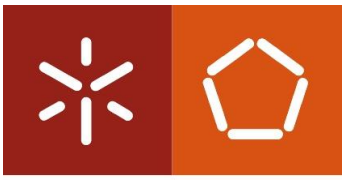


Universidade do Minho

Departamento de Informática

Sistema de Gestão de Vendas

Grupo Nº 39

Filipe Felício (85983)

Henrique Ribeiro (A89582)

João Correia (A84414)

May 30, 2020

Contents

Chapter 1

Introduction

The following pages serve the purpose of in-depth reviewing the sophistication of our software implementation of a modular Sales Management System that is capable of processing sales data in a large scale, whilst also being able to quickly provide products and clients info and answer multiple questions and challenges with optimal solutions.

With the keyword on the last paragraph being *sophistication*, this report shall sustain the thesis of our choices in terms of Algorithms, Data Structures and Architecture being the most effective, reliable and appropriate, within each of the singular queries as attested by a series of tests.

In order to achieve peak performance in a multi-variable set of goals, such as time, scalability, encapsulation and memory use, this study will reflect the potential of our own approach to solve the given task with logical allocation of resources to give the best overall experience to a user of this application, given that on a micro dissection not all the choices were optimal we aim at the highest degree of optimization at the amortized macro analyses.

Chapter 2

Problem interpretation and challenges

The proposed project was to develop a sales management system (SGV) capable of emulating a mid sized business. There are three files that detail the emulated environment, they are:

- A file containing Clients' IDs;
- A file containing Products' IDs;
- A file describing sales information containing the ID of the bought product, the ID of the client who bought it, the price of an individual unit, the number of units bought, whether a product was bought normally or through a sale and in which branch it was bought;

The program must also be able to answer 2 statistical queries upon reading the files, these are:

- Present data referring to the last sales file read, namely the name of the file, number of invalid lines, total number of products, total number of products bought, total number of products not bought, number of clients, number of clients which made purchases,the total value of those purchases and the number of sales that had a cost of 0.
- Number of sales by month, total billing by month and branch and number of distinct purchases made by a client in each month and branch.

Finally, the program must also answer 10 interactive queries:

1. List the ID's of the of clients which never made a purchase, sorted alphabetically.
2. Return the number of sales as well as the number of clients which made purchases in a given month.
3. Given a client's ID, indicate, for each month, how many purchases he made, how many distinct products he bought and how much he spent.
4. Given a product's ID, indicate, for each month, how many times it was bought, how many distinct clients bought it and the total value of those sales.
5. Given a client's ID, determine a list, sorted by quantity, of the codes of the products he most bought.
6. Given a limit number N, list the most bought products, regarding the number of units sold, indicating also the number of clients which purchased each one.
7. List, for each branch, the top 3 buyers in terms of sales value.

8. Given a limit number N, list the N clients which bought more distinct product, sorting the list by the number of products.

9. Given a product's ID and a limit number N, list the N most clients which bought it, sorting them by total expenditure.

10. For each month and branch, present the total value billed.

The main purpose of this project, as we understood it, was to learn the best practices of developing scalable mid-to-large scale software while keeping in mind crucial design principles, such as encapsulation and modularity.

One of the main challenges was the process of choosing the data structures, since it demanded a careful and methodical approach, as detailed on chapter 4.

Chapter 3

Project architecture and Modules

The project adopts the *MVC* (Model, View, Controller) architecture.

Since the project adopts an *Interface based programming* approach, the details exposed below are referent to the interfaces and not the which implement them.

3.1 Model

The model is the part of the software responsible for inserting, managing and manipulating data. In our program, the modules that constitute the model are explained below, being the interface module the main module, since it is the one who encapsulates all the other ones. A class diagram of the Model can be found on attachment A.1.

3.1.1 ClientInterface

The Client interface is responsible for representing a client. This interface is able to retrieve basic information about the client, such as it's ID. It's also responsible for verifying if a given ID follows the correct stipulation.

3.1.2 ClientsInterface

This is the interface responsible for storing all the clients. It is responsible for registering clients in the application as well as verifying if a given ID is already registered or not.

3.1.3 ProductInterface

This interface represents a product in a similiar logic to what happens with the interface CLientInterface. It must be able to answer basic queries about a product.

3.1.4 ProductsInterface

This interface represent all the products. It contains some metadata about them, such as the number of clients registered. As with the clients interface, this interface is also able to register and verify if a given product exists.

3.1.5 SaleInterface

This interface represents a sale. It must be able to retrieve all the relevant information about a sale, such as how many units were bought and also the total amount of money involved in that specific sale.

3.1.6 SalesInterface

This interface represent all the sales in the application. Although it contains information that can also be found in another parts of the app, it's inclusion was important as it facilitates some queries immensely. It will also ease the implementation of new functionalities, if desired, as it contains sales in the crudest form.

3.1.7 BranchInterface

This interface stores information about the sales made in a certain branch of our business. For each branch, it establishes a relation between all the products a client bought and all the clients to whom a product was sold.

3.1.8 BranchesInterface

This interface stores all the branches of our business. It must be able to answer queries regarding a specific branch as well as queries regarding all branches.

3.1.9 BillInterface

This interface represent a products' billing, that is, information about the total values of the sales in which a product is involved. It must be able to discriminate this information by month and branch.

3.1.10 BillsInterface

This interface it responsible for mapping each product with its corresponding BillInterface. This way, this module contains the information about all the values transacted in the application.

3.1.11 GuestVendasInterface

This class aggregates all of the model module. It will be called by the controller to answer every query, as well as load information from log files or recreate objects using *ObjectStream*.

3.2 Controller

This is the layer responsible for interaction with the user. It calls the View layer to present the options that the user may call. After an option is chosen, the Controller will then communicate with the Model layer to either change the internal representation of the data or to provide the answer to the chosen query.

3.2.1 Controller

This module implements functions capable of navigating the View layer whilst also calling the Model layer to obtain answers to the proposed queries.

3.2.2 QueryProcessor

This module is complimentary to the Controller class and is responsible for verifying the inputs the user submitted when calling a query. (eg: if the month is valid, if an ID follows the required stipulation,etc).

3.3 View

This layer is responsible for presenting the user with relevant information, either from loading/saving files or from answering queries. It will also be responsible for presenting large amounts of data in a format the user can easily navigate.

3.3.1 View

The view module presents the applications' menu to the user with a terminal-like format, in contrast to the approach taken in the C project. This decision was made as a way to try to give a new feel to the project.

3.4 Utilities

These are the modules who will be used by more than one of the above layers and, as such, cannot be labelled as one of them.

3.4.1 Query2Pair/Query3Triple/Query4Triple

These classes will represent pairs/triples that will be returned by queries in which it was decided that using Map.Entry or SimpleEntry was not the best option in terms of usability or interpretability.

3.4.2 Crono

Wrapper around *NanoTime* that will be used to measure how long each method takes.

3.4.3 Config

This class contains user defined macros for our program. The value of these macros can be modified to increase or decrease the application's dimension, making it possible to attain software scalability without having to know how the software is implemented. Some examples of these macros are: the number of different types of sales, the number of branches, as well as the default path to the info folders.

Chapter 4

Testing

A big part of developing software is testing: either testing performance or testing correctness. Therefore, a fair amount of our time was spent in trying to find the best structures and also ensuring they were serving their purpose correctly.

4.1 Performance tests

When designing mid to large scale software, one of the most crucial decisions consists on which data structures to use in order to strike a balance between memory usage, speed and scalability.

Different data structures we're tried for each module and the results will be presented in the following subsections. Every alternative implementation of the data structures is in the **test** folder of the project.

Description of the computer used for the tests: CPU: Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz. 4 cores, 8 threads. RAM: 8GB total, 3,4GB available

Number of times each test was made: 3

4.1.1 ProductsInterface

The main purpose of this module is to associate each product in a way that makes it fast to access each individual product. Knowing that, an implementation of the Map interface was decided as the most logical approach. Knowing that, an implementation of the Map interface was decided as the most logical approach. Since there are times where having the products already sorted by their ID may come to be useful, data structures with implicit ordering were also tested. Therefore the tested data structures were **Hashtable**, **HashMap** and **TreeMap**. An implementation using Lists was also tested but, since one of the main purposed of the productsInterface is to indicate if a product is registed in the application, this implementation was extremely slow, since the contains function is *O(Number of Products)*. The results using the map implementations are presented on annex B

4.1.2 ClientsInterface

As with the ProductsInterface, main purpose of this module is to have fast access to an individual client. Therefore, and since in no part of the program having the clients sorted by some natural order was important, the only two data structures testes were **Hashtable** and **HashMap**. These structures presented no relevant difference in performance and, as such, **Hashtable** was chosen.

4.1.3 SalesInterface

This module is responsible for holding all the sales. Since it doesn't need to have any particular organization or specific need, using the list interface comes as the most sensible choice. The

three different implementations tested were **ArrayList**, **Vector** and **LinkedList**. The results are presented on annex C. There was no significant change in performance with any of the different implementations, so we went with the ArrayList.

4.1.4 BillInterface

With this structure, instead of just changing which structure implementation was used, different ways of organizing the data we're tested. In the *first* one*(Bill)*, the data refering the the billing of a product is stored in multi-dimensional arrays of integers and floats, such as float[][][] expenditures where each dimension corresponds to a month, branch or type of sale. In the second implementation*(SingleBill and BillsAlternative)*, instead of a single object holding all the billing information of a product, each object only holds the information for a specific month, branch and type of sale These are then organized by the BillsAlternative class using Lists of Lists. This implementation, however, was slower than the first one, mainly because the first one uses primitive types such as float[] and int[] and the second uses the List interface and it's known implementation, which are slower.

4.1.5 ReadingBenchmarks

This module measures the efficiency of the program when processing files with 1, 3 and 5 million lines. Firstly only the time taken to read the lines and store them in a list is measured. Secondly the time to parse these lines, split them in the required fields and create the sales objects is measured. Finally also the time of validating those lines is accounted for. The times are presented on annex D

4.2 Unit tests

Unit tests are pieces of software where a single unit or component is tested. They are usually composed of some small input and an assertion that must be met for the test to be considered valid. In this project the testing library used was **JUnit5**. All the tested classes satisfied the required tests. These tests were also useful when trying different data structures, as they assured that the structures met basic requirements.

4.3 Final implementation

After testing all the structures, a final implementation was decided on, using Hashmaps and ArrayLists, since these were the ones which offered both a good performance and were relevant to the nature of the class in which the were used, which could help when developing new features.

The performance times are presented on annex E.

Chapter 5

Conclusion

In conclusion, the proposed queries are implemented efficiently with execution times which we consider within reasonable.

We feel this project is an improvement from our C project, since it now allows the introduction of new features in an easier manner, in comparison with the last project, where a part of the sales information was lost during the loading process.

Managing to reduce the load time would be on the top of our priorities as we feel there may be a potential benefit in using parallel streams.

Trying to implement data bases in our program would also be of our interest, as it would allow the application to scale exponentially more since it would not be bound by the heap size of the JVM.

Appendix A

Model class diagram

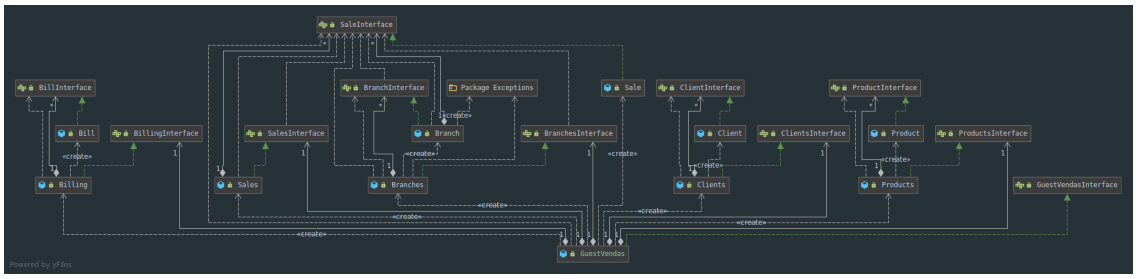


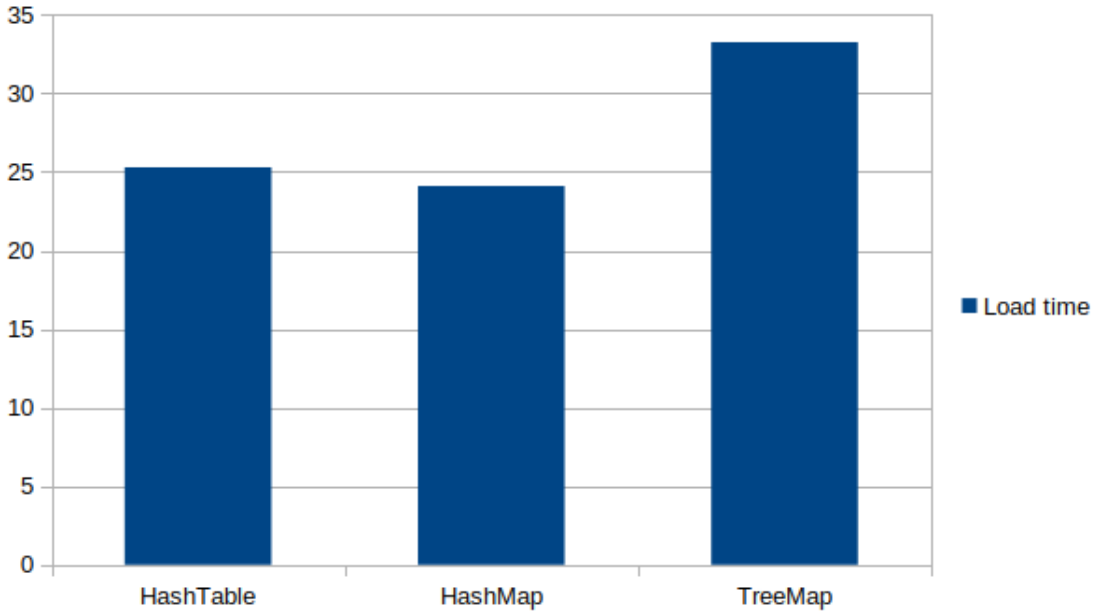
Figure A.1: Diagram of the Model module.

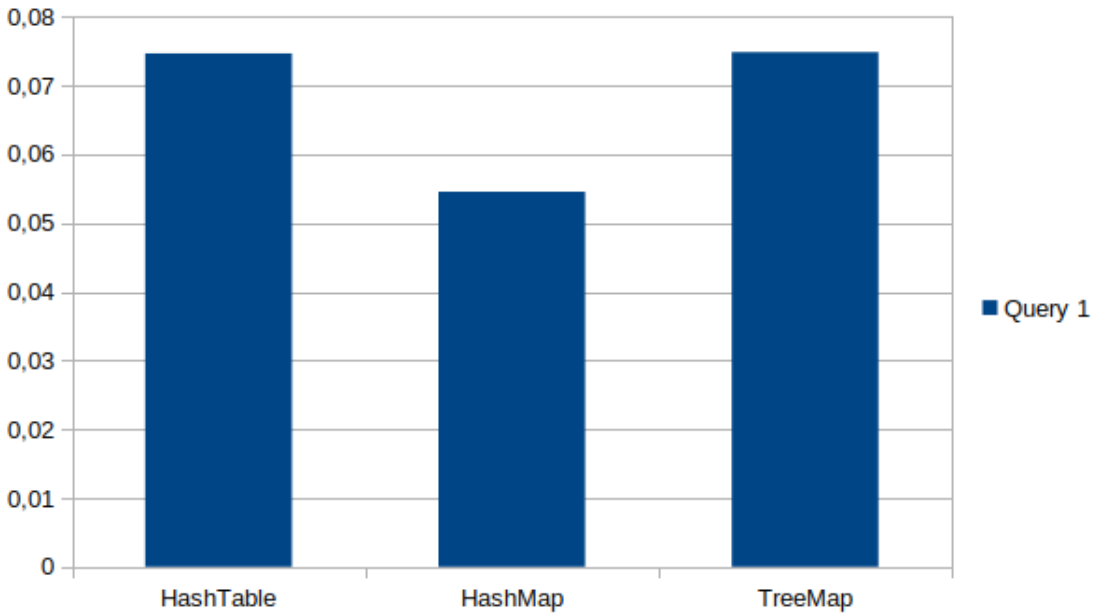
Appendix B

Products tests

Operation	HashTable	HashMap	TreeMap
Load time	25.29	24.10	33.25
Query 1	0.075	0.055	0.075

Table B.1: Load time and first query for different implementations of the Products class



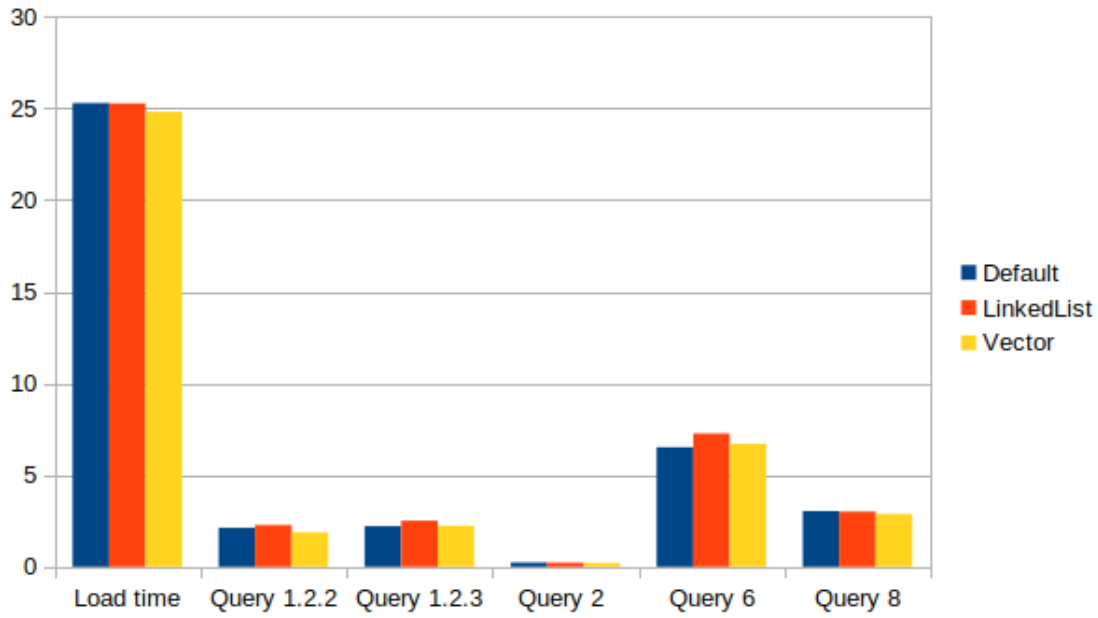

0,08
0,07
0,06
0,05
0,04
0,03
0,02
0,01
0
HashTable
HashMap
TreeMap
Query 1

Appendix C

Sales tests

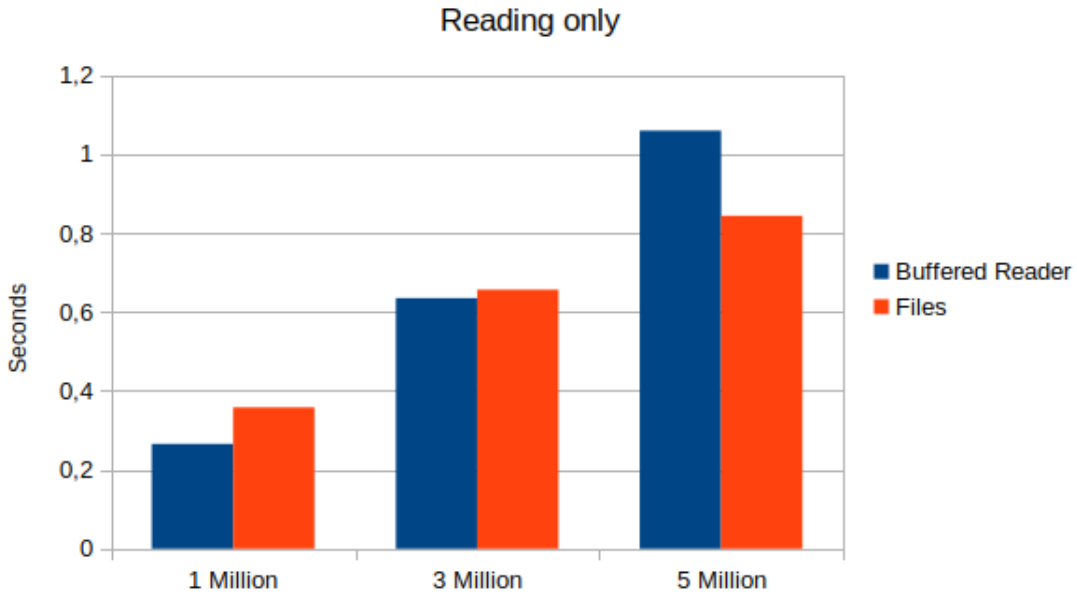
Table C.1: Time of important operations involving sales for different implementations.

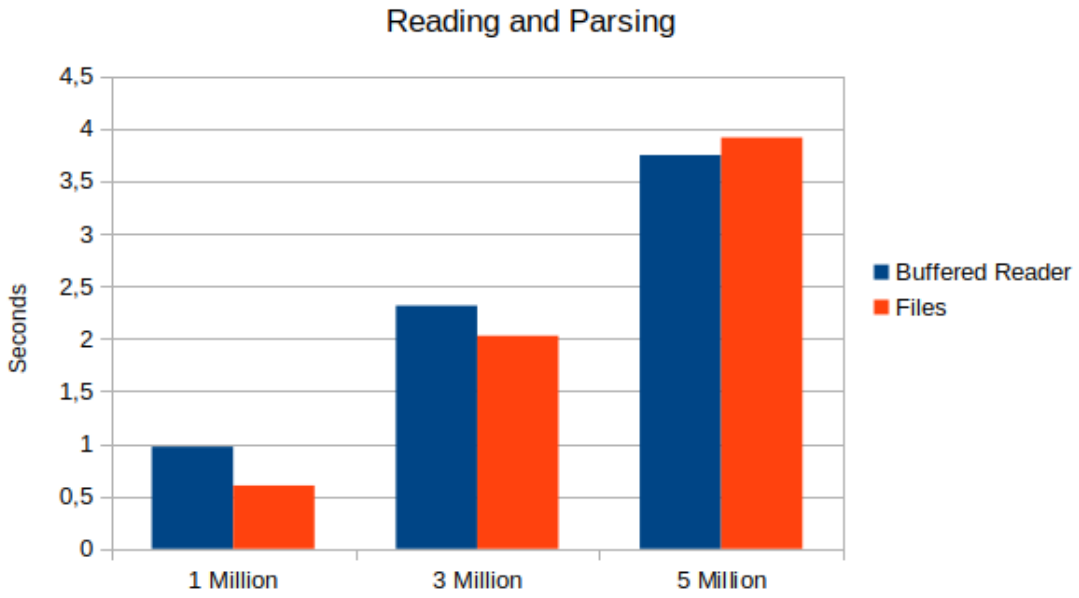
	Default	LinkedList	Vector
Load time	25,29	25,27	24,82
Query 1.2.2	2,12	2,28	1,88
Query 1.2.3	2,22	2,51	2,24
Query 2	0,23	0,22	0,21
Query 6	6,52	7,27	6,71
Query 8	3,04	3,01	2,88

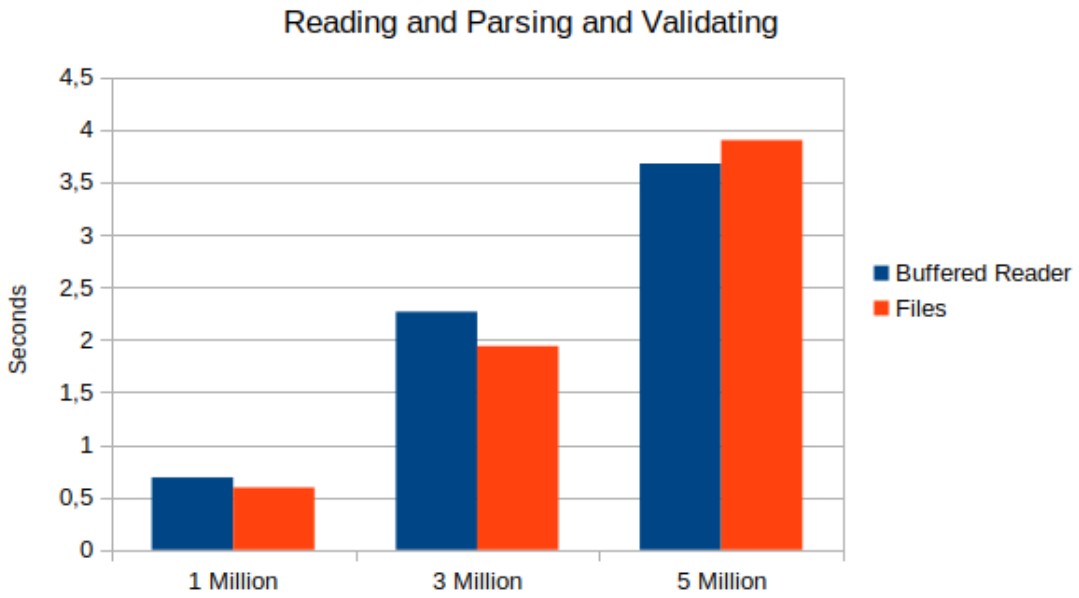


Appendix D

Reading benchmarks




Reading and Parsing
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Seconds
1 Million
3 Million
5 Million
Buffered Reader
Files


Reading and Parsing and Validating
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Seconds
1 Million
3 Million
5 Million
Buffered Reader
Files

Appendix E

Query and load time measures

Table E.1: Query runtime for different numbers of sales records

	Input	1 million sales	3 million sales	5 million sales
Load Time	———	7,836	14,758	25,291
Query 1.2.2	———	1,422	1,898	2,122
Query 1.2.3	———	0,403	1,457	2,217
Query 1	———	0,070	0,081	0,075
Query 2 part 1	Month = 2;	0,064	0,219	0,310
Query 2 part 2	Month = 2, Branch = 2;	0,042	0,147	0,226
Query 3	ClientID: “F2916”	0,005	0,009	0,005
Query 4	ProductID: “AF1184”	0,002	0,003	0,002
Query 5	ClientID:”L4891”	0,005	0,013	0,010
Query 6	Limit = 10;	1,240	3,445	6,119
Query 7	———	0,189	0,592	0,767
Query 8	Limit = 20;	0,528	1,925	2,976
Query 9	ProductID: “AF1184”, limit = 10	0,003	0,011	0,004
Query 10	———	0,245	0,275	0,271