Physical Design in Databases

Endre Palatinus

Thesis for obtaining the title of
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken, Germany
July 2016

To my family

Acknowledgements

I would like to thank my supervisor, Jens Dittrich for guiding me during my PhD. I appreciate his efforts on showing me how to present research ideas and results clearly and effectively. Besides doing research together, I was also lucky enough to take part in his experiments in novel teaching approaches, namely in multiple flavours and iterations of the flipped classroom model.

I would also like to thank Dr. Michelle Carnell, the Program Manager of the Saarbrücken Graduate School, for convincing me to do my PhD at Saarland University, and for the very welcoming atmosphere at the Graduate School.

The research work presented in this thesis was supported by the funding from the German Ministry of Education and Science (BMBF).

Abstract

We live in an age where data has become one of the most important assets of companies. Data in itself is valuable, yet it has to be turned into information to become useful. This is where database management systems come into the picture. They allow for efficient processing of even terabytes of data, and thus provide the basis of knowledge extraction and information retrieval. A high-performance database system is an essential requirement for making big data analysis possible.

The performance of database systems can be improved at multiple levels of the system, and using various approaches. In this work we focus on data layouts, and also investigate the performance implications of compiling hand-written queries and whole database systems as well.

We present an exhaustive experimental study on vertical partitioning algorithms. Vertical partitioning itself is a physical design technique to partition a given logical relation into a set of physical tables, called vertical partitions. It is a crucial step in physical database design in legacy row-oriented databases.

We show a survey of query processing on top of flat files, which are text files containing data encoded in some standard text format. Flat files are commonly used in various fields of science to store experimental results in a human-readable format.

We explore the performance implications of compiling both hand-written queries, and whole database systems as well. We present two techniques for improving query performance that build upon changing compiler setups, and apply them in a main-memory database system.

Zusammenfassung

Wir leben in einem Zeitalter, in dem Daten eine der wichtigsten Ressourcen eines Unternehmens darstellen. Obwohl Daten bereits in ihrer Rohform ein wertvolles Gut sind, müssen zunächst die Informationen aus ihnen herausgearbeitet werden, um sie verwertbar zu machen. Genau an diesem Punkt treten Datenbanksysteme in Erscheinung.

Diese ermöglichen eine effiziente Verarbeitung von Daten in der Größenordnung von Terabytes und stellen damit die Grundlage von Wissensextraktion und Informationsgewinnung dar. Ein Hochleistungsdatenbanksystem ist daher eine unentbehrliche Anforderung, um Big Data Analysen überhaupt erst möglich zu machen.

Die Leistungsfähigkeit von Datenbanksystemen kann auf mehreren Ebenen und unter dem Einsatz verschiedenster Techniken verbessert werden. In dieser Arbeit konzentrieren wir uns auf die (physische) Anordnung von Daten und untersuchen darüber hinaus die Auswirkungen von Kompilierung auf handgeschriebene Anfragen sowie auf komplette Datenbanksysteme.

Zunächst präsentieren wir eine ausführliche experimentelle Studie über vertikale Partitionierungsalgorithmen. Vertikale Partitionierung ist eine Methode aus dem Bereich des physischen Datenbankentwurfs, bei der eine logische Relation in eine Menge von physischen Tabellen zerlegt wird. Es stellt einen fundamentalen Schritt des physischen Datenbankentwurfs in zeilenorientierten Datenbanksystemen dar.

Darüber hinaus präsentieren wir eine Studie über Anfrageverarbeitung auf einfach strukturierten Textdateien. Dieses Format wird in verschiedenen Bereichen des wissenschaftlichen Arbeitens verwendet, um experimentelle Ergebnisse in einer lesbaren Form abzuspeichern.

Des weiteren untersuchen wir den Einfluss der Kompilierung auf das Laufzeitverhalten von handgeschriebenen Anfragen sowie von kompletten Datenbanksystemen. Wir präsentieren zwei verschiedene Methoden zur Beschleunigung der Anfrageverarbeitung, die auf Anpassungen der Kompilierungseinstellungen beruhen, und wenden diese in einer Hauptspeicherdatenbank an.

Contents

Chapter 1

Introduction

This work is a journey from data layouts to compiling database systems with a few pit stops at the blend of both topics – data layouts for flat file processing, managing large-scale experimental results in a statistically sound way, and efficiently compiling a new main-memory data layout. In the following we elaborate on each of these topics.

1.1 Vertical Partitioning for Legacy Row Stores

Vertical partitioning is a physical design technique to partition a given logical relation into a set of physical tables, called vertical partitions. It is a crucial step in physical database design in legacy row-oriented databases. A number of vertical partitioning algorithms have been proposed over the last three decades for a variety of niche scenarios. In principle, the underlying problem remains the same: decompose a table along its attributes into one or more vertical partitions. However, it is not clear how good different vertical partitioning algorithms are in comparison to each other. In fact, it is not even clear how to experimentally compare different vertical partitioning algorithms.

1.1.1 Research questions

Vertical partitioning strongly affects the query performance in many ways. Various vertical partitioning algorithms have been proposed by several researchers over time. As a result, users now have the problem of choosing the right one for their needs. In contrast to physical design tools, which choose a layout given a vertical partitioning algorithm, the problem here is to choose the vertical partitioning algorithm in the first place. Essentially, the questions that we are looking at are:

- Which are the major algorithms proposed? What is the difference between those algorithms?
- For which settings were different algorithms proposed? What are their pros and cons?
- What are the primary differences between different vertical partitioning settings? Can we abstract the settings from the algorithms?
- How do we compare different algorithms in a common setting? What would be the right measures for comparison?
- How do the different algorithms compare against each other? When to use which algorithm?

Thus, there is an absence of a systematic and comparative study of vertical partitioning algorithms.

1.1.2 Contributions

We present an exhaustive experimental study on vertical partitioning algorithms. Our main contributions are as follows:

- Given the large number of vertical partitioning algorithms proposed in the literature, we first understand the fundamental differences between them. To do so, we first classify them along three dimensions, namely: (i) search strategy, (ii) starting point, and (iii) candidate pruning.
- From the above categories, we survey six representative vertical partitioning algorithms, namely: (i) AutoPart [51], (ii) HillClimb [28], (iii) HYRISE [26], (iv) Navathe's algorithm [46], (v) O_2P [35], and (vi) Trojan layouts [37]. We present a brief summary and the context of each of the algorithms.
- We describe how the different vertical partitioning algorithms can be applied in the same setting. Even though each algorithm was proposed for a different setting, we can still unite them under a common umbrella.
- We present a systematic way of comparing different vertical partitioning algorithms. For this purpose, we introduce four metrics, namely: (i) *how fast* in terms of computation times, (ii) *how good* in terms of workload runtimes, (iii) *how fragile* in terms of predictable runtimes, and (iv) *where does it make sense* to use vertical partitioning.

- We show detailed experimental results from six vertical partitioning algorithms over the TPC-H benchmark and with row and column layouts as baselines. We discuss each of the four metrics for the six vertical partitioning algorithms.

- Finally, we discuss the four key lessons learned.

1.1.3 Publications

Alekh Jindal, Endre Palatinus, Vladimir Pavlov, Jens Dittrich
A Comparison of Knives for Bread Slicing
PVLDB, 6(6):361–372, 2013

1.2 Query Processing on Top of Flat Files

In this chapter we show a survey of query processing on top of flat files, i.e. text files containing data encoded in some standard text format. Flat files are commonly used in various fields of science to store experimental results in a human-readable format. These datasets are considered external to a database management system, and to efficiently process them using a DBMS, they first have to be loaded into the database, which imposes a higher time-to-query (the time we have to wait till we can fire the query). Other ways of processing these datasets are using custom-built applications that operate on the flat files directly. These approaches are expected to yield an inferior query performance, yet with zero time-to-query. We will show cases where this is not completely true, and where flat file processing tools can even outperform a DBMS.

1.2.1 Research questions

Both databases and scripting languages have a large user base, and both of them would swear on their tool being the right one for processing data residing in flat files. In this chapter we try to give a guideline on when one camp should be better off using the other camp's tool by answering the following research questions:

- Can scripting languages compete with database systems in query processing?

- Does it pay off to invest in loading the data into a database?

- How should we choose the proper tool and data format for flat file processing?

1.2.2 Contributions

Our main contributions in this chapter are as follows:

- We show how to load data into Postgres in the most efficient way. Hereby, we discuss various database tuning steps and tools for data loading.
- We compare three different tools for processing flat files: the Postgres database system, an AWK script, and a hand-written C-application. We compare them on their query time, and time-to-query as well. The latter once includes the cost of loading data into the database, which is only applicable to Postgres.
- We provide two representative examples of queries that exemplify two extreme cases. For single-table queries containing filtering but no grouping, flat file processing tools are better suited, and databases might never pay off because of the upfront costs of loading the data. For more complex queries involving joins, aggregations, and the like, a database system can drastically improve subsequent query times and thus the loading times will eventually pay off.

1.3 Computer Systems Performance Analysis

Performance is a key aspect of every system in computer science. As such, it is often required to build a system that has the highest performance at a given cost. Often, we have multiple solutions for a given problem, and we have to compare their performance to choose the most efficient one. The comparison is done along some metrics of the system, e.g. runtime, latency, throughput, memory usage, etc. The metrics themselves are either measured or estimated. In the former case the measurements are repeated multiple times to reduce the possibility of measurement errors, and to increase the confidence of the results. To summarise the measurements, some statistics of the data are provided, which is most often the average value, i.e. the mean. This, however, is not the most reliable statistics, and thus researchers should favour other statistics.

When comparing a few algorithms that solve a given problem, we usually do not have a lot of measurements available. We typically calculate some statistics over the runtime or memory usage of the different algorithms, apply some statistics over the measurements, and choose the one with the best results. However, when we also consider multiple factors that could have an influence on the performance of these algorithms, we end up adding new dimensions to the problem space. Eventually, managing the experimental results becomes a problem.

1.3.1 Contributions

In this chapter we make the following contributions:

- We present a uniformly applicable method for storing experimental results and problem dimensions in a relational database.
- We describe a flexible way of exploring the effects of the problem dimensions on the performance in a statistically sound way.
- We also highlight the importance of statistical confidence when publishing research results, and discuss the implications on doing statistics the wrong way.

1.4 Runtime Fragility in Main Memory

A compiler is just another abstraction layer. It is safe to use whatever default compiler we have on our system. It has a default O-level, which is just fine for most purposes, thus also for building our database system as well. If we would like to generate the most efficient code, we just go for the highest O-level available. Maybe we even use some fancy optimizations our compiler supports on top of that. Furthermore, if my compiler setup worked well for me, it should work well for anyone else as well. Why are all of the previous statements plain wrong? That is what the last two chapters of this thesis are about.

1.4.1 Fragility of Hand-coded Queries in Main-Memory

In this chapter we consider the following problem: Given a database workload (tables and queries), which data layout (row, column or a suitable PAX-layout) should we choose in order to get the best possible performance? We show that this is not an easy problem.

Contributions

- We explore careful combinations of various parameters that have an impact on the performance including: (1) the schema, (2) the CPU architecture, (3) the compiler, and (4) the optimization level. We include a CPU from each of the past four generations of Intel CPUs.
- We demonstrate the importance of taking variance into account, when deciding on the optimal storage layout. We observe considerable variance throughout our measurements which makes it difficult to argue along means over

different runs of an experiment. Therefore, we compute confidence intervals for all measurements and exploit this to detect outliers and define classes of methods that we are not allowed to distinguish statistically.

- We show that the variance of different performance measurements can be so significant that the optimal solution may not be the best one in practice. Our results indicate that a carefully or ill-chosen compilation setup can trigger a performance gain or loss of factor 1.1 to factor 25 in even the simplest workloads: a table with four attributes and a simple query reading those attributes.
- We highlight, that besides the compilation setup, the data layout is another source of query time variance. Various size metrics of the memory subsystem are round numbers in binary, or put more simply: powers of 2 in decimal. System engineers have followed this tradition over time. Surprisingly, there exists a use-case in query processing where using powers of 2 is always a suboptimal choice, leading to one more cause of fragile query times. Using this finding, we will show how to improve tuple-reconstruction costs by using a novel main-memory data-layout.

Publications

Endre Palatinus, Jens Dittrich
Runtime Fragility in Main Memory
Proceedings of the 2016 Joint Workshop on Accelerating Analytics and In-Memory Data Management Systems

1.4.2 Fragility of Compiling a Database System

High performance database systems are typically written in a compiled programming language, most of the times in C/C++, with few notable exceptions that are written in an interpreted or JIT-compiled language. The previous systems are compiled into machine code specific to a target system (hardware and OS) using a compiler. The compiler is considered as just another abstraction layer in the software development pipeline, and is either used ”as is”, or with a fixed setting. Any possible interaction between the compiler settings and the target system and use case are neglected.

Research Questions

Many high-performance database systems are pre-compiled software. Compiling the system yourself is seen as yet another step in the software development pipeline, and is typically not considered as a performance factor. However, if a developer would like to tune the compiler setup of a database system, he might end up finding the compiler having more tuning knobs than the database system itself. Thus, there are many open questions about compiling a whole database system:

- Does changing the compiler settings have only a negligible effect on the query performance of a whole database system?
- Can we improve query performance by choosing a suitable compiler setup system-wide?
- Can we further improve query performance by choosing for each query the most suitable compiler settings?
- Do our findings still hold if we use another machine?
- Ultimately, can we consider the compiler as just another abstraction layer?

Contributions

In this work we present an exhaustive experimental study on compiling the MonetDB database system. Our main contributions are as follows:

- Given the large number of tuning knobs of compilers we first discuss the ones that could be the most important starting points of performance tuning when compiling a whole database system, namely: i) the compiler itself, ii) the optimization level, and iii) advanced compilation modes.
- For the above categories we consider: i) the three most popular C/C++ compilers (GCC, clang, and the Intel C/C++ compiler), ii) all five standard O-levels, and iii) link-time optimization (LTO), and profile-guided optimization (PGO).
- We consider all combinations of the above knobs, which we will call compiler setup, and build 90 separate MonetDB instances using each of them. We then measure the query performance of the resulting system instances on the TPC-H benchmark.
- We show the differences in the efficiency of the compiler setups when using a different machine. Here we consider six servers equipped with CPUs of subsequent generations.
- We present two techniques for improving query performance that build upon changing compiler setups. These approaches work on two different levels: on a per-query level, and on the physical database operator level.

Publications

Endre Palatinus, Jens Dittrich
90 Shades of Compiling a Main-Memory Column Store
Manuscript in preparation.

Chapter 2

Vertical Partitioning for Legacy Row Stores

Vertical partitioning is a crucial step in physical database design in row-oriented databases. A number of vertical partitioning algorithms have been proposed over the last three decades for a variety of niche scenarios. In principle, the underlying problem remains the same: decompose a table along its attributes into one or more vertical partitions. However, it is not clear how good different vertical partitioning algorithms are in comparison to each other. In fact, it is not even clear how to experimentally compare different vertical partitioning algorithms. In this chapter, we present an exhaustive experimental study of several vertical partitioning algorithms.

We categorize vertical partitioning algorithms along three dimensions: search strategy, starting point, and candidate pruning. We survey six vertical partitioning algorithms and discuss their pros and cons. We identify the major differences in the use-case settings of the investigated algorithms and describe how to make an apples-to-apples comparison in general of different vertical partitioning algorithms under the same setting. We propose four metrics to compare vertical partitioning algorithms: optimization time, quality of solution, fragility of solution, and optimal target setting. We show experimental results from the TPC-H and SSB benchmark and present four key lessons learned: (1) we can do four orders of magnitude less computation and still find the optimal layouts, (2) the benefits of vertical partitioning depend strongly on the database buffer size, (3) HillClimb is the best vertical partitioning algorithm, and (4) vertical partitioning for TPC-H-like benchmarks can improve over column layout by only up to 5%.

The results of this chapter have been accepted for publication in PVLDB, a peer-reviewed journal [36]. Our implementation of the vertical partitioning algorithms have been released at `https://github.com/palatinuse/database-vertical-partitioning`.

2.1 Introduction

2.1.1 Background

Vertical partitioning is a physical database design technique to partition a given logical relation among its attributes into a set of physical tables. This is a common design step with analytical workloads in traditional as well as in modern data management systems such as HBase [29], Vertica [61], Hadoop++ [37], and HYRISE [26]. The basic purpose is to improve the I/O performance of disk-based systems. More formally this is an optimization problem with the goal of finding a complete and (disjunct) partitioning of the set of attributes of a table such that it is optimal with respect to a cost function.

For instance, consider the TPC-H `PartSupp` table and the following query workload:

```
Q1: SELECT PartKey, SuppKey, AvailQty, SupplyCost
    FROM PartSupp;
Q2: SELECT AvailQty, SupplyCost, Comment
    FROM PartSupp;
```

For such a workload, we could choose to partition `PartSupp` into three vertical partitions: $\texttt{P}_1$(`PartKey, SuppKey`), $\texttt{P}_2$(`AvailQty, SupplyCost`), and $\texttt{P}_3$(`Comment`). Now Q_1 accesses partitions $\texttt{P}_1$ and $\texttt{P}_2$, while Q_2 accesses partitions $\texttt{P}_2$ and $\texttt{P}_3$. Thus, both Q_1 and Q_2 read only the required attributes and this improves the I/O performance of these queries.

Vertical partitioning not only improves the query I/O performance, but also strongly affects other physical design decisions such as compression and indexing, as well as query processing techniques such as parallel and distributed query processing. Thus, it is no surprise that vertical partitioning has been researched extensively in the past with researchers proposing a plethora of approaches [30, 27, 46, 17, 47, 18, 15, 28, 51, 4, 26, 37, 35]. As special cases, two extremes of vertical partitioning are traditionally more popular, namely: (i) full vertical partitioning (i.e. column layouts) and (ii) no vertical partitioning (i.e. row layouts).

2.1.2 Effects of Vertical Partitioning

Let us now understand the effects of vertical partitioning on database design decisions in more detail. The major trade-off in vertical partitioning is the row size of partitions: large row sized (wide) partitions resemble row layout, while smaller row sized (narrow) partitions are more similar to column layout. Below, we briefly discuss the major pros and cons of vertical partitioning by contrasting the wide and narrow vertical partitions. Note that in this work we are considering the majorly used row-oriented database systems and how to boost their performance using vertical partitioning. Of course, the other alternative could be using a different system, e.g. column store, in order to boost performance. However, replacing the existing, typically row-oriented, database system is not possible in many situations due to legacy reasons.

Bandwidth. The width of vertical partitions has a considerable effect on I/O bandwidth, and hence on query performance. Wide vertical partitions force the queries referencing fewer attributes to additionally read the accompanying attributes in the partition. For example, for queries Q_1 and Q_2 above, if we split `PartSupp` into the following two vertical partitions: $\texttt{P}_4$(`PartKey`, `SuppKey`, `AvailQty`, `SupplyCost`) and $\texttt{P}_5$(`Comment`) then query Q_2 is forced to read attributes `PartKey` and `Suppkey` in addition to `AvailQty`, `SupplyCost`, and `Comment`. These additional reads affect the I/O bandwidth of Q_2. In the extreme case, if all attributes are put together into a single vertical partition (which yields a row layout), then all except the referenced attributes are read unnecessarily.

Robustness. On the other hand, wide vertical partitions produce predictable query run times, because the majority of queries would have to touch the same number of partitions. For example, for queries Q_1 and Q_2 above, if we keep all attributes of table `PartSupp` in a single vertical partition (i.e. row layout) then both queries Q_1 as well as Q_2 have the same I/O performance, since they both access all five attributes. Scan-only systems such as [55] are examples of such robust query processing systems.

Joins. Narrow vertical partitions penalize queries referencing lots of attributes. This is because the queries need to touch multiple vertical partitions. For example, for the workload in Section 2.1.1, if we split table `PartSupp` into three vertical partitions: $\texttt{P}_1$(`PartKey`, `SuppKey`), $\texttt{P}_2$(`AvailQty`, `SupplyCost`), and $\texttt{P}_3$(`Comment`) then query Q_2 must touch partitions $\texttt{P}_2$ and $\texttt{P}_3$. With this, the database engine needs to reconstruct the tuples from the referenced vertical partitions using tuple reconstruction joins. Since each vertical partition is stored as a separate physical table, these tuple reconstruction joins could be pretty expensive: they can negatively affect the query plans and incur CPU overheads.

Random I/O. Tuple reconstruction joins in narrow vertical partitions incur very high random I/O costs. This is because all referenced vertical partitions must be read into the database buffer at the same time for tuple reconstruction. For this to happen, the database buffer must be split into *sub-buffers* for each referenced vertical partition. As a consequence, now we have random I/Os each time any of the sub-buffers needs to be filled. For instance, Q_1 has twice the number of random

I/Os for partitions $\texttt{P}_1$(PartKey, SuppKey) and $\texttt{P}_2$(AvailQty, SupplyCost) than for partition $\texttt{P}_4$(PartKey, SuppKey, AvailQty, SupplyCost).

2.1.3 Choosing a Vertical Partitioning Algorithm

Vertical partitioning strongly affects the query performance in many ways, as discussed above. A number of vertical partitioning algorithms have been proposed by several researchers over time [30, 27, 46, 17, 47, 18, 15, 28, 51, 4, 26, 37, 35]. As a result, users now have the problem of choosing a vertical partitioning algorithm. In contrast to physical design tools, which choose a layout given a vertical partitioning algorithm, the problem here is to choose the vertical partitioning algorithm in the first place. Essentially, the questions that we are looking at are:

- Which are the major algorithms proposed? What is the difference between those algorithms?
- For which settings were different algorithms proposed? What are their pros and cons?
- What are the primary differences between different vertical partitioning settings? Can we abstract the settings from the algorithms?
- How do we compare different algorithms in a common setting? What would be the right measures for comparison?
- How do the different algorithms compare against each other? When to use which algorithm?

Thus, there is an absence of a systematic and comparative study of vertical partitioning algorithms. This work fills this gap.

2.1.4 Contributions

In this chapter, we present an exhaustive experimental study on vertical partitioning algorithms. Our main contributions are as follows:

- Given the large number of vertical partitioning algorithms proposed in the literature, we first understand the fundamental differences between them. To do so, we first classify them along three dimensions, namely: (i) search strategy, (ii) starting point, and (iii) candidate pruning (Section 2.2).

- From the above categories, we survey six representative vertical partitioning algorithms, namely: (i) AutoPart [51], (ii) HillClimb [28], (iii) HYRISE [26], (iv) Navathe's algorithm [46], (v) O_2P [35], and (vi) Trojan layouts [37]. We present a brief summary and the context of each of the algorithms (Section 2.3).

- We describe how the different vertical partitioning algorithms can be applied in the same setting. Even though each algorithm was proposed for a different setting, we can still unite them under a common umbrella (Section 2.4).

- We present a systematic way of comparing different vertical partitioning algorithms. For this purpose, we introduce four metrics, namely: (i) *how fast* in terms of computation times, (ii) *how good* in terms of workload runtimes, (iii) *how fragile* in terms of predictable runtimes, and (iv) *where does it make sense* to use vertical partitioning (Section 2.5).

- We show detailed experimental results from six vertical partitioning algorithms over the TPC-H benchmark and with row and column layouts as baselines. We discuss each of the four metrics for the six vertical partitioning algorithms (Section 2.6).

- Finally, we discuss the 4 key lessons learned (Section 2.7).

2.2 Classification of Vertical Partitioning Algorithms

There are several vertical partitioning algorithms proposed in the literature. Instead of simply listing them, it would be more interesting to see the major differences between the core ideas of those algorithms. To do this, we categorize the vertical partitioning algorithms along three dimensions based on the way they attack the vertical partitioning problem. Table 2.1 shows the classification of the evaluated vertical partitioning algorithms. We describe each of these dimensions and categories in the following.

2.2.1 Search Strategy

First of all, we differentiate different vertical partitioning algorithms based on their search strategy in the solution space.

Brute Force. Algorithms in this category follow the naive approach of enumerating all possible vertical partitionings and picking the one giving the best

		AutoPart	HillClimb	HYRISE	Navathe	O2P	Trojan	Brute Force
Starting Point	Whole workload				🟩	🟩		🟩
	Attribute subset	🟩	🟩	🟩				
	Query subset						🟩	
Search Strategy	Brute force							🟩
	Top-down				🟩	🟩		
	Bottom-up	🟩	🟩	🟩			🟩	
Candidate Pruning	No pruning	🟩	🟩	🟩	🟩	🟩		🟩
	Threshold-based						🟩	

Table 2.1: Classification of the evaluated vertical partitioning algorithms.

estimated query performance. In this way, a brute force algorithm computes the best possible vertical partitioning over a given query workload and cost model. Unfortunately, the number of possible vertical partitionings grow dramatically with the number of attributes. For instance, for the 16 attributes of the TPC-H Lineitem table, the number of possible vertical partitionings is 10.5 million. Therefore, brute force is not a practical approach for large number of attributes.

Top-down. Algorithms in this category start from the set containing all attributes and try to break it into smaller and smaller subsets. The idea is to assume *no-vertical-partitioning*, i.e. row layout, as the ground truth and to improve upon it as much as possible. The improvement is usually measured in terms of the expected cost of a query workload (using a cost model). Early vertical partitioning algorithms of Navathe [46, 47] were based on this approach. As the starting point the attributes are arranged in some *order*, e.g. an ordered sequence in [46] or a connected graph in [47]. Typically, there is a preparatory step which determines this order, e.g. attribute affinity matrix clustering in [46]. Thereafter, the attribute set is recursively (and greedily) divided into smaller subsets till no improvement in the expected workload costs is seen. Every split step preserves the initial ordering of the attributes. Inspired from those early works, a recent algorithm, O_2P [35], does online vertical partitioning using the top-down approach. The vertical partitioning algorithms in the top-down category converge faster for highly regular attribute access patterns, i.e. lots of queries accessing almost the same attributes. This is because only few splitting steps are required. On the other hand, top-down algorithms consider vertical partitions incrementally. This means that for any vertical partition to appear in the final solution, its supersets must appear in all previous iterations. This might not happen in many situations.

Bottom-up. In contrast to top-down, the bottom-up approach starts with minimally small vertical partitions. All algorithms in this category define the latter property of a partition differently. The underlying assumption is that it does not make sense to sub-divide these initial vertical partitions into smaller vertical partitions. The idea then is to recursively merge the vertical partitions into bigger partitions as long as there is an improvement in expected query costs. Three main algorithms fall into this category: Chu and Ieong's algorithm [15], HillClimb [28], and AutoPart [51]. As the preliminary step, the algorithms produce the set of minimally small vertical partitions. These can be partitions containing only a single attribute (column layout), as in HillClimb [28], or the set of *primary partitions*, which are partitions containing attributes that are always accessed together in all queries, as in Chu and Ieong's algorithm and AutoPart [15, 51]. Thereafter, the algorithms recursively consider merging two or more partitions. Additionally, AutoPart [51] also creates overlapping partitions, i.e. partitions having one or more attributes in common, thereby allowing for partial replication of attributes. The bottom-up algorithms converge faster for highly fragmented attribute access patterns, i.e. queries accessing little or no attributes in common. This is because after a few merge steps the query costs will not improve any more. Similar to the top-down class, the bottom-up algorithms consider vertical partitions incrementally, i.e. greedily. For bottom-up algorithms this means that for any vertical partition to appear in the final solution, its subsets must appear in all previous iterations.

2.2.2 Starting Point

Apart from the search strategy, different vertical partitioning algorithms may have different starting points. For example, an algorithm may start with only a subset of the attributes or with only a subset of the workload queries. This is an important consideration because it helps to first sub-divide the vertical partitioning problem into smaller problems and find the solution to each of them.

Whole workload. Algorithms in this category neither divide the queries nor the attributes at the start.

Attribute subset. Algorithms in this category compute vertical partitioning for a subset of the attributes. For example, HYRISE [26] first sub-divides attribute sets into groups using a k-way partitioner and then computes the vertical partitioning for each group using a top-down algorithm. Finally, to produce the final solution, HYRISE [26] combines the solutions from different sub-problems. Computing vertical partitioning for attribute subsets reduces the complexity of the algorithm dramatically. However, such algorithms find the solution for each subset locally and have to later merge them, which might result in a suboptimal solution for the global problem.

Query subset. Algorithms in this category compute vertical partitioning for only a subset of the queries in the workload. For example, Trojan [37] first sub-divides the workload into query groups depending on the similarity between queries and finds the layout for each query group using a bottom-up algorithm. It is easier to find vertical partitioning for query subsets, since they are likely to have more similar access patterns, and hence the algorithm converges quickly. Trojan [37] does not combine the solutions from different query subsets, as it creates multiple vertical partitionings, one for each dataset replica. Starting from query subsets is a very practical approach because typical workloads contain several classes of queries, each having very similar access patterns.

2.2.3 Candidate Pruning

Finally, vertical partitioning algorithms may also prune the vertical partitioning candidates in order to reduce the search space.

No pruning. Most algorithms considered in this work do not apply pruning to the search space, but generate possible solutions in each iteration excluding locally sub-optimal ones.

Threshold-based. Algorithms with threshold-based pruning prune the input set based on some heuristics. For example, the algorithm of Agrawal [4] and Trojan [37] prune the set of column groups based on their interestingness, which denotes how well a given column group speeds up the queries. The complexity of these algorithms therefore depends on the effectiveness of their pruning threshold. Threshold-based pruning algorithms face one basic problem: the algorithm needs to generate all candidates before actually pruning them. This could be pretty expensive and hence slow. On the flip side, however, threshold-pruning approach sees the global picture (not local or incremental) and hence is expected to produce better results.

2.3 Evaluated Algorithms

In this work, we cover a wide range of representative vertical partitioning algorithms from the early state-of-the-art to the most recent ones. We have chosen these algorithms to cover all categories and include the earliest vertical partitioning algorithm as well as five other recent vertical partitioning algorithms published in the last decade. Below we describe each of these algorithms.

Brute Force. The set of all possible vertical partitionings of a table can be enumerated using a brute force algorithm. The total number of vertical partitionings of a set that has exactly n elements are given by the n^{th} Bell number [8]:

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k, \text{ and } B_1 = B_0 = 1$$

For example, for the TPC-H customer table, having eight attributes, the number of possible vertical partitionings is $B_8 = 4\,140$. On the other hand, for the TPC-H lineitem table, which has 16 attributes, the number of vertical partitionings is as high as 10 480 142 147, which makes it computationally rather intensive to search for the optimal solutions using brute force.

Bell numbers can also be represented as a sum of *Sterling numbers*:

$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$$

where the Sterling number of the second kind $\left\{ {n \atop k} \right\}$ gives the number of ways to partition n attributes into exactly k partitions. They obey the recurrence relation:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \cdot \left\{ {n-1 \atop k} \right\}, \text{ and } \left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1.$$

Here is how this equation can be explained: a partitioning of the n attributes into k nonempty subsets either contains the n-th attribute as a singleton or it does not. The number of ways that the singleton is one of the subsets is given by $\left\{ {n-1 \atop k-1} \right\}$ since we must partition the remaining $n-1$ attributes into the available $k-1$ subsets. In the other case the n-th object belongs to a subset containing other objects. The number of ways is given by $k \cdot \left\{ {n-1 \atop k} \right\}$ since we partition all attributes other than the n-th into k subsets, and then we are left with k choices for inserting attribute n.

Inspired by the representation of Bell numbers as a sum of Sterling numbers of the second kind we can come up with an algorithm to *enumerate* all possible vertical partitionings. *Note:* an empty vertical partitioning does not make sence, thus we neglect the $k = 0$ cases. The pseudocode of the algorithm is shown in Listing 1. The complexity of this brute force vertical partitioning algorithm is $O(n^n)$ (for n attributes).

Navathe. One of the earliest approximation-based approaches to vertical partitioning was proposed by Navathe et al [46]. This is a top-down algorithm and focuses primarily on disk-based systems. The core idea of this algorithm

Algorithm 1 The Brute Force Algorithm

```
1: procedure BellNumbersBasedEnumeration($n$)
2:     for $k \leftarrow 1$ to $n$ do
3:         StirlingNumbersBasedEnumeration($n$, $k$, new int[$n$])

4: procedure StirlingNumbersBasedEnumeration($n$, $k$, $a$)
                                      ▷ $n$ is the number of (unpartitioned) attributes
                                            ▷ $k$ is the desired number of partitions
                              ▷ $a$ is the (partial) mapping of attributes to partitions
      ▷ $a[1 \dots n]$ are unmapped, while $a[n+1 \dots \text{length}(a)]$ are already mapped

5:     if $k = 1$ then
6:         for $i \leftarrow 1$ to $n$ do
7:             $a[i] \leftarrow k$                ▷ All attributes go into a single partition
8:         EmitPartitioning($a$)
9:     else if $k = n$ then
10:        for $i \leftarrow 1$ to $n$ do
11:            $a[i] \leftarrow i$       ▷ All attributes go into a different singleton partition
12:        EmitPartitioning($a$)
13:    else
14:        $a[n] \leftarrow k$              ▷ Put the $n$-th attribute into a singleton partition
15:        StirlingNumbersBasedEnumeration($n - 1$, $k - 1$, $a$)
16:        for $i \leftarrow 1$ to $k$ do
17:            $a[n] \leftarrow i$                    ▷ Put the $n$-th attribute into partition $i$
18:            StirlingNumbersBasedEnumeration($n - 1$, $k$, $a$)
```

is as follows. Given a set of attributes and a set of queries referencing those attributes, the algorithm constructs an *attribute affinity matrix.* Cell (i, j) of the attribute affinity matrix denotes the number of times attribute i co-occurs with attribute j (also called their *affinity*). Thereafter, the algorithm clusters the cells of the matrix such that attributes with higher affinity are close together. The authors propose to use the bond energy algorithm [39] for matrix clustering. After that, the algorithm splits the clustered set of attributes into vertical partitions recursively.

HillClimb. The HillClimb algorithm is a bottom-up algorithm proposed in the early 2000s [28]. This algorithm focuses on data layouts within a data page. It proceeds as follows. It starts with column layout, i.e. each attribute resides in a different vertical partition. Thereafter, in each iteration, the algorithm finds and merges two partitions which, when merged, provide the best improvement in expected query costs. This means that in each iteration the number of vertical partitions is reduced by one. The algorithm stops iterating when there is no improvement in expected query costs. To facilitate computing the expected query costs, the algorithm pre-computes and maintains a dictionary of the costs of all possible vertical partitions (or column groups). However, the size of such a dictionary grows quickly to several gigabytes in case the number of attributes is large. As a result, we have found that the runtime of the algorithm can be dramatically improved without maintaining such a dictionary. Thus, we used this improved version of HillClimb.

AutoPart. The AutoPart is a bottom-up algorithm introduced in 2004 to compute vertical partitionings over large scientific datasets [51]. First, AutoPart *categorically* partitions the table horizontally (based on selection predicates), such that each horizontal partition is accessed by a different subset of queries. Thereafter, AutoPart finds a vertical partitioning for each horizontal partition. As a starting point, AutoPart generates the set of primary partitions (called *atomic fragments*). A vertical partition is atomic if all queries accessing it, reference all attributes in the partition. In other words, there are no queries which access a subset of an atomic fragment. Thereafter, in each iteration, the fragments are extended by either combining them with atomic fragments or with fragments from the previous iteration. The process is repeated till there is no improvement in estimated costs of the query workload. Note that an attribute may occur in multiple fragments (i.e. replicated) when combined. Thus, it might be possible that multiple partition combinations are now suitable to answer a given query. In such a case, we need to select the partitions to read. It turns out that partition selection is as difficult a problem as vertical partitioning itself.

HYRISE. The HYRISE is a multi-level algorithm proposed in 2010 to compute vertical partitionings for main-memory resident data processing systems [26]. In contrast to disk-based systems, the goal here is to minimize the number of cache misses. In the first step, the algorithm generates the set of *primary partitions*, which are the same as the atomic fragments in AutoPart, i.e. sets of attributes that are always accessed together. Then, the algorithm builds an affinity graph for the primary partitions, where primary partitions are represented as nodes and the co-accessed frequency of two primary partitions as edge weights. HYRISE then partitions this graph such that each sub-graph contains at most K primary partitions. This is done using a K-way graph partitioner. Thereafter, HYRISE finds the layout for each sub-graph separately. In each iteration, the primary partitions (belonging to the same sub-graph) which give the maximum cost improvement are merged. The merged partition replaces the primary partitions and the process is repeated until there is no more improvement in cost. As the final step, HYRISE tries to combine the vertical partitions obtained from different sub-graphs.

O$_2$P. One-dimensional online partitioning (O$_2$P) is a top-down algorithm proposed in 2011 with the focus on real time partitioning [35]. The goal is to determine a vertical partitioning in an online setting, i.e. *while* the query workload is being executed. It starts from Navathe's algorithm and transforms it into an online vertical partitioning algorithm. To do so, it dynamically updates as well as clusters the affinity matrix for each incoming query. This is done by adapting the bond energy algorithm [39], used in Navathe, to an online setting. To compute the vertical partitioning, O$_2$P employs a greedy approach to create one (the best) new vertical partition in each step. It also uses dynamic programming to remember the costs of non-best vertical partitions from the previous step. These two techniques make the partitioning analysis in O$_2$P extremely fast and hence suited for an online setting.

Trojan. The Trojan layouts algorithm was proposed in 2011 to create vertical partitioning for big data [37]. It is a threshold-pruning based algorithm. Unlike previous algorithms, it considers large data block sizes and existing data block replication, both being a reality for big data. As the first step it enumerates all possible column groups and keeps only the ones that are *interesting*. It introduces a novel interestingness measure for column groups, based on the mutual information between the attributes of a column group. The algorithm prunes all column groups whose interestingness fall below a certain threshold. The interesting column groups are then merged into a *complete* (i.e. containing all attributes) and *disjoint* (i.e. not containing any attribute twice) set of vertical partitions. This is done by mapping vertical partitioning to a 0-1 knapsack problem. The Trojan algorithm works especially well with data replication, such as found in HDFS. To take into account the default data replication in HDFS, it first groups queries and maps each query group to a different data replica. It uses the same column grouping algorithm for query grouping as well. Then, for each query group, it computes the column groups independently.

		AutoPart	HillClimb	HYRISE	Navathe	O2P	Trojan
Granularity	FILE	🟩		🟩	🟩	🟩	
	DATA PAGE		🟩				
	DATABASE BLOCK						🟩
Hardware	HARD DISK	🟩	🟩		🟩	🟩	🟩
	MAIN MEMORY			🟩			
Workload	OFFLINE	🟩	🟩	🟩	🟩		🟩
	ONLINE					🟩	
Replication	NONE		🟩	🟩	🟩	🟩	
	FULL						🟩
	PARTIAL	🟩					
System	CUSTOM		🟩	🟩			
	COST MODEL	🟩			🟩		
	OPEN SOURCE					🟩	🟩

Table 2.2: Settings for different vertical partitioning algorithms.

2.4 Methodology

The vertical partitioning algorithms described in the previous section have all been proposed for different scenarios and under different settings. In the following, let us try to understand the major differences between them.

(1.) *Granularity.* Different algorithms are targeted for different data granularity, such as data page, e.g. HillClimb, database block, e.g. Trojan, and file, e.g. AutoPart.

(2.) *Hardware.* The algorithms can optimize for different hardware, such as hard disk, e.g. Navathe, and main-memory, e.g. HYRISE.

(3.) *Workload.* The algorithms may work with different assumptions for the query workload. We can consider a fixed set of queries (offline workload), e.g. AutoPart, or a dynamically growing set of queries (online workload), e.g. O_2P.

(4.) *Replication.* An algorithm may or may not consider data replication. Even if the algorithm considers replication, it may either consider replicating all attributes (full replication), e.g. Trojan, or only a subset of the attributes (partial replication), e.g. AutoPart.

(5.) *System.* Different algorithms are proposed in different implementations of data management systems, e.g. Hadoop (Trojan), BerkeleyDB (O_2P), main-memory implementation (HYRISE), etc.

Table 2.2 classifies the six algorithms (from Section 2.3) analyzed in this work according to their granularity, hardware-, workload-, and replication characteristics. We can see that no two algorithms have the same combination of these parameters. It seems that the different vertical partitioning algorithms use quite different configurations even though they have exactly the same underlying functionality: decompose a table into vertical partitions. In order to have an apples-to-apples comparison, we use the same configuration for all vertical partitioning algorithms. Our common settting is marked with green background in the second column of Table 2.2. To the best of our knowledge, this is the first work to survey vertical partitioning algorithms under a common configuration. Essentially this means that we strip the granularity, hardware-, workload-, and replication characteristics from the different vertical partitioning algorithms, leaving just the core vertical partitioning functionality. Below we describe the common configuration used in our experiments.

Common Granularity. For all algorithms, we consider the storage layout of the vertically partitioned table to be as follows: the table is split into one or more vertical partitions (column groups), which are stored in separate files. Thus, each data page contains data from only a single vertical partition. At query time, we assume that the database system does the following: read all vertical partition's files which contain any of the attributes referenced by the incoming query. This means that even if a query references only some of the attributes in a vertical partition, we still need to read all attributes in the vertical partition's file.

Common Hardware. We use the following common testbed for all algorithms: a single node machine with a quad-core Intel Xeon 5150 processor running at 2.66 GHz with 4 MB L2 cache, having 16 GB RAM and 1.5 TB HDD, running OpenSuse 12.1 64 bit. We consider the commonly used disk-based storage when evaluating vertical partitioning algorithms. We measured the disk characteristics of our testbed using Bonnie++ [12]. We obtained the following results: a disk read bandwidth of 90.07 MB/s, disk write bandwidth of 64.37 MB/s and average disk seek time of 4.84 ms.

Common Workload. We consider read-only analytical applications for comparing different vertical partitioning algorithms. To do so, we take the query workload from the widely used TPC-H benchmark, and assume a scale factor of 10. We partition each table in TPC-H separately, as done by other researchers [15]. We take all 22 queries from the TPC-H benchmark. However, we consider only scan and projection query operators. This is because in our cost model, we model only the I/O costs for accessing the data, while excluding the query execution costs. We do this for two reasons. First, almost all vertical partitioning algorithms consider only scan and projection operators. Since we are doing a comparative study of different algorithms, we consider the same set of operators for all algorithms. Extending these algorithms to consider other operators, such as selection, will be an interesting future work. Second, the overall query execution costs make sense only when all physical design decisions, including indexes and materialised views, are considered. Instead, in this work, we are focussing on vertical partitioning and hence we want to isolate the impact of vertical partitioning created by different algorithms. Furthermore, overall query execution costs depend heavily on the query optimizer and executor of the database system and hence it is not possible to model them in a general setting.

Common Replication. AutoPart and Trojan make use of partial and full data replication respectively. However, in order to make a fair comparison, we would need to tweak other algorithms to allow for data replication as well. Moreover, data replication adds several new dimensions for consideration. These include storage space constraints, read versus update performances, and most importantly picking the right replica at query time. Hence, we believe that vertical partitioning with data replication requires an independent exhaustive study, which is beyond the scope of this work. Instead, in this work, we limit to no data replication.

Common System. We evaluate all algorithms using the estimated costs from our I/O cost model. We do this for two reasons. First, as discussed before, we focus on the I/O costs of queries in order to understand the effects of vertical partitioning in row-oriented database systems. Second, database systems typically create a different table for each vertical partition and later use joins for tuple reconstruction. This makes running just the leaf plans (in order to see the I/O costs) very expensive because no operators can be pushed down and we end up with high join cardinalities. As a result, the I/O costs are overshadowed by the join processing costs. In a recent work [38], UDFs were exploited to store and access data in column layouts without performing a join, i.e. we simply *merge* the columns. However, this works only for highly selective queries, since for low selectivities the UDF call overhead shadows the performance gain due to a different layout. To the best of our knowledge, there is no freely available database system which queries vertically partitioned data without performing table joins.

Our system assumes buffered read- and write mechanisms for transferring data between disk and memory. This means that at query time we read all vertical partitions which contain any of the attributes referenced by the incoming query into an I/O buffer (say of size *Buff*). In our experiments we assume per-tuple query processing, i.e. the database system passes data tuple-by-tuple to the query executor. For this, the database system needs to reconstruct the tuples *while* reading the vertical partitions. To do so, we will require to buffer-read the vertical partitions at the same time. This means that we have to share the I/O buffer among the multiple vertical partitions being read. In our cost model, we share the I/O buffer in proportion to the tuple size of the vertical partitions being read. If S is the total row size of all referenced partitions and s_i is the row size of vertical partition i, then the I/O buffer allocated to partition i is given as:

$$\mathrm{buff}_i = \left\lfloor \mathrm{Buff} \cdot \frac{\mathrm{s}_i}{\mathrm{S}} \right\rfloor .$$

Given block size b, the number of blocks that can be read at a time into the buffer for partition i are:

$$\mathrm{blocks}_i^{\mathrm{buff}} = \left\lfloor \frac{\mathrm{buff}_i}{\mathrm{b}} \right\rfloor .$$

If the table has `N` rows, the total number of blocks on disk for partition i are:

$$\mathrm{blocks}_i = \left\lceil \frac{\mathrm{N}}{\left\lfloor \frac{\mathrm{b}}{\mathrm{s}_i} \right\rfloor} \right\rceil .$$

Assume that we have to perform a seek every time the I/O buffer for partition i needs to be filled. Then the number of times the I/O buffer gets full determines the seek cost of reading partition i. Given an average seek time t_s of the disk, the seek cost of reading partition i is given as:

$$\mathrm{cost}_i^{\mathrm{seek}} = t_s \cdot \left\lceil \frac{\mathrm{blocks}_i}{\mathrm{blocks}_i^{\mathrm{buff}}} \right\rceil .$$

On the other hand, the scan cost of partition i is determined by the total number of blocks of partition i to be read. Given disk bandwidth BW, the scan cost of partition i is given as:

$$\mathrm{cost}_i^{\mathrm{scan}} = \frac{\mathrm{blocks}_i \cdot \mathrm{b}}{\mathrm{BW}} .$$

Finally, for a query q referencing a P_q set of vertical partitions, the total I/O cost is the sum of the seek- and scan costs of all referenced partitions:

$$\mathrm{cost}_q = \sum_{i \in P_q} \left(\mathrm{cost}_i^{\mathrm{seek}} + \mathrm{cost}_i^{\mathrm{scan}} \right) .$$

The total I/O costs of the entire workload will be the sum of the I/O costs of each query in the workload.

2.5 Comparison Metrics

As discussed in the previous section, we apply the same setting to all vertical partitioning algorithms. However, since there is no prior work comparing different vertical partitioning algorithms, it is not clear how to compare them, i.e. the comparison metrics are not defined. The authors of HYRISE compared their algorithm against HillClimb in terms of query costs. However, we believe that other measures such as time taken to compute the layouts are equally important. Thus, in this section, we systematically introduce four comparison metrics for vertical partitioning algorithms and describe them below.

How fast? Vertical partitioning being an NP-hard problem, the first thing that comes to mind is *how fast* is a given algorithm, i.e. how long does it take to come up with a solution. Additionally, the optimization time should be seen in comparison to the table size (or indirectly the layout creation time). For example, if it takes fifteen minutes to create the layouts (i.e., a large table) then it might be acceptable to spend an hour to find the layouts.

How good? Since the goal of a vertical partitioning algorithm is to improve the workload runtime, it is important to know the expected workload runtime. Additionally, it is important to know how much does vertical partitioning improve the workload runtime over row and column layouts. Note that this improvement comes at a price: we need to invest in the optimization and the creation time. Thus, we need to see the time invested (optimization + creation) compared to the expected workload execution cost benefits.

In fact, the ratio of these two quantities gives the fraction (or the multiple) of query workload that we need to execute before the time invested pays off over the workload runtime improvements.

How fragile? Heterogenous hardwares/software settings are common in data centers these days. However, vertical partitioning algorithms can be computationally expensive, therefore it is not possible to recompute them for each and every hardware/software setting. Thus, we need to know how fragile the different vertical partitioning algorithms are over different parameters in the cost model (which models the hardware/software settings). We measure algorithm fragility as the change in workload runtime when there is a change in a cost model parameter. Fragility, thus defined, gives hints on whether or not we should re-run the vertical partitioning algorithm if the hardware/software settings change.

Where does it make sense? The fragility metric above measures how far off is the workload performance, if we optimize vertical partitioning for one cost model and use it over another. However, at the same time, it is also important to know how does the workload performance change if we re-optimize vertical partitioning over different cost models. Thus, we optimize for each new cost model parameter and show the workload performance. This helps us to find the sweet spots for vertical partitioning, i.e. the cost model parameters for which vertical partitioning makes the most sense.

2.6 Simulations and Experiments

We now present the results from the six vertical partitioning algorithms considered in Section 2.3. We implemented all algorithms in Java 6 and tried to keep the implementations as close to the original descriptions as possible. However, we did adapt the algorithms to the unified settings shown in Table 2.2. For example, Trojan was adapted to work without considering data replication. We ran all experiments on the common hardware described in Section 2.4. We organize the results along the four comparison metrics introduced in Section 2.5. We repeated each measurement five times and report the average. We discarded the results of the first five runs to allow for just-in-time compilation in the JVM to complete and use the results of the second five runs. We used cold caches, both for the operating system as well as the hard disk, for all runs.

2.6.1 Comparing Optimization Time

In this section we address the following questions:

How do the algorithms compare in terms of optimization time?

Figure 2.1 shows the optimization times for different vertical partitioning algorithms.

We can see that the fastest algorithm (O_2P) is 5 orders of magnitude faster than BruteForce. Even the slowest algorithm (Trojan) is 2 orders of magnitude faster than BruteForce. Thus, all algorithms find a vertical partitioning solution much faster than BruteForce. The optimization times of AutoPart, HillClimb, HYRISE, Navathe, and O_2P are quite acceptable (at most 5 seconds), however, Trojan and BruteForce have very high optimization times (1.5 minutes and 1 hour, respectively). The time to transform from row layout to vertically partitioned layout for scale factor 10 is around 420 seconds for all algorithms. This means that it takes much longer to transform the layout than it takes to compute the layout using one of the fast algorithms mentioned above.

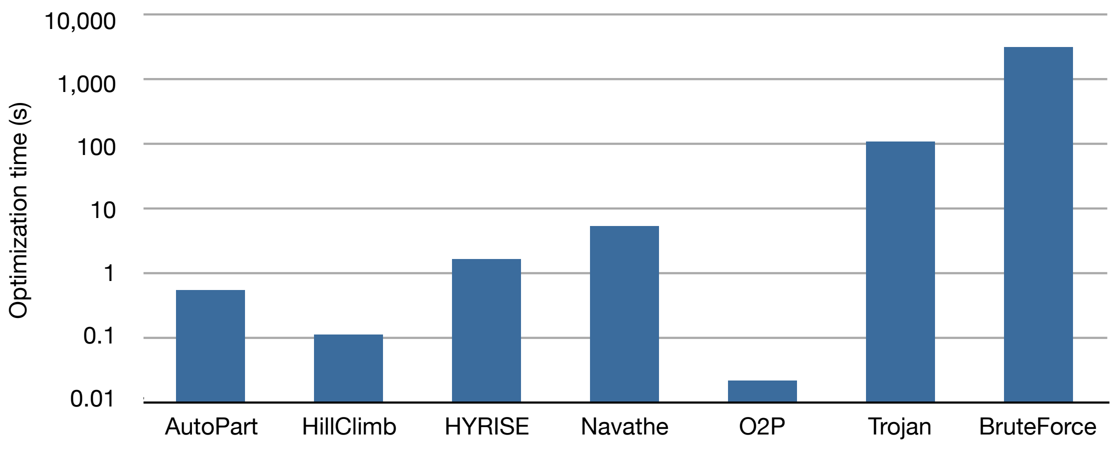


Figure 2.1: Optimization time [log scale] for different algorithms

How do the optimization times change with the workload size?

Let us now see how the optimization times change with the workload size. Recall that, for every vertical partitioning candidate, an algorithm computes the expected cost of each query in the query workload. Thus, we expected higher optimization time for larger query workloads. Figure 2.2 shows the optimization times of the different algorithms over varying workload size. We vary the TPC-H workload size by taking the first k queries, k varying from 1 to 22.

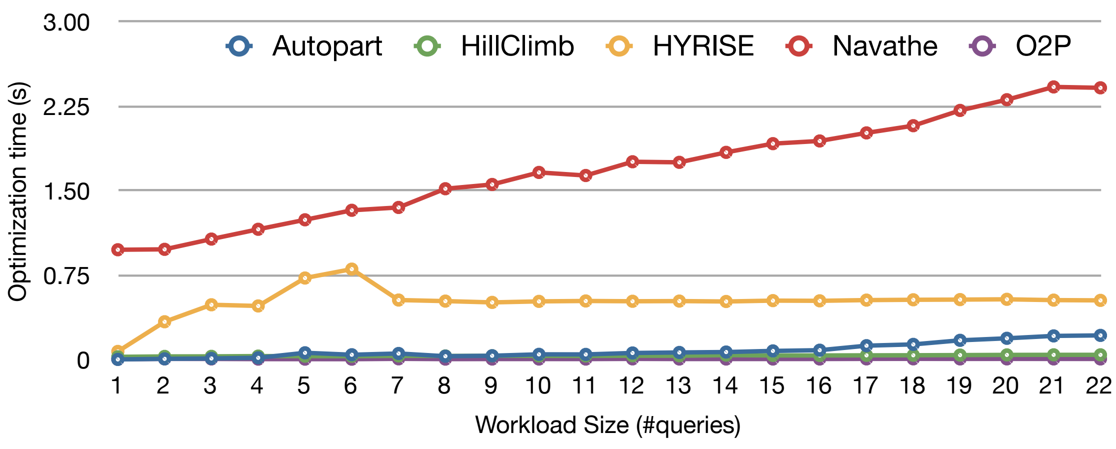


Figure 2.2: Optimization time over varying workload size

We can see that Navathe and AutoPart have a much steeper increase in optimization time in comparison to HYRISE, HillClimb, and O_2P. In general, these algorithms scale well with the workload size. We have excluded Trojan and Brute-

Force in the figure because of their extremely high optimization time (at least 2 orders of magnitude higher than the others), which distorts the graph.

The most important lesson learned in this section is that the optimization time of a vertical partitioning algorithm can be several orders of magnitude less than BruteForce. Still, as we will see in the next section, some algorithms can find the same (optimal) solution as the BruteForce.

2.6.2 Comparing Algorithm Quality

We investigate a series of five questions in this section. Let's start with the following one:

How do algorithms compare in terms of query performance?

Figure 2.3 shows the *estimated workload costs* for all queries of the TPC-H Benchmark — when using the partitionings produced by the different vertical partitioning algorithms. By estimated workload cost we mean the total I/O cost of the entire workload as described in Section 2.4.

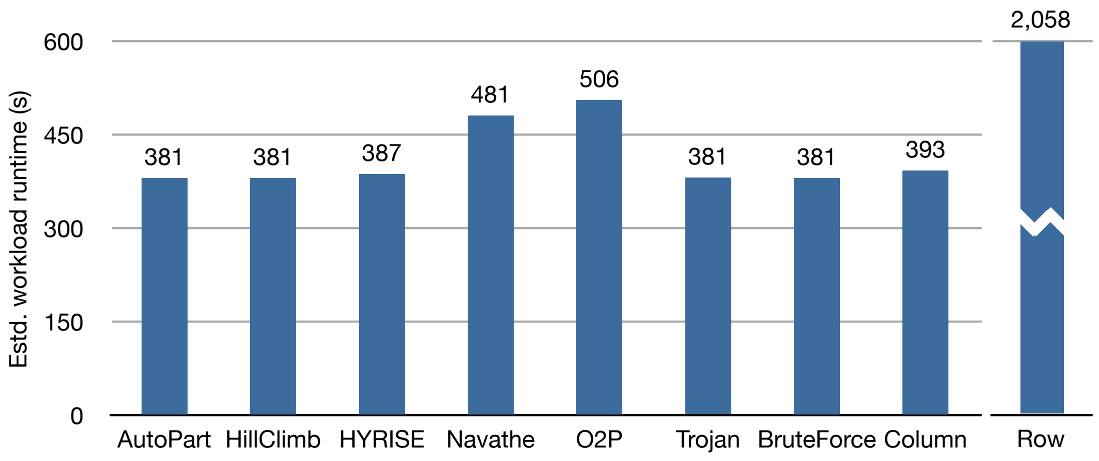


Figure 2.3: Estimated workload runtime for the produced solutions

We can see that except for Navathe and O_2P, all algorithms, including BruteForce, have very similar estimated workload costs. In fact, the layouts produced by AutoPart, HillCimb and Trojan have exactly the same workload cost as that by BruteForce. This is despite HillClimb requiring 5 orders of magnitude less optimization time than BruteForce. As a result, vertical partitioning with HillClimb can payoff for as little as 25% of the TPC-H workload (See Appendix 2.A.1 for details).

Now let us analyze the improvement of vertical partitioning over Row or Column. We can see that the improvement over Row is as high as 80.11%. However, over Column the maximum improvement is only 4.75%. Column even outperforms the vertically partitioned layouts of Navathe and O_2P by 21% and 28%, respectively. This is a surprising result because we expected vertical partitioning to be very effective for analytical workloads. Let us now dig deeper to understand the high improvements over Row and low improvements over Column, by asking the following questions.

What fraction of the data read is unnecessary?

Note that a suitable vertical partitioning improves over Row because it reads less unnecessary data. Figure 2.4 shows the percentage of data read which is unnecessary, i.e. not needed by the queries. The amount of unnecessary data read is calculated as follows:

$$\text{Unnecessary data read} = \frac{\text{Data read} - \text{Data needed}}{\text{Data read}} * 100\%$$

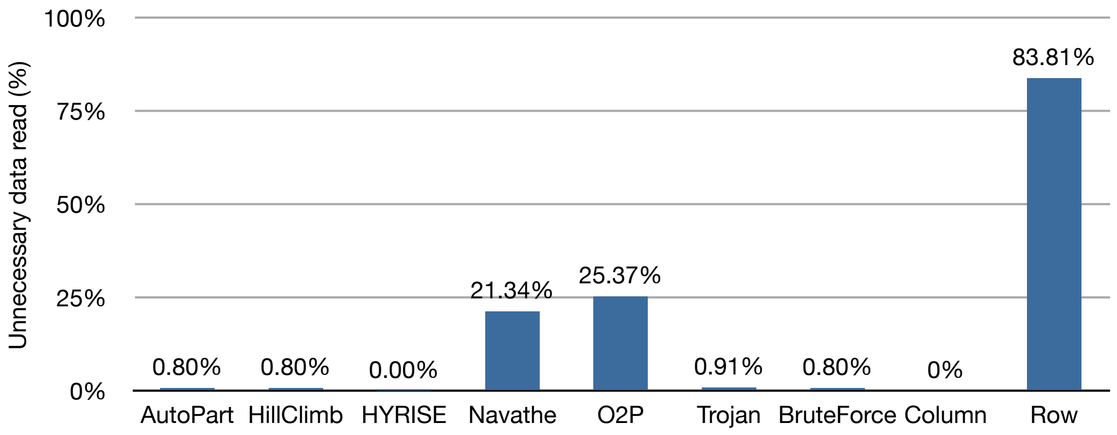


Figure 2.4: Fraction of unnecessary data read

We can see that Row reads 84% unnecessary data and all vertically partitioned layouts have a significant improvement over that. The layouts produced by AutoPart, HillClimb, and BruteForce read only 0.8% unnecessary data, while the layouts from HYRISE do not read *any* unnecessary data. This explains the dramatic improvements over row.

How many tuple reconstruction joins are performed?

Next, let us understand the low improvements of vertical partitioning over Column. Note that a suitable vertical partitioning improves over Column since it performs less tuple reconstruction joins. For each query, the number of tuple reconstruction joins per tuple are given as:

$$\#\text{Tuple-reconstruction joins} = \#\text{Vertical partitions accessed} - 1$$

Figure 2.5 shows the tuple reconstruction joins averaged over all tuples and all queries, when using each of the layouts. Column has to join all attributes referenced by the query. However, vertically partitioned layouts also perform at least 72% of the joins performed by Column. Thus, none of the algorithms produce layouts which would dramatically reduce the tuple reconstruction joins, which increases the number of random I/Os in our cost model, hence the marginal improvement over Column. Note that the above estimated improvements are only in terms of I/O costs. In practice, tuple-reconstruction incurs additional CPU-costs as well.

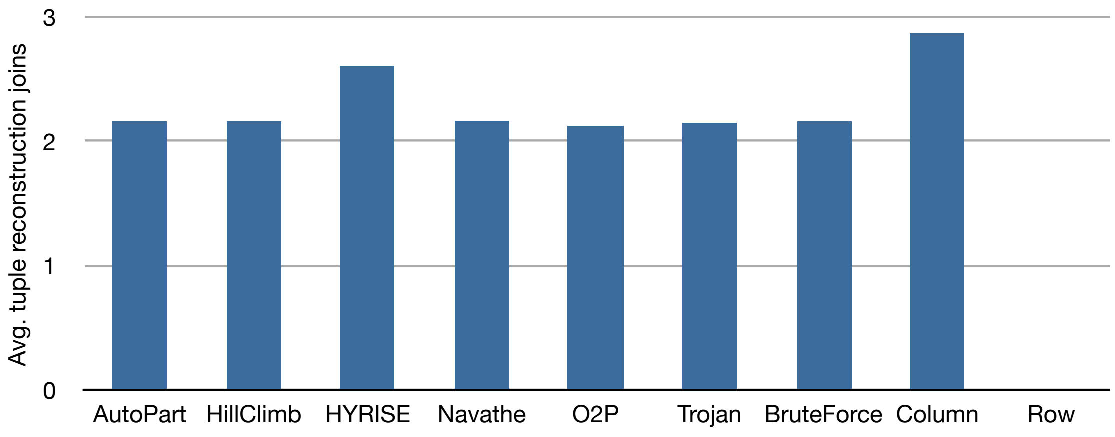


Figure 2.5: Average tuple reconstruction joins

How far is vertical partitioning from perfect materialized views?

We have seen above that the layouts produced by the vertical partitioning algorithms improve marginally over Column. This is in spite of almost all algorithms having estimated costs very close to the BruteForce, which produces optimal layout (See Figure 2.3). Let us now see how far are the vertical partitioning layouts from perfect materialized views — a vertical partition, created for each query, containing exactly the attributes referenced by that query. Figure 2.6 shows the distance of each of the layout from the perfect materialized views (PMV). The distance from PMV is calculated as follows:

$$\text{Distance from PMV} = \frac{\text{Est. costs of layout} - \text{Est. costs of PMV}}{\text{Est. costs of PMV}} * 100\%$$

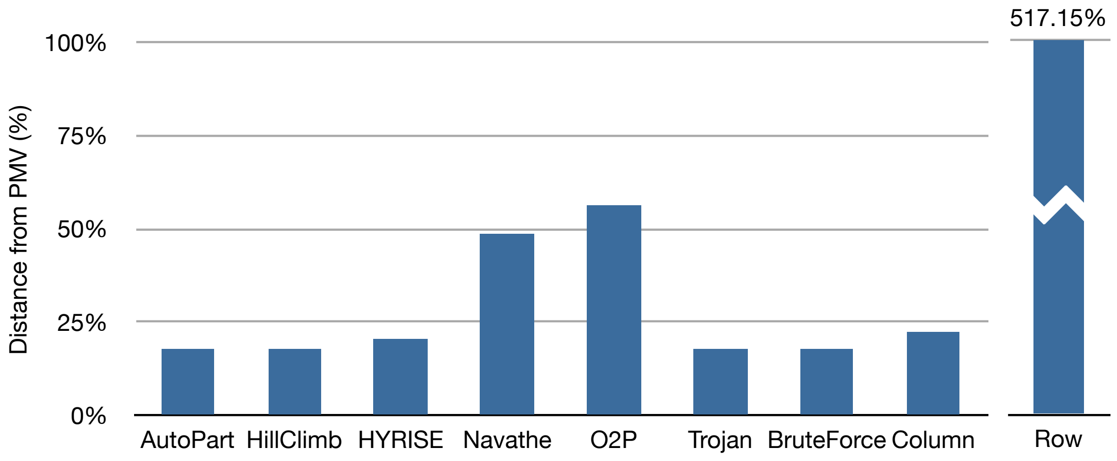


Figure 2.6: Distance from perfect materialized views

We can see that while Navathe and O_2P are 49% and 56% off from the perfect materialized views, respectively, HillClimb and AutoPart are as low as just 18% off from it. This is in spite of perfect materialized views needing much more storage space.

What is the effect of workload size on query performance?

We saw above that vertically partitioned layouts are up to 56% off from the perfect materialized views. Perfect materialized views and vertically partitioned layouts are two extremes: creating vertical partitions for *each* query versus creating vertical partitions for the *entire* query workload. Let us now see how the query performance changes in the middle.

In this experiment, we start from the perfect materialized views and gradually increase the workload size k (from 1 to 22). For each workload size, we compute the layouts and workload costs. Note that the partitionings produced by AutoPart, HillClimb, HYRISE, Trojan, and BruteForce have roughly the same estimated costs (See Figure 2.3), while the costs for Navathe and O_2P are always much higher, but quite close to each other. Thus, in the following we only consider HillClimb and Navathe. Figure 2.7 shows the estimated workload runtime improvements over Column for the layouts computed by HillClimb and Navathe, calculated in the following way:

$$\text{Improvement over Column} = \frac{\text{Est. costs of Column} - \text{Est. costs of layout}}{\text{Est. costs of Column}} * 100\%$$

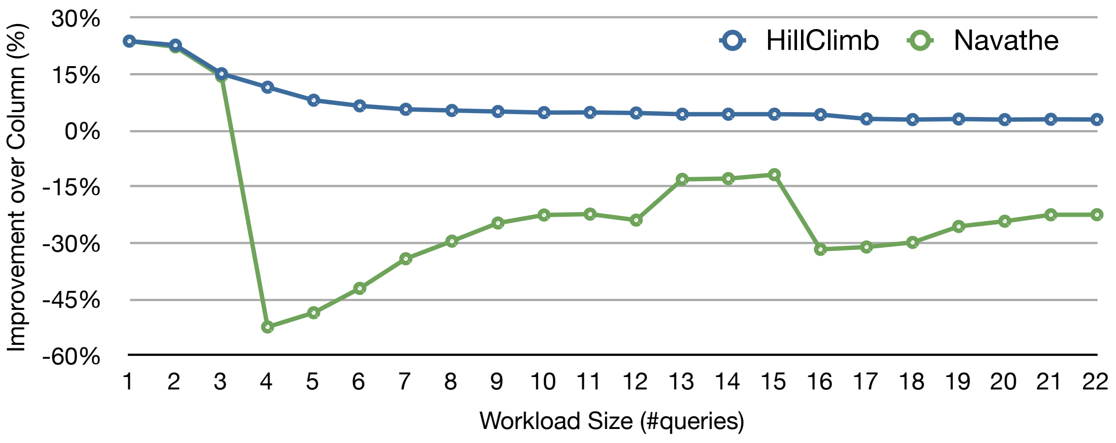


Figure 2.7: Estimated workload runtime improvements over Column when re-optimizing for the first k queries.

The improvement over Row remains roughly the same for both of them, so we have excluded it from this graph. However, the improvement over Column shows an interesting finding: for up to the first 3 queries, Navathe improves at least 15% over Column, but afterwards there is no improvement and it is always worse than Column. HillClimb on the other hand starts with an improvement of 24% over Column, which decreases to 6.5% for the first 6 queries, and remains roughly the same afterwards.

Let us now investigate the reason for this behavior, considering only the first 6 queries, i.e. k ranging from 1 to 6. Table 2.3 below shows the percentage of unnecessary reads for these workloads:

k	1	2	3	4	5	6
HillClimb	0%	0%	0%	0%	0%	0%
Navathe	0%	0%	0%	37%	32%	30%

Table 2.3: Unnecessary data reads over the Lineitem table for the first k queries.

From the table, we can see that in case of Navathe, starting from $k = 4$ the fraction of unnecessary data read has jumped from 0% to more than 30%. This explains why Navathe suddenly became worse than Column. On the other hand, the fraction of unnecessary data read for HillClimb and Column stays 0% for all these values of k. To understand the declining performance of HillClimb in Figure 2.7, let us take a look at tuple-reconstruction joins. Table 2.4 shows the average number of tuple-reconstruction joins over the Lineitem table for up to the first 6 queries. From the table, we see that more tuple-reconstruction joins were

k	1	2	3	4	5	6
HillClimb	0.00	0.00	1.00	1.00	1.75	2.00
Column	6.00	6.00	4.50	3.67	3.50	3.40

Table 2.4: Average number of tuple-reconstruction joins per row of the Lineitem table for the first k queries.

performed with larger workload size. This is because, with increasing workload size, the size of partitions decreases and thus the number of referenced partitions increases. Thus, with increasing values of k, the difference between the query performances of HillClimb and Column decreases. As a result, we can conclude that the random I/O accounts for most of the difference in estimated costs between HillClimb and Column.

In summary, the most important conclusion in this section is that while vertically partitioned layouts improve significantly over Row on the TPC-H benchmark, the improvement over Column is still less than 5%.

2.6.3 Comparing Algorithm Fragility

Below we will try to understand the fragility of each of the algorithms with the following main questions.

What is the effect of disk characteristics on query performance?

We proceed this experiment as follows. First, we run the algorithms for the same disk characteristics: 8 KB block size, 8 MB buffer size, 90 MB/s disk read-bandwidth and 4.84 ms seek time. Then, we take the layouts obtained from these disk characteristics and see how query performance would be affected, if these disk characteristics would change at query time. The idea is to see how much does the query performance deviate from the original setting's performance, if the layouts computed under one setting were used in another setting — also defined as fragility in Section 2.5. Figure 2.8 shows the fragility of layouts produced by each of the algorithms, when changing the buffer size. The fragility itself is defined as follows:

$$\text{Fragility} = \frac{\text{Est. costs with new settings} - \text{Est. costs with old settings}}{\text{Est. costs with old settings}}$$

From the figure, we can see that changing the buffer size can significantly affect the workload runtime, increasing it by up to a factor 24. This is because buffer size

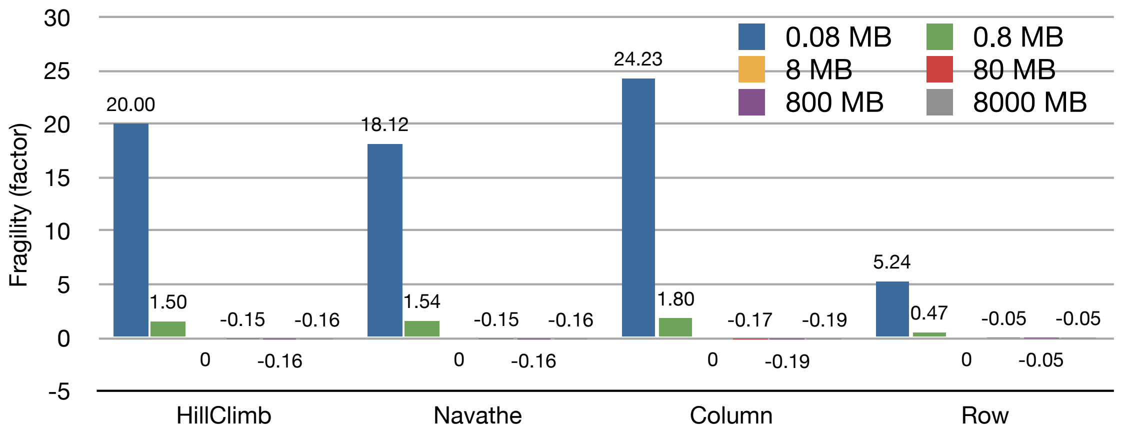
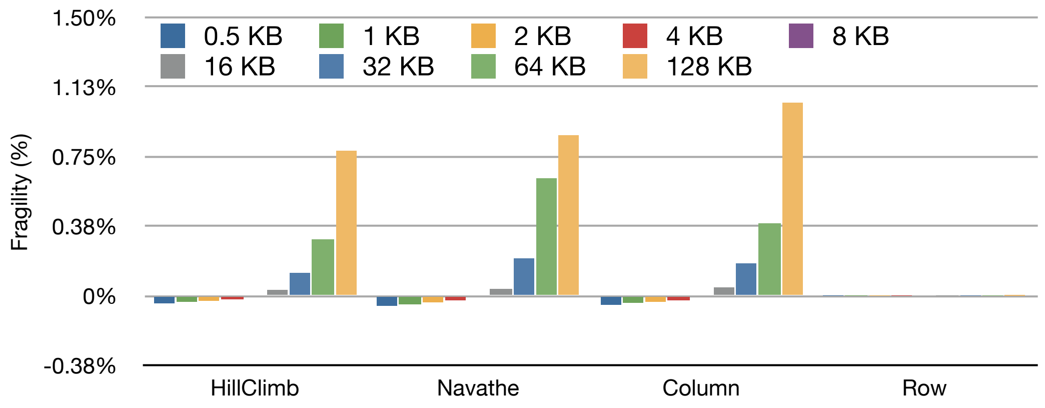


Figure 2.8: Algorithm fragility — estimated change in workload runtime due to changing the buffer size at query time.

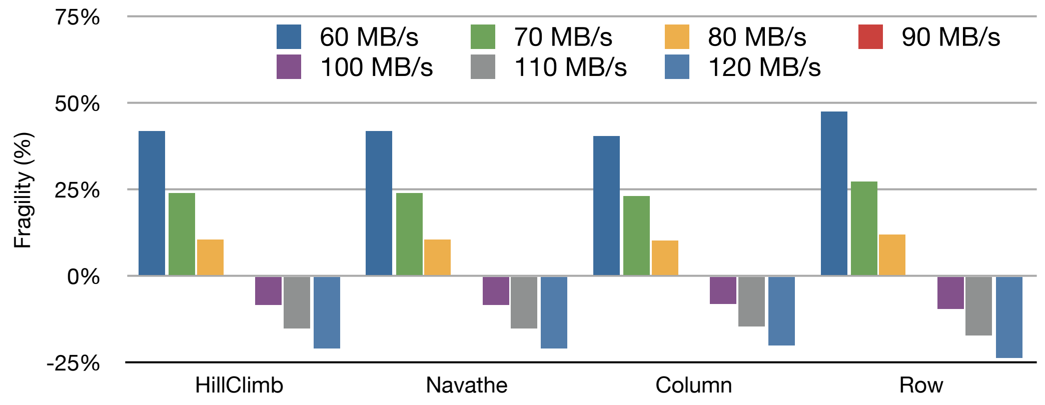
strongly determines the number of random I/Os during query processing. Other disk parameters like block size, disk bandwidth, and disk seek time do not have such an impact on query performance, as shown in the following.

How fragile are different algorithms to block size, disk bandwidth, and disk seek time?

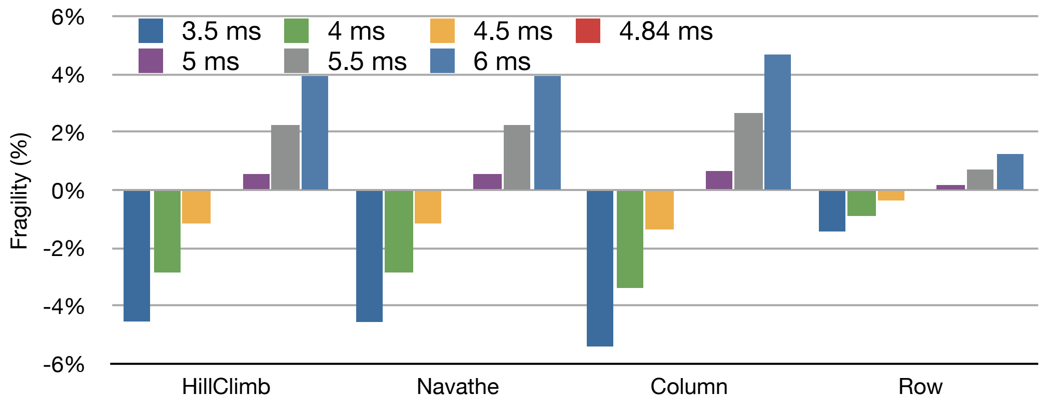
Figures 2.9a to 2.9c show the fragility of vertical partitioning algorithms when changing block size, disk bandwidth, and disk seek time at query time, respectively. From Figure 2.9a, we can see that changing disk block size has negligible impact — less than 1% — on query workload performance. This is because a database system needs to read integral number of blocks and changing the block size affects only the last block. Changing the disk bandwidth deviates the workload runtime by up to 42% (Figure 2.9b), while changing the seek time deviates the workload runtime by less than 5% (Figure 2.9c). Thus, we see that the performance of vertically partitioned layouts are stable over block size and disk seek time, they are affected marginally by disk bandwidth, but they highly depend on buffer size, as seen above. Thus, the take away message is that the performance of vertically partitioned layouts depends highly on the buffer size.



(a) Changing the block size



(b) Changing the disk's bandwidth



(c) Changing the seek time

Figure 2.9: Algorithm fragility — estimated change in workload runtime due to changing a single parameter at query time.

What happens if the workload changes?

We also ran an experiment to see how the query workload costs change with changes in the query workload, i.e. to see how fragile the algorithms are to the workload changes. In this experiment, we split the TPC-H queries on Linetem table into two sets, query set A and query set B, having roughly equal total costs. We ran HillClimb for one of the query sets, and gradually changed the mixture of the two sets of queries in the workload. Figure 2.10 shows the results. We change the fraction of set A queries along the X-axis.

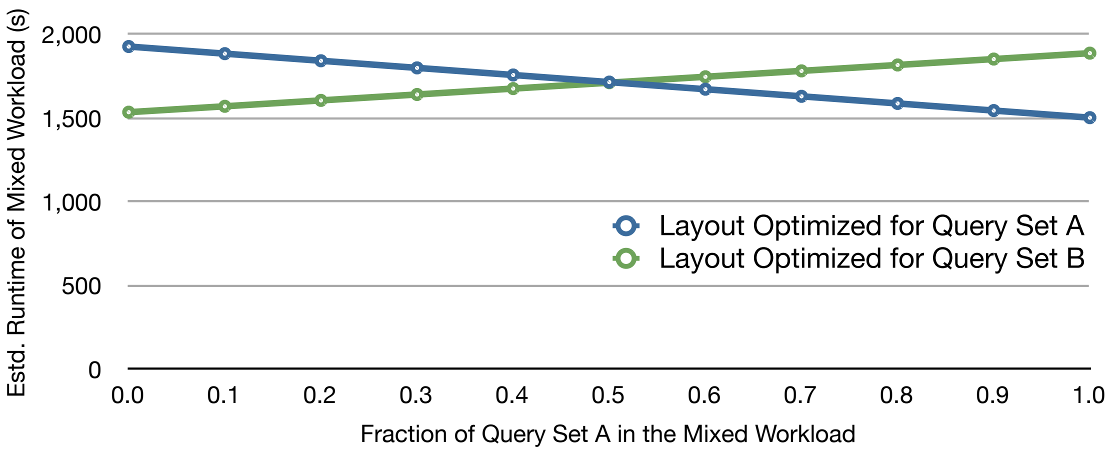


Figure 2.10: Fragility of algorithms over changing workload.

From Figure 2.10, we can see that the query costs change only by 14% for a 50% change in query workload from set A to set B. Thus, we can say that the algorithm is not fragile to such workload changes.

2.6.4 Where does vertical partitioning make sense?

In this section we concern ourselves with the following issues:

What happens if we adapt to different disk characteristics?

In the previous section, we saw that the performance of vertically partitioned layouts depend strongly on the buffer size. So let us now see how much do the query times change, if the partitioning is adapted to the different buffer sizes. Figure 2.11 shows the estimated workload runtimes for two vertical partitioning algorithms (HillClimb and Navathe) normalized by the estimated workload runtime for Column, when the buffer size is changed. The Normalized Estimated Costs is calculated as follows:

$$\text{Normalized Estimated Costs} = \frac{\text{Estimated costs of the layout}}{\text{Estimated costs of Column}} * 100\%$$

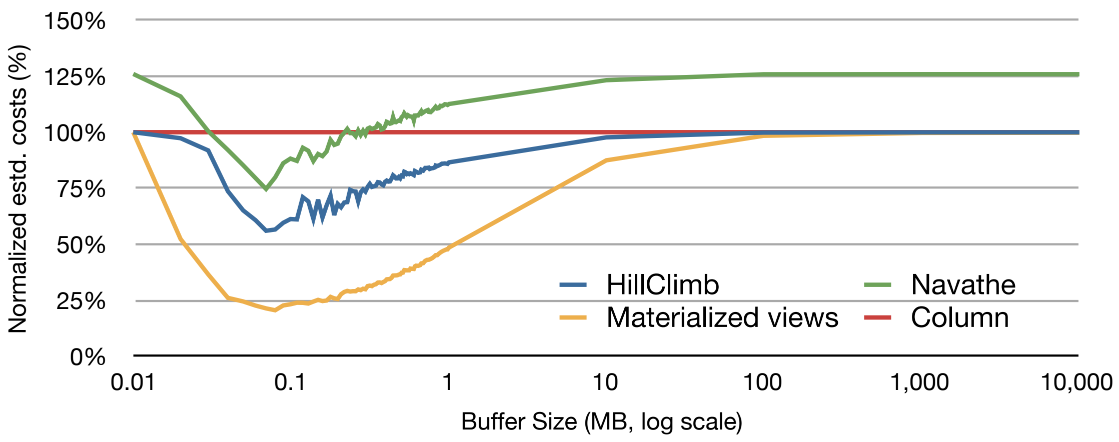


Figure 2.11: Estimated workload runtime compared to Column when re-optimizing for each buffer size.

Additionally, we also show the workload costs of the perfect materialized views as well as for Column. We do not show Row because it is out-performed by all other layouts for all buffer size values. In order to amplify the variation we compare the workload costs to Column for different buffer sizes. The first thing that we see is that in the best case, i.e. for the perfect materialized views, vertical partitioning pays off over Column only up to a buffer size of 100 MB. The layouts produced by HillClimb perform either better or the same as Column. HillClimb has the best improvement over Column for a buffer size of 100 KB. The layouts produced by Navathe, on the other hand, perform better than Column only in a narrow range of approximately 30 KB to 300 KB. For all remaining buffer size values, Navathe performs worse than Column.

For the sake of completeness, we also ran experiments to see the adaptivity of vertical partitioning algorithms over block size, disk bandwidth, and disk seek time. We have additionally examined the effects of scaling the dataset.

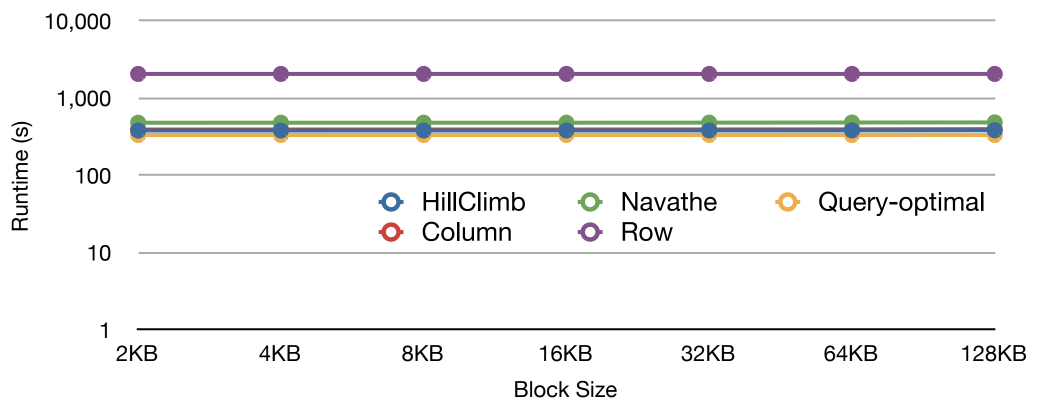
What are the sweet spots for block size, disk bandwidth, and disk seek time?

Figures 2.12a to 2.12c add to our findings on adaptivity, and show the estimated workload costs for the vertical partitioned layouts for different block sizes, disk bandwidths and disk seek times. We can see that the algorithms are almost unaffected by changes in block size (Figure 2.12a) and disk seek time (Figure 2.12c) — the standard deviations of the estimated costs compared to the averages are less than 0.5% and 9% respectively. To a certain degree, the algorithms are affected by changes in disk bandwidth (Figure 2.12b) — the afore mentioned metric is 30% in this case. But there are no interesting regions.

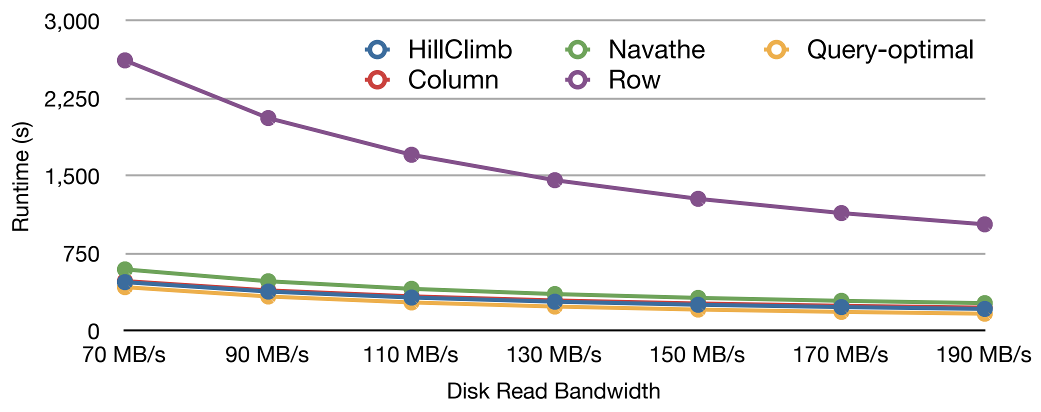
Do the sweet spots change with dataset size?

Let us now examine the effects of changing the buffer size together with scaling the dataset (i.e. varying the scale factor of TPC-H). We recompute the layouts for every buffer-size and for every scale-factor, and compare the workload costs to Column. Figures 2.13a and 2.13b show the results for HillClimb and Navathe. We can see that there is a jump in improvements over Column from scale factor 0.1 to 1.0 and buffer size larger than 1 MB. This is because for scale factor 0.1 (i.e. 100 MB data size), each query reads the same amount of data as the buffer size. For all other regions in Figures 2.13a and 2.13b, the impact of dataset size is negligible.

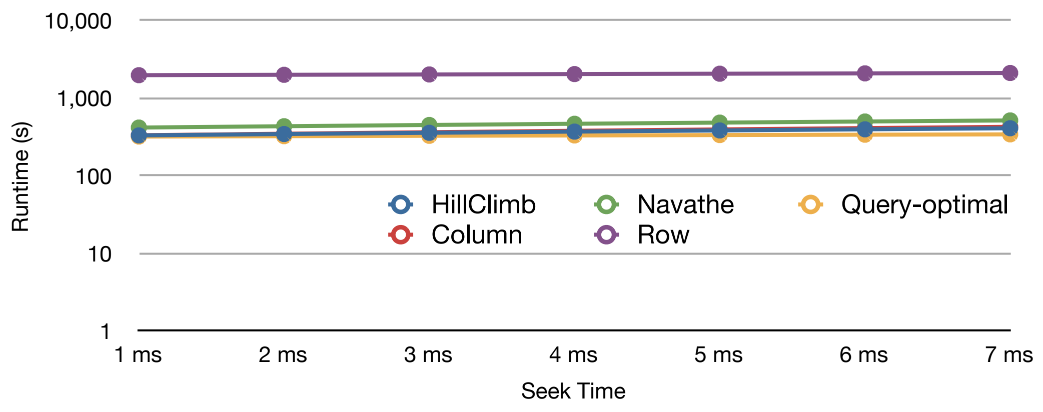
The key message from this experiment, and also from this chapter, is that vertical partitioning makes sense only for small buffer sizes, e.g. less than 100 MB. This is indeed the case for many data management systems. For example, PostgreSQL has a default buffer size of 8 MB. In case we can afford to have big buffers (due to large main-memory or dedicated nodes) it is better to use column layout.



(a) Changing block size

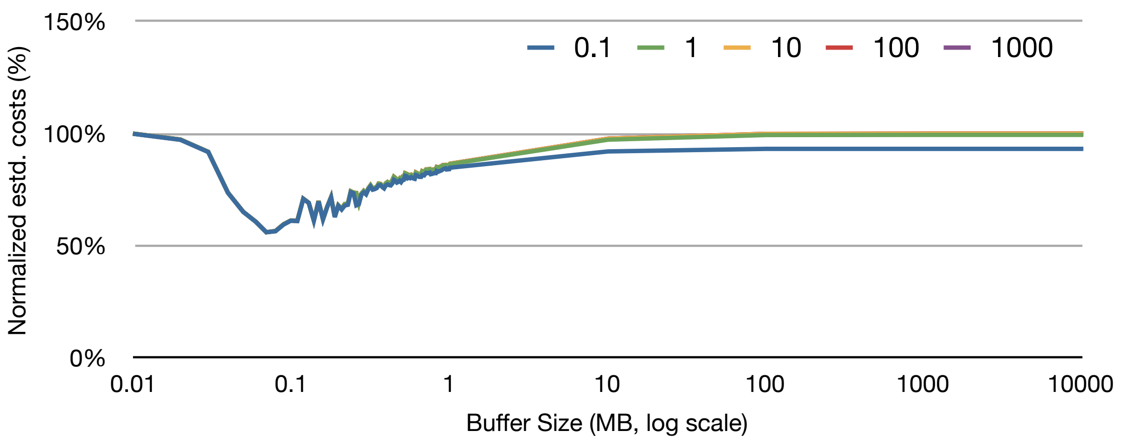


(b) Changing disk bandwidth

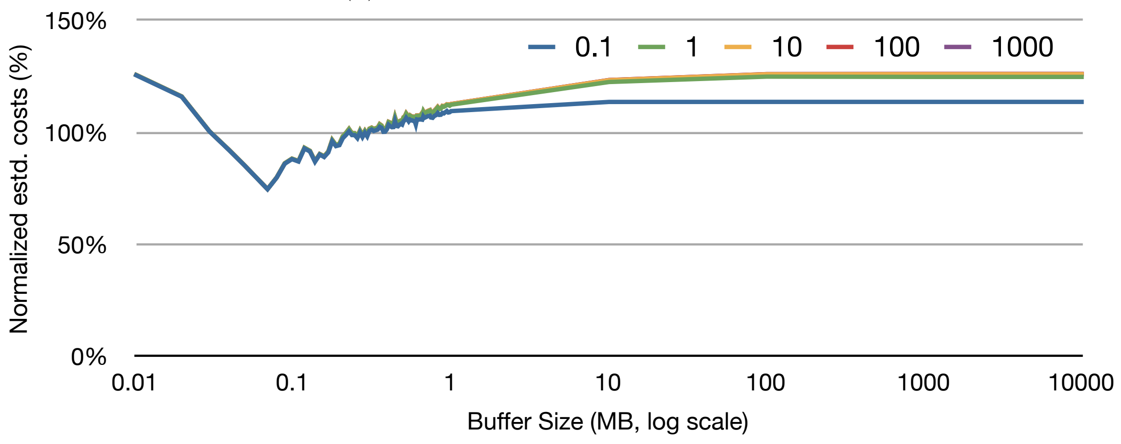


(c) Changing disk seek time

Figure 2.12: Estimated workload runtime when re-optimizing for each block size, disk bandwidth and seek time



(a) Scaling dataset with HillClimb



(b) Scaling dataset with Navathe

Figure 2.13: Sweet-spots for vertical partitioning — re-optimizing for each buffer size and each dataset size, and showing the estimated workload runtime compared to Column.

2.7 Lessons Learned

In this chapter, we compared different vertical partitioning algorithms and studied ways to pick one vertical partitioning algorithm over another for row-oriented database systems. Traditionally, vertical partitioning and index selection have been treated as different problems[1] and hence we do not consider selection predicates and indexes in our study. However, we did consider putting the selection attributes in a different partition. But it turns out that this affects the data layouts only when the selectivity is higher than 10^{-4} for uniformly distributed datasets, such as TPC-H. Below we discuss the key lessons learned in this chapter.

1. We don't really need brute force. The brute force algorithm spends an extremely long time to compute the layouts (more than an hour for TPC-H). On the other hand, the vertical partitioning algorithms evaluated in this work terminate in at most a few minutes. In fact, AutoPart and HillClimb take less than 1 second to compute the layouts for all tables in the TPC-H benchmark. Still both AutoPart and HillClimb find *exactly* the same solution as the brute force algorithm. HYRISE takes slightly more than a second to compute the layouts but it is only 2.21% off from the brute force algorithm, in terms of query costs. Similarly Trojan takes a couple of minutes for optimization, however it is just 0.01% off from the brute force algorithm in terms of estimated runtime. This is an important result and shows that we do not really need the brute force algorithm. Several heuristics, as proposed in different algorithms, are good enough.

2. Watch out for the buffer size. The performance of vertically partitioned layouts depend heavily on the database buffer size. In fact, the buffer size can impact the query workload runtimes by as much as factor 20. Thus buffer size is a crucial consideration when computing vertical partitioning. Furthermore, our measurements reveal that vertical partitioning improves over column layout only for buffer sizes less than 100 MB. This means if we can have a system with buffered reads of more than 100 MB at a time, then we better use column layout. Put another way: if we want to avoid vertical partitioning then we must increase the buffer size of our database system. This is one of the core results of this chapter.

3. HillClimb is the best algorithm for disk-based systems. Amongst the six vertical partitioning algorithms compared in this chapter, HillClimb turns out to be the best for the TPC-H queries. HillClimb offers the best trade-off between optimization time and workload runtime performance. It spends 4 orders of

[1]In fact, most of the vertical partitioning algorithms do not consider selectivities.

magnitude less time in optimization and still finds the same vertical partitioning as the brute force algorithm. As a result, the optimization time of HillClimb pays off the earliest (just after 25% of TPC-H workload) over row layout. Furthermore, from our experience HillClimb is also one of the easiest algorithms to understand and implement.

4. Column layouts are often good enough. On the TPC-H benchmark (i.e. all 22 queries) the vertical partitioning algorithms could improve over column layout by only up to 3.7%. This is because the attribute access patterns over all 22 queries are quite fragmented and it is hard to find column groups which satisfy most of the queries. Indeed, the improvements over column layout go up to 24% when using a small subset of the TPC-H workload (see Figure 2.7). But still the improvements over column layout are not dramatic. To investigate this further, we tried three changes in our experimental setup — using a different benchmark, using a different cost model, and using a commercial database system which supports column grouping.

(a) *Using a different benchmark.* We used the Star Schema Benchmark[58]. The Star Schema Benchmark has less fragmented access pattern and so we expect wider column groups. Table 2.5 compares the results on the TPC-H and the Star Schema Benchmark (SSB).

Layout	**TPC-H**	**SSB**
AutoPart	3.71%	5.29%
HillClimb	3.71%	5.29%
HYRISE	1.58%	5.27%
Navathe	-21.47%	1.64%
O_2P	-27.74%	1.64%
Trojan	3.71%	0.05%
BruteForce	3.71%	5.29%

Table 2.5: Estimated improvement over column layout with different benchmarks.

We can see that even though column grouping improves over column layout by up to 5.29% on the Star Schema Benchmark, still the improvement is not dramatic. Thus, using column layouts in the first place for TPC-H-like workloads is not a bad idea. This will avoid the complicated vertical partitioning machinery.

(b) *Using a different cost model.* We used the main-memory cost model from the paper on HYRISE [26]. It models the number of cache misses when accessing data from a column grouped layout. For TPC-H queries, we show the estimated workload runtime improvements over column layout. Table 2.6 compares the results when using disk (HDD) and main-memory (MM) cost models.

Layout	**HDD Cost Model**	**MM Cost Model**
AutoPart	3.71%	0.00%
HillClimb	3.71%	0.00%
HYRISE	1.58%	0.00%
Navathe	-21.47%	-15.07%
O_2P	-27.74%	-15.53%
Trojan	3.71%	0.00%
BruteForce	3.71%	0.00%

Table 2.6: Estimated improvement over column layout with different cost models.

From the table we see that except for Navathe and O_2P, all other algorithms have no improvement over column layout in main-memory. This is due to the fact that the seek-costs compared to the scan costs are way smaller in main-memory than for disk-based systems, which means that a column-group cannot significantly decrease the data access costs in main-memory. Instead, column groups can potentially increase the amount of data read and hence be even worse than column layout (see Navathe and O_2P for main-memory). On the other hand, reading data in column layout causes the least possible number of cache-misses, thus allows for the fastest data access. Therefore, in terms of data access costs, it is hard to beat column layout in a main memory-based system. Indeed, in the HYRISE-paper[26], the hybrid layouts improve over column layout by just 3.8% in the total workload cost. This is even when the workload chosen in the paper on HYRISE uses very wide tables with up to 150 attributes and several queries accessing a large fraction of those attributes.

(c) *Using a commercial database system.* Finally, we used a commercial disk-based column-oriented database system (referred to as DBMS-X in the following), which supports column grouping. The idea is to compare vertically partitioned layouts with column layouts on the TPC-H benchmark. To do so, we created and loaded two TPC-H databases with scale factor 10, one with column layout and the other with a vertically partitioned layout calculated by HillClimb. Like any other column store, DBMS-X relies heavily on compression and it cannot be turned off. The default compression for string and floating point numbers is Lempel-Ziv-Oberhumer-based (LZO), while for integer and date types the compression scheme is delta encoding. We executed the unmodified queries of the TPC-H workload on these two databases. Table 2.7 shows the total workload runtime[2] for row, column, and the vertically partitioned layout produced by HillClimb.

Compression	**Row**	**Column**	**HillClimb**
Default (LZO or Delta)	1652 s	377 s	450 s
Dictionary	1265 s	511 s	532 s

Table 2.7: TPC-H workload runtimes with scale factor 10 in DBMS-X for different layouts and compression schemes

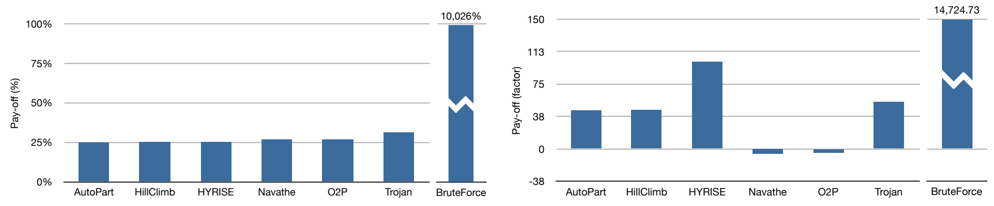
When using the default compression the difference between column layout and HillClimb is quite high. This is due to the varying length encoding, used in the vertically partitioned layout as well, which makes the tuple-reconstructions within a segment of a column-group costly. We ran another experiment in which we forced all layouts to use the dictionary compression, which is a fixed-size encoding. With dictionary compression, the gap between column and HillClimb layout reduces. Still, column layout outperforms HillClimb.

Having said the above, however, there are several practical limitations to using column layouts in legacy row stores. For instance, the standard practice to create a separate table for each vertical partition causes the column layouts to incur the maximum tuple header overheads. Thus, vertical partitioning is still a necessity for the majority of row stores.

2.8 Conclusion

There are a number of vertical partitioning algorithms proposed in the literature. In this work, we presented a systematic and comprehensive study of vertical partitioning algorithms. We categorized vertical partitioning algorithms along three dimensions and surveyed six different algorithms. We experimentally evaluated these six algorithms under a common configuration setting. We introduced four metrics to compare different vertical partitioning algorithms and showed results from the TPC-H benchmark. Our results identified the trade-offs between optimization time and workload runtime improvements, improvements over row and column layouts, and effects of database buffer size.

[2]We excluded query 9 since DBMS-X has chosen a sub-optimal query plan for it, which caused an enormously high runtime.



(a) Pay-off over Row (b) Pay-off over Column

Figure 2.14: Pay-off in workload runtime improvements over optimization- and creation times.

Appendix

2.A Additional Results

2.A.1 How soon does vertical partitioning pay-off?

Now let us see how soon do the efforts made in vertical partitioning pay off, i.e. the fraction (or factor) of workload for which the accumulated workload cost improvements exceed the optimization and layout creation costs. Thus, we define pay-off as follows:

$$\text{Pay-off} = \frac{\text{Optimization time} + \text{Creation time}}{\text{Improvement in estimated workload costs}} * 100\%$$

Figure 2.14a shows when the algorithms pay off over Row. We can see that all algorithms pay off after approximately 25% of the workload has been executed. Due to the very high query costs for Row we do not see a variation of the pay-off for the different layouts. Pay-off after 25% of the workload means that just 25% of the TPC-H workload is enough motivation for computing the vertically partitioned layouts.

Figure 2.14b shows how soon vertical partitioning pays-off over Column . We can see that AutoPart pays off the earliest, after running the TPC-H workload 44.5 times. HYRISE is the last to pay off (after running the TPC-H workload 101 times). This long time to pay off over Column is due to the very small improvement (up to only 5%) in workload costs over Column. As a final remark, we see in Figure 2.14b that Navathe and O_2P have negative pay-off factors, This is because these two algorithms do not improve workload costs over Column.

2.A.2 Extending our Model to Consider Selectivity

In this chapter we have excluded query execution costs from our cost model, and only considered data access costs. Furthermore, we did not distinguish between columns used in the final result vs. columns used in the selection predicate only. Instead we have assumed that the benefits and costs of vertical partition lie simply in columns that are accessed. However, it would be interesting to investigate the potential benefits of putting the columns used in the selection predicate in a separate vertical partition. For example, suppose that there were a workload composed mostly of queries that invoke a highly-selective predicate on just `PartKey`. In that case, it could be best to have the `PartKey` in a vertical partition by itself, and group the rest of the attributes together.

It is to be noted again that almost all vertical partitioning algorithms consider the scan and projection operators only. Since we were doing a comparative study of different algorithms, we have considered the same set of operators for all algorithms. Nevertheless, in order to investigate the effects of selection operators on the results produced by vertical partitioning algorithms, we have extended our cost model to take selectivity into account. Now, for each query in the workload, we provide the selection attributes and the overall selectivity, in addition to the list of referenced attributes. Note that since we are considering vertical partitioning in legacy row-stores, i.e. tuple-at-a-time processing, we assume buffered-reading all relevant vertical partitions. This means that we still need to share the read buffer between selection and projection attributes. Our query processing model is similar to the one used in the paper on HillClimb [28]. We first execute a full scan on the partitions containing the selection attributes and collect the row IDs of the qualifying tuples. Using these row IDs, we can determine which blocks we need to read from the remaining referenced partitions. Depending on the query selectivity, we might be able to skip reading some of the blocks from the remaining referenced partitions.

In our extended cost model we assume a uniform distribution of search keys, like in the TPC-H benchmark, so the distance between two qualifying rows (`jump`) is constant. For each referenced partition we can calculate the number of rows per block (`blocking factor`) by dividing the block size with the row size of the partition. If `jump` is less than the `blocking factor`, we cannot skip reading any blocks of that partition, so we have to do a full scan. Otherwise, we can skip some of the blocks. From the `jump` and `blocking factor` we can easily calculate the proportion of blocks to read from a given partition. However, it might not pay off to skip the unreferenced blocks if the additional seeks imposed by skipping these blocks have a higher cost than what we gain on reading fewer blocks. Therefore we always compare the costs of full scanning a referenced partition and reading only the blocks that contain qualifying tuples, and choose the execution plan that is cheaper.

Let us now see what happens if we have a query with a highly selective predicate on a single attribute. We let the vertical partitioning algorithms use our extended cost model, as described above, when computing the layouts. We consider the following query on the TPC-H PartSupp table projecting two attributes and preforming a selection on a third one:

```
SELECT availQty, suppKey
FROM PartSupp
WHERE partKey > c;
```

We vary the selectivity of the query by setting the value of the constant `c`. We chose a scale factor of 1000 for the TPC-H database, which yields a table size of 172 GB, and we read 12 bytes from each 216 bytes long row. We vary the selectivity from 1, 10^{-1}, ..., 10^{-8} to see when the algorithms choose to put `partKey` into a separate partition. Figure 2.15 shows the resulting estimated query costs for two partitionings p1 and p2, when varying the query selectivities. Partitioning p1 is produced by the unmodified HillClimb algorithm and p2 extends p1 by putting the selection key (`partKey`) in a different partition.

```
p1: {partKey, suppKey, availQty}, {supplyCost}, {comment}
p2: {partKey}, {suppKey, availQty}, {supplyCost}, {comment}
```

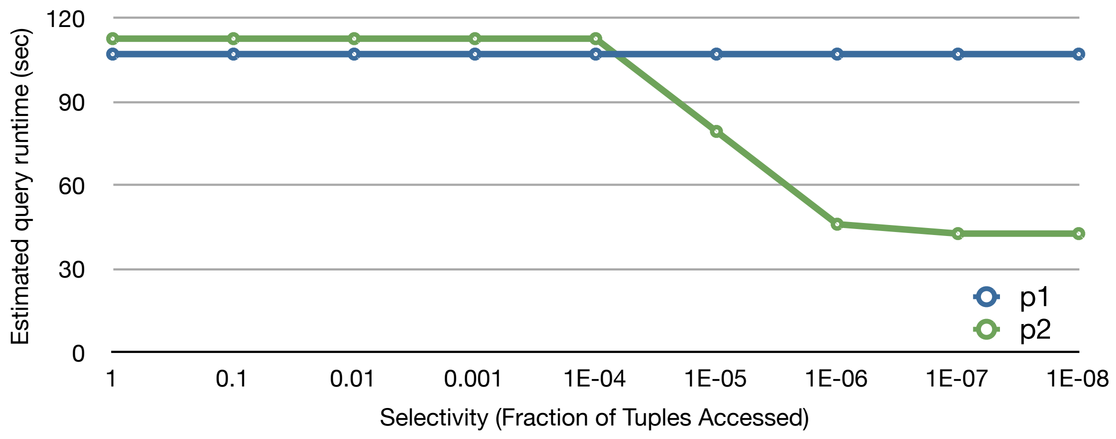


Figure 2.15: Estimated query I/O cost over varying selectivity

From Figure 2.15, we see that p2 (which puts the selection attribute in a separate vertical partition) becomes better than p1 only for selectivities higher than 10^{-4}. We also tried considering a third partitioning p3:

```
p3: {partKey}, {suppKey, availQty, supplyCost, comment}
```

Partitioning p3 simply puts the selection key into a singleton partition and all the remaining attributes into another partition. However, we found that p3 becomes better than p1 (and similar to p2) only for selectivities higher than 10^{-5}. For lower selectivies, the performance of p3 becomes much worse than p1 and similar to row layout. Thus, we conclude that considering selectivity affects vertical partitioning only for very high selectivities. We recalculated the TPC-H query costs with this new cost model, and interestingly none of the costs change. This indicates that the above discussed late materialization does not help TPC-H like datasets. This makes sense because given the uniform data distribution in TPC-H, only very highly selective queries can skip reading some data blocks. An interesting follow-up might be to consider vertical partitioning along with sort orders (and indexing) for selection predicates. However, this calls for developing newer physical design algorithms and is beyond the scope of this work. In this work, we have focused only on vertical partitioning in order to compare different vertical partitioning algorithms.

2.B Layouts

Figure 2.16 shows the vertical partitioned layouts for all tables in the TPC-H workload. Two or more attributes having the same color in a given row means that they belong to the same vertical partition. For the Lineitem table (Figure 2.16b) AutoPart, Trojan, and Optimal produce the same results. HillClimb's results differ only in not grouping the two unreferenced attributes (LineNumber and Comment) together. The same occurred for the Part table (Figure 2.16f) where HillClimb left the two unreferenced attributes (RetailPrice and Comment) in separate partitions, contrary to AutoPart, HYRISE, Trojan and Optimal. The reason for Trojan producing a slightly different layout for the Customer and Supplier tables — compared to the other algorithms in the "HillClimb-class" — is that it uses an interestingness-measure as a heuristic making it sometimes chose sub-optimal column-groups as well. Navathe and O_2P form the second class of the vertical partitioning algorithms we have considered which is clearly visible on the partitioning results, since they always produce a partitioning which has significant differences from the results of the "HillClimb-class". For the Nation and Region tables (Figures 2.16g and 2.16e), the partitioning doesn't influence the I/O cost. This is because these two tables have only 25 and 5 rows, respectively, and hence they fit into one block.

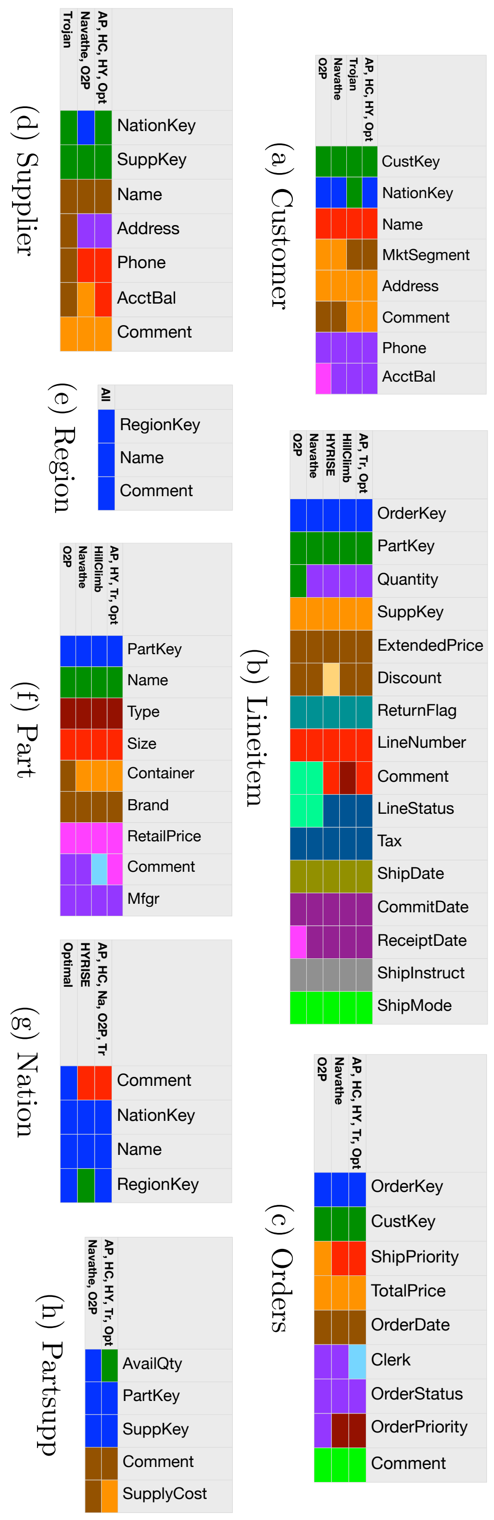


Figure 2.16: The computed partitions for the TPC-H workload.

2.C Implementation Notes

Our implementation of the vertical partitioning algorithms have been released at `https://github.com/palatinuse/database-vertical-partitioning`. In this section we provide some background information on the implementation, and use an example workload to describe the key concepts necessary to understand our codebase.

2.C.1 A short walk-through of vertically partitioning the TPC-H PartSupp table

Let's take a look at the TPC-H PartSupp table's workload, and it's vertical partitioning. Table 2.8 shows the following informations: which query references which attribute, marked by an X.

In our implementation each `Workload` is first reduced by kicking out queries which do not touch the given table. Furthermore, the workload is converted to a more efficient representation using primitive types only (called `SimplifiedWorkload` in the code), using GNU Trove collections. As a result of this step performed on the TPC-H PartSupp table we get only 5 queries (with IDs 0 to 4, shown in the second column in Table 2.8).

Now let's run the AUTOPART algorithm for the PartSupp table. Here's the output of the run:

```
partsupp
1 #AUTOPART
0.0038134 #seconds computation time
4 #partition count -- partitions:
1 0
2
3
4
5 #query count -- best solutions:
2 0
2 0
2 1 0
0
1 0
```

We get two mappings as a result. *Partitions* is the result of a VP algo: the list of partitions with the attributes in them. Thus the first line (1 0) means Partition 0 contains attributes 1 and 0 (marked green in the table).

TPC-H Query	Query ID	*Partition 0*	*Partition 0*	*Partition 1*	*Partition 2*	*Partition 3*
		PARTKEY	SUPPKEY	AVAILQTY	SUPPLYCOS	COMMENT
		Attribute 0	Attribute 1	Attribute 2	Attribute 3	Attribute 4
Q1						
Q2	0	X	X		X	
Q3						
Q4						
Q5						
Q6						
Q7						
Q8						
Q9	1	X	X		X	
Q10						
Q11	2	X	X	X	X	
Q12						
Q13						
Q14						
Q15						
Q16	3	X	X			
Q17						
Q18						
Q19						
Q20	4	X	X	X		
Q21						
Q22						

Table 2.8: Vertical partitioning of the TPC-H PartSupp table.

The second mapping, *best solutions* is a bit more tricky. If we have a vertically partitioned table, and a query comes in, we have to decide which vertical partitions we should scan to answer the query. This itself is an NP-hard problem when partial attribute replication is allowed, like in the AUTOPART algorithm (Note: these algos are called `AbstractPartitionsAlgorithm` in the code, in contrast to `AbstractPartitioningAlgorithms` that create a disjunct set of partitions, i.e. without overlaps). The best solutions tells us for each query which partitions to scan, in order to answer the query WITH THE LOWEST COST, according to the cost model used (*Note*: this is only non-trivial for overlapping (non-disjunct) partitionings). Thus the first line (2 0) means query 0 (which is TPC-H Q2) needs to scan partitions 2 and 0 (the red and green columns in the table).

2.C.2 A Quick-start Guide

- Build metis (under lib/metis) on your machine
- Create a new instance of `experiments.AlgorthmRunner` using its default constructor setting up to run all VP algorithms, the whole TPC-H benchmark using scale factor 10, and using an HDD cost model.
- Call the `experiments.AlgorthmRunner.runTPC_H_Tables()` method to get the vertical partitionings of all TPC-H tables, for each VP algorithm.
- Print the resulting partitionings calling the `experiments.AlgorithmResults.exportResults()` method, passing in the results attribute of the `experiments.AlgorthmRunner` instance.

```
AlgorithmRunner algorithmRunner = new AlgorithmRunner();
algorithmRunner.runTPC_H_All();
System.out.println(
  AlgorithmResults.exportResults(algorithmRunner.results));
```

Chapter 3

Query Processing on Top of Flat Files

In this chapter we show a survey of query processing on top of flat files, i.e. text files containing data encoded in some standard text format. Flat files are commonly used in various fields of science to store experimental results in a human-readable format. These datasets are considered external to a database management system, and to efficiently process them using a DBMS, they first have to be loaded into the DBMS, which imposes a higher time-to-query. Other ways of processing these datasets are using custom-built applications that operate on the flat files directly. These approaches are expected to yield an inferior query performance, yet with zero time-to-query. We will show cases where this is not completely true, and where flat file processing tools can even outperform a DBMS.

3.1 Introduction

Querying data stored in structured text files, called flat files, requires different methods than data stored in a database. This is imposed by the differences in their format, data types and storage-layout.

3.1.1 Flat File Databases

A flat file database is a way of encoding and storing data of a single table in a single file. The data can be stored in text- and binary format as well, but it is more common to use the text format because of its human-readable form. In each case we need some means to logically separate the records and the fields from each other in the file. It is common to store one record per line, therefore the record-separator is often the character, or depending on the platform a pair of characters, denoting the end of a line. There are two main ways of encoding records: the delimiter-separated values and the fixed-width format.

In delimiter-separated values files the fields are separated by delimiter characters, such as commas, tabs, or vertical bars. Using delimiters imposes an overhead on processing, since they have to be located each time when a record is read. Yet, they are much less wasteful in storage space compared to XML. A well-known example of this format is CSV, i.e. comma-separated values, where the fields are separated by commas, and one record is stored per line.

In case of a fixed-width formatted files the records and fields are implicitly defined by their position within the file, since each field has the same, pre-defined length in each record. For this format there is no need for record- and field-separator characters, however, to improve human-readability we can still separate records by end of line characters. The position of a given field in a given record can be easily calculated, though this comes at the price of having an increased storage requirement.

3.1.2 The Processing Model of Flat File Databases

In the following discussion we assume that each file contains data of a single relation only, and all records have the same schema within a file. The latter restriction can be easily lifted with some added complexity in the processing steps.

The processing of flat files starts with identifying the tuples. It consists of finding the records within the file, separating the fields within each record, and converting the required fields into the proper datatype. Once the tuples are available the actual processing can begin. In more detail this process consists of the following steps:

1.) Determining the boundaries of each record using the record-separators.
2.) Tokenisation: determining the boundaries of each field within a record using the field-separators.
3.) Parsing: converting the required fields into the appropriate binary format.
4.) Selection: evaluating the selection predicates on the record.
5.) Processing: in case of a match, the desired data processing steps are executed, e.g. printing or aggregation.

The previously introduced two record-encoding formats have different trade-offs for the above processing model. The most important benefit of the fixed-width format over the delimiter-separated one is that we do not need to tokenize the fields at all. Searching for the field-separators imposes a CPU-overhead while processing delimiter-separated files. On the flip side we have to read less data from storage if the file is in delimiter-separated values format.

3.1.3 Research questions

Both databases and scripting languages have a large user base, and both of them would swear on their tool being the right one for processing data residing in flat files. In this chapter we try to give a guideline on when one camp should be better off using the other camp's tool by answering the following research questions:

- Can scripting languages compete with database systems in query processing?
- Does it pay off to invest in loading the data into a database?
- How should we choose the proper tool and data format for flat file processing?

3.1.4 Contributions

Are main contributions in this chapter are as follows:

- We show how to load data into Postgres in the most efficient way. Hereby, we discuss various database tuning steps and tools for data loading.
- We compare three different tools for processing flat files: the Postgres database system, an AWK script, and a hand-written C-application. We compare them on their query time, and time-to-query as well. The latter once includes the cost of loading data into the database, which is only applicable to Postgres.
- We provide two representative examples of queries that exemplify two extreme cases. For single-table queries containing filtering but no grouping, flat file processing tools are better suited, and databases might never pay off because of the upfront costs of loading the data. For more complex queries involving joins, aggregations, and the like, a database system can drastically improve subsequent query times and thus the loading times will eventually pay off.

3.2 Related Work

Many scientists store their experimental results in flat files [32], and query them with tools meant for flat file processing, e.g. AWK [5]. The reasons against using a DBMS for data storage and exploration is the high upfront cost of *loading* the data into the database and the *administration* overhead of setting up and tuning the DBMS in the first place. For instance, having to migrate the results of an experiment into the DBMS might be a huge waste of time if it turns out after the first query that it does not contain any information of interest to the researcher. On the other hand once the data is in the database the user can take advantage of the sophisticated query processing capabilities of the DBMS and achieve a considerable performance gain.

To solve this problem NoDB [7], an extension of PostgreSQL, was proposed, which allows for querying flat files with a DBMS, without having to load the data into the database first, and enjoying the benefits a DBMS can offer, i.e. SQL and caching at the same time. NoDB creates a positional map, which is a cache of the tokenization results, i.e. the field boundaries. It is coined as indexing by the authors, since it is an index on the starting positions of the fields within the flat file. Building indexes on the field values themselves as well would make sense, and would require further extensions of the scan operator.

Invisible loading [3] introduces a technique to speed up future MapReduce jobs operating on the same data. It proposes to load the attribute values extracted by the RecordReader into a local column-store database and to read data requested by subsequent queries from the database instead of the distributed filesystem, if it has already been loaded.

Data vaults [33] enables a common view of relational databases and scientific file repositories by offering a common interface for querying data regardless of its location. Data from file repositories are loaded just-in-time when a query requests data not residing in the database. Yet, it is unclear what are the costs of this loading procedure, and whether and for how long the loaded data is kept in the database.

Efficient loading of CSV-files into a main-memory DBMS has been presented in [45]. They point out that the traditional bulk-loading does not exhaust the resources of a modern, multicore CPU and introduce a parallel and vectorized method for loading CSV-files. They also show how to bulk create index structures by using merge-able index structures.

The UNIX-community has several tools for querying flat files, e.g. various shells, perl, python, awk, and grep, which all do a full-scan of the whole file for each query. Physical design techniques, e.g. data layout transformation and compression, however, have not yet been integrated into these tools. The more powerful tools perform also a tokenization for each input line, allowing for accessing each field separately. Building hash indexes on the queried fields on-the-fly can be used to speed-up subsequent queries touching the same fields, as done in [21]. On non-indexed data AWK has been shown to outperform MySQL for simple queries involving selections and joins [44].

3.3 Benchmarking Flat File Processing

In this section we evaluate flat files processing in various systems, and point out the possible points of improvements in the processing pipeline.

3.3.1 Experimental Setup

We have used the following common testbed for all experiments: a single node machine with a dual-core Intel Core i3-2120 processor running at 3.30 GHz with 3 MB Intel smart cache, having 16 GB RAM and 2 TB HDD, running Ubuntu 13.10 64 bit with Linux kernel version 3.11.0. We measured the disk characteristics of our testbed using Bonnie++ [12], a hard disk and filesystem benchmarking tool, and obtained the following results: a disk read bandwidth of 172 MB/s, disk write bandwidth of 128 MB/s and average disk seek time of 3.4 ms.

Unless otherwise stated we have cleared the OS and HDD caches before each run of an experiment, and have performed 5 runs for each measurement. We have used the TPC-H database with scale-factor 10 as a data source in our experiments.

3.3.2 Loading Data into PostgreSQL

In this experiment we measure the time it takes to load data into a PostgreSQL database (version 9.2.4) in various configurations. Our datasets are the Lineitem and Customer tables from the TPC-H Benchmark with scale-factor 10, stored in text files in delimiter-separated format. This format is essentially the same as CSV, except for the field-separator character, where we have used a vertical bar instead of a comma.

We will examine the following three methods for loading data into the database:

1. `psql \copy`: This is a command that is executed by the client, and therefore moves files through the network between client and server.
2. `SQL COPY`: This command is executed on the server, thus it operates on files local to the server, bypassing any network communication.
3. pg_bulkload module: This is a PostgreSQL extension module capable of multi-threaded loading.

In Table 3.1 we can see the loading times for both tables, and the two loading commands mentioned above. We have to note that the client-based `psql \copy` command has been executed on the server machine, yet still through a network connection. Since the Customer table is relatively small, we do not see a significant difference between the loading times for the two commands. However, for *both* the Lineitem and Customer tables, the COPY command executed by the database server is actually 5% faster than the client-based one.

Table	**psql \copy**		**SQL COPY**	
Customer	14.1 sec	16.5 MB/s	13.4 sec	17.3 MB/s
Lineitem	568.3 sec	12.9 MB/s	540.9 sec	13.6 MB/s

Table 3.1: Loading from text files in PostgreSQL with default configuration settings.

The PostgreSQL manual contains some tuning tips explicitly for loading data. The most important ones are the following:

- Don't interleave loading with index maintenance. It is cheaper to first drop all indexes on the table, then load the data, and finally bulk-load the indexes. However, the TPC-H benchmark does not enforce creating indexes on the tables, therefore we haven't created any indexes either.
- Adjust WAL-checkpointing behaviour. PostgreSQL writes new transactions to the database in files called write-ahead log segments that are 16 MB in size. Every time `checkpoint_segments` worth of these files have been written, by default 3, an automatic WAL-checkpoint is created. We have increased this parameter value to 64, thus there is a checkpoint created per 1 GB of WAL-segments written. The `checkpoint_completion_target` parameter specifies the target of checkpoint completion, as a fraction of total time between checkpoints. This should be adjusted from the default value of 0.5 to 0.9.
- Adjust the amount of memory used for caching. Even on modern Linux platforms the default kernel settings allow PostgreSQL to allocate not more than 32 MB of (shared) memory. Therefore we first have to adjust the kernel settings, and then increase the `shared_buffers` configuration parameter to 25% of the RAM size; in our case it has been set to 4 GB.

In Table 3.2 we can see the loading times for the Lineitem table with the `SQL COPY` command executed on the server. We have used three Postgres instances with different levels of tuning:

None: no tuning

Checkpointing : WAL-checkpointing behaviour adjusted

Checkpointing + caches : WAL-checkpointing behaviour and the amount of memory used for caching also adjusted

In the third column of Table 3.2 we can see the loading time reduction compared to that of the untuned instance. It shows that adjusting the WAL-checkpointing behaviour alone brings only 5% improvement over the untuned instance, however, when adjusting the amount of memory used for caching as well we can achieve a significant reduction in the loading time: namely 17%.

We can conclude that the most efficient way of loading data into Postgres is by using the `SQL COPY` command, and the latter tuning of the configuration settings.

Tuning	Loading time	Improvement
None	541 sec	0%
Checkpointing	517 sec	5%
Checkpointing + caches	447 sec	17%

Table 3.2: Loading text files into PostgreSQL with changing level of configuration settings tuning.

3.3.3 A Simple Aggregation Query

Let us take a simple aggregation query and execute it both in a DBMS, and with custom-built applications operating on flat files. Our goal is to get a rough idea about how these two approaches compare to each other in terms of performance. For that we are going to use the following subquery of Q22 from the TPC-H benchmark:

```
SELECT AVG(c_acctbal)
FROM customer
WHERE c_acctbal > 0.00
AND SUBSTRING(c_phone, 1, 2)
IN ('[I1]','[I2]','[I3]','[I4]',
    '[I5]','[I6]','[I7]')
```

Listing 3.1: $Q22_{sub}$ - a subquery of TPC-H Q22

In Listing 3.1 I1 ... I7 are numbers randomly chosen without repetition from the range $[10, \ldots, 34]$.

The first system in our evaluation is PostgreSQL. In this case we have to load the data into the database first, which for the Customer table takes 13.4 seconds. Only after doing this can we execute the first query. The second one is a hand-coded C-implementation of the query using the `getline` function of the GNU C library (glibc), which is not shown here due to its simplicity. The third one is an AWK implementation, as listed in the following:

```
1 BEGIN { FS="\t" }
2
3 $6 > 0 && substr($5, 1, 2) <= 16 {
4   customerCount++
5   sum_acctbal = sum_acctbal + $6
6 }
7
8 END { print "The result is: " sum_acctbal /
     customerCount }
```

Listing 3.2: Q22$_{sub}$ implemented in AWK

An AWK program consists of pattern-action pairs. For each input record the patterns are evaluated one-by-one, and in case of a match the corresponding action is executed. This provides the recipe of transforming single-table select-project SQL queries into AWK programs: the selection becomes the pattern, and the projection becomes the corresponding action. In case of aggregation queries we can use the special `BEGIN` and `END` patterns matching the beginning and the end of an input file, respectively, to initialize the variables used for the aggregation, and to perform the aggregation and print the aggregate values after the input has been processed completely. Before processing a file we have to specify the record- and field-separator characters (line 1 in Listing 3.2). With this hint AWK can tokenize the fields for each record, and we can access them as `$i` variables, where i denotes the ordinal number of the field within a record. In the TPC-H Customer table the `c_acctbal` and `c_phone` fields are the 6th and 5th columns, respectively, which explains how we have transformed the WHERE condition of the SQL query from Listing 3.1 into the AWK pattern in line 3 in Listing 3.2.

In this experiment we have used two variants of the flat file as input to the C program and the AWK script: one in tab-separated format and one in fixed-width format. The sizes of the flat files are 233 MB and 308 MB, respectively. A full-scan of the files takes on average 2.12 seconds and 2.67 seconds, respectively. Since Postgres only accepts delimiter-separated files as input to the `SQL COPY` operator, we have not conducted experiments using fixed-width files as input to Postgres.

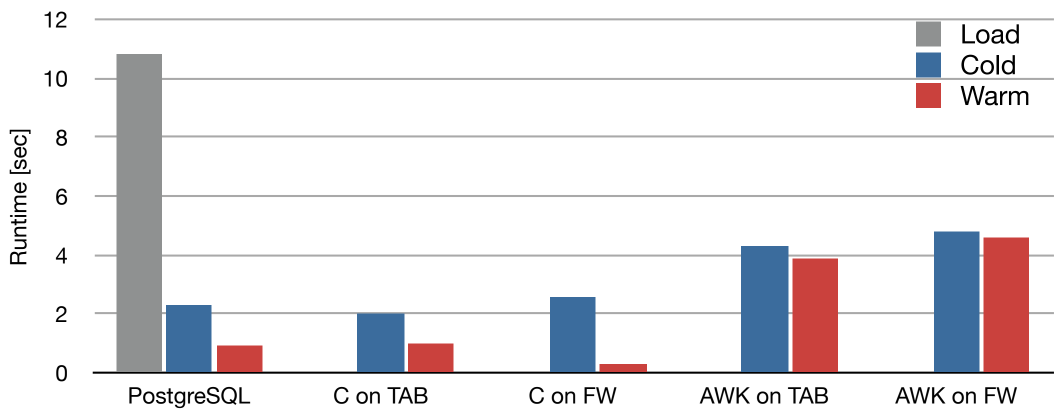


Figure 3.1: Runtimes of the TPC-H Q22-Customer subquery.

The loading- and query runtimes can be seen in Figure 3.1. The runtimes have been measured with cold and warm caches as well, the latter achieved by performing 5 warm-up runs without dropping the caches in between before executing and measuring runtimes for the warm runs. For the tab-separated layout it is interesting to note that for cold caches the hand-coded C-application outperforms PostgreSQL, and in the case of warm caches their performance is the same. For the fixed-width formatted file the hand-coded C-application outperforms PostgreSQL for warm caches by far, and in the case of cold caches their performance is almost the same. The query times using AWK are higher than both that of Postgres and the C application, however, they are still lower than the loading times for Postgres.

When considering the culprit of using Postgres, we can conclude that the relatively high initialisation cost of loading the data into PostgreSQL does not pay off. We have to note, thought, that creating an index on the `c_phone` field could change the overall picture. We can also conclude that the time-to-query, i.e. the sum of the loading time and the runtime of the first query, is significantly lower for both C and AWK, than for PostgreSQL. For performing a quick analysis it is a good choice to take AWK executed against the dump file, in case we perform only a very few number of runs. In terms of coding effort it is easier than implementing it in C, but not as straight-forward as writing a SQL query for that. The fastest system and file-format combination is the hand-coded C-application on files in fixed-width format with warm caches, due to the better cache-locality and reduced CPU costs.

3.3.4 A More Complex Query

On the previous single-table query the hand-coded solutions outperformed the DBMS. In the following we will consider a multi-table query with grouping to see whether the DBMS can perform better on home-ground, more complex queries. For the purpose of evaluation we are going to use Q4 from the TPC-H benchmark:

```sql
SELECT o_orderpriority,
  COUNT(*) AS order_count
FROM orders
WHERE o_orderdate >= '1993-07-01'
AND o_orderdate < '1993-10-01'
AND EXISTS (
  SELECT *
  FROM lineitem
  WHERE l_orderkey = o_orderkey
  AND l_commitdate < l_receiptdate
)
GROUP BY o_orderpriority
ORDER BY o_orderpriority;
```

Listing 3.3: TPC-H Q4

```awk
FILENAME == "lineitem.tbl"
    && $12 < $13 {
    orderkeys[$1] = 1
}
FILENAME == "orders.tbl"
    && $5 >= "1993-07-01"
    && $5 < "1993-10-01"
    && $1 in orderkeys {
        count[$6]++
}
END {
    asorti(count, keys)

    for (i = 1; i <= length(keys); i++)
        print keys[i] " " count[keys[i]]
}
```

Listing 3.4: TPC-H Q4 implemented in AWK

This query is relatively complex to implement in C++, therefore we are only going to use an AWK script in our comparison. Let's assume the flat files for the Lineitem and Orders tables are called lineitem.tbl and orders.tbl, respectively. Providing these files as inputs to AWK in this given order, the script in Listing 3.4 returns the same results as the SQL query executed in a DBMS. The script first collects the qualifying orderkeys from the Lineitem table, reading 7.3 GB of data. The intermediate results have a size of 116 MB, and 13.75M tuples qualify out of the 60M input tuples. After that it reads the Orders table, processing 1.7 GB of data, and performs a hash-based aggregation.

We have executed the query in PostgreSQL with all configuration parameters tuned as described in Section 3.3.2. As a first step we loaded the Lineitem table in 447 seconds, and the Orders table in 92 seconds, which yields a total of 539 seconds spent on loading.

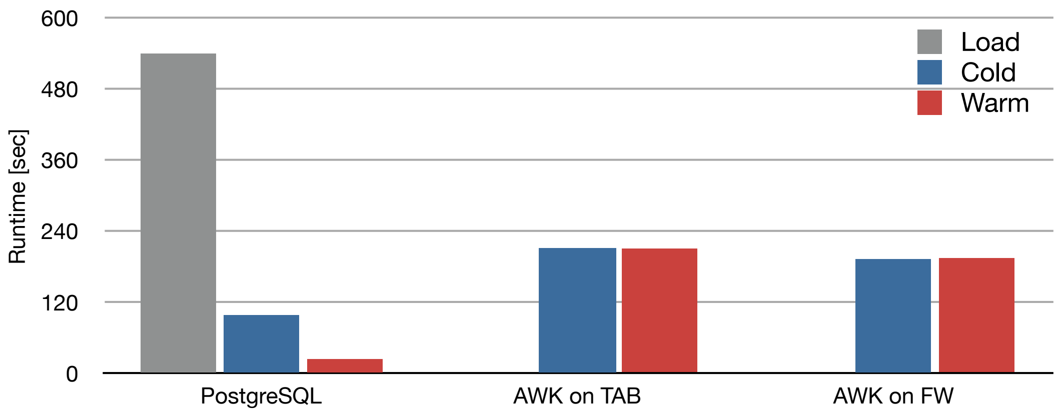


Figure 3.2: Runtimes of the TPC-H Q4 query.

The loading- and query runtimes can be seen in Figure 3.2. The runtimes have been measured with cold and warm caches as well, as described in the previous section. PostgreSQL has the fastest query time with 98 seconds, which is twice as fast as that of the AWK script. However, the time-to-query for PostgreSQL is 638 seconds, which is a significant upfront cost. Nevertheless, owing to the very good query performance it does pay off to use PostgreSQL for complex queries like the one used in this experiment. We can see an additional artefact of DBMSs, namely that they are capable of drastically improving their performance over subsequent runs, which is not the case for flat file processing tools.

3.3.5 The Inherent Costs of Flat File Processing

The processing of flat files can be subdivided into five steps, as described in the introduction. The selection and processing steps operate on binary data, similarly to a DBMS, though specialised processing methods, e.g. vectorisation, usually cannot be applied. Parsing is known to be an expensive step, especially for floating point numbers and dates. Separating the records is a necessary step in any reasonable query, and is a relatively simple step. Tokenisation is far more complex, and includes a lot of branching, which yields branch mispredictions in most of the cases, i.e. for almost every character in the file. In the following we will show the costs inherent to tokenisation in simple queries.

As a first step we are going to investigate the costs of processing a varying number of fields in each input record. In this experiment we read the Lineitem table with scale factor 10 in some text format using AWK, and project the first k fields of each record. Here we are going to show the total cost till the following processing steps:

Input: scanning the data on hard-disk,

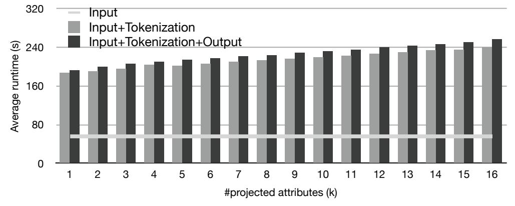
Tokenization: tokenising the fields, and

Output: projecting the fields.

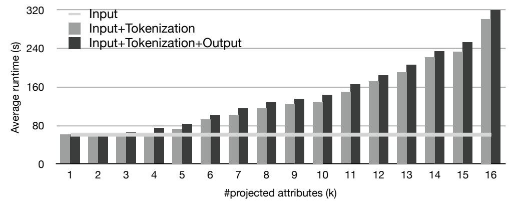
The resulting runtimes for tab-separated input can be seen in Figure 3.3a. What we can immediately notice, is that tokenization is the most expensive step, and it increases only slightly with growing k. To better dissect tokenization, let us consider counting the number of fields in each record, that is using the value of the `NF` built-in variable. Executing this operation takes 184.05 seconds on average, which is slightly lower than the cost of projecting the first field only. We can conclude that the tokenization, i.e. finding the boundaries of the fields within a record has a significant cost, and it has the highest cost in AWK's processing pipeline.

As seen before, AWK is capable of reading from flat files in fixed-width format, too. In this case there is no need for field-separator characters, since the fields are implicitly defined by their position within the file. This could reduce the tokenization cost, but on the other hand it increases the size of the file. Projecting the first k fields of each record again from the Lineitem table in fixed-width format results in the runtimes shown in Figure 3.3b. It has turned out, that AWK clearly benefits from the fixed-width layout, which suggests that AWK is able to exploit the fixed-width format to increase the processing performance. When comparing the runtimes for the tab-separated and the fixed-width format we can conclude that a better processing performance can be achieved if the flat file is in the fixed-width format, especially if only a fraction of the tuples are accessed from each record.

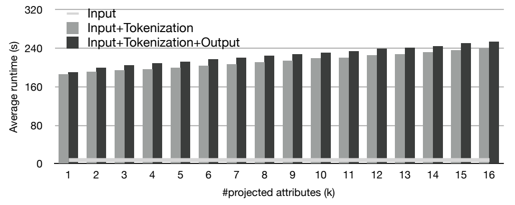
Reducing the tokenization costs was achieved by using a different input format. However, for reducing the cost of reading the input, we have to place the input



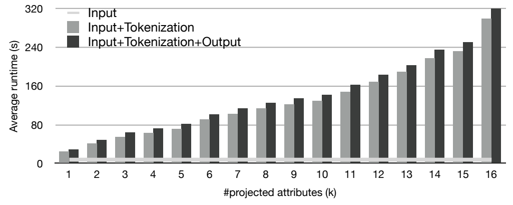
(a) Flat file in tab-separated format, disk-resident



(b) Flat file in fixed-width format, disk-resident

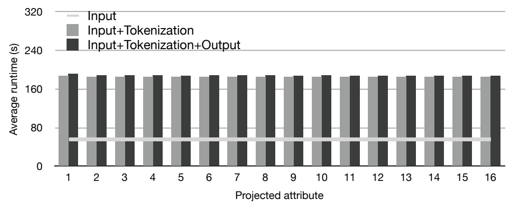


(c) Flat file in tab-separated format, memory-resident (ramfs)

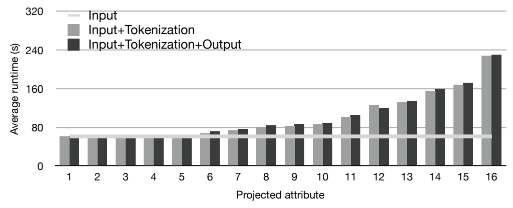


(d) Flat file in fixed-width format, memory-resident (ramfs)

Figure 3.3: Runtimes of projecting the first k fields of each record of the Lineitem table in AWK showing the total costs till each processing step.



(a) Flat file in tab-separated format

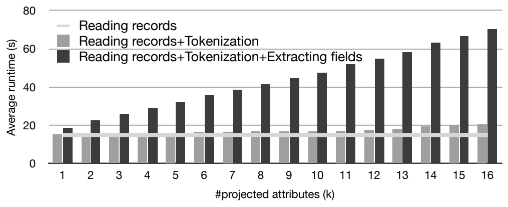


(b) Flat file in fixed-width format

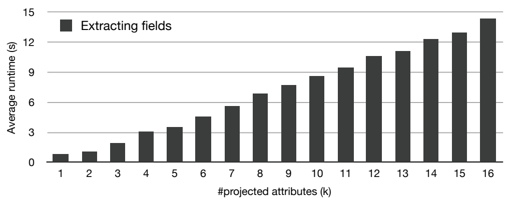
Figure 3.4: Runtimes of projecting the k. field only of each record of the disk-resident Lineitem table in AWK showing the total costs till each processing step.

into main-memory, instead of storing it on disk. Thus, we have repeated the same experiment as in the previous case, but now with the flat file residing in memory, in a `ramfs` partition. In Figures 3.3c and 3.3d we can see the results for both memory-resident input files. As expected, the time it takes to scan through the data is decreased, which allows for a better runtime for fixed-width files for $k < 4$. For all other cases there is no significant difference in the query times. From these results we can conclude that these queries in AWK are CPU-bound.

In our next experiment we have examined the costs of accessing *only the k^{th} field* in each record. The resulting runtimes can be seen in Figure 3.4a. We have to note here, that the cost of accessing a column in the fixed-width format depends on the column's position as well, while for the tab-separated file it does not. As



(a) Flat file in tab-separated format



(b) Flat file in fixed-width format

Figure 3.5: Runtimes of projecting the first k fields of each record of the memory-resident Lineitem table in C++, showing the total costs till each processing step.

in the previous experiment, the fixed-width format yield faster processing here as well.

We can imitate the basic tokenization functionality of AWK in a simple C++ program as well. Separating the records can be done by the `fgets` built-in function, if the record-separator character is the new-line, and separating the records by collecting the positions of the field-separator characters found in the record being processed.

Since we pointed out, that the tokenisation step yields branch mispredictions, it is interesting to measure the costs of this step when all data is pre-loaded into memory, so that we can investigate the pure CPU costs. Projecting the first k fields of each record from the Lineitem table in tab-separated format results in the runtimes shown on Figure 3.5a. Here we show the total cost till the following processing steps: scanning the data in main-memory, tokenising the fields, and copying these fields into local variables. For data in fixed-width format we only have to extract the fields without tokenising, which is shown on Figure 3.5b.

In contrary to AWK, tokenisation does not increase the runtimes significantly in our C++ application. By this we mean the runtime of reading in records and tokenization is not much higher than that of only reading in the records. The reason behind the moderate cost increase is that tokenisation is performed on the records that have just been read, and therefore are likely to reside in the caches. When copying the field values into a different memory location, i.e. extracting fields, we see a high increase in the costs for the tab-separated layout, and a smaller increase for the fixed-width format. The latter is due to the completely predictable memory access pattern over the data stored in fixed-width format. We can conclude from this experiment that when using a custom-built application for processing memory-resident data in textual representation, it is much faster to operate on data in fixed-width format.

3.3.6 Files in Binary Format

Converting a text file into binary would also result in a file with fixed-width fields, and could reduce the size of numeric fields in many cases. However, there is no function available in AWK for reading a serialised binary number, nor are there bit-shifting operators, which makes the deserialization of numbers in AWK costly.

3.4 Conclusions

We have learned that for relatively simple queries operating on a single file only both a custom-built C++ application and AWK can offer a shorter time-to-query than PostgreSQL, with the former being the fastest. For more complex queries operating on multiple files PostgreSQL is the best option. Though it still has the longest time-to-query, it offers a superior query performance on subsequent runs, which makes the initial data loading costs quickly pay off.

We have also shown that the standard configuration settings for PostgreSQL are inappropriate, and using a tuned configuration can reduce the loading time by 17%. Furthermore, if the flat files reside on the server, then we can reduce the loading time by 5% just by using the right loading command.

We have seen that when processing flat files with AWK, it is better if the flat files are stored in fixed-width format. This is especially true, if only a few attributes are accessed from each record, where the increase in performance can be up to factor 8.

Appendix

3.A The C I/O Library

In order to perform file processing tasks we need to use an input-output library. A widely known one to programmers is the I/O library of the C programming language. It originates from the early years of UNIX, and is the first device-independent model of input and output, capable of reading and writing files and devices as well. We are going to discuss this library in the following.

Input and output to and from both devices and files are mapped into logical data streams. Two forms of mappings are supported: text streams and binary streams. A text stream consists of one or more lines, consisting of zero or more characters plus a terminating new-line character. A binary stream on the other hand is merely an ordered sequence of characters. Apart from the semantic differences, there are functional ones as well between the two stream types: some methods of the I/O library work differently on each stream.

We can perform the following operations (among others) on streams: open, close, read, write, and flush. The prerequisite of performing any kind of I/O on the stream is to open it. When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device, e.g. a terminal. Therefore, the I/O library at hand already performs buffered I/O for files. The size of the associated buffer is implementation dependent, however, it is also possible to associate a user-allocated buffer to a stream.

Chapter 4

Computer Systems Performance Analysis

Performance is a key aspect of every system in computer science. As such, it is often required to build a system that has the highest performance at a given cost. Often we have multiple solutions for a given problem, and we have to compare their performance to choose the most efficient one. The comparison is done along some metrics of the system, e.g. running time, latency, throughput, memory usage, etc. The metrics themselves are either measured or estimated. In the former case the measurements are repeated multiple times to reduce the possibility of measurement errors, and to increase the confidence of the results. To summarize the measurements, some statistics of the data are provided, which is most often the average value, i.e. the mean. This, however, is not the most reliable statistics, and thus researchers should favour other statistics instead.

When comparing a few algorithms that solve a given problem, we usually do not have a lot of measurements available. We typically collect the runtime or memory usage of the different algorithms, apply some statistics over the measurements, and choose the one with the best result. However, when we also consider multiple factors that could have an influence on the performance of these algorithms, we end up adding new dimensions to the problem space. Eventually, managing the experimental results becomes a problem. In this chapter we present a uniformly applicable method for storing experimental results and problem dimensions in a relational database. We describe a flexible way of exploring the effects of the problem dimensions on the performance in a statistically sound way.

4.1 Performance Analysis

4.1.1 Motivating Example

The most common way of measuring the performance of algorithms, systems, or components in the database community is to report the average runtime out of 3 or 5 runs. Let's look at an example: assume we measured runtimes of a query when executed against two different layouts. Layout A has an average runtime of 1.75 seconds and Layout B of 1.82 seconds. In this case we would clearly declare Layout A as superior to Layout B.

However, if we take a look at the runtimes of all 5 runs in Figure 4.1, we can see that Layout A has a high variance (0.06), whereas the query time for Layout B is rather stable (its variance is 0.00075). Most system designers would probably prefer Layout B, due to its performance being more predictable. This example demonstrates that reporting the average runtime alone is not sufficient for comparing two solutions [34, Chapter 13]. Therefore at a minimum the variance or standard deviation of the sample should be provided along with the average to get a proper description of the sample.

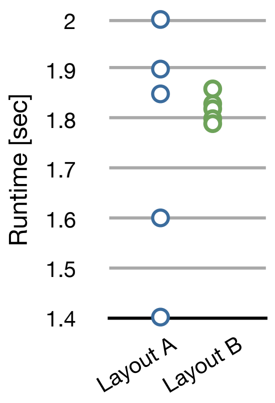


Figure 4.1: Query times for two different layouts, each measured five times

We should keep in mind that when experimentally comparing multiple systems, we only get a *sample* of their performance metrics which can only be used to *estimate* the populations' performance metrics. Thus, there is always a level of uncertainty in our estimates, which renders the necessity of expressing this uncertainty in some way. One possible way to do this is to use confidence intervals, which express the following in natural language: "There is a 95% chance that the actual average runtime of System A is between 1.7 and 1.8 seconds."

4.1.2 Confidence Intervals

To create a confidence interval we first have to choose our confidence level, typically 90%, 95% or 99%, denoted by $1-\alpha$, where α is called the significance level. We require the sample size n, the sample mean $\overline{x}$, sample standard deviation σ, and the significance level α. Then the confidence interval is defined as follows: $(\overline{x} - C \times \frac{\sigma}{\sqrt{n}}, \overline{x} + C \times \frac{\sigma}{\sqrt{n}})$, where C is the so-called confidence coefficient. The choice of the confidence coefficient is determined by the sample size [34, p. 206]. If we have a large sample ($n \geq 30$), we can use the $1-\alpha/2$-quantile of the standard normal distribution for the confidence coefficient: $C = Z_{1-\alpha/2}$. However, in experiments we usually run only 5 measurements, thus we have a sample size of $n = 5$. Therefore, we should only use the $1 - \alpha/2$-quantile of the Student's t-distribution with $n - 1$ degrees of freedom: $C = t_{[1-\alpha/2,n-1]}$. The prerequisite is that the population needs to have a normal distribution, which is a fair assumption for our runtime measurements. For instance, the 95% confidence intervals for our example in Figure 4.1 are: (0.23, 3.27) for Layout A, and (1.65, 1.99) for Layout B. This makes Layout B a safer choice, if predictability is of great importance for the system designer. (See [34, Chapter 13] for details.) When looking at the measured query times on Layout A in Figure 4.1, we can see that the relatively wide confidence interval for Layout A is due to the large variance of the sample: the points are scattered out across the (1.4, 2.0) interval. However, a sample can have a large variance even if most measured values are "near" to each other, and only a few of them having a higher or lower value than the rest. These latter are called outliers.

4.1.3 Outlier Detection

An outlier is an element of a sample that does not "fit" into the sample in some way. It is hard to quantify the criteria for labelling an element as an outlier, and it also depends heavily on the use-case. Therefore, the most common technique used for detecting outliers is plotting the sample on a scatter plot, and visually inspecting the plot by a human. If we assume, that there is only one outlier in the sample, and it is either the minimum, or the maximum value, then we can use Grubbs' test [24] to automatically detect outliers. The only problem is that this method tends to identify outliers too often for samples with less than eight elements. To counter the error rate of the method we have included an additional condition for labelling an outlier: $\text{margin\_or\_error}/\overline{x} \geq 2.5\%$, where the margin of error is defined as the radius of the confidence interval. This reduces the detection rate to 3% in the experiments in Chapter 5, and those elements proved to be outliers after manual inspection.

4.1.4 Choosing the Best Solution when there is no Single Best Solution

Choosing the best solution using the average runtime is easy, we simply take the one with the smallest one. We have also seen that this can be arbitrarily wrong, and that is why confidence intervals provide a better basis of comparison than the sample mean. However, comparing confidence intervals is not that straight-forward as comparing scalars. If two intervals are disjoint, they are easily comparable. It they are not disjoint, and the mean of one sample is inside the other sample's confidence interval, they are indistinguishable from each other with the same level of confidence, as that of the intervals. Finally, if they are not disjoint, but their means do not fall into the other sample's interval, an independent two-sample t-test (Welch's t-test [62]) can decide whether they are distinguishable, and if so, which one is better.

4.2 A Framework for Statistical Analysis of Experimental Results

Experiments in computer science usually have the goal of measuring the performance of a couple of algorithms, methods, or the effects of their parameters. The experimenting itself often consists of changing a single parameter and analysing the effects of that change. As the number of parameters, methods, algorithms, and the such, increases it becomes increasingly difficult to analyse the results. This is when it makes sense to collect and store the experimental results in a database. Such a database can be composed of several dimension tables — one for each parameter — and a single fact table, thus it is essentially a star-schema database. As it is a necessity to repeat experiments 3 or 5 times, we need a time dimension as well. The fact table might have more than just a single attribute, depending on how many attributes are needed to characterise the performance. Such a schema is illustrated in Figure 4.2. Note that for simplicity we have excluded the keys of the dimension tables in the figure.

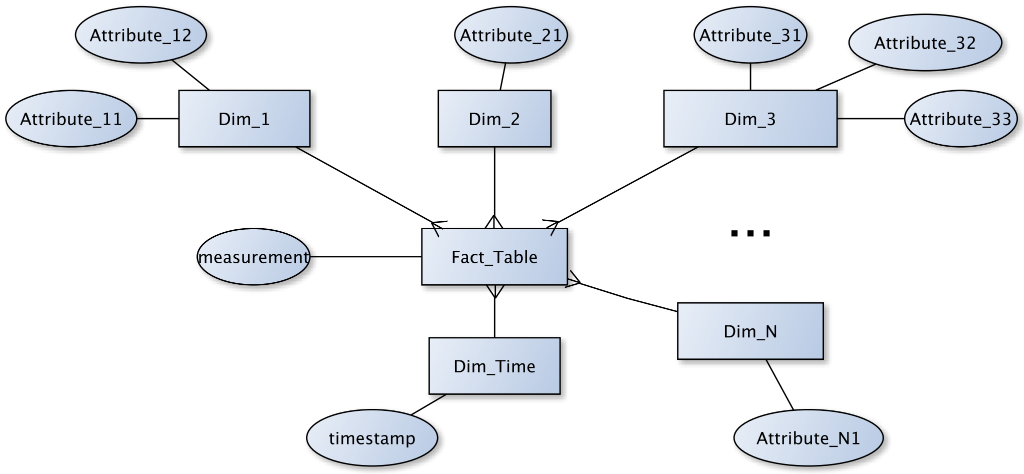


Figure 4.2: The general schema of a star-schema database for storing experimental results.

As mentioned above, we often need to investigate what happens if we change some parameter values. However, a changed parameter value might have a different effect on the performance of each method, algorithm, or system tested. Thus, what we would alternatively like to see is the effect of changing some parameters on the performance of *each* method, algorithm, or system. To answer that question we can create a view on top of the (possibly filtered) fact table, containing four columns: problem key, solution key, timeID, and measurement, as shown in Listing 4.3. Here, the two key columns are compound keys formed by combining some foreign key fields in the fact table. The solution key identifies the parameters, whose values are changed in the experiment, while the problem key identifies the methods, algorithms, or systems, that are affected by this change. To create such a view we can use the SQL query in Listing 4.4.

```
analysis_view (
  problem_key INTEGER[],
  solution_key INTEGER[],
  timeID INTEGER,
  measurement DOUBLE PRECISION
)
```

Figure 4.3: The schema of the view representing the input of the analysis.

```
CREATE VIEW analysis_view AS
SELECT ARRAY[Dim_1_key, Dim_2_key, ...] AS
   problem_key,
  ARRAY[Dim_4_key, Dim_5_key, ...] AS
     solution_key,
  timeID, measurement
FROM fact_table;
```

Figure 4.4: The SQL query for performing the mapping from the fact table.

4.2.1 Calculating statistics in SQL

We have mentioned several statistical functions and methods, which are not available in a typical relational database system. It would be most straight-forward and efficient to implement these in a language for statistical analysis, and call the implemented functions from SQL. Fortunately, Postgres has support for calling functions implemented in R, thus we have been using this system setup. We have been using PL/R [16], a language extension to PostgreSQL that allows us to write PostgreSQL functions and aggregate functions in the R language for statistical analysis.

With PL/R we can create Postgres functions that can call any function available in R. We can even make the R-function to be called a parameter of the Postgres function, as shown in Listing 4.5.

```
CREATE OR REPLACE FUNCTION rsql_call_func(
  x FLOAT8[],
  function_name VARCHAR) RETURNS FLOAT8 AS
  'do.call(function_name, list(x))'
LANGUAGE 'plr' STRICT;
```

Figure 4.5: The PL/R function for calling an arbitrary R-function that operates on vectors.

For calculating confidence intervals we need to determine the sample mean and the margin-of-error. The latter one cannot be calculated using plain SQL stored procedures or functions. Thus, we have implemented it also in PL/R, which is shown in Listing 4.6.

```
CREATE OR REPLACE FUNCTION
   rsql_margin_of_error(x FLOAT8[]) RETURNS
   FLOAT8 AS
  'n = length(x)
  qt(0.975,df=n-1) * sd(x) * sqrt(n) '
LANGUAGE 'plr' STRICT;
```

Figure 4.6: The PL/R function for calculating the margin-of-error of a set of measurements.

Outliers can distort some of the statistics, and thus it would be favourable to remove them before calculating confidence intervals. As mentioned in Section 4.1.3, this can be done by using Grubb's test for outlier detection. The implementation is given in Listing 4.7.

When choosing the best solution to a problem, we are going to adapt our search to consider confidence intervals. When comparing two intervals we will essentially

```
CREATE OR REPLACE FUNCTION rsql_remove_outlier
    (x FLOAT8[]) RETURNS FLOAT8[] AS '
  library(outliers)
  testResult = grubbs.test(x, two.sided = TRUE
     )
  n = length(x)
  margin_of_error = qt(0.975, df=n-1) * sd(x)
     * sqrt(n)

  if (testResult[''p.value''] <= 0.05 &&
     margin_of_error / mean(x) >= 0.025) {
    outlier = strsplit(grubbs.test(x, two.
       sided = TRUE)$alternative, " ")[[1]][3]
    x = setdiff(x, outlier)
  }

  x
' LANGUAGE 'plr' STRICT;
```

Figure 4.7: The PL/R function for removing outliers using Grubb's test.

decide whether they are statistically indistinguishable. This can be done by the two-sample t-test, implemented in Listing 4.8.

```
CREATE OR REPLACE FUNCTION rsql_t_test(
  x FLOAT8[],
  y FLOAT8[])
  RETURNS BOOLEAN AS '
    t.test(x,y)[''p.value''] >= 0.05
' LANGUAGE 'plr' STRICT;
```

Figure 4.8: The PL/R function for calculating the two-sample t-test.

4.2.2 Finding the best solution of each problem

After we have successfully identified the problem- and solution dimensions, and created the analysis input accordingly, we can start analyzing the results. As a first step we calculate the mean, standard deviation, and median values over the multiple runs of the same experiment. We also calculated the confidence intervals, where we rely on the PL/R functions defined in the previous section. This is represented in the form of a database view, shown in Listing 4.9. Building on this view we can remove the outliers and improve the calculated statistics and narrow the confidence intervals. This is also represented as a database view, shown in Listing 4.10.

Finally we can determine the set of best solutions for each problem. This is done in two steps. First we search for the solutions of each problem, that have the lowest sample mean. This is shown in Listing 4.11. After that we compare all solutions of a problem to the one having the lowest sample mean collected in the first step, and decide using the two-sample t-test whether they are indistinguishable at the same level of confidence as that of the confidence intervals, which is 95% in our example. This database view provides the set of indistinguishable best solutions for each problem. This is shown in Listing 4.12.

```
CREATE OR REPLACE VIEW v_experiment_stats AS
  SELECT problem_key, solution_key, timeid,
    array_accum(measurement) as runtimes,
    rsql_call_func(public.array_accum(
       measurement), 'median') AS
       MedianRuntime,
    AVG(measurement) AS avg_runtime,
    stddev_samp(measurement) AS stdev_runtime,
    rsql_margin_of_error(array_accum(
       measurement)) AS
       margin_of_error_runtime
  FROM analysis_input
  GROUP BY problem_key, solution_key, timeid;
```

Figure 4.9: Creating the view for calculating statistics on each experiment.

```
CREATE OR REPLACE VIEW
   v_experiment_stats_outliers_removed AS
  SELECT problem_key, solution_key, timeid,
    rsql_call_func(runtimes, 'median') AS
       MedianRuntime,
    rsql_call_func(runtimes, 'mean') AS
       avg_runtime,
    rsql_call_func(runtimes, 'sd') AS
       stdev_runtime,
    rsql_margin_of_error(runtimes) AS
       margin_of_error_runtime,
    runtimes
  FROM (
         SELECT
           problem_key, solution_key, timeid,
           rsql_remove_outlier(runtimes) AS
              runtimes
         FROM v_experiment_stats
       ) AS t;
```

Figure 4.10: Creating the view for removing outliers.

```
CREATE OR REPLACE VIEW v_problem_minimal_mean
   AS
  SELECT problem_key, solution_key, runtimes,
    MedianRuntime, avg_runtime,
    margin_of_error_runtime
  FROM (
         SELECT problem_key, solution_key,
            runtimes,
           MedianRuntime, avg_runtime,
           margin_of_error_runtime,
           MIN(avg_runtime) OVER
             (PARTITION BY problem_key) AS
                mean_min
         FROM
            v_experiment_stats_outliers_removed

       ) AS t
  WHERE avg_runtime = mean_min;
```

Figure 4.11: The SQL commands for finding the best solution of each problem.

```
CREATE VIEW v_problem_bests_using_mean AS
  SELECT vl.problem_key, vl.solution_key,
    vb.MedianRuntime,
    ARRAY[vl.avg_runtime - vl.
       margin_of_error_runtime,
     vl.avg_runtime + vl.
        margin_of_error_runtime]
     as confidence_interval
  FROM v_experiment_stats_latest vl
    INNER JOIN problem_minimal_mean vb
      ON (vb.problem_key = vl.problem_key)
  WHERE public.rsql_t_test(vl.runtimes, vb.
     runtimes);
```

Figure 4.12: The SQL commands for finding the set of indistinguishable best solutions per problem.

Chapter 5

Runtime Fragility of Hand-coded Queries in Main Memory

In this chapter we investigate the following problem: Given a database workload (tables and queries), which data layout (row, column or a suitable PAX-layout) should we choose in order to get the best possible performance? We show that this is not an easy problem. We explore careful combinations of various parameters that have an impact on the performance including: (1) the schema, (2) the CPU architecture, (3) the compiler, and (4) the optimization level. We include a CPU from each of the past four generations of Intel CPUs.

In addition, we demonstrate the importance of taking variance into account, when deciding on the optimal storage layout. We observe considerable variance throughout our measurements which makes it difficult to argue along means over different runs of an experiment. Therefore, we compute confidence intervals for all measurements and exploit this to detect outliers and define classes of methods that we are not allowed to distinguish statistically. The variance of different performance measurements can be so significant that the optimal solution may not be the best one in practice. Our results indicate that a carefully or ill-chosen compilation setup can trigger a performance gain or loss of factor 1.1 to factor 25 in even the simplest workloads: a table with four attributes and a simple query reading those attributes.

Besides the compilation setup, the data layout is another source of query time variance. Various size metrics of the memory subsystem are round numbers in binary, or put more simply: powers of 2 in decimal. System engineers have followed this tradition over time. Surprisingly, there exists a use-case in query processing where using powers of 2 is always a suboptimal choice, leading to one more cause of fragile query times. Using this finding, we will show how to improve tuple-reconstruction costs by using a novel main-memory data-layout.

The results of this chapter have been accepted for publication in IMDM@PVLDB, a peer-reviewed workshop [50].

5.1 Introduction

The two most common data layouts used in todays database management systems are row and column layout. These are only the two extremes when vertically partitioning a table. In-between these extremes there exists a full spectrum of column-grouped layouts, which under certain settings can beat both of the aforementioned traditional layouts for legacy disk-based row-stores [36]. However, for main-memory systems column grouped layouts have not proved to be of much use for OLTP workloads [25], unless the schema is very wide [53].

Another axis of partitioning a table is horizontal partitioning, where the partitions are created along the tuples instead of along the attributes. This is usually based on the values of an attribute with low cardinality, e.g. geographical regions, but this is not a strict requirement. Forming horizontal partitions can also be done by simply taking repeatedly k records from the table, which we will call chunks in the following. Within a horizontal partition we can have any vertically partitioned layout, including row and column as well. One notable example in disk-based database systems is the PAX-layout [6], where the horizontal partitions have a size that is the multiple of the hard disk's block size, and inside these partitions the tuples are laid out in column layout.

We can apply a similar strategy in main memory as well, however, we have more freedom in choosing the size of the horizontal partitions. Therefore in main-memory we can simply form so-called chunks of the table by repeatedly taking k records from the table and laying them out in column layout *within* the chunk. We denote this layout by memPAXk. In this sense, row layout is the same as memPAX1, and column layout is equivalent to memPAXn, where n is larger or equal to the cardinality of the table. The chunks of these layouts are analogous to PAX pages [6], however, there are two important differences: (1) we can choose any chunk size (in bytes or tuples) that is a multiple of the tuple size, while for PAX we are restricted to multiples of the disk's block size, and (2) we neither store any helper data structures per chunk, nor use mini-pages as in the disk-based PAX-layout. The possible memPAX layouts of a table having 2 columns and 8 records, and using chunk sizes of powers of 2 are illustrated in Figure 5.1. Here we can see the two extremes: row- and column layout, and memPAX layouts with a chunk size of 2- and 4 tuples.

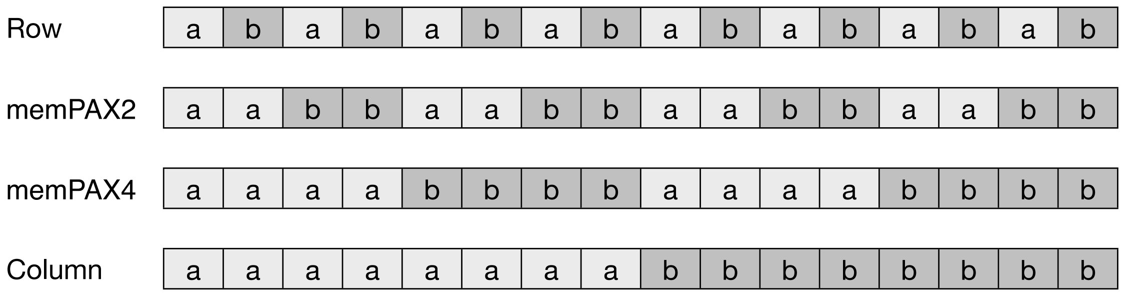

Row
a b a b a b a b a b a b a b a b
memPAX2
a a b b a a b b a a b b a a b b
memPAX4
a a a a b b b b a a a a b b b b
Column
a a a a a a a a b b b b b b b b

Figure 5.1: memPAX layouts of a table having 2 columns and 8 records, considering powers of 2 chunk sizes.

5.2 The six-dimensional Parameter Space of our Experiments

We are going to explore a six-dimensional parameter space of a fairly simple workload: a table with four attributes and two simple queries reading those attributes. The whole experiment is conducted on memory resident tables, and using hand-coded queries implemented in C++. We are going to refer to this workload as our micro-benchmark. In the following we specify the dimensions:

(1) The datatype used in the schema. Our dataset is a single memory-resident table with four integer columns, with a total size of 10 GB. Depending on the data type chosen (1-byte, 4-byte, or 8-byte integers denoted by `int1`, `int4`, and `int8`, respectively) we get the following scenarios:

Label	Schema	Tuple count
char	`(a int1, b int1, c int1, d int1)`	$2560 * 1024^2$
int	`(a int4, b int4, c int4, d int4)`	$640 * 1024^2$
long	`(a int8, b int8, c int8, d int8)`	$320 * 1024^2$

Table 5.1: The schemas used in our experiments

(2) The presence of conditional statements in the query code. We use two queries requiring all tuples to be reconstructed for processing as shown in Figure 5.2. Q1 performs a minimum-search on the sum of all attributes of a tuple, which being a conditional expression yields a branch in the implementation. We have tried out a branch-free implementation of the min[1] calculation as well, which, however, was consistently slower. Q2 on the other hand performs a branchless cal-

[1]min = min XOR ((temp XOR min) AND NEG(temp ¡ min));

culation: it sums up the product of the attribute values of each tuple. Since Q2 has no branches, the measured query times are not affected by branch-mispredictions.

```
Q1: SELECT MIN(a+b+c+d) FROM T;
Q2: SELECT SUM(a*b*c*d) FROM T;
```

Figure 5.2: The queries used in the experiments

(3) The CPU architecture. The performance characteristics of a main-memory database system are influenced the most by the machine's CPU. As there are usually significant changes between the subsequent CPU architectures, we have chosen machines equipped with Intel CPUs of four subsequent architectures, all running Debian 7.8.0 with Linux kernel version 3.2.0-4-amd64 as shown in Table 5.2, with hyper-threading either disabled or not supported.

CPU	**Architecture**	**RAM**	
Xeon 5150	Core	16 GB DDR2	@ 266 MHz
Xeon X5690	Westmere	192 GB DDR3	@ 1066 MHz
Xeon E5-2407	Sandy Bridge	48 GB DDR3	@ 1333 MHz
Xeon E7-4870 v2	Ivy Bridge	512 GB DDR3	@ 1600 MHz

Table 5.2: The machines used in our experiments

(4) The compiler. In our experiments we have chosen the three most commonly used compilers[2]: clang (3.0-6.2), gcc (Debian 4.7.2-5), and icc (15.0.0). clang and gcc are both open-source, while icc is proprietary software. clang is actually a C-compiler front-end to the LLVM compiler infrastructure. It compiles C, Objective-C, and C++ code to the LLVM Intermediate Representation (IR), similar to other LLVM front-ends, which allows for a massive set of optimizations to be performed on the IR before translating it to machine code. GCC is short for GNU Compiler Collection, a compiler supporting among others the C/C++ language. It support almost all hardware platforms and operating systems, and it is the most popular C/C++ compiler, and also the default one in most Linux distros. Intel's C/C++ compiler can take advantage of Intel's insider knowledge on Intel CPUs. It is said to generate very efficient code especially for arithmetic operations.

(5) The optimization level. We intuitively expect to get higher performance from higher optimization levels, yet there is no guarantee from the compiler's side

[2]More precisely their C++ front-ends: clang++, g++, and icpc

that this will also hold in practice. Thus, we have decided to evaluate all three standard optimization levels: `-O1`, `-O2`, and `-O3`.

(6) Compile time vs. runtime layouts. The tables in our dataset are physically stored in a one-dimensional array of integers, using a linearisation order confirming to one of the layouts described in Section 5.1. Any query fired against this dataset needs to take care of determining the (virtual) address of any attribute value, and possibly reconstructing tuples as well. To do this it is required to know the chunk size, which can either be specified prior to compiling a given query, i.e. at compile time, or only provided at runtime.

To allow for any compiler optimization to take place, we have been extensively using templates to create a separate executable for each element of the parameter space, i.e. we have an executable for every dataset, query, machine, compiler, O-level, and layout. In case of compile time chunk sizes we have created a separate executable for each chunk size, while for runtime memPAX layouts only a single generic one. The generic executable processes the query chunk by chunk, for which it needs the chunk size provided the latest at runtime. For smaller chunk sizes this approach has an inherent CPU-overhead caused by the short-living loops.

5.3 Managing the Experimental Results

We have seen in Section 5.2 that there are 6 dimensions in the parameter space of our experiments. We actually need to add a time dimension as well, as we have executed 5 runs of each experiment. To make analysing our experimental results easier, we have created a database for storing the results. The schema of this database can be seen in Figure 5.3. It is basically a star-schema database with a dimension table for each dimension of the parameter space, and additionally a dimension table for the time. We have a single fact table storing the measured query times. The `Dim_Implementation` table is used for storing whether the chunk size was provided at compile time or only at runtime.

We have introduced a framework for statistical analysis of experimental results in Section 4.2. To take advantage of it, we first have to map our fact table to the framework's fact table (`experiments.fact_table`). There we have to split the foreign key fields of our fact table into two groups: *problem_key* and *solution_key*. As we are looking for the best way to execute a given query on a given machine, and on a given table, our *problem_key* is going to be composed of the following tuple: (machineid, datasetid, queryid). The "best way" itself is described by the (compilerid, layoutid, implementation) tuple which thus becomes the *solution_key*. Therefore, we can perform the mapping by executing the query in Figure 5.4.

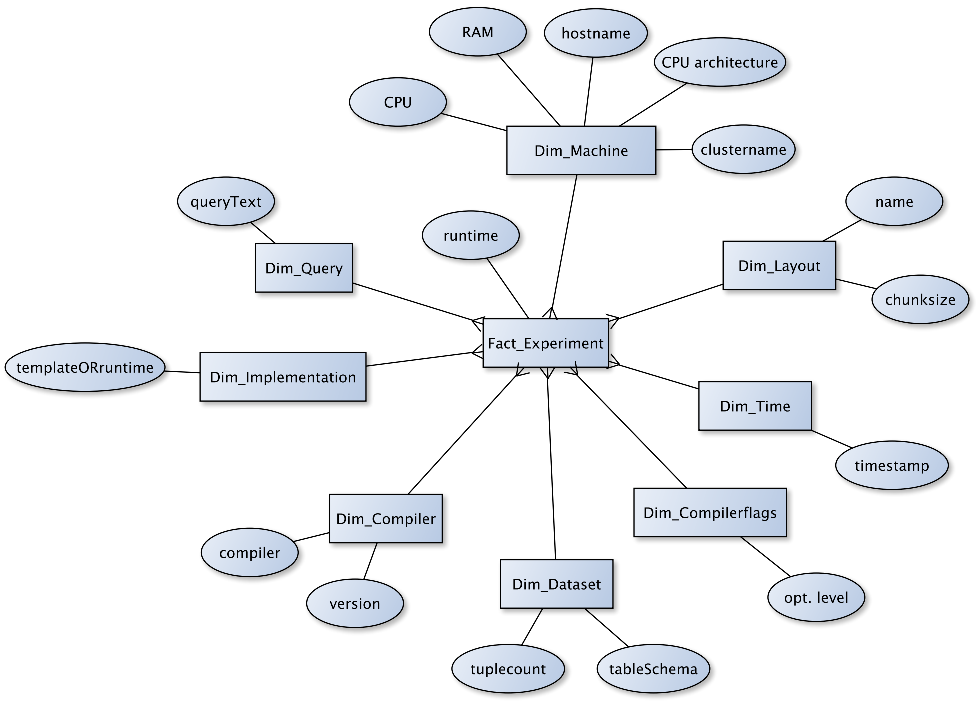


Figure 5.3: The schema of the database storing the measured query times.

Once we have the best solution for each problem, we need to map the compound keys back to the foreign keys used in the fact table of our experiments. This can be done by splitting the two array fields in the framework's view calculating the best solutions (`experiments.v_problem_bests`), as shown in Figures 5.5.

```
INSERT INTO experiments.fact_table
  (problem_key, solution_key,
   timeid, measurement)
  SELECT
    ARRAY[i.experimentid::INT,
     machineid,
     datasetid,
     queryid] AS problem_key,
    ARRAY[compilerid,
     compilerflagsid,
     layoutid,
     implementationid] AS solution_key,
    timeid,
    runtime AS measurement
  FROM mempax.fact_mempax_experiment f JOIN
    mempax.dim_mempax_experiment i
    ON i.id = f.implementationid;
```

Figure 5.4: The SQL query for performing the mapping between the fact tables.

```
SELECT
  problem_key[1]::INT AS experimentID,
  problem_key[2] AS machineid,
  problem_key[3] AS datasetid,
  problem_key[4] AS queryid,
  solution_key[1] AS compilerid,
  solution_key[2] AS compilerflagsid,
  solution_key[3] AS layoutid,
  solution_key[4] AS implementationid,
  medianruntime, confidence_interval
FROM experiments.v_problem_bests;
```

Figure 5.5: The SQL query for mapping the `problem_key` and `solution_key` fields of the best solutions calculated by our framework to foreign key fields in our database.

5.4 Micro Benchmark Results

Table 5.3 displays our recommendations for choosing a data layout and implementation strategy for each machine, schema and query[3]. The best solutions are described by the compiler, optimization level, data layout, and the time the chunk size should be provided (R for runtime- and C for compile-time layouts). We also show the 95% confidence interval of the query time of the best solution. Notice that in some cases there are multiple best solutions. This is a direct consequence

[3]We have noticed that varying the chunk size of the memPAX layouts between 2^{16} and the biggest possible one does not make a significant difference in the query times, regardless of the query, machine, and compiler. Thus, we have excluded those results from our discussion.

Machine	Table schema	Query	Compiler	Opt. level	Layout name	Chunk size provided at	95% confidence interval of query time
Core	char	Q1	icpc	O3	memPAX32	R	(3.39, 3.391)
			icpc	O2	memPAX32	R	(3.389, 3.391)
		Q2	g++	O2	memPAX1	C	(4.655, 4.658)
			icpc	O1	memPAX1	C	(4.656, 4.657)
			g++	O3	memPAX2	C	(4.655, 4.658)
			g++	O3	memPAX4	C	(4.656, 4.657)
			g++	O2	row	n/a	(4.656, 4.657)
			icpc	O1	row	n/a	(4.656, 4.658)
	int	Q1	icpc	O1	row	n/a	(3.245, 3.246)
		Q2	icpc	O1	memPAX1	C	(3.195, 3.196)
	long	Q1	icpc	O1	row	n/a	(3.089, 3.092)
		Q2	clang++	O1	memPAX2	C	(3.081, 3.083)
Westmere	char	Q1	g++	O3	memPAX4096	R	(0.956, 0.958)
			g++	O3	memPAX4096	C	(0.956, 0.958)
		Q2	g++	O1	memPAX8192	C	(3.043, 3.048)
	int	Q1	icpc	O1	memPAX1024	C	(1.003, 1.011)
		Q2	icpc	O1	memPAX1024	C	(1.002, 1.007)
	long	Q1	icpc	O1	memPAX512	C	(0.921, 0.924)
		Q2	g++	O1	memPAX512	R	(0.921, 0.926)
Sandy Bridge	char	Q1	icpc	O2	memPAX8192	C	(1.431, 1.441)
		Q2	g++	O2	memPAX8192	C	(4.006, 4.008)
	int	Q1	icpc	O1	memPAX4096	C	(1.517, 1.519)
		Q2	g++	O2	memPAX1024	C	(1.51, 1.522)
			g++	O2	memPAX2048	C	(1.51, 1.52)
			g++	O1	memPAX2048	C	(1.511, 1.522)
	long	Q1	icpc	O1	memPAX512	R	(1.34, 1.375)
		Q2	g++	O1	memPAX1024	C	(1.331, 1.357)
Ivy Bridge	char	Q1	icpc	O2	memPAX4096	C	(1.246, 1.253)
		Q2	g++	O2	memPAX4096	C	(3.841, 3.844)
	int	Q1	icpc	O1	memPAX1024	C	(1.28, 1.29)
		Q2	clang++	O3	memPAX1024	C	(1.28, 1.289)
			g++	O1	memPAX1024	C	(1.281, 1.287)
			clang++	O1	memPAX1024	C	(1.281, 1.289)
			clang++	O2	memPAX1024	C	(1.28, 1.287)
			g++	O2	memPAX1024	R	(1.28, 1.288)
			g++	O2	memPAX1024	C	(1.28, 1.289)
	long	Q1	icpc	O2	memPAX512	R	(1.199, 1.208)
			icpc	O3	memPAX512	R	(1.2, 1.207)
		Q2	clang++	O1	memPAX512	R	(1.249, 1.256)

Table 5.3: The best layouts and most efficient ways of implementing a query in our micro-benchmarks

Machine	char		int		long	
	Q1	Q2	Q1	Q2	Q1	Q2
Core	6.8	3.3	1.3	1.4	1.1	2.0
Westmere	25.0	3.8	1.9	2.5	1.0	1.3
Sandy Bridge	13.2	3.6	1.6	2.4	1.5	1.9
Ivy Bridge	14.4	3.3	1.7	2.6	1.3	1.7

Table 5.4: The performance drop between worst and best query times for each experiment in our micro-benchmarks. We observe up to a factor 25 difference in runtime.

of using confidence intervals on the query times when choosing the best solutions. Looking at these results the question arises: what influences the choice of best solution? Thus, let us investigate the connection between the elements of the parameter space and the best layout.

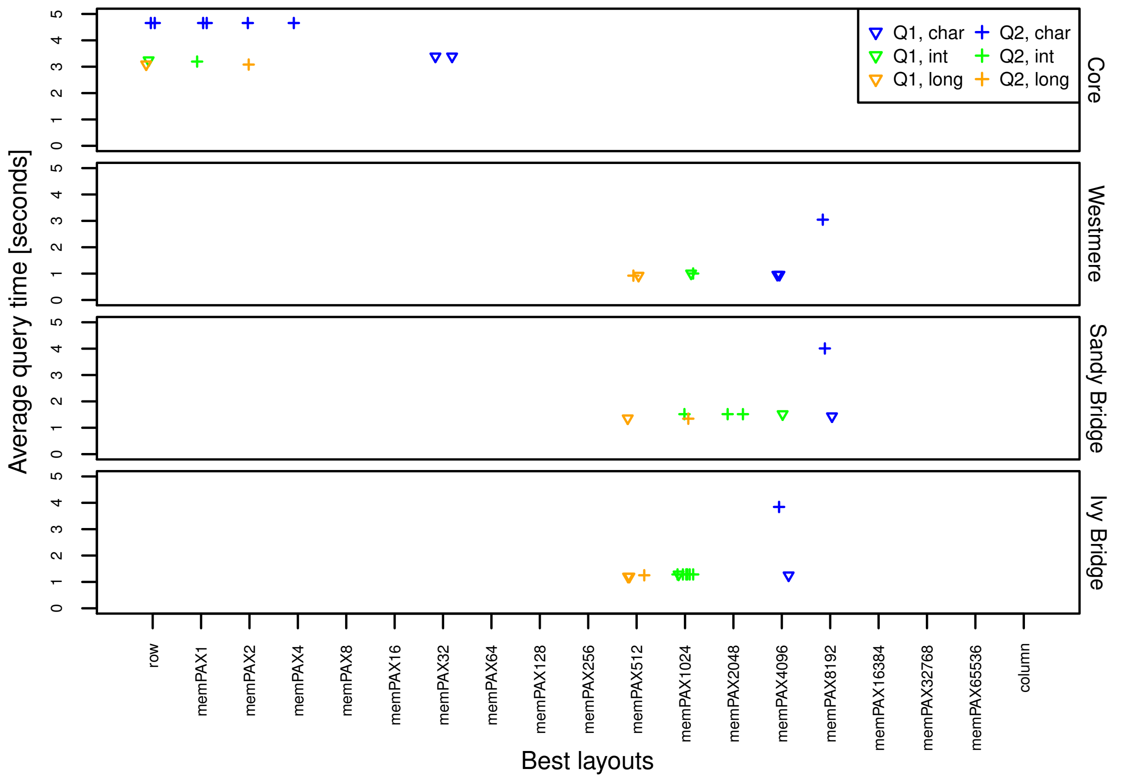


Figure 5.6: Best layouts and their query times. Drilled-down along machine, schema, and query.

In Figure 5.6 we can see the query times of the best layouts, drilled-down along machine, schema, and query. We can immediately notice the radical difference between Core and the other three CPU architectures. The oldest one, Core, prefers layouts with smaller chunk sizes, i.e. close to row layout. The three newer ones on the other hand prefer larger chunk sizes, i.e. close to column layout. For the latter CPUs we can further notice that the best layout for a dataset is often the one, where the following holds: k `* attribute_size * tuple_count = 4KB`, $k \in \{1 \ldots$ `attribute_count`$\}$ — which is when the chunk or an attribute's column inside a chunk perfectly fits the memory page: memPAX4096 for `char`, memPAX1024 for `int` and memPAX512 for `long`.

So far we have seen the best solutions, but have not talked about the performance of the other ones. In Table 5.4 we show the performance drop between the worst and the best query times, drilled-down along machine, schema, and query. For `char` we can get factor 3.3 to factor 25 worse by choosing the wrong layout and/or compiler setup. At this point it would be interesting to know, how does the best solution's runtime compare to the others solutions' runtimes?

5.4.1 Runtime Fragility

We show the runtime fragility of the various data layouts in our micro benchmarks in Figure 5.7. The fragility is presented by using box plots, which show the minimum, first quartile, median, third quartile, and the maximum value of the query times of each layout. Notice that the vertical axis displays the performance overhead of that method over the best one (displayed till at most 250% overhead; notice that some box plots leave their plot). As expected according to Table 5.4, Q1 on `char` has the highest fragility, with Q2 on `char` also showing a considerable variance. Core's preference towards layouts with smaller chunk sizes, and the three newer architectures' preference towards larger chunk sizes is again apparent from this plot.

From Figure 5.7 we could already identify that the query time of the set of best layouts for a particular experiment is not always considerably faster than that of the second or even the k-th-best method. A database architect may be willing to live with a data layout that is suboptimal for a very specific case, but does not incur too much performance overhead in the general case. To facilitate this decision, we introduce robustness graphs. Given runtime measurements for N different methods, we depict the overhead of the k-th best method over the best method set. The result for the 24 different experiments is shown in Figure 5.8. They display the impact of non-optimal layouts on runtime performance. The horizontal axis displays the k-th-best data layout picked (where k is in $1, \ldots, N$; normalized to 100%). The bigger the area under the curve, the more likely it is that a decision for a non-optimal data layout will trigger a performance loss.

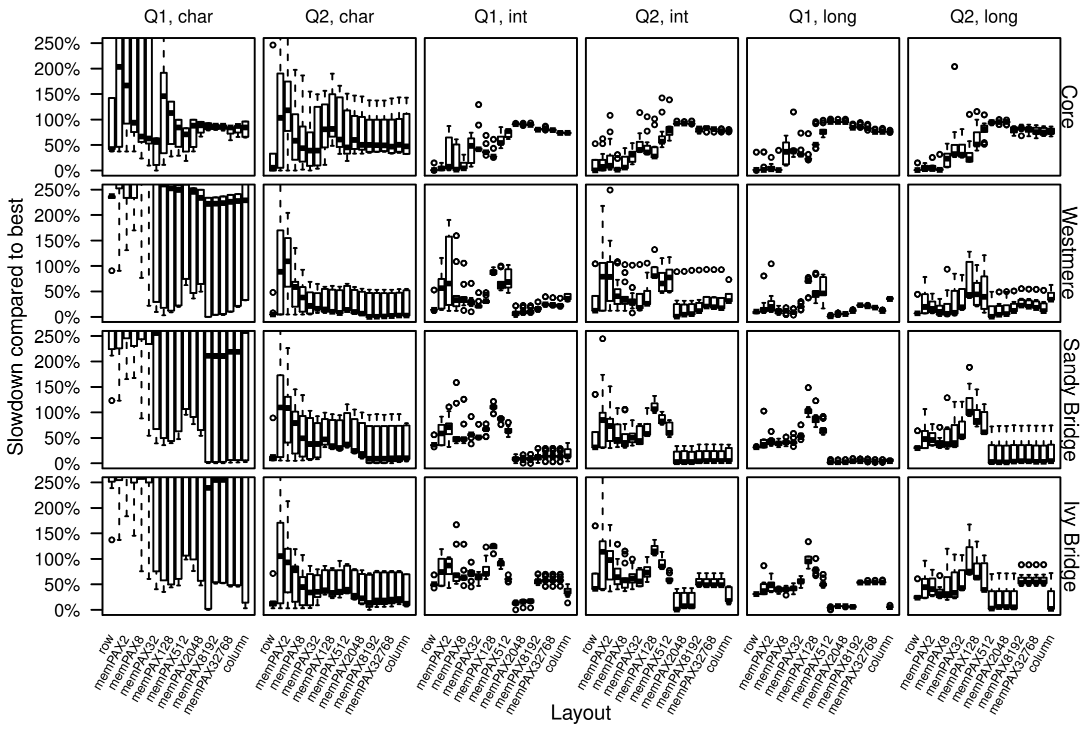


Figure 5.7: Runtime fragility of the various data layouts in our micro benchmarks

The graphs show that for some situations, e.g. Q1 on `char` run on Sandy Bridge, one should be more careful in choosing the layout than in others, e.g. Q2 on `long` run on Westmere.

5.4.2 Conclusions and Guidelines

Our guideline for choosing the best layout is as follows: For servers equipped with a Core CPU it is a safe bet to use row layout, while for machines with the subsequent Westmere, Sandy Bridge, and Ivy Bridge architectures it is just fine to use column layout. For the latter machines we can exploit the schema for some fine-tuning, by creating PAX-blocks with the same size as the virtual memory pages. Having branches in the query is an additional argument for this optimization. The compiler, O-level, and compile time vs. runtime layouts will not change the choice of best layout (see Q2 on `int` run on Ivy Bridge), but they are to be chosen very carefully for the best performance. For the optimal choice of these latter settings, however, one has to try out all possible combinations, since it highly depends on the target system

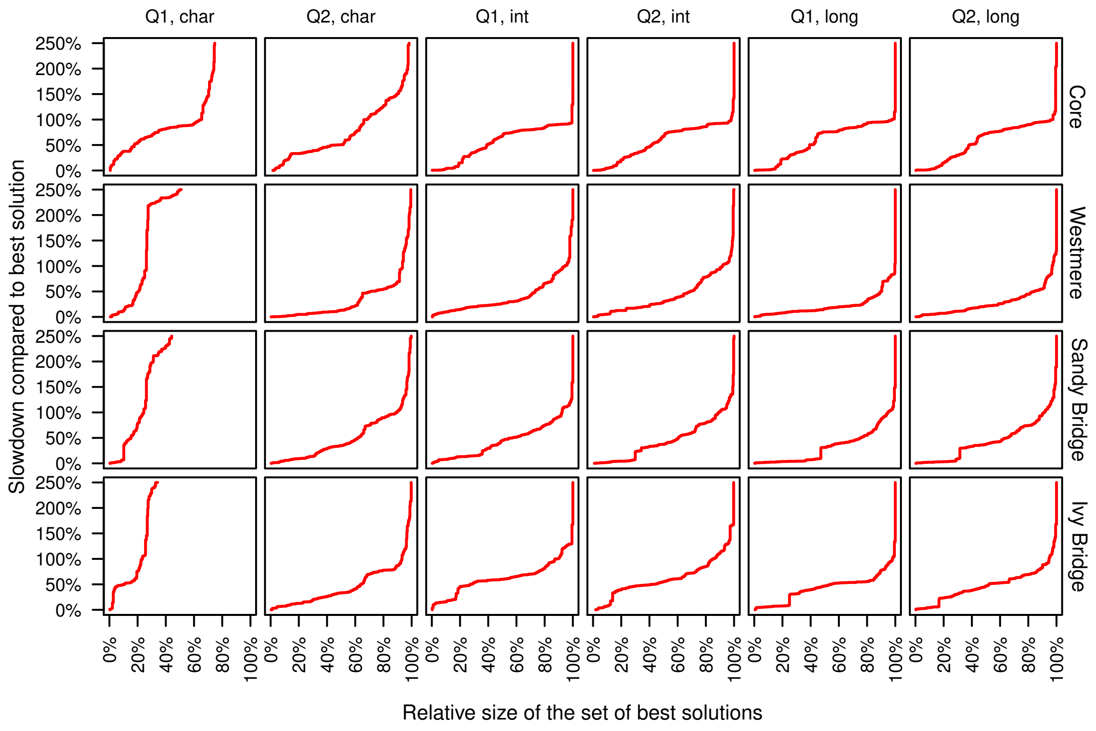


Figure 5.8: Robustness of the best solutions in our micro benchmarks

We have also shown how misleading it can be to choose the best solution along means. Take the case of Q2 on `char` run on Core, where the 6 best solutions are statistically indistinguishable from each other with 95% confidence, yet they differ either in the layout, the compiler, or the optimization level.

5.5 Revisiting Strided Memory Access

5.5.1 Motivation

Various size metrics of the memory subsystem are round numbers in binary, or put more simply: powers of 2 in decimal. System engineers have followed this tradition over time. Some well known examples of objects with powers of 2 sizes: cachelines, caches, RAM modules, HDD blocks, virtual memory pages, and even HDFS blocks. Surrounded by this flood of round binary numbers a data engineer feels pressed to develop data structures with similarly ,,round" sizes. So did we feel, until one day we started to question the optimality of this tradition, and dared to look at memPAX layouts with chunk sizes in between powers of 2.

5.5.2 Background

One of the CPU events debunking the random-access nature of main memory is the memory bank conflict. To understand this event, we first have to explain interleaved memory. DRAM and caches are both organised into banks. In case of DDR3 there are typically 4 banks. Caches on the other hand can have a varying number of banks, depending on the actual CPU generation. Interleaved memory means that the memory addresses are split among the banks in a round-robin fashion, i.e. membankID = address mod 4, which allows for requests to different banks to be fetched — though not transferred — in parallel, thereby improving the bandwidth utilisation. (See [52, Section 5.2] for more details.)

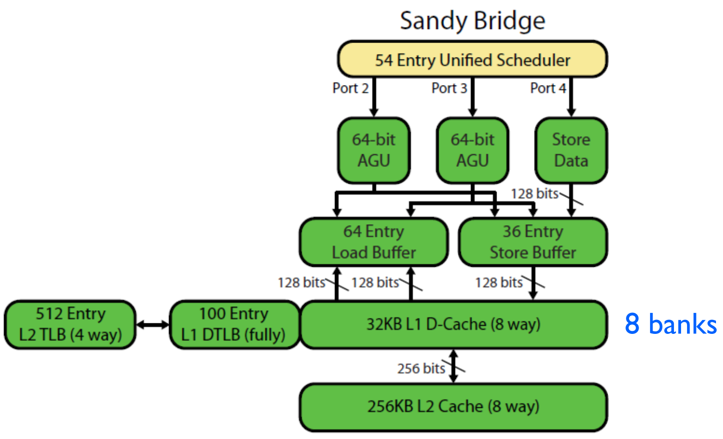


Figure 5.9: The architecture diagram of Intel Sandy Bridge. Image source: `http://www.realworldtech.com/sandy-bridge/7`

In Figure 5.9 we can see the part of the Sandy Bridge architecture diagram that is related to the memory subsystem. There are two important improvements over previous generations [19]. Firstly, the Sandy Bridge architecture has two memory read ports where previous Intel processors had only one. The maximum throughput is now 256 bits read and 128 bits write per clock cycle. The flip side of this coin is that the risk of contentions in the data cache increases when there are more memory operations per clock cycle. It is quite difficult to maintain the maximum read and write throughput without being delayed by cache bank conflicts. The second improvement is, that there is no performance penalty for reading or writing misaligned memory operands, except for the fact that it uses more cache banks so that the risk of cache conflicts is higher when the operand is misaligned.

Getting back to memory bank conflicts, the Intel Architecture Optimization Manual [19, Sections 2.2.5.2 and 3.6.1.3] gives a precise description on this event for the Sandy Bridge architecture: "A bank conflict happens when two simultaneous load operations have the same bit 2–5 of their linear address but they are not from the same set in the cache (bits 6–12)." Thus, in contrast to our expectations, it is actually not beneficial for the performance of load bandwidth-bound code to perform a strided access of addresses with a stride that is a multiple of the cache line size. In that case the addresses will have the same bits 5–0, but different bits 12–6, thus a bank conflict will occur.

5.5.3 Available Resources for Performance Monitoring

Performance monitoring is the action of collecting information related to how a system performs. This information is obtained from the CPU itself by reading its hardware performance counters. There are counters for measuring clock cycles, stall cycles, cache misses, TLB misses, memory bandwidth utilisation, and many more. Monitoring is crucial for detecting and solving hardware-related performance problems.

All modern CPUs have a Performance Monitoring Unit (PMU), which is a piece of hardware collecting micro-architectural events. There are large differences even inside a processor family, e.g. Intel's Westmere and Sandy Bridge have fairly different sets of events that they can measure.

Some of the notable monitoring interfaces include:

- perf: `https://perf.wiki.kernel.org/index.php`
- OProfile: `http://oprofile.sourceforge.net`
- Intel VTune Amplifier: `https://software.intel.com/en-us/intel-vtune-amplifier-xe`
- Likwid PerfCtr: `https://code.google.com/p/likwid/wiki/LikwidPerfCtr`
- Performance Application Programming Interface (PAPI): `http://icl.cs.utk.edu/papi/`
- perfmon2: `http://perfmon2.sourceforge.net`

Out of these `perf` is the most simple one; it merely reports a few basic counter values. `OProfile` is a widely used open-source alternative to the commercially available `VTune Amplifier` from Intel. Both provide access to the full set of performance counters, with `VTune` also providing derived metrics from the raw counter values. Additionally `VTune` provides an intuitive GUI for analysing hotspots, timelines, etc. For advanced instrumentation of user code `Likwid PerfCtr` or `PAPI` also has to be used together with `OProfile`. `perfmon2` provides built-in code instrumentation support, therefore it should be considered instead of `OProfile`. In our experiments we have been using Intel's `VTune Amplifier`.

5.5.4 Performance Implications on Tuple-reconstruction

To demonstrate the effects of bank conflicts on the performance of an application, lets consider Q1 and Q2 executed on Sandy Bridge on `char` fields, compiled with `g++ -O2`, and the chunk sizes being provided at compile time. Let us take a look at the query times for all chunk sizes [measured in tuples] between 2 and 1024, considering multiples-of-2 chunk sizes as well, in Figure 5.10. The black symbols on the left show the query times for row layout, while the ones on the right show the query times for column layout. The red line shows the query times for powers-of-2 chunk sizes, while the blue line shows the runtimes for multiples-of-2 chunk sizes, which is more fine granular. This exemplifies the details that can get overlooked when not performing a fine-granular exploration of the parameter space. Interestingly, there is a periodic spike in the query time, with a period size of 64, which happens to be the cache line size. Recall, that when executing Q1 we have to reconstruct the tuples for computing the aggregate value. As we have two attributes only, the stride of the memory access equals to the chunk size multiplied by the field size. Thus, for `char` fields the stride equals the chunk size. From the above discussion we know that a strided access of memory addresses with a multiple of 64 stride should result in a bank conflict.

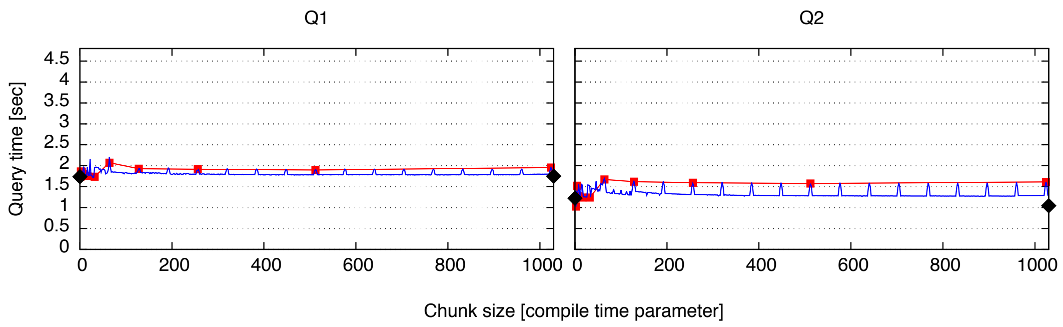


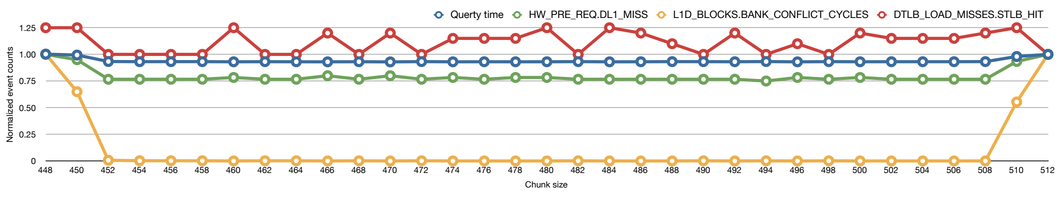
Figure 5.10: Query times of Q1 and Q2 executed on Sandy Bridge on `char` fields, compiled with `g++ -O2`, and the chunk size being provided at compile time.

Therefore, we have decided to validated this claim by letting VTune find the hardware events responsible for the spikes in the query time. We have taken a sample of the experiments, those with a chunk size between 448 and 512. Both endpoints of this interval are multiples of 64, and where the query time has its spikes. We have measured all existing PMU events and looked for those that have a linear correlation with the query time. We have found out that out of the ca. 200 PMU events available for Sandy Bridge, only three correlate significantly with the query time:

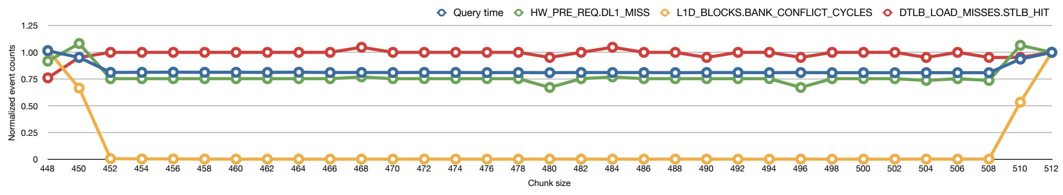
DTLB_LOAD_MISSES.STLB_HIT: data TLB load misses that hit in the second level TLB

HW_PRE_REQ.DL1_MISS: hardware prefetch requests that miss in the L1 data cache

L1D_BLOCKS.BANK_CONFLICT_CYCLES: memory bank conflict in the L1 data cache



(a) Q1



(b) Q2

Figure 5.11: PMU events of Q1 and Q2 executed on Sandy Bridge on `char` fields, compiled with `g++ -O2`, for chunk sizes in $\{448, 450, \ldots, 510, 512\}$.

We have plotted these three PMU events and the query time in Figure 5.11, normalised to the respective values measured for chunk size 512. As we have the same spikes in the query time for the two endpoints of the chunk size interval, the normalised query times equal 1 at these points, and are below 1 for all other points. We can see that the memory bank conflicts in the L1 data cache have a very strong linear correlation to the query time. Basically, both the query time and the latter metric have only 3 different values. The query time is the lowest when there are no L1D bank conflicts at all, and it increases together with the metric just next to the chunk sizes where the spikes are, and reaches its maximum together with the metric. The other two events also show a strong correlation, however, they do not drop to 0 inside the considered chunk size intervals.

As we can see in Figure 5.10, for Q2 choosing a memPAX layout which is not a power of 2 improves the query time by approximately 20%. This is definitely a significant improvement in the spectrum of what can be expected from data layouts. Q2 is a typical example of tuple-reconstruction, and thus memPAX layout can also be used for improving the tuple-reconstruction part of more complex queries.

5.6 TPC-H Experiments

Real world analytical workloads are significantly more complex, than our micro-benchmarks. They have a wider schema with different attribute types, and the queries use more expensive operators as well, including aggregation and joins. In order to investigate the runtime fragility of more complex workloads, let us consider the TPC-H benchmark [59].

5.6.1 Experimental setup

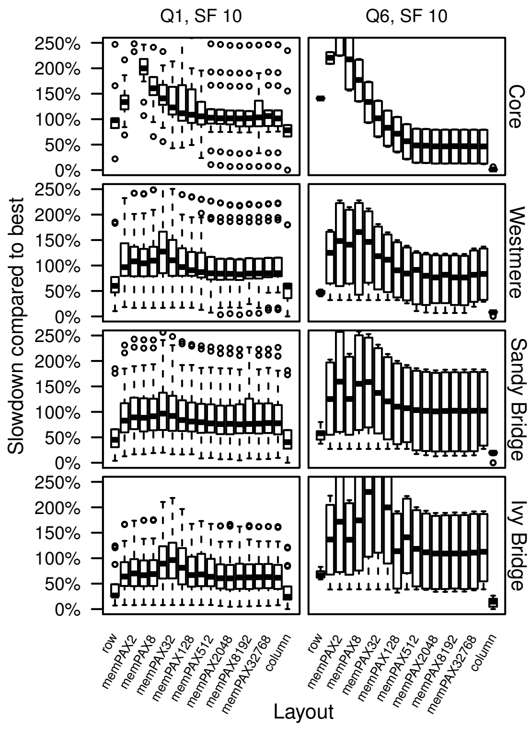
We have implemented Q1 and Q6 of the TPC-H benchmark as hand-coded applications written in C++. These two queries are single-table queries touching only the Lineitem table. We have implemented two variants of the Lineitem table: one matching the schema described in the benchmark, which we will refer to as *uncompressed*. The second version, on the other hand, is a *compressed* table. We have applied some compression schemes to the Lineitem table, as explained in Table 5.5, using the information in Section 4.2.3 ”Test Database Data Generation” of the TPC-H Standard Specification.

Field name	**DDL-compliant data type**	**Compressed type**	**Encoding**	**Reason**
L_LINENUMBER	int32_t	uint8_t	domain	in [1..7]
L_QUANTITY	int64_t	uint8_t	domain	random value [1..50]
L_DISCOUNT	int64_t	uint8_t	domain	random value [0.00 .. 0.10]
L_TAX	int64_t	uint8_t	domain	random value [0.00 .. 0.08]
L_SHIPINSTRUCT	char[25]	uint8_t	dictionary	random string from list Instructions
L_SHIPMODE	char[10]	uint8_t	dictionary	random string from list Modes
L_COMMENT	char[44]	uint32_t	dictionary	random text [10,43]

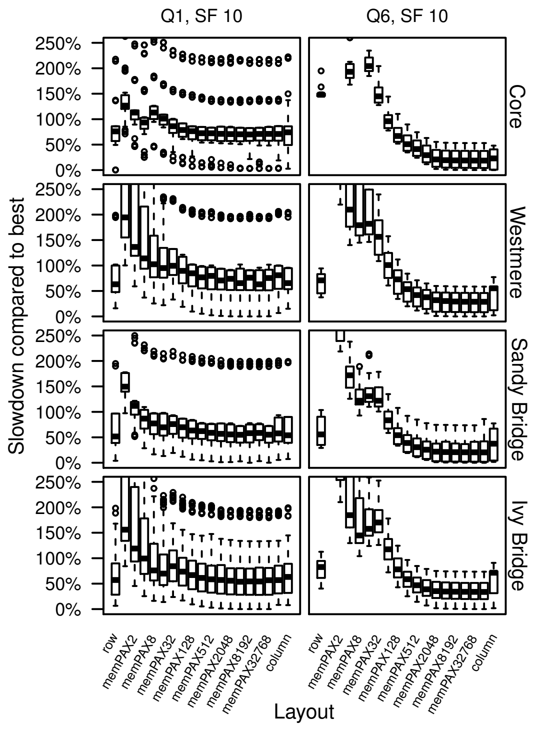
Table 5.5: The compression schemes applied to the TPC-H Lineitem table

5.6.2 Runtime Fragility

We show the runtime fragility of the various data layouts for Q1 and Q6 in the TPC-H benchmarks in Figure 5.12, for both the uncompressed and the compressed Lineitem table. For the uncompressed Lineitem table, column layout is the clear winner in terms of performance. What is more important, is that it also has the lowest fragility — for Q6 it has almost no variance compared to the other layouts. On the other hand, for the compressed Lineitem table column layout is not a clear winner. If we consider the median query times — depicted by the strong dash inside the boxes — for Q6 it is significantly worse than the memPAX layouts with larger chunk sizes. There is one very interesting differences to the results on the uncompressed Lineitem table. First, the layouts of the compressed Lineitem table are much less fragile: for Q6 the boxes are 2–5 times narrower than that of the uncompressed table, which means less fragility.



(a) Over uncompressed tables



(b) Over compressed tables

Figure 5.12: Runtime fragility of the various data layouts for TPC-H queries

5.7 Conclusions

In this chapter we have identified various sources of query time fragility – implementation factors that can change the performance of a query by factors in an unpredictable way. We have investigated the fragility of both micro-benchmarks and complex analytical benchmarks. We have considered the CPU architecture, the compiler, and the compiler flags as important factors. We have introduced the memPAX layout and compared its fragility to column layout and row layout.

We have shown that when querying tables with 1–byte integer columns a very high fragility is to be expected, in our case leading to a performance drop of up to factor 25. In case of more complex schemas and queries the inhomogeneity of the schema has a direct effect on the fragility. Applying dictionary- and domain encoding to the columns have reduced fragility by 50% to 80% in our experiments on the TPC-H benchmark.

We have found a use-case in query processing where using powers of 2 is always a suboptimal choice, leading to one more cause of fragile query times. We have shown how to choose the chunk sizes of the memPAX layouts to improve tuple-reconstruction costs by 20%.

Chapter 6

The Performance Implications of Compiling a Main-Memory Database System

A compiler is just another abstraction layer. It is safe to use whatever default compiler we have on our system. It has a default O-level, which is just fine for most purposes, thus also for building our database system as well. If we would like to generate the most efficient code, we just go for the highest O-level available (or one level higher just to make sure). Maybe we even use those fancy optimizations our compiler supports on top of that. And anyway, if my compiler setup worked fine for me, it will work just fine for you as well. Why are all of the previous statements plain wrong? That is what this chapter is about.

In this work we thoroughly study compiling a whole database system written in the C programming language, taking MonetDB as our example. We investigate the effects of varying the compiler, the optimization level, and miscellaneous compiler flags, and using advanced compilation techniques, like link-time optimization and profile-guided optimization. We also show what happens when using different target hardware, considering six subsequent Intel CPU generations. We show when and how to choose compiler settings to improve query performance for free, just by compiling the system the most suitable way.

6.1 Introduction

High performance database systems are typically written in a compiled programming language, most of the times in C/C++ (MonetDB [10], HyPer [40], Aerospike [1], Redis [2], SAP HANA [20], Microsoft Hekaton [43]), with few notable exceptions that are written in an interpreted or JIT-compiled language

(LegoBase [41] was written in Scala). The previous systems are compiled into machine code specific to a target system (hardware and OS) using a compiler. The compiler is considered as just another abstraction layer in the software development pipeline, and is either used "as is", or with a fixed setting. Any possible interaction between the compiler settings and the target system and use case are neglected.

6.1.1 Motivation

Let us see through the eyes of a representative DBA, say Bob, who wants to try out a new main-memory database system. He is a fan of open-source software, and thus chooses MonetDB [10]. Bob prefers compiling everything from source rather than installing a package, thus he downloads the source code of MonetDB. He then goes through the regular bootstrap, configure, make, and install steps, and successfully installs his new DBMS. It worked like a charm, the build system did his job silently, no questions asked. What Bob did not realize, is that under the hood there was a decision made by the build system on which compiler and compiler flags to use. In his particular case the compiler was the default system compiler, and the flags were chosen by the build system depending on the compiler. Now let us not be hard on Bob. Who of us has ever changed the default compiler and flags upon installing software?

One day Bob decides to try out Gentoo Linux. That distribution encourages users to build a Linux kernel tailored to their particular hardware and to customize which services are installed and running. They even provide an Optimization Guide for GCC[1] where they describe many important knobs of the GCC compiler, called compiler flags. Bob applies the hints from that guide and manages to reduce the boot time of his shiny new Linux installation by a factor of 4. He is delighted from the results and starts to think about which software to recompile next. Then he remembers that open-source database system he installed the other day, called MonetDB. Bob already sees himself getting promoted to chief DBA for his new findings: forget column layout, dictionary compression, and indexes. Simply compile with `gcc -O42 -mavx2048 -Ofast`.

Just before getting his hands dirty with compiling, Bob spots the huge `configure.ac` file among MonetDB's sources. He starts to wonder what could be so complicated that takes 3,478 lines to ./configure. So he skims through the script and finds out that there are other compilers out there as well, icc and clang, for which the build script chooses different compiler options. Even worse, the script further differentiates between the numerous versions of these compilers. Bob gets

[1]`wiki.gentoo.org/wiki/GCC_optimization`

really curious about these *other* compilers and installs them on his system. Bobs saga continues in the following sections of this work.

6.1.2 Research Questions

Many high-performance database systems are pre-compiled software. Compiling the system yourself is seen as yet another step in the software development pipeline, and is typically not considered as a performance factor. However, if a developer would like to tune the compiler setup of a database system, he might end up finding the compiler having more tuning knobs than the database system itself. Thus, there are many open questions about compiling a whole database system:

- Does changing the compiler settings have only a negligible effect on the query performance of a whole database system?
- Can we improve query performance by choosing a suitable compiler setup system-wide?
- Can we further improve query performance by choosing for each query the most suitable compiler settings?
- Do our findings still hold if we use another machine?
- Ultimately, can we consider the compiler as just another abstraction layer?

6.1.3 Contributions

In this work we present an exhaustive experimental study on compiling the MonetDB database system. Our main contributions are as follows:

1. Given the large number of tuning knobs of compilers we first discuss the ones that could be the most important starting points of performance tuning when compiling a whole database system, namely: i) the compiler itself, ii) the optimization level, and iii) advanced compilation modes.
2. For the above categories we consider: i) the three most popular C/C++ compilers (GCC, clang, and the Intel C/C++ compiler), ii) all five standard O-levels, and iii) link-time optimization (LTO), and profile-guided optimization (PGO).
3. We consider all combinations of the above knobs, which we will call compiler setup, and build 90 separate MonetDB instances using each of them. We then measure the query performance of the resulting system instances on the TPC-H benchmark.

4. We show the differences in the efficiency of the compiler setups when using a different machine. Here we consider six servers equipped with CPUs of subsequent generations.

5. We present two techniques for improving query performance that build upon changing compiler setups. These approaches work on two different levels: on a per-query level, and on the physical database operator level.

6.2 The Six-dimensional Parameter Space of our Experiments

Architecture	**CPU (Xeon)**	**NUMA**	**#CPUs**	**#cores /CPU**	**Clock**	**LLC**	**RAM [Size @ Speed]**
Core	5150	no	2	2	2.66 GHz	4 MB L2	8 x 2 GB @ 266
Penryn	E5430	no	1	4	2.66 GHz	12 MB L2	4 * 4 GB @ 667
Nehalem	E5506	yes	2	4	2.13 GHz	4 MB L3	12 * 4 GB @ 800
Westmere	X5690	yes	2	6	3.46 GHz	12 MB L3	12 x 16GB @ 1066
Sandy Bridge	E5-2407	yes	2	4	2.2 GHz	10 MB L3	6 x 8GB @ 1066
Ivy Bridge	E7-4870 v2	yes	4	15	2.3 GHz	30 MB L3	32 x 16GB @ 1333

Table 6.1: The machines used in our experiments

(1) The CPU architecture. The performance characteristics of a main-memory database system are significantly influenced by the machine's CPU. As there are usually significant changes between the subsequent CPU architectures, we have chosen servers equipped with Intel CPUs of six subsequent architectures. This allows us to investigate whether different CPU architectures favour different compiler settings.

(2) The compiler. In our experiments we have chosen the three most commonly used C/C++ compilers: clang (3.5.0-10), gcc (4.9.2), and Intel's C compiler – icc (16.0.0). clang and gcc are both open-source, while icc is proprietary software. clang is actually a C-compiler front-end to the LLVM compiler infrastructure. It compiles C, Objective-C, and C++ code to the LLVM Intermediate Representation (IR), similar to other LLVM front-ends, which allows for a massive set of optimizations to be performed on the IR before translating it to machine code. GCC is short for GNU Compiler Collection, a compiler supporting among others the C/C++ language. It support almost all hardware platforms and operating systems, and it is the most popular C/C++ compiler, and also the default one in most Linux distros. Intel's C/C++ compiler can take advantage of Intel's insider knowledge on Intel CPUs. It is said to generate very efficient code especially for arithmetic operations.

(3) The optimization level. Compilers can perform various transformations during the pre-processing of the code, on the intermediate representation, and when generating machine code as well. Most of these transformations aim to improve the performance of the resulting object file or executable, and thus are called optimizations. Each of these optimizations can be toggled by separate flags, though there are some special flags that enable a given set of optimizations, which is termed the optimization level. The optimizations offered by the different compilers and the definition of the O-levels differ from compiler to compiler. Yet, the general goals and trade-offs of the O-levels are very similar across compilers. These are described for the five standard O-levels as follows:

O-level	**Description**
-O0	No optimization
-O1	Moderate optimizations, fast compiling
-Os	Optimize for size
-O2	Optimize for speed
-O3	Expensive optimizations, slow compiling

Table 6.2: Compiler optimization levels

In case of clang and gcc the set of optimizations enabled by higher O-levels includes all optimizations enabled by lower O-levels (excluding `-Os`). For icc this only holds for `-O2` and `-O3`. We have to note that compilers differ in optimizing for size, i.e. how they handle the `-Os` flag. For clang `-O2` and `-Os` are the same (i.e they activate the same optimization passes), for icc `-O1` and `-Os` are the same, and for gcc `-Os` activates all flags that `-O2` does, except for those that tend to increase the executables' size. This is summarized in the following table:

Compiler	**O-level inclusion**
clang	$\mathrm{O1} \subset \mathrm{Os} = \mathrm{O2} \subset \mathrm{O3}$
gcc	$\mathrm{O1} \subset \mathrm{Os} \subset \mathrm{O2} \subset \mathrm{O3}$
icc	$\mathrm{O1} = \mathrm{Os} \not\subseteq \mathrm{O2} \subset \mathrm{O3}$

Table 6.3: Optimization level inclusion for different compilers

Architecture	**clang**	**gcc**
Default	x86_64	x86_64
Core	core2	core2
Penryn	penryn	core2
Nehalem	corei7	nehalem
Westmere	corei7	westmere
Sandy Bridge	corei7-avx	sandybridge
Ivy Bridge	core-avx-i	ivybridge

Table 6.4: The CPU architecture detected by clang and gcc when generating native (architecture specific) code.

We intuitively expect to get higher performance from higher optimization levels, yet there is no guarantee from the compiler's side that this will also hold in practice. Thus, we have decided to evaluate all five standard optimization levels in our experiments.

(4) Miscellaneous compiler flags. Each compiler has a mechanism for detecting the CPU architecture of the machine they are being executed on. They can use this information to generate the most efficient machine code for the target CPU, which is activated by the `-march=native` flag. This includes choosing vectorized instructions that use the largest SIMD registers available on the target CPU. The default behaviour is, however, to generate code that runs on all x86_64-compatible architectures. In the following we list the architecture tags detected by clang and gcc. Note that icc does not allow the programmer to detect which target architecture it has chosen, however, we can assume it makes a reasonable choice.

We can see that the compilers do not distinguish some of the subsequent CPU architectures. Nevertheless, we did not override the architecture choices made by them.

(5) Compilation mode. We can distinguish between the following two compilation modes: conventional compilation and link-time compilation. In conventional compilation each (preprocessed) source file gets compiled separately, one at a time. At any given time the compiler has knowledge of a single compilation unit only, and thus is unable to perform optimizations that would consider the whole program. On the other hand, link-time compilation involves converting all compilation units into an intermediate representation (IR), and eventually merging all IRs together and compile and optimize the whole program. This allows for optimizations that over-cross module boundaries. GCC in particular supports this feature as link-time optimization [23].

Compilers make the same effort when optimizing each part of the code. It generally holds that 90% of the runtime is spent in 10% of the code, therefore this equal-efforts strategy is suboptimal. However, it is usually not known during compilation which part of the code will belong to the so-called hot-spots. This can only be found out after the code has been compiled, and the program gets executed. Profiling means to gather runtime information and use it to optimize the hot-spots of the code. One group of profilers gives the programmer information on where to manually optimize the code, e.g. perf, oprofile, or Intel's VTune Amplifier. Another group of profilers gather profiling information that needs to be fead back to the compiler, which then uses it to make better optimization choices, and concentrates the optimization efforts on the hot-spots. This latter process is called profile-guided optimization [14]. It requires two compilation passes with a profiling step in between to guide the second pass in choosing branches, etc. This technique allows us to optimize our program for a target workload, and not just for the average case, which is determined by the compiler makers. Due to this tedious double-compiling process it has not yet gained widespread usage, despite its potential.

(6) DBMS settings. Modern database systems can gain high performance by using inter- and intra-query parallelism. We only consider inter-query parallelism, i.e. using multiple threads to execute a given query. The level of parallelism, however, has to be chosen wisely, as our experiments have shown. We will consider single-threaded query execution as well, since it allows us to factor out effects caused by parallel execution, moving around data between threads, etc. We do not focus on inter-query parallelism in this work, thus we do not conduct experiments where we would execute *different* queries in parallel.

6.3 Methodology

6.3.1 Building MonetDB

We have built MonetDB 11.21.5 from source on the six different machines described in Table 6.1. For benchmarking and production environments MonetDB's install guide advises to pass the following options to the configure script: `--enable-debug=no --enable-assert=no --enable-optimize=yes`. All three compilers used in this work are supported by MonetDB's build scripts, and the `--enable-optimize=yes` option yields the `-O3` optimization level for all of them. We have modified the build script so that we can specify the optimization level and any other compiler flags we would like to use, instead of using the default O-level.

We have built a separate MonetDB instance for all possible combinations of compiler settings investigated in this work: 3 compilers $\times$ 5 O-levels $\times$ native/non-native code, yielding 30 different instances using traditional compilation. The documentation of advanced compilation modes, LTO and PGO, is quite shallow for clang. Moreover, using these techniques is not applicable to MonetDB's build process without major modifications, thus we have decided to not build MonetDB using clang and the aforementioned advanced compilation modes. For gcc and icc, however, we have compiled instances using i) only LTO, ii) only PGO, and iii) both LTO and PGO, yielding an additional 60 instances: 3 advanced compilation modes $\times$ 2 compilers $\times$ 5 O-levels $\times$ native/non-native code, thus ending up with altogether 90 different MonetDB instances.

We assume the following scenario to be the *default compiler setting* for compiling MonetDB: `gcc`[2]`, -O3`, and non-native code. It is to be noted that the compiler switch needed to generate machine code exploiting larger registers, vectorised instructions, etc. (`-march=native`) is not enabled by default in MonetDB's build scripts when building for a production environment. We are always going to use the MonetDB instance compiled with the default compiler settings as a baseline in our comparison.

The build process of MonetDB is comprised of running `configure`, `make`, and `make install`. Depending of the compiler settings and the machine, the `configure` step runs for 20-40 seconds, the `make install` step for 10 seconds, and the `make` step, comprising the actual compiling of the system, varies heavily depending on the compiler settings.

The compile times against the O-level are shown in Figure 6.1 for the six machines and for the thirty possible compiler settings. We are going to use the visualization of this figure throughout this chapter, which is to be parsed as follows: each measurement has its complete compiler setup encoded using the shape (for the compiler), fill (for the native code), and color (for the O-level). Furthermore, to make the figure more readable, we have added some jitter along the categorical axis, in this case the x-axis showing the O-level. We can see that, as expected, increasing the O-level yields higher compile times. Furthermore, icc has a steeper increase in compile time than the others, making `icc -O3` especially expensive compared to the other two compilers' compile times. If we would rather go for the fastest compile time despite performing expensive optimizations as well, clang

[2]GCC is currently the most popular C/C++ compiler, see http://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/

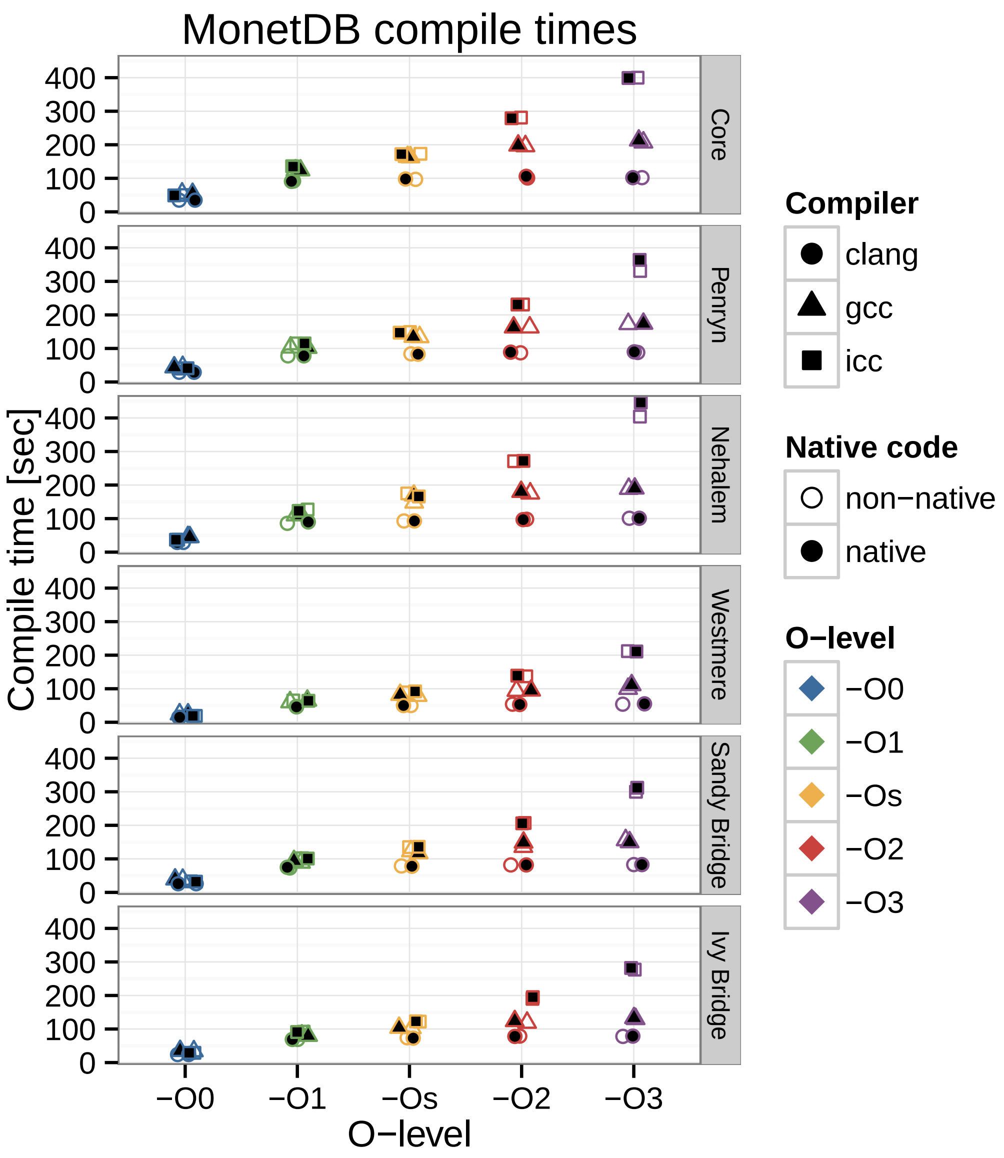


Figure 6.1: MonetDB compile times using the standard compilation mode.

would be our choice. However, as we compile the system only once per setting, we are more interested in the query performance of the resulting MonetDB instances, rather than their compile times.

6.3.2 Experimental setup.

To evaluate the performance of the various MonetDB instances we are going to measure the query times of the 22 different TPC-H queries. As described in the literature on performance analysis [34] and in Chapter 4, reporting merely the average of the measured query times can present an arbitrarily imprecise information about the query times, caused by high variance or outliers. Therefore, we report the median query time of 5 runs, since it is not as sensitive to outliers as the average value.

To achieve high query performance MonetDB tries to utilise all available hardware resources by caching any tables read in main memory, and using all available CPU-threads for intra- and inter-query parallelism. Since we are interested in main-memory database performance, we have performed a warm-up run before each experiment by executing all 22 TPC-H queries to let MonetDB cache all tables in main-memory. We have set the TPC-H scale factor to 10, yielding approx. 10 GB of data, which fits into each of our server's RAM.

Our machines were running Debian 8.2 with Linux kernel version 3.16.0-4-amd64, with hyper-threading either disabled or not supported. For single-threaded experiments we have pinned the MonetDB server process to a given CPU core to avoid runtime variance cased by data- and thread-shuffling.

6.4 90 shades of compiling MonetDB

In this section we are going to benchmark MonetDB, an open-source database system developed at CWI. MonetDB is a column-store pioneer, and as such it is most suited for OLAP-style workloads. We are going to evaluate the performance implications of the MonetDB instances compiled with various settings on the TPC-H benchmark [60]. Let us now follow Bob on his journey through the 90 shades of compiling MonetDB.

6.4.1 What happens if we use another compiler?

While Bob was skimming through MonetDB's `configure.ac`, he has discovered that there are other C-compilers as well, apart from gcc. Thus, he gave it a try to compile MonetDB using clang and icc as well. He then executed the TPC-H benchmark in the MonetDB instances compiled with the three different compilers, and measured the query times. This can be seen in Figure 6.2. When looking at the query times, Bob was really surprised to see at least 10% difference in performance for 11 out of 22 queries, both improvements and performance drops. For Q20 we can improve over gcc by 15% when using icc instead. On the flip side, for Q11 icc has 30% worse query time than gcc. Thus, it follows that using another compiler changes query times significantly, sometimes improving it, while at others making it worse.

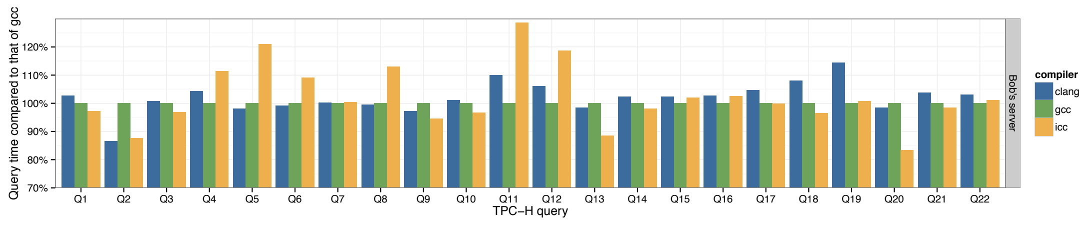


Figure 6.2: TPC-H query times using different compilers, compared to that of gcc (lower is better).

6.4.2 What happens if we change the O-level as well?

Bob was saying to himself, if changing the compiler alone had an effect on the query times, what would happen when changing the O-level as well, which unlike the compiler is actually expected to have an effect on performance. Thus he has decided to compile five-times more MonetDB instances by using all five standard O-levels with each compiler. This can be seen in Figure 6.3, where the colors denote the compiler, and the saturation denotes the O-level.

Bob found that surprisingly `-O2` often yields better runtimes than `-O3`, like for Q2 and Q14 using gcc, Q5 and Q11 using icc. What's more: `clang -O1` has proved to be the most efficient system for queries Q4, Q6, Q14, and Q20. Put another way: compiler optimization hurts the performance of those queries.

The question is, whether there is a clear winner among the compilation settings. The workload times are shown in Table 6.5 as a factor of the default workload time. Here we see up to 44% overall performance drop when using `clang -O0` or `icc -O0`. What comes as a surprise is, though, that `gcc -O0` instead of scoring similiar to the other compilers with `-O0`, it actually slightly improves the workload time over `gcc -O3`. When looking at how much can we improve over the default setting, we find that more often than not the workload time is slightly above the default, and the best we can get is 3% improvement when using `icc -O2` or `icc -O3`.

6.4.3 What else can happen inside traditional compilation?

Till now we have explored two out the six dimension of our parameter space: compilers and O-levels. In the following we are going to explore three more dimensions, and leave the last dimension, using advanced compilation modes, to the very end.

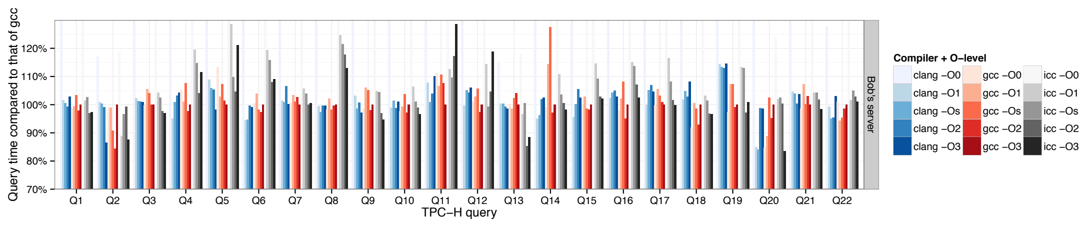


Figure 6.3: TPC-H query times using different compilers and O-levels, compared to that of gcc -O3 (lower is better).

Compiler	-O0	-O1	-Os	-O2	-O3
clang	1.44	1.01	1.00	1.00	1.01
gcc	0.99	1.00	1.02	0.99	1.00
icc	1.44	1.04	1.04	0.96	0.97

Table 6.5: TPC-H workload times as a factor of the default workload time, that of `gcc -O3` (lower is better).

Forcing single-threaded execution

MonetDB by default uses all available hardware resources for executing a query, including all CPU threads and caching everything possible in memory. Let us now restrict MonetDB to using a single CPU thread in order to factor out the potential performance penalties of multi-threaded execution, including moving data between NUMA regions, remote reads, and scheduling software threads. The resulting query times can be seen in Figure 6.4. Compared to the multithreaded query times we can see less query time fragility, and also smaller maximal improvements over the default setup, namely at most 10%. On top of that, the resulting query times are more often worse than the default one, than what we have experienced by the multithreaded case. This suggests that a suitable compiler setup can improve the component in MonetDB responsible for intra-query parallelism.

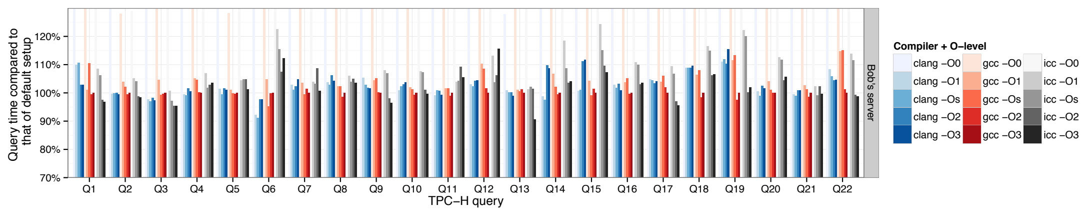


Figure 6.4: TPC-H query times using different compilers and O-levels, compared to that of gcc -O3 (lower is better). *Single-threaded* query execution.

Using native code

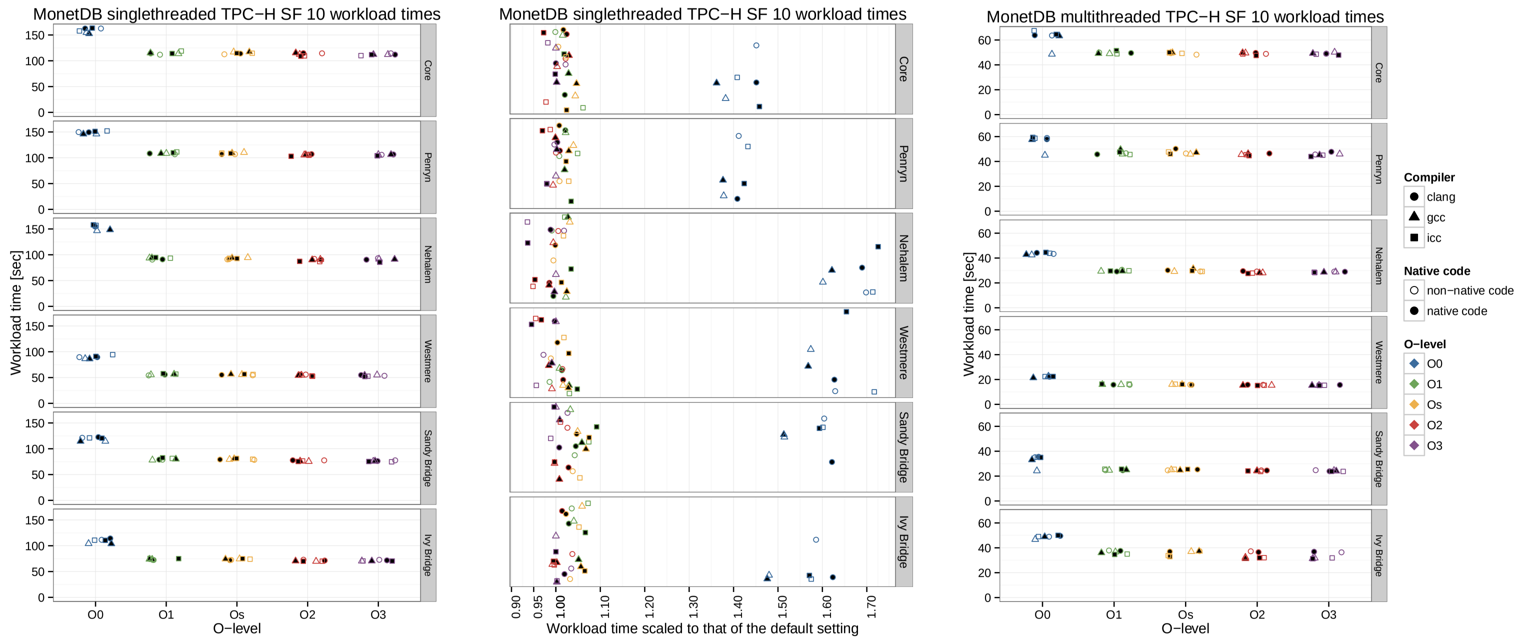
Bob remembered reading something about generating native-code in Gentoo's Optimization Guide for GCC. Thus he sets out and recompiles all 30 MonetDB instances with using this additional flag that instructs the compiler to generate machine code that exploits all specialized hardware operations and registers available on that particular machine. Then he compares the resulting query times with- and without generating native code. We can see no clear benefit for single-threaded execution, with improvements and performance drops of up to 10% in some few cases. In case of multithreaded execution, however, we can see larger changes in the query times. For Q11, Q14, and Q20 the improvement over non-native code is about 25%.

6.4.4 What changes if we use another machine?

Bob has remembered something about various system descriptors appearing in MonetDB's build scripts, upon which it differentiates what compiler flags it should use. Bob has found out that these system tags actually refer to the CPU-generation, or CPU architecture.

Figure 6.5a shows the workload times on the single-threaded MonetDB instances, and all six machines described in Section 6.2. The machines we used vary in their CPU clock speed and maximal memory bandwidth, and thus have different workload times for the same compiler setup. Our server with the Westmere CPU has much higher CPU clock speed than the others, which explains why it performs best. Except for this outlier machine, the newer machines (CPU generation increasing from top to bottom) perform increasingly better on the TPC-H benchmark. To better see the effects of the O-level, we have put it on the horizontal axis. As expected, the instances compiled without optimization (`-O0`) have a significantly worse performance: their workload times are on average 50% higher than that of the default setup.

The difference between the rest of the O-levels, that unlike `-O0` do perform some optimizations, are hard to see in this graphic. This is why we have decided to scale each workload time to that of the default setting for each machine, which allows us to spot compiler setups that yield a better performance than the default one. This is shown in Figure 6.5b. Please note, that this is essentially a one-dimensional chart with some added jitter on the (non-existing) vertical axis to enhance readability. The performance gain achieved over the default setup depends on the machine, and gives us three cases: on Core and Penryn we can gain 2-3%, on Nehalem and Westmere 5-7%, on Sandy Bridge and Ivy Bridge less than 2%. It is notable, that the best workload times are achieved by the MonetDB instances compiled with `icc -O2`/`-O3`. On the flip side we can more often than



(a) single-threaded (b) single-threaded scaled (c) multithreaded

Figure 6.5: The workload time of the TPC-H benchmark using the MonetDB instances compiled with different compiler setups

not actually get a worse query performance by choosing a non-default compiler setup, yielding up to 5-10% worse workload time (excluding workload times of instances compiled with `-O0`).

The MonetDB database systems aims for exploiting all available hardware resources by caching all data in main-memory and using all hardware threads. In the previous experiment we have restricted ourselves to single-threaded execution to factor out the potential performance penalties of multi-threaded execution, including moving data between NUMA regions, remote reads, and scheduling software threads. In the following we will discuss the effects of the compiler setup for all six machines in case of multi-threaded execution. The workload times are shown in Figure 6.5c. It is somewhat surprising, that we again find `gcc -O0` to yield comparable workload times, as we have already seen in Table 6.5. The same situation occurs now for the Core, Penryn, and Westmere machines as well.

With some profiling we could confirm that the actual number of threads MonetDB used during query processing equaled to the number of hardware threads on each machine. Our fastest machine for single-threaded execution, Westmere, had an advantage in this aspect as well: it has 12 hardware threads and the highest CPU clock, which together secured again the fastest average query times in our comparison. It comes as a surprise that the Ivy Bridge machine with its 60 hardware threads was even outperformed by the Nehalem machine that has only 8 threads. This is in heavy contrast to what we have seen in the single-threaded case, so we have investigated the reason of this machine performing poorly in the multi-threaded mode. We have found that TPC-H Q2, Q17, and Q21 have an extremely high query time when compared to that of Sandy Bridge: a factor of 19, 15, and 2.2 times higher, respectively, for the default setup (see Figure 6.6). These are queries that are hard to parallelize according to an analysis on the TPC-H benchmark [11], which perfectly fits our case, since the Ivy Bridge machine has way more threads than the Sandy Bridge machine. We think that when parallelizing the query plan for 60 threads and 10 GBs of data (i.e scale factor 10) MonetDB makes a suboptimal query plan.

In Table 6.6 we show the call counts of the most-expensive operators for some representative TPC-H queries for Sandy Bridge and Ivy Bridge, obtained using MonetDB's tomograph tool [22]. Queries Q1, Q6, and Q13 have a call count ratio that is near to the ratio of the two machines hardware thread counts (60 / 8 = 7.5), and have a lower query time on Ivy Bridge than on Sandy Bridge. Those queries that have a much higher runtime on Ivy Bridge, also have a call count ratio way above the hardware threads ratio, which proves our assumption on MonetDB choosing a suboptimal plan for these queries.

Similarly to the single-threaded case, the MonetDB instances compiled with some optimization have very comparable workload times. There are, as mentioned

Query	Operator	Sandy Bridge	Ivy Bridge	Ratio
Q1	batcalc.*	27	183	6.78
Q2	algebra.subjoin	106	3865	36.46
Q6	algebra.subselect	34	242	7.12
Q13	algebra.subjoin	8	57	7.13
Q17	algebra.subjoin	72	3660	50.83
Q21	algebra.subjoin	152	7380	48.55

Table 6.6: Call counts of the most-expensive operator in the multi-threaded query execution traces of some TPC-H queries.

before, some interesting outliers: on Core, Penryn and Sandy Bridge the instances compiled with `gcc -O0` even outperform the default one. We can conclude that except for Nehalem and Westmere there is no compiler setup that would yield a significant improvement in the cumulated query times for the whole TPC-H workload. However, this does not necessarily mean that the compiler setups have no effect on the individual TPC-H queries' runtimes, as we have already seen in the previous sections.

6.4.5 What if we use advanced compilation modes?

Bob has seen a lot of query time variation till now, and is really curious about what is still to come. Thus, he decides to let the compilers throw all what they got at MonetDB, and use their advanced compilation modes. Since they are called advanced, Bob expects to get larger improvements by using them, than what changing O-levels and the such has brought. There are two major techniques in this field that should be mentioned: link-time optimization (LTO) and profile-guided optimization (PGO), both of which have been described in Section 6.2. Thus, Bob decides to use both of them, and performs first the profiling and eventually builds the MonetDB instances using the gathered profiling information in the hope of improving MonetDB exactly there, where it would help the most. Note that when using advanced compilation modes, we cannot consider clang for reasons detailed in Section 6.2.

We compared the query times for each query and each MonetDB instance compiled using the *same* compilation settings, but differing only in the compilation mode – one using LTO and PGO, while the other one using traditional compilation without these two techniques. When compiling with icc the improvements are between 5% to 30%, and is on average 20%. gcc using these two advanced compilation modes, on the other hand, reaches at most 10% improvement, averaging at around 5%. Yet, using `icc -Os/-O2/-O3` with LTO and PGO reduces

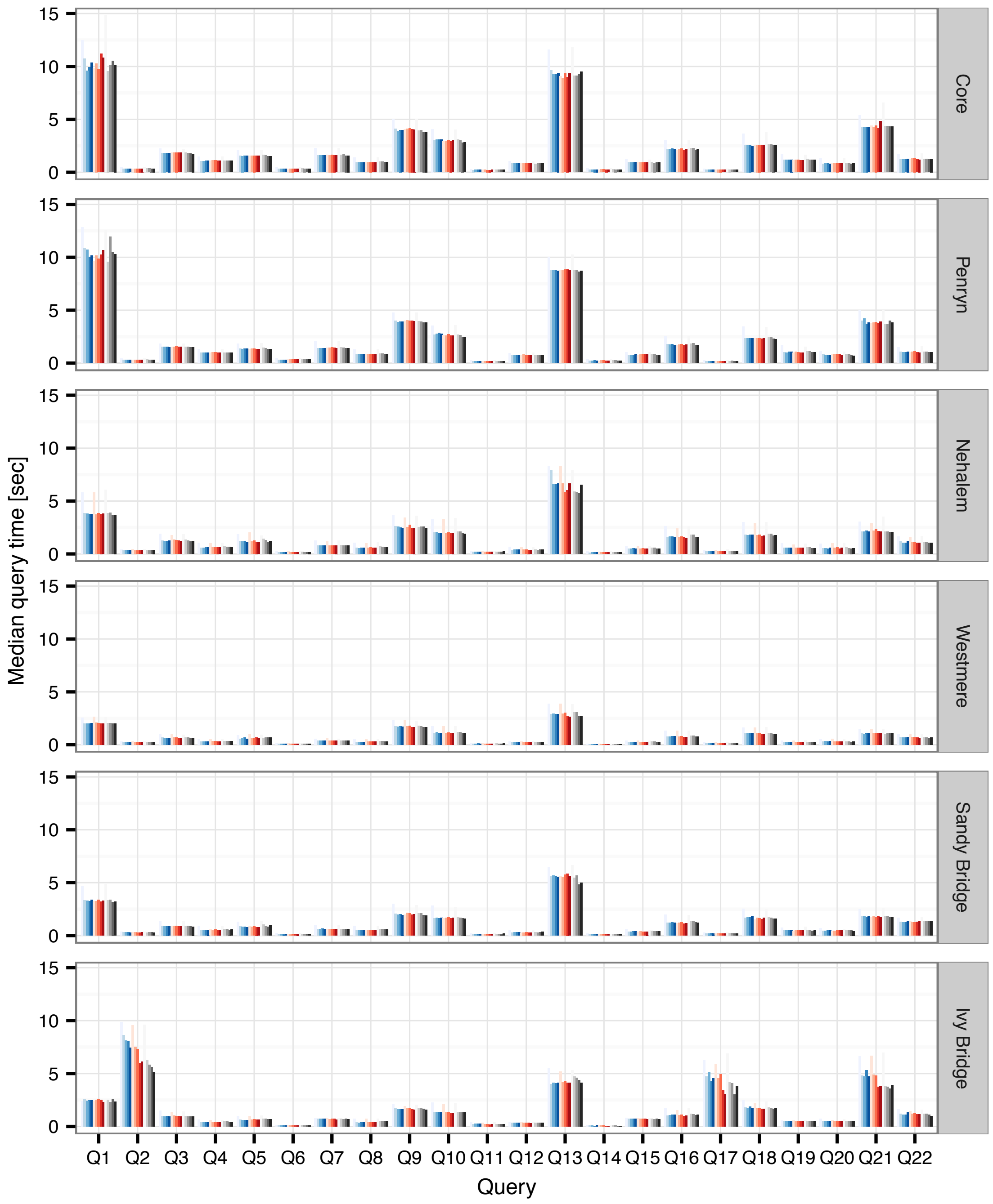


Figure 6.6: TPC-H multithreaded query times on different machines using different compilers and O-levels (lower is better).

the workload time by a mere 10% compared to the workload time of the default MonetDB instance. Thus, we can conclude, that there is no single best choice for the compiler settings, and Bob has to give up his dream of becoming the new chief DBA that easily. Luckily for Bob, he has an idea on what he could do with all this query time fragility.

6.5 Query routing

So far we were most interested in the cumulated query times of the TPC-H benchmark. Let us now pick a single machine, and look at the individual query times. Figure 6.7 shows the query times measured on our Westmere machine scaled to the default MonetDB instances's query times for each query. Here we consider only compiler setups with non-native code, and single-threaded execution. In this figure we depict the compilers with three different colors, while the O-levels give the saturation of the colours. Note that we do not show measurements for `-O0`, because they all perform at least 50% worse than the default instance. We can see that there is a lot of variance in the performance of the different MonetDB instances on a per-query level, and it is hard to detect any pattern in which compiler setups and when do they yield good performance. E.g. for Q6 `clang -Os` performs best, but for Q13 the same compiler setup is second to worst. Similarly, `icc -O1` is the best for Q13, but the worst in case of Q15.

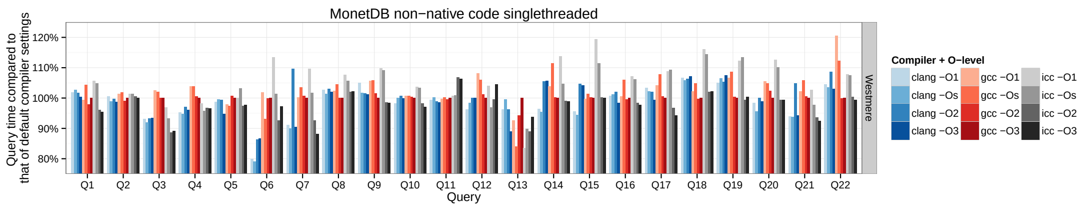


Figure 6.7: Query times of the TPC-H benchmark on Westmere scaled to that of the default MonetDB instance.

This leads us to the idea of routing each query to the MonetDB instance that is expected to yield the best query time for that particular query. The routing mechanism can be based on some initial tuning on a representative workload, or we can do continuous tuning on the actual production workload as well. When the system has been tuned, and a new and yet unseen query comes in, we can decide to first tune it, or to use some query similarity metric to identify the most similar query we have already tuned, and route the new query accordingly.

In the tuning process we can start with routing a given incoming query each time to a different MonetDB instance until it has been executed on all instances.

After that we can route the query there where it has run fastest. This process takes in our case 90 iterations to complete.

As we have benchmarked all MonetDB instances, we actually have the same information as a fully-tuned system would have. Therefore, we can calculate the improvements we could achieve by query routing. Figure 6.8 shows the query time reduction over the default setup for an interesting subset of the benchmark, for both single-threaded and multithreaded query execution. We can see that just by executing a query on a properly compiled MonetDB instance the query time can be reduced by up to 25% compared to the query time on the default instance. The improvement that can be achieved depends on the machine, the thread count, and the query as well. Therefore, for each particular machine and workload we need to perform the initial tuning of the system to explore the potential in query routing to eventually use it to run each query on the most suitable MonetDB instance.

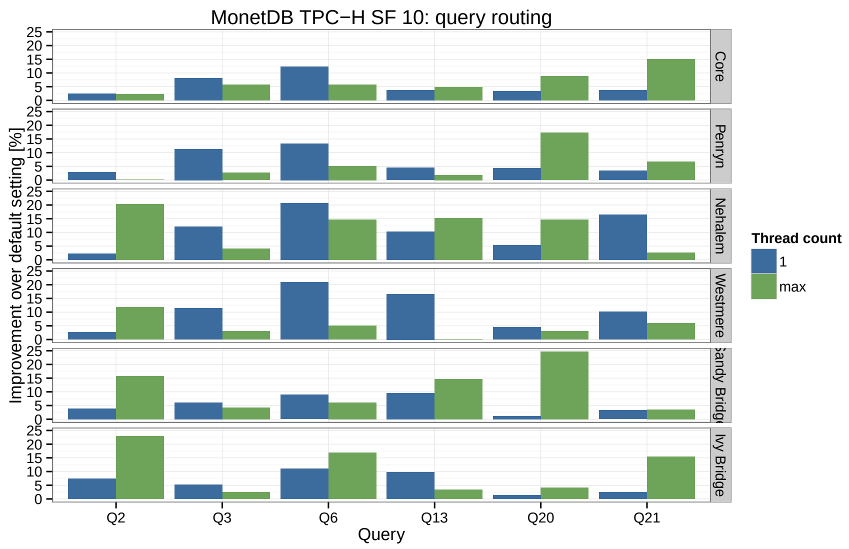


Figure 6.8: Query time reduction over the default setup when using query routing.

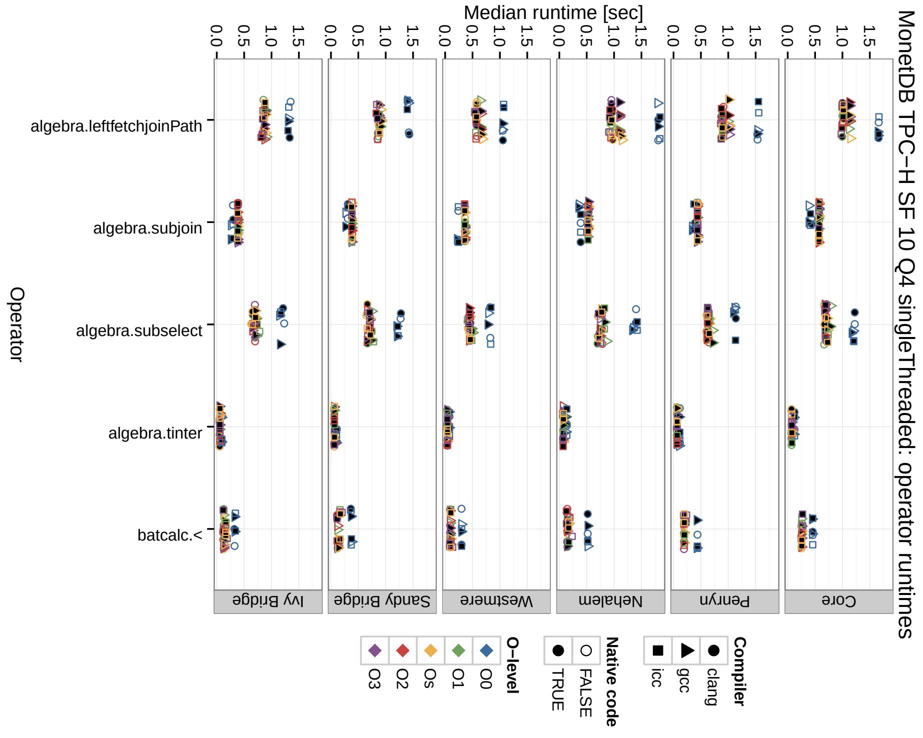
6.6 Operator routing

When we fire a query at a database system, under the hood the query goes through the following conceptual steps: it gets parsed, converted into a logical query plan, and eventually converted into a physical query plan by the optimizer. In this last step, among others, the set of physical operators and the execution model are chosen.

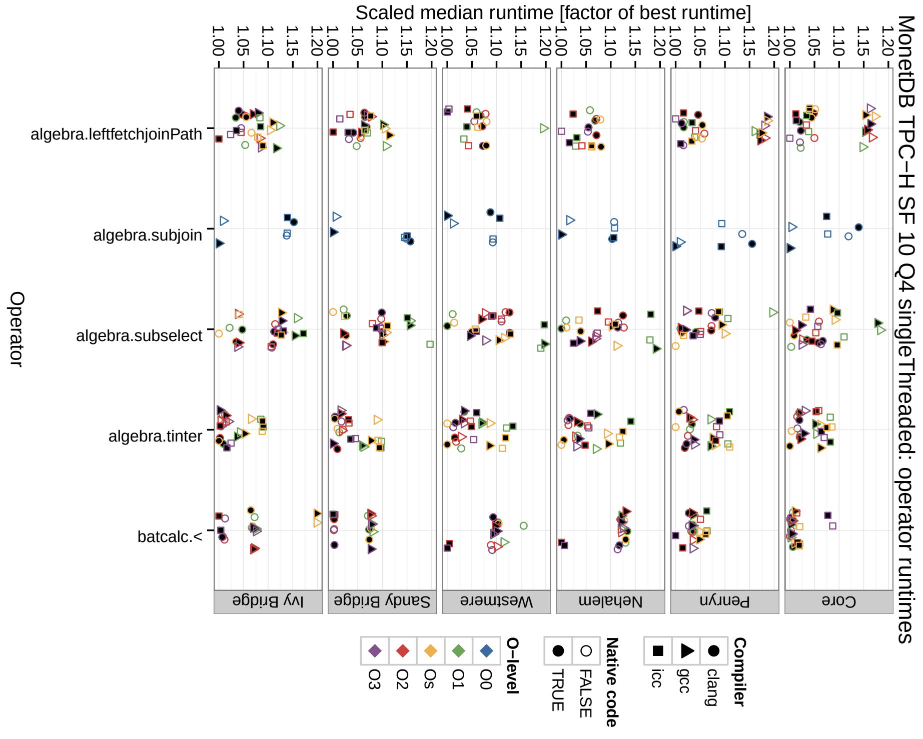
MonetDB uses the operator-at-a-time query execution model, i.e. each operator consumes its input completely before starting to produce the output. This can also be described as: all operators are blocking. If we restrict ourselves to single-threaded execution, we can freely exchange the implementation of each operator in a query plan, as long as they produce the same results. Choosing another algorithm (like hash-join vs. sort-merge join), or implementation technique (like branching vs. branch-free selection) are the typical options considered in state-of-the-art database systems. We do not consider these orthogonal techniques in the following. Rather, we consider that we could also exchange the physical operators' binary representation to another one, compiled using a compiler setup that would yield better performance. To make this possible the operators' code need to be dynamically loaded from a library, or the database system has to use just-in-time compiling. Unfortunately, MonetDB does not currently implement any of these techniques. Nevertheless, dynamic loading of operators could easily be integrated into MonetDB.

We can, however, estimate the effects of compiling physical operators using different setups without actually building such a MonetDB instance. Our assumption is that since MonetDB uses the operator-at-a-time execution model, a single operator's runtime does not depend on the compilation setup of the rest of the operators. Let us illustrate this with an example: Consider the case when the whole system is compiled with `gcc -O3`, and we would like to estimate the effect of exchanging the `algebra.subselect` operator with one compiled using a different setup. We first compile a MonetDB instance for all of the alternative compiler setups we would like to consider (e.g. `icc -O3` and `clang -O3`). Eventually we execute a representative query that uses the `algebra.subselect` operator in its execution plan in all of the MonetDB instances, and measure the operator's running time. If we measured the lowest runtime in the instance compiled with `icc -O3`, then we should use `icc -O3` to compile that particular operator, and can compile the rest of the system using `gcc -O3`. The expected performance gain according to our assumption should be the difference in the `algebra.subselect` operator's runtime in those two systems.

For the purpose of measuring operator runtimes we can again use MonetDB's tomograph tool, which can calculate the running time of each physical operator in a query plan by collecting traces and cumulating the measurements. This is what we can see in Figure 6.9 for TPC-H Q4. We only show the five most expensive operators' runtimes measured in all 30 MonetDB instances compiled without advanced compilation techniques, i.e. without link-time optimization and profile-guided optimization. To better see the differences among the top performing instances we have scaled each operator's runtime to that of the best one, which we show on the right figure. We have a very interesting case here:



(a) single-threaded



(b) single-threaded scaled

Figure 6.9: Operator runtimes of TPC-H Q4 in MonetDB

to achieve the lowest query time we need to compile the three most expensive operators using three different compilers, and three different O-levels, as shown in Table 6.7

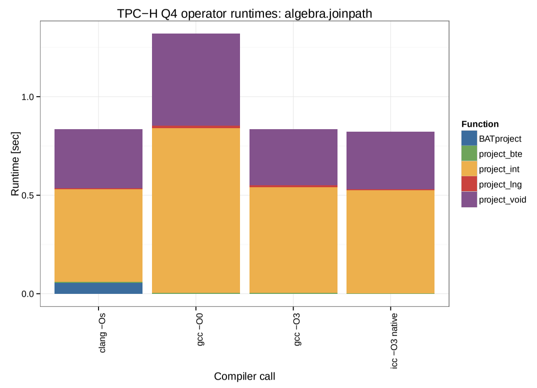
Operator	**Best compiler setup**
`algebra.leftfetchjoinPath`	`icc -O2 -march=native`
`algebra.subjoin`	`gcc -O0 -march=native`
`algebra.subselect`	`clang -Os`

Table 6.7: Best compiler setups for the long-running operators of TPC-H Q4 on Sandy Bridge.

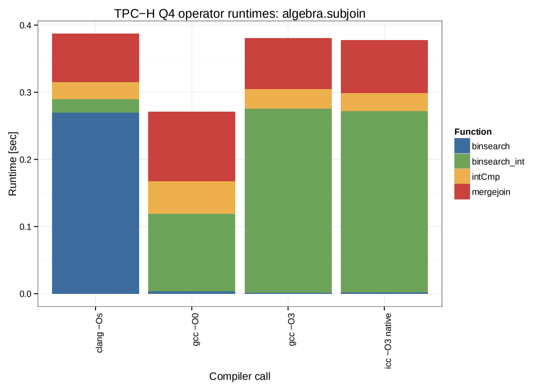
As a proof of concept one can hand-compile a MonetDB instance for a given query. This requires identifying the compilation units where each operator is implemented. If the implementation of multiple operators reside in the same compilation unit, and they happen to require two different compiler setups to reach their best runtime, we cannot build the optimal MonetDB instance without modifying the build process. This requires to split such compilation units, that contain more than one database operator's implementation, into multiple compilation units. Only then can we apply the right compilation setup for each database operator separately.

Thus, we have decided to use runtime profiling to identify the compilation units and functions where each database operator spends its runtime. For this experiment we consider TPC-H Q4 executed single-threaded on Sandy Bridge. The runtime breakdowns of the operators in Table 6.7 are shown in Figures 6.10a, 6.10b, and 6.10c. It is interesting to identify where `gcc -O0` wins its performance gain for the `algebra.subjoin` operator, namely in the binary search function, by spending only 40% of the time in that function compared to the other three compilation setups' runtimes shown in that Figure.

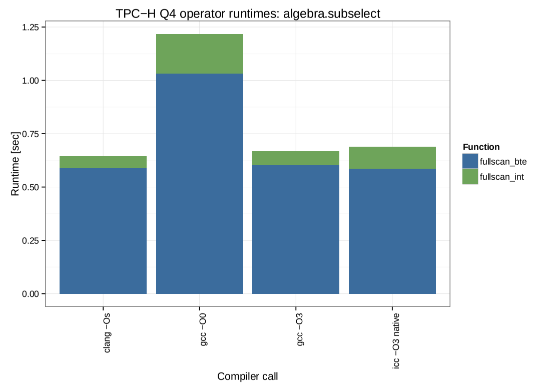
This procedure can be applied to any query for identifying the compilation units that are relevant for each operator in the query plan. Upon conflicting compilation setups for the same compilation unit for two or more different operators, special care has to be taken to split up the affected compilation units into multiple ones, such that each operator's code can be compiled using the proper compiler setup. Due to the complexity of this procedure, we have decided to leave building a perfect MonetDB instance for a given query as future work.



(a)



(b)



(c)

Figure 6.10: Runtime breakdown of the three most expensive operator in TPC-H Q4 on Sandy Bridge.

6.7 Related work

The major C/C++ compilers offer the same standard optimization levels (e.g. `-O3`), however, different compilers do not include the same optimizations for the same O-level. Nevertheless, these compilers allow the users to toggle specific optimizations one-by-one as well, instead of relying on the predefined set of optimizations included in the O-levels. This step requires deep understanding of both the compiler at hand, and general compiler construction knowledge. Since most compilers have a very large number of individual optimization flags (e.g. GCC had 207 flags at the time of this writing), it is prohibitive to find the optimal set of flags for compiling a given application by the user on a trial-and-error basis. Instead, this should be performed using a heuristic optimizer algorithm. The authors of COLE: Compiler Optimization Level Exploration [31] have done this for the SPEC CPU Benchmark and the GCC compiler, and have shown that only 25% of the optimizations included in standard O-levels contributed to at least one of the Pareto-optimal optimization levels. There is one prohibitive factor of doing the same for the TPC-H benchmark: one benchmark run for TPC-H on a disk-based database system can take much longer than one run for the SPEC CPU benchmark, however, for in-memory database systems the runtimes should be much more comparable.

Some state of the art database systems perform query compilation just-in-time [64, 65, 66, 57, 42]. This means that each incoming query is compiled into a separate executable or library function. Yet, to the best of our knowledge, there is no system that compiles the queries using different compilers and compiler options. One notable example of a system performing just-in-time query compilation is LegoBase [41]. The input for the LegoBase system is a physical query plan. As a first step the nodes in the plan are replaced with operators written in Scala, by which they get an intermediate representation (IR). By using the Lightweight Modular Staging [56] system they can perform so-called staging optimizations on the IR that standard query compilers cannot do. These include inter-operator optimizations, like transforming a Volcano-style engine to a push-based one and eliminating redundant materialization steps. Further staging optimizations can perform data-structure specializations or change the data layout. Another major in-memory database system that does query compiling is HyPer [40]. It translates queries [48] using C++ templates and the LLVM framework into very efficient machine code.

Vectorwise (now Actian Vector) is an improved version of MonetDB, that makes use of extensive vectorisation. Compared to MonetDB, it has the advantage of performing chunked/vectorised query execution. The authors of [54] propose a framework to be used with Vectorwise, that keep multiple flavours of each database operator, and choosing the right one adaptively. Vectorised execution makes it possible to choose a (possibly) different flavour for the operators for each chunk of the data. This allows for a faster learning process, since feedback on the flavours' performance is available already during the query execution. Even more importantly it gets possible to choose a different operator flavour during query execution when an abrupt performance deterioration is detected. However, this is a hard online problem, and on top of that it is not known whether the performance drop can be mitigated by choosing the right operator flavour (e.g. upon changing branching probabilities). Our work differs from the previous one in that we perform a more systematic approach for choosing the compiler setup, i.e. not only the compiler, but also the optimization-level and some other flags as well. Furthermore, we also explore some advanced compilation modes, including link-time optimization and profile-guided optimization.

Vectorization and exploiting architecture specific CPU instructions has been shown to boost performance of some common database operators, e.g. exploiting SIMD for sorting and fast scans [63, 13]. Their effect on system-wide performance, however, have not yet been explored. Considering the TPC-H benchmark, it has been shown in [9] that the TPC-H queries use a wide variety of database operators, and require a broad set of query plan optimizations to achieve peek performance.

6.8 Conclusion and Future Work

In this work we have considered multiple compilation strategies and approaches, which we would like to summarize in the following.

6.8.1 When to compile

We can distinguish between static and dynamic compiling. In case of static compiling the system is compiled before its first use, and a running system does not get recompiled. On the other hand, dynamic compilation involves either recompiling parts of the system, or creating a separate executable for each incoming query. The latter approach is termed just-in-time compiling, where each incoming query is first compiled and the resulting binary gets eventually executed. This allows for extensive optimizations, like avoiding function call overhead, iterator interfaces, etc. This technique is applied in some very efficient main-memory database systems [49, 41].

As introduced before, in case of static compiling we do not recompile a running system, nor perform on-the-fly compiling on parts of it. Nevertheless, when making the upfront decision on the compiler settings to be used, we want to make a decision that yields good query performance. This is an optimization problem that has multiple flavors, as there are multiple granules in query processing where we can choose a static compiling strategy:

Upon building the system. The simplest and almost omnipresent approach is to choose a single uniform compiling strategy for the whole data management system. In this case the compiler and possible optimization flags are typically set in a Makefile and applied throughout the whole compilation process. Thus, there is a single instance of the executables and libraries that make up the whole system.

Upon executing a query. Assume we have multiple versions of a database system, each of them compiled with different compilers and/or compiler flags. In this case for each incoming query we can choose which instance of the system to route the query to. The goal of routing is to execute the query on the system that would yield the lowest query time. Note that in this work we are not considering this as a load balancing problem, since we do not want to restrict ourselves to routing each query to be executed in parallel to different instances of the system. It is further not assumed that the different instances of the system have to share the computer's resources according to an upfront decision.

Upon executing an operator. Database systems create a logical query plan for each incoming query, and their query optimizer transforms this plan to a physical query plan. A logical database operator might have multiple matching physical database operators that conceptually all perform the same operation, but differ either in algorithmic aspects (like sort-merge join vs. hash-join), or in implementation details (e.g. branch-free selection vs. branching selection). One dimension not yet fully explored is to choose the compiling strategy as well for a given physical operator. Data management systems are typically complex enough such that dynamically loading operator libraries can pay off. This case makes it technically viable to choose the compiling strategy of the physical database operators as well, since that would just necessitate to choose the according flavor of the library storing the procedure to be loaded. In case the whole database system is built statically, without any components of it being loaded dynamically, some sort of name mangling has to be applied when linking multiple versions of the same function compiled with different compiler settings.

6.8.2 Putting it all together

In this concluding section we are going to put all approaches discussed in Section 6.8.1 together. In Figures 6.11a and 6.11b we show the query time improvements (if any) over the query time of the MonetDB instance compiled with the default settings, i.e. `gcc -O3`. The compiling approaches used in the Figures correspond to the following techniques discussed so far:

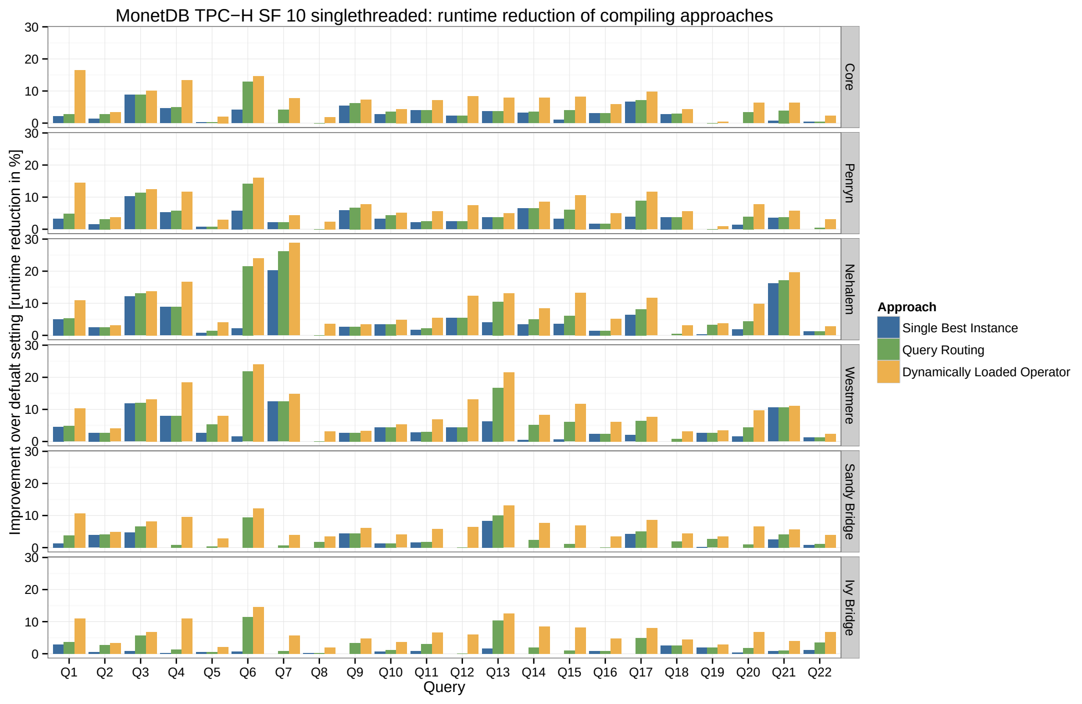
Single Best Instance: Choosing a single, global compiler setup upon building the system.
Query Routing: Choosing the compiler setup upon executing each incoming query.
Dynamically Loaded Operators: Choosing a compiler setup upon executing each operator in the query plan. Notice that these query times are only estimated based on the individual operators runtimes measured on a uniformly compiled MonetDB instances. For reasons discussed in Section 6.6 this leads to a reasonable approximation of the actual query times.

When looking at the results we can in general tell that the improvements over the default setup are higher for single-threaded execution, than for multithreaded. In the latter case we can improve at least 5% for most queries, with some improvements reaching up to 20%. On the other hand, in the single-threaded case we can improve at least 8% for most queries, and some improvements reach even 30%. We can also discover some patterns in the results: The four older machines (Core, Penryn, Nehalem, and Westmere) reach on average significantly higher improvements, than the two newest ones (Sandy Bridge and Ivy Bridge) in the single-threaded case. Interestingly, Q1 and Q6 significantly benefited from the various compiling approaches in all machines. In the multithreaded case there is one interesting phenomena: seemingly there are missing data points for some queries, e.g. Q1 and Q22 on Core and Penryn, which actually mean non of the approaches could improve the query time of the default setup.

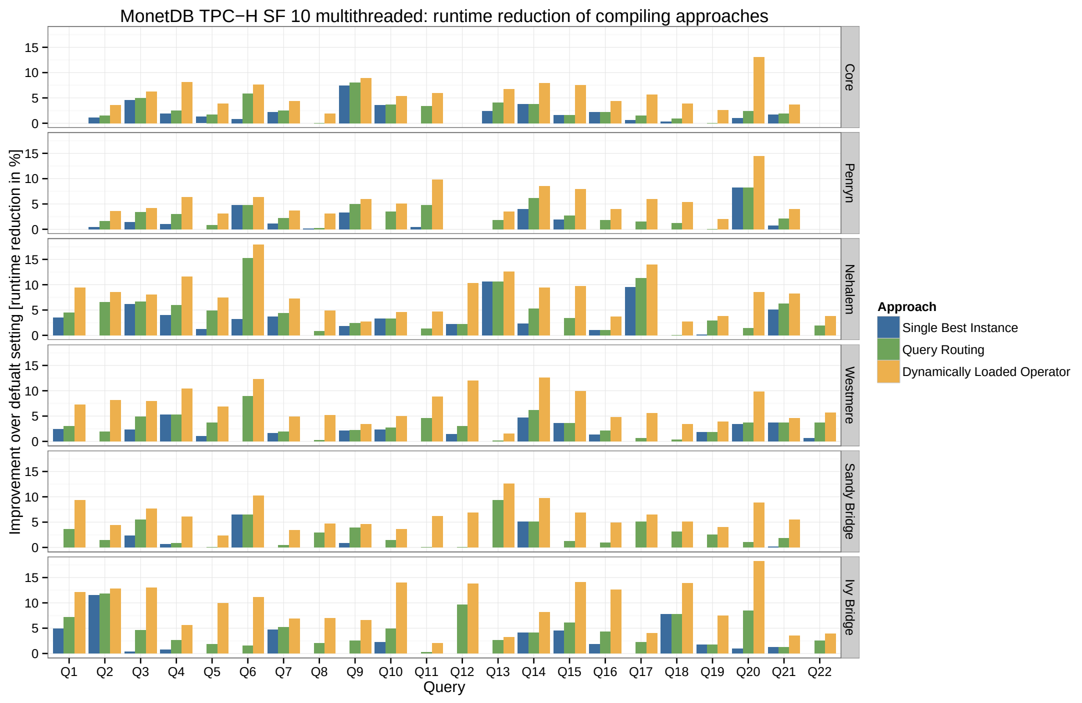
To conclude, by using the techniques introduced in this chapter we can achieve a non-negligible performance improvement of at least 5%–8% for most queries, and for some few queries even 20%–30% when executing the TPC-H benchmark in MonetDB. We can improve query performance in 50% of the cases by choosing a suitable compiler setup system-wide. However, for improving the query performance in the rest of the cases, we have to use query routing or eventually dynamically loaded operators. We have also shown that the efficiency of these approaches varies drastically between machines with different CPU architectures. All these findings support, that we should not consider the compiler as just another abstraction layer, but as a valid candidate for performance tuning when building main-memory database systems.

6.8.3 Future work

It is interesting future work to explore the same compilation techniques we did for MonetDB in LegoBase as well. LegoBase produces C-code as an intermediate result, which gets eventually compiled into machine code. Here we could also choose proper compiler settings to improve the performance of the resulting query executable.



(a) Single-threaded execution



(b) Multithreaded execution

Figure 6.11: Query time improvements for the three routing approaches.

Appendix

6.A The performance implications of compiling a key-value store

On the market of key-value stores, redis is one of the most popular systems. As a key-value store it is most suited for point queries, so we have decided to evaluate its performance on the benchmarks shipped with redis, measuring the throughput (requests fulfilled per second) of all basic operations of a key-value store. We report the median throughput for reasons detailed in Section 6.3.2.

We have built redis 3.0.4 from source on the six different machines described in Section 6.2. We have modified the build script to allow for specifying the optimization level, or any other compiler flags we would like to use.

We have built a separate redis instance for all possible combinations of compiler settings investigated in this work: 3 compilers $\times$ 5 O-levels $\times$ native/non-native code, yielding 30 different instances. It is worth mentioning that the compiler switch needed to generate assembly code exploiting larger registers, vectorised instructions, etc. (`-march=native`) is not enabled by default in redis's build scripts. We assume the following scenario to be the default compiler setting for compiling redis: `gcc -O2`[3], and non-native code.

The build process of redis is comprised of running `make`, and `make install`. The latter step runs in less than 1 second. Depending of the compiler settings and the machine, the runtime of the `make` step, comprising the actual compiling of the system, varies heavily depending on the compiler settings. These latter compile times are shown in Figure 6.12.

[3]-O2 is set by default in the Makefile of redis

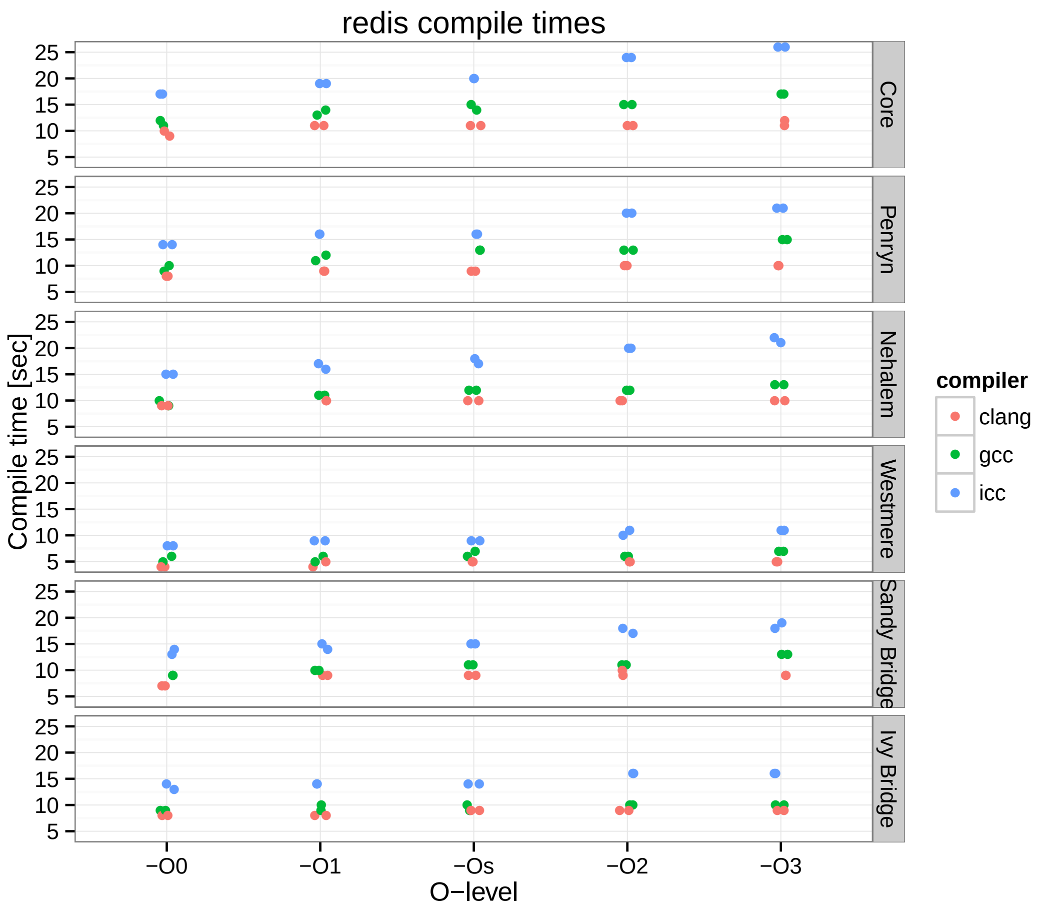


Figure 6.12: redis compile times.

6.A.1 Experiments

When using a redis instance ”as is”, i.e. without any tuning and special care in starting up the server and client processes, we have experienced huge throughput variance: for the `get` operation it oscillated between 120.000 and $200,000$, which is clearly unacceptable. After some deep investigation on the internals of redis, we have come to the following guideline on how to make the throughput of redis stable:

- disable snapshotting of the database to disk
- pin server and client processes to the same CPU, but to different cores

By doing so we could achieve less than 2,5% maximal discrepancy between median and average throughputs.

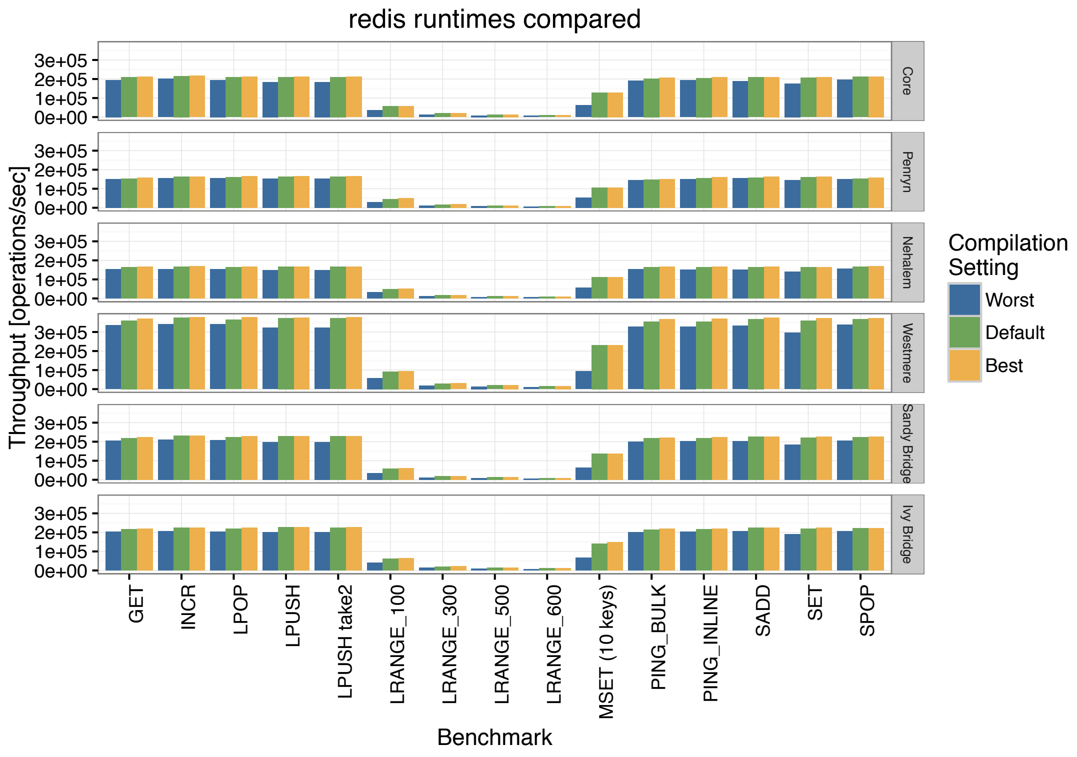


Figure 6.13: Best and worst redis benchmark throughputs (higher is better) showing the throughputs of the default compiler settings (gcc -O2, non-native code) as well.

The benchmark results are shown in Figure 6.13. The peek throughput of the benchmark operations are strongly influenced by the CPU clock speed, similarly to the case of compiling MonetDB. Clearly the GET and SET operations are the most interesting ones for a key-value store. Nevertheless, there are other benchmark operations also included in the standard redis benchmark, performing utility operations that build upon the basic GET and SET operations.

On all machines there is a negligible difference between the performance of the best and the default redis instances. On the other hand, the worst possible compilation setting can result in a visible performance loss, which effects the basic SET operation as well, especially for our fastest machine, Westmere. The more complex operations are affected even more: the MSET operation can become factor 3 slower.

We can conclude, that the performance span of the various redis instances is much more narrow, than what we have seen in case of compiling MonetDB. redis uses a protocol for communicating between clients and servers, which adds an overhead on each call to the GET and SET operation. This, ultimately cannot be "optimized away" by using a suitable compiler setup, nor can it be improved much, as our experiments have shown. The take-away message is that redis and probably other key-value stores are not fragile to compilation settings.

Chapter 7

Summary

In this last chapter of this thesis we summarize the main results and lessons learned. We conclude this work by highlighting some possible future work on vertical partitioning and compiling database systems.

7.1 Vertical Partitioning for Legacy Row Stores

There are a number of vertical partitioning algorithms proposed in the literature. In this work, we presented a systematic and comprehensive study of vertical partitioning algorithms. We categorized vertical partitioning algorithms along three dimensions and surveyed six different algorithms. We experimentally evaluated these six algorithms under a common configuration setting. We introduced four metrics to compare different vertical partitioning algorithms and showed results from the TPC-H benchmark. Our results identified the trade-offs between optimization time and workload runtime improvements, improvements over row and column layouts, and effects of database buffer size.

7.1.1 Lessons Learned

In this chapter, we compared different vertical partitioning algorithms and studied ways to pick one vertical partitioning algorithm over another for row-oriented database systems.
1. We don't really need brute force. The brute force algorithm spends an extremely long time to compute the layouts (more than an hour for TPC-H). On the other hand, the vertical partitioning algorithms evaluated in this work terminate in at most a few minutes. In fact, AutoPart and HillClimb take less than 1 second to compute the layouts for all tables in the TPC-H benchmark. Still both AutoPart and HillClimb find *exactly* the same solution as the brute

force algorithm. HYRISE takes slightly more than a second to compute the layouts but it is only 2.21% off from the brute force algorithm, in terms of query costs. Similarly Trojan takes a couple of minutes for optimization, however it is just 0.01% off from the brute force algorithm in terms of estimated runtime. This is an important result and shows that we do not really need the brute force algorithm. Several heuristics, as proposed in different algorithms, are good enough.

2. Watch out for the buffer size. The performance of vertically partitioned layouts depend heavily on the database buffer size. In fact, the buffer size can impact the query workload runtimes by as much as factor 20. Thus buffer size is a crucial consideration when computing vertical partitioning. Furthermore, our measurements reveal that vertical partitioning improves over column layout only for buffer sizes less than 100 MB. This means if we can have a system with buffered reads of more than 100 MB at a time, then we better use column layout. Put another way: if we want to avoid vertical partitioning then we must increase the buffer size of our database system. This is one of the core results of this chapter.

3. HillClimb is the best algorithm for disk-based systems. Amongst the six vertical partitioning algorithms compared in this chapter, HillClimb turns out to be the best for the TPC-H queries. HillClimb offers the best trade-off between optimization time and workload runtime performance. It spends 4 orders of magnitude less time in optimization and still finds the same vertical partitioning as the brute force algorithm. As a result, the optimization time of HillClimb pays off the earliest (just after 25% of TPC-H workload) over row layout. Furthermore, from our experience HillClimb is also one of the easiest algorithms to understand and implement.

4. Column layouts are often good enough. On the TPC-H benchmark (i.e. all 22 queries) the vertical partitioning algorithms could improve over column layout by only up to 3.7%. This is because the attribute access patterns over all 22 queries are quite fragmented and it is hard to find column groups which satisfy most of the queries. Indeed, the improvements over column layout go up to 24% when using a small subset of the TPC-H workload (see Figure 2.7). But still the improvements over column layout are not dramatic. To investigate this further, we tried three changes in our experimental setup — using a different benchmark, using a different cost model, and using a commercial database system which supports column grouping.

7.1.2 Future Work

We have brought several vertical partitioning algorithms for the sake of an apples-to-apples comparison under the same unified setting. There we have found that HillClimb is a simple, yet very fast and efficient algorithm. Thus, it would be straightforward to adjust HillClimb to work in other interesting settings as well. In particular, it would be quite promising to apply it for full-replication scenarios, like Hadoop MapReduce, similarly to Trojan Layouts. More specifically the query- and attribute partitioning parts of Trojan Layouts could be completely replaced by HillClimb. This would have the immediate consequence of a runtime reduction by orders of magnitudes, while still maintaining the close to optimal solution quality, as observed in the setting without replication.

7.2 Query Processing on Top of Flat Files

We have learned that for relatively simple queries operating on a single file only, both a custom-built C++ application and AWK can offer a shorter time-to-query than PostgreSQL, with the former being the fastest. For more complex queries operating on multiple files PostgreSQL is the best option. Though it still has the longest time-to-query, it offers a superior query performance on subsequent runs, which makes the initial data loading costs quickly pay off.

We have also shown that the standard configuration settings for PostgreSQL are inappropriate, and using a tuned configuration can reduce the loading time by 17%. Furthermore, if the flat files reside on the server, then we can reduce the loading time by 5% just by using the right loading command.

We have seen that when processing flat files with AWK, it is better if the flat files are stored in fixed-width format. This is especially true, if only a few attributes are accessed from each record, where the increase in performance can be up to factor 8.

7.3 Runtime Fragility in Main Memory

7.3.1 Fragility of Hand-coded Queries in Main-Memory

In this chapter we have identified various sources of query time fragility – implementation factors that can change the performance of a query by factors in an unpredictable way. We have investigated the fragility of both micro-benchmarks and complex analytical benchmarks. We have considered the CPU architecture, the compiler, and the compiler flags as important factors. We have introduced the memPAX layout and compared its fragility to column layout and row layout.

We have shown that when querying tables with 1–byte integer columns a very high fragility is to be expected, in our case leading to a performance drop of up to factor 25. In case of more complex schemas and queries the inhomogeneity of the schema has a direct effect on the fragility. Applying dictionary- and domain encoding to the columns have reduced fragility by 50% to 80% in our experiments on the TPC-H benchmark.

We have found a use-case in query processing where using powers of 2 is always a suboptimal choice, leading to one more cause of fragile query times. We have shown how to choose the chunk sizes of the memPAX layouts to improve tuple-reconstruction costs by 20%.

7.3.2 Fragility of Compiling a Database System

In this chapter we have presented an exhaustive experimental study on compiling the MonetDB database system. Given the large number of tuning knobs of compilers we first discuss the ones that could be the most important starting points of performance tuning when compiling a whole database system, namely: i) the compiler itself, ii) the optimization level, and iii) advanced compilation modes. For the above categories we have considered: i) the three most popular C/C++ compilers (GCC, clang, and the Intel C/C++ compiler), ii) all five standard O-levels, and iii) link-time optimization (LTO), and profile-guided optimization (PGO). We have considered all combinations of the above knobs, and have built 90 separate MonetDB instances using each of them. We have evaluated their performance on the TPC-H benchmark, using six servers equipped with CPUs of subsequent generations in our experiments. We have presented two techniques for improving query performance that build upon changing compiler setups. These approaches work on two different levels: on a per-query level, and on the physical database operator level.

The techniques introduced in Chapter 6 can help us to achieve a non-negligible performance improvement of at least 5%–8% for most queries, and for some few queries even 20%–30% when executing the TPC-H benchmark in MonetDB. We can improve query performance in 50% of the cases by choosing a suitable compiler setup system-wide. However, for improving the query performance in the rest of the cases, we have to use query routing or eventually dynamically loaded operators. We have also shown that the efficiency of these approaches varies drastically between machines with different CPU architectures. All these findings support, that we should not consider the compiler as just another abstraction layer, but as a valid candidate for performance tuning when building main-memory database systems.

7.3.3 Future Work

It is interesting future work to explore the same compilation techniques we did for MonetDB in LegoBase as well. LegoBase produces C-code as an intermediate result, which gets eventually compiled into machine code. Here we could also choose proper compiler settings to improve the performance of the resulting query executable.

List of Figures

List of Tables

Bibliography

[1] Aerospike DBMS. `http://www.aerospike.com`.

[2] redis key-value store. `http://redis.io`.

[3] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *EDBT*, pages 1–10, 2013.

[4] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *ACM SIGMOD*, pages 359–370, 2004.

[5] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[6] Anastassia Ailamaki et al. Weaving Relations for Cache Performance. In *VLDB 2001*, pages 169–180, 2001.

[7] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *ACM SIGMOD*, pages 241–252, 2012.

[8] Eric T Bell. Exponential numbers. *The American Mathematical Monthly*, 41(7):411–419, 1934.

[9] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[10] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2):1648–1653, 2009.

[11] Peter A. Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, pages 61–76, 2013.

[12] Bonnie++. `coker.com.au/bonnie++`.

[13] David Broneske, Sebastian Breß, and Gunter Saake. Database scan variants on modern cpus: a performance study. In *In Memory Data Management and Analysis*, pages 97–111. Springer, 2015.

[14] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52. ACM, 2010.

[15] W. W. Chu and I. T. Ieong. A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems. *IEEE Trans. Softw. Eng.*, 19(8):804–812, 1993.

[16] Joe Conway. PL/R - R Procedural Language for PostgreSQL. `www.joeconway.com/plr/`.

[17] Douglas W. Cornell and Philip S. Yu. A Vertical Partitioning Algorithm for Relational Databases. In *ICDE*, pages 30–35, 1987.

[18] Douglas W. Cornell and Philip S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Trans. Softw. Eng.*, 16(2):248–258, 1990.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual.*

[20] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[21] Glenn Fowler. cql: Flat file database query language. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 11–21, 1994.

[22] Mrunal Gawade and Martin Kersten. Tomograph: Highlighting query parallelism in a multi-core system. In *Proceedings of the Sixth International Workshop on Testing Database Systems*, DBTest '13, pages 3:1–3:6, New York, NY, USA, 2013. ACM.

[23] Taras Glek and Jan Hubicka. Optimizing real world applications with GCC Link Time Optimization. *arXiv preprint arXiv:1010.2196*, 2010.

[24] Frank E Grubbs. Sample criteria for testing outlying observations. *The Annals of Mathematical Statistics*, pages 27–58, 1950.

[25] Martin Grund et al. HYRISE: a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[26] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[27] Michael Hammer and Bahram Niamir. A Heuristic Approach to Attribute Partitioning. In *ACM SIGMOD*, pages 93–101, 1979.

[28] Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, 2003.

[29] HBase. `hbase.apache.org`.

[30] Jeffrey A. Hoffer and Dennis G. Severance. The Use of Cluster Analysis in Physical Data Base Design. In *VLDB*, pages 69–86, 1975.

[31] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174. ACM, 2008.

[32] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.

[33] Milena Ivanova, Martin Kersten, and Stefan Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDBM*, pages 485–494. Springer, 2012.

[34] Raj Jain. *The art of computer systems performance analysis.* John Wiley & Sons, 1991.

[35] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In *BIRTE*, pages 65–80, 2011.

[36] Alekh Jindal, Endre Palatinus, Vladimir Pavlov, and Jens Dittrich. A Comparison of Knives for Bread Slicing. *PVLDB*, 6(6):361–372, 2013.

[37] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *ACM SOCC*, pages 21:1–21:14, 2011.

[38] Alekh Jindal, Felix Martin Schuhknecht, Jens Dittrich, Karen Khachatryan, and Alexander Bunte. How Achaeans Would Construct Columns in Troy. In *CIDR*, 2013.

[39] William T. McCormick Jr., Paul J. Schweitzer, and Thomas W. White. Problem Decomposition and Data Reorganization by a Clustering Technique. *Operations Research*, 20(5):993–1009, 1972.

[40] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[41] Ioannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. In *Proceedings of the VLDB Endowment*, volume 7, 2014.

[42] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 613–624. IEEE, 2010.

[43] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. The Hekaton Memory-Optimized OLTP Engine. *IEEE Data Eng. Bull.*, 36(2):34–40, 2013.

[44] Konrad Lorincz, Kevin Redwine, and Jesse Tov. Grep versus FlatSQL versus MySQL. 2003.

[45] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant Loading for Main Memory Databases. *PVLDB*, 6(14):1702–1713, 2013.

[46] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical Partitioning Algorithms for Database Design. *ACM TODS*, 9(4):680–710, 1984.

[47] Shamkant B. Navathe and Mingyoung Ra. Vertical Partitioning for Database Design: A Graphical Algorithm. In *ACM SIGMOD*, pages 440–450, 1989.

[48] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[49] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[50] Endre Palatinus and Jens Dittrich. Runtime Fragility in Main Memory. In *Proceedings of the 2016 Joint Workshop on Accelerating Analytics and In-Memory Data Management Systems*, LNCS, 2016.

[51] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, pages 383–392, 2004.

[52] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2008.

[53] Holger Pirk et al. CPU and cache efficient management of memory-resident databases. In *ICDE 2013*, pages 14–25, 2013.

[54] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242. ACM, 2013.

[55] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-Time Query Processing. In *ICDE*, pages 60–69, 2008.

[56] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[57] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40. ACM, 2011.

[58] Star Schema Benchmark. `www.cs.umb.edu/~poneil/StarSchemaB.pdf`.

[59] TPC-H. `http://www.tpc.org/tpch/`.

[60] Transaction Processing Performance Council. TPC-H benchmark specification. *Published at http://www.tpc.org/tpch*, 2013.

[61] Vertica. `vertica.com`.

[62] Bernard L Welch. The generalization of Student's problem when several different population variances are involved. *Biometrika*, pages 28–35, 1947.

[63] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.*, 2(1):385–394, August 2009.

[64] Barry M Zane, James P Ballard, Foster D Hinshaw, Dana A Kirkpatrick, and Premanand Yerabothu. Optimized SQL code generation, September 30 2008. US Patent 7,430,549.

[65] Rui Zhang, Richard T Snodgrass, and Saumya Debray. Application of micro-specialization to query evaluation operators. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 315–321. IEEE, 2012.

[66] Rui Zhang, Richard T Snodgrass, and Saumya Debray. Micro-Specialization in DBMSes. In *2012 IEEE 28th International Conference on Data Engineering*, pages 690–701. IEEE, 2012.