Distributed and Multi-Threaded Learning of Regression Models

Pierre-Yves Bigourdan

St Hugh's College

University of Oxford

Supervised by Professor Dan Olteanu

Department of Computer Science, University of Oxford

Master of Science in Computer Science

Trinity Term 2016

Acknowledgements

First of all, I would like to express my most sincere gratitude to my project supervisor, Professor Dan Olteanu. I really appreciated our countless meetings and discussions related to many different aspects of the project, and am thankful for the feedback, advice and help he gave me during the last five months.

I also wish to thank Maximilian Schleich, one of Professor Olteanu's doctoral students who initiated the F project we will be building upon in this thesis. He provided valuable guidance throughout the duration of the project and our brainstorming sessions on implementation and experimental considerations were very enriching.

Finally, I would like to manifest my recognition to the University of Oxford as a whole for granting me the opportunity to study in such a stimulating place, and in particular St. Hugh's College which offered a very peaceful and friendly environment I was able to work and thrive in.

Abstract

Machine learning and databases are two closely related fields that have played an increasingly central role in the recent years, both in terms of theoretical work and practical applications in our daily lives. Unfortunately, most machine learning systems rely on their own dedicated way of representing and handling data, which makes integration with existing database systems more challenging and also slows down their development.

Some of the research conducted at the University of Oxford has been focused on better integrating these two fields, most recently with the development of F, a system capable of learning regression models over factorised joins. F introduces novel ideas on how to perform some common machine learning tasks on databases, and is capable of outperforming popular commercial systems by several orders of magnitude. Nevertheless, F is by design single-threaded and can only run on a single machine, limitations that undermine its scalability.

In addition to some extensive re-engineering work on F, the thesis focuses on these threading and distribution aspects, and shows how a state-of-the-art system can still be improved by orders of magnitude and guarantee a high degree of scalability. An end-to-end system is presented, capable of loading and parsing database tables, distributing the data around a cluster of machines, sorting it efficiently, launching an optimised and multi-threaded version of F on each machine, and merging the partial results together in order to produce the final result.

Contents

List of Figures

List of Tables

List of Code Excerpts

1 Introduction

1.1 Motivation

The development of the so-called ”big data” in the recent years has considerably increased the quantity of data that is produced, stored and processed on a daily basis. New applications have emerged and are playing a more and more central role in our daily lives. Nowadays, our extensive usage of social media, online gaming, cloud platforms or more generally our interactions with the Internet, have driven the ongoing innovations in data management. Although the price of both primary storage (essentially RAM) and secondary storage (essentially hard disk drives) has progressively fallen, storing and using data in an efficient way has become an even more central challenge.

Therefore, one of the aims is to avoid storage of duplicate information, in order to minimise the quantity of memory used, and avoid redundant computations, in order to boost the performance of algorithms. The FDB[1] project at the University of Oxford focuses on how to factor out information in databases and how to build lighter memory representations of the same information [2]. Many database operations and algorithms can take advantage of the factorised form to lower the computational cost of a given task by avoiding redundancies in the computations, and can therefore outperform standard systems which represent data and do calculations eagerly.

Nevertheless, the evolutions regarding the way data is dealt with have exponentially increased the complexity of calculations, and relying on more efficient algorithms and more succinct representations is no longer enough. In the past decade, the clock speed and the computational power of single CPU cores have progressively stagnated, but on the other hand there has been an increased focus on parallelism, with a growing number of cores built in a single CPU chip. New algorithms must adapt to this trend by taking advantage of multi-threaded processing in order to achieve optimal performance.

Furthermore, if a closer look is taken at the current systems that are in use nowadays, it can be noticed that many applications can no longer rely on a single database instance running on a single machine. By distributing operations over several machines, processing power can be considerably increased and more reliable and efficient services can be built. When considering a single machine, the main challenge was to design a robust data representation and fast algorithms; when considering a distributed system, new challenges arise, such as limiting the communication overhead inherent to any distributed system and reduce discrepancies in the amount of work performed by each node. The way communication is done and also what information needs to be communicated are problems that must be considered seriously.

[1]www.cs.ox.ac.uk/projects/FDB

Overall, it can be argued that distributed and factorised database systems are of interest in terms of storage capacity, processing power and scalability, and represent a current field of research with many applications in the industry. These advancements are not only confined to databases, but are also closely linked to machine learning. This field has been increasingly popular in the recent years, with evolutions driven by tier-1 web-based companies such as Facebook, Twitter or Google, but also by other sectors such as the pharmaceutical or retailer industries [8, 9]. Nevertheless, it has been more and more difficult for them to cope with the ever growing flow of information and complexity of calculations. One of the main issues comes from the fact that the database is generally seen as a separate and specialised item in the technology stack, and non-trivial time is often spent to export the data and convert it to a format that can be used by the machine learning software.

Better integrating machine learning and databases constitutes a topical area of research [10, 11, 12] and in particular, the F project conducted at the University of Oxford aims at dealing with some of the concerns previously highlighted. F was introduced through a recent publication [1] and presented at the ACM SIGMOD/PODS San Francisco conference in June 2016. It proposes a novel approach to learning regression models over factorised joins, and avoids redundant calculations and representations of information that are not needed for the final solution. Benchmarked against MADlib [26], Python StatsModels [27] and R [28], it manages to outperform these systems by up to three orders of magnitude. F is at the time of writing the state-of-the-art in terms of learning regression models, but even though its novel ideas bring new insights on how to perform machine learning tasks on databases, there is still room for improvement, in particular in terms of scalability. Currently F is single-threaded and can only run on one machine at a time, which makes it unsuited for very large datasets. This project addresses these concerns by introducing novel approaches on scalability and turns F into an even more competitive, well engineered and powerful system.

1.2 Contributions

The contributions to the F research project focus on three main areas:

- **Re-engineering of F.** By redesigning some parts of F, the time taken by an end-to-end run is improved by almost an order of magnitude. The following aspects are explored and implemented:
  - Compiler tuning, by selectively enabling several optimisations relevant to the project.
  - Redesigning the way data is loaded and parsed.
  - Redesigning the way data is sorted.
  - Tweaks to the core computation functions of F, among which data structure optimisation, function inlining and extended guidance provided to the compiler.

  - Profile-Guided Optimisations.
  - Loop unrolling.
  - Implementation of an enhanced build system.

- **Multi-threading of F.** By multi-threading the core calculations of F, it is possible to take advantage of all the available thread contexts of a given machine, and reduce the time taken to do the computations by a factor proportional to the number of launched threads. The thesis demonstrates how to ensure that multi-threading does not break the correctness of F's results, and investigates some of the challenges that arise when partitioning the tasks between several threads. Merging parallelism and factorisations is a yet unexplored field, and novel insights are introduced on how to handle this.

- **Distribution of F.** By creating a distribution framework, F is made highly scalable and can be run on a cluster of machines. The project provides an implementation of the state-of-the-art Hypercube algorithm [5], which ensures correctness of F's computations when shuffling the database tables and which limits skew in the data partitioning. This distributed environment enables to use F in parallel on an arbitrary number of machines, and takes care of gathering and producing a final centralised result. Adapting the Hypercube algorithm to a factorised machine learning setting is an unprecedented accomplishment and provides F with a tremendous scalability.

The overall contribution is an end-to-end piece of software implementing several novel approaches, and capable of parsing command line options in a user-friendly manner, loading and parsing database tables contained in CSV files stored on disk, synchronising and shuffling data around an arbitrary number of nodes, sorting the data efficiently, launching an improved version of F to calculate regression aggregates in a multi-threaded fashion, and finally gathering all the partial calculations in order to produce the final result. We are currently in the process of wrapping up the whole work, and preparing a publication aimed at presenting the theoretical and experimental aspects of our work.

The whole project is written in the C++ language, and is compatible with any modern versions of the GCC or Clang compilers, as well as with different Unix distributions, including Mac OS X. Even though the thesis is mainly focused on F, the implementation is designed to be as modular as possible, in such a way that integrating a different machine learning task in the future while still keeping a similar wrapping would require little recoding work. The implementation follows some rigorously defined coding conventions, and is available in a private repository on *github.com*, with instructions to set it up and run it in a new development environment, as well as files to generate a comprehensive Doxygen [29] documentation of the code. Throughout the thesis, a mixture of new theoretical insights, experiments and code excerpts are provided, but giving a complete coverage of the almost 7000 lines of code contained in the system is obviously beyond the scope of this document. Nevertheless, in addition to the USB key handed in with the printed version of this report, a full access to the GitHub repository of the project can simply be requested by contacting Dan Olteanu (*dan.olteanu@cs.ox.ac.uk*).

1.3 Overview

Sections 2 gives a brief presentation of the initial F system, in order to understand the main challenges of this project.

Sections 3 goes through the different engineering aspects previously mentioned, and provides some compiler and language oriented insights as well as experiments to quantitatively measure the performance improvements.

Section 4 and 5 detail the multi-threading and distribution aspects of F. In these two parts, theoretical background and novel insights are presented; the work that was accomplished is then detailed, and finally a series of experiments to show the impact of the new implementations is given.

Section 6 is aimed at bringing all the contributions to F together, and running it in the best possible setting at our disposal, with both multi-threading and distribution enabled.

Section 7 is focused on giving a conclusive overview of the project, and suggesting some new paths to explore in order to further build upon the F research project in the future.

Appendix A contains references to the different publications and tools that were used in this project, as well as references to some relevant C++ documentation to better grasp the implementation choices of the project.

Appendix B gives information about the datasets used in the experiments.

Appendix C provides details about the hardware used and other experimental considerations.

2 F: Learning regression models over factorised joins

The F project was started at the University of Oxford in 2015 and enables to build least square regression models over arbitrary join queries on database tables. This system computes batch gradient descent by taking advantage of a factorised computation, by decoupling the calculation of the model's cofactors from their convergence and by exploiting the commutativity of cofactor computation with relational union and projection. A brief overview of how F works is given; the recent publication [1] in the SIGMOD journal details the research that was conducted and should be referred to for a more complete understanding of the underlying theoretical aspects.

This section first introduces the context in which F is used, then goes deeper into factorisations; a higher level component view of the program is provided and finally performance figures about the system as it was at the start of this project are given.

2.1 What is F used for?

In the following section, the example presented in the SIGMOD publication [1] is reproduced to show how F works.

F considers the join of several database tables as an input, and applies machine learning regression models to analyse the data and make predictions. For instance, F can be used to predict the price of houses based on database tables containing information about geographical locations, population, market trends, local facilities, amount of taxes, or any other data source that could be relevant.

As stated in *Database Systems - The Complete Book* [14], "the natural join of two relations R and S, denoted $R \bowtie S$, [...] pairs tuples from R and S that agree in whatever attributes are common to the schemas of R and S". Consider a database containing three relations, *Shops*, *House* and *TaxBand*, as displayed below on the left part of the figure. To join the three tables, the common values for attributes Z and S must be paired, leading to the result displayed on the right:

Shops

Z	H
z_1	h_1
z_1	h_2
z_1	h_3
z_2	h_4

House

Z	S	P
z_1	s_1	p_1
z_1	s_1	p_2
z_1	s_2	p_3
z_2	s_2	p_4

TaxBand

S	T
s_1	t_1
s_1	t_2
s_1	t_3
s_2	t_4
s_2	t_5

Shops ⋈ House ⋈ TaxBand

Z	H	S	P	T
z_1	h_1	s_1	p_1	t_1
z_1	h_1	s_1	p_1	t_2
z_1	h_1	s_1	p_1	t_3
z_1	h_1	s_1	p_2	t_1
z_1	h_1	s_1	p_2	t_2
z_1	h_1	s_1	p_2	t_3
z_1	h_1	s_2	p_3	t_4
z_1	h_1	s_2	p_3	t_5
		....		
	the above for h_2 and h_3			
		....		
z_2	h_4	s_2	p_4	t_4
z_2	h_4	s_2	p_4	t_5

Figure 1: Shops, House and TaxBand tables with their corresponding join result

F uses a similar join result as an input to train its regression models. To do this, the result of the join is interpreted under the following form, where each group of values corresponds to a database tuple, or in other words a line in the join result:

$\{(y^1, x_1^1, x_n^1), ..., (y^m, x_1^m, ..., x_n^m)\}$

The objective is to define a function $h_\theta(x) = \theta_0 + \theta_1 x_1 + .. + \theta_n x_n$ such that given a new set of features $\widetilde{x} = (x_1, ..., x_n)$, it is possible to predict the value of the unknown label y such that $y = h_\theta(\widetilde{x})$. The θ variables used in the h_θ function are called the parameters of the model. For instance, in the previous example, one possible task would consist in predicting a new value P given a value for Z, one for H, one for S and one for T.

To correctly learn the parameters of the model, the most natural approach would be to minimise the prediction error of the h_θ function on the training dataset, in other words on the join table for which we already know the y labels. If the function closely matches the data currently available, it is possible to predict the value of a new label y for data that is not yet available, by assuming that the new data follows similar patterns or trends, and that there are correlations between the values of different table columns in the database.

A common approach to minimising such a prediction error consists in considering what is called the least squares objective function, and apply batch gradient descent to learn the parameters of the model. The full mathematical analysis is not provided here as it is not required in order to understand the work done in this thesis, but can be found in the SIGMOD publication [1]. Nevertheless, it is important to state the fact that this approach leads to calculating multiple cofactors, which are combinations of database values, also called aggregates, of the following form:

$Cofactor[j, k] = \sum_{i=1}^{m} x_k^i x_j^i$

The method used by F decouples the calculation of these cofactors from the computation of the parameters, so it is possible to first obtain all the cofactors, and use them during a convergence step in order to compute the final θ parameters of the considered model.

2.2 Factorisations in F

In order to calculate and represent the join results which are used to learn linear regression models during the cofactor computation phase, F relies on a factorised form. By taking advantage of relational algebra laws such as distributivity of the Cartesian product over union, factorised databases reduce data usage and redundant calculations [2].

By observing the join result shown in the above example, it can be noticed that it features a high degree of redundancy, with each value appearing numerous times in repeating combinations. By not eagerly materialising the join result and by representing the Cartesian products and the unions symbolically, the following factorised representation can be obtained, which contains only 18 values instead of the 130 ones in the eager join:

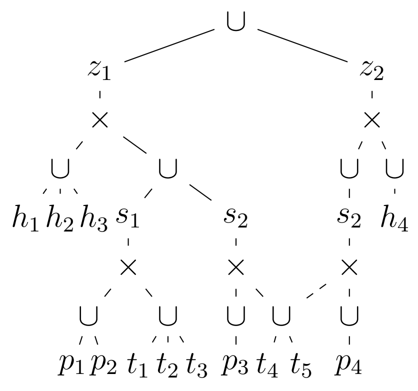


Figure 2: Factorisation of the join result example

For instance, for a given value of attribute Z, enumerating all the combinations with values of attributes H and S is avoided by instead using union and Cartesian products. This saves both space and computation and is one of the key elements to F's performance.

The previous factorised representation of the join result follows a nesting structure, which corresponds to what is called a d-tree [3]:

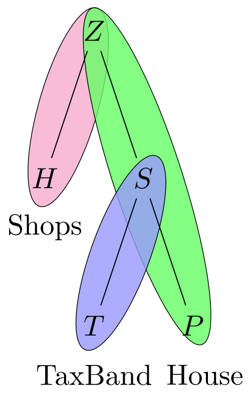


Figure 3: Database d-tree example

When working on a specific dataset, F is given a similar d-tree contained in a configuration file, which enables it to know how to build the factorisation depicted in Figure 2. Indeed, for a given database query, the d-tree is in general not unique. In the previous example, the join between tables *TaxBand* and *House* could have been moved at the root of the d-tree, and the join between tables *Shops* and *House* would have therefore been situated at a lower level in the d-tree. The resulting factorisation would have been different, yet perfectly correct.

As can be noticed at the bottom right of the factorised join result in Figure 2, the factorisation form goes even further by caching common sub-expressions. Each value s_2 is paired with the values $t_4 \cup t_5$, which can be stored a single time and reused for every occurrence of s_2. By doing so, recomputing redundant regression aggregates during the cofactor computation phase is avoided. F relies heavily on this caching approach to boost its performance, and preserving caching is a key challenge when multi-threading F in Section 4.

In order to learn regression models, regression aggregates must be computed over the join result. Joining can be expensive both in terms of computational cost and memory usage, and to avoid materialising the factorised join explicitly, F intertwines join and cofactor calculation, and iterates over paths of values in the tables that follow paths of attributes given by the d-tree. To make this iteration over values possible, the relations are sorted following a partial order given by a depth-first traversal of the d-tree. Indeed, F internally uses an implementation of the LeapFrog TrieJoin algorithm to iterate through these paths [7], and this specific sorting order is necessary for the process.

2.3 Main components of F

The main components of the F system can be represented as shown in the figure below. These four components are the ones on which the benchmarking experiments are focused, with two additional ones when distributing the system (communication over the network and synchronisation).

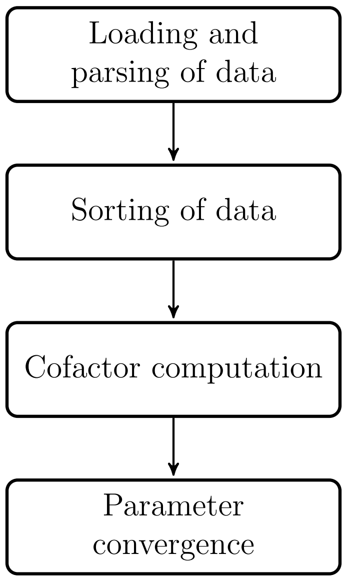


Figure 4: Main Components of F

- F relies on datasets contained in CSV files with a structure similar to the following:

  ```
  0.84795042898|0.226739223218|0.000167282398206|
  0.84795042898|0.262857447717|0.000144600039127|
  0.84795042898|0.341389244558|0.000419623642957|
  ```

  The objective of the first component is to load these files initially stored on disk, parse them, and arrange the values into arrays, each array corresponding to a distinct tuple.

- The sorting of the data is done in the second component in order to follow a partial order given by a depth-first traversal of the d-tree, as required by the cofactor processing.

- In the third component, the system intertwines cofactor and factorised join computation.

- Finally, by using the cofactors computed during the previous step, F calculates the parameters corresponding to a given model by performing batched gradient descent. This component is referred to as "convergence".

2.4 Reported performance

As demonstrated in the SIGMOD publication [1], F can outperform commercial systems such as MADlib, Python StatsModel and R by up to three orders of magnitude, which makes it an extremely competitive system. The thesis focuses on the US retailer, LastFM and MovieLens datasets; they are described in more details in Appendix B, and the following table reproduces an excerpt of the timings reported in the original paper.

		US retailer	LastFM	MovieLens
Time (ms)	**F**	16290	250	2120
	MADlib	680600	196600	7080
	R	2249190	804620	19120
	Python StatsModel	2613310	539140	23550
Speedup	**F** vs. MADlib	41.78×	786.40×	3.34×
	F vs. R	138.07×	3218.48×	9.02×
	F vs. Python StatsModel	160.42×	2156.56×	11.11×

Table 1: Performance of F vs. state-of-the-art

The timings only account for cofactor and convergence calculation. The times relative to the two first components of F, namely data loading and parsing as well as sorting of the tables, were not reported in the paper, but are monitored in the upcoming experiments to give a more complete end-to-end overview of the system.

3 Re-engineering of F

As stated in Section 2.4, F significantly outperforms its competitors in terms of cofactor and convergence computation. The aim of this section is to show how it is still manageable to greatly improve these performance timings and build a robust system that also becomes competitive when loading the data, parsing it, storing it in memory and sorting it. From now on, the version of the F program that is presented in the SIGMOD paper [1] is specifically referred to as ***OldF***; this constitutes the starting point of the project, and ***F*** is used to refer to the system as it evolves over time through the contributions that are exposed.

This section first introduces investigation about how to tune the compiler to improve OldF's performance. Implementation work on the data loading and sorting components is then presented, as well as enhancements on the cofactor and convergence components. Two other families of optimisations are explored, namely loop unrolling and profile-guided optimisations. Finally, the build modes created to compile and modify F are introduced.

3.1 Compiler flags tuning

Unfortunately, the times reported in the SIGMOD publication [1] were run on a version of OldF that did not make use of any compiler flags. This thesis explores the area of compiler optimisations in order to generate enhanced assembly code. Letting the compiler optimise the C++ code is not only beneficial in terms of performance, but also allows for a fairer comparison with commercial systems which make use of powerful compilers to speed up their code. The following flags were selected when building with the GCC compiler:

- *-Ofast* - This is the highest global optimisation flag available in GCC. Similarly to the widely used *-O3* flag, it enables numerous standard compiler optimisations such as loop vectorisations, constants merging, loop nests optimisations, common sub-expression eliminations, optimised instruction scheduling, better stack handling, and many more; an exhaustive list is available in GCC's documentation relative to optimisation flags [15]. In addition, this flag also enables some optimisations that are aimed at increasing the speed of floating point computations, which are widely used in F.

- *-fassociative-math* - This flag allows the compiler to re-associate operands in floating point calculations.

- *-freciprocal-math* - This flag allows the compiler to use the reciprocal of a value in a division. For instance $\frac{x}{y}$ can be changed to $x.\frac{1}{y}$, which may allow the compiler to simplify some floating point expressions.

- *-fno-signed-zeros* - The C++ standard specifies two zeros for float numbers, +0.0 and -0.0; this flag allows the compiler to ignore the signedness of a zero and to treat them similarly.

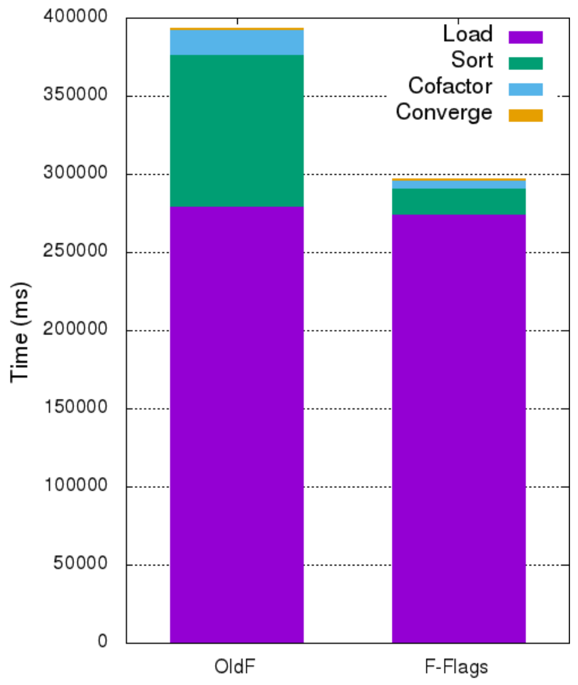
- *-frename-registers* - This flag allows the compiler to improve register allocation and is most beneficial for processors with many registers. This is generally the case with recent CPUs, and in particular with the machines used to benchmark F.

- *-mtune=native* - This flag tunes the compiled code for the specific CPU of the machine and favours instruction sequences that run faster on that CPU. This is relevant for the current benchmarking environment; nevertheless, if F is compiled on a machine with a given CPU architecture but deployed on a machine that uses a different CPU architecture, this may affect performance negatively. More information is available in GCC's documentation relative to options [18].

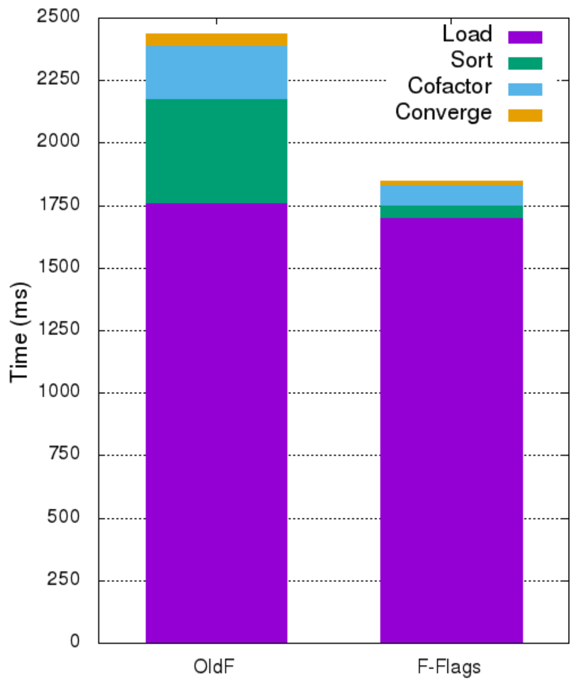
Some of the optimisations enabled by *-Ofast* as well as the *-fassociative-math*, *-freciprocal-math* and *-fno-signed-zeros* flags disregard strict compliance with the C++ standard. They allow operations that are not permitted by the IEEE representation of float point numbers [41], but can increase the performance of floating point calculations for programs that do not rely of the exact implementation of the standard [15]. F heavily uses floating point calculations in the cofactor and convergence phases, and after running some extensive tests and verifying the accuracy of the results obtained with these flags enabled, it was concluded that non strict compliance didn't alter the computations negatively and that OldF could be safely compiled with these flags.

Research and benchmarking was conducted to examine other flags that appeared as candidates to improve OldF's performance. Nevertheless they did not bring any noticeable improvements and even affected performance negatively in some cases; consequently the six flags listed above were selected in the end. They will be used from now on throughout the different experimental sections of the project, when reporting timings for F, MF or DF (respectively the multi-threaded and distributed versions of F). These six flags are referred to as ***F-Flags*** from now on.

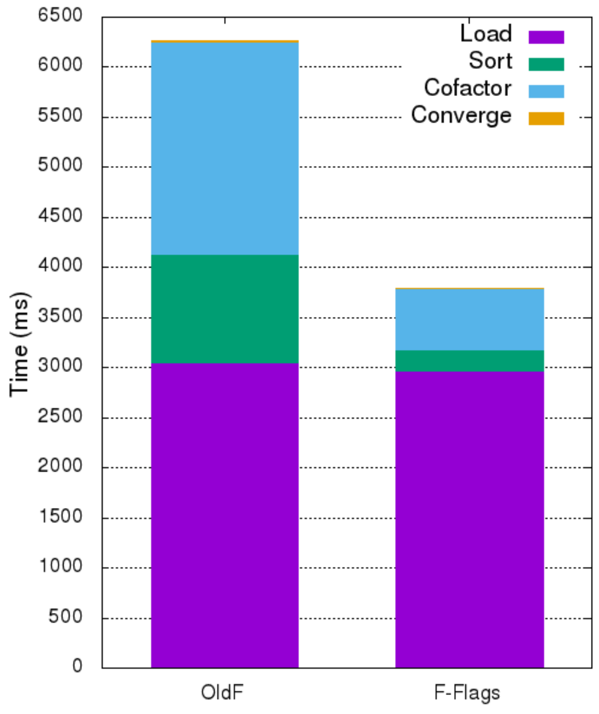
In the following figure, it can be measured to what extent a compiler can improve a given piece of software, OldF in this specific case. The speedups observed by using F-Flags over OldF alone are indicated below each plot. The three datasets described in Appendix B are used as a point of comparison.



(a) US retailer: 1.02× load, 5.63× sort, 3.10× cofactor, 3.66× converge



(b) LastFM: 1.03× load, 8.33× sort, 2.51× cofactor, 4.00× converge



(c) MovieLens: 1.03× load, 5.12× sort, 3.47× cofactor, 3.25× converge

Figure 5: Performance with or without F-Flags

Based on the plots, the components of F can here be divided in two separate groups:

- **Load:** for this component the speedup is relatively marginal, accounting for a few seconds for US retailer and a few milliseconds for the two other datasets. This component essentially relies on I/O handling and the choice of the parsing function, which the compiler has little effect on.

- **Sort, cofactor and convergence:** for these components the speedups are much more significant. The code basis involved is more extensive, with a lot of loops, functions related to algorithms, mathematical expressions and operations on data structures. The compiler has a lot more freedom when optimising such elements and can more easily perform optimisations at the level of a whole compilation unit, in other words by considering the code of an entire class for instance.

Therefore, by understanding how the code behaves, what the target system is and by making efficient use of a compiler's manual, it is showed that it is possible to significantly improve performance of a piece of software without modifying a single line of code.

3.2 Redesigning data loading and parsing

As can be noticed on the above figure, the loading and parsing of the CSV data in OldF is extremely slow, accounting for more than 90% of an end-to-end run with F-Flags activated. This not only limits OldF's scalability compared to database systems such as PostgreSQL which can load huge amounts of data in memory within seconds, but also makes extensive testing or benchmarking difficult, as a single execution takes several minutes as soon as the sizes of the files reach a few gigabytes. Improvements on this component of OldF cover three main areas:

- **Recycling C++ objects:** by extracting variable instantiations out of loops, such as the string receiving the lines from the CSV data files or the counters used for the column index in a tuple, the same objects can be reused throughout the parsing of datafiles. For tables containing millions of tuples, this prevents from instantiating millions of single usage new objects, which saves a considerable amount of computational power.

- **Using more adapted C++ file handlers:** OldF used the *FILE* type with the *fopen* function [30], which are old constructs from the C language. By switching to the *std::ifstream* class [31], which is a more modern construct exclusive to the C++ language, it is possible to achieve better I/O handling of the different data files.

- **Modifying the parsing function:** OldF used the *fscanf* C function [30], which featured poor performance when parsing a CSV line extracted from a data file. A first attempt consisted in inputing the file lines in an *std::stringstream* object [32], and then extracting the values from the stream by searching for CSV separators and casting them to *doubles* by using the *std::stod* C++ method [33]. This already gave significant performance improvements over the approach used in OldF. However, after some extra research and testing, it was decided to switch to the Boost Spirit library [21], which further divided the loading and parsing time by a factor 2. Spirit is a powerful header-only library, which can support complex parsing grammars and achieve excellent performance. Its syntax is slightly unusual, but allows for greater freedom when writing expressions. Here is for instance a code except used to parse a CSV line (similar to 0.84795042898|0.226739223218|0.000167282398206|), contained in the *line* string variable:

Code Excerpt 1: Parsing of a CSV line

```
1  /* Create a new tuple. */
2  Tuple tuple = new DataType[_attrIDs[table].size()];
3
4  column = −1;
5
6  /* Parse each double one by one; skip VALUE_SEPARATOR_CHAR. */
7  parsingSuccess = qi::phrase_parse(line.begin(), line.end(),
8
9      /* Begin Boost Spirit grammar; qi::_1 contains the parsed
          double value. */
10     (repeat(_attrIDs[table].size())
11     [qi::double_[phoenix::ref(tuple)[++phoenix::ref(column)]
12        = qi::_1]]),
13     /* End grammar. */
14
15     VALUE_SEPARATOR_CHAR);
16
17 assert(parsingSuccess && "The parsing of a tuple has failed.");
```

A new tuple to insert in the database is created on line 2. The *qi::_1* on line 12 represents the binary fragment to be parsed. Inside a Spirit grammar, a *phoenix::ref* must be used to pass standard C++ object references to the parser. A reference to the *column* variable and to the *tuple* are provided to the parser, which interprets the *qi::_1* fragment as a double (*qi::double_* on line 11) and writes the value in the tuple at the specified *column* index. This parsing is repeated for each column in the tuple (line 10); the *column* index is incremented inside the parsing grammar on line 11. On line 15, a separator is specified (| in the current setting) for the parser to know when to move on to the next value to parse.

The following table gives more insight on the performance improvements that were achieved:

		US retailer	LastFM	MovieLens
Load and parse time (ms)	OldF	279864	1762	3045
	OldF + F-Flags	274387	1703	2965
	F	32393	218	383
Speedup	**F** vs. OldF	8.64×	8.08×	7.95×
	F vs. OldF + F-Flags	8.47×	7.81×	7.74×

Table 2: Performance after redesigning the load and parsing component

These three areas of research and improvements enabled to speed up the loading and parsing of data files by almost an order of magnitude on the three datasets. Runs that would previously take several minutes due to OldF's poor loading can now be completed within seconds. On the machine used for benchmarking in this section (detailed characteristics can be found in Appendix C), the 5.3 gigabytes of the US retailer dataset are loaded, parsed and stored into in-memory data structures in about 32 seconds, which accounts for a processing speed of 170 MB/s for CSV data.

3.3 Redesigning data sorting

Using compiler optimisations already increased the speed of the data sorting by 4 to 8 times depending on the dataset considered. Nevertheless, this component would still account for a considerable proportion of an end-to-end run of F, and therefore effort was put into redesigning it. Improvements on this component of OldF cover two main areas:

- **Rewriting the comparison funtion:** to sort the tuples, which are represented as arrays in memory, it is necessary to define a custom comparison function which is used by the C++ standard library sort algorithms [35]. The data must be sorted following a partial order given by a depth-first traversal of the d-tree, which does not necessarily match the order of the attributes in the tuples. Therefore, a data structure that links between the order required for sorting and the order of the attributes in the tuples is needed; this data structure is passed as an input of the comparison function. OldF used a variable length *std::vector* [34] for this; the new implementation uses a slightly lighter heap-allocated array instead (obtained by a construct such as *new double[5]*). Arrays and vectors have almost equivalent performance, and many C++ developers will advocate using vectors because of their safer and easier to use implementation. Nevertheless, the comparison function is a performance-critical hotspot of the program and is called hundreds of millions of times while dealing with datasets such as US retailer. Bringing micro-optimisations by using simpler data structures in this function does impact the overall performance significantly. Furthermore, the loop used in the comparison function was rewritten, to improve branch prediction and get some extra performance enhancements.

- **Sorting in parallel:** the *std::sort* algorithm [35] is used to sort the tuples in OldF. This function benefits from a powerful and highly optimised implementation, and is used as is in many commercial systems. Nevertheless, the main drawback is that it only takes advantage of a single CPU core, which is inefficient on machines with many cores. GCC features a Parallel Mode [16], which provides parallel implementations of many of the algorithms in the C++ Standard Library. These implementations are based on the OpenMP API [20], and require relatively few changes in the code to be enabled. They are built into GCC and do not require any external libraries; they can be enabled by adding the *-fopenmp* compiler flag, and switching to a different header inclusion and a different namespace in the relevant files. Nevertheless, care has to be taken to automatically handle the case where another compiler is used, which is achieved by introducing pre-processor macros and a small wrapper around the sorting function. Instead of sorting by using a single core like in OldF, F is now able to take advantage of all the cores available thanks to this powerful parallel implementation of the sort algorithm.

The following table gives more insight on the performance improvements that were achieved:

		US retailer	LastFM	MovieLens
Sort time (ms)	OldF	97237	418	1085
	OldF + F-Flags	17258	49	212
	F	4106	12	51
Speedup	**F** vs. OldF	23.68×	34.83×	21.27×
	F vs. OldF + F-Flags	4.20×	4.08×	4.16×

Table 3: Performance after redesigning the sorting component

If OldF is compiled with F-Flags to exclusively measure the improvements brought by sorting in parallel and rewriting the comparison function, the speedup is over 4 times on all datasets.

3.4 Improvements on the core parts of F

The thesis now discusses some of the work that was achieved on the core parts of OldF, namely the cofactor and convergence calculations.

Simplification of the caching data structure

As previously mentioned, F features a cache system to avoid computing redundant aggregates repeatedly. The data structure to manage the cache in OldF initally featured

the following design:
$std :: map < int, std :: map < std :: vector < double >, RegressionAggregate* >>$

The integer key in the outermost map corresponds to the ID of the node in the d-tree. Each of these IDs in the map is paired with a value which is another map. This innermost map has keys formed by a vector of doubles; these doubles correspond to values in the database tables that follow a given path in the d-tree. Each time a regression aggregate needs to be calculated for a given ID in the d-tree and a path of values, the cache must first be probed; if found, the previously computed aggregate can be directly used, or computed and inserted into the cache otherwise.

Therefore the cache is repeatedly used throughout the cofactor calculations, and boosting its efficiency can significantly improve performance. Two observations can be made regarding the initial implementation just described:

- Having a nested map inside a map is heavy in terms of memory usage and computational cost when probing for a value. Work was done to simplify this structure and turn it into the following form:

  $std :: map < std :: vector < double >, RegressionAggregate* >$

  In other words, the node ID information is directly merged into the values path vector, by adding an extra node ID value at the end; conceptually this is equivalent to merging the keys used in each map into a single key containing all the information. This enhanced cache implementation required changes to the way the cache is dealt with, basically meaning the code fragments used to access and populate the cache were rewritten in OldF's core functions.

- Maps in C++ are data structures that internally rely on binary search trees [36]. Inserting and searching in such a structure has logarithmic time complexity. Instead, an *std :: undordered_map* [37] is now used, which is another C++ data structure that relies on hashes instead of trees. Inserting and searching has constant time complexity, and the C++ implementation of the *unordered_map* features better performance than that of the *map*. Some additional work was needed when switching the data structures, as there is no built-in hash function available for vectors of doubles, and keys can't be inserted in the *std :: undordered_map* without a hash. With the *map* type, keys are inserted in the internal tree following an ordering function, which is available for vectors of doubles in C++, so a similar problem did not occur. Therefore, a custom hash function for the keys of the *std :: undordered_map* had to be implemented. The function needs to have good properties to avoid different keys from hashing to the same value, and to achieve this the source of the Boost library's *hash_combine* function [42] was used as an inspiration. For each double value in the vector, a hash is generated by using the standard C++ hash function for individual doubles; it is combined by XOR-ing it with the hash computed so far on the previous elements in the vector and two bit shifting operations are performed. This approach leads to a robust yet fast hash function which is currently in use in F's cache system.

To sum up, after these two implementation changes, the cache used by F now has the following form:

$$std :: unordered\_map < std :: vector < double >, RegressionAggregate* >.$$

Extended guidance provided to the compiler

A variety of optimisations can be performed by analysing the program's execution flow and by determining the functions that are most critical to its performance. Many compilers such as GCC feature special function attributes to help make decisions based on such information [17]. In particular, the *hot* function attribute was used in critical parts of the code. This attribute indicates that a given function is a hotspot of the program and therefore the compiler tries to optimise it more aggressively. The function is also placed in a special subsection of the text section (ie. the memory section where the instructions of the program are stored at runtime), alongside other hot functions, for better locality. For instance the function *seekValue*, which is repeatedly used during F's cofactor calculation to find values in the database following a d-tree ordering and specified value bounds, is given the *hot* attribute. To declare a function as hot, *__attribute__((hot))* can simply be added in front of the function; the project provides a slightly more complex implementation based on the pre-processor to ensure compatibility with compilers that do not support these additional attributes. Other attributes such as *cold*, *pure* or *leaf* are also available [17], but not used in the current code basis.

Following a similar idea, several functions were inlined. The objective of inlining is to get rid of the overhead brought by a call to a repetitive and small function, by directly including its body at the point of call. This is for instance the case of the functions used in the convergence component of OldF. The general C++ pattern consists in declaring a function in a header file (with its name, parameters and return type), and defining it with its body in a separate implementation file. In order to inline a function, it is in general required that it be fully defined in the header file, to make it directly available in the translation unit where it is accessed. The keyword *inline*, which behaves like a hint to suggest that the compiler ought to inline the function, should also be added. After performing these changes on OldF's relevant functions, it was possible to verify that they had been successfully inlined by generating the corresponding assembly code and by seeing that the bodies of the functions had been inserted at their point of call.

Further improvements and partial conclusions

Various other improvements were performed in OldF's core functions, such as more efficient vector filling, numerous operations rewriting or redundant code elimination. Their exhaustive listing would be tedious and rather uninteresting. Nevertheless, the work that was done on memory management can be highlighted. Indeed, the way OldF handled memory was initially flawed, and huge memory leaks would occur; for instance with the MovieLens dataset, as many as 44058582 bytes of memory were leaked during each execution, and several memory locations were used without having been previously initialised, which could lead to undefined behaviour. Research was first done with the Valgrind tool [25], in order to understand why such memory issues would arise; it was then possible to correct these problems. The F system now runs cleanly, and no longer suffers from any leaks or operations on uninitialised memory.

Performance figures are now given, in order to measure the impact of the enhancements that are discussed in this section, namely the simplification of the caching data structure, the extended guidance provided to the compiler and the other improvements just mentioned. Here are the timings for the cofactor and convergence components:

		US retailer	LastFM	MovieLens
Cofactor time (ms)	OldF	15728	211	2122
	OldF + F-Flags	5072	84	611
	F	4720	78	459
Speedup	**F** vs. OldF	3.33×	2.71×	4.62×
	F vs. OldF + F-Flags	1.07×	1.08×	1.33×

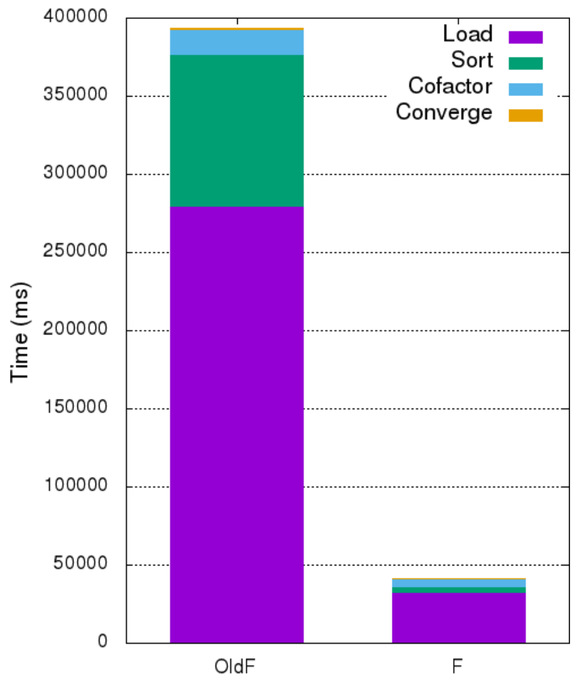
Table 4: Performance after improvements on the core parts of F (cofactor)

		US retailer	LastFM	MovieLens
Convergence time (ms)	OldF	317	44	13
	OldF + F-Flags	87	11	4
	F	75	10	3
Speedup	**F** vs. OldF	4.23×	4.40×	4.33×
	F vs. OldF + F-Flags	1.16×	1.10×	1.33×

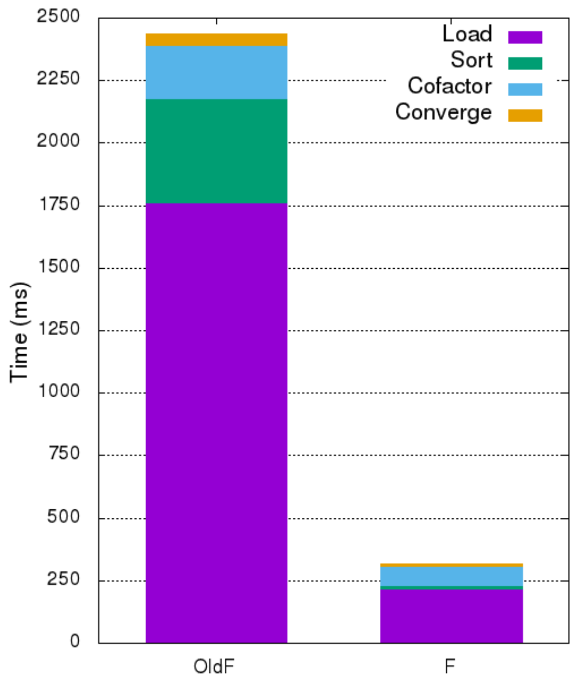
Table 5: Performance after improvements on the core parts of F (convergence)

The changes on the cofactor and convergence components of OldF were essentially improvements based on the existing code and not a complete redesign like the loading and sorting components. Therefore, the performance gains are less pronounced than previously demonstrated, yet still substantial. Several hundreds of milliseconds are still saved on cofactor calculation with datasets such as US retailer, and this showcases how important engineering considerations are.

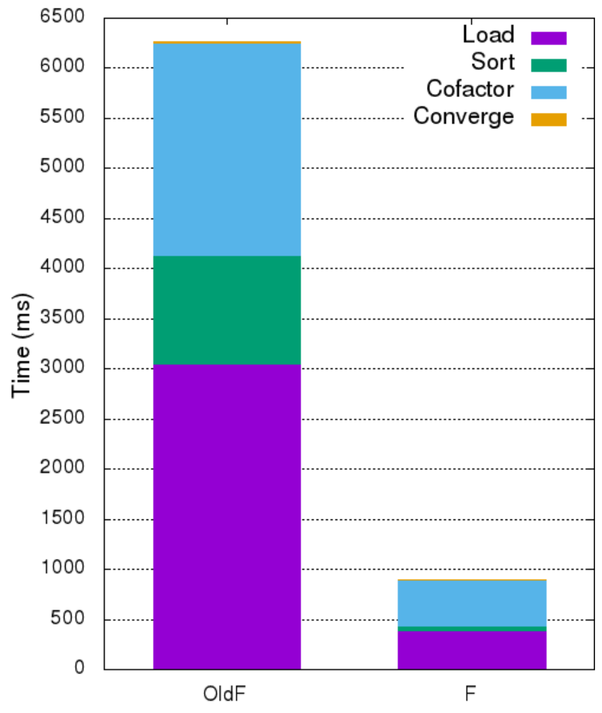
All the contributions discussed so far are put together, and the performance of the system is compared to the one reported in the SIGMOD paper [1]. As can be noticed in the below plots, depending on the dataset considered, the performance of an end-to-end run of F has been improved by up to an order of magnitude compared to OldF.



(a) US retailer: 9.53× end-to-end run speedup over OldF



(b) LastFM: 7.59× end-to-end run speedup over OldF



(c) MovieLens: 6.98× end-to-end run speedup over OldF

Figure 6: Performance by summing up contributions discussed so far

3.5 Loop unrolling

Loops bring overhead to the execution of a program, due to conditions checking, branching, and reduced optimisation possibilities for the compiler. Nevertheless, if the number of iterations of a loop is known at compile time, it is possible for the compiler to get rid of the loop entirely, therefore eliminating the overhead it brings and possibly increasing performance if the loop is called repeatedly.

To enable loop unrolling, new compiler flags were implemented into OldF. The variables covered by these flags are the number of tables in the database (flag *-DTABLES*), the number of workers in the network (flag *-DWORKERS*), the number of attributes in the database (flag *-DATTRIBUTES*) and the number of features (flag *-DFEATURES*) and interactions (flag *-DINTERACTIONS*) used for F's regression learning. For instance, *-DTABLES=3* can be added to the compiler flags when compiling F to run on the US retailer dataset.

When specified at compile time, the compiler replaces the variables previously mentioned as constants in the code, and eliminates the loops it can. These flags are optional, and if not specified, the values of the different variables will automatically be determined at runtime, based on information extracted from the configuration files. The project contains a header file *GlobalParams.hpp* where each variable is given a type; note that the name of the flag and the name of the variable do not need to be the same. If a variable has a corresponding flag specified at compile time, a pre-processor macro defines it as a C++ *const*; its value can be used by the compiler directly throughout the code. If a variable has no flag, a pre-processor macro declares it as a C++ *extern* variable, and the variable must then be defined and given a value in an implementation file at runtime. The compiler will not be able to make any assumptions about the value of the variable in this latter case.

To be a bit more specific about the effects of unrolling, consider the following code which is used in the distributed setting to determine whether a given tuple must be sent to a node or not (more details in Section 5):

Code Excerpt 2: Loop based on number of workers

```
/*
 * Check hash value inequality for each node.
 * If compiler flag available, enables loop unrolling and other
     compiler optimisations.
 */
for (size_t nodeID = 0; nodeID < NUM_OF_WORKERS; ++nodeID)
{
  if (_nodesHashAddresses[nodeID
       + NUM_OF_WORKERS * attributes[attr]] != hash)
  {
    ignoredWorkers[nodeID] = true;
  }
}
```

When compiling without additional compiler flags, the following assembly instructions are generated for this C++ code fragment:

Code Excerpt 3: Assembly without loop unrolling

```
1    mov   0x262b4f(%rip),%r8
2    test  %r8,%r8
3    je    .42a604
4    lea   0x0(,%r8,8),%rdi
5    xor   %r11d,%r11d
6    imul  %r10,%rdi
7    add   0xe8(%r13),%rdi
8    nopl  0x0(%rax)
9  .42a5f0:
10   cmp   (%rdi,%r11,8),%rdx
11   je    .42a5fc
12   movb  $0x1,0x0(%rbp,%r11,1)
13 .42a5fc:
14   inc   %r11
15   cmp   %r8,%r11
16   jne   .42a5f0
17 .42a604:
18   mov   0x8(%r14),%r10
```

In this case, the value of the NUM_OF_WORKERS variable is loaded by instruction 1. Instruction 2 tests whether the variable is equal to 0 and instruction 3 ignores completely the loop if it is. Instructions 4 to 8 prepare the registers for the comparison in the if clause inside the loop (lines 7 and 8 of the C++ code). Instructions 10, 11 and 12 respectively correspond to the if condition checking, branching when the condition is false, and writing true to the *ignoredWorkers* array otherwise. Other instructions relative to the loop can be observed, namely instructions 14, 15 and 16 situated under label .42a5fc, which respectively correspond to loop index incrementation, loop condition checking and branching backwards if the upper limit is not yet reached. If the loop has not yet ended, the processor jumps to label .42a5f0, and execute instructions 10, 11 and 12 again.

When compiling with the *-DWORKERS=4* compiler flag setting the value of NUM_OF_WORKERS to 4, the following assembly instructions are generated for this C++ code fragment:

Code Excerpt 4: Assembly with loop unrolling

```
1    mov   0xe8(%r12),%rdi
2    shl   $0x2,%r9
3    cmp   (%rdi,%r9,8),%rdx
4    je    .42a3a3
5    movb  $0x1,0x0(%r13)
6  .42a3a3:
7    lea   0x8(,%r9,8),%r10
8    cmp   (%rdi,%r10,1),%rdx
9    je    .42a3b6
10   movb  $0x1,0x1(%r13)
11 .42a3b6:
12   cmp   0x8(%rdi,%r10,1),%rdx
13   je    .42a3c2
14   movb  $0x1,0x2(%r13)
15 .42a3c2:
16   cmp   0x10(%rdi,%r10,1),%rdx
17   je    .42a3ce
18   movb  $0x1,0x3(%r13)
19 .42a3ce:
20   mov   0x8(%r14),%r11
```

In this second case, the loop has disappeared, and the NUM_OF_WORKERS variable is now a constant that is directly used by the compiler. Four groups of three equivalent instructions (instructions 3 to 5, 8 to 10, 12 to 14 and 16 to 18) follow one another; the three instructions respectively correspond to the if condition checking, branching when the condition is false, and writing true to the *ignoredWorkers* array otherwise. There are no longer any instructions relative to loop counters, loop upper bound checking, or branching if the limit is not yet reached. As the compiler knows that the loop will contain exactly four iterations no matter what, it can simply duplicate the body of the loop four times and get rid of any extra overhead.

Loop unrolling does seem like an interesting way to improve the performance of a loop, but from a different perspective, the examples above can lead to a different conclusion. Except for loops with a small body and a low number of iterations, loop unrolling tends to increase the size of the binary by duplicating instructions. Even though the overhead brought by the loop is removed, as the number of assembly instructions grows for a same C++ code fragment, the CPU's instruction cache is filled up faster, negatively affecting performance. Therefore loop unrolling has both benefits and drawbacks, and may or may not increase performance in different cases.

The following effects were observed when using loop unrolling across the different components of F:

- **Load:** loop unrolling negatively affects this component. The functions used to load are parse the CSV data yield an underlying very lengthy assembly code, and therefore loop unrolling significantly increases the size of the binary and decreases the performance of this component.

- **Sort:** there is no noticeable change in performance brought to this component.
- **Cofactor:** loop unrolling improves the performance of this component. The functions related to cofactor computation contain numerous loops with relatively constrained bodies. For instance, on the US retailer dataset, this performance boost accounts for about 60ms comparatively to the 4720ms reported in Table 5.
- **Convergence:** loop unrolling has a slight negative effect on this component. Most of the functions used for this component are inlined, and as stated previously inlining is mostly beneficial for small functions. Therefore by increasing the number of instructions contained in the functions, the effects of inlining are also hit by loop unrolling.

In the upcoming experiments throughout the thesis, loop unrolling is only used when explicitly stated.

3.6 PGO

All the previous work done of OldF was focused on enhancements brought by improving the code or by tuning the compiler in a specific way. A new area will now be explored, namely Profile-Guided Optimisation (PGO). Contrarily to the previous work, PGO is a dynamic form of optimisation that requires running the program in order to operate.

PGO makes the program generate extra profiling information at runtime, such as various probabilities and frequencies of function executions or branching paths. This information is then used by the compiler to improve the generated code on aspects such as branch prediction, function reordering, loop behaviour or function inlining. The unusual requirement for this approach is to have datasets that are representative of a program's execution, and unfortunately these are not always available or easy to obtain. In F's case, the representativeness of the datasets was checked by generating the profiling information with a given dataset, and using this information to execute an optimised program on a different dataset; these tests all gave similar performance speedups when switching datasets, which indicates that the three datasets in use in this project are representative of what could be defined as a standard execution flow of the F program.

The GCC compiler has built-in functionality to use PGO. The official documentation is very scarce on this topic, nevertheless some extra insight can be gained by referring to Drepper's publication about memory and processors [13]. The following steps must be observed when using PGO:

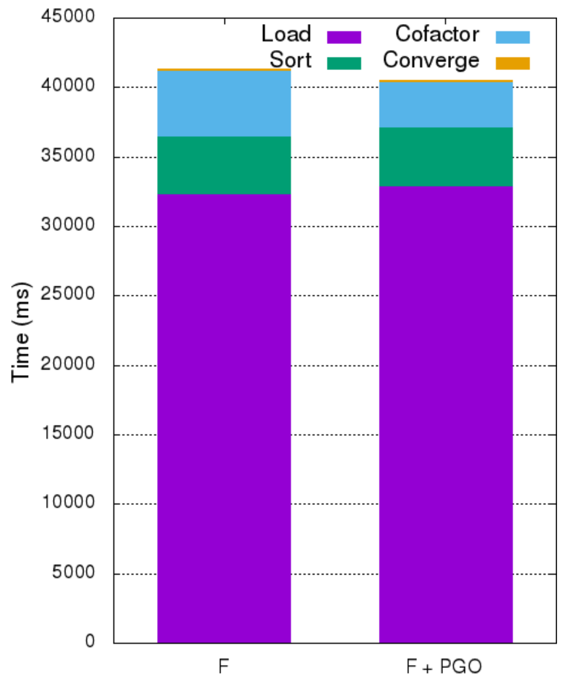
- Compile the program a first time by adding the *-fprofile-generate* flag to the usual compiler options.

- Let the program run several times on data that is considered as sufficiently representative of a program's execution flow. The execution time in this step is significantly slower as the code is also generating profile information and saving it into *.gcda* binary files.

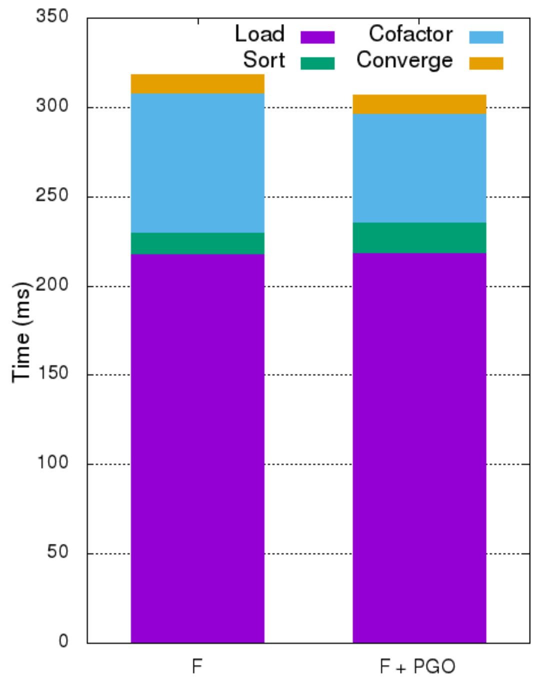
- Recompile the program with *-fprofile-use*. The compiler reads the information contained in the *.gcda* files, and uses it to improve the code it generates on various aspects previously mentioned. If the application is multi-threaded, as it is the case for MF (multi-threaded version of F), the *-fprofile-correction* flag must also be added. Indeed, the profile information generated by multiple threads may become inconsistent due to missed concurrent updates, and has to be automatically corrected by the compiler.

It is important to highlight the fact that a given batch of profiling information cannot be reused if the code has been changed in the meantime. Indeed, if new functions or branches are added, there are inconsistencies between the code and the corresponding profiling data, and the compiler fails to compile the program with *-fprofile-use*. Therefore it is best to only use PGO when having a stable code, and not when testing on a daily basis.

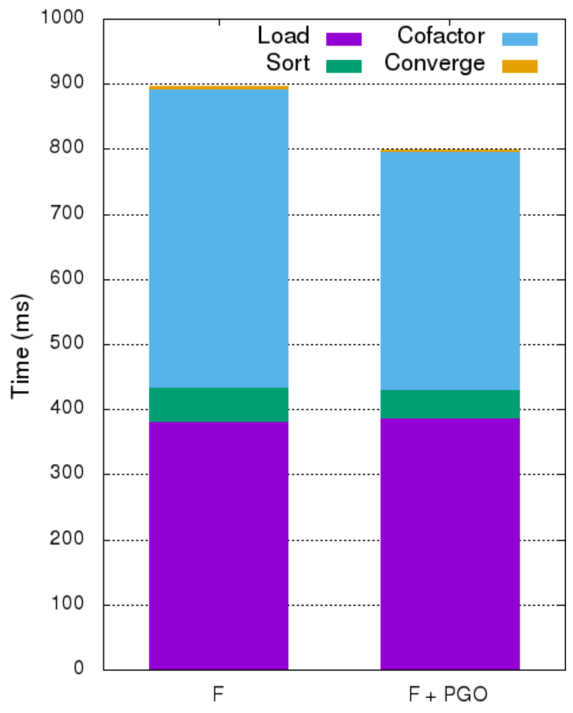
After compiling F for PGO, generating profile information by running the different datasets, and recompiling the program with corrections enabled, the following improvements are achieved:



(a) US retailer: 0.98× load, 0.97× sort, 1.45× cofactor, 0.86× converge



(b) LastFM: 0.99× load, 0.71× sort, 1.29× cofactor, 1.00× converge



(c) MovieLens: 0.99× load, 1.19× sort, 1.25× cofactor, 1.50× converge

Figure 7: Performance with or without PGO

The following facts can be observed for each component of F:

- **Loading:** this component seems marginally slower with PGO. This is mainly due to the fact that this component handles I/O operations, which is difficult for the compiler to profile in an efficient way.
- **Sorting:** the results are mixed for this component. The performance for the US retailer dataset is roughly the same, is improved for the MovieLens dataset, but is negatively affected for the LastFM dataset. The function used for sorting is based on a highly optimised algorithm and multi-threaded implementation, efficient profiling may be more difficult depending on the dataset.
- **Cofactor:** PGO gives an impressive performance boost for this component. Up to one and a half seconds of computation are saved when running the US retailer dataset. A lot of branching, functions calls and loops are used in the corresponding code basis, which enables to generate extensive profiling information.
- **Convergence:** as for sorting, results are mixed and depend on the dataset. The code basis is rather small for this component, and the execution is fast, limiting the effect of profiling.

Overall, the times for an end-to-end run of F are significantly improved as shown in the plots. As a more general conclusion, PGO is a simple to use yet powerful tool, that challenges the programmer into building executions that are as much as possible representative of his program's flow.

In the upcoming experiments throughout the thesis, PGO is only used when explicitely stated.

3.7 Three modes of building F

Finally the different build modes that are implemented around this project are presented. This section is not directly related to performance, but is more aimed at introducing an easier and more professional developing experience compared to what was in use in OldF.

Three build modes were implemented:

- The **Debug** build mode. All optimisation flags are disabled for faster compiling and better debugging experience. Symbol information is kept when compiling the code, meaning that the names of the variables and functions are preserved and can easily be linked to the program's source code. Extra messages are logged when running the program.

- The **Release** build mode. The F-Flags presented in Section 3.1 are activated. Logging is kept to a minimum to provide the most relevant information to the user.
- The **Benchmark** build mode. The F-Flags presented in Section 3.1 are activated. All standard logging messages are disabled, but instead performance timings are reported.

These modes can be simply selected by editing the *CMAKE_BUILD_TYPE* variable in the *CMakeLists.txt* file of the project. With this simple switch, it is possible to quickly enable extra logging information or run benchmark tests without modifying any of the C++ code. Furthermore, the functions used in one build mode do not affect the other modes. For instance, the initialisation of timers and the benchmark logging function calls are not even compiled when using Release mode. This is handled by pre-processor macros which automatically select code fragments relevant to the current build mode at compile time. Therefore there is no overhead in adding extensive debugging information or benchmarking checkpoints in the code when using Release mode. To avoid having to systematically write the pre-processor macros when adding new logging information, the C++ standard logging functions are packaged into simple wrappers, which are either eliminated or kept by the pre-processor. For instance, simply writing *DINFO("Hello World!")* in the code logs the corresponding text in the console in Debug build, and is ignored at compile time when using the two other build modes.

In the upcoming experiments throughout the thesis, the performance figures are always related to code compiled in Benchmark mode.

4 MF: Multi-threading F

When learning regression models over factorised joins, OldF only exploited one CPU core due to its single-threaded design. This limits its scalability and results in a lot of unused computational power on a machine with many CPU cores. In Section 3 it was demonstrated how to multi-thread the sorting component of F; the thesis now focuses on a more complex task, introducing multi-core processing to the cofactor calculation component. Novel insights about how to implement efficient parallel processing in factorised settings are introduced. From now on, the multi-threaded version of F is specifically referred to as ***MF***.

This section first discusses how to implement multi-threading to guarantee the correctness of the final result. Efficiency concerns are then discussed, focusing on which table to partition and how to deal with the caching system. Implementation details are provided and an experimental analysis is finally given.

4.1 Correctness

To be able to multi-thread the cofactor computation, a disjoint way of partitioning the total workload must be found. The simplest and most natural approach to do this is to run F in parallel on different partitions of the data, and merge the results obtained on each one of these partitions. Nevertheless, even though data partitioning is a general approach to multi-threading tasks, it cannot be done arbitrarily, and care must be taken to ensure it does not lead F to compute incorrect results.

This approach is based on a simple observation regarding joins. Consider tables A and B, as well as A_1 and A_2 two partitions of A such that $A = A_1 \cup A_2$. Due to the distributivity of the join operator, the following equality is verified:
$A \bowtie B = (A_1 \bowtie B) \cup (A_2 \bowtie B)$.

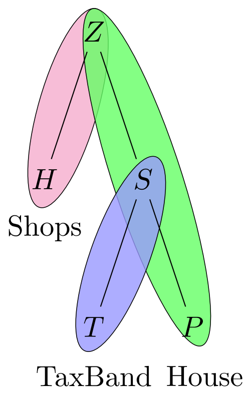
All the aggregates computed by F are based on the join of a table column and another one. As mentioned previously, cofactor computation commutes with relational union and projection, and cofactors can therefore be computed on each partition independently. Based on the above observation about joins, if one of the database tables is selected and expressed as a union of disjoint table partitions, it is possible to learn the linear regression model over each sub-join in the union, and merge all the partial results.

Due to the distributivity property, the final cofactors are obtained by summing the different cofactors for each disjoint partition of the entire dataset. The convergence step can then be performed as usual, independently of the cofactor step.

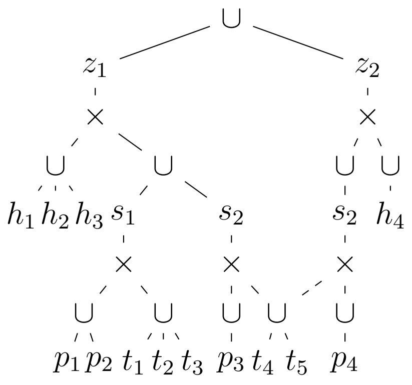
This horizontal partitioning scheme, which assigns equivalent amounts of tuples of a table to each thread, gives an efficient way to divide the workload of F's cofactor component between several threads, while ensuring correctness of the overall computation.

4.2 Table to partition

After some preliminary testing, it was noticed that there are great performance gaps depending on the table selected for the partitioning. The following section demonstrates why this happens, as well as refinements of the horizontal partitioning to deal with this new issue. Going back to the example presented in Section 2, its d-tree and join factorisation are recalled:



(a) D-tree



(b) Factorisation

Figure 8: D-tree example and factorisation of the join result

As previously mentioned, F's caching system in this example relies on the fact that each value s_2 is paired with the values $t_4 \cup t_5$, which can be stored a single time and reused for every occurrence of s_2. The corresponding aggregate only needs to be computed once, and can be reused each time the factorisation fragment is encountered again. Consider the following scenarios:

- If the partitioning were to be done on table *TaxBand* which contains attribute T and if values t_4 and t_5 ended up in two separate partitions, this would have the side-effect of "breaking up" the aggregate; an aggregate corresponding to $t_4 \cup t_5$ would no longer be cached and reused in the calculations. If such a table is partitioned, smaller and more numerous aggregates are cached, which reduces the efficiency of the system.

- On the contrary, if partitioning is done on a table that joins at the root of the tree, namely table *Shops* or table *House*, the cached aggregate corresponding to

$t_4 \cup t_5$ is preserved. By selecting one of these two tables instead of *TaxBand*, the issue no longer occurs in this example.

By analysing this simple illustration, a new issue that arises when dealing with factorisations is highlighted and choosing the wrong table can thus negatively affect the performance of the multi-threaded system. More precisely, partitioning a table that joins at a lower level in the d-tree breaks up aggregates at lower levels in the join factorisation, and therefore negatively affects more branches throughout the factorisation. One of the tables joining at the root of the d-tree must therefore be partitioned in order to limit the effects of aggregate fragmentation.

The issue of choosing to partition between table *Shops* or table *House* remains, in other words selecting the table between all the ones joining at the top of the d-tree. Without having any extensive profile information about the data itself, it is hard to predict which table is most negatively affected by partitioning based only on theoretical considerations. This second selection step is therefore built on experimental observations.

Choosing a table joining at a lower level in the tree would hit performance badly as stated previously. Nevertheless, small performance differences still occur when choosing different tables joining at the root of the d-tree. The following facts were noticed when running benchmark tests:

- Partitioning the largest table joining at the root of the d-tree yields better performance.
- The performance gap between two root tables is more pronounced if the size gap between them is larger.

This behaviour was observed for the US retailer, LastFM and MovieLens datasets, plus an additional Housing dataset not used for benchmarking in this thesis.

The following interpretation can be proposed for this behaviour. For simplicity we consider a scenario where the database contains only two tables that join at the root of the d-tree. If the larger table is partitioned, the threads operate on their own partition of this table, but on the full range of the other table. Therefore all the tuples in the smaller table are used by all threads. If the table is small enough, the CPU may be able to cache it completely or at least in part in its L3 level cache, which is shared by all CPU cores. This benefits all threads, as they are able to get data from the L3 CPU cache instead of receiving it from main memory. On the contrary, if the smaller table is partitioned and the larger one is used entirely by all threads, it may only be possible for the CPU to cache small chunks of the large table. In the case where the threads are operating on different sections of this table concurrently, they are not able to benefit from the fact that they all need the same memory chunks for their processing. Therefore the performance is superior when partitioning the larger table. More insight about CPU caches and prefetching effects can be gained by referring to Drepper's publication about memory [13].

To sum up, MF assigns the table to partition following a selection procedure based both on theoretical and practical considerations, and proposes a novel approach to multi-threading in factorised settings. MF divides the workload between several threads by partitioning one of the database tables horizontally, and initially considers all the tables joining at the root of the d-tree as potential candidates for partitioning. Among all these candidates, the largest one is selected and threads are launched to compute cofactors on disjoint partitions of this table. This heuristic turned out to be optimal for all the datasets presented in this thesis plus an additional *Housing* dataset, but may be refined when exploring more complex d-trees and analysing their implications on the caching system used by MF.

4.3 Minimising redundancies across threads

In the initial implementation of multi-threading, independent threads were launched to compute the cofactors when partitioning a table; these threads would not cooperate in any way and would have their own local instance of the caching data structure. This approach had the side effect of increasing the overall workload needed for cofactor computation. Indeed, this meant that an aggregate computed by a given thread had to be computed again by other threads who encountered the same factorisation fragment, which caused redundancies in the overall computation and reduced the efficiency of the caching system.

In a second approach, it was decided to implement a unique cache that would be shared between the different threads. Although this required to add some fine-grained synchronisation points on the cache data structure to avoid any race conditions, this enabled to reduce the overall workload by avoiding many redundant computations. If a thread computes a specific aggregate and if another thread later on needs to compute the same aggregate, it is able to benefit from the work of the first thread by simply probing the shared cache and retrieving the aggregate that was inserted earlier.

There is nevertheless a caveat with this approach. Indeed, if two threads encounter the same new aggregate at the same time, they compute it in parallel, and the first thread that finishes successfully inserts its aggregate in the cache. The insert by the second thread fails as an aggregate with the same key already exists in the cache, and when detecting this insertion failure, the second thread simply discards its redundant aggregate and moves on with its processing. Therefore there still can be redundancies in the computation. With the current multi-threading scheme based on horizontal table partitioning, this cannot be avoided without increasing the amount of locking around the cache data structure or significantly modifying the way F deals with cofactor computation.

The extent to which this scenario occurs in a real setting is measured quantitatively. Each dataset is tested ten times with eight threads operating in parallel, and the number of redundant aggregate computations as previously described are averaged on these executions. Indeed, depending on the relative speed of the threads during a given execution, different amounts of redundancies are likely to happen. For the US retailer dataset, 0 redundant aggregates are computed, for a total of 1276 aggregates in the cache. Relatively to the huge size of this dataset, the number of aggregates is small, making it unlikely for two threads to compute the same aggregate in parallel. For the LastFM dataset, on average 6.8 redundant aggregates are computed, for a total of 1892 aggregates in the cache. For the MovieLens dataset, on average 4.2 redundant aggregates are computed, for a total of 6040 aggregates in the cache.

Overall, it can be argued that the amount of redundant calculations is very low with the approach suggested by this thesis, especially regarding the fact that a given aggregate may be reused a vast amount of times throughout the cofactor calculation and that a redundant computation on that aggregate can only occur once after initial insertion in the cache.

4.4 Implementation

Additional implementation details are now discussed, in order to better understand how multi-threading is done in practice and to show its impact on the cofactor computation's performance.

It is important to underline the fact that the partitioning of the tables is done logically and not physically. Therefore no new structures are created when partitioning, only lower and upper bounds are passed to MF's cofactor function in order for it to know on which range of the partitioned table it must operate. This is possible due to the fact that F does not modify the tables it works on, except for the sorting that is done beforehand; thus several threads can safely access the same table structures concurrently without any synchronisation. Operating with logical partitions also ensures better memory locality of the database table, which leads to better performance.

The table to partition is selected programatically following the previously presented algorithm, but the user can use command line options to chose the number of threads and the number of partitions to work on; if no command line options are specified, this is also determined programatically based on the processor's architecture. As far as the number of partitions is concerned, two aspects must be considered:

- A given partition may take more time to process than another, and even if the workloads of all the partitions are similar, a thread may take longer to complete its task if the OS for instance decides to deschedule it during the calculation in favour of a system or another user process. This leads to discrepancies in the time taken to process each partition. By increasing the number of partitions, these differences tend to be erased, as a given thread can start work on a new partition if another one is slower and is still processing a previous partition. If the partitions are small enough, this prevents some threads from being idle because no more partitions are available but some other threads haven't yet finished their work. In other words, by increasing the number of partitions, all the threads can be kept busy during the entire processing, which leads to a uniform distribution of the processing time among all the available threads.

- On the other hand, by increasing the number of partitions, the caching may become less efficient depending on the structure of the d-tree due to aggregate fragmentation; therefore datasets that heavily rely on caching may be negatively affected by a higher number of partitions. There also is a very small initialisation overhead when launching work on a new partition, and the data structures that are used to store the subresults of each partition also grow in size as the number of partitions grows. These effects are negligible when dealing with a few partitions, but may become visible when increasing their number significantly.

These two competing effects can be observed in the upcoming experimental section, where performance is plotted for a range of different numbers of partitions for each dataset.

Finally, focus is put on how the partition allocating scheme is implemented in practice. Allocation must be done dynamically, as predefining a set of partitions for each thread to work on would be inefficient. Indeed, if one of the threads turns out to be faster than others, it must be able to work on additional partitions as was discussed previously. The solution adopted is to create a shared counter keeping track of the index of the next partition to work on. This index is used to determine the partitioned table bounds that MF must observe to compute the cofactors of this partition. The partitioned table is divided into chunks of equal size; for instance, when using two partitions on a table containing one thousand tuples, partition with index 0 corresponds to tuples 0 to 499 and partition with index 1 corresponds to tuples 500 to 999. In the case where the number of tuples is not divisible by the number of partitions, the remainder tuples are allocated to the last partition. To avoid synchronisation overhead required to access the shared counter, a C++ atomic integer is used (more precisely the *atomic_uint_fast32_t* type [38], which guarantees an unsigned integer of at least 32 bits long), called *_nextPartition*. For instance, to retrieve and increment such a variable, the following construct can be used:

Code Excerpt 5: Incrementing partition index

```
/* Retrieve and increase value of next partition to work on. */
currentPartition = _nextPartition.fetch_add(1, memory_order_relaxed);
```

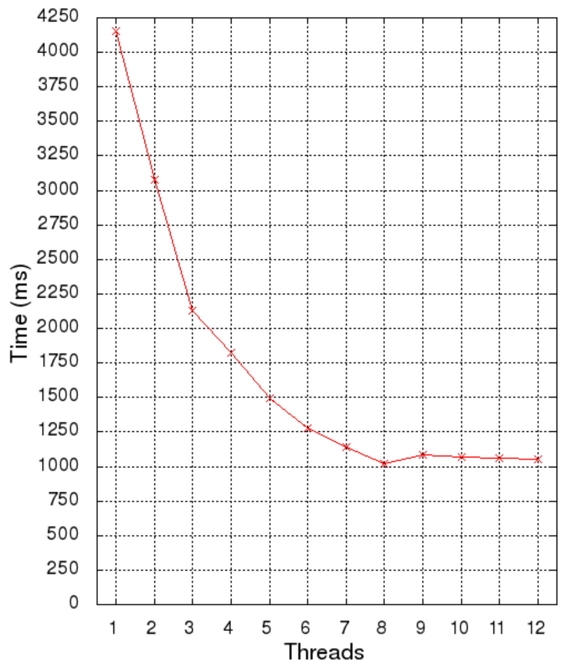
The atomic classes ensure that concurrent operations on the integer variable are well defined; for instance, if a thread increments the counter, this is translated by a single instruction at the CPU level, and all other threads immediately see the effect of this operation, even if they previously had an outdated version of the counter in their respective L1 or L2 CPU caches. The *memory_order_relaxed* used in the above code snippet is a constant from the *std::memory_order* enum [39], and is used to indicate how the compiler is allowed to rearrange other instructions around an atomic operation; the details of why this specific memory order is chosen are not considered in this thesis as this would be a fairly long and complex discussion. This way, partitions are allocated to each thread smoothly without any locking, and each thread knows what its next task is by simply referring to this shared counter and by using it to determine the bounds of its partition. Once the counter has reached the total number of partitions specified by the user when launching MF, processing can stop. All the cofactors computed on each partition are then summed up and the convergence component is called in the main thread of execution.

4.5 Experiments

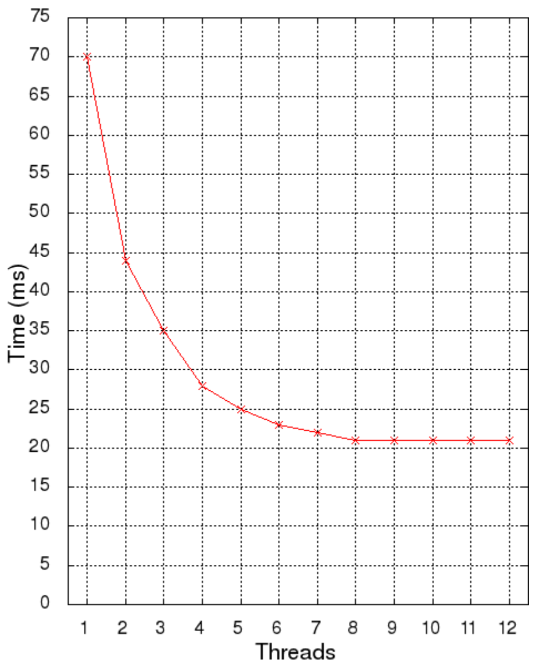
A series of experiments is conducted to measure the impact of multi-threading and to illustrate some of the ideas previously discussed.

In parallel of this project, Maximilian Schleich continued on developing F. A more efficient and more general version of cofactor computation was introduced, and it is the one on top of which MF and DF are implemented. F is now able to perform additional machine learning tasks such as factorisation machines or polynomial regression, and the way cofactors were computed until now was improved. These developments are orthogonal to this thesis and mostly concern the performance of regression models of higher degrees, but explain why the performance figures for single-threaded MF cofactor computation are superior by about 10% to the ones reported in Section 3 for the current setting.

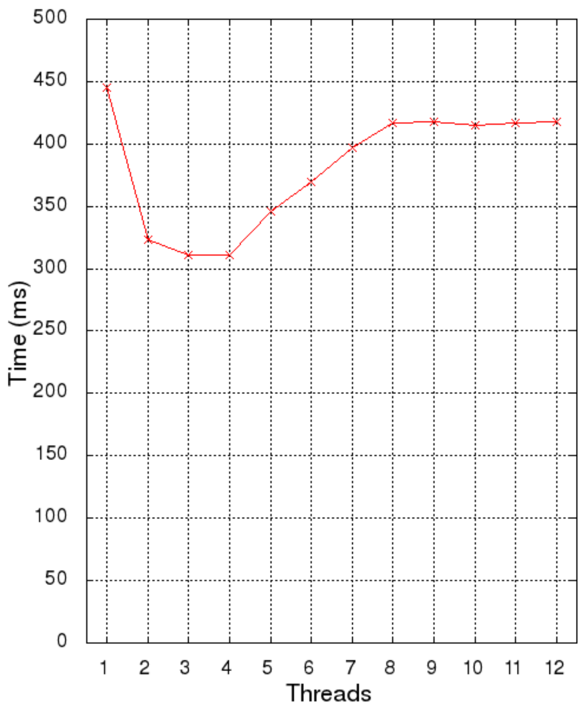
The first group of plots shows the evolution of the cofactor computation performance depending on the number of threads, with one single partition per thread. The second group shows the evolution of the cofactor computation performance depending on the number of partitions used while keeping the number of threads constantly equal to eight, which is the total number of cores available on the machine used for testing in this section, as detailed in Appendix C. There is one plot for each of the three datasets described in Appendix B.



(a) US retailer

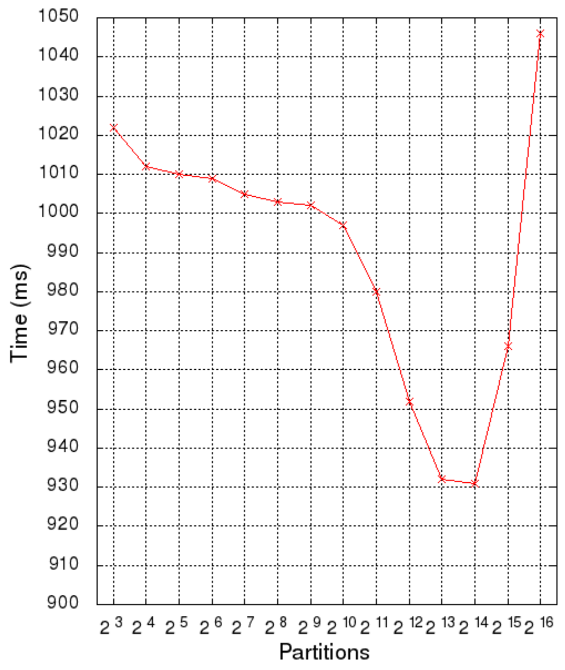


(b) LastFM

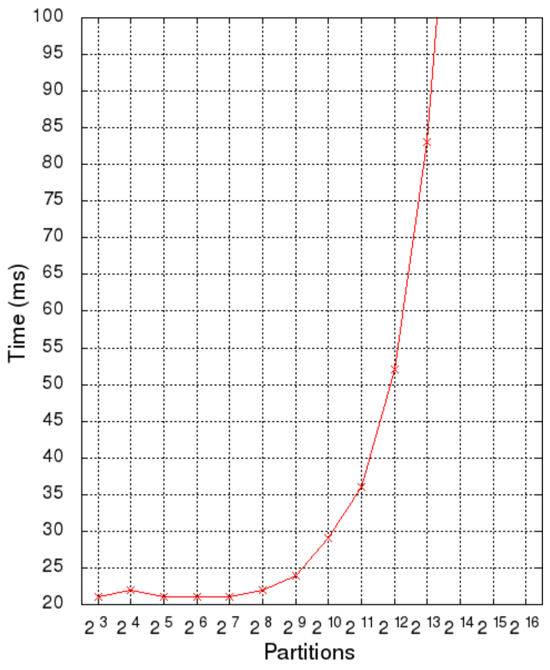


(c) MovieLens

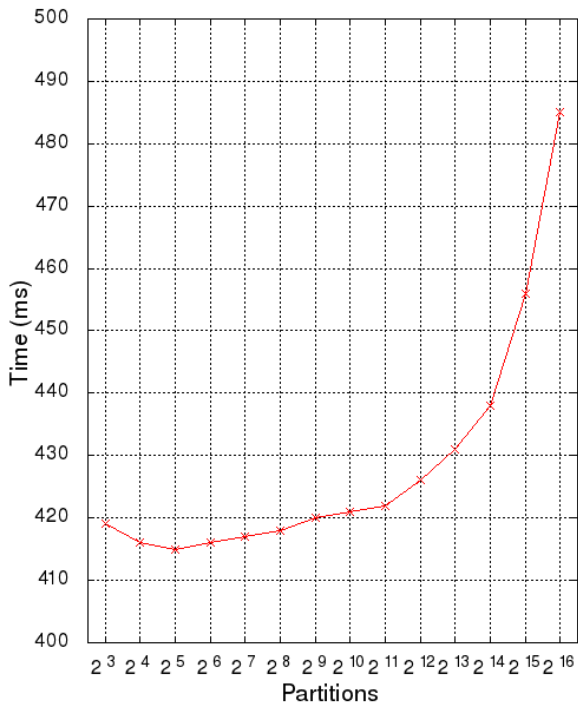
Figure 9: Performance of cofactor calculation depending on number of threads (machine with 8 cores)



(a) US retailer



(b) LastFM



(c) MovieLens

Figure 10: Performance of cofactor calculation depending on number of partitions

- **Influence of the number of threads:**

  Regarding the US retailer and LastFM datasets, best performance is reached when all the cores of the machine are used, namely for eight threads. When using more threads, the performance stagnates, as the processor's cores have to share their processing time between several of the program's threads and all the threads can therefore no longer be scheduled at the same time. When increasing significantly further the number of threads, although not shown in the plots, performance starts plummeting due to heavy context switching, in other words the CPU having to repeatedly schedule and deschedule numerous different threads.

  When using eight threads instead of one, cofactor computation is $4.07\times$ faster with the US retailer, and $3.33\times$ faster with the LastFM dataset.

  As far as the MovieLens dataset is concerned, performance reaches a peak for four threads, and then decreases slowly up to eight threads where is stagnates. It is unclear why this happens, as MovieLens is the "middle" dataset in terms of number of tuples, join size and factorisation size compared to LastFm and US retailer. This unexpected behaviour is currently still under investigation in order to understand the reasons why this dataset does not scale as well when multi-threading.

- **Influence of the number of partitions:**

  The amount of caching that occurs with the US retailer dataset is reduced with regard to its size, and therefore increasing the number of partitions does not affect the efficiency of the caching system, but instead tends to eliminate the discrepancies encountered between the workloads of the different partitions, as discussed previously. Therefore performance improves when increasing the number of partitions, up to a certain point where the overhead of creating more partitions takes over. The minimum is reached for 2^{14} partitions, where the performance boost is of about 10% compared to the one partition per thread setting.

  The MovieLens dataset relies more heavily on caching; increasing slightly the number of partitions brings a peak in performance at 32 partitions, but beyond this point the negative effect on the caching system prevails.

  The LastFM dataset relies the most heavily on caching relatively to its size, having one partition per thread is the best configuration in this case; performance is badly hit when using a high number of partitions, with cofactor computation taking almost an order of magnitude more when using 2^{14} partitions.

5 DF: Distributing F

Multi-threading F improved its scalability, and enabled to boost its efficiency on machines with several cores. Nevertheless this is still not enough for very heavy calculations or datasets that are too big to fit in the memory of a single machine. In this section, distribution is explored and a novel approach to learning regression models on a cluster of machines is introduced. This project exploits the state of the art Hypercube algorithm [5] and shows how the cofactor calculation can be dispatched on several nodes, and the results merged back to perform the final convergence step in a centralised setting. From now on, the distributed version of F is specifically referred to as ***DF***; multi-threading is not enabled in any of the experiments of this section.

This section first introduces the Hypercube algorithm with a general approach and then with an example. The entire distribution framework is then presented, explaining the interactions and the roles of the different nodes in the network. Implementation details are provided and an experimental analysis is finally given.

5.1 Hypercube algorithm

Building on some work by Afrati and Ullman [4], Beame et al. have recently designed the Hypercube algorithm [5] [6], which is a state-of-the-art data distribution policy that ensures correctness of join computation, where the join result is the disjoint union of local results computed at each node in the network. Its objectives are to aim at minimising the amount of data that is distributed across the network in a single communication round setting whilst being resistant to data skew. A brief overview of how it works is given in the upcoming paragraphs.

Consider a network composed of n distinct machines and a database query with k join attributes. Emphasis is put on the fact that k is not the total number of attributes in the database, as non join attributes are not accounted for. n is expressed as a product of k factors:
$n = d_1 * ... * d_k$

Each factor d_i represents the size of the dimension for the i^{th} join attribute. The n servers are arranged in a k-dimensional space, and each one of them is given a unique address $(x_1, ..., x_k)$ where each coordinate x_i is strictly inferior to the i^{th} dimension's size d_i. In other words, x_i must be in the integer interval $[\![0; d_i - 1]\!]$.

Now consider k hash functions $h_1, ..., h_k$. To avoid skew, distinct hash functions are used, or more precisely similar hash functions with different initial seeds. A given hash function h_i must hash database values and output integers in the interval $[\![0; d_i - 1]\!]$.

For a given tuple, each of its different values must be considered one by one. If a value corresponds to an attribute i used in the join query, it is hashed with the associated hash function h_i. If a value does not match a join attribute, it is simply ignored and will not influence the shuffling. By doing so for all values in the tuple, a coordinate $(h_1(a), *, h_3(b), ..., h_k(z))$ can be created by inserting the tuple's hashed values at the right indexes.

This coordinate is a mixing of hash values and stars *. The * notation indicates that the tuple does not contain the corresponding join attribute, in other words the current database table from which the tuple is extracted does not contain all the join attributes used in the query. The * acts like a "universal" hash value, taking any value strictly inferior to the dimension for that join attribute. For a join attribute i, * covers the whole $[\![0; d_i - 1]\!]$ integer interval. By referring to this coordinate, or to these several possible coordinates if the * notation is used, the tuple is dispatched to all the servers which were given a matching coordinate in the Hypercube. The process is repeated with all the tuples in the database to complete the Hypercube shuffling.

Intuitively, the * notation is used to ensure that all the nodes have all the tuples needed to evaluate the join; even if a tuple is missing some of the join attributes, a node may need the tuple to join on the other attributes it contains. The Hypercube algorithm ensures the correctness of the overall parallel query processing. This result is not detailed in this thesis, but the literature [5] can be referred to for a more rigorous treatment. To obtain the full result of the join query, the union of all the sub-joins computed on each machine in the cluster is performed.

A simple example based on the US retailer dataset is now presented. There are two join attributes in the corresponding query, *locn* and *zip*. A two-dimensional Hypercube is therefore considered, with the first dimension corresponding to *locn* and the second dimension corresponding to *zip*. Suppose there are four servers ($n = 4$). A size of two for each dimension is chosen:
$n = d_1 * d_2$, where $d_1 = d_2 = 2$

The four servers are arranged in a virtual two-dimensional space as follows, where each vertex represents a server, with an arbitrarily chosen number to identify it and a coordinate as defined previously:

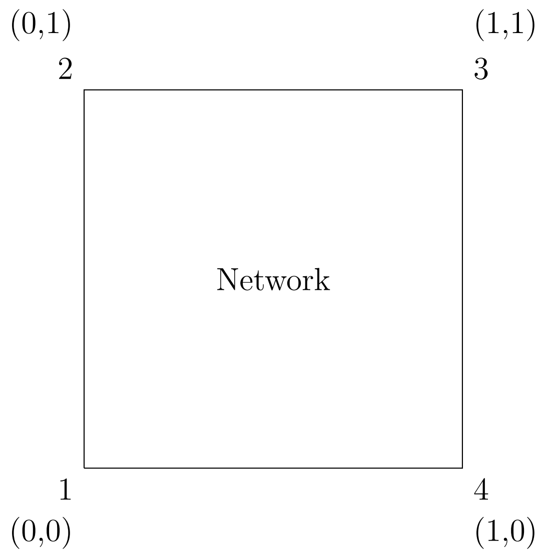


Figure 11: Hypercube network example

Now consider two tuples, one from table *Inventory* with a value a for attribute *locn* and the other from table *Location* with values b and c for attributes *locn* and *zip*. All the other values contained in the two tuples are ignored, as they belong to non join attributes.

The first tuple corresponds to coordinate $(h_1(a), *)$. The * is used as table *Inventory* does not contain the join attribute *zip*. Suppose that $h_1(a) = 0$. Due to the *, two coordinates are possible, (0,0) and (0,1). According to the previous diagram, the tuple must be dispatched to servers number 1 and 2.

The second tuple corresponds to coordinate $(h_1(b), h_2(c))$. Suppose that $h_1(b) = 1$ and that $h_2(c) = 0$. This leads to a single coordinate, (1,0). According to the previous diagram, the tuple must be dispatched to server number 4.

Following a reasoning similar to the one discussed in Section 4, this shuffling policy can be used for aggregates on top of joins and enables F to learn regression models in a distributed environment.

5.2 Interactions within the network

Once the theoretical background of the Hypercube algorithm was understood, one of the objectives of the project was to build an efficient and highly configurable implementation of the algorithm, as well as a more general framework for the nodes to be able to communicate and synchronise with one another.

In order to to achieve this implementation, two external C++ libraries are needed:

- The Boost Iostreams library [22]: this library is used to communicate between several machines over a network by means of input and output streams.
- The MurmurHash library [24]: this library is used to provide efficient hashing functions to shuffle the tuples across the network.

With the current implementation, the network can be formed of an arbitrary number of nodes, as long as the requirements of the Hypercube algorithm are followed regarding the dimensions of the join attributes. All the nodes that take care of cofactor calculation are called worker nodes; they load their own local partition of the data, shuffle it, receive the data from other worker nodes, sort the received data, learn the regression model over the factorised join and send the cofactors to the master node. The master node does not deal with any data and does not compute any cofactors; it takes care of synchronising the different worker nodes and performs the final convergence calculation once all the cofactors are gathered from the worker nodes. As discussed in Section 4 on multi-threading, cofactor computation commutes with relational union and projection, and the cofactors on each node can be computed in order to obtain the final cofactors by summing all the different partial results that are received.

It is possible and recommended to place the master and a worker node on a same physical machine by assigning them to a different communication port. Indeed, the master node mainly sends short synchronisation messages to the workers throughout the program's execution, and its only CPU intensive task is the final convergence component of F. Convergence computation is done once all the cofactors have been calculated and therefore after the workers have shut down. Thus master and workers do not perform any heavy processing concurrently, and placing the master on a different machine is a loss of computational power during all but the convergence component.

In order to use the program in distributed mode, the user must modify two configuration files: *network.conf* where are specified the IP addresses of the different nodes alongside their communication ports; and *hypercube.conf* where all the attributes are listed with the size of their dimension; non join attributes must have size 0. These configuration files feature a simple layout and also support comments. Once launched on each node starting with the master node, the program takes care of the rest of the processing and coordination automatically.

The distribution process follows four main steps:

- **The handshake step:** each node in the network sends a handshake message to the master node. This allows each node to initialise and wait for all the other nodes to be ready to run.
- **The connection step:** the *DataWriter* and *DataReader* components of each worker are initialised and connect to their counterparts in all the other worker nodes. They are respectively used to send tuples to other worker nodes and to

receive data from other worker nodes. Each worker node then synchronises with the master node.

- **The query step:** the tuples are shuffled across the network and the cofactors are computed on each worker node. This is a single round computation, in the sense that all the data is shuffled at the beginning and the whole learning process is then done; there is no intertwining between shuffling and partial computations, as it would be the case in a multi round communication scenario.
- **The result step:** each worker node sends the cofactors to the master node and shuts down. The master node sums up all the cofactors received from the different nodes and takes care of the final convergence computation.

The implementations of some of the main functions used in the distributed framework are now closely considered.

5.3 Implementation

The code relative to distribution is quite extensive, and unfortunately giving an exhaustive insight is beyond the scope of this report. Nevertheless, two functions that are important in the distribution process and that are representative of several more general ideas that are used in the project are presented.

The shouldDispatchToNode function

The *shouldDispatchToNode* function is used to determine whether a given tuple should be sent to the different nodes in the network; part of its implementation was already extracted during the discussion about loop unrolling, and the full function is now provided:

Code Excerpt 6: Function shouldDispatchToNode

```
inline void shouldDispatchToNode(dfdb::types::Tuple tuple,
    bool* ignoredWorkers, const std::vector<uint_fast16_t>& attributes)
{
  /*
   * Initially, we consider that the tuple must be sent to all nodes
       and set all values to 0 (= false);
   * a node is dropped if one of the dimension values in its address
       doesn't match the hash of a value in the tuple.
   * We simply set to true when a node is no longer candidate.
   * Use compiler flag if available.
   */
  std::memset(ignoredWorkers, false, NUM_OF_WORKERS);

  /* Scan through all the attributes in the input tuple. */
  for (size_t attr = 0; attr < attributes.size(); ++attr)
  {
```

```
15      /* If the dimension for the current attribute is equal to 0 or 1,
           ignore it. Else check hash values. */
16      if (_attributeDimensions[attributes[attr]] > 1)
17      {
18        /* Stores the result of the hash function. */
19        uint64_t result[2];
20
21        /* Invoke the hash function. */
22        MurmurHash3_x64_128(&tuple[attr], sizeof(dfdb::types::DataType),
23            _attributeHashSeeds[attributes[attr]], &result);
24
25        /* Get a 16 bit hash, modulus the attribute dimension. */
26        uint_fast16_t hash = ((uint_fast16_t) result[0])
27            % _attributeDimensions[attributes[attr]];
28
29        /*
30         * Check hash value inequality for each node.
31         * If compiler flag available, enables loop unrolling and other
              compiler optimisations.
32         */
33        for (size_t nodeID = 0; nodeID < NUM_OF_WORKERS; ++nodeID)
34        {
35          if (_nodesHashAddresses[nodeID
36              + NUM_OF_WORKERS * attributes[attr]] != hash)
37          {
38              ignoredWorkers[nodeID] = true;
39          }
40        }
41      }
42    }
43  }
```

The *DataWriter*, which is in charge of sending data to other workers, iterates through the local database tables, and for each tuple calls *shouldDispatchToNode*, which is the core Hypercube method deciding on tuple shuffling by following the algorithm previously described in Section 5.1. The *DataWriter* then iterates through the *ignoredWorkers* array which was passed as an input of *shouldDispatchToNode*, and writes the tuple to the output buffers corresponding to each node in the network. These output buffers were implemented on top of the iostreams during the project, as eagerly writing tuples is inefficient. Each node has a dedicated output buffer, which is flushed to the network when full. Several interesting facts about the *shouldDispatchToNode* implementation can be highlighted:

- The coordinate for a given tuple is not explicitly represented for efficiency reasons, contrarily to what was shown in the previous example. Instead, it is initially assumed that the tuple should be sent to all nodes in the network (line 10), and as soon as a hash that does not match the dimension of a node is calculated (lines 35 and 36), the corresponding node is marked as ignored to prevent the tuple from being sent to it.

- The MurmurHash library produces a 128-bit hash, stored in an array of two 64-bit integers (line 19). This gives best performance and hash properties on 64-bit systems. To ensure that the value of the hash is strictly inferior to the size of the attribute's dimension, only one of the 64-bit integers is considered and the modulus operator is applied to bring it in the $[\![0; dimensionsize - 1]\!]$ integer interval (line 27). The structure *attributeHashSeeds*, which is used as one of the inputs of MurmurHash (line 23), contains a different hash seed for each attribute. The seeds are extracted from the article *Good HashTable Primes* [42], and have proven to give good results when used in hashing contexts by preventing that too many values end up colliding with the same hash.

- Join attributes with dimension size equal to 1 and non join attributes with size 0 are treated similarly (line 16). Indeed, attributes with dimension size 1 have all their values hashing to 0; all the nodes in the network have their coordinate for that dimension equal to 0, as the dimension must be strictly inferior to the dimension size which is equal to 1. Join attributes with dimension size equal to 1 can therefore be ignored, and it is unnecessary to compute a hash for them. Only attributes with a dimension greater or equal to 2 may prevent a tuple from being sent to a given node.

- The usage of the NUM_OF_WORKERS variable can be highlighted; this will enable loop unrolling on this variable if specified at compile time, as discussed in Section 3.

The readFromStreams function

The *readFromStreams* function is used by DF to receive data from other nodes in the network. An excerpt of the core part of its implementation is provided below:

Code Excerpt 7: Function readFromStreams

```
/* Launch threads to read from each other node from the network; skip current node. */
for (int worker = (_nodeInfo->getNodeID() + 1) % NUM_OF_WORKERS; worker != _nodeInfo->getNodeID(); worker = (worker + 1) % NUM_OF_WORKERS)
{
  streamReaders[worker] = thread([this, worker]()
  {
    /* Iterate through all the tables in the database. */
    for (size_t table = 0; table < NUM_OF_TABLES; ++table)
    {
      /* Size of the current table's tuples, in bytes. */
      const size_t SIZE_OF_TUPLE =
                _dataHandler->getTableAttributes()[table].size() * sizeof(DataType);

      /* Construct the delimiter tuple, to detect when a new table is being transmitted. */
14        char* delimiterTuple = new char[SIZE_OF_TUPLE];
15        /* The delimiter tuple has all its bytes equal to the DELIMITER
             byte. */
16        memset(delimiterTuple, DELIMITER, SIZE_OF_TUPLE);
17
18        /* Keep on reading, unless the stream is closed. */
19        while(_nodeConnections[worker]->first)
20        {
21          /* Create a new tuple. */
22          Tuple tuple = new DataType[_dataHandler->getTableAttributes()
                [table].size()];
23
24          /* Assign all the values in the tuple by reading from the
                iostream. */
25          _nodeConnections[worker]->first.read(reinterpret_cast<char
                *>(tuple), SIZE_OF_TUPLE);
26
27          /* Check that the received tuple is not equal to the
                delimiterTuple. */
28          /* Expect tuple not to be delimiter; indication for compiler
                optimisations for this performance critical section. */
29          if(likely(memcmp(tuple, delimiterTuple, SIZE_OF_TUPLE) != 0))
30          {
31            /* Add tuple to the _receivedDataToProcess database. */
32            _dataHandler->getReceivedDataToProcess()[worker][table].
                  push_back(tuple);
33          }
34          else
35          {
36            delete[] tuple;
37            /* Move on to the next table. */
38            break;
39          }
40        }
41        delete[] delimiterTuple;
42      }
43    });
44  }
```

This function is called a single time at the beginning of the query step of the distribution, and enables to receive and parse all the tuples sent by other nodes. Several interesting facts can be highlighted about the implementation:

- A new thread is launched for each other worker in the network (lines 2 to 4). This enables to read in parallel from all the nodes and avoids any delays in the operations, as the reader is able to process the received information as soon as it becomes available.

- Each thread adds the tuples to its own database structure (line 32), which is later on efficiently merged before the learning process. This avoids synchronisation such as having to lock a whole database structure each time a tuple is added, which would bring overhead in the case of numerous workers and a large amount of tuples.

- Raw bytes are sent and read from the network (line 25), and it is up to the reader to determine what amount must be read at a time to fill in a tuple entirely. In the project's current setting, *double* data types of size 8 bytes are sent across the network; when receiving tuples from the US retailer *Census* table containing a total of 17 attributes, 17*8 = 136 bytes would have to be read from the network to fill in the tuple; to be more precise the network is not directly read from, but information is extracted from the reception buffer of the underlying iostreams. The number of bytes to read at a time is defined at lines 10 and 11.

- All the tables are sent one after the other without any high level mechanism to alert the reader that the writer is moving on to the next table. Therefore what is called the *delimiterTuple* is used (line 14) to indicate to the reader that it is now reading tuples belonging to the next database table, and must in consequence update the *table* index variable defined at line 7. This delimiter tuple is initialised in such way that all its bytes are set to a maximum value (the *DELIMITER* byte), in other words the 8 bytes of all the doubles in a tuple are set to the *DELIMITER* byte. This would correspond to a succession of *NaN* (Not-a-Number) doubles in the case of DF, which are non-representable values as defined by the C++ standard. This guarantees that a regular tuple cannot be considered as a delimiter tuple by mistake, and that the delimiter tuple is uniquely defined. After having read a new tuple, the function checks that it is not equal to the delimiter tuple with a low level memory comparison function for better efficiency (line 29). Otherwise, the current thread moves on to the next table.

- The *likely* construct can be noticed (line 29); it corresponds to *__builtin_expect((x), 1)* behind the scenes [19]. This helps the compiler do better optimisations by providing it with branch prediction information, and it can therefore rearrange the generated assembly code for better efficiency. These constructs are used at several occasions throughout the code, alongside the *unlikely* construct which has the opposite meaning (equivalent to a *__builtin_expect((x),0)*). In this specific case it is assumed that there are several tuples in each table and that the delimiter tuple indicating that the function should move on to the next table is unlikely to be received (in other words the memory comparison is likely to fail in the code). It is important to underline the fact that these *likely* and *unlikely* constructs are indications for better code optimisation, and that the unexpected outcome just leads to a branch misprediction at the CPU level, but does not yield any incorrect results.

5.4 Experiments

A series of experiments is conducted to measure the impact of distribution. In this thesis, focus is not put on finding the best possible Hypercube configuration, but rather showing the impact of distribution in simple settings. Each dataset is tested in several distribution settings, featuring from one to six nodes. The performance for a single node setting is different from what was reported in the previous parts of the thesis; indeed, as detailed in Appendix C, the machines available in the cluster of six nodes are different from the one used in the SIGMOD publication [1], which is also the one used in all the previous experiments. The experiments in this section are conducted with single-threaded F running on each node, to be able to measure the impact of distribution independently of multi-threading.

The following plots contain six groups of histograms in each one of them; they correspond to configurations from one to six nodes in the network. Two additional time components are reported compared to what has been done up to now. Communication corresponds to the time taken to send and receive tuples from the network. Synchronisation corresponds to the time taken to synchronise at the different steps described in the distribution process. It is mainly due to the discrepancies resulting in the workload of different nodes, as the overhead brought by sending the synchronisation messages themselves is very small. For instance a node that has completed its cofactor computation faster than others will wait until everyone is finished before sending its results to the master node.

Node 0 corresponds to both a master and a worker running on a same machine, as described previously. It is therefore the only node to feature a convergence step and produces the final result. Node 0 is consequently the one to be considered to get the end-to-end runtime in a distributed setting. The other nodes correspond to workers in the system.

Initially each node contains a distinct subset of the whole dataset. For instance, when working with six nodes, each CSV file corresponding to a table in a given dataset is divided linearly into six different CSV files containing an equivalent number of tuples. These partitions of the database tables are uploaded to the different nodes in the network before DF is launched, so that each worker loads and shuffles one sixth of the whole dataset. Similarly, when working with two nodes, each node has a half of the whole dataset.

As usual, the US retailer, LastFM and MovieLens datasets described in Appendix B are considered separately with a distinct plot.

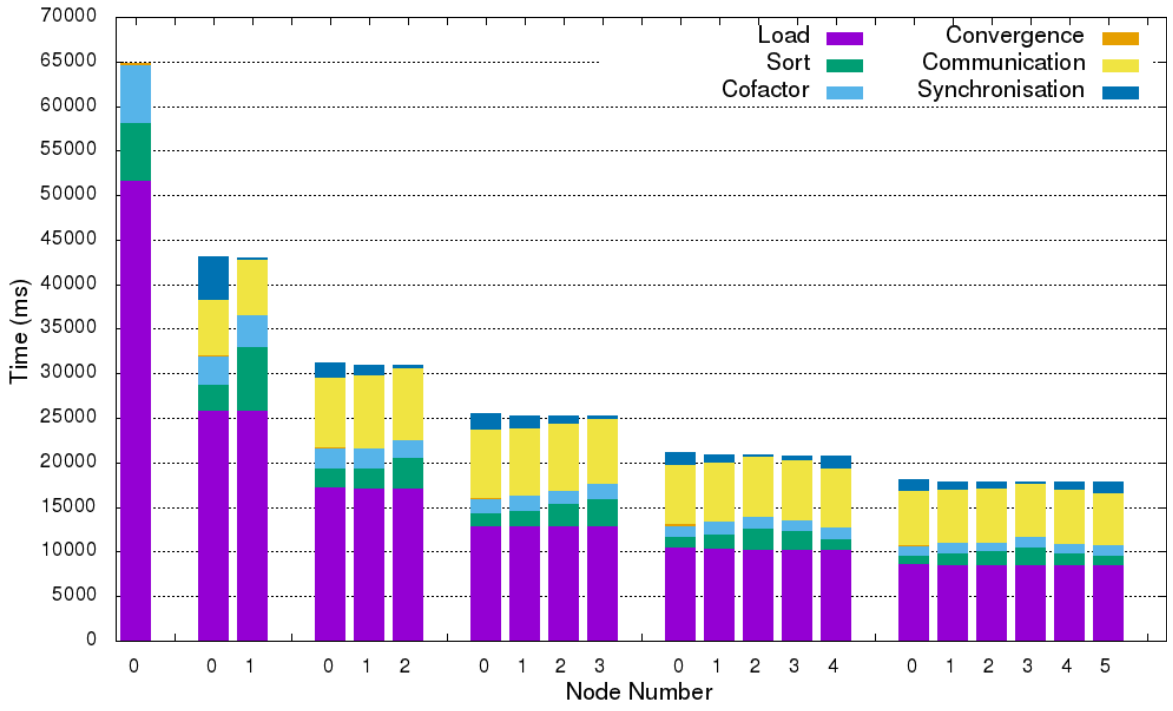


Figure 12: Performance of DF on US retailer depending on the number of nodes in the network; each group of bars corresponds to a setting with a different number of nodes

The US retailer dataset is from far the largest one used in the experiments, and as could be expected it significantly benefits from distributing the overall workload on a cluster of machines. When using six nodes instead of one, a $3.58\times$ speedup can be observed for an end-to-end run of the program.

The Hypercube configuration considered in this example is such that the dimension size for the *locn* join attribute is equal to the number of nodes in the network, and the dimension size for the *zip* join attribute is equal to 1 ($n = d_1 * d_2$, where $d_1 = n$ and $d_2 = 1$). The impact of shuffling the *Census* and *Location* tables is ignored as their size is negligible compared to that of the *Inventory* table. In the two nodes configuration, it can be expected that each node sends about half of its *Inventory* table partition to the other node, assuming that the hash function uniformly clusters the tuples. Each node having an initial partition containing half of the total data, it sends a quarter of the 84055817 tuples contained in the *Inventory* table. As there are two nodes in the network, overall half of the database is being shuffled across the network. In the six nodes configuration, each node sends $\frac{5}{6}$ of its partition containing a sixth of the total dataset, and keeps $\frac{1}{6}$ locally, assuming uniform clustering with the hash function. This means that it sends and receives about $\frac{5}{36}$ of the total dataset. But as there are six nodes in the network, overall $\frac{30}{36}$ of the whole dataset is shuffled. When looking at the real number of tuples shuffled, the assumption regarding the uniformity of the hash function is generally verified, as there are only small discrepancies in the partitioning of the data in most cases. One would therefore expect each node spending less time on communication as the number of nodes increases, since it receives and sends a smaller proportion of the whole dataset. Nevertheless, this does not happen in practice in the experimental setting, and the time spent on communication even increases with three or four nodes compared to the case with two nodes. This is due to the fact that overall more tuples are being shuffled. The cluster of nodes physically share the same network components due to their location, and the network is simply saturated by the overall communication process involving the sharing of the huge amount of data contained in the US retailer dataset.

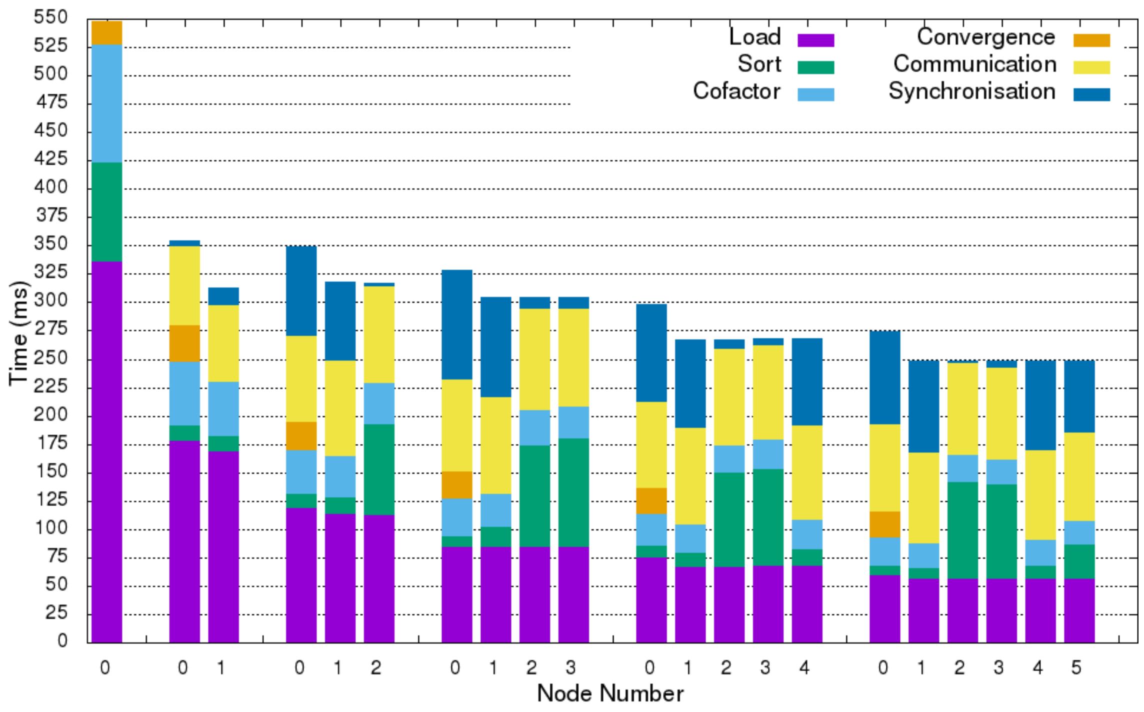


Figure 13: Performance of DF on LastFM depending on the number of nodes in the network; each group of bars corresponds to a setting with a different number of nodes

The LastFM dataset is the smallest one used in the experiments, by more than two orders of magnitude compared to US retailer, and does not benefit as much from distribution. When using six nodes instead of one, a 1.99× speedup can be observed for an end-to-end run of the program.

It can be noticed that due to the small size of the dataset, depending on how the data is shuffled, the workload on a given node can be fairly different from the workload of others. This is in particular the case with the sorting component. The sort function's performance is very dependant on the amount of data received by the node and the way the data is initially disposed, especially regarding the fact that it is a parallel implementation. The underlying algorithm featured in GCC's parallel mode [16] is based on Quicksort, and worst case scenarios can therefore degenerate in squared time complexity instead of the average $n.log(n)$ complexity.

All these discrepancies in calculations infer huge synchronisation times on some nodes, which are negligible with the two other larger datasets. As it is generally the case with most software systems, it can therefore be concluded that using distribution on small datasets is less beneficial in the case of DF.

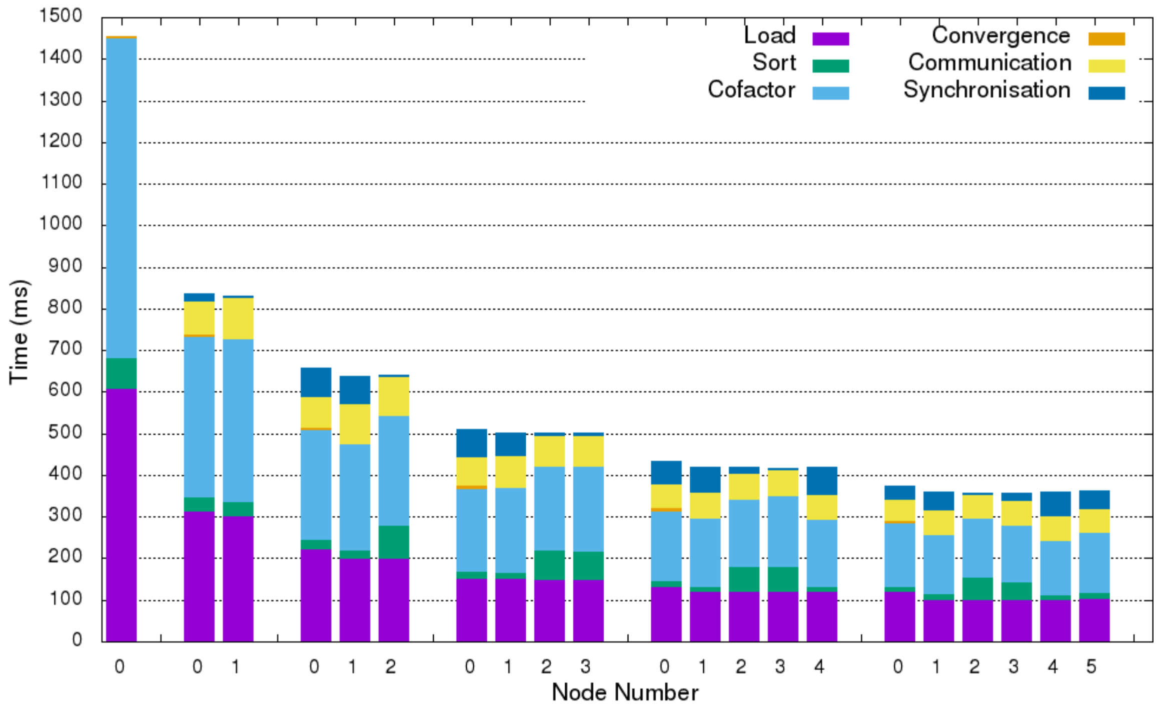


Figure 14: Performance of DF on MovieLens depending on the number of nodes in the network; each group of bars corresponds to a setting with a different number of nodes

This dataset is significantly larger than LastFM and benefits greatly from distribution. When using six nodes instead of one, a 3.78× speedup can be observed for an end-to-end run of the program.

Contrarily to the US retailer and LastFM datasets, the cofactor computation outweighs the combination of the communication and loading components. Indeed, compared to the two other datasets, the size of the factorised join with MovieLens is relatively bigger compared to the number of tuples in the dataset. Therefore less time is spent on loading and shuffling the data, but more time is spent on doing the in-memory processing.

In all cases, but in particular with the US retailer and LastFM datasets, it can be noticed that the cost of communication is relatively high, as a lot of tuples must be shuffled around the network. However, this is not a major concern from a more general point of view. Indeed, in real world scenarios, either this shuffling process would only be done a unique time on the whole dataset and would then only happen for new tuples being added to the database, either the data would be pre-partitioned in such a way that only a small amount of tuples needs to be sent to other nodes. Without the communication component, the speedup between a single-node environment and a network composed of six nodes would be 5.37× for US retailer, 2.76× for LastFM and 4.34× for MovieLens. Nevertheless, even in the current experimental setting with a very simple way of dividing the datasets and with shuffling fully happening at each run of DF, the speedups are substantial. Dividing the data between several nodes means that each node considered individually has a smaller amount of the overall data to load, sort and operate on for calculations. This significantly outweighs the overhead brought by communication and accounts for the important performance increases than are observed when using DF.

6 F++: Running F with the best configuration

In this section, F is used with what is considered to be the best found configuration according to the observations of all the previous experiments. The following three settings are benchmarked:

- OldF, in other words the original version of F used in the SIGMOD publication [1].
- F with the latest code basis, including the enhancements discussed in sections 3.1 to 3.4 (F-Flags, loading and sorting rewriting, improvements on the core parts).
- **F++**, which denotes the current best found setting for each dataset. This version uses both DF (on six nodes) and MF, with an optimised number of partitions and threads (4 threads with 2^{14} partitions for US retailer, 8 partitions for LastFM and 32 partitions for MovieLens). In addition some variable unrolling features and profile-guided optimisations are used (Sections 3.5 and 3.6).

The overall time reported at the master node is first considered. This represents the time taken to produce the final parameter coefficients of the calculation.

		US retailer	LastFM	MovieLens
Overall	OldF	742232	4614	11569
time	F	64851	551	1462
(ms)	F++	17497	272	310
Speedup	**F++** vs. OldF	42.42×	16.96×	37.32×
	F++ vs. F	3.71×	2.03×	4.72×

Table 6: Overall performance of F++ vs. F vs. OldF

Compared to what was obtained with the original version of OldF, this project has brought massive speedups ranging from 16.96× to 46.42× for an end-to-end run of F, depending on the dataset considered. Calculations that previously took several seconds or minutes can now be respectively completed within milliseconds or seconds, which makes F even more competitive compared to other state-of-the-art systems.

The time for the cofactor computation is now considered alone. Indeed, the multi-threading implementation as discussed in Section 4 only improves this component, so it is relevant to measure the impact on what can be considered as the core machine learning component of F independently of the others. Cofactor computation occurs in parallel on six nodes, and as discussed in Section 5, there can be discrepancies between the workload of each node. Therefore, the timing reported by the slowest worker is selected, as the full cofactor matrix that would be computed in a single-threaded and non distributed environment can easily be obtained at that point in time.

		US retailer	LastFM	MovieLens
Cofactor	OldF	25964	331	3598
time	F	6487	87	771
(ms)	F++	660	20	77
Speedup	**F++** vs. OldF	39.34×	16.55×	46.73×
	F++ vs. F	10.28×	4.35×	10.01×

Table 7: Cofactor performance of F++ vs. F vs. OldF

Compared to the timings obtained with the original version of OldF, this project has brought speedups ranging from 16.55× to 46.73× for cofactor computation, depending on the dataset considered. The speedup obtained by using limited multi-threading with only four threads, limited distribution with six nodes and PGO is of an order of magnitude for the US retailer and MovieLens datasets, compared to the standard optimised F obtained at the end of Section 3.4. The speedup for the LastFM dataset is slightly below expectations, and is due to the relatively small size of the dataset, as demonstrated in Section 5.

It is interesting to notice that for cofactor computation, the project has the biggest impact on the performance for the MovieLens dataset, then the US retailer dataset, and finally the LastFM dataset for which the speedup is less pronounced. Nevertheless, as highlighted in Section 2.4, OldF held the strongest advantage over its competitors for the LastFM dataset, then the US retailer dataset, and finally the MovieLens dataset. This thesis therefore enabled to give F a stronger performance boost for datasets it did not deal as well with, therefore uniforming its competitive advantage over other systems for different datasets.

7 Conclusion and future work

The objective of this thesis was to turn F into a scalable system for learning regression models over factorised joins.

A first phase consisted in improving F from an engineering point of view, by redesigning new data loading and sorting components, optimising data structures, exploring compiler flag optimisations specific to the project, fixing memory management issues and introducing a multitude of other enhancements. These changes not only improved F's performance by almost an order of magnitude, but also integrated it in a rigorously designed framework, with features such as multiple levels of logging, different build modes, advanced command line interactions with the user, and files to generate a comprehensive technical documentation of the C++ project.

A second phase consisted in finding new ways to scale F even further. Multi-threading in a factorised setting was explored; novel issues such as partition selection and cache sharing were highlighted and solutions to cope with them were implemented. Distribution was also explored. The Hypercube algorithm was implemented for the first time in a context of learning regression models, and a complete distribution framework was provided, in order for several machines running F to be able to cooperate with one another. Bringing parallelism to F, at the level of a single machine with multi-threading and at the level of several machines with distribution, enabled to significantly improve the scalability of the system. In the setting used in this thesis, combining multi-threading and distribution brought further performance improvements of up to an order of magnitude on some of F's components.

In addition to all these contributions, there are still many areas to explore, and the following ideas are suggested as a starting point for future research work or MSc Thesis projects:

- Exploring new schemes to divide the work between threads and compare them to the current partitioning scheme implemented by this thesis. For instance, integrating multi-threading more deeply into F was discussed during this project, by dispatching threads while recursively exploring paths of values and computing regression aggregates in the factorisation tree.

- Implementing multi-threading in the convergence component. Currently F explores only one possible regression model during this step, and this could be extended by letting different threads work with different rules and calculate separate parameters for each one of them.

- Focusing on Hypercube configurations. The algorithm was fully implemented in this project, but used only in simple configurations throughout the experiments. Studying the impact of different configurations on F's performance would allow to design an algorithm that would automatically select the best possible setting based on dataset and d-tree considerations.

- Investigating new ways to further reduce the overhead of communication. Factorising tuples or using compression before sending data over the network could for instance bring performance improvements if the in-memory pre and post-processing is done efficiently enough.

This concludes the presentation of the work that was accomplished and of the contributions that were brought by this thesis. The following appendices contain references as well as details about the datasets and the experimental setup.

Appendices

A References

A.1 Scientific publications

[1] M. Schleich, D. Olteanu, R. Ciucanu, *Learning Linear Regression Models over Factorized Joins.* In SIGMOD, 2016.

[2] N. Bakibayev, D. Olteanu, J. Zavodny, *FDB: A Query Engine for Factorised Relational Databases*, In PVLDB, 2012.

[3] D. Olteanu, J. Zavodny, *Size Bounds for Factorised Representations of Query Results.* In FOCS, 2015.

[4] F. N. Afrati, J. D. Ullman, *Optimizing Joins in a Map-Reduce Environment.* In PVLDB, 2010.

[5] P. Beame, P. Koutris, D. Suciu, *Communication Steps for Parallel Query Processing.* In PODS, 2013.

[6] P. Beame, P. Koutris, D. Suciu, *Skew in Parallel Query Processing.* In PODS, 2014.

[7] T. L. Veldhuizen, *Triejoin: A Simple, Worst-Case Optimal Join Algorithm.* In ICDT, 2014.

[8] M. Aref, B. ten Cate, R. J. Green, B. Kimefeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, J. Washburn, *Design and Implementation of the LogicBlox System.* In SIGMOD, 2015.

[9] K. Hansen, F. Biegler, R. Ramakrishnan, W. Pronobis, O. A. von Lilienfeld, K.-R. Mülle, A. Tkatchenko, *Machine Learning Predictions of Molecular Properties: Accurate Many-Body Potentials and Nonlocality in Chemical Space.* In JPCL, 2015.

[10] S. Venkataraman, A. Yang, D. Liu, E. Liang, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. Franklin, I. Stoica, M. Zaharia, *SparkR: Scaling R Programs with Spark.* In SIGMOD, 2016.

[11] C. Zhang, J. Shin, C. Ré, M. Cafarella, F. Niu, *Extracting Databases from Dark Data with DeepDive.* In SIGMOD, 2016.

[12] J. Ortiz, B. Lee, M. Balazinska, *PerfEnforce Demonstration: Data Analytics with Performance Guarantees.* In SIGMOD, 2016.

[13] U. Drepper, *What Every Programmer Should Know About Memory*, 2007.

[14] H. Garcia-Molina, J. D. Ullman, J. Widom, *Database Systems - The Complete Book.* Page 43, 2001.

A.2 Compilers

[15] GNU Project, *Optimize Options*, gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/Optimize-Options.html, accessed April 2016.

[16] GNU Project, *Parallel Mode*, gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html, accessed June 2016.

[17] GNU Project, *Common Function Attributes*, gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html, accessed May 2016.

[18] GNU Project, *×86 Options*, gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/x86-Options.html, accessed June 2016.

[19] GNU Project, *Other Builtins*, gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html, accessed June 2016.

A.3 Libraries

[20] OpenMP, *API for parallel programming*, openmp.org, accessed June 2016.

[21] Boost C++ Libraries, *Spirit 2.5.2*, boost.org/doc/libs/1_61_0/libs/spirit/doc/html/index.html, accessed June 2016.

[22] Boost C++ Libraries, *Boost Iostreams*, boost.org/doc/libs/1_61_0/libs/iostreams/doc, accessed May 2016.

[23] Boost C++ Libraries, *Hash Combine*, boost.org/doc/libs/1_61_0/doc/html/hash/combine.html, accessed May 2016.

[24] A. Appleby, *MurmurHash*, github.com/aappleby/smhasher, accessed May 2016.

A.4 Tools and software

[25] J. Seward et al, *Valgrind*, valgrind.org, accessed April 2016.

[26] The MADlib contibutors, *Apache MADlib: Big Data Machine Learning in SQL for Data Scientists*, madlib.incubator.apache.org, accessed May 2016.

[27] The StatsModels contibutors, *StatsModels: Statistics in Python*, statsmodels.sourceforge.net, accessed May 2016.

[28] R. Gentleman, R. Ihaka et al, *The R Project for Statistical Computing*, r-project.org, accessed May 2016.

[29] D. van Heesch, *Doxygen*, stack.nl/ dimitri/doxygen, accessed August 2016.

A.5 C++ language documentation

[30] C++ Reference, *C library to perform Input/Output operations*, cplusplus.com/reference/cstdio, accessed June 2016.

[31] C++ Reference, *File Streams*, cplusplus.com/reference/fstream, accessed June 2016.

[32] C++ Reference, *String Streams*, cplusplus.com/reference/sstream, accessed May 2016.

[33] C++ Reference, *Strings*, cplusplus.com/reference/string, accessed April 2016.

[34] C++ Reference, *Vector Data Structure*, cplusplus.com/reference/vector, accessed April 2016.

[35] C++ Reference, *Standard Template Library: Algorithms*, cplusplus.com/reference/algorithm, accessed May 2016.

[36] C++ Reference, *Map Data Structure*, cplusplus.com/reference/map/map, accessed May 2016.

[37] C++ Reference, *Unordered Map Data Structure*, cplusplus.com/reference/unordered_map/unordered_map, accessed May 2016.

[38] C++ Reference, *Atomic Types*, cplusplus.com/reference/atomic, accessed July 2016.

[39] C++ Reference, *Memory Order*, cplusplus.com/reference/atomic/memory_order, accessed July 2016.

[40] C++ Reference, *System Clock*, cplusplus.com/reference/chrono/system_clock, accessed May 2016.

A.6 Miscellaneous

[41] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, 2008.

[42] A. Krowne, *Good HashTable Primes.*, planetmath.org/goodhashtableprimes, 2013.

B Datasets

Three datasets are used in this thesis, and are detailed in this section of the appendix. These datasets are also used in the SIGMOD publication [1].

B.1 US retailer

The aim of F with this dataset is to forecast customer demands and sales by predicting the inventory quantities based on other features. It features three database tables:

- **Inventory:** this relation contains 84055817 tuples and stores information about inventory items at a given location and date.
- **Census:** this relation contains 1293 tuples and stores information about population for a given zipcode.
- **Location:** this relation contains 1317 tuples and stores information about store locations for a given zipcode.

The natural join of these three database tables is considered, based on the following d-tree:

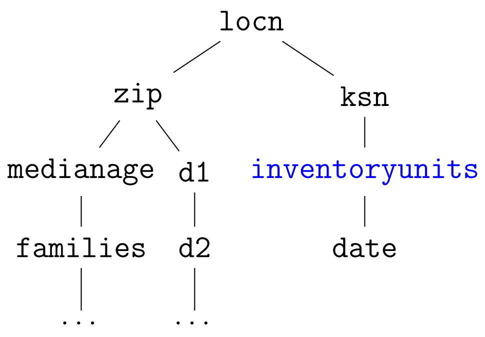


Figure 15: D-tree for US retailer

The size of the factorised join is 97134867 values without caching and 97134675 values with caching. Caching therefore does not benefit this dataset much.

B.2 LastFM

The aim of F with this dataset is to measure how often friends listen to the same musical artists. It features three database tables:

- **Userfriends:** this relation contains 25434 tuples and stores information about users and their friends.
- **Userartists:** this relation contains 186479 tuples and stores information about the artists users listen to.
- **Usertaggedartiststimestamps:** this relation contains 92834 tuples and stores information about the artists tagged by users.

Two copies of the *Userartists* and *Usertaggedartiststimestamps* tables are considered and the join is done according to the following d-tree (the copies are labelled 1 and 2):

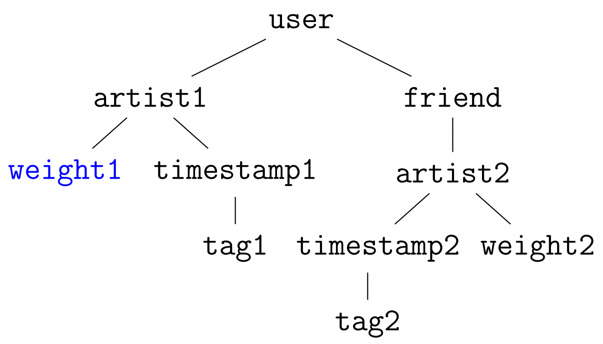


Figure 16: D-tree for LastFM

The size of the factorised join is 2379264 values without caching and 315818 values with caching. This dataset therefore features a very high amount of caching.

B.3 MovieLens

The aim of F with this dataset is predict the rating a user will give to a movie. It contains three database tables:

- **Users:** this relation contains 6040 tuples and stores information about users.
- **Movies:** this relation contains 3883 tuples and stores information about movies.
- **Ratings:** this relation contains 1000209 tuples and stores information about the ratings users give to movies.

The natural join of these three tables is considered, with the following d-tree:

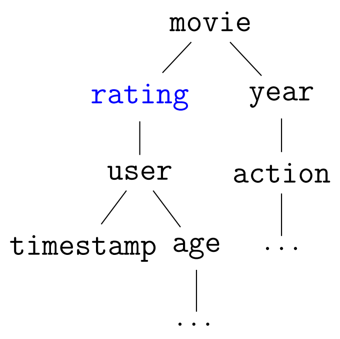


Figure 17: D-tree for MovieLens

The size of the factorised join is 6092186 values without caching and 2115610 values with caching. This dataset therefore features a high amount of caching, though not as important as with LastFM.

C Experimental setup

Two different types of machines are used in this project. This was necessary to be able to compare with the previous times reported on F, but also to test and benchmark the distribution implementation on a network of machines. Many thanks to the DIADEM research group of the University of Oxford for granting a full access to all these machines.

For Sections 3 and 4, corresponding to experiments in a single node environment, the same machine as the one reported in the SIGMOD publication [1] is used. The performance timings at the end of Section 2 ("Performance of F vs. state-of-the-art") are therefore also obtained with this same machine, allowing to directly compare the times reported. It has the following characteristics:

- Intel(R) Core(TM) i7-4470 3.40GHz
- 64-bit architecture
- 8 cores
- 32GB DDR3 RAM
- Samsung SSD 840 PRO Series
- Linux Mint 17 Qiana (GNU/Linux 3.13.0-24-generic x86_64)
- GCC 4.8.4
- Boost 1.61.0

For Sections 5 and 6, six different machines with the specifications listed below are used. All the experiments in these two sections, including the times reported with OldF or single-threaded F, use one or several of these machines in order to establish fair comparisons with regard to distribution. It is important to underline the fact that these machines are slower than the one used in Sections 2, 3 and 4 as well as in the SIGMOD publication [1], especially in terms of hard disk loading, but they enabled to conveniently build a cluster of nodes for distribution.

- Intel(R) Xeon(R) E5-2407 v2 2.40GHz
- 64-bit architecture
- 4 cores
- 32GB DDR3 RAM

- NearLine SAS 7.2K RPM Hard Drive
- Linux Ubuntuu 14.04 (GNU/Linux 3.16.0-76-generic x86_64)
- GCC 4.8.4
- Boost 1.58.0
- Gigabit Ethernet

All the timings reported in the different experiments discussed throughout this document are averaged on **ten consecutive executions** of the program. All the timings are in milliseconds rounded to the closest unit and represent the **wall-clock** time of the program's execution, in other words the "real" time perceived by a user. They were retrieved using *system_clock* class objects from the C++ *chrono* header [40].