tuples. The second one is when dealing with longs and doubles. It is the slowest one, taking an average of 46.5 seconds in the last test, with 470 millions of tuples. As for MonetDB, it has only one behavior, applied for all the numerical types. It is linear and it needs 2 seconds to answer the last query.

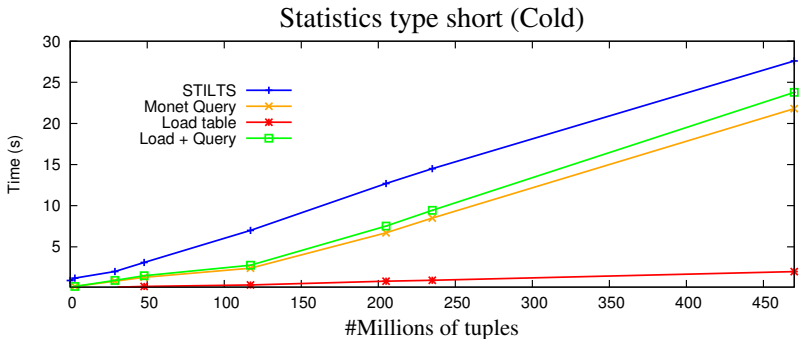

Figure 5.18: Performance of MonetDB and STILTS in statistical operations for short type

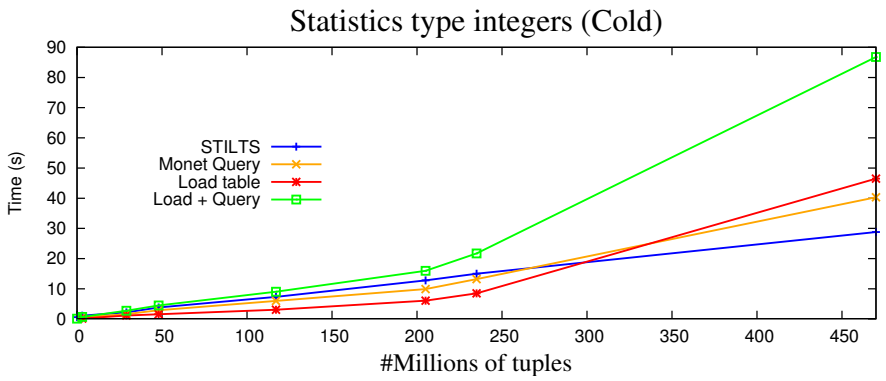

Figure 5.19: Performance of MonetDB and STILTS in statistical operations for integer type

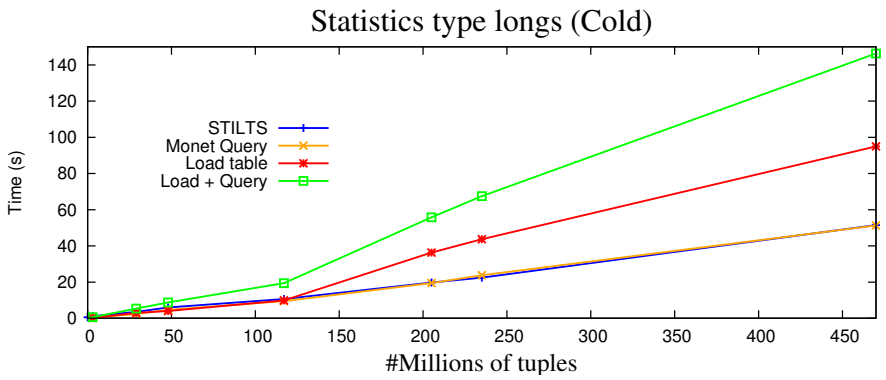

Figure 5.20: Performing of MonetDB and STILTS in statistical operations for long type

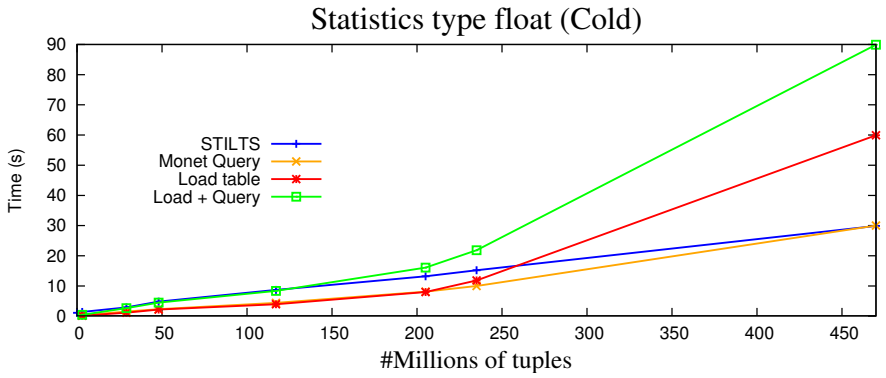

Figure 5.21: Performance of MonetDB and STILTS in statistical operations for float type

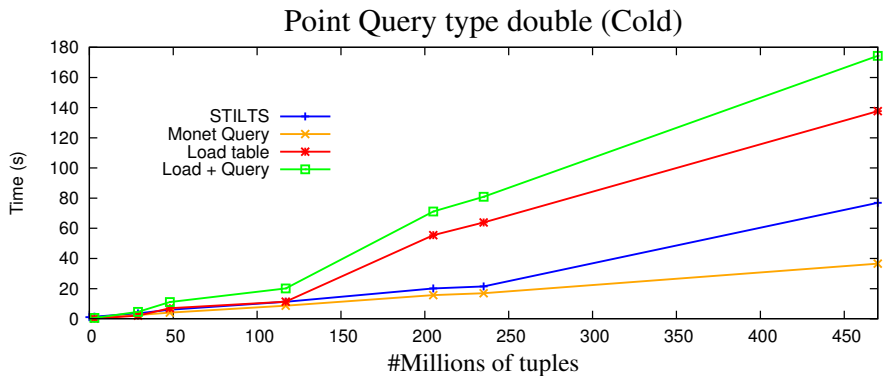

Figure 5.22: Performance of MonetDB and STILTS in statistical operations for double type

As for the cold memory tests, represented in **Figure 5.18**, **Figure 5.19**, **Figure 5.20**, **Figure 5.21** and **Figure 5.22**, the results of MonetDB get worse, due to the computation required in statistical operations and due to the loading time that needs to be added to the total time to execute the query.

5.3.4 Selection and Filter Delegation for Group number 2

Finished the tests for the first group of FITS files, we will now perform the tests for the second group of FITS files. They are represented in **Table 5.1** and the first test is Point Query with hot memory. In MonetDB, it can be represented as:

- `select count(*) from binary_table where I=4;`

and in STILTS:

- ./stilts tpipe binary_table.fit cmd='select (I==4)' omode=count

This test filters data, taking a table as input, and returning the number of rows that satisty the boolean expression. **Figure 5.23** shows that MonetDB is faster than STILTS in all tests. STILTS behavior is linear, growing in average 1.7 times in each iteration, until it reaches the 7GB file, where the time starts to grow exponentially in the files with 8GB and 16GB. This happens because the system is running out of memory. As mentioned above, STILTS perform a row at a time scan.

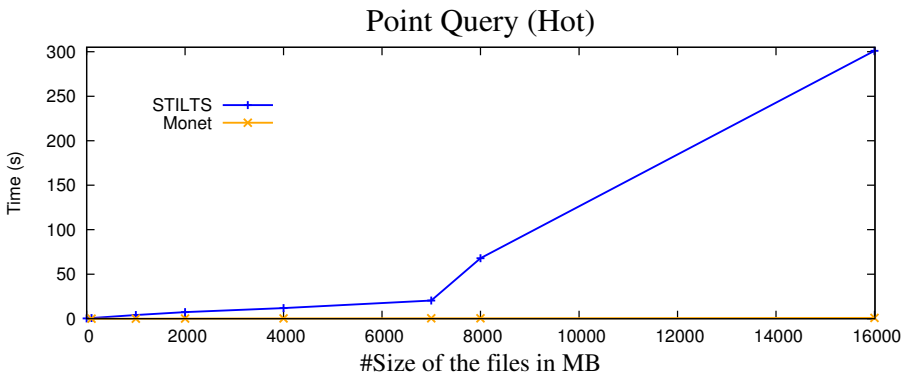

Figure 5.23: Performance of Monet and STILTS in Point Query operations

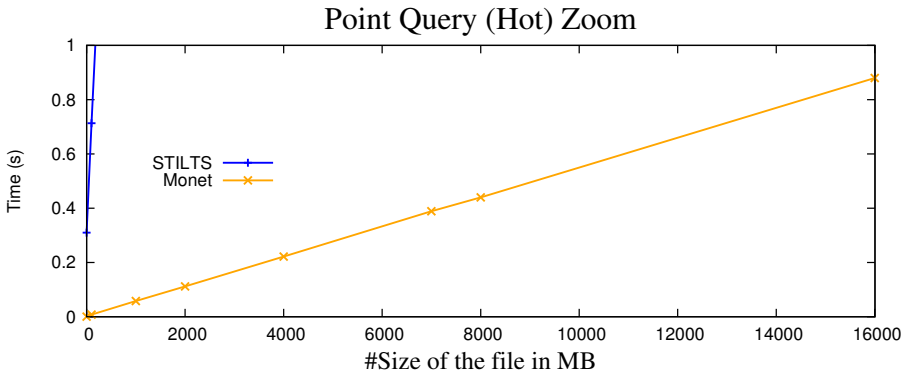

Figure 5.24: Performance of Monet and STILTS in Point Query operations

Figure 5.24 shows a zoom into MonetDB performance to better understand its behaviour. It is also linear, growing in average 2 times in each iteration. It does not have the same problem as STILTS, growing linearly instead. After running the *mserver* with the *algorithms* option, we noticed that MonetDB builds a hash table in memory, taking less than one second in all the Point Query operations.

For the cold memory tests, represented in **Figure 5.25**, the time for MonetDB and STILTS to execute the query is worse, when compared to the hot memory tests. For STILTS, this happens until it reaches the 8GB file, after that, the time is the same as in hot memory tests, because the system runs out of memory and it has to perform swap operations. The time for MonetDB to execute the query is more or less the same as STILTS for all the tests, except for the 16GB file, where the time of MonetDB is better. The problem is, mentioned above, when the loading time is added to the total time of MonetDB. STILTS gets, in fact, a much better performance.

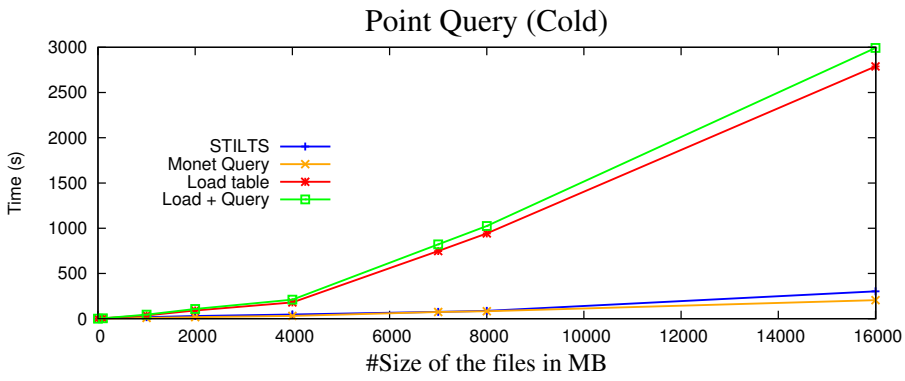

Figure 5.25: Performance of Monet and STILTS in Point Query operations

5.3.5 Range delegation for Group number 2

In the sequence of tests that follows we will explore different kinds of ranges of data selected from the tables, more precisely **1%**, **5%**, **50%**, **99%** and **100%**. When the files are created, a random number is generated between 0 and 32767, for the column named **I**. Knowing that, it is easy to choose what range of data it will be selected.

For the **first** test, 1% of the data is collected. The SQL expression for MonetDB is:

- `select count(*) from table where I>=0 and I<=327;`

And the expression for STILTS is:

- ./stilts tpipe table.fit cmd='select (I>=0); select 'I<=327') omode=count

For the **second** test, 5% of the data is collected. The SQL expression used in MonetDB is:

- `select count(*) from table where I>=0 and I<=1638;`

And the expression in STILTS is:

- ./stilts tpipe table.fit cmd='select (I>=0); select 'I<=1638') omode=count

The **third** test collects 50% of the data. The following SQL expression is used for MonetDB:

- `select count(*) from table where I>=0 and I<=16380;`

And the expression for STILTS is:

- ./stilts tpipe table.fit cmd='select (I>=0); select 'I<=16380') omode=count

The **forth** test collects 99% of the data. The SQL expression used in MonetDB is:

- `select count(*) from table where I>=0 and I<=32439;`

And the expression in STILTS is:

- ./stilts tpipe table.fit cmd='select (I>=0); select 'I<=32439') omode=count

The **fifth** test collects 100% of the data. The SQL expression used in MonetDB is:

- `select count(*) from table where I>=0 and I<=32767;`

And the expression in STILTS is:

- ./stilts tpipe table.fit cmd='select (I>=0); select 'I<=32767') omode=count

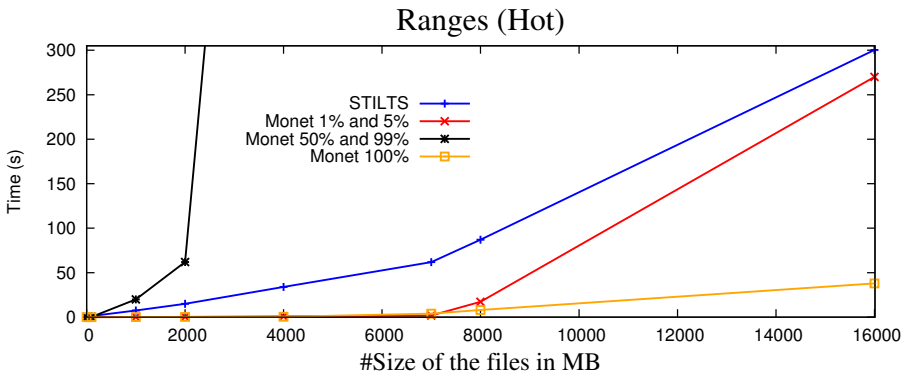

Figure 5.26: Performance of Monet and STILTS in Range operations

Figure 5.26 represents all the range tests in hot memory for MonetDB and STILTS. The behaviour of STILTS is similar to the Point Query test in hot memory, for all the range tests. This happens because, as it was said before, STILTS checks one row at a time for all the filtering operations.

As for MonetDB, there are some variations on the behaviour of the system during the different range tests.

For the ranges of **1%** and **5%**, the results differ from the Point Query tests in hot memory. This happens because a range operation requires more computation than a Simple Query operation and instead of using the hash table structure, MonetDB uses a sequential scan (leading to a worse result), although still better than STILTS. For the first time MonetDB exceeded one second in a hot memory operation.

While performing a trace in the ranges of 1% and 5% of the data, we verified that MonetDB takes most of the time in **uselect** and **leftjoin** operations.

As for the selections of **50%** and **99%** of the data, the values on the graphic diverge from the observations in the previous tests. The time that MonetDB took in the file with 1G was 20.0 seconds, in the file with 2G 62 seconds and in the file with 4G 1267 seconds. Those are really high values, once MonetDB only needed a few milliseconds to perform the previous tests. We encountered a problem with MonetDB performance. An analysis between execution traces of **Range 5%** selection and **Range 50%** selection showed that both have the same plan. However, **Range 50%** selection takes much more time executing **uselect** and **leftjoin** operations. This happens due to a combination of several factors:

- the count operator is executed on the first column in the table of type string
- mitosis optimizer splits it into chunks
- operations over chunks lead to copying the string heaps instead of using the original copying, that makes it slow

For the last test, that collects **100%** of the data, an interesting fact happened. In the previous tests with hot memory, the time of MonetDB and STILTS, for the Range operations, was more or less the same in the 16G file. That is not what happen in the test with a **Range** of 100% of the data. The performance of MonetDB is incredibly faster than the other tests. The reason for that is that MonetDB returns the number of tuples (**count(*)**) without touching them.

For the next row of tests, the same sequences of experiments will be performed, but this time with cold memory.

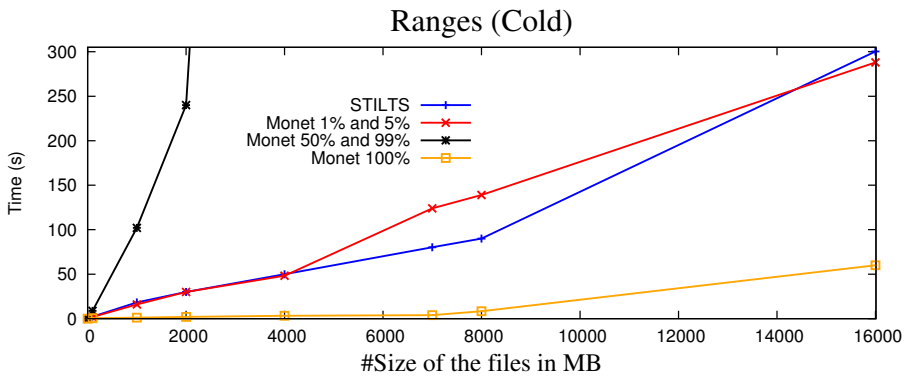

Figure 5.27: Performance of Monet and STILTS in Range operations

Figure 5.27 represents all the range tests in cold memory for MonetDB and STILTS. Comparing to the Point Query test in cold memory, the performance of STILTS is quite the same and it behaves the same for all the range tests, due to the row at a time check for all the filtering operations.

As for MonetDB, it will be inspected the performance of the system for each one of the range tests.

For the ranges of **1%** and **5%**, the performance of MonetDB is worse comparing to the Point Query test in cold memory. First, the time that MonetDB takes to execute the query in the 7G and 8G files is actually bigger than STILTS. Second, although MonetDB gets again faster than STILTS for the 16G file, that difference is not so accentuated like in the Point Query test in cold memory. And the loading time of the tables, that is the same for all the tests in cold memory, also need to be added to the total time of MonetDB to return the result. Summarizing, the performance of STILTS is better than MonetDB for the **1% and 5% Ranges** test in cold memory.

For the range of **50%** and **99%**, the same problem, that was reported for the tests with hot memory, remains for the tests in cold memory, with a even more accentuated curve due to the load of the data.

As for the test that collects **100%** of data, the same fact that occurred in the test with hot memory it happens in this test. In the previous tests with cold memory, it was obvious that the time that **MonetDB** took to perform the tests was more or less the same. That is not what happen in the test with a **Range** of 100% of the data. The reason for the fact was already explained in the test with hot memory.

5.3.6 MonetDB problem

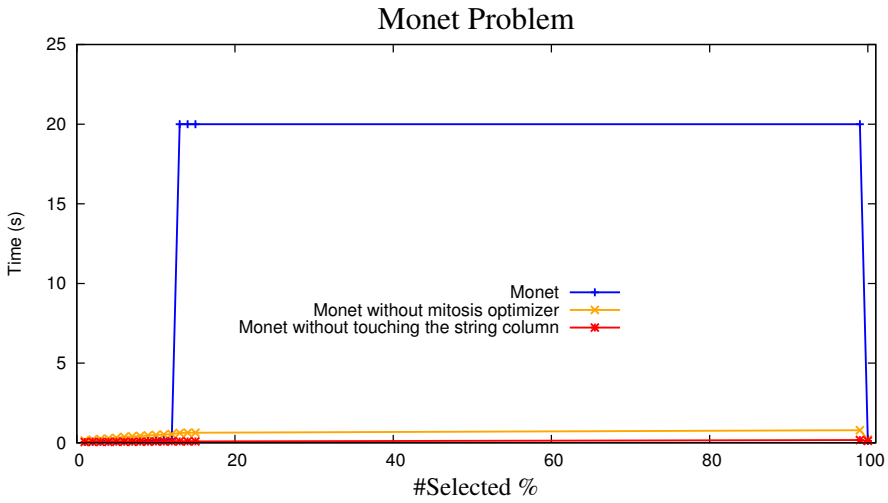

Figure 5.28: Problem on Monet with the table of 1G

Figure 5.28 shows the problem of range selection performance for the table of size 1G. Different ranges of data that were collected for the table with 1G, and the respective times of each operations. When the range of data changes from 12% to 13%, the times grows to 20-22 seconds.

This problem can be solved with two alternatives. The first is to produce an alternative query, that could be:

- `select count(I) from feb1g_2 where I>=0 and I<=16380;`

When this query is executed, instead of picking the column with the string type, a column with the short type is selected. And the result is in fact faster, taking only a few milliseconds, as we can see in **Figure 5.28**, in the line **Monet without touching the string column**.

Another way to solve this bug is by disabling the **mitosis** optimizer. Doing that, the performance of the system changes, taking also a few milliseconds to execute the query, as we can see in **Figure 5.28**, in the line **Monet without mitosis optimizer**.

We can notice that the first alternative, that selects without touching the string column, is more efficient than the second alternative, that disables the mitosis optimizer. This happens because in the first case MonetDB takes advantage of all the optimizers that are available in order to execute the query, and in the second case one of them is disable, leading to a worse performance. Knowing that, and being the **Point Query** and

Range tests the ones that have the problem with the string column, we will repeat those tests with a **count(I)** instead of a **count(*)**, making sure that the string column is not touched.

For the Point Query test in hot memory the performance of MonetDB is the same with count(I) and count(*). As explained before, MonetDB builds a hash table in memory, taking less than one second in all the **Point Query** operations.

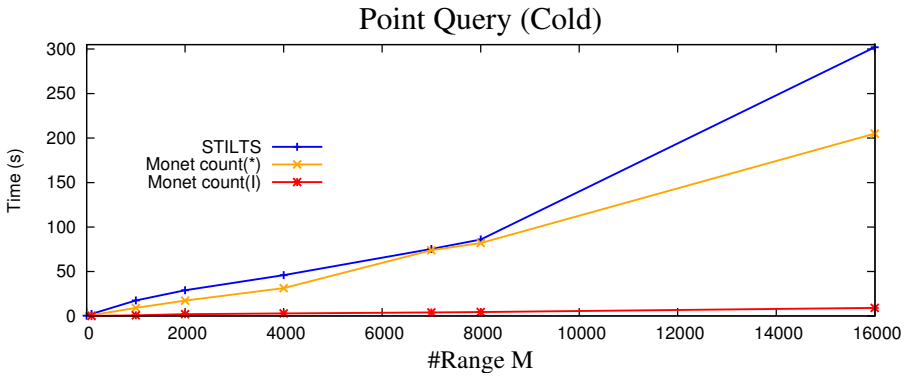

Figure 5.29: Performance of MonetDB and STILTS in Point Query Operations

For the Point Query test with cold memory, performing a **count(I)** is much more efficient than a **count(*)**, as we can see in **Figure 5.29**. Analysing the trace of each one of the alternatives, we realized that the two operations that take more computation time in both queries are an **uselect** and a **leftjoin**. The time of **uselect** is the same for both **count(I)** and **count(*)**. As for the **leftjoin** operation, it takes significantly more time in the **count(*)** query, because it is choosing the string column to perform the join, instead of the integer column picked in the **count(I)** query, that is faster and easier to operate.

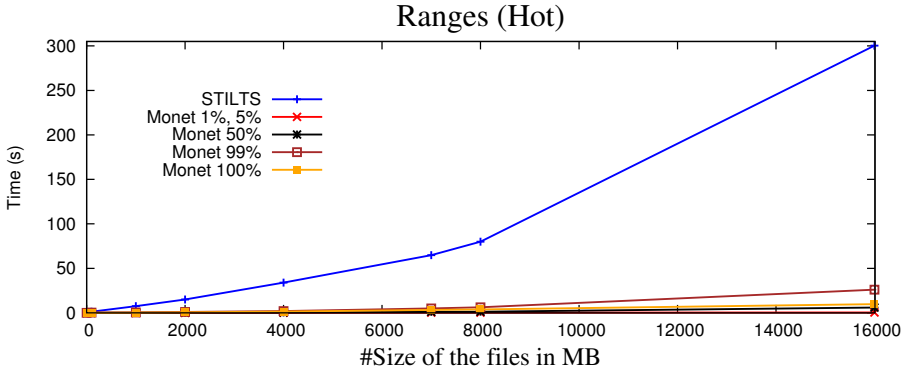

Figure 5.30: Performance of MonetDB and STILTS in Range Operations

Figure 5.30 shows the performance of MonetDB and STILTS for the range operations with hot memory. However, this time the query is **count(I)** instead of **count(*)**. Comparing to the performance of MonetDB in **Figure 5.26**, we can easily conclude that the **count(I)** is much more efficient, once again due to the string column. For the tests that collect **1%** and **5%** of the data, MonetDB only took 0.5 seconds for the 16g file. For the count(*) test, the same test took 280 seconds. The same scenario is reflected in the following range tests. For the range of **50%**, MonetDB takes 6 seconds for the 16g file, and this values are getting bigger and bigger as the range that is collected also increases, till it reaches the test that collects **99%** of the data, where it takes 26 seconds for the 16g file. For the **100%** range test, MonetDB realizes that all the data is being collected and it takes less time, as it was explained before.

Observing the traces of both queries, **count(I)** and **count(*)**, we realized that the most expensive one are the **uselects** and **leftjoins**. The computation time of **uselect** is the same in both queries. This happens because they take only the integer column and perform the select operation on it. As for the computation times of **leftjoin** operations, they are the responsible for the worse time in the **count(*)**, because, as it was explained before, the string column is taken in order to perform the join and it takes more time than the integer column that is picked in the **count(I)** query.

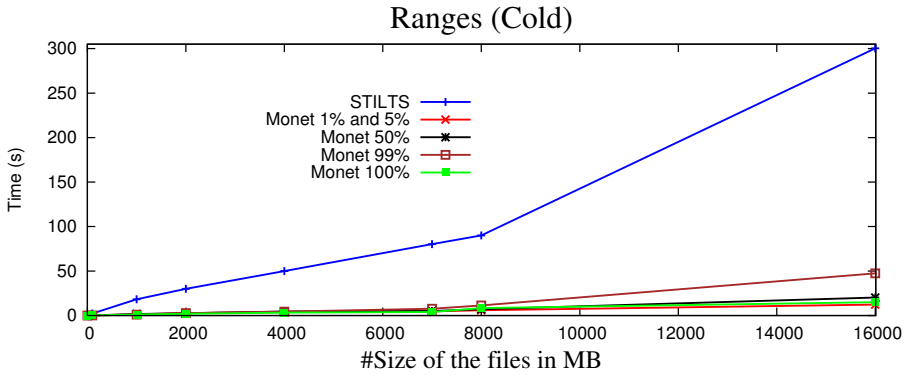

Figure 5.31: Performance of MonetDB and STILTS in Range Operations

Figure 5.31 shows the performance of MonetDB in all the range tests in cold memory. As in **Figure 5.30**, it was used the **count(I)** instead of **count(*)**. We can notice that the difference between the tests with **count(*)** represented in **Figure 5.27** is substantial. The **count(I)** takes considerable less time to compute the final result. As an example of that difference: in the **count(*)** tests, perform a range of **1%** and **5%** in the 16g file take 288 seconds and the same queries, but with **count(I)** take 12.3 seconds. The same happens for the other range tests, taking more time as the amount of data that is collected. The

time is growing till the **99%** range test, where it needs 47.4 seconds, for the 16g file. After that, for the **100%** range test, MonetDB realizes that is collecting all the data of the table, and takes less time to compute the final result.

Like in **Figure 5.30**, it was done a trace between the **count(*)** and **count(I)** queries, and it happens that, once again, the **lefjoin** operations are the ones that take more time, due to the string column, as explained before.

5.3.7 Projection delegation for Group number 2

For the projection operations, with the group number two, the following SQL expression is used in MonetDB:

- `select count(A) from binary_table;`

And for STILTS:

- ./stilts tpipe feb1g.fit cmd='keepcols "A"' omode=count

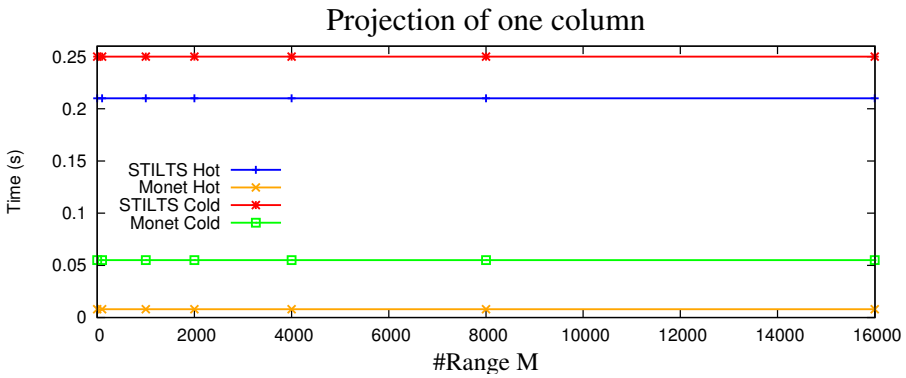

Figure 5.32: Performance of Monet and STILTS in Projection operations

Figure 5.32 shows the performance of both tools with hot and cold memory, in projection operations. Projection is a very efficient operation for column-store MonetDB and STILTS.

5.3.8 Statistical Delegation for Group number 2

For the **Statistics** tests, the following SQL expressions are used in MonetDB:

- SELECT

```
    min(I), max(I), avg(I),
    min(J), max(J), avg(J),
    min(K), max(K), avg(K),
    min(E), max(E), avg(E),
    min(D), max(D), avg(D)
FROM binary_table;
```

For STILTS:

- ./stilts tpipe binary_table.fit omode=stats;

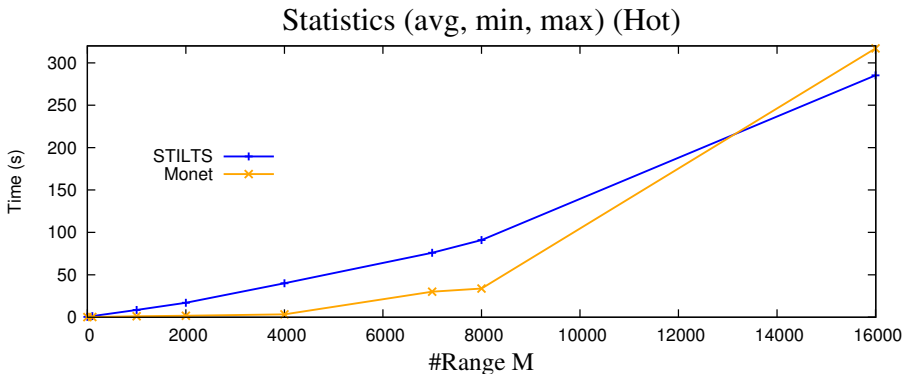

Figure 5.33: Performance of Monet and STILTS in Statistics operations

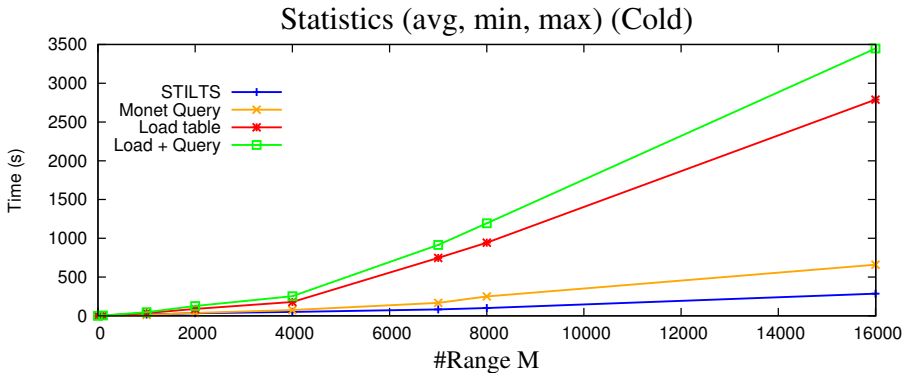

Figure 5.34: Performance of Monet and STILTS in Statistics operations

Figure 5.33 shows the first time that MonetDB was actually worse than STILTS in the hot memory tests, however, that fact happens only for the 16GB file. The behavior of STILTS is the same as the previous tests because, as said before, it executes a row at a time. As for MonetDB, a trace was done to check which algorithms were being used. It happens that MonetDB uses a **scan select** and performs a lot of intermediate calculations (for example: **aggr.sum**, **aggr.count**, **aggr.max** and **aggr.min**), to give as result the min, the max and the mean of each column. Examining the trace of MonetDB, a total number of 591 operations are performed for the 16GB file and 449 for the 8GB file. In the previous tests, Point Query and Range operations, the trace took only a few lines.

At this point, something can be done to improve the performance of MonetDB in statistic operations:

- add statistical information to the metadata of each column
- scan and save potential statistic results of the column (**min**, **max**, **count**, **aggregate** and **average**) instead of looking for the entire column every time that an statistical operation request is done

As expected, STILTS is better than MonetDB in the statistics tests with cold memory, as it can be seen in the **Figure 5.34**. It is a reflection about what was explained before for the hot memory tests.

5.3.9 Summary of the tests for the first and second groups

In all the tests performed so far, can be evidenced that MonetDB is better than STILTS in the hot memory tests (with the exception of the **Range 50%** and **Range 99%** tests due to the bug of MonetDB already explained and also for the statistics tests, for the 16GB file). The scenario changes when the tests with cold memory are executed. Because MonetDB not only needs the time to execute the query (that is more or less the same as STILTS), but also the time to load the table into the database system. When this second time is added to the total time of MonetDB, the performace is, in fact, worse for MonetDB in these tests.

After discovering the problem with MonetDB we decided to perform the Point Query and Range tests one more time. This time without touching the string column, doing a **count(I)** instead of a **count(*)**. It was obvious that the performance of MonetDB increased significantly with the count(I) query, being always better than STILTS, for all the hot and cold memory tests.

We can also notice that STILTS always performs a sequential scan over all the data for this filtering operations, leading to a linear behavior in all the tests, because all the

data has to be touched and all the tests have to perform the queries through the same sequence of files.

As for MonetDB, either a hash table structure can be built, or a sequential scan is performed all over the columns of the table. Participating in the query, for the first case, we have the Point Query tests and for the second case, there are the Range tests and the Statistics tests (which also need some extra computation time besides the sequential scan).

The following tests concern to the third group os FITS files. We will study the **equi-** and **theta joins**.

5.3.10 Equi-join delegation for Group number 3

We performed two sets of equi-join experiments to study the effect of file sizes and join fan-out factors on the performance.

First, we created a basic set of files, setting a column **K** to be the primary key. The values of **K** start from 1 and end in the respective number of rows. It could have been a generation of a random number, but in that case we would lose the control of the fan-out factor that is produced as a final result. Those files are used as a left-hand join operand.

In order to control the join operations and the expected number of rows affected, a **fan-out factor** of 1, 3, 5 and 10 was attributed to each one of the files.

In the first set of experiments we vary both the fan-out factor and the operand and result size. For each file of size S and fan-out factor f we created a right-hand join operand file of size $f * S$ such that each row in the left-hand operand file matches exactly f tuples in the right-hand one. For fan-out factor of 1 we perform a self-join on the unique column. Note, that along with the fan-out factor this way of file construction increases the size of both the right-hand operand and the result set.

The test involves two tables, and in MonetDB can be represented as:

- ```
  select count(*)
  from mar1mjoin1, mar1mjoin2
  where mar1mjoin1.K=mar1mjoin2.K;
  ```

Where **mar1mjoin1** is the left table with 1MB and **mar1mjoin2** is the right table with 3MB. The resulting table is of size 3MB.

And in STILTS:
  ```

- ./stilts tmatch2 matcher=exact in1=table1.fit in2=table2.fit values1=K values2=K progress=none find=all omode=count

Figure 5.35 and **Figure 5.36** present the performance and the memory consumption of the equi-join tests, with different fan-out factors, in each one of the tools.

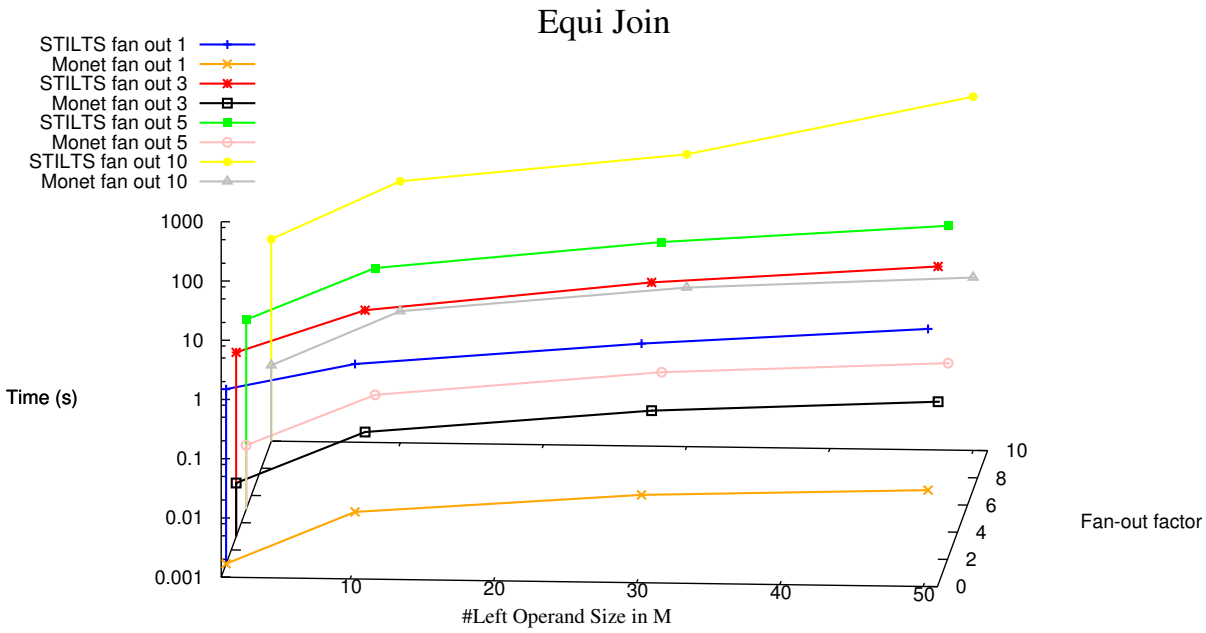

Figure 5.35: Performance for the different fan-out factors

It can be observed, in **Figure 5.35**, that for the **fan-out factor of 1** MonetDB is 2 orders of magnitude faster than STILTS. Even the biggest value of MonetDB, for the 50MB file, is smaller than the smallest value of STILTS, for the 1MB file. As for the memory consumption, in **Figure 5.36**, we can clearly notice that MonetDB uses less memory than STILTS. In the worst case, STILTS uses 16.0% of the memory and MonetDB uses 0.4% (the fewer that STILTS can get). This happens because the data has to be accessed and a join algorithm has to be applied, by both tools. The join algorithm performed in MonetDB was the **hash join**, that only builds an hash table, consuming much less memory than the one used by STILTS.

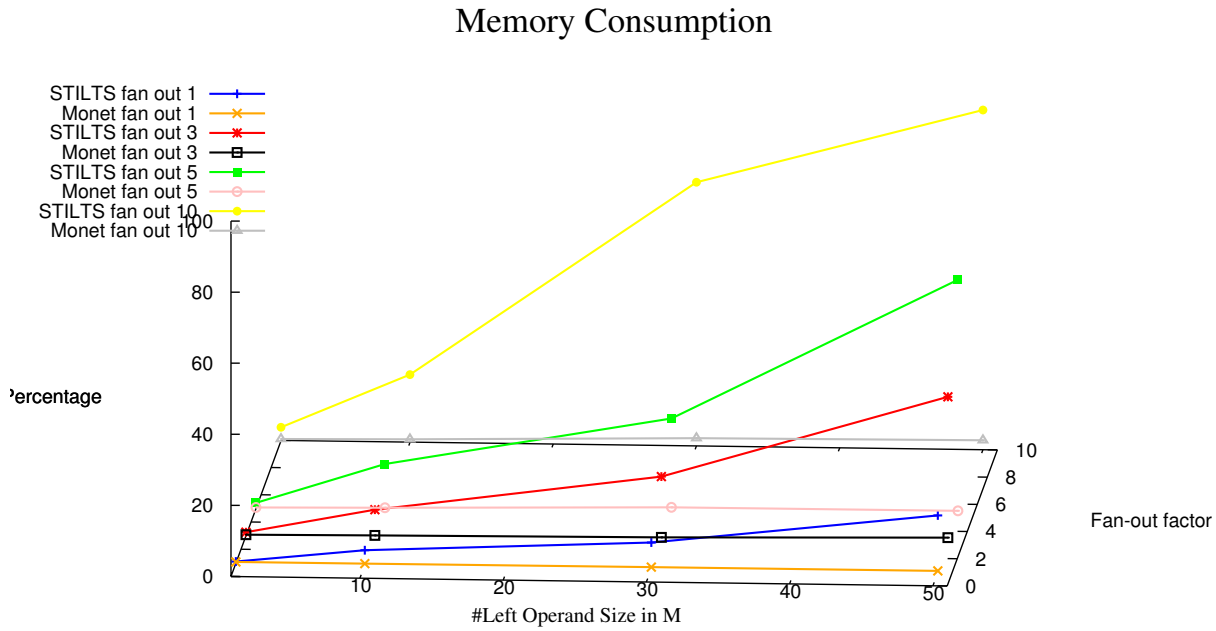

Figure 5.36: Percentage of memory consumed

For STILTS, as said before, the complexity of the join operations is **O(N log(N))**, where **N** is the total number of rows in all the tables being matched. The algorithms that have this complexity are called **loglinear** and their purpose is the sorting of an array. Following, we have some examples of algorithms with this complexity:

- **heapsort**: begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs the heap, removes the largest remaining item, and places it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

- **quicksort**: sorts by employing a divide and conquer strategy to divide a list into two sub-lists. The steps are:

  - pick an element, called a pivot, from the list.

  - reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal

values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

- recursively sort the sub-list of lesser elements and the sub-list of greater elements.

- **mergesort**: It is also a divide and conquer algorithm and the steps are:

  - divide the unsorted list into two sublists of about half the size

  - sort each sublist recursively by re-applying the merge sort

  - merge the two sublists back into one sorted list

For the **fan-out factor of 3**, both tools take more time and consume more memory than in the tests with **fan-out factor of 1**. This should be expected because the size of the right table gets 3 times bigger, as the final result. The join algorithm applied by MonetDB is, once again, **hash join**. And, once again, we can notice that, in **Figure 5.36**, the biggest value of MonetDB is smaller than the smallest value of STILTS.

For the **fan-out factor of 5**, the memory consumption gets even more bigger, reaching 67% in the worst case for STILTS. In the last tests, the memory consumption for STILTS in the worst case was 16% for the fan-out factor of 1 and 41.7% for the fan-out factor of 3.

The last test is with the **fan-out factor of 10**. It can be noticed that the performace of **STILTS** starts getting worse when it has to deal with bigger files. The last test of the fan-out 1:10 involves a file with 50MB on the left and a file with 500MB on the right, creating a new table with 500MB. It can be noted that for **STILTS**, the last test does not describe the same line, as for the previous fan-outs of 1,3 and 5. On the other hand, the behavior of **MonetDB** is more or less the same in all the tests involving different fan-outs, and the performance is always better than **STILTS**.

The fan-out 10 is the one that demands more computation by the tools. As it can be seen in **Figure 5.36**, STILTS starts to work really bad with the 30MB and 50MB files, that are joined with 300MB and 500MB files, producing tables of 300MB and 500MB respectively. In the last test, STILTS consumes 95.6% of the memory. It is the maximum that it can take because the memory is been used by other processes.

In the second set of experiments we fix the size of the right hand join operand and the result set to be equal to the size of the left-hand operand, and vary only the fan-out factor.

For each file of size S and fan-out factor f we created a right-hand join operand file of size S such that every $\frac{1}{f}$ row in the left-hand operand file matches exactly f tuples in the right-hand one.

The results for 1:1 are the same as above, because we are dealing with the same files, so they are skipped.

We can notice that there is no substantial differences between the fan-out factors represented in **Figure 5.37**. Both systems have the same behavior in all the fan-outs. This happens because the table on the right side and the result table have the same size as the table on the left.

Hence, the processing time is affected more substantially by the sizes of the operands, and not so much by the fan-out variation of files with the same size.

As expected, also the memory consumption is similar in the fan-out experiences, represented in **Figure 5.38**, with STILTS being always the tool that consumes more memory.

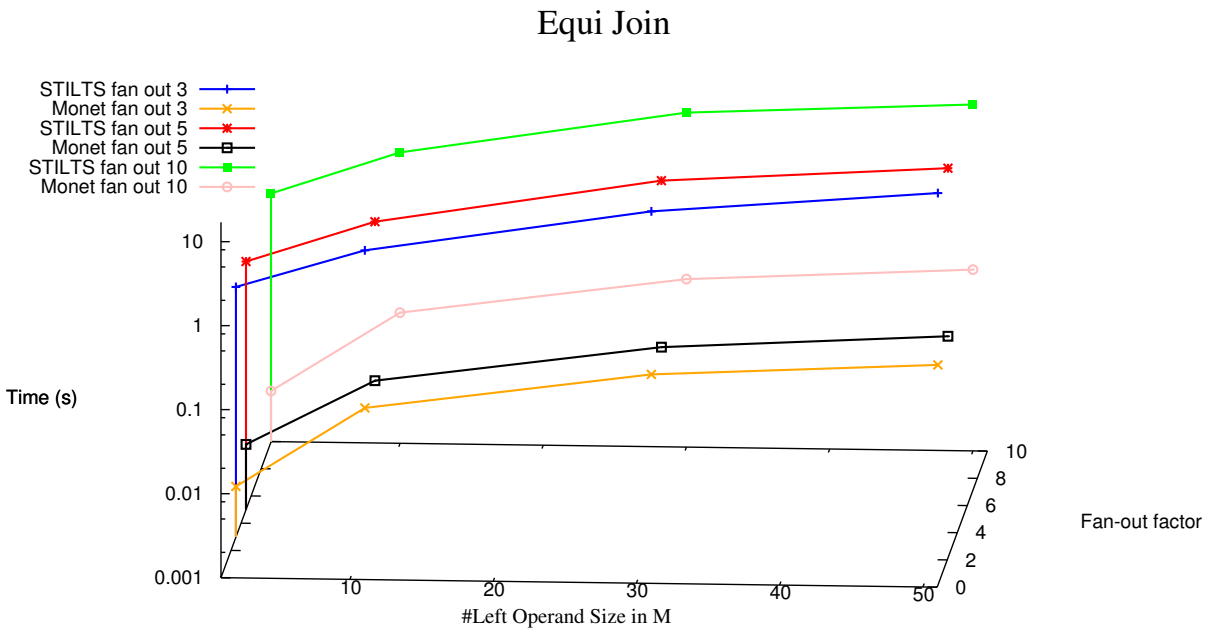

Figure 5.37: Performance for the different fan-out factors

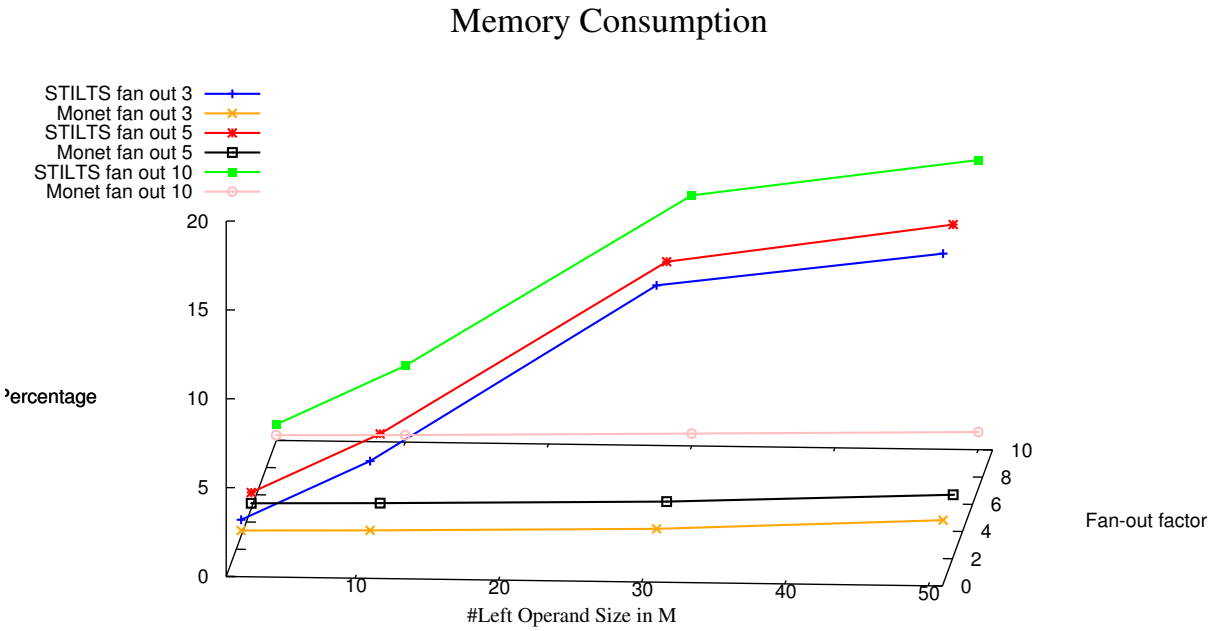

Figure 5.38: Percentage of memory consumed

5.3.11 Band-join delegation for Group number 3

We reuse the set of basic files from **Section 5.3.10**. We perform a self-join on the unique column with varying error bound which also determines the fan-out factor. An error of 1 gives fan-out of 3, an error of 2 gives an fan-out of 5 and an error of 4 gives an fan-out of 9. In this band-join tests, the join is performed between files with the same size, producing a result 3 times bigger in the case of the fan-out factor of 3, 5 times bigger in the case of the fan-out factor of 5 and 10 times bigger in the case of the fan-out factor of 10.

We will start with the fan-out factors that increase the file size. The **first** test is the fan-out factor of 3, with an error of 1. The SQL expression in MonetDB is:

- `select count(*) from tst as t1, tst as t2`

  `where t1.id between t2.id-1 and t2.id+1;`

And in STILTS, the expression is:

- ./stilts tmatch2 matcher=1d in1=mar1mjoin1.fit in2=mar1mjoin1.fit values1=K values2=K params=1 progress=none find=all omode=count

The **second** test is the fan-out factor of 5, with an error of 2. The SQL expression in MonetDB is:

- `select count(*) from tst as t1, tst as t2`

  `where t1.id between t2.id-2 and t2.id+2;`

And in STILTS, the expression is:

- ./stilts tmatch2 matcher=1d in1=mar1mjoin1.fit in2=mar1mjoin1.fit values1=K values2=K params=2 progress=none find=all omode=count

The **third** and last test is the fan-out factor of 9, with an error of 4. The SQL expression in MonetDB is:

- `select count(*) from tst as t1, tst as t2`

  `where t1.id between t2.id-4 and t2.id+4;`

And in STILTS, the expression is:

- ./stilts tmatch2 matcher=1d in1=mar1mjoin1.fit in2=mar1mjoin1.fit values1=K values2=K params=4 progress=none find=all omode=count

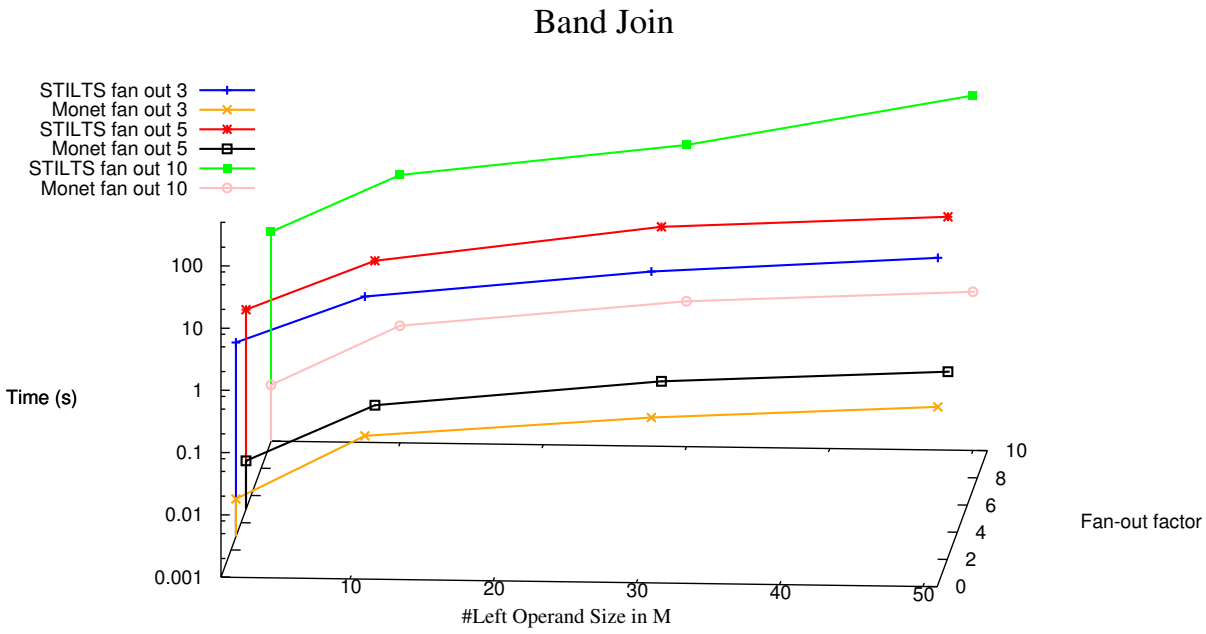

Figure 5.39: Performance for the different fan-out factors

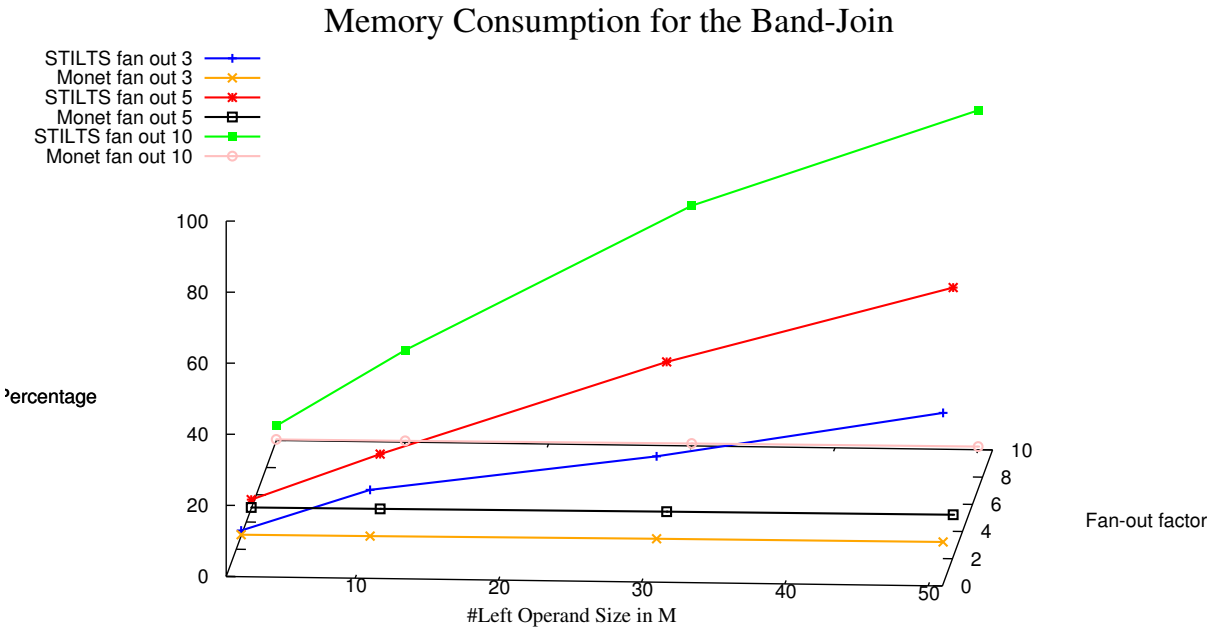

Figure 5.40: Percentage of memory consumed

Figure 5.39 and **Figure 5.40** represent the performance and the memory consumption of the band-join tests, with different fan-out factors, in each one of the tools.

For the **fan-out** factor of **3** and **5**, **Figure 5.39** shows that the band-join is simpler than the Equi-Join. Actually, all the results in all the experiments (both MonetDB and STILTS) for the fan-out factor of 3 take less time in the **Figure 5.39**, comparing to the tests in the **Figure 5.35**. Also for the memory consumption, the band-join tests consume less memory than the equi-join tests. The main reason for occurrence is the size of the files. As it explained before, the band-joins perform always a self-join, varying the fan-out with the error factor, producing a bigger file. The equi-joins work with a smaller file in the left-hand, a bigger file in the right-hand, and produce a file with the same size as the right-hand file used for the join operation.

Apart from that, MonetDB is always better than STILTS, both in the performance tests and in the percentage of memory consumed.

The same that happened for the fan-out factor of 10 in the Equi-Join tests, also occur in the tests of **Figure 5.39**. The performace of **STILTS** starts getting worse when it has to deal with bigger files and for the last test, the same line that appeared in the **Figure 5.35**, also occurs in this test. The same for the memory consumption. STILTS consumes 95.6% of the memory.

For the second row of experiments, with files with the same size, but with different fan-outs we will not perform the experiments, because we already conclude that the processing time is affected more substantially by the sizes of the operator.

Chapter 6

Related Work

There are several tools that can access and manipulate FITS files. In this chapter we present them and attempt to answer a set of questions in order to analyse their strengths and weaknesses. The questions will be the following: Do they have a functional completeness with respect to the use case? What is the programming model? What is the maturity and the user community? What is the vitality of the product-team? What is the positioning with respect to the database technology? What is the ability to scale to the requirements posed? It is possible the multiple access to the same FITS file? How many FITS files can be simultaneously opened? What is the maximum FITS file size supported? Can it read and output FITS files in UNIX compress format? And , finally, which library do the tools use?

We will also investigate the alternatives of storing astronomical data which compete with the FITS format. By building a user matrix that determines a set of tasks, we were able to present the similarties and differences between the alternatives, demonstrating which formats are better for storing astronomical data.

6.1 CFITSIO

CFITSIO [16] is a library of routines, written in C language, for reading and writing data files in the FITS data format. It was initially developed by the *HEASARC* (High Energy Astrophysics Science Archive Research Center) with the purpose of converting astronomical data sets into FITS format. With all the contributions given by the Integral Science Data Center in Switzerland, and the XMM/ESTEC project in The Netherlands, a second version of the package was created. That version is available on the Nasa's software website [3]. It is machine-independent and it runs on most commonly used computers and workstations. *CFITSIO* will probably run on most Unix platforms. Cray

supercomputers are currently not supported.

The small example bellow demonstrates *CFITSIO* routines:

```
#include <string.h>
#include <stdio.h>
#include "fitsio.h>

int main(int argc, char *argv[])
{
   fitsfile *fptr;
   char card[FLEN_CARD];
   int status = 0, nkeys, i;

   fits_open_file(&fptr, argv[1], READONLY, &status);
   fits_get_hdrspace(fptr, &nkeys, NULL, &status);

   for (i=0; i <= nkeys; i++) {
      fits_read_record(fptr, i, card, &status); /* read keyword */
      printf("%s\n", card);
    }
   printf("END\n\n"); /* terminate listing with END */
   fits_close_file(fptr, &status);

   if(status)         /* print any error messages */
      fits_report_error(stderr, status);
   return(status);
}
```

CFITSIO is the result of many years of development, providing 100% complete support for the FITS Standard. Namely, *CFITSIO* supports simultaneous read and write access to multiple HDUs in the same FITS file. Thus, one can open the same FITS file twice within a single program and move to 2 different HDUs in the file, and then read and write data or keywords to the 2 extensions just as if one were accessing 2 completely separate FITS files.

The maximum number of FITS files that may be simultaneously opened by CFITSIO is set by NMAXFILES as defined in fitsio2.h. It is currently set = 300 by default. On some systems it has been found that gcc supports a maximum of 255 opened files.

The current maximum FITS file size supported by CFITSIO is about 6 terabytes (containing 2^{31} FITS blocks, each 2880 bytes in size). Currently, support for large files in CFITSIO has been tested on the Linux, Solaris, and IBM AIX operating systems.

FITS files can be read and written in shared memory. This can potentially achieve better data I/O performance compared to reading and writing the same FITS files on magnetic disk.

Compressed FITS files in gzip or Unix compress format can be directly read. Output FITS files can be written directly in compressed gzip format, thus saving disk space.

The I/O speed depends of course on the type of computer system that the *CFITSIO* library is running on. The average of the workstations can achieve speeds of 2-10 MB/s when reading or writing FITS tables. The speed can go even faster if the FITS file is in main memory (30MB/s or more)

For small transfers of data, *CFITSIO* maintains a set of internal Input-Output (IO) buffers in main memory, each one containing a FITS block (2880 bytes) of data. When data on the FITS files needs to be accessed, *CFITSIO* first transfers the FITS block containing the bytes into one of the IO buffers in memory and the next time that needs to access the data it can use the fast IO buffer rather than using a slower disk access. The number of available IO buffers is determined by the NIOBUF parameter (in fitsio2.h) and is currently set to 40 by default. Knowing that, there are two aspects that must be considered: First, whenever *CFITSIO* reads or writes data it first checks to see if the block is already loaded into one of the IO buffers. If not, and if there is an empty IO buffer available, then it will load that block into the IO buffer (when reading a FITS file) or will initialize a new block (when writing to a FITS file). Second, when all the IO buffers are being used, the system must decide which one is going to reuse (normally is the one that has been accessed least recently).

For big transfers of data, *CFITSIO* simply avoids all this IO buffers process and straightforwardly reads or writes the bytes of interest directly to disk through a read or write routine. The minimum threshold for the number of bytes to read or write this way is set by the MINDIRECT parameter and is currently set to 3 FITS blocks = 8640 bytes. Using this strategy, transfer rates of 5-10MB/s or greater can be achieved. Note that this fast direct IO process is not applicable when accessing columns of data in a FITS table because the bytes are generally not contiguous since they are interleaved by the other columns of data in the table. This explains why the speed for accessing FITS tables is generally slower than accessing FITS images.

Summarizing, the first strategy is used when dealing with FITS tables and the second strategy is used when dealing with FITS images.

Reading or writting small chunks of data via the IO buffers is in fact less efficient

because it requires an extra copy operation and bookkeeping steps. Conventionally, it is more efficient to read or write as large array as possible. Nevertheless, if the array becomes so large that the operating system cannot store it all in RAM memory, swapping operations need to be done, degrading the performance.

As said and explained before, The IO buffers strategy is used to read and write FITS tables. In order to make it efficient, a single pass through the FITS file is required. An example of a poor program plan is to read a large, 4-column table by sequentially reading the entire first column, then the second, the third and finally the forth column. This design of the program requires 4 passes through the file which could quadruple the execution time of an IO limited program. The tactic used by the *CFITSIO* library consists in reading or writting as many rows of the table as possible into the available IO buffers and then proceed to the next range of rows. The optimal number of rows to read or write at one time in a given table depends on the width of the table row and on the number of IO buffers that have been allocated in *CFITSIO*. There is a routine in *CFITSIO* that will return the optimal number of rows for a given table given a FITS file as an input: *fits_get_rowsize*. Using a very small value however can also lead to poor performance because of the overhead from the larger number of subroutine calls.

6.1.1 Fv

Fv [18] is a tool that is built under the *CFITSIO* library. It is a graphical program, easy to use and focus on viewing and editing FITS images and tables.

After reading the *Fv* documentation and experimenting the tool we realized that it is a powerful program but there are some functionalities missing. It provides a good visualization of the data present in the tables and allows the user to click in a particular element of the table and edit it right away. The same edition can be done to an entire column, selecting the name of the column that we want to modify and enter the expression that will apply the changes to all the elements of the column. An entirely new table column can also be created based on an expression. Insertion procedures and deletion by selecting the rows or columns that we want to delete is also possible. The delete feature can also be done through an expression. Sort in ascending or descending orders are also permitted. The entire table can also be copied to a plain ASCII format file. The ASCII file can then be printed or edited with an ordinary text editor program. As for the statistical and projection operations, it is equally well presented and easy to obtain the information, choosing which column we want to either project or know statistical information about it (number of values, minimum value, maximum value, mean and standard deviation). Finally, the most recent functionality provided by *Fv* is the histogram tool, where the information about how the values in the column are distributed can be checked and also

which bin size we want to apply.

However, this tool is still missing some features which do not allow a functional completeness with respect to our use case. It does point query filtering and range selection operations but only highlights the rows within the table that fulfill the criteria, instead of creating a complete new table with the rows that match the parameter. If the table is small then we can assimilate the rows efficiently. However, if the tables are too large and the rows that match are only a portion of the table, then it becomes unreadable and too complex to analyse the results.

Finally, join cannot be undertaken, because this tool only operates with a single table, performing all the operations that we described before.

6.2 STIL

Similar to *CFITSIO*, *STIL* [23], that stands for **Starlink Tables Infrastructure Library**, is also a library which allows the input, manipulation and output of tabular data and metadata. It is written in Java and it has been developed for use in astronomy. Besides the FITS, it also supports many others tabular formats: VOTable, text-based and SQL databases. All of them can be received as input and exported as output. If the user has a table format which is unsupported by *STIL*, a new input handler and a new output handler can be written. The user just have to follow the instructions on the documentation. It works only with tables and it can provide to the user information about:

- table metadata: all the information related the table name, the location, the number of columns, the number of rows, the description and some more additional comments
- column metadata: provide information about each column, such as name, type of the values, units of the values and a small description of the column
- table cell data: multi-dimensional array data of numerical, string or other types

STIL uses two different methods to access the data in their tables: random-access and sequential access. The sequential access starts with the first row and reads a row at the time, until the last one.

Here is an example of how to sum the values in one of the numeric columns of a table. Since only one value is required from each row, getCell is used:

```
double sumColumn( StarTable table, int icol ) throws IOException {

// Check that the column contains values that can be cast to Number.
ColumnInfo colInfo = table.getColumnInfo( icol );
Class colClass = colInfo.getContentClass();
if ( ! Number.class.isAssignableFrom( colClass ) ) {
   throw new IllegalArgumentException( "Column not numeric" );
}
// Iterate over rows accumulating the total.
double sum = 0.0;
RowSequence rseq = table.getRowSequence();
while ( rseq.next() ) {
   Number value = (Number) rseq.getCell( icol );
   sum += value.doubleValue();
}
rseq.close();
return sum;
}
```

In the random access it is possible to access the cells of a table in any order. Once the random access methods are called, the user needs to make sure that the table is a random table. In more detail, *STIL* supports the following input formats:

- *FITS*: As we described in the **Background** section, FITS tables can be divided into Binary and ASCII tables. Both are supported by the library. If only a single extension is required, this is indicated by giving the extension number after a # at the end of the table location. For example, *astro.fits#3* refers to the third extension (forth HDU) in the file *astro.fits*.

- *VOTable*: VOTable is an XML-based format for tabular data. It can read tables in which the cell data are included in-line as XML elements (VOTable/*TABLEDATA format*), or included/referenced as a FITS table (VOTable/*FITS*) or included/referenced as a raw binary stream (VOTable/*BINARY*).

- *ASCII text file*: In many cases tables are stored in some sort of unstructured plain text format, with cells separated by spaces or some other delimiters. There is a wide variety of such formats depending on what delimiters are used, how columns are identified, whether blank values are permitted and so on. It is impossible to cope with them all, but STIL attempts to make a good guess about how to interpret

a given ASCII file as a table, which in many cases is successful. CVS (Comma-separated value), TST (Tab-Separated Table), IPAC [1] (Infrared Processing and Analysis Center) and WDC [8] (World Data Center) are some of the examples that the *STIL* library supports.

As for the **output**, STIL supports the following formats:

- **FITS**: When saving in FITS format a new file is written consisting of N+1 HDUs (Header+Data Units) for N tables: the primary HDU (required by the FITS standard that has no interesting content), and subsequent ones (the extensions) are of type BINTABLE, one for each output table.
- **VOTable**: When a table is saved to VOTable format, it will write a well-formed VOTable document with a single resource element holding one or more table elements.
- **ASCII text file**: Tables can be written using a format which is compatible with the ASCII input format. It writes as plainly as possible, so should stand a good chance of being comprehensible to other programs which require some sort of plain text rendition of a table. Some more examples of output formats can be: **CSV**, **TST**, **Human-readable text**, **HTML**, **LaTeX** and **Mirage** (a powerful standalone Java tool for analysis of multidimensional data).

With a proper configuration, *STIL* can read and write tables from a relational database. The result of a SQL query on a database table can be handled by *STIL*. Also a new table can be properly stored into an existing database. Note that this does not allow you to work on the database *live*. The classes that control these operations are contained in the *jdbc* package. In short, what the user needs to do is define the "jdbc.drivers" system property to include the names of the JDBC drivers which the user wishes to use. For instance to enable use of MySQL with the Connector/J database the user might start up java with a command line like this:

```
java -classpath /my/jars/mysql-connector-java-3.0.8-stable-bin.jar:myapp.jar
     -Djdbc.drivers=com.mysql.jdbc.Driver
      my.path.MyApplication
```

STIL supports the following RDBMSs and drivers:

- MySQL: has been tested on Linux with the Connector/J driver; tested versions are server 3.23.55 with driver 3.0.8 and server 4.1.20 with driver 5.0.4

- PostgreSQL: the version 7.4.1 works with its own JDBC driver. Note the performance of this driver appears to be rather poor, at least for writing tables
- Oracle: uses the JDBC driver
- SQL server: There is more than one JDBC driver known to work with SQL Server, including jTDS and the Microsoft JDBC driver. Some evidence suggests that jTDS may be the better choice
- Sybase ASE: There has been a successful use of Sybase 12.5.2 and jConnect (jconn3.jar) using a JDBC

To read a result of an SQL query on a relational database as a table, the query string is as follows:

```
jdbc:<driver-specific-url>#<sql-query>
```

Here is an example for a MySQL database:

```
jdbc:mysql://localhost/astro?user=mb#SELECT ra, dec FROM swa WHERE vmag<18
```

To write a new table in an SQL RDBMS, the general form of the string which specifies the destination of the table being written is:

```
jdbc:<driver-specific-url>#<new-table-name>
```

Here is an example for a MySQL database using Connector/J:

```
jdbc:mysql://localhost/astro?user=mbt#table1
```

This query will write a new table called *table1* in the MySQL database *astro*.

When a document is accessed for the first time, the data present in the file is parsed and stored to *STIL* internal structures so it can avoid parsing operations later. The obvious thing to do is to store such data in object arrays or lists in memory. However, if the tables get very large this is no longer appropriate because memory will fill up, and the application will fail with an *OutOfMemoryError*. So sometimes it would be better to store the data in a temporary disk file. There may be other decisions to make as well, for instance if the data will be stored per row or per column. Those decisions are based on a set of policies, that will be listed:

- PREFER_MEMORY: Stores table data in memory. Currently implemented using an *ArrayList* of *Object[]* arrays

- PREFER_DISK: Generally attempts to store data in a temporary disk file, using row-oriented storage (elements of each row are mostly contiguous on disk)
- ADAPTIVE: Stores table data in memory for relatively small tables, and in a temporary disk file for larger ones. Storage is row-oriented
- SIDEWAYS: Stores data in temporary disk files using column-oriented storage (elements of each column are contiguous on disk). This may be more efficient for certain access patterns for tables which are very large and, in particular, very wide. It's generally more expensive on system resources than PREFER_DISK however, (it writes and maps one file per column) so it is only the best choice in rather specialised circumstances.
- DISCARD: Metadata is stored but the rows are thrown away.

The default policy is not specified explicitly, so each time *STILS* needs to know the policy that needs to use, it calls the method *StoragePolicy.getDefaultPolicy()*.

In the next paragraphs we will be analysing the strategies used by *STIL* for table processing. In the manual they advice to not read the sections related with the table processing if we are only interested in read tables in or write them out. But in order to have a clear idea how the library work, we will present here the strategies and decisions made by the tool. The first one is called *Writable Table* and writes all the data present in the table into memory. It is ideal if the data fit all in memory. The second is called *Wrap It Up* and makes solid use of the "pull-model" processing, in which the work of turning one table to another is not done at the time such a transformation is specified, but only when the transformed table data are actually required, for instance write in disk as a new FITS or display the information in a Graphical User Interface component. One big advantage of this approach is that calculations which are never used never need to be done. The second advantage is that we can process big large tables without allocating big amounts of memory. The concept here is to build a "wrapper" on the table, that will decide which portions of the table should be touched and brought to main memory. Working with wrappers can often be more efficient than, for instance, doing a calculation which goes through all the rows of a table calculating new values and storing them in a new table. *STIL* library provides a set of wrapper classes:

- *ColumnPermutedStarTable*: views the table with the columns in a different order
- *RowPermutedStarTable*: views the table with the rows in a different order
- *RowSubsetStarTable*: views the table with only some of the rows showing
- *JoinStarTable*: stick a number of tables together side-by-side

- *ConcatStarTable*: stick a number of tables together top-to-bottom

The wrapper classes can be also used and adapted to perform useful table processing. If the user follows a set of steps proposed by the *STIL* documentation, the following features are examples of what can be achieved: sort a table, turn a set of arrays into a new table (useful for gathering information spread along different tables), add a new column that will contain the sum of all the numeric cells in that row.

6.2.1 TOPCAT

Stands for **Tool for Operations on Catalogues and Tables** (*TOPCAT*) [25] is an interactive graphical viewer and editor for tabular data, based on the *STIL* library.

In the main menu it has a list of tables that were brought into the program. Those tables can be accessed (by visualization of the data present in the table), edited and rows can be added or deleted. New tables can be created based on existing ones. When a table in the list is selected, general information about the table is displayed and some action can be taken. The changes that are made do not directly modify the tables on disk. However, if the user wants to save the changes, the modified table can be written to a new location on disk. Since *TOPCAT* is built based on the *STIL* Java library, the user has full access to the table cell data, the table metadata and the columns metadata. Also for the table input and output formats, it supports the same file extensions as the *STIL* library. Some additional functionalities are also provided by this tool:

- *Row Subset*: reduces the number of displayed rows of the table. It can be done through three distinct methods: define a new row subset containing all selected rows, define a new row subset containing all the unselected rows or define an algebraic expression. All this subsets can be viewed as independent tables. It is an advantage comparing to the *Fv* tool, that do not allow this.

- *Row order*: sorts the table rows according to ascending or descending value of the contents of the column. Only available if some kind of order (e.g. numeric or alphabetic) can sensibly be applied to the column

  Column Set: during the lifetime of the table within *TOPCAT*, the list of columns can be changed by adding new columns, hiding existing columns, and changing their order. The current state of the columns present and visible and what order they are in is collectively known as the *Column Set*, and affects the display of the table. The current Column Set is always reflected in the order in which columns are displayed.

- *Statistics*: display statistics of each column: mean, standard deviation, minimim value, maximum value and the number of non-blanks cells

- *Histogram*: information about the distribution of the data, allowing the possibility to choose the bin size
- *Display of data*: The data can be displayed on a two-dimensional table, a three-dimensional, or even a spherical polar. This last one gives a pleasant and friendly idea how about the sources are spread along the land surface of the earth.

In contrast with *Fv*, this tool allows the user to join two or more tables together, in order to produce a new one. The join can be done *top-to-bottom* and *side-by-side*. A top-to-bottom join, also known as concatenation just requires that the user decides which columns in one table correspond to which columns in the other (in database terms it is called a *union*). A side-by-side join needs some sort of matching between rows in different tables. It can be *Pair match*, *Triple match* or *Quadruple Join*. For each one, it uses two, three and four tables respectively. It will return a new table with the tuples that match in the two, three or four tables. The result will depend on the algorithm that is chosen to perform the match criteria. The algorithms available were already presented in the **Performance Experiments** section. If the chosen algorithm is *Exact Value*, then it is similar to the equi-join used in database terms. In the other hand, if the algorithm is *1-d Cartesian*, then it is similar to band join. Conceptually, it is done by looking at each row in the first table and identify which rows in the second table "refer to the same thing" and put a new row in the joined table which consists of all the fields of the first table, followed by all the fields of the matched row in the second table.

The complication here is to define what is meant by "refer to the same thing". This may not be straightforward. There is also the problem of actually identifying these matches in a relatively efficient way (without explicitly comparing each row in one table with each row in the other, which would be far too slow for large tables).

Subsequently we will present an example given by the *TOPCAT* documentation that is interesting for a comparison with our use case. Supose we have the following catalogues:

Xpos	Ypos	Vmag
1134.822	599.247	13.8
659.68	1046.874	17.2
909.613	543.293	9.3

and

x	y	Bmag
-	-	----
909.523	543.800	10.1
1832.114	409.567	12.3
1135.201	600.100	14.6
702.622	1004.972	19.0

and we wish to combine them to create one new catalogue with a row for each object that appears in both tables. To do this, you have to specify what counts as a match. In this case let us say that a row in one table matches (refers to the same object as) a row in the other if the distance between the positions indicated by their X and Y coordinates matches to within one unit (sqrt($(Xpos - x)^2$ + $(Ypos - y)^2$)<=1)). Then the catalogue we will end up with is:

Xpos	Ypos	Vmag	x	y	Bmag
----	----	----	-	-	----
1134.822	599.247	13.8	1135.201	600.100	14.6
909.613	543.293	9.3	909.523	543.800	10.1

There are a number of variations on this. However, the match criteria of *TOPCAT* might involve sky coordinates instead of Cartesian ones (or not be physical coordinates at all). The match window of *TOPCAT* allows the user to specify which tables will be matched, what is the criteria for the matching rows and what rows will be included in the output table.

Here comes one of the main limitations of *TOPCAT*. It can only compare and match sources in their celestial coordinates (right ascension and declination). As a consequence, the use case studied in the **Functionality** section cannot be directly undertaken by this tool. However, there are two alternatives that can be followed in order to accomplish that goal. First, convert the spatial coordinates (x,y,z) into celestial coordinates (ra,dec) and then invoke the sky match join. The second alternative is to join the two tables with a cartesian join. This will give us all the possible combinations between the two tables and also access to all the columns present in the joined table. With this ability we can apply a filter to the joined table, calculating the distance between the two sources and filter out the rows that do not fulfill the condition. This alternative can be heavy to the system and might require a big memory consumption.

The basic algorithm for matching is based on dividing up the space of possibly-matching rows into an (indeterminate) number of bins. These bins will typically correspond to disjoint cells of a physical or notional coordinate space, but need not do so. In the first step, each row of each table is assessed to determine which bins might contain

matches to it - this will generally be the bin that it falls into and any "adjacent" bins within a distance corresponding to the matching tolerance. A reference to the row is associated with each such bin. In the second step, each bin is examined, and if two or more rows are associated with it every possible pair of rows in the associated set is assessed to see whether it does in fact consitute a matched pair. This will identify all and only those row pairs which are related according to the selected match criteria. During this process a number of optimisations may be applied depending on the details of the data and the requested match.

This means that the matching algorithm is precisely an *O(N log(N))* process, where N is the total number of rows in all the tables participating in a match. It has *loglinear* complexity, which is the case of *quicksort* and *merge sort* algorithms. This is good news, since the naive interpretation would be $O(N^2)$ (quadratic complexity). This can break down however if the matching tolerance is such that the number of rows associated with some or most bins gets large, in which case an $O(M^2)$ component can come to dominate, where M is the number of rows per bin. The average number of rows per bin is reported in the logging while a match is proceeding, so you can keep an eye on this.

TOPCAT is capable of dealing with large datasets. In fact, it does not read entire files into memory in order to do its work, so it is not mandatory to use files which fit into the Java virtual machine *heap memory* or into the physical memory of the machine. The program works with tables that contain millions of rows at a reasonable speed. However, the way the user invokes the program will affect how well it can cope with large tables. Sometimes the user can get the message *OutOfMemoryError* and for that, there are several things the user can do. Increase the Java heap memory is one alternative. When a Java program runs, it has a fixed maximum amount of memory that it will use. The default maximum is typically 64Mb. The *-Xmx* flag can be invoked in this case. For example, **topcat -Xmx256M** means that the size of the memory grow up to 256 megabytes. It is convenient that the user do not specify a heap size larger than the physical memory of the machine that is TOPCAT. The second alternative is the use of FITS files, as a consequence of the way they are organized and compressed. As well as speeding things up, using FITS files will also reduce the need to use *-disk* or *-Xmx* flags. The third alternative is the use of the *-disk* flag. The way *TOPCAT* stores table data is configurable. The default storage policy is *adaptive* and it means that the data for relatively small tables is stored in memory, and for larger ones in temporary disk files. This usually works fairly well, but the user can save some memory by encouraging it to store all table data on disk, by specifying the *-disk* flag on the command line. The forth alternative is to run in 64-bit mode. If the user is working with a file or files whose total size approaches or exceeds about 2 Gbyte, a 64-bit version of Java should be used. This means that a 64-bit operating system is required, and also a 64-bit version of the Java Virtual Machine. And the last and fifth

alternative is the use of a column-oriented storage. For really large tables storing them in the colfits output format can significantly improve performance. This stores all the elements of a single column contiguously on disk, which means that scanning through all the values in one or a few columns can proceed with much less unnecessary I/O than in normal (row-oriented) FITS format. It will make most difference when the table is larger than the amount of physical memory available, and the table has many columns. Be aware however that operations which require all the cells in all the rows (for instance, calculating row statistics) may be somewhat slower using this format.

To conclude, the memory size of the machine will make the difference. If the size of the dataset fits into unused physical memory then everything will run very quickly, because the operating system can cache the data in memory. In the other hand, if the dataset is larger than the memory, the data has to keep being re-read from disk and most operations will be much slower.

6.2.2 STILTS

The STILTS tool [24] was already introduced in the **Performance Experiments** section, where some stress tests were done in order to compare the performance against MonetDB.

It stands for **Starlink Tables Infrastructure Library Tool Set** and it is a set of command-line tools for table manipulation. Like *TOPCAT*, it is built based on the *STIL* java library, so the user will have full access to the table cell data, the table metadata and the columns metadata. Also for the table input and output formats, it supports the same file extensions as the *STIL* library.

Most of the functionalities of *STILTS* were already explored during the tests. So, in this section, we will rather go into a comparison with *TOPCAT* and also explore the limitations of this tool. It is better to use *STILTS* instead of *TOPCAT* when the user only wants to examine the metadata, a few rows, or even a statistical summary of the table without having to load the whole thing into *TOPCAT* or some other table viewer application. Another advantage of *STILTS* is if the user wants to perform a conversion from one file format to another, it is better to do it with a streaming application that executes the job easily and efficiently on the fly. *STILTS* provides that opportunity. That is done by explicitly describe in the command line what is the input table and in which format it is, and what is the output table and in which format it is. The formats can be any one of the available formats described in the *STIL* library. One case where *TOPCAT* is better than *STILTS* is plotting, because it was built with that purpose. It gives a friendly interface to the user so the data can be easily viewed in a variety of ways. *STILTS* only provides plot in 2d, 3d and the histogram. However, *STILTS* allows plots to be made from datasets of

unlimited size. While *TOPCAT* has an effective limit of a few million rows, *STILTS* can stream data from tables to do its plotting, so a plot can be made representing an unlimited number of rows without large memory requirements.

The same limitation studied in the *TOPCAT* tool, when trying to undertake the use case studied in the **Functionality** test, is also applied to *STILTS*, with the same reasons and the same alternatives.

STILTS also allows the user to change the heap memory size (by default is 64MB), provide the *-disk* flag (by default, the storage policy is *adaptive*) and run in in 64-bit mode.

6.3 Comparison between tools

Tool	Selection	Projection	Join	Union	Difference	Sort	Group by
Fv	X	X	-	-	-	X	-
TOPCAT	X	X	X	X	-	X	-
STILTS	X	X	X	-	-	X	-

Table 6.1: List of operations performed by the tools

In **Table 6.1** we can analyze what the three different tools that we studied before can and cannot accomplish. All of them were analyzed in detail in the previous sections. As a consequence, this table only tries to provide a better and general idea about the maturity of the tools.

MonetDB, as a powerful and robust database, can afford all this operations and reinforces our belief that this project brings innovation and it can be seen with credibility, as a future alternative to access and manipulate astronomical data.

6.4 Astronomical data formats

In this section we will do a brief introduction about the other existing formats that deal with astronomical data. We will also build a user matrix, listing a set of tasks that can be undertaken or not by the different formats.

6.4.1 HDF5 Array Database

HDF5 [9], and [14] can be seen as a *data model*, a *library* or a *file format*, and its main goal is to store and manage scientific data. HDF5 can store two primary objects:

- *datasets*: multidimensional array of data elements that can store almost any kind

of scientific data structure, such as images, arrays of vectors and structured and unstructured grids. The array variables and the respective elements of the multidimensional array can be stored and organized in two different ways:

Contiguous: as a single sequence in the *HDF5* array database

Chunked: as a collection of fixed-size regular sub-arrays

- *Groups*: allows the data to be organized in a tree-like structure, where the root is the group "/" which serves as an entry or reference point

An application that uses the HDF5 format for analysing, managing, manipulating and viewing data, desires flexibility and efficiency when the subject is I/O and the wish is the ability to store high-volumes of data, supporting an unlimited variety of datatypes. An application, a tool, or an high-level API interact with an *HDF5 array database* through the *HDF5 library API*, that allows the access and management of items within the *HDF5 array database*, called *HDF5 array variables*. As some applications of the *HDF5* file format:

- *NeXus*: data format for neutron, x-ray and muon science
- *HDF*: data format used by NASA's Earth Observing System (EOS) that gathers environmental data for a future reserach on global climate change. EOF program will contain more then 15 petabytes of data in 2015
- *JPSS*: stands for Joint Polar Satellite System. Is the sucessor of the EOS program and it will be used for climate and weather predictions, space weather observations and search and rescue detection
- *LOFAR*: stands for Low Frequency array. Is a multi-propose sensor array and its main application is astronomy at low frequencies (10-250 MHz)

HDF5 is the recommended standard format for storing earth science data. After a talk with K.Anderson, an astronomer that works in Science Park, Amsterdam and one of the authors of the document *LOFAR: Data Format Representation* [10], some of the differences between HDF5 and FITS were clarified. Firstly, *FITS* has been used for decades and it is the most used standard for the astronomical data. HDF5 is a new format, still being developed for some applications (for example, *LOFAR*). For other applications, outside the astronomical field, there is a standard and HDF5 is already being used. Secondly, FITS is a strong and robust standard with a vast number of libraries that can access and manipulate the data. The libraries of HDF5 are still being implemented. As a third and last statement, FITS has the limitation of scability, because all the data has to be aggregated

in one file. That is unacceptable when the size of the images or tables is hundreads of terabytes. HDF5 has the concept of groups, enabling the data to be spread along different nodes, instead of being stored in one single file.

6.4.2 VOTable

The second format that we will study is the *VOTable* [20]. *VO* stands for **Virtual Observatory**, that is a collection of archives containing astronomical data and software tools, that work together to form a scientific research environment, allowing astronomical research programs to be developed. The VOTable format is an XML standard for the interchange of data represented as a set of tables. A table can be seen as set of rows. Each row has a uniform structure, that is specified in the table description (the metadata). Each row in a table is a sequence of table cells. Each of those cells contains either a primitive data type, or an array containing a collection of such primitives. The main goal of this format is to provide a flexible storage and a interoperability of astronomical data, encouraged by the vast number of applications using XML. The data in a VOTable can be represented using one of three different formats: *TABLEDATA*, *FITS* and *BINARY*. The most common is the *TABLEDATA*.

The *TABLEDATA* is a pure XML format and has the advantage that XML tools can manipulate and present the table data directly. The metadata and the elements of the table are all reported in the XML document.

The *VOTable/FITS* format makes a *VOTable* compatible with the *FITS Binary Table* format. Given a FITS file that represents a binary table, the header of the FITS file, that contains the metadata, is converted into a VOTable. Each one of the *FITS keywords* is converted to a *PARAM keyword*, and the data itself is remotely stored and gzipped at an FTP site. The access is done by *streaming*. The VOTable specification does not define the behavior when the parser has to read the metadata twice. The parser can either ignore the FITS metadata or compare it with the VOTable metadata for consistency:

```
<RESOURCE>
 <PARAM name="EPOCH" datatype="float" value="1999.987">
  <DESCRIPTION> Original Epoch of the coordinates</DESCRIPTION>
 </PARAM>
 <PARAM name="TELESCOP" datatype="char" arraysize="*" value="VTel" />
  <INFO name="HISTORY">
    The very first Virtual Telescope observation made in 2002
  </INFO>
 <TABLE>
```

```
  <FIELD (insert field metadata here) />
   <DATA>
    <FITS extnum="2">
     <STREAM encoding="gzip" href="ftp://archive.cacr.caltech.edu/myfile.fit.gz"/>
    </FITS>
   </DATA>
 </TABLE>
</RESOURCE>
```

The third and last format is the **BINARY**. It is a sequence of bytes with the lenght specified in the *FIELD* elements in the metadata. The *encoding* attribute is a string that should indicate to the parser how to undo the encoding that has been applied:

```
<DATA>
 <BINARY>
  <STREAM encoding='base64'>
    AAAAA00zMUAlAQYk3S8bQESiDEm6XjUAAAADTTU3QHG0DEm6XjVAQIQ5WBBiTgAA
    AANNODJAYpgYk3S8akBRa7ZFocrB
  </STREAM>
 </BINARY>
</DATA>
```

All the *VOTable* files are written in XML, that makes them readable by any text editor. However, the data itself can only be visualized and read by humans if the format is *TABLEDATA*. For the other two formats (*FITS* and *BINARY*) an application needs to be called in order to read the data and present it to the user. The disadvantage of the *TABLEDATA* format is that is built to handle with small tables and it is not very efficient.

6.4.3 Comparison between file formats

Task	FITS	VOTable	HDF5
Metadata and data stored separately	-	X	-
Easy to stream	-	X	-
Specification of the number of rows in the table	X	-	-
Uses XML	-	X	-
See data in a text editor	yes for ASCII	X	-
Array can be stored as a table cell	X	X	X
Binary	X	X	-
Astronomical data	X	X	X
Quality check	-	X	-
Easy to detect a broken table	X	-	-

Table 6.2: Tasks performed for each one of the file formats

Store the data and the metadata separately is clearly more efficient, because if we are able to read the metadata first, the applications will be able to "get ready" for the input data and to organize some sort of parallel transfers of the data.

In the Grid scenario, work with large tables and perform data streaming between processors, with flows being filtered, joined, selected, etc, it would be very difficult if the number of rows of the table were required in the header (like in *FITS* files). In those cases, the whole table has to be streamed to cache, the number of rows have to be computed and afterwards streamed again for a further computation. This obstacle, makes the stream of FITS files a difficult task: it is a blocking operation for pipelined execution.

In contrast, for other operations it may be preferable to know the size in advance. For instance when loading data the application can allocate memory more efficiently if the size is known.

FITS files and *HDF5* files do not use XML and an application needs to be called in order to read the data and present it to the user.

Multidimensional arrays are one of the most complex structures that can be stored in a table cell, and they are supported by all the three file formats.

As for the binary format, it is not supported by the HDF5. The binary format is the most efficient to process, store, access and transmit large amounts of data and additional libraries are not required. In the *BINARY* format of the VOTables, no FITS library is required, and the streaming paradigm is supported.

FITS and *VOTable* were built to deal with astronomical data. The *VOTables* with astronomical tables more precisely. The *FITS* files with astronomical tables and also with astronomical images. The complex semantics and the large number of conventions that FITS has, makes it able to cope with the increasing complexity of astronomical instru-

mentation. The same scenario is not seen with *VOTables*.

Chapter 7

Conclusion

In this thesis we have been exploring the possibilities of an integration between astronomical data and databases. Astronomical data is present in our lives for many years, it is used by a whole community of scientists, in different formats with distinct purposes. However, a complete integration with the database world is not easy, because of the obvious complexity that it brings. It also requires a deep research, trying to understand how can the data present in the astronomical files can be described and brought into the relational database world. We focused on the study of that integration, finding the advantages and the disadvantages of it, and also which are the consequences of some of the decisions made.

7.1 Results and Overview

We started this project by exploring deeper into the structure of the FITS vaults, investigating which functionalities a database module should have to make the integration possible. We wanted to know what it takes to efficiently integrate a FITS file in MonetDB. In order to achieve a successful integration it is important to understand the following questions: Firstly, What is a FITS vault? Secondly, what metadata do the files inside the same vault have in common? And finally, how can the data model be efficiently expressed in a relational database system? All these questions were satisfied, and the concept of the FITS vault in conjunction with how they can be represented in the SQL catalog was acquired.

After a successful integration of the FITS files with MonetDB through the development of a module inside MonetDB code that provides a set of functionalities concerning to that integration, we started to think further. We performed an in depth comparison between the performance of a well known tool called STILTS and the performance of

MonetDB. The idea was to demonstrate that a delegation of work is possible and when and why MonetDB is faster and slower than STILTS.

The delegation of work was proved, showing how the same result can be obtained thought STILTS and also thought MonetDB. Point Query, Range operations, Projection, Statistics and Join operations are some examples of that delegation of work.

The results of the experiments were helpful, enabling us to discover some bugs in MonetDB, to give some architectural suggestions and to realize that STILTS and MonetDB compete side by side in the simple query tests, when the memory is empty and the data needs to be loaded (scenario that does not occur when the data is loaded and in main memory, being MonetDB much faster than STILTS). As for the complex queries (Join operations), the performance of MonetDB is outstanding comparing to the performance of STILTS. While MonetDB only needs a few milliseconds, STILTS takes thousands of seconds. These tests were proof that databases are in fact more powerful in answering complex queries that need more computation time and wise decisions about which algorithms should be executed.

As a final task, we took a set of tasks proposed by an astronomer working at CWI. Our aim was to demonstrate that these proposals could be undertaken using MonetDB. We accomplished this by building a tutorial with all the steps that an astronomer should do in order to get his results and made suggestions allowing for faster computation. As an example, we suggested that an auxillary table should be created, containing all the distances and brightnesses of all the objects. This allows for further filtering of results making the process faster, avoiding the need to calculate every time the query is posted.

7.2 Future Work

Integrating a vault into a database system is a complex topic where many problems must be noted. We studied that integration, but space is still open for further work and research.

Encapsulating functionality of external libraries During the tests we realized that *STILTS* is in fact a powerful tool that manipulates and accesses data present in the FITS files. This fact is proven on the Statistical tests for the second group of FITS files, where *STILTS* is actually faster than MonetDB in both hot and cold memory tests, for the last file with 470 million tuples. We made some suggestions in order to improve the performance of MonetDB. However, there is another viable alternative. Encapsulate functionality of external libraries, like the one used by *STILTS* in the Statistical tests. This requires a deep understanding of the *STIL* java library, that is the base of *STILTS*.

Load on demand It would be interesting if the user could write a query to the system and the system itself had the ability to automatically decide which tables should be loaded in order to give the answer to the user. This functionality is helpful for the cases where we have thousands of attached files and we do not know which tables we should load in order to answer our question. However, it is a feature that requires a deep study. Concepts like logic and artificial inteligence need to be brought to discussion, in order to let the system decide by itself what tables should be loaded. If the user is looking for information about the planet Earth, the system and the database must have present the concept of planets, solar system and so on.

Synchronization when repository is updated Sometimes we add, delete and rename the files in our directory. It would be appealing if whenever a change on the directory is applied, the system recognises it and performs that change also inside the SQL catalog. For this task, a repository with all the information about the files needs to be added to the system. It will store the current name of the file, the directory and also a flag, that is activated everytime when a change occurs in a file. That change will be reflected in the SQL catalog.

Creation of a new FITS file based on the result of a SQL query Instead of creating a new table and insert on it the results of a particular query, the idea is to export the result of the query to a completly new FITS file, which will contain the primary HDU and an extension. The extension will be the table that resulted from the query. For this exercise, we need to bring the results of the query into the code, in the form of BAT structures. Afterwards, this BATs can be accessed, and the data can be taken out, in order to create a new FITS file.

Bibliography

[1] Infrared Processing and Analysis Center. http://www.ipac.caltech.edu/. [Online, accessed 2010-10-07].

[2] MonetDB. http://monetDB.cwi.nl/. [Online, accessed 2010-10-07].

[3] NASA's HEASARC: Software. http://heasarc.gsfc.nasa.gov/fitsio/. [Online, accessed 2010-10-07].

[4] SDSS. http://www.sdss.org/. [Online, accessed 2010-10-07].

[5] SkyServer SDSS. http://cas.sdss.org/astro/en/tools/search/IQS.asp. [Online, accessed 2011-07-05].

[6] SpaceGuarduk. http://www.spaceguarduk.com/download-fits. [Online, accessed 2011-07-05].

[7] UKIDSS. http://www.ukidss.org/surveys/surveys.html. [Online, accessed 2010-10-07].

[8] World Data Center System. http://www.ngdc.noaa.gov/wdc/wdcmain.html. [Online, accessed 2010-10-07].

[9] *HDF5 User's Guide*, 1.8.7 edition, May 2011.

[10] L. Bahren, K. Alexov, A. Anderson, and J. Griemeier. *LOFAR Data Format ICD: Representations of World Coordinates*, 2.05.05 edition, May 2011.

[11] P. Boncz, S. Manegold, and M. Kersen. Database Architecture Optimized for the new Bottleneck: Memory Access. *In Proc. of the 25th VLDB Conference, Edinburgh, Scotland*, 1999.

[12] CSIRO. Resolution and Sensitivity
. http://outreach.atnf.csiro.au/education/senior/astrophysics/resolution_sensitivity.html. [Online, accessed 2010-10-07].

[13] FITS Working Group. *Definition of the Flexible Image Transport System (FITS)*, 3.0 edition, November 2010.

[14] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An Overview of the HDF5 Technology Suite and its Applications. *Uppsala, Sweden*, March 2011.

[15] J. M. Gonzalez. *Free Software / Open Source: Information Society Opportunities for Europe?*, 1.2 edition, April 2000.

[16] HEASARC. *CFITSIO User's Reference Guide*, 3.2 edition, December 2010.

[17] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[18] B. Irby. *Fv: The Interactive FITS File Editor*, 5.3 edition, July 2009.

[19] M. Ivanova, N. Nes, R. Goncalves, and M. Kersen. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. *In Proc. of the 19th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2007.

[20] F. Ochsenbein, R. Williams, C. Davenhall, D. Durand, P. Fernique, D. Giaretta, R. Hanisch, T. McGlynn, A. Szalay, M. Taylor, and A. Wicenec. *VOTable Format Definition*, 1.2 edition, November 2009.

[21] B. Pribyl, S. Feuerstein, and C. Dawes. *Oracle PL/SQL Language Pocket Reference*. O'Reilly Media, 1999.

[22] B. Scheers. *Transient and Variable Radio Sources in the LOFAR sky*. PhD thesis, Universiteit van Amsterdam, 2011.

[23] M. Taylor. *STIL - Starlink Tables Infrastructure Library*, 3.0-2 edition, June 2011.

[24] M. Taylor. *STILTS - Starlink Tables Infrastructure Library Tool Set*, 2.3-1 edition, June 2011.

[25] M. Taylor. *TOPCAT - Tool for OPerations on Catalogues and Tables*, May 2011.

[26] R. L. White. FIRST. http://sundog.stsci.edu/. [Online, accessed 2010-10-07].