Cache Conscious Data Layouting for In-Memory Databases

A Thesis submitted for the degree of Diplom Informatiker
at the Institute of Computer Science/Humboldt-Universität zu Berlin

Holger Pirk <pirk@informatik.hu-berlin.de>
Matrikelnummer: 195155

Gutachter:

- Professor Doktor Ulf Leser, Humboldt-Universität zu Berlin
- Doktor Alexander Zeier, Hasso Plattner-Institut für Softwaresystemtechnik

Betreuer:

- Martin Grund, Hasso Plattner-Institut für Softwaresystemtechnik
- Jens Krüger, Hasso Plattner-Institut für Softwaresystemtechnik

Abstract

Many applications with manually implemented data management exhibit a data storage pattern in which semantically related data items are stored closer in memory than unrelated data items. The strong sematic relationship between these data items commonly induces contemporary accesses to them. This is called the principle of data locality and has been recognized by hardware vendors. It is commonly exploited to improve the performance of hardware. General Purpose Database Management Systems (DBMSs), whose main goal is to simplify optimal data storage and processing, generally fall short of this claim because the usage pattern of the stored data cannot be anticipated when designing the system. The current interest in column oriented databases indicates that one strategy does not fit all applications. A DBMS that automatically adapts it's storage strategy to the workload of the database promises a significant performance increase by maximizing the benefit of hardware optimizations that are based on the principle of data locality.

This thesis gives an overview of optimizations that are based on the principle of data locality and the effect they have on the data access performance of applications. Based on the findings, a model is introduced that allows an estimation of the costs of data accesses based on the arrangement of the data in the main memory. This model is evaluated through a series of experiments and incorporated into an automatic layouting component for a DBMS. This layouting component allows the calculation of an analytically optimal storage layout. The performance benefits brought by this componentt are evaluated in an application benchmark.

Zusammenfassung

Viele Anwendungen mit selbst implementierter Datenhaltung zeigen ein Speicherverhalten bei dem semantisch zusammenhängende Daten im Speicher näher beieinander gespeichert werden als semantisch unzusammenhängende Daten. Der starke semantische Zusammenhang zwischen diesen Daten führt zu häufigen zeitnahen Zugriffen auf sie. Dieses Prinzip, das Prinzip von semantischer Datenlokalität, wurde von Hardwareherstellern erkannt und ist die Basis für viele Verbesserungen der Hardware die die Ausführungsgeschwindigkeit der Software steigern. Nicht spezialisierte Datenbank Management Systeme (DBMSe), deren Ziel die Vereinfachung einer optimalen Datenspeicherung und -verarbeitung ist, verfehlen dieses Ziel oft weil das Nutzungsmuster der gespeicherten Daten beim Entwickeln des Systems nicht bekannt ist und daher nicht zur optimierung genutzt werden kann. Das aktuelle interesse an spaltenorientierten Datenbanksystemen macht deutlich das eine Strategie nicht für alle Fälle geeignet ist. Ein Datenbanksystem dass seine Speicherstrategie dem Nutzungsmuster der Datenbank anpasst kann deutliche Leistungssteigerungen erzielen, da es die Optimierungen der Hardware am besten ausnutzen kann.
Diese Diplomarbeit soll einen Überblick über die Optimierungen geben, die auf dem Prinzip der semantischen Datenlokalität basieren. Ihr Effekt auf die Ausführungsgeschwindigkeit soll untersucht werden. Aufbauend auf den resultierenden Erkenntnissen soll ein Modell eingeführt werden dass eine Abschätzung der Datenzugriffskosten in Abhängigkeit der Anordnung der Daten im Speicher erlaubt. Dieses Modell wird durch eine Reihe von Experimenten evaluiert und anschliessend in eine Komponente zur Anordnung der Daten eines DBMSs eingearbeitet werden. Diese Komponente erlaubt die Berechnung einer analytisch optimalen Anordnung der Daten im Speicher. Theorie, Umsetzung und Leistungssteigerungen durch diese Optimierung werden beschrieben.

Acknowledgements

I want to thank Prof. Dr. Hasso Plattner and Dr. Alexander Zeier for giving me the possibility to work on this thesis. My mentors for this thesis, Prof. Dr. Ulf Leser, Martin Grund and Jens Krüger provided valuable feedback for which I am grateful. My father, Thomas Pirk, helped me by providing valuable insights into 1980's hardware. This thesis would not have been possible without Anja Prüfert. She took care of our daughter when I was busy writing and listened to my half finished ideas when I needed to talk things through.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Potsdam, den 17. Januar 2010

Einverstandniserklärung

Hiermit erkläre ich mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Potsdam, den 17. Januar 2010

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Recent developments in Database Management Systems (DBMS) have produced an interesting new concept: column oriented DBMS (column-stores or CStores) [3]. In contrast to the traditional record-wise storage of row oriented DBMS (row-stores) [4] data is stored attribute-wise. This improves the performance of queries that operate on many tuples but few attributes, most notably analytical (OLAP) queries. Column oriented DBMS are seen as a strong competitor to classical warehouses that preaggregate data to support performant analytics [5, 6]. Transactional (OLTP) performance, however, is diminished by column-based storage because transactional queries usually operate on many attributes but few tuples. In column oriented storage, These have to be reconstructed from the values of their attributes which takes time [7].

The sacrifice of transactional performance for analytical performance is feasible if one usage pattern outweighs the other by far. Since most businesses have transactional as well as analytical needs, it is common practice to have dedicated, redundant systems with different schemas for each.

Redundant DBMS

Redundant copies of transactional data can be stored in two forms: preaggregated in a data warehouse or in the unaggregated form, but in a specialized analytical DBMS, e.g., a column-store.

A common representative of the redundant storage approach is the following setup [8]: a row oriented transactional DBMS for the OLTP-load and a Data Warehouse for the analytical needs. New data enters the operational system as it occurs and is loaded into the Warehouse in intervals using an *Extract, Transform and Load (ETL)* process [9]. This, however, has several drawbacks:

1. The data that has not been transferred to the OLAP-store yet will not appear in the aggregated results, which renders the OLAP-store constantly out of date [10].

2. All data has to be held twice which increases the costs for hardware acquisition and maintenance [9].

3. The update process has to be maintained and run periodically to keep the OLAP-store reasonably up to date. Since this process can be complicated, the added costs in hardware and personal can be high [9].

The costs may increase even further with the complexity of the user's requirements. A common requirement that is especially interesting is *real-time reporting*, i.e. reporting on data that has just entered the transactional system. Established vendors support real-time reporting, e.g., through means of *Active Warehousing*.

Active Warehousing

To increase the efficiency of business operations it is often required to do analytics on a relatively short period of time (an hour or even minutes). This kind of *Operational Reporting* [11] is a trend that has been recognized by vendors [12]. They aim at supporting it by means of *Active Warehousing*: The shortening of the update interval. This reduces the deviance of the aggregates from the real, transactional data and therefore allows almost real time reporting. It does however increase the load on both, the transactional and the analytical database. The transactional database has to handle additional extracts which cannot, as it is common in traditional warehousing, be scheduled in the downtime of transactional operations but have to be executed concurrently to the transactional load.

Lazy Aggregates

The update interval in Active Warehouses is shorter than in traditional warehouses but still a constant. The deviance between the real data is therefore undetermined because it may be changed arbitrarily by a transaction unless special restrictions are implemented. A possibility to limit this deviance is provided by a technique known as *Lazy Aggregates* [13]. The warehouse update is not triggered after a given interval but when the deviance exceeds a predefined threshold. This assumes that it is significantly faster to calculate the deviance that is induced by a processed transaction than to run the update. Depending on the aggregation function calculating the deviance without calculating the value can be costly or oven impossible (e.g., for holistic functions). In that case this approach fails to yield any benefit.

Fractured Mirrors

Fractured Mirrors [14] is a technique that can be regarded as a special case of both, Active Warehousing and Lazy Aggregates. The update interval and the accepted threshold are set to zero. This means that every modifying transaction is instantly reflected in the analytical database. Ramamurthy et al. [14] introduced and evaluated the concept for a column-store and a row-store operating in parallel. Each reading query is answered by the most appropriate database, each writing query executed on both. Like all other redundant storage schemes this introduces additional load, hardware and administration costs. The fact that updates/inserts may take a different time in each database further increases the programmatic complexity.

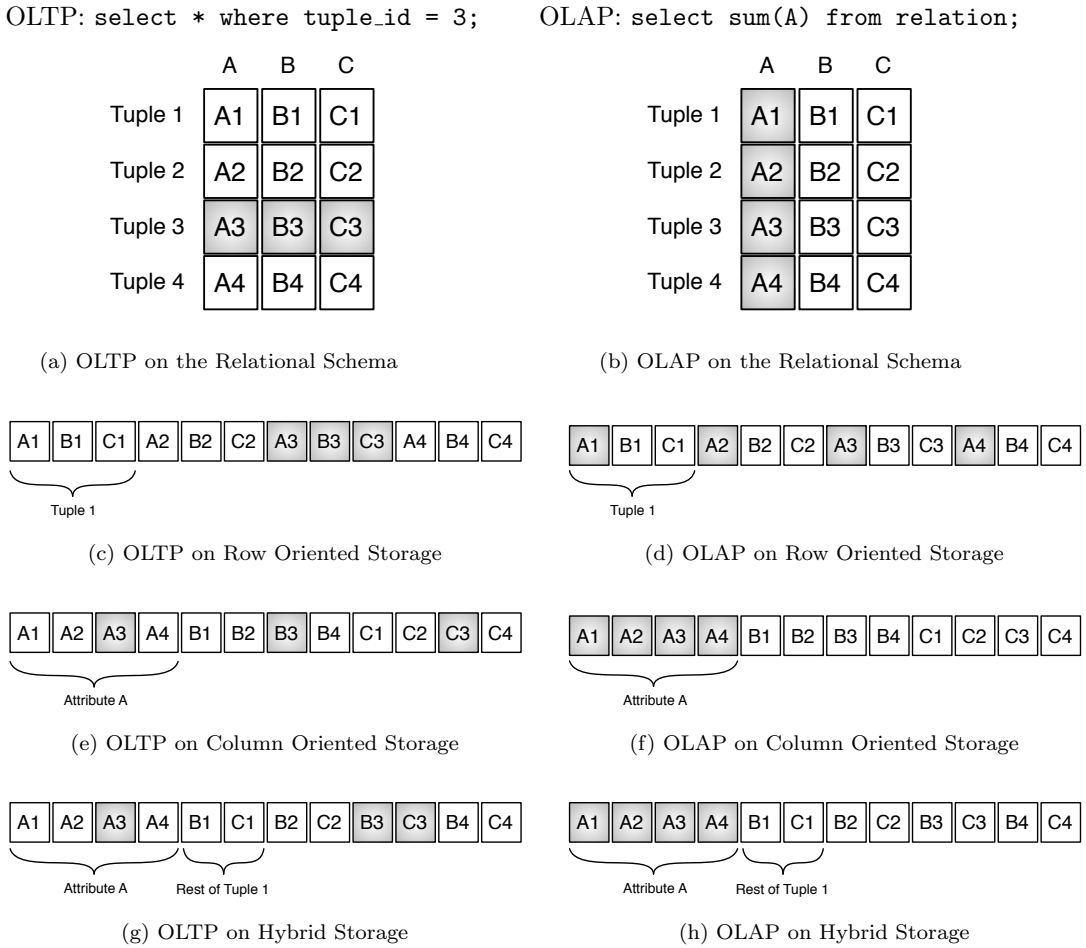
Hybrid DBMS

Both problems, the additional costs as well as the delayed updates, originate in the redundant storage. Therefore both can be solved by eliminating the need for redundant storage. We believe that, although a database may be used for multiple purposes, a single attribute is often used primarily for one purpose. We believe that it is possible to achieve performant real-time (in fact on-the-fly) aggregates without paying the costs for redundant data storage. We believe that the schema[1] can be divided into disjoint partitions that are mainly used for transactional operations and partitions that are also used for analytics. Each partition can be stored in it's most appropriate layout to maximize the data locality for the given workload. The capability of storing data into row- and column-based partitions is the defining feature of what will be called a hybrid database in this thesis. Such a system could be implemented either as a wrapper [15] on top of two existing DBMSs, a row- and a column-store, or as a single DBMS that supports both storage models.

A Motivating Example

To illustrate the potential of hybrid storage consider the example in Figure 1.1. It shows two sample queries, an OLTP and an OLAP query as well as their access pattern on each layout.

[1] Horizontal partitioning, i.e., the storage of the values of one attribute in different layouts is possible but out of scope of this thesis



(a) OLTP on the Relational Schema

(b) OLAP on the Relational Schema

(c) OLTP on Row Oriented Storage

(d) OLAP on Row Oriented Storage

(e) OLTP on Column Oriented Storage

(f) OLAP on Column Oriented Storage

(g) OLTP on Hybrid Storage

(h) OLAP on Hybrid Storage

Figure 1.1: Memory Accesses for different Sample Queries on different Layouts

When evaluated on a row-store, the OLTP query accesses all values from one contiguous area[2]. The OLAP query results in four random accesses to the memory in a row-store.

When evaluated on a column-store, the OLTP query induces three random accesses. The OLAP query can be evaluated by accessing one contiguous block, i.e., one random and multiple sequential accesses.

The last presented storage option is a hybrid layout (note that other hybrid layouts exist). Attribute `A` is stored column-oriented whilst `B` and `C` are stored row-oriented. To evaluate the OLTP query, two random accesses to the memory are needed. This is an increase compared to the row-store but still less than the column-store. For the OLAP query the hybrid store behaves just like the column-store.

Depending on the and the relative costs of random and sequential misses and the number of times each query is executed any of the presented layouts may have the least costs.

A hybrid DBMS would have to decide on the appropriate layout for every single piece of data. This decision is hard and can only be made if the usage pattern of each piece of data is known. This usage pattern is best derived from a Workload, i.e., a set of queries weighted with the frequency at which they are evaluated. Assembling a representative workload for a Database can be done upfront by a domain expert or by tracing the queries in a running instance of the database.

[2]Chapter 2 illustrates that reading data sequentially from a contiguous area of the memory is faster than reading data that is further apart

1.2 Problem Statement

Based on a given workload for a database we want to find the non-redundant, partitioned layout with the minimal overall query costs. To solve this problem it is necessary to solve two subproblems:

Finding an estimation for the query costs on a given layout that is as accurate as possible while still being computable in a reasonable time. Since the query costs are highly dependent of the hardware that runs the DBMS, the model has to take parameters of the hardware into account. The estimation should be based on a generic cost-model that allows an accurate estimation with few parameters.

Finding the partitioned layout with the least estimated costs for a given workload. Since automated schema partitioning has been studied for a long time [16, 17, 18, 19, 20, 21, 22, 23] it seems reasonable not to develop a new solution from scratch but to use an existing approach and adapt it to our needs. This may pose some restrictions on the cost model that have to be identified and met to allow an optimal solution.

The focus of this work are in-memory databases [24, 2, 25, 26, 27, 28, 29, 30] because (a) they allow an easier model of the hardware since parameters that are unique to disk-based storage (number of disks, varying latency, failure rate, ...) can be neglected and (b) they have gained in practical relevance lately due to the decreasing cost and increasing capacity of memory chips. This development makes in-memory databases an interesting topic for research as well as compelling option for practical application.

The cost-model is expected to be generic enough to provide a *very simple* model of disk-induced costs as well. Accurately modeling the specific effects of mechanical disk hardware is not possible with the an interesting challenge for future work.

1.3 Structure of This Work

The rest of this work is structured as follows: Chapter 2 gives the necessary background on in-memory databases and the hardware parameters that determine the performance of data accesses. The respective advantages of row- and column-based storage in an in-memory database as well as their origins will be discussed. In Chapter 3 the (existing) cost-model and our extensions are described. Some existing approaches to vertical partitioning are introduced and extended to support hybrid partitioning. In Chapter 4 we will show how these findings were incorporated in *Spades*, a tool that automatically calculates an (analytically) optimal layout for a given schema, workload and hardware configuration. In Chapter 5 the accuracy of the cost-model is evaluated through a series of simple experiments. The layouting performance of *Spades* in a more complex scenario will also be evaluated. We will conclude and present ideas for future work in Section 6

Chapter 2

Background on Database Storage Performance

DBMS performance traditionally has been limited by the performance of the underlying storage device [31]. The performance of many storage devices has been tuned under the assumption of strong data locality (see Section 2.2.1). Traditional databases, that are either row- or column-oriented, often fail to provide the data locality that is needed for optimal performance [14].

Databases that allow the storage of relational data in either row or column orientation are called *Hybrid Databases* in this thesis. Hybrid Databases allow to increase the locality of stored data with respect to a workload which can in turn improve the data access performance. To maximize the benefit of this technique an arrangement of the data in memory has to be found that maximizes the data locality with respect to the underlying hardware and the workload. This requires some understanding of modern computer hardware and the way it is utilized for storage and processing in DBMSs.

In this chapter, the necessary background on modern computer hardware is provided and existing approaches that aim at increasing data locality for various database applications will be discussed. Since disk-based databases have been the focus of researchers for a long time, we will start with a very brief introduction to disk-based data access performance. Our investigation into main-memory data access performance will be initiated by challenging the assumption of constant-latency random access of the main memory through some simple experiments. The results are analyzed and explained and the determining factors for data access performance are illustrated. Following that, an overview over the caches in modern CPUs will be given. The chapter will be concluded with a description of existing research on data access performance of databases and the problem of optimal relational data storage.

2.1 Disk-Based Data Access Performance

Although not strictly the focus of this thesis, in this section, a short introduction to disk-based data access performance will be given. This allows us to draw parallels between disk-based and in-memory data access and helps to leverage some of the findings of disk-based data access research to the new area of in-memory databases.

The performance of disk-based data accesses are determined by a number of factors. The most important are [32]

(a) the disk rotation speed, which mainly determines the maximum *transfer rate* and

(b) the seek time, which is the time it takes to locate an arbitrary piece of data and move the arm to its position. This is the key factor to the *latency* of the disk. The seek time may vary depending on the distance by which the arm has to be moved. It is therefore usual to specify a minimal, a maximal and an average seek time.

The physical parameters themselves are not as much of interest to us as the effects they induce when accessing data: transfer rate, which is the same as bandwidth, and latency (see Section 2.2.2).

Transfer Rate Benchmarks of the harddisk in our test system, a *Western Digital VelociRaptor WD3000BLFS*, show a maximal transfer rate of 119 MByte/s [1]. This transfer rate can only be achieved for sequentially read data since sequential reads do not require a seek. Linux `fdisk` reports a block size of 512 bytes, which means that a block can be transmitted in 4.1 microseconds. When accessing data this way, about 31,2 million integer values can be read per second.

Latency The average (read) seek time of the WD3000BLFS is specified as 4.7 ms[2]. Adding the time for the transmission of a block, a random access to a value on this disk takes 4.741 ms. When reading 32bit integers that are spread (pseudo-)randomly across the disk[3], every read integers induces a seek and a block access. When accessing data this way, 211 integer values can be read per second. The factor between the random and the sequential access performance is about $1,47e5$.

This factor lead to the conclusion that the only determining factor for disk-based data accesses is the number of induced seeks [27]. While this is certainly oversimplified it gives an impression of the importance of data locality for disk-based DBMS: bandwidth is assumed to be virtually infinite but latency very high.

For main memory, however, the latency is supposed to be a constant, independent of the location of the accessed data. In the following section this assumption will be challenged.

2.2 A Primer on Main Memory Data Access

Before discussing the hardware factors that determine In-Memory DBMS performance specifically it is useful to know some more general properties of transistor-based memory. A good place to start is the assumption of true random access.

2.2.1 The Myth of Random Access

A computer's main memory is accessed by the CPU by supplying an integer address and receiving the value that is stored at this address. In theory the main memory is a Random Access Memory (RAM), i.e., any value is supposed to be read in the same constant time. This sets it apart from non-random-access memory like a disk, CD or tape which may take an undetermined time to access a value [33, ps. 681 - 683].

This theoretical assumption can be challenged with a simple experiment: a program that accesses a constant number of addresses but varies the distance (stride) between them. When plotting the average processing time per value in dependence of the stride we would expect an even graph — the execution time should be the same for every stride because every access of a value should take the same constant time.

Figure 2.1 shows a plot of the results of this simple experiment (see Appendix A.1 for the source code) when executed on our test system, an *IBM BladeCenter HS21 XM* with an Intel Xeon E5450 Processor (3 GHz) and 32 GB RAM. The plot does not show the expected uniform access costs. Instead, it shows that the wider the stride the more time is spent processing a single value up to a stride of 32KByte after which the costs are constant. There are also several points of discontinuity in the curve.

Figure 2.2 shows a plot of the results of a similar experiment (see Appendix A.2 for the source code). In this experiment the stride was kept constant (64 bytes) and the size of the accessed area in memory was varied (the number of accesses was constant). The plot shows virtually constant costs up to six megabytes. After that point the costs are increasing steeply up to six megabytes. After that they are constant again.

These plots show that RAM is indeed not a random access memory and provides motivation for an investigation into the reasons. Especially the points of discontinuity appear interesting. To understand the reasons for the different access costs it is necessary to understand the operating mode of the hardware

[1] available at `http://www.storagereview.com/WD3000BLFS.sr`

[2] Vendor specification is available at `http://www.wdc.com/en/products/products.asp?driveid=494`

[3] This happens e.g. when accessing a tuple in a column-store

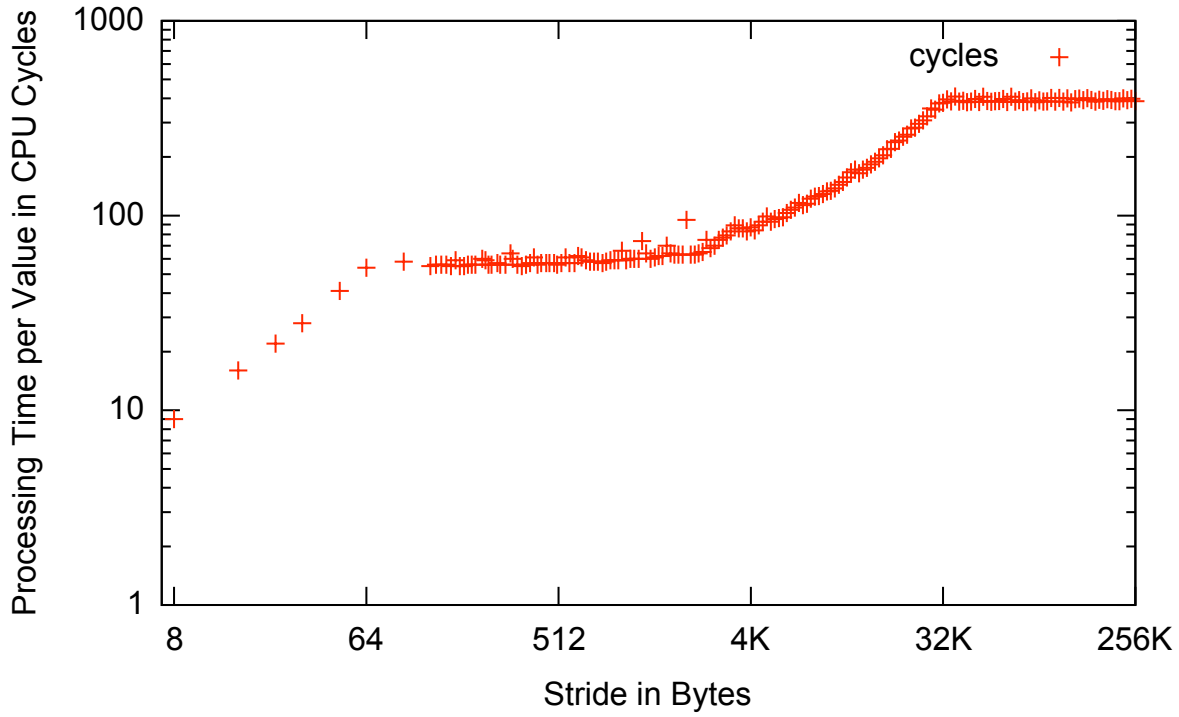


Figure 2.1: Costs of a Data Access with varying stride

components that influence memory access performance. Most of them are tuned under an assumption called *Data Locality*.

Data Locality

A data access pattern that many applications expose is known as *Data Locality* [33]. Data that has a strong semantic relationship, like two attributes of an object or struct, are stored close to each other in memory, i.e., the difference between their addresses is small, and often accessed together. This is called *Spatial Data Locality*. Data locality is also exposed in the dimension of time (*Temporal Data Locality*): data with a strong semantic relationship is often accessed over a relatively short time. A special case of temporal data locality is the repetitive access of the same piece of data over a relatively short time. Hardware vendors have recognized this pattern and adopted their hardware to it. Data accesses according to this assumption are performed faster than arbitrary data accesses. Most commonly this is done by storing data in blocks, areas of the memory with a common size and a predefined position. Access to multiple values in one block is usually much faster than access to values from multiple blocks. It is therefore sensible to develop applications according to this pattern to take advantage of the hardware optimizations that assume data locality.

When talking about data locality the concept of *Blocked Data Locality* will sometimes be used in this thesis.

Blocked Data Locality means that it is not necessary to store data items that are accessed together as close as possible but merely within a block of the respective memory layer to achieve optimal performance.

Absolute Data Locality means data locality where data items that are accessed together are stored as close as possible in the memory.

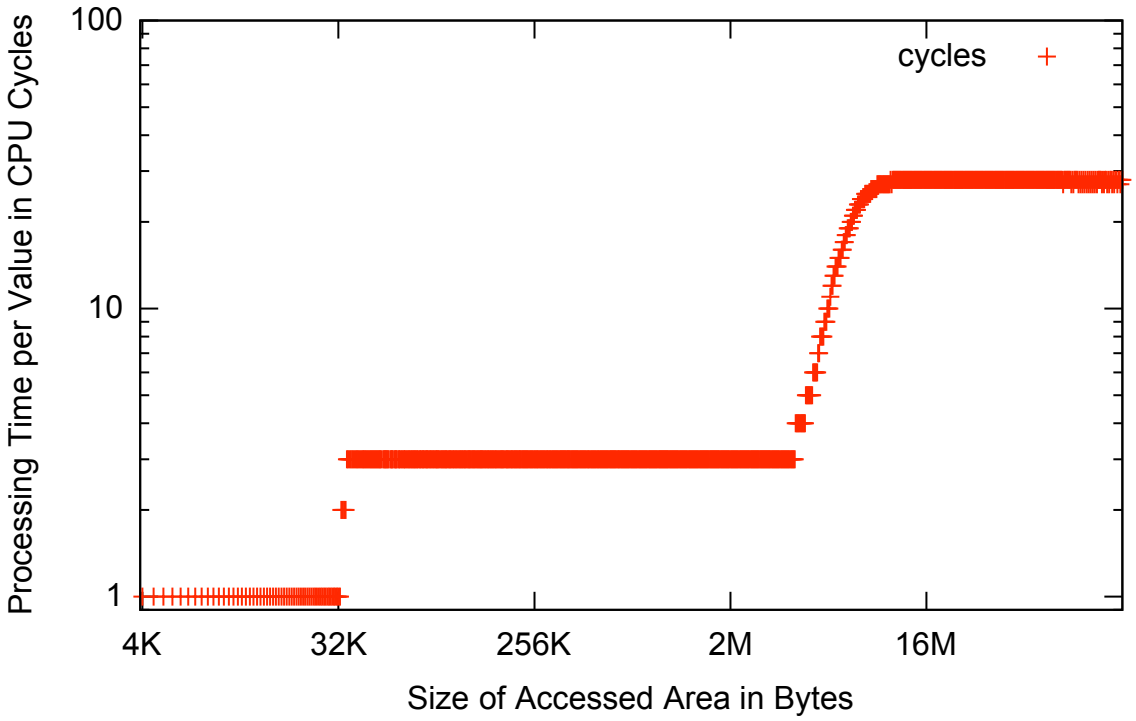


Figure 2.2: Costs of a Data Access to an Area of Varying Size

2.2.2 Determining Factors for Data Access Performance

The memory structure between RAM and a modern CPU usually includes the CPU registers, at least two levels of CPU caches and the RAM itself which in turn is organized into blocks. Figure 2.3 shows a diagram of the hardware components of our test system that have an impact on the memory data access performance. A detailed description of the impact each component has will be given in Section 2.2.3. For now, it is enough to know that data that is about to be processed is transmitted from the RAM towards the CPU Cores through each of the memory layers. Every layer provides a cache for the underlying layer which decreases the latency for repetitive accesses to a piece of data. A request to a piece of data that is not currently stored in a cache is called a miss [33]. A full miss, i.e., a needed piece of data that is only present in the RAM, results in an access to the RAM and the transmission through all layers of memory. The time this takes is determined by two factors: the (minimal) bandwidth and the latency of the RAM [33, p. 393].

Bandwidth

The (digital) bandwidth of a data transmission channel is the amount of data that can be transmitted through the channel in a given time [34]. It is usually measured in bytes or bits per second. When processing data from the RAM it has to be transmitted to the processing core through a number of channels:

- from the RAM through the Front Side Bus (FSB) to the Level 2 Cache
- from the Level 2 Cache through the CPU-Internal Bus to the Level 1 Cache
- and from the Level 1 Cache through the Core-Bus to the Registers of the CPU Core.

The bandwidth of the channel from RAM to CPU Core is the minimal bandwidth of any of the channels. Some channels like the Front Side Bus or the CPU-Internal Bus are shared between multiple processing

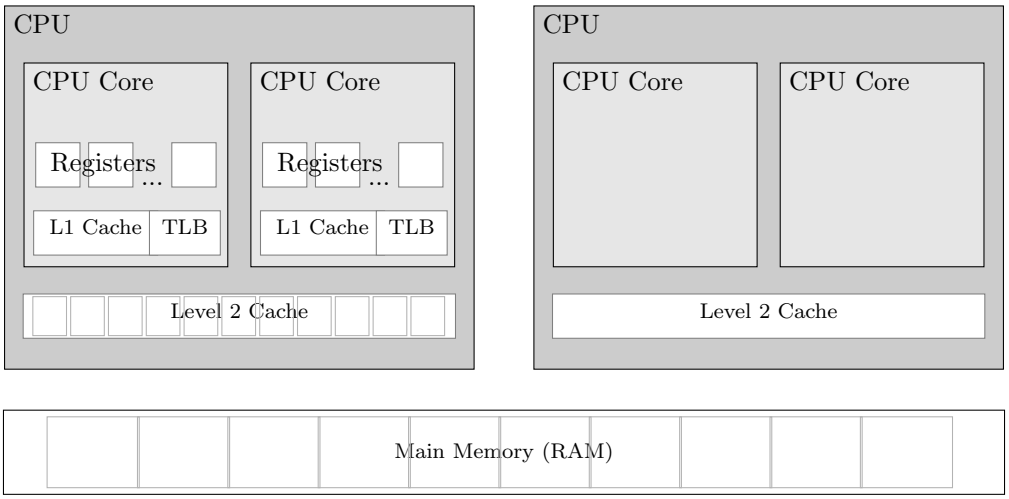


Figure 2.3: Schema of the Relevant Hardware for In-Memory DBMS

units (Cores) which may decrease the effective bandwidth of the channel.

Latency

The Latency of a storage device is the time between the request of a piece of data and the beginning of its transmission. The latency of transistor-based memory is comparable to the seek time of a harddisk (see Section 2.1) in its effect. The cause however, is very different: instead of the mechanical inertia of the disk arm it is the time to decode a memory address and connect the transistors that contain the requested piece of data to the bus [32]. For caching memories the latency also originates from the time it takes to determine if, and if so where in the memory, a given block is cached (see Section 2.2.3).

The Relation between Latency and Capacity of Memory

Even though latency has been a known problem for some time, effective solutions to the problem are still lacking [25]. The most common approach to decreasing the latency is to simply increase the clock rate of the transistors. This is limited by physical constraints. A major factor is the high address decoding and transmission effort for hierarchical high capacity memory [33, p. 448 - 455] [35, 32]. Memory is addressed using contiguous integer values by programs but is physically accessed by activating or deactivating the charge on pins of the memory chip. The translation of the integer address to the tuple of activated pins is called address-decoding and takes a time that scales linear with the width of an address[4] [33, p. 448 - 455]. Since larger memories need wider addresses, transmitting and decoding these addresses becomes increasingly time-consuming. This makes it currently impossible (let alone cost effective) to build a large memory, like the RAM, with a low latency. This also means that low latency memory like the Level 1 Cache has a relatively low capacity.

Block Access Time

Applying the principle of data locality to the problem of memory latency, a very simple solution is reasonable: data is always accessed in blocks. This does not diminish the latency but only induces it per block instead of per data-word (the width of the system Bus, most commonly 64 bits). Consistency requirements in each memory layer make it necessary that a full block is transmitted before a new block can be accessed. Thus latency, transmission bandwidth and block size can be combined into the *Hit Time* [33] or *Block Access Time* (BAT), the time it takes to activate and transmit a block from a memory layer to the next.

[4] for a fixed number of transistors in the decoder

Knowing the latency, access-bandwidth and block size of each memory layer the BAT can be calculated using Equation 2.1.

$$BAT = latency + \frac{blocksize}{bandwidth} \tag{2.1}$$

In the following the BAT will be used as the primary metric to determine the cost of a memory access. This is valid since every access to a memory layer takes time for activation and transmission [32].

2.2.3 Caches in Current CPUs

To speed up data access on high capacity main memory, most current CPUs include one or many low-latency/low-capacity caches. Data accesses that fulfill the assumption of data locality can greatly benefit from these caches. In this section their most relevant properties will be discussed.

Cache lines

Caches usually do not cache single values but rather blocks an equal number of values. These are called *Cache Lines*. Cache Lines are the atomic storage unit of a cache. A cache can not contain a strict subsets of a cache line. If one value of a cache line is modified, the whole cache line is written back to the memory [33]. Cache lines start at predefined positions that do not overlap and span the whole cacheable memory.

Slots

The capacity of the cache is defined by the size of a cache line and the number of available storage slots. Since the cache is generally much smaller than the cached memory, the mapping of a block of addresses to their cache line is not trivial. One slot can hold one of many addresses and an address could potentially be cached in one of many slots. The mapping of memory addresses to the set of possible slots is defined by the *associativity* of the cache.

The Relation between Latency and Capacity of a Cache

The relation between the latency and the capacity of a memory has been discussed in Section 2.2.2. For caching memories, the latency is increased even further with their capacity because a fully associative cache could store a cache line in any location [33]. When trying to locate a cache line all location have to be checked [33]. The impact on the latency is obvious. To decrease the latency it is common to allow only a few locations for a given memory address. In such a *Set Associative* Cache, only those locations have to be checked. This may however result in early evictions which will be discussed Section 3.1.4.

Evictions

Due to it's relatively small capacity, the cache is filled fast when operating on large datasets. It is therefore necessary to remove a cache line before a new one can be loaded into the cache. Which cache line is evicted is determined by the *Eviction Strategy*. For the rest of this thesis we will assume a *Least Recently Used (LRU)* eviction strategy, which means that the cache line which has not been accessed the longest is evicted.

Address Translation

Another layer of blocks that has to be considered is introduced by the caching of the mapping from virtual to physical addresses. The emerging of multi-processing made it necessary to protect the address space of one process from access by another process. This protection is provided by the operating system through the concept of *Virtual Memory* [33]. Every process that requests memory from the operating system is supplied with an area of memory that is marked as belonging to this process. This happens transparently to the process: it can access its private virtual memory as a contiguous space using integer

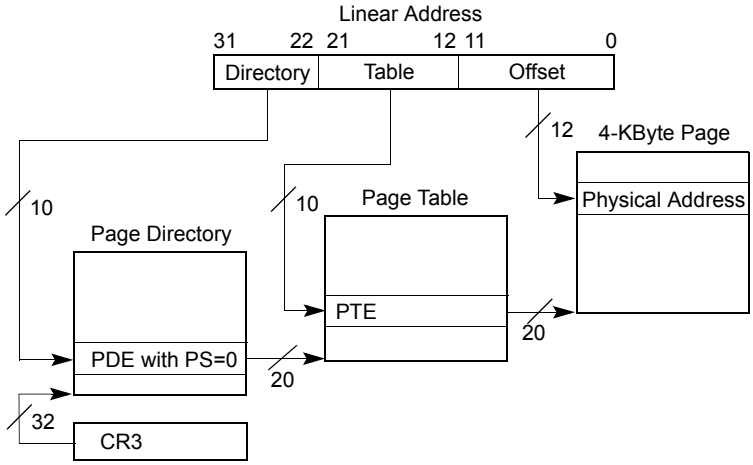


Figure 2.4: Address Translation of a 32 Bit Address in the Intel Core Architecture (taken from [1])

addresses. Since the address spaces of multiple programs may be interleaved the virtual address has to be mapped to the physical address where the data is stored. This mapping is based on a number of parameters (including the process id, the virtual address and parameters to manage shared address spaces) and takes time. To speed up address translation the result is cached in the *Translation Lookaside Buffer (TLB)*. In general, the TLB caches only a pointer to the first address of a block of memory addresses. The size of such a block is dependent of the system. Most common are 4 KByte with an option of switching to large pages of 2 MByte each [1]. These blocks are called (virtual) memory pages and introduces another blocking of memory.

If a TLB miss occurs, the virtual address has to be translated into a physical address. Figure 2.4 illustrates the address translation of a 32 Bit address in the Intel Core Architecture. A miss in the TLB results in a lookup in the *Page Directory* using the first 10 bits of the address. The entry in the Page Directory designates the address of a *Page Table* which holds $2^10 = 1024$ *Page Table Entries (PTEs)*. The PTE is a pointer to the starting address of the physical memory page. The next 10 bits of the address are used to select the PTE from the Page Table. This pointer is cached in the TLB. The Page Directory as well as the Page Table are stored in the regular main memory. Therefore, each of the lookups may induce a Level 2 Cache miss.

Writing to the Memory Through Caches

Reading a cache line from the memory results in a single fetched cache line. *Writing* the memory, however, is more complicated. Depending on the write strategy (*write through* or *write back* [33]), data is written to the underlying memory either immediately (write through) or delayed until the modified cache line is evicted (write back). All modern Intel [1] as well as AMD [36, page 170]CPUs allow picking the write strategy per address (block). Our experiments (see Section 5.2) indicate that heap variables are always stored in a write through block (by the used compiler). This makes modifications that are made in one thread immediately available to all other threads. Therefore, every modification would block the memory and induces costs like a cache miss. If multiple modifications are made within "a small window of time" [1, Vol. 3, page 11-12] they are buffered in a *Write Combine* buffer and written consecutively. This optimization makes a write through behave like a read. We will therefore treat it just like a read.

Stack variables are stored in a write back block (by our compiler). The data is written to the memory once the modified (a.k.a. dirty) cache line is evicted. Integrity constrains make it necessary to fetch the (unmodified) rest of the cache line on a write [1, Vol. 3, page 11-10]. This induces a cache miss on the first modification of a cache line in addition to the miss induced by the writing. Unless a cache line is accessed is modified many times the additional miss decreases application performance.

Due to their potential size, intermediate results in an in-memory database are usually allocated on

the heap and thus in a write through block. We will therefore assume a write through cache for the rest of this thesis. Investigating in the performance implications of a different writing strategy is left for future work.

Prefetching in the Level 2 Cache

To reduce the penalty when retrieving a cache line from the underlying memory layer some caches try to anticipate the line that will be accessed next and start fetching it before it is requested [33, 37]. This is called *Prefetching* and will be discussed in this section.

Processing without Prefetching Values that are not present in any of the CPU caches are requested from main memory before being processed. After the activation latency of the memory the transmission of data begins. This usually happens in the so called *Burst Mode*: not only the requested data word is transmitted but a whole block of words without the CPU explicitly requesting them. The size of a burst is usually the same as the size of a cache line. This allows efficient filling of a cache line without additional overhead for addressing all words individually. Assuming data locality this improves the performance.

Figure 2.5 shows a sequence diagram of the caches and the CPU processing one value from a cache line. First, the CPU requests the value of address 1 which is not present in the cache. The cache requests the address from the memory, which triggers a burst of the block that contains the address. The memory transmits all values from the block to the cache, the CPU stalls. When the cache line is transmitted completely, the CPU starts processing the requested value and the memory/Bus is idle. When the CPU finished processing the requested values, it requests the next piece of relevant data. In the example, this induces another cache miss which triggers another burst of values from the memory. As illustrated in Figure 2.5, the CPU and the memory spend time idle while the other is working.

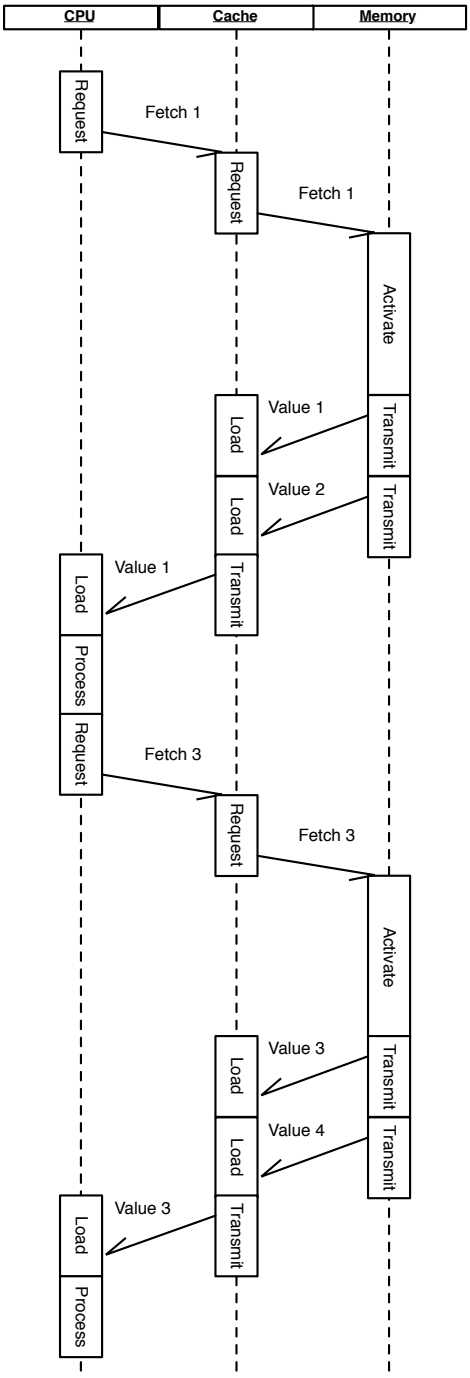


Figure 2.5: Activities on the Different Memory Layers when Processing Values without Prefetching

Prefetching Strategies Since the Memory Bus is the main limiting factor for data transmission performance [25], keeping the Bus busy is crucial for the overall performance. This can be done by transmitting data that has *"not yet"* been requested by the CPU. Applying the principle of data locality it is reasonable to transmit data that is located near

the currently accessed. Whether any prefetching is triggered and if so which cache line will be prefetched is determined by the prefetching strategy. The Intel® Core™ Microarchitecture [1] e.g., defines two different prefetching strategies for the Second Level Cache [38] :

- *The Data Prefetch Logic (DPL)*(the default) is a sophisticated prefetcher that attempts to recognize strides (consecutive accesses to addresses with a constant distance) and anticipate the next fetched cache line based on the recognized stride. The second prefetching strategy is the
- *The L2 Streaming Prefetcher* that simply fetches the next adjacent cache line.

The prefetching strategy can be selected or completely disabled at runtime using *Machine Specific Registers*.

The correct prediction of the next requested cache line and it's transmission will be called *correct prefetching*. An incorrect prediction and the transmission of a cache line that will not be requested is called *incorrect prefetching*. The effect of either will be discussed in the following.

The Benefit of Correct Prefetching Correct prefetching can improve the bandwidth utilization by keeping Bus and memory busy and thus improve the overall performance. Figure 2.6 shows the beneficial effect of correct prefetching. Again, the CPU starts by requesting value 1 and stalls while the cache line is transmitted. When the last piece of data has been transmitted to the cache, the CPU processes it. At the same time, the prefetching unit of the cache fetches the next cache line (the one containing value 3 and 4). Since the prefetching was correct, the CPU requests value 3 when it has finished processing value 1. The memory has already started transmitting it and the cache has a head start. In the best case the complete cache line has already been loaded and the cache can supply the CPU with the values right away. This shortens the stalling periods keeping the Bus and the CPU busy which in turn increases the data throughput of an application.

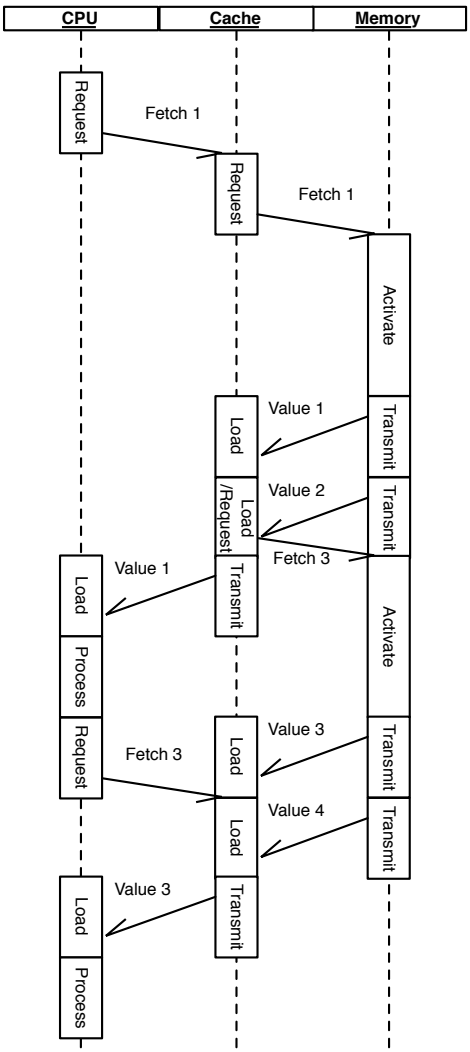


Figure 2.6: The Effect of Correct Prefetching

The Effect of Incorrect Prefetching Whilst correct prefetching can improve performance, incorrect prefetching can lead to a decrease in application performance. This is due to the fact that any fetch, regular or prefetch, blocks the memory and the Bus. Since a memory burst cannot be interrupted once it is started, Bus and memory are blocked until the end of the burst. In case of an incorrectly fetched cache line this prevents the transmission of the correct cache line. Figure 2.7 illustrates this effect. The CPU requests the value 1 and processes it. When the transmission of the first cache line is complete, the prefetching unit triggers the (incorrect) prefetching of value 5 and 6.

When the CPU is done processing value 1, it requests value 3 but the burst of value 5 and 6 has already been triggered and has to be completed. The cache has to wait until the transmission is completed before value 3 can be requested from the memory. Prefetching value 5 and 6 blocked the Bus and prolonged the stalling of the CPU.

In addition to blocking the bus, incorrectly prefetching a cache line evicts a cache line from the cache. This can effectively double the penalty because the evicted line may be requested again later on, which induces an additional cache miss. In our experiments (see Section 5.2) the additional evictions had only minor influence on the overall performance and were, therefore, is not considered in the model.

The Performance Impact of the Prefetching Strategy When comparing Figures 2.6 and 2.7 one may notice that the penalty of an incorrectly prefetched cache line may be very high in comparison to the benefit for a correctly prefetched one. A good and above all cautious prefetching logic is therefore crucial to preserve the positive effects of prefetching. The *Data Prefetch Logic* is such a cautious strategy since it only triggers prefetching on detection of a constant stride. For some applications this may be too cautious and may not trigger any prefetches even though the application may benefit from them (see Section 3.1.4). In these cases it may be beneficial to change the prefetching strategy to one that better suits one's needs (HYRISE allows to change the prefetching strategy per relational operator).

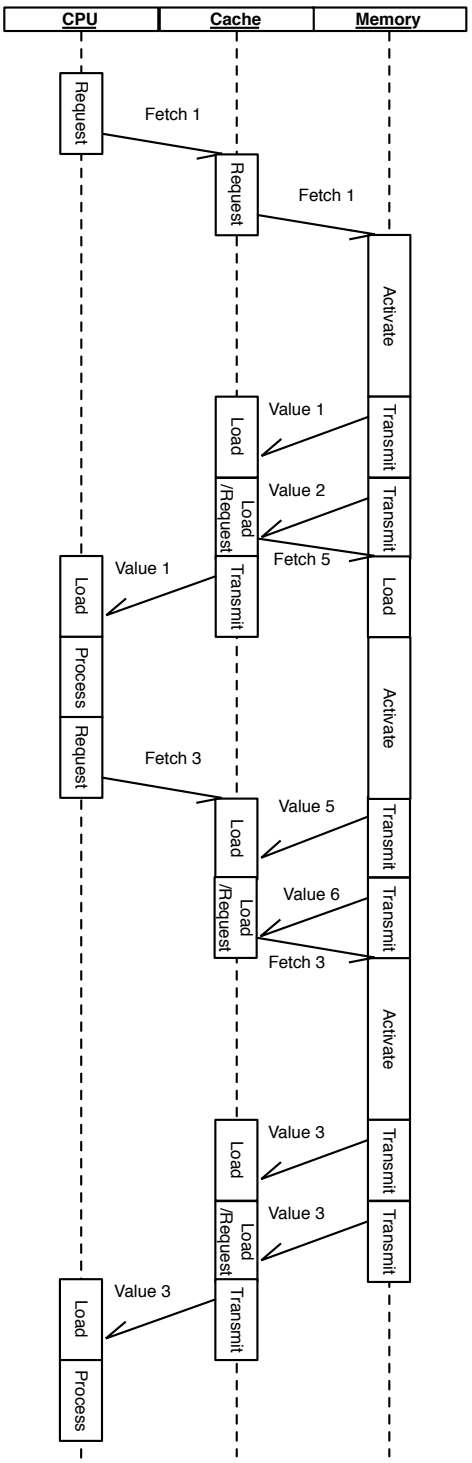


Figure 2.7: The Effect of Incorrect Prefetching

2.2.4 Blocks in the Memory Modules

In addition to the blocking that is introduced by the caches, the memory modules themselves are blocked too. The memory chips are organized in a matrix [32]. To address a cell in the matrix a row and a column address have to be provided. The memory modules do, however, keep a row active after a value has been transmitted. When accessing a value from the same row it is therefore unnecessary to readdress a row. The mapping of the integer addresses to a row and a column address is performed by the memory controller. It receives the address and determines the row at which the requested address is located. If the row differs from the last accessed row,

the controller sends a command to the memory module to change the selected row. This command is called the *Row Address Strobe (RAS)* Signal. After that, the column is calculated and a command is issued to access the value. This is called the *Column Address Strobe (CAS)* Signal.

2.3 Data Access Performance of Different Database Management Systems

Database Performance on modern CPUs is very much I/O-Bound [25]. The query throughput is not limited by the processing speed but by the speed at which the processor can be supplied with the necessary data. It is therefore crucial to make good use of the available bandwidth and avoid latency stalling.

As seen in Section 2.2, hardware vendors often tune hardware to improve the performance for applications that expose a strong data locality. It is relatively easy to develop applications according to this assumption if the usage pattern of the data is known in advance. For a DBMS that is not aware of the it's usage pattern in advance one design does not fit all. The application of a database is greatly influencing the usage pattern of the held data and thus the optimal arrangement of data in memory.

In this section the existing approaches to improve the data locality for a known workload in managed databases will be presented.

2.3.1 In-Memory Database Management Systems

While the disk used to be the only storage option, the decreasing (physical) size and costs of transistors made an increase in memory capacity possible. This made it feasible to use the RAM as the primary storage and benefit from its superior performance in terms of bandwidth as well as latency.

Many large DBMS vendors recognized the potential of in-memory DBMSs and already ship or at least announced in-memory data management solutions. These are either mere caches for disk-based DBMS like Oracles TimesTen [26] or standalone solutions using the memory as the primary storage and the disk only as a backup like IBMs SolidDB [39].

Since the bandwidth as well as the latency of RAM are outperforming disks by some orders of magnitude a performance benefit in this dimension is expected. When carefully designing an in-memory DBMS this advantage in data access speed can be leveraged to build high performance DBMSs. As shown in Section 2.2 the determining factors for in-memory data access performance are similar to those determining disk-based performance.

Existing research on data storage strategies for in-memory databases will be discussed in this section.

First In-Memory DBMS Implementations

With the decrease of transistor size, RAM capacities grew to a size that allowed storage of a reasonably sized database. In the 1980's, research on in-memory DBMS began [40].

At that time, blocked data transmission, as described in Section 2.2.3, was not as common as it is today. The burst mode for RAM-modules was patented in 1985 [41] and standardized e.g. in the Burst EDO RAM in 1996 [42]. The first Intel CPU with an integrated cache was the i486DX that was introduced in 1989 [32]. In 1992 Garcia-Molina and Salem [27] described the main memory as "not block-oriented" — the RAM was still considered a random access memory. Under this assumption it was reasonable to disregard existing findings on data locality that originated from disk-based DBMS and base the storage layout purely on concerns of the implementation. An implementation that followed this assumption is MM-DBMS [28]. As shown in Section 2.2.1 the assumption of true random access does no longer hold true: In-Memory Databases now follow rules that are very similar to those for disk-based databases and therefore many of the existing optimizations can be reused.

Row-Based DBMS

Traditionally, relational structures have been mapped to the one-dimensional memory strictly record-wise. All attributes are written to a consecutive area in memory, one slot followed by the next [43] [5]. In terms of data locality this means that there is a strong locality between the attributes of a tuple. For a transactional application this seems the most suitable layout because transactional queries usually operate on few tuples and often access many attributes. The strongest representative of transactional queries is the INSERT-statement that accesses *many or all* attributes of a *single* tuple.

On the other hand, storing data in this layout means that the values of the same attribute but of different tuples are at least separated by the length of one tuple. The consequence is a very low degree of data locality for the values of an attribute. Analytical queries, that usually access values of few attributes but many tuples, are therefore executed on a suboptimal storage layout.

This effect has been recognized by database administrators and lead to research regarding decomposition of relations. Since attribute values are separated by at least a tuple length the data locality between them can be increased by reducing the length of a tuple. The most extreme case of this technique is called Decomposition Storage Model (DSM) [20]. When stored in DSM every attribute of a logical relation is stored in one physical relation together with the id of the tuple. This reduces the distance between two values of an attribute to exactly the size of the id and thus increases its data locality. DSM has a major disadvantage: all queries that access more than one attribute have to be rewritten to reconstruct the logical tuples from the physical relations using a join on the id. Since the selectivity is expected to be low for such OLTP queries an indexed join is most appropriate for tuple reconstruction. This does however result in additional costs for write intensive workloads because every insert of an n-tuple triggers n inserts into the indices.

If supported by the DBMS implementation it is possible to remove the explicit id and the needed index from the attribute's relation and use an implicit id that is calculated from the memory address (e.g., $id = address\,(tuple) - offset$). This greatly simplifies the tuple reconstruction and is therefore a very sensible optimization [6]. A database that stores all relations in DSM and does the reconstruction transparently is called column-oriented [3].

Column-oriented DBMS

Column-based DBMS [6, 7, 44, 3] store all data in single-value columns: the values of a given attribute of all tuples are stored in a (practically[6]) contiguous area in memory. As an alternative, several strategies have been proposed to allow column-based storage on the level of storage pages instead of the level of relations [45, 46].

Both approaches increase data locality between values of an attribute and are therefore suited for analytical applications. For transactional queries however column-stores face the same problem as DSM on row-stores: logical tuples have to be reconstructed from the physically stored relations.

Tuple Reconstruction in Columnstores A drawback of a column oriented data storage is that a single tuple is spread over as many locations as it has attributes. When a tuple is requested, the DBMS has to reconstruct it from these locations. n requested attributes result in n (pseudo-)random accesses to the memory. Unless many tuples are requested and the values for their attributes are located on a single memory block this also results in n block accesses per tuple. If the value for an attribute only occupies a fraction of the block, transmitting the rest wastes memory bandwidth.

Late Materialization The high number of memory accesses for the reconstruction of requested tuples has been recognized and tackled through a technique called *Late Materialization* [7]: the requested tuples are passed from relational operator to relational operator (see Section 3.1.1) not in their actual representation but as an integer id. When the values of an attribute are needed by an operator they are read from the stored relation using the tuple id. If the number of tuples decreases during the evaluation

[5] not all slots have to be filled at all times

[6] Even though suboptimal memory management will sometimes split the area in memory we assume that the number of such splits is very small in comparison to the total number of tuples and neglect it's effect.

of the operator tree the number of cache misses that is induced by tuple reconstruction is decreased. If an attribute is used by more than one operator, however, it is read from the stored relation multiple times. Depending on the selectivity of the operators this can increase the number of cache misses. In this case an *Early Materialization* is suited best. Abadi et al. [7] have shown that picking the most appropriate materialization strategy is not trivial.

Indices A common misconception about column-stores is that they do not need indices to efficiently locate a stored tuple [6]. Indeed, the costs of a single attribute scan in a column-store are lower than the costs of the same scan in a row-store. The decreased costs are due to the reduced number of memory blocks that have to be read. Column-oriented storage will, however, only decrease the costs by a constant factor. The complexity of a scan is still in $O(n)$ while an index lookup is in $O(\log(n))$. Thus an index that speeds up a row-oriented also speeds up a column-oriented database in the same manner. The problem of automatic index selection [47, 48] is orthogonal to the problem of optimal cache conscious storage and is considered out of scope of this thesis.

Existing Hybrid DBMSs

Similar to column-oriented storage, hybrid storage can be implemented on the layer of logical relations or on the layer of data storage pages. EaseDB [24] is an implementation of the first, Data Morphing [17] of the second approach.

The capability of hybrid storage introduces new options for database optimizations but also makes the process of optimization more complex. The layout with maximal data locality for a given workload depends on the usage pattern of every piece of the stored data. The optimal layout can therefore only be found if the usage pattern is known in advance. Database administrators can, based on their experience, extrapolate the usage pattern from informal application requirements. EaseDB, e.g., relies on the manual definition of the partitioned, oriented schema by an administrator. This is, however, expensive and possibly error-prone. It is therefore desirable to automate this process.

Data Morphing adapts the schema at runtime using a very simple cost model. The Data Morphing technique has two important drawbacks:

1. the cost model only considers a single layer of blocked caches without further optimizations (like prefetching, parallel address translation, ...) and
2. the running time of the layout algorithm scales exponentially with the number of attributes of a relation which makes it unfit for wide tables.

To improve the automatic optimization a model is needed that takes account of the workload and specific parameters of the hardware.

HYRISE A prototype of a hybrid DBMS is currently developed at the *Hasso Plattner-Institut*. It is called *HYRISE* [49] and, as this is written, still in a very early stage of development. So far it consists of a hybrid storage layer and implementations of the most important relational operators on top of that storage layer. The operators allow early as well as late materialization (see Section 2.3.1).

As we have seen in this chapter there are a number of hardware parameters that have to be taken into account. Due to the blocked storage of data it is not necessary to achieve absolute data locality but merely blocked data locality for optimal performance. Since there may be many layers of blocking with different Block Access Times and block sizes a detailed model of the hardware is needed to find an optimal layout. Such a model will be discussed in the next section.

Chapter 3

Data Layouting based on Estimated Query Costs

As illustrated in the last chapter, the performance of a DBMS largely depends on the data access performance. Since data is always accessed and cached in blocks, the data access costs can be measured in the number of cache misses that are induced in each of the memory layers. Based on this finding, a model can be developed to estimate the execution costs of a given query by estimating the number of cache misses. Hardware parameters and the storage layout have to be taken into account for this estimation. Such a model is described in Section 3.1. How to use this model to automatically find an (analytically) optimal storage layout for a given workload will be discussed in section 3.2.

3.1 Query Cost Estimation

To estimate the costs of a query, it is not only necessary to know the query itself, but also how it is evaluated by the system. Since most DBMSs base their query execution on relational algebra, this section will be started with a very short recapitulation of relational algebra and it's use in DBMSs. This will be followed by an overview of existing research on query cost estimation. In section 3.1.3, the model that was used in this thesis will be described followed by a description of the extensions that were made. How to model a query processor is illustrated in section 3.1.5.

3.1.1 Relational Algebra

Even though most DBMSs use a higher level querying language like SQL for their external interface, most of them rely on relational algebra [43] (or a dialect of it) to represent a query internally. When a query is entered into the system it is first compiled to a relational algebra tree which is then optimized and executed. Figure 3.1 shows an example of such a relational operator tree. For a description of the used operators the reader is referred to [43].

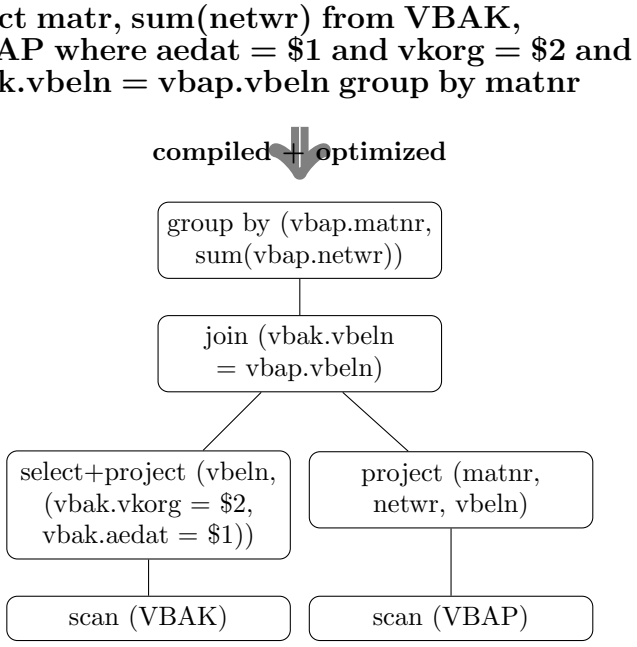


Figure 3.1: An Example of a Relational Operator-Tree

The tree is then processed bottom up. Every operator is executed and produces the input for the next. Data flows from the bottom to the top where it is returned as the result of the query. The operator tree is, thus, a high level description of how the query is evaluated. To reflect this way of evaluating queries it is reasonable to start cost estimation the same way: by compiling SQL to relational algebra. Implementing a simple SQL to relational algebra compiler is fairly straight forward and well documented [43]. Section 4.2 describes how the compiler component of *Spades* has been implemented.

Insert Queries in Relational Algebra Relational algebra is an algebra to query relations, not to modify them. It is not possible to express a modifying query with the relational operators [43]. Therefore, DBMS implementations have to find another way to represent updates/inserts. In terms of memory access, however, insert/update queries do not differ from accesses to a single tuple: both have to locate a memory region of one tuple (or an empty slot for an insert) and access it [1]. We, therefore, represent an insert query as a selection on the tuple id. As explained in Section 2.2.3, writing to memory through the cache generally induces no different costs than reading it.

Implementation of Relational Operators The functionality of an operator is usually implemented in the native language of the DBMS. The description of the implementation in a form that can be used to estimate the execution costs is difficult. One way to describe the implementation are *access patterns*, a concept that is part of the generic cost model [2] that is used in this thesis. In the next section, we will give an overview of alternative models, followed by a detailed description of the generic cost model as defined by Manegold et al.

3.1.2 Query Cost Estimation

Estimating the costs of a query is necessary for query as well as layout optimization. It is generally desirable to estimate the costs in a metric that has a total order (e.g., a simple integer value). This allows to compare two values in the metric and determine which one is "better". To calculate this value, the model may use any number of intermediate metrics. As explained in Section 2.1, e.g., disk based DBMS performance is often measured in the number of induced seeks and the number of read bytes. The costs are derived from that. Simple models like the one used in [43, pgs. 441ff.] calculate the costs from one metric, e.g., the page I/O-operations.

To find the optimal query plan, a query optimizer needs to compare different plans for a single query. Consequently, the costs have to be derived from the operator tree. A layout generator, since it is only interested in the relative costs of queries to each other, may use a cost model that estimates the costs directly from the query. We will discuss both approaches in the following.

Estimating Costs directly from the Query

Data Morphing [17] is an approach to the layouting problem that estimates the costs directly from the query. It relies on an input that specifies the percentage of the values that are accessed of every attribute. From that, the number of induced cache misses per tuple is estimated using a simple formula. This is done under a number of assumptions:

- the values of all attributes are accessed in a uniform and random fashion,
- a read cache line is not removed from the processor cache before all the values in that cache line have been processed and
- all operations are execute on an empty cache

While these assumptions may hold for simple queries that can be evaluated in a single scan of attributes, it fails for complex queries that involve intermediate results or repetitive accesses to values (joins, group-bys, late materializing operators, ...).

[1] overhead for ACID-Properties is neglected

Estimating Costs from the Operator Tree

To take intermediate results and repetitive accesses to values into account, a more accurate model of the evaluation of the query by the DBMS is needed. Since the relational operator tree is such a model, it can be used for a more accurate cost estimation. The estimation of query costs from the operator tree is similar to the evaluation of the tree: The costs of each executed operator are estimated (bottom up) and the overall costs of the query derived from that (usually by simply summing the costs of the operators [43]).

Estimating Operator Input and Output Size To estimate the costs of an operator it is necessary to know the number of tuples it has to process. Most operator-based cost models assume "a perfect oracle" [2] to estimate the number of input and output tuples.

This estimation is not trivial because it is influenced not only by the number of tuples in each relation, but also by the selectivity of the predicates that are used in the query. Substantial research exists on the estimation of predicate selectivity [50, 51, 17]. It is largely based on histograms, that represent the distribution of the values of an attribute. In this thesis we will assume that the values are distributed randomly and equally. This eliminates the need for histograms and reduces the needed statistical information to the number of unique values (cardinality) of each attribute. Incorporating more sophisticated selectivity estimation should be straight forward.

Disk-Based Cost Models As shown in Section 2.1, disk based data access costs do not in principle differ from main memory data access costs. It is, therefore, reasonable to investigate into disk based cost models as well. Simple models [43, pages 439ff] are solely based on the number of disk operations, i.e., the number of accessed blocks. They do not differentiate random and sequential access. Some (very simple) models [52, 53] only consider the number of accessed items without considering that two accesses might happen on the same block. Some models [27] only consider random misses (seeks), since they are much more costly than sequential misses (see Section 2.1). All of them are, too simple to be applied to our case. An appropriate model for disk-based data access would be based on latency and bandwidth [2]. Models for main memory access cost are best based on random and sequential misses [2].

Main-Memory Cost Models Listgarten and Neimat [54] differentiate Main-Memory Cost models into three categories: application-based, engine based and hardware-based.

Application based cost models estimate costs based on the limiting factor of each executed operator. Rules how to find the limiting factor are usually defined manually. E.g., join-performance may be limited by the memory access speed if the relations are large in relation to the available cache or limited by the processing speed if the joined relations fit into the cache. This makes application based cost models very unattractive because they are specific to the hardware and the implementation of the DBMS. Such a model is, e.g., used in [55].

Engine-based cost models are based around executed operations. Operations in this context are not hardware operations, like requesting an address or adding two values, but operations of the execution engine, like the comparison of two tuples or the output of a tuple. Such a model is introduced by Listgarten and Neimat in the same work [54]. Engine-based models are more generic than application-based models but still do not take parameters of the hardware into account.

The last category are hardware-based cost models. Execution costs are measured in the number and type of hardware operations. Since database performance is mainly determined by data access costs it is reasonable to only take data access operations into account. Such models are widely used [56, 2, 17] because they provide very generic models and good prediction performance. The generic cost model is the most advanced of which, since it allows the estimation of the different kinds of costs. It was therefore used as the model for out data layouting algorithms. One may note that hardware-based cost models can be considered engine based models as well. The difference is merely the degree of abstraction.

3.1.3 The Generic Cost Model

Manegold et. al [2] have defined a generic model to estimate the execution time of algorithms that are commonly used when implementing database operators (most importantly join algorithms). Even though the original application of the model is very different from ours, we will show in the following how the model, with slight modifications, can be applied to our use case.

The model is built around the idea of access patterns and the estimation of their execution costs. Access Patterns are a generic framework to describe the way in which an algorithm reads and writes data from and to memory and estimate the data access costs. Manegold et al. describe several atomic access patterns and an algebra to construct complex access patterns from these atomic patterns. In the following the atomic access patterns will be introduced followed by the definition of the algebra based on them. The estimation of the costs of the execution of the various access patterns will be described as well.

Data Regions

An assumption of Manegold's model is that (relational) data is read from a contiguous section of the memory. Fragmentation of memory due to suboptimal allocation is neglected. Thus, every relation is held in what is called a *data region*. A data region R is an area of memory that is characterized by

- it's length ($R.n$), i.e., the number of stored tuples and
- it's width ($R.w$), the size of a tuple in processor words (we will assume a processor with 64bit words).
- The size of the region ($\|R\|$) is defined as the product of length and width.

Atomic Access Patterns

To describe the implementation of the relational operators that are covered in this thesis, several, though not all, of the access patterns that were defined by Manegold [2] are needed. These are:

s_trav A *single sequential traversal* (see Figure 3.2) of a region of memory is reading some values and optionally skipping constant parts of it. This pattern is exhibited, e.g., by a projection operator when reading it's input from a row-based table. It traverses all tuples of a relation (sequentially) but only reads a non-empty subset of the attributes. Thus, a constant part is skipped after every read tuple. The number of words that are read from each tuple is u.

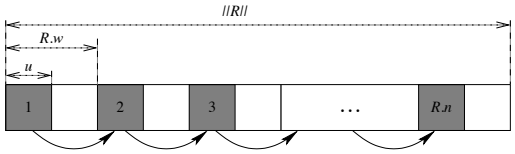


Figure 3.2: *s_trav*: Single Sequential Traversal, figure taken from [2]

r_trav A *random traversal* (see Figure 3.3), like a *s_trav*, accesses *all* values from a region of the memory with a constant unread area of size $R.w - u$ between them. They are, differing from *s_trav*, accessed in a random order.

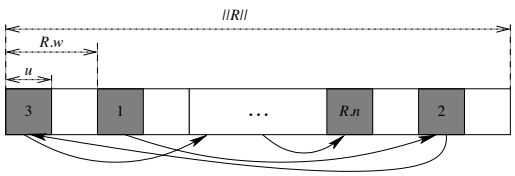


Figure 3.3: *r_trav*: Single Random Traversal, figure taken from [2]

This Access Pattern is most important for the description of the hashing-phase of a hashjoin where every tuple is put into the hashmap at the position that is defined by it's hash. Assuming a good hash function, the positions are pseudo-random and the skipped width, $R.w - u$, is 0 (differing from Figure 3.3). The effect of collisions, which would spoil the random pattern, will be neglected.

rr_acc A *repetitive random access* (Figure 3.4) accesses values from a data region that have a constant distance ($R.w$) in a random order. As before, the number of used words is u. Each value is accessed multiple times, once or not at all. This Access Pattern is most important for the description of the probing-phase of a hashjoin. It is characterized by the total number of accesses r (which may be less, equal or more than the number of tuples in the region). Notwithstanding Manegold's implementation [2], an access of a single tuple (a lookup) will be modeled as a special case of *rr_trav* with the number of accesses being one.

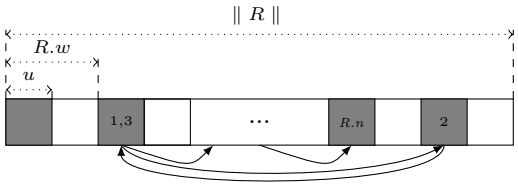


Figure 3.4: *rr_acc*: Repetative Random Access

Complex Access Patterns

Most algorithms expose data access patterns that are more complex than these atomic access patterns. Therefore, Manegold et al. defined an algebra to construct complex access patterns from these atoms. This algebra contains two operators:

$\mathcal{P}_1 \oplus \mathcal{P}_2$ is the sequential execution of the access patterns $\mathcal{P}_1$ and $\mathcal{P}_2$

$\mathcal{P}_1 \odot \mathcal{P}_2$ **(alternatively denoted as** $\odot(\mathcal{P}_1, \mathcal{P}_2, ...)$ **to emphasize the parallel execution of all patterns)** which is the concurrent execution of access patterns.

This algebra allows a description of the data accesses of a piece of code which can be used to estimate the number of cache misses it induces on every layer of memory.

Examples for Complex Access Patterns To give an impression of the model recall the SQL queries in Figure 1.1 on Page 9:

OLTP: `select * where tuple_id = 3;`
OLAP: `select sum(A) from relation;`

In this example, all attributes of the schema have the same width, 1.

Evaluation in a Row-store When executed on a row-store, the first is a simple lookup, i.e., a rr_acc (as mentioned, a lookup is modeled by a random access with $r = 1$) and a sequential traversal of the output region. The access pattern would therefore be

$$\mathcal{P}_{\text{OLTP/Row}} = rr\_acc(R.w = 3, u = 3, R.n = 4, r = 1) \odot s\_trav(R.w = 3, u = 3, R.n = 1)$$

The second is a sequential traversal and a concurrent repetitive access to the output region to update the sum.

$$\mathcal{P}_{\text{OLAP/Row}} = s\_trav(R.w = 3, u = 1, R.n = 4) \odot rr\_acc(R.w = 1, u = 1, R.n = 1, r = 4)$$

Evaluation in a Column-store When executed on a column-store, the first query is more complicated. It involves lookups in all of the three columns (note that $R.w = u = 1$) and the sequential traversal of the output region. The access pattern would therefore be

$$\begin{aligned}\mathcal{P}_{\text{OLTP/Column}} =& rr\_acc(R.w = 1, u = 1, R.n = 4, r = 1) \odot rr\_acc(R.w = 1, u = 1, R.n = 4, r = 1) \\ & \odot rr\_acc(R.w = 1, u = 1, R.n = 4, r = 1) \odot s\_trav(R.w = 3, u = 3, R.n = 1)\end{aligned}$$

The access pattern of the OLAP query on a column-store is very similar to the pattern on a row-store. The only difference is the tuple width ($R.w$) of the sequential traversal.

$$\mathcal{P}_{\text{OLAP/Column}} = s\_trav(R.w = 1, u = 1, R.n = 4) \odot rr\_acc(R.w = 1, u = 1, R.n = 1, r = 4)$$

Evaluation in a Hybrid Store On a hybrid layout, the OLTP query involves two lookups, one in each partition, and the sequential traversal of the output region. The access pattern would therefore be

$$\begin{aligned}\mathcal{P}_{\text{OLTP/Hybrid}} =& rr\_acc(R.w = 1, u = 1, R.n = 4, r = 1) \odot rr\_acc(R.w = 2, u = 2, R.n = 4, r = 1) \\ & \odot s\_trav(R.w = 3, u = 3, R.n = 1)\end{aligned}$$

The OLAP query has the same access pattern as on a column-store.

$$\mathcal{P}_{\text{OLAP/Hybrid}} = s\_trav(R.w = 1, u = 1, R.n = 4) \odot rr\_acc(R.w = 1, u = 1, R.n = 1, r = 4)$$

Based on the descriptions of the access patterns, we can estimate the costs of the queries on the different layouts. To estimate the costs of the access patterns, we first estimate the number of induced cache misses on each memory layer.

Estimating the Number of Cache Misses

The estimation of the cache misses that are induced by such an access pattern as well as the hardware parameters that influence the number of cache misses are discussed in this section.

Parameters of the Cache To calculate the number of misses on a level of the memory hierarchy (denoted with the subscript i) a couple of parameters of the memory layer have to be known:

- The capacity (C_i) of the cache is the total number of words that can be stored and
- The block size (B_i) of the cache is the minimal number of words that can be read at a request. This value is usually greater than one.

A parameter that is derived from these is

- the number of cache lines that can be stored in the cache $\#_i = \frac{C_i}{B_i}$.

In combination with the parameters that define a data region

- the number of cache lines that are covered by a data region R is defined as $|R|_{B_i} = \frac{||R||}{B_i}$.
- The number of tuples of a data region that can be contained in the cache at a given time $|C_i|_{R.w}$ is defined as $|C_i|_{R.w} = \frac{C_i}{R.w}$.

From the parameters of the cache and the size of the data region it is possible to give an estimation of the number of cache misses each atomic access pattern induces.

The Translation Lookaside Buffer (TLB) is regarded as just another cache layer by Manegold's model. This abstracts from the real functionality of the TLB (see Section 2.2.3) but is valid nonetheless, since the TLB influences data access costs just like a data cache [2]. The block size is the virtual memory page size (usually 4Kb) and the capacity is the product of the page size and the number of page-addresses the TLB can hold.

Random and Sequential Misses Cache misses are distinguished into random and sequential misses which may have different costs due to performance optimization features of the CPU (see Section 2.2).

A sequential miss is a miss of a block that is close to the previously read. It can therefore benefit from hardware that exploits data locality.

A random miss is a miss that is not located close to the previous miss. It can not benefit from data locality tunings and therefore induces the full costs of a memory access.

A detailed discussion of the different costs is given in Sections 3.1.3 and 3.1.4. The number of cache misses at a given memory level is therefore not an integer but rather a tuple of two integers which can later be weighted with their respective costs. Following Manegold we will denote

- The number of random misses induced by pattern $\mathcal{P}$ at memory level i with $M_i^r(\mathcal{P})$
- The number of sequential misses induced by pattern $\mathcal{P}$ at memory level i with $M_i^s(\mathcal{P})$
- The total number of misses induced by pattern $\mathcal{P}$ at memory level i is denoted with M_i, $M_i = M_i^s(\mathcal{P}) + M_i^r(\mathcal{P})$

How the number of misses is estimated from the access patterns will be discussed in the following. The atomic patterns will be discussed first, followed by a description of the evaluation of the complex patterns.

A Single Sequential Traversal (s_trav) does only produce one random miss: the first access obviously can not benefit from previous accesses of the access pattern (it may benefit from previous patterns which will be discussed in section 3.1.3). Therefore, the number of random misses is defined by Equation 3.1.

$$M_i^r(s\_trav) = 1 \tag{3.1}$$

The number of sequential misses depends on the width of the gap between the accessed parts of a tuple ($R.w - u$). If this gap is smaller than a cache line, every cache line contains some data that has to be read and no cache line can be skipped. Thus, the number of sequential cache misses is the number of cache lines spanned by the read data region minus the one random miss. It can be calculated using Equation 3.2.

$$M_i^s(s\_trav) = \frac{R.w \cdot R.n}{B_i} - 1 \tag{3.2}$$

If the gap is greater than a cache line some lines may be skipped. Thus the number of misses is determined by the number of words that are actually read of each tuple (u) or, more accurately, the number of cache lines they span $\left\lceil \frac{u}{B_i} \right\rceil$. The number of misses for reading all tuples of the region can be calculated using Equation 3.3.

$$M_i^s(s\_trav) = R.n \cdot \left\lceil \frac{u}{B_i} \right\rceil - 1 \tag{3.3}$$

If more than one data word is read ($u > 1$) it could happen that the chunk of read data is suboptimally aligned which would result in reading an additional cache line. Figure 3.5 shows a chunk of data that is small enough to fit into one cache line but yields two misses nonetheless. To model this effect the average number of sequential misses per tuple is increased by the probability of this effect. This yields Equation 3.4.

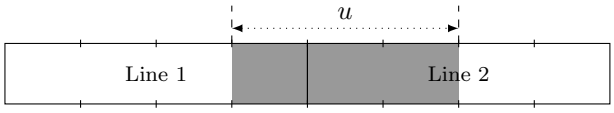


Figure 3.5: Additional Miss for Suboptimally aligned Data

$$M_i^s(s\_trav) = R.n \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i} \right) \tag{3.4}$$

A Single Random Traversal (r_trav) does not induce any sequential misses (equation 3.5).

$$M_i^s(s\_trav) = 0 \tag{3.5}$$

The number of random misses again depends on the gap between accessed data.

If the gap is wider than a cache line ($R.w - u \geq B$) no data access can benefit from a previous one since every tuple lies on a new cache line. As in the previous case tuples may be suboptimally aligned to the beginning of a cache line (as mentioned in Section 2.2.3, cache lines start at fixed memory addresses)

. Thus, the number of random misses can be estimated using equation 3.6 (the reader may notice the similarity to the previous cases, Equation 3.4).

$$M_i^r\left(r\_trav\right) = R.n \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i} \right) \tag{3.6}$$

If the gap between the read data is smaller than a cache line, some cache lines have to be accessed multiple times (once for every tuple that has relevant values on that cache line). This becomes a significant factor if the region is larger than the cache because cache lines may be evicted before all accesses to them are processed. The probability of an early eviction increases with the size of the data region (relative to the cache size) : It is $\left(1 - \min\left\{1, \frac{C_i}{||R||}\right\}\right)$. Evictions do not happen for the first lines that are read because no lines from the read region are present in the cache. The number of tuples that can be read without eviction is limited by the number of slots of the cache $\#_i$ because every tuple occupies at least one cache line. It is, however, also influenced by the width of a tuple R_w. If a tuple tuple occupies more than one slot the number of tuples that can be stored in the cache is $|C_i|_{R.w}$. The expected number of tuples that can be read in a `r_trav` without eviction, $\mathbf{E_{r\_trav/NE}}$, can, therefore, be calculated using Equation 3.7.

$$\mathbf{E_{r\_trav/NE}}\left(C_i, R\right) = \min\left(\#_i, |C_i|_{R.w}\right) \tag{3.7}$$

This results in Equation 3.8 for the expected number of random misses for a single random traversal if the gaps are smaller than a cache line.

$$M_i^r\left(r\_trav\right) = |R|_{B_i} + \left(R.n - \min\left(\#_i, |C_i|_{R.w}\right)\right)\left(1 - \min\left\{1, \frac{C_i}{||R||}\right\}\right) \tag{3.8}$$

A Repetitive Random Access (rr_acc) is similar to a r_trav in that it produces no sequential misses. The number of random misses depends on the number of access-operations r as well as the total number of tuples $R.n$. Since multiple accesses to a tuple are possible, not every access necessarily induces a cache miss. If a tuple is still resident in the cache from a previous access it does not induce an additional miss. To calculate the probability of this, the total number of distinct accessed tuples (or records) $\mathbf{I}$ is of importance.

The Problem of Distinct Record Selection Estimating the number of distinct accessed records when records are accessed randomly, independently and possibly repetitively is the problem of *Distinct Record Selection*. To estimate this number Manegold uses Equations 3.9 and 3.10.

$$\mathbf{I}\left(r, R.n\right) = \frac{\sum_{j=1}^{\min(r,R.n)} \left(\begin{array}{c} R.n \\ j \end{array} \right) \cdot \left\{ \begin{array}{c} r \\ j \end{array} \right\} \cdot j!}{R.n^r} \tag{3.9}$$

with

$$\left\{ \begin{array}{c} x \\ y \end{array} \right\} = \frac{1}{y!} \cdot \sum_{k=0}^{y-1} (-1)^k \cdot \left(\begin{array}{c} y \\ k \end{array} \right) \cdot (y-k)^x \tag{3.10}$$

The evaluation of this equation is very complex for large values of x and y. This is due to the extensive use of the binomial factor which in turn is evaluated using the factorial function $x!$. The factorial of x, the product of $1, 2, ..., x$, needs x multiplications.

Calculation of the factorial for large numbers cannot be done using integer arithmetics. The value for 21! already exceeds the maximal value of a 64 bit unsigned integer. One application of Equation 3.9 is the estimation of the misses for a hash join. If both joined relations have 30000 tuples, which we consider relatively few, $x = y = 30000$. The factorial of x, $x! \approx 2.7595 \times 10^{121287}$, has to be calculated using floating point arithmetics. Floating point arithmetics for numbers of that dimension, however, results in a loss of accuracy. The high calculation effort and the fact that the result will be inaccurate at any rate encourages investigations into an estimation that is less computation intensive.

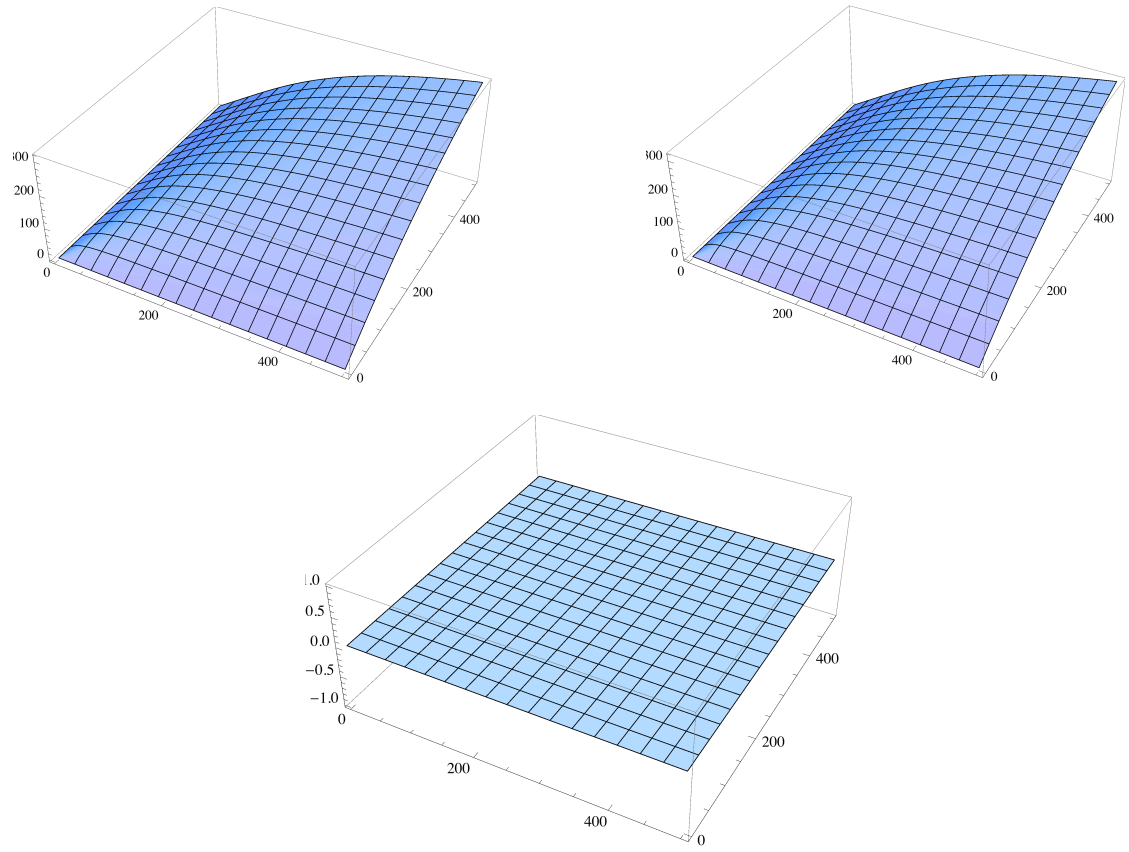


Figure 3.6: Manegolds Equation for distinct record access (top left), Cardenas' Approximation (top right) and their deviation (bottom) for the first 500x500 Values

Independently of the work of Manegold et al., the problem of distinct record selection is widely (and surprisingly controversially) discussed. Cardenas [48] provides Equation 3.11 as an approximation for the number of accessed tuples.

$$\mathbf{I}\left(r, R.n\right) = R.n \cdot \left(1 - \left(1 - \frac{1}{R.n}\right)^{r}\right) \tag{3.11}$$

The correctness of this approximation has been challenged repeatedly [57, 58, 59]. A detailed discussion of this approximation is out of scope of this thesis. To illustrate the accuracy of Cardenas' approximation, however, Figure 3.6 shows plots of Equations 3.11 and 3.9 as well as their deviation for $0 < R.n < 500, 0 < r < 500$. The deviation is virtually 0.

Based on the number of distinct accessed tuples the number of distinct accessed cache lines (not the number of cache misses) can be calculated similarly to the previous cases. If the gap between the tuples is larger than a cache line, the number of accessed cache lines $\mathbf{C}$ can be calculated using Equation 3.12 which is similar to Equation 3.6:

$$\mathbf{C}_i = \mathbf{I} \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i}\right) \tag{3.12}$$

If the gap between two tuples is smaller than a cache line, calculating the number of distinct accessed cache lines becomes more complicated. If all accessed tuples are adjacent, thus no cache line is skipped, the number of distinct accessed cache lines is estimated with Equation 3.13.

$$\check{\mathbf{C}}_i = \left\lceil \frac{\mathbf{I} \cdot R.w}{B_i} \right\rceil \tag{3.13}$$

If not all tuples are read ($\mathbf{I} < R.n$), the gap between two read tuples may still be larger than a single cache line in which case equation 3.14 applies.

$$\hat{\mathbf{C}}_i = \mathbf{I} \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i} \right) \tag{3.14}$$

The smaller $\mathbf{I}$ is in relation to $R.n$, the more cache lines will be skipped. Thus, Manegold proposes to weight both cases with a factor that is to reflect the probability of each case. This differentiation is similar to the one made for a *r_trav* (equation 3.8) but much more complicated due to the fact that not all tuples are accessed. This results in Equation 3.15 for the number of touched cache lines.

$$\mathbf{C}_i = \frac{\mathbf{I}}{R.n} \cdot \check{\mathbf{C}}_\mathbf{i} + \left(1 - \frac{\mathbf{I}}{R.n}\right) \cdot \hat{\mathbf{C}}_\mathbf{i} \tag{3.15}$$

The weighting of $\check{C}_i$ and $\hat{C}_i$, as proposed by Manegold, is independent of the cache line size of the cache the pattern is performed on. For large cache lines, however, the probability that a cache line is used by multiple data items is higher than for small cache lines. Thus, we used a weighting that takes the block size of the cache into account. It is given in Equation 3.16. Our experiments (see Section 5.2.2) show that this alternative weighting increases the accuracy of the prediction when caches with large line sizes are involved.

$$\mathbf{C}_i = \left(1 - \left(1 - \frac{\mathbf{I}}{R.n}\right)^{\frac{B_i}{u}}\right) \cdot \check{\mathbf{C}}_\mathbf{i} + \left(1 - \frac{\mathbf{I}}{R.n}\right)^{\frac{B_i}{u}} \cdot \hat{\mathbf{C}}_\mathbf{i} \tag{3.16}$$

The number of cache misses of a rr_acc also depends on the capacity of the cache. If the capacity is higher than the number of touched cache lines, the number of misses is equal to the number of touched cache lines. If the number of touched cache lines exceeds the capacity of the cache the number of cache misses increases. Since the expected number of accesses to a cache line is $\frac{r}{\mathbf{I}}$ the number of subsequent accesses is $\frac{r}{\mathbf{I}} - 1$. Thus all cache lines induce $\frac{r}{\mathbf{I}} - 1$ additional misses, if they are not already present in the cache when accessed. Every of the $\#_i$ lines that are already in the cache can be reused with the probability $\frac{\#_i}{\mathbf{C}_i}$. The number of cache misses decreases by that factor, resulting in Equation 3.17 for the number of random misses.

$$\mathbf{M}_i^r = \begin{cases} \mathbf{C}_i & \text{if } \mathbf{C}_i \leq \#_i \\ \mathbf{C}_i + \left(\frac{r}{\mathbf{I}} - 1\right) \cdot \left(\mathbf{C}_i - \frac{\#_i}{\mathbf{C}_i} \cdot \#_i\right) & \text{if } \mathbf{C}_i > \#_i \end{cases} \tag{3.17}$$

Based on the cost estimation of these atomics it is possible to estimate the costs of complex access patterns: sequentially and concurrently executed access patterns.

Sequential Execution The costs of sequentially executed access patterns is fairly simple to evaluate. Sequentially executed access patterns do not compete for space in the cache. The number of cache misses can therefore be at most the sum of the misses of the individual patterns. They may indeed benefit from each other, if the lines that reside in the cache after the execution of the first can be reused by the second pattern. To model this effect, Manegold et al. introduced the state of a cache $\mathbf{S}_i$. It is defined as the percentage of cache lines from each data region it contains (see Equation 3.18).

$$\mathbf{S}_i = \{\langle R, \rho \rangle\} \subset \mathbb{D} \times [0,1] \text{, with } \langle R, \rho_1 \rangle \in \mathbf{S} \wedge \langle R, \rho_2 \rangle \in \mathbf{S} \Rightarrow \rho_1 = \rho_2 \tag{3.18}$$

The cache state after performing an access pattern on the data region R is defined by Equation 3.19 in Manegold's model. The reader may note that $\mathbf{C}_i$ is the number of distinct accessed cache lines and not to be confused with C_i which is the capacity of the cache.

$$\mathbf{S}_i = \left\{ \left\langle R, \min\left(\frac{\mathbf{C}_i}{||R||}, 1\right) \right\rangle \right\} \tag{3.19}$$

The expected number of random and sequential misses induced by a pattern $\mathcal{P}$ on a cache in the state S_i is denoted with $M_i^x(S_i, \mathcal{P}), x \in \{r, s\}$.

Sequentially executed patterns only benefit if the cache lines that are stored in the cache when starting the pattern are amongst the first to be read. Since this is very hard to determine Manegold et al. assume that sequential patterns can only benefit if the complete region they read is in the cache (Equation 3.20).

$$M_i^s(S_i, s\_trav) = \begin{cases} 0 & \text{if } \langle R, 1\rangle \in S_i \\ M_i^s(s\_trav) & \text{else} \end{cases} \tag{3.20}$$

Random patterns benefit from the resident cache lines with a the probability that is equal to the percentage of accessed cache lines that are already in the cache $\frac{\rho \cdot |R|_{B_i}}{\mathbf{M}_i^r}$. Since there are $|R|_{B_i}$ lines in the cache, the expected number of reusable cache lines is calculated using Equation 3.21.

$$\mathbf{E}(\mathcal{P}, S_i, R) = \frac{\rho_R \cdot |R|_{B_i}}{\mathbf{M}_i(\mathcal{P})} \cdot |R|_{B_i} \tag{3.21}$$

The number of random misses for a random access pattern (r_trav or rr_acc) that is executed on a cache in this state is reduced by that number. The expected number of cache misses of a random access pattern $\mathcal{P}$ on a cache in state S_i can, therefore, be calculated using Equation 3.22.

$$M_i^r(S_i, \mathcal{P}) = M_i^r(\mathcal{P}) - \frac{\rho \cdot |R|_{B_i}}{\mathbf{M}_i^r} \cdot |R|_{B_i} \tag{3.22}$$

Concurrent Execution Concurrently executed access patterns could also benefit from each other but only if they act on the same data region. This effect has been studied by Zukowski et al. [60] and is exploited to improve the performance of concurrent scans. It is a relatively new technique, not reflected in Manegold's model and considered out of scope of this thesis as well.

Therefore, it is assumed that concurrent access patterns negatively affect each other: they compete for space in the cache. This is modeled by assigning each access pattern a fraction of the cache according to their footprint $\mathbf{F}$. The footprint represents the number of cache slots that are occupied by an access pattern. It is defined by Manegold as the number of cache lines that are potentially revisited, i.e., read more than once. Since s_trav does not revisit any cache lines it always only occupies one slot, hence it's footprint is 1. The same holds true for r_trav if the gap is larger than a cache line. For all other cases Manegold defines the size of the whole data region as footprint because all cache lines could potentially be revisited.

For sequential execution the footprint is defined by Equation 3.23.

$$\mathbf{F}_i(\mathcal{P}_1 \oplus \mathcal{P}_2) = \max(\mathbf{F}_i(\mathcal{P}_1), \mathbf{F}_i(\mathcal{P}_2)) \tag{3.23}$$

For concurrent execution the footprint is defined by Equation 3.24.

$$\mathbf{F}_i(\mathcal{P}_1 \odot \mathcal{P}_2 \odot ... \odot \mathcal{P}_m) = \mathbf{F}_i(\mathcal{P}_1) + \mathbf{F}_i(\mathcal{P}_2) + ... + \mathbf{F}_i(\mathcal{P}_m) \tag{3.24}$$

Based on the footprint the number of cache misses of an access pattern is evaluated on a virtually smaller cache. The relative size of the cache for pattern $\mathcal{P}_n, \mathcal{P}_n \in \odot(\mathcal{P}_1, ..., \mathcal{P}_m)$ is determined by the factor $v_{\mathcal{P}_n}$ as defined in Equation 3.25

$$v_{\mathcal{P}_n} = \frac{\mathbf{F}(\mathcal{P}_1 \odot \mathcal{P}_2 \odot ... \odot \mathcal{P}_n \odot ... \odot \mathcal{P}_m)}{\mathbf{F}(\mathcal{P}_n)} \tag{3.25}$$

The number of cache misses of a pattern is then simply estimated on a cache of the size $\frac{C}{v}$. The state after performing a concurrent pattern is simply the union of the states of the individual pattern.

Estimating the Execution Costs

Based on the number of cache misses (M^r and M^s) on each level of cache it is possible to evaluate the overall execution costs (T_{Mem} in CPU cycles). Manegold et al. did this by simply weighting and summing the cache misses based on the Hit Time (l^r and l^s) of the respective cache (equation 3.26).

$$T_{Mem} = \sum_i (M_i^r \cdot l_i^r + M_i^s \cdot l_i^s) \tag{3.26}$$

To determine the parameters of each level of cache Manegold developed the *The Calibrator (v0.9e)*[2]. It measures the capacity, line size and hit time of each of the caches (and the TLB) as well as the hit time of the memory itself. This is done similar to the way Figure 2.1 and 2.2 in Section 2.2.1 were produced: by accessing the memory with varying strides and analyzing the average read time. This allows determining the line size and the latency by finding points on the x-axis (the stride) at which the proportionality to the average access time changes. To determine the capacity, a varying number of values is repeatedly accessed. When accessing a number greater than the capacity of the cache the average access time will increase disproportionally. A detailed discussion of the determination of the parameters is given in Section 5.2.1.

3.1.4 Extensions to the Generic Cost Model

Though very powerful, Manegolds model was mainly intended to model and evaluate the performance of join algorithms. It proofed insufficient for modeling our query processor because it lacks a pattern to accurately model tuple reconstruction in a column-store (see Section 5.2). This is why an extension to the model was developed that allows a more accurate estimation of the costs of the tuple reconstruction that is performed in a column-based database (see Section 2.3.1).

A second shortcoming of Manegold's model is its age: properties of the hardware in 2002 are not the same of current hardware. We found that the most important unconsidered factor was the Level 2 Cache Prefetching (see Section 2.2.3). This is why the model was extended further with an alternative function for the cost estimation. Like the original cost function, it weights the misses on different levels but also takes the effects of Level 2 Prefetching into account. This extension will be discussed in this section.

Sequential Traversal Conditional Read (s_trav_cr)

When evaluating a query on a relation that is stored column-oriented it is often necessary to sequentially traverse a region in memory but only read it in case a condition holds. To illustrate this, consider the case shown in Figure 3.7. To evaluate the query in a non-indexed column-store, a scan of the attribute `ADRC.NAME` is necessary. The values of `ADRC.KUNNR`, however, are traversed but only read when the condition holds (which it does for 0.02% of the tuples in this example, hence the selectivity of 0.0002 in the *s_trav_cr*). This means that the number of cache misses is significantly lower than it would be if every value of `ADRC.KUNNR` was read. The expected number of cache misses depends on the selectivity, the width of a cache line and the distribution of the values of that attribute. Cache line widths, Level 1 as well as Level 2, of most modern CPUs are 64 Bytes, thus holding 16 32bit integer values. Assuming equally distributed values in a column, the probability that a cache line has to be read is the probability that one or more of the 16 values have to be read. The probability for each one having to be read is the selectivity. The probability for each cache line to be read into cache i is P_i.It can be calculated using Equation 3.27.

$$P_i = 1 - (1 - selectivity)^{B_i} \tag{3.27}$$

The expected number of cache misses (sequential and random) is the product of the number of spanned cache lines and the probability of an access to a cache line (equation 3.28).

$$M_i\,(s\_trav\_cr) = P_i \cdot |R|_{B_i} \tag{3.28}$$

[2]available at `http://monetdb.cwi.nl/Calibrator/`

Schema:

Table	Attribute	Cardinality	Data type
ADRC	ID[a]	5000[b]	Word
ADRC	NAME	5000	Word
ADRC	KUNNR	5000	Word

Query:

```
select KUNNR from ADRC where NAME = $1;
```

Access Pattern in a column store:

$s\_trav(R.w = 1, R.n = 5000, u = 1)$ // scan `NAME`
$\odot$ $s\_trav\_cr(R.w = 1, R.n = 5000, u = 1, selectivity = 0.0002)$ // reconstruct tuple
$\odot$ $s\_trav(R.w = 1, R.n = 1, u = 1)$ // tuple output

[a]The ID is the position of a tuple – an implicit attribute, i.e., not physically stored
[b]The cardinality of the ID column is the number of stored tuples

Figure 3.7: A Very Simple Query and It's Access Patterns

Random and Sequential Misses for s_trav_cr As described in Section 2.2.3, the Intel® Core™ Microarchitecture defines two prefetching strategies, *DPL* and *Streaming Prefetching*. Assuming randomly distributed values in the column, the distance of accessed values in the memory is not constant. Thus, the *DPL* will not detect a stride and, therefore, not trigger any prefetches. The *Streaming Prefetcher* however can be useful since it always fetches the next adjacent cache line. The exact benefit (or penalty) of the prefetching depends on other parameters that we will discuss later in this section. For a `s_trav_cr`, the prefetching strategy is assumed to be *Streaming Prefetching*.

Using the *Streaming Prefetcher* we expect a sequential miss, i.e., a prefetched cache line, if and only if a cache line is accessed with the previous cache line being accessed as well. Therefore, the probability of a cache line to be a sequential miss is the probability of an access to the current and the previous cache line. The probability of this combined event is the product of the probability of the individual events. Since the probability is the same for every cache line (Equation 3.27), the combined probability P_i^s can be calculated using Equation 3.29.

$$P_i^s = \left(1 - (1 - selectivity)^{B_i}\right)^2 \tag{3.29}$$

The expected number of sequential cache misses can be calculated using equation.

$$M_i^s\left(s\_trav\_cr\right) = P_i^s \cdot |R|_{B_i} \tag{3.30}$$

All misses that are not sequential misses are random misses. The expected number of random cache misses can, therefore, be calculated using Equation 3.31.

$$M_i^r\left(s\_trav\_cr\right) = \left(P_i - P_i^s\right) \cdot |R|_{B_i} \tag{3.31}$$

Figure 3.8 shows the number of cache misses and their type for a s_trav_cr for a selectivity from 0 (no values are read) to 1 (all values are read). We can see that the number of sequential misses is growing fast due to the high width of a cache line (for a narrower cache line it is growing slower). The number of random misses is growing even faster at first but is then decreasing in favor of more sequential misses. For a selectivity of 1 we have no random misses at all and all values have to be read sequentially.

The state left by s_trav_cr The state that is left by a s_trav_cr can be calculated straight forwardly from the expected number of cache misses. Since the number of cache misses also denotes the percentage of the data region that is read, the state can be calculated using Equation 3.32.

$$\mathbf{S}_i = \left\{\left\langle R, \min\left(\frac{C_i}{||R||}, 1, \frac{M_i}{|C_i|_{R.w}}\right)\right\rangle\right\} \tag{3.32}$$

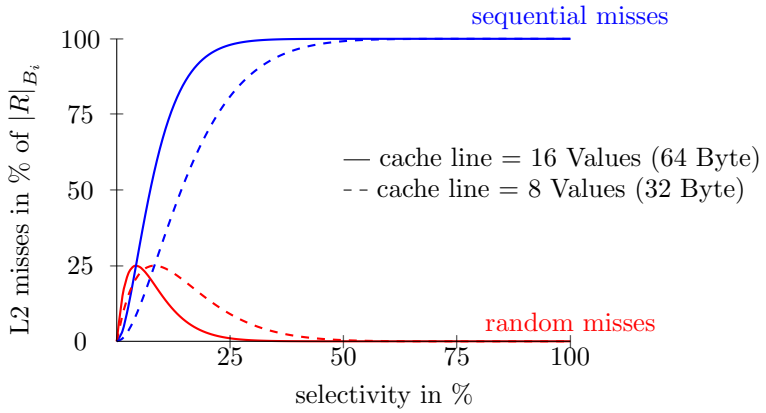


Figure 3.8: Random vs. Sequential Misses for s_trav_cr

CPU Prefetching

The estimation of the number of cache misses has been described. To estimate the data access costs it is important to take the additional latency of a random miss into account. As explained in Section 2.2.3 the CPU speeds up sequential misses by means of cache line prefetching. To estimate the costs more accurately, the second extension to Manegold's generic cost model will be described in the following: an alternative cost function that takes prefetching into account.

Correct Prefetching (Sequential Access) When traversing the memory sequentially the Level 2 cache asynchronously fetches the next adjacent cache line from memory *while* the CPU is processing the current. When the CPU requests the next cache line the cache has a head start since it already started fetching it. The benefit of the prefetching is, therefore, highly dependent of the time it takes to process the current cache line. Following the rationale that execution time is determined by cache misses it depends on the number of misses that are induced at the levels above the L2: the L1 and the processor registers (which we consider just another layer of memory). The benefit of prefetching (as illustrated in Section 2.2.3) depends on the processing time on the upper levels. The costs for sequential L2 misses are reduced by the time it takes to process its values. If the processing of the values takes longer than the L2-fetching, the overall costs are solely determined by the processing time (recall Figure 2.6 from Section 2.2.3) . If it takes longer to process than to fetch, the costs will obviously be 0 and not negative. The overall costs for sequential misses in the Level 2 cache can be calculated using Equation 3.33.

Following [2], the costs (T_{Mem} in CPU cycles) for an access to level i (i.e., a miss on local $i-1$) will be denoted with l_i. Since we regard the CPU's registers as just another level of memory, l_1 denotes the time it takes to load and process one value and M_0 the number of values that have to be processed.

$$T_{2s} = max\left(0, M_2^s \cdot l_3 - \sum_{i=0}^{1} M_i \cdot l_{i+1}\right) \tag{3.33}$$

Incorrect Prefetching (Random Access) When looking at random accesses, prefetching is an important factor as well. Incorrect prefetching has two different drawbacks: on the one hand it does not allow benefiting from correct prefetching and on the other hand it even decreases the performance because any fetching blocks the I/O bus and the memory. Thus the CPU spends double the time waiting for the L2. We model this effect by doubling the penalty for random misses (equation 3.34).

$$T_{2r} = M_2^r \cdot 2 \cdot l_3 \tag{3.34}$$

Blocks in the Page Table

As discussed in Section 2.2.3, lookups in the Page Table and the Page Directory may induce additional Level 2 Cache misses. If two Page Table Entries are stored in the same Level 2 cache line, the address translation is significantly faster than it is otherwise. Thus, the cache line size of the Level 2 Cache introduces another layer of blocking: blocks in the page table. The size of a Page Table Block is the product of the number of PTEs on a Level 2 Cache Line and the size of a Virtual Memory Page. We will model the Page Table Blocks as another layer of memory.

The overall costs T_{Mem} are calculated by summing the weighted misses of all cache layers except the level 2 cache and the TLB. The costs for level 2 and TLB misses are calculated using Equations 3.34 and 3.33 and added to the overall costs (Equation 3.35).

$$T_{Mem} = T_{2s} + T_{2r} + \sum_{i=0}^{1} M_i \cdot l_{i+1} + \sum_{i=3}^{N} M_i \cdot l_{i+1} \tag{3.35}$$

Untackled Limitations of the Generic Cost Model

Due to it's statistical nature, Manegold's cost model has strong limitations when it comes to the prediction of the number of cache misses. It can not take effects into account that are not statistical in nature. One of these is the effect of early evictions through bad associativity. When cache lines are mapped to the same set they may be evicted even though there are still "free" slots in other sets. In the worst case, when all accessed cache lines are mapped to the same set, the cache size is effectively reduced to the size of one set.

This effect occurs, e.g., when reconstructing tuples in a column store. It may happen that, by bad arrangement in memory, the start addresses of all columns fall into the same set. In that case, when reconstructing more attributes than there are slots in a set, early in-set-evictions may happen. This, can be avoided by careful implementation of the memory allocation of the DBMS. *HYRISE* is implemented that way and does therefore not suffer from this problem[3] [49].

Another important limitation is the assumption of contiguous accessed attributes. The estimation of the number of cache misses is only accurate if the accessed values of a tuple are indeed read from a contiguous area in memory. Thus, the model always assumes an optimal ordering of the attributes in a row-store. Picking the optimal ordering of attributes within a partition is left for future work.

Modeling column compression

Even though, the statistical nature of the generic cost model is a problem in some cases it provides a simplification in other cases. Optimizations of the query processor that expose a benefit that can be statistically modeled can easily be integrated into the model. A good example is the compression of stored values which is common in column stores [6]. This increases the expected number of values per transmitted cache line which saves bandwidth. Since the values have to be decompressed in the CPU, the expected processing time per value, however, increases. This changes the expected cache line width and the expected processing time. Due to the linearity of the Expected Value, this can be modeled by simply substituting the real values with the expected values. Since *HYRISE* does not yet incorporate any compression it was not modeled.

3.1.5 Modeling the Query Processor

Based on the formalism of the generic cost model, the data access behavior of a query processor can be described. Assuming the query is evaluated operator-by-operator it suffices to describe the behavior of the operators and combine them using the $\oplus$-operator to model their sequential execution. A more sophisticated evaluation scheme can easily be modeled with a different evaluation function.

Relational algebra operator trees were discussed in Section 3.1.1 (a description of a simple SQL to relational algebra compiler is given in Section 4.2.2). The overall access pattern is generated by mapping

[3] At least not when reconstructing tuples, other cases where this effect occurs are not known to us

Operator		**Column-Store (EM)**	**Column-Store (LM)**	**Row-Store**
projection	input	**s_trav(attr1) ⊙ s_trav(attr2) ⊙ ...**		**s_trav(attr1 | attr2 | ...)**
	output	**⊙ s_trav(attr1 | attr2 | attr3 | ...)**		**⊙ s_trav(attr1 | attr3 | ...)**
selection	input	**s_trav(attr1)**	**s_trav(id) ⊙ s_trav_cr(attr1)**	**s_trav(attr1 | attr2 | ...)**
	output	**⊙ s_trav(selectivity · (attr1))**	**⊙ s_trav(selectivity · (id))**	**⊙ s_trav(selectivity · (attr1 | attr2 | ...))**
projection + selection	input	**s_trav(attr1) ⊙ s_trav_cr(selectivity, attr2) ⊙ ...**	**s_trav(id) ⊙ s_trav_cr(attr1)**	**s_trav(attr1 | attr2 | attr3 | ...)**
	output	**⊙ s_trav(selectivity · (attr2 | ...))**	**⊙ s_trav(selectivity · (id))**	**⊙ s_trav(selectivity · (attr1 | attr2 | ...))**
group by	input	**s_trav(attr1) ⊙ s_trav(attr2)**	**s_trav(rel1.id) ⊙ s_trav(rel2.id) ⊙ s_trav_cr(rel1.attr1) ⊙ rr_acc(|rel2.id| · rel2.attr2)**	**s_trav(rel1.attr1 | rel1.attr2 | ...)**
	output	**⊙ rr_acc(extension(attr1[a]) · (rel1.attr1 | rel1.attr2 | ...))**	**⊙ rr_acc(extension(attr1) · (rel1.attr1 | rel1.attr2 | ...))**	**⊙ rr_acc(extension(attr1) · (rel1.attr1 | rel1.attr2 | ...))**
(hash)join	input	**s_trav(rel1.attr1)** ⊙ r_trav(rel1.attr1)[b] ⊕ **s_trav(rel2.attr1)** ⊙ rr_acc(rel1.attr1)	**s_trav(rel1.id) ⊙ s_trav_cr(rel1.attr1)** ⊙ r_trav(rel1.attr1) ⊕ **s_trav(rel2.id) ⊙ s_trav_cr(rel2.attr2)** ⊙ rr_acc(rel1.attr1)	**s_trav(rel1.attr1 | rel1.attr2 | ...)** ⊙ r_trav(rel1.attr1 | rel1.attr2 | ...) ⊕ **s_trav(rel2.attr1 | rel2.attr2 | ...)** ⊙ rr_acc(rel1.attr1 | rel1.attr2 | ...)
	output	**⊙ s_trav(selectivity · (rel1.attr1 | rel2.attr1))**	**⊙ s_trav(selectivity · (rel1.id | rel2.id))**	**⊙ s_trav(selectivity · (rel1.attr1 | rel1.attr2 | rel2.attr1 | rel2.attr2 | ...))**

Table 3.1: Relational Operators and Their Access Patterns

[a] attribute that is grouped by (potentially multiple attributes)
[b] Access Patterns that are set in normal text are performed on temporary/internal data regions

every operator to its appropriate pattern. Since column and row oriented query processors have different implementations for the same operators, each has to be described with its own access pattern. Table 3.1 shows the set of relational operators that were modeled (union and difference have not been modeled because they were not needed to support the use cases defined in Chapter 5).

Due to the importance of *Late Materialization* (see Section 2.3.1) for the performance of a column store it is reasonable to consider this optimization. As shown by Abadi et al. [7], selecting the optimal materialization strategy for a query is not trivial. Even though the cost model could be used to make a qualified decision on the optimal materialization strategy this is not the focus of this thesis. When evaluating the costs of a layout, our implementation (see Section 4.3) allows a selection of the used materialization strategy by hand (the options are *As Early As Possible* and *As Late As Possible*). To this end, all operators of a column store have two implementations: one for *Early Materialization (EM)* and one for *Late Materialization (LM)*. When calculating the access pattern from the operator tree it is possible to switch the materialization strategy per operator. This is necessary because the root-operator would always have to use *Early Materialization* to reconstruct the tuples for the output. All operators on top of a materializing operator are implemented using the Row-Store because the tuples have been reconstructed and materialized row-wise.

To clarify the implementation for some of the non-trivial cases in table 3.1, a brief discussion will be provided in the following.

Projection (CStore/LM) A Projection using *Late Materialization* is a no_op because the input is a vector of ids and the output is the same vector.

Projection+Selection (CStore/EM) This is the most common operator in a column store. A predicate is applied to a relation and some of the attributes selected. The first column (the selection attribute `attr1`) is scanned completely using a s_trav. All other attributes (selection as well as projected attributes) are scanned using a s_trav_cr because they are only read if all previous conditions hold true. The selectivity of the s_trav_cr decreases with every condition.

Projection+Selection (CStore/LM) For *Late Materialization*, the Projection+Selection operator reads it's input as a vector of ids and uses them as its first condition (holding true for all values that are contained in the vector). Thus all further conditions are evaluated using a s_trav_cr. Projected attributes are not read nor materialized and the output is only the ids.

If the input is a stored relation, the id does not have to be read because it is implicit from the address of the tuple. This holds for all operators.

Group By (CStore/EM) The access pattern of a group by is largely defined by its aggregation function. Only the case of an algebraic aggregation function that uses only a single intermediate result per group was covered. It can therefore group the results by traversing the grouping (`attr1`) and aggregation (`attr2`) columns and storing the aggregated result at the appropriate memory location. Since each aggregated tuple may be stored in any of the slots of the output relation (of size extension(attr1)) it is stored using a repetitive random access (rr_acc).

Group By (CStore/LM) This access pattern is very similar to the Group By (CStore/EM) pattern. In addition to the EM-version, the input also includes the reconstruction of the necessary attributes through a s_trav_cr and a rr_acc. The reason for the random traversal is the sorting at which the tuples come in. An id vector that contains ids of multiple relations is only sorted by one of them (w.l.o.g. the first) and may even contain duplicates in the other columns[4]. Thus only the values from the first relation can be constructed using a sequential traversal. All other relations are reconstructed using a random traversal, hence the r_acc. The output of a *Late Materialization* group by has to contain the aggregates in materialized form because they can not be reconstructed from the relations. The grouping

[4]it may also contain duplicates in the first column as well but they do not change the access pattern because they appear in a consecutive area in the vector and can therefore be reconstructed with a single lookup

attributes however could be reconstructed from the relations and could therefore be returned using their id. This, however, is generally not advisable because the group by is expected to be the last operator that is evaluated (except for nested queries).

Join (CStore/EM) Only one join algorithm is modeled: a hashjoin. The modeling of alternative join algorithms is straight forward ([2] contains models of alternative join algorithms). The selection of the optimal algorithm for a query, however, is not trivial. Since this problem is not focus of this thesis, only one join algorithm was considered. To recall a hashjoin [43]: in the first phase the elements of the first relation are scanned (s_trav) and assigned to slots in a temporary hashmap buffer based on the value of a hash function. Since every slot of the output buffer is filled with a tuple but the order in which they are filled is (pseudo-)random the assignment results in a r_trav. In the second phase, the second relation is scanned (s_trav) and every tuple is looked up in the index using a position that is again calculated using the hash function. On a match each tuple is written to the output buffer (s_trav).

Join (CStore/LM) Just like the group by, the *Late Materialization* join differs in the additional s_trav_cr for tuple reconstruction and the output of the ids instead of the values of the result tuples. For a join, however, both id vector inputs are sorted and the values of both relations can be reconstructed using a s_trav_cr.

3.2 Data Layouting

Based on the model of the query execution and its costs, a vertically partitioned layout can be proposed that reduces the costs for a given workload. Since the body of existing research on the topic of vertical partitioning is immense [16, 17, 18, 19, 20, 21, 22, 23] we decided to show how to integrate the generic cost model into existing approaches instead of developing a layouting method from scratch.

We implemented two different algorithms: a linear program that is based on the simplex algorithm [61] and an extension of the Optimal Binary Partitioning (OBP) Algorithm [18]. The earlier approach is called *Simplex Based Layouting*, the later *Oriented Optimal Binary Partitioning (OOBP)*.

The earlier approach was selected for its simplicity and low computational complexity, but only generates analytically optimal *unpartitioned* layouts, i.e., it selects the analytically optimal orientation for each relation. We therefore expect it to be best suited for fully normalized schemas, because they are expected to already possess strong transactional affinity between the attributes of a relation. The Simplex Based Layouting has the advantage of a relatively low optimization effort but is less useful if some attributes in a relation are used for OLAP and others only accessed in OLTP queries. Since it does not generate a partitioning, all attributes of a relation are stored in the same orientation which may be inappropriate.

OOBP aims at calculating the optimal partitioned layout. This makes it a more powerful approach than the Simplex Based Layouting but comes at the cost of higher optimization effort. Especially for a highly denormalized schema with few, broad relations this algorithm yields better results (see Chapter 5). An inspection of the relations in an SAP R/3 system (see Chapter 5) suggests that a schema with (relatively) few, broad tables is the more common case in an enterprise scenario. Although more complex than the Simplex Based Layouting, OOBP still has a complexity that is independent of the number of attributes in a schema. This makes it applicable for large schemas as well (in contrast to, e.g., the Data Morphing approach [17]).

3.2.1 Formal Problem Definition

Datatype A Datatype $type = (name, int)$ is a tuple of a name (an alphanumerical string) and a size

Attribute An Attribute $a = (name, type)$ is a tuple of a name and a datatype

Table A Table $t = (name, \{a_1, a_2, ...\})$ is a tuple of a name and a set of attributes

Schema The Schema of a Database is a set of tables $S = \{t_1, t_2, ...\}$ with pairwise unequal names, the size of the schema $|s|$ is the number of tables

Partition A Partition $p = \{a_1, a_2, ...\}$ of a Table t is a subset of it's attributes

Orientation An orientation o is an element of the set $\{\text{row}, \text{column}\}$

Oriented Partition An oriented partition $op = (p, o)$ is a tuple of a partition p and it's orientation o

Layout of a Table A Layout l_t of a Table t is a set of oriented partitions of the table with disjoint attributes. The union of the attributes of the partitions is equal to the set of attributes of t

Layout of a Schema The Layout $l_s = (op_1, op_2, ...)$ of a Schema s is the union of a set of layouts, on for each of the tables in s. When convenient we will write $o_{l_s}(p)$ to denote the orientation of the partition p in the layout l_s.

Unpartitioned Layout of a Schema The Unpartitioned Layout ul_s of a Schema s is a layout of the schema in which every table is divided into exactly one oriented partition, i.e., $|ul_s| = |s|$. When convenient we will write $o_{ul_s}(t)$ to denote the orientation of the table t in the layout ul_s.

Query A query $q = (string_{SQL}, w)$ is a tuple of query in SQL notation and an integer w that represents the relative frequency at which the query is executed

Workload A workload w is a set of queries

Costs The costs $c(w, l_s)$ of a workload w on a layout l_s is the estimated costs of the compiled queries as defined in Section 3.1

Costs Induced on a Table The costs $c(t)$ of a workload w on a table t is the estimated costs of the compiled queries that are induced by access patterns that are executed on this table. $c_{row}(w, t)$ denotes the induced costs if the table is stored row oriented. $c_{column}(w, t)$ denotes the induced costs if the table is stored column oriented. $c_{temp}(w, l_s)$ denotes the costs that are induced on temporary tables. This assumes that the costs that are induced on one table are independent of the costs induced on another. In the following section we will discuss this assumption.

3.2.2 Independence of Relation Orientation

When determining the optimal partitioning, OBP assumes that the partitioning of one relation has no influence on the optimal orientation of another [18]. As we will illustrate in the following, this assumption does not hold in general.

Obviously, the access pattern that is performed on one relation when evaluating a query is not influenced by the orientation of others (see Table 3.1). The costs of the query may however change due to the way the cache is divided between access patterns. The costs of a random traversal (`r_trav`) and the costs of a repetitive random access (`rr_acc`) are influenced by the available cache capacity (see their definitions in Section 3.1.3 and Section 3.1.3). The available cache capacity, however, does change if the pattern is executed concurrently to another pattern with a footprint greater than 1[5]. Patterns that potentially have a footprint greater than one are `r_trav` and `rr_acc`.

According Table 3.1, multiple `r_trav`s or `rr_acc`s are only executed concurrently when materializing tuples from their IDs in an LM-Columnstore for queries with more than two input relations. This can only happen if a query contains more than one join. For queries with more than one join, the orientation of one relation may influence the optimal orientation of another and neither OOBP nor the Simplex Based Layouting can ensure optimality. For this thesis we will exclude queries with more than one join. Even though a limitation in general, for an Operational Reporting workload and especially our use case (see Section 5.3) we consider this case to be of little importance.

[5]The available cache capacity does also change in this case but only to a very small degree, which we will neglect

3.2.3 Unpartitioned Layouting

Finding the optimal unpartitioned layout for a given schema and workload can be regarded as an optimization problem. Optimization problems have been studied extensively and solutions exist for several classes of optimization problems [61]. A class that has been researched particularly well are linear optimization problems. Several algorithms have been proposed (see [61] for an overview) with different complexity. In this section, we will show how the problem of optimal unpartitioned layouting can be solved using methods of linear programming.

Linear Programming

Finding the optimal solution to a linear problem, i.e., a set of linear inequalities and a linear objective function is called *linear programming*. A linear program is formally defined by:

- The set of constraints C, i.e., inequalities of first order polynomials that have to be fulfilled by the solution of the problem. For the layouting problem, the set of constrains is very simple: Every relation can be stored in either row- or column-orientation. The number of inequalities is denoted with n.
- The objective function $f(\vec{x})$ that is defined by a first order polynomial. For the layouting problem this is the cost of a layout.

The dimension d of the problem is the number of independent variables that occur in the inequalities. For the layouting problem this is the number of relations. The feasible region R is the set of elements of the solution space that fulfill all equations.

The linear problem is to find an element *opt* in the solution space at which the objective function is, w.l.o.g., maximal (see Equation 3.36 for a formal definition).

$$\forall s \in R : f(s) \leq f(opt) \tag{3.36}$$

In the following, we will show that the cost function of the layouting problem is a linear function of it's solution space.

Is the Cost Function linear?

Since the target layout is unpartitioned, every relation can only have one orientation. A layout can therefore be represented as a vector, each dimension representing the orientation of one table (the mapping of a relation to it's dimension in the vector could, e.g., be done lexicographically). Since the schema s has $|s|$ relations, the layout vector ul_s has $|s|$ dimensions. To construct a linear cost function we map the solution space to $\Pi_{i=1}^{|n|}\{0,1\}$[6] by applying equation 3.37 to each dimension of the vector.

$$o_{binary}(t) = \begin{cases} 1 & \text{if } o(t) = \text{Row} \\ 0 & \text{if } o(t) = \text{Column} \end{cases} \tag{3.37}$$

Since we assume that the costs of access patterns on one relation do not influence the costs on another (see Section 3.2.2), the overall costs can be calculated as the sum of the costs that are induced on each relation and the costs induced on temporary relations. Using the mapping defined in Equation 3.37, the cost function for a workload can, therefore, be described by Equation 3.38. It is obvious that this function is a linear function of the representation and thus, the optimal representation can be found using techniques of linear programming.

$$c(w, l_s) = \sum_{t \in l_s} \left(o_{binary}(t) \cdot c_{row}(w, w) + (1 - o_{binary}(t)) \cdot c_{column}(w, t)\right) + c_{temp}(w, l_s) \tag{3.38}$$

[6] Π denotes the Karthesian Product of all operands

The simplex algorithm

An optimal solution of a linear problem can always be found at a vertex of the feasible region [61]. Therefore, it suffices to check the value of the objective function at every vertex and pick the optimal to solve the linear problem. The number of vertices, however, is in $\Omega\left(n^{\frac{d}{2}}\right)$ [61]. A large d is not uncommon in practice (remember, d is the number of variables/relations of the problem). An algorithm is needed that finds the optimum without considering all vertices. One algorithm that does this is the simplex method.

The Idea of the simplex method is fairly intuitive: a vertex of the feasible regions is picked (randomly or deterministically), and the value of the objective function at this vertex is calculated. The value at all adjacent vertices is also calculated. The vertex with the largest decrease/increase of the objective value is selected as vertex for the next iteration. If more than one value are optimal both have to be checked, the algorithm branches. When all unvisited, adjacent vertices have objective values less than the current one, the optimum has been found and the algorithm terminates. This relies on the assumption that the feasible region is convex which holds for linear programs.

Complexity Examples exist that show that in the worst case the simplex algorithm still has to visit all vertices of the solution space [62]. Thus, in the worst-case the complexity of the simplex method grows with the number of vertices. In practice it has been shown to "usually take polynomial time" [63][7].

It shall be noted that there are other methods that are theoretically faster than the simplex-method (see ,e.g., [64]). For the layouting problem, however, it does have an attractive property: it can be "jumpstarted". If the problem (the workload), changes only slightly, the former solution can be picked as initial vertex. The former optimum is expected to be close to the new optimum which would reduce the number of simplex iterations. An example would be a single query that is added to the workload. Only the relations that are accessed by that query could potentially change their optimal layout. Thus, the number of needed simplex steps is in the worst case the number of relations that are accessed by the new query. Compared to the worst case of a simplex run with an arbitrary start point ($\Omega\left(n^{\frac{d}{2}}\right)$) this improves the worst case running time. Changes of the workload are, however, not covered in this thesis.

3.2.4 Vertically Partitioned Layouting

Optimal Binary Partitioning [18] aims at finding the optimal partitioning of a given schema based on a set of transactions/queries.

Reasonable Cuts

A partitioning of the schema such that all attributes in a partition are accessed in one transaction[8] is called a *reasonable cut*. Chu and Ieong prove that, if all values of an attribute are accessed uniformly, an optimal partitioning is found at such a reasonable cut. For n transactions that query a relation, the number of reasonable cuts within the relation is $2^n - 1$. To avoid an exhaustive search for the optimal reasonable cut, Chu and Ieong propose an algorithm that is based on the *branch and bound method* [65].

[7] For the layouting problem the worst case complexity is even bound by the number of tables due to the independence of the table orientations

[8] Note that this does not mean that all attributes that are accessed in a transaction are stored in one partition

Optimal Binary Partitioning

Chu and Ieong define the search space as the set of all reasonable cuts. The (binary) search tree is constructed from the transactions (see Figure 3.9). Each node represents one transaction (all nodes with the same distance from the root represent the same transaction). The children of every node represent the reasonable cut if the transaction either included or excluded. A solution to the problem is a path from the root to a leaf the path defines the inclusion/exclusion for each of the transactions.

To determine the optimal reasonable cut, the search tree is traversed bottom up. For each traversed node, the costs are estimated. The costs of the right child of a node do not have to be calculated because the reasonable cut does not change be excluding a transaction. If the estimated costs of including a transaction are equal or greater than the costs of excluding it the respective subtree is pruned.

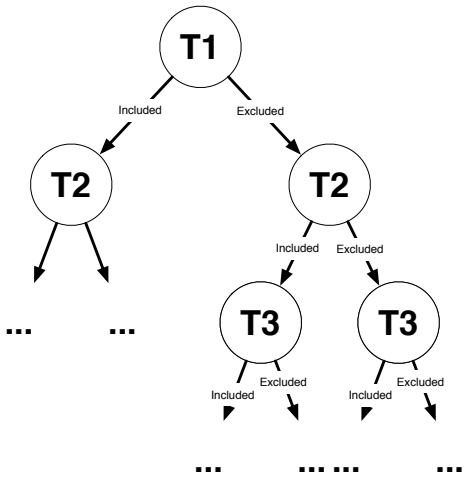


Figure 3.9: The search tree for OBP

In [18], the costs of a reasonable cut are evaluated using a very simple cost function. We used the extended generic cost model to evaluate the costs of a partitioning. OBP requires a cost function that does not increase if a reasonable cut is expanded, i.e., an additional transaction is included. Since partitioning only reduces the width $R.w$ of a data region, which can only reduce the costs, this requirement is fulfilled by the extended generic cost model.

Every time a relation is partitioned, each of the two resulting partitions can be stored either row or column oriented. Thus, four different options exist for the orientation of the resulting partitions. The partitioning may only be beneficial for one of the options, thus, all have to be considered. Since the number of options is constant this does not increase the theoretical complexity of the algorithm. The optimal orientation of every partition is determined by the generic cost model.

Complexity of the Optimal Binary Partitioning Algorithm

OBP has been proven to have a worst case run time of $\mathcal{O}(2^n)$ [9], n being the number of transactions. In practical results the authors report an average run time of $O\left(2^{0.6n}\right)$.

To further reduce the run time, a minimal benefit of including a transaction in the reasonable cut can be defined. Unless including a transaction yields an estimated benefit above this threshold the subtree is pruned. The threshold spoils the optimality but also reduces the average run time (potentially down to $O(n)$).

Binary Partitioning improved (BPi) is an alternative algorithm proposed by the authors of OBP, further trading layouting costs for optimality. BPi could be used for very large schemas or when a low optimization effort is crucial. It was, however, not investigated during the work for this thesis.

Extensions of the Optimal Binary Partitioning Algorithm

As mentioned, Chu and Ieong assume that all values of an attribute are accessed uniformly in a query. This assumption does not generally hold true: As shown in Section 3.1.4, attributes of a relation that are accessed within one transaction may be accessed in a different pattern. Some may be accessed in a s_trav and some in a s_trav_cr which could make different layouts feasible for each attribute.

To illustrate this consider the example in Figure 3.10, which is identical to the example in Figure 3.7 on Page 37. The evaluation of the condition of the query is characterized by a full sequential traversal. The reconstruction, however, is performed using a traversal with conditional reads. The selectivity of the

[9] note that it is independent of the number of relations

Schema:

Table	Attribute	Cardinality	Data type
ADRC	ID[a]	5000[b]	Word
ADRC	NAME	5000	Word
ADRC	KUNNR	5000	Word

Query:

```
select KUNNR from ADRC where NAME = $1;
```

Most appropriate partitioning: (`ADRC.NAME`), (`ADRC.KUNNR`)

[a]The ID is the position of a tuple – an implicit attribute, i.e., not physically stored
[b]The cardinality of the ID column is the number of stored tuples

Figure 3.10: A case for extended reasonable cuts

s_trav_cr is expected to be low which could greatly reduce the number of necessary reconstructions when storing `KUNNR` and `ADRC` in separate partitions. As presented in [18], however, OBP would not consider partitioning along that cut because it only differentiates accessed and unaccessed attributes. To remove this limitation, the definition of a reasonable cut was extended.

An extended reasonable cut is a partitioning of the schema such that all attributes in a partition may be accessed in an equal pattern in at least one operator of at least one query. Attributes that may be accessed in an equal pattern in at least one operator of at least one query are called an *extended transaction*. According to Table 3.1, different access patterns within one operator only occur in selections, projection+selections and joins. In all of these cases, the evaluation of a condition may expose a different pattern than the evaluation of another condition or the reconstruction of the tuples. Therefore, the overall costs could be reduced by storing them in separate partitions.

Depending on the selectivity of the conditions it may also be beneficial to consider a cut where the attributes that are used in the conditions are stored in a partition together. If, e.g., the selectivity of the condition in Figure 3.10 was 1 (all values match) an optimal decision would be to place `NAME` and `KUNNR` in a partition together. This is why all combinations of the partitions have to be checked as well.

This extension accounts for different access patterns in one transaction. It does, however, increase the number of transactions in the worst case exponentially with the sum of the number of conditions in every query. This in turn increases the worst case run time.

Chapter 4

Implementation of Spades - an Automatic Data Layouter

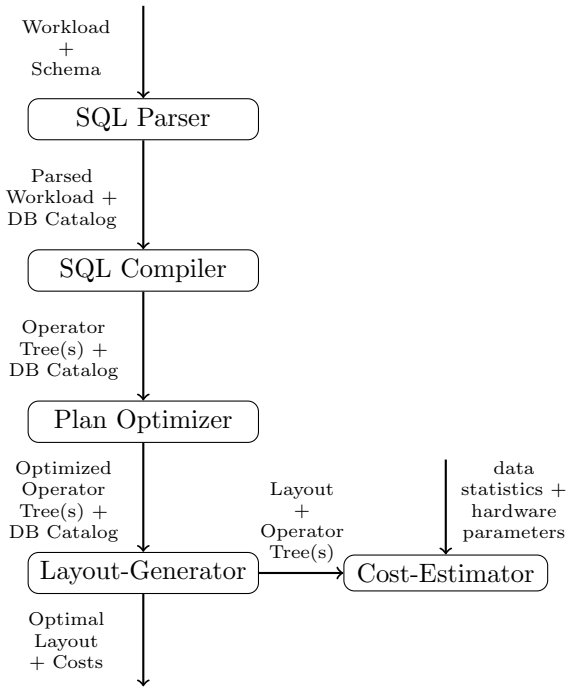


Figure 4.1: The Architecture of Spades

The findings from the previous chapters were implemented in a tool called *Spades*. Spades was designed to be an automated tool to propose an analytically optimal layout for a given schema, workload and hardware configuration. Figure 4.1 shows the overall architecture of *Spades*. In this chapter the implementation of the components is described and reasons for design decision are given.

4.1 Requirements

Spades was designed to be usable as either a standalone tool, or as layouting component that can be integrated into a hybrid DBMSs. The primary platform for an integration is the *HYRISE* DBMS [49].

HYRISE Due to the early stage of the *HYRISE* development efforts, the implementations of the relational operators are under constant development. To keep *Spades*' model in sync with *HYRISE*' implementation, it was required that the access patterns of the operators can easily be changed to reflect changes in the implementation of the operators.

HYRISE is implemented in C++ and, consequently, *Spades* was implemented in C++ as well to allow an easy integration.

***Spades*' Components** *HYRISE* does not yet include an SQL-Compiler. To support SQL in *Spades* we, therefore, decided to define a minimal set of relational operators and implement an **SQL-compiler** to compiles a subset of SQL to our relational algebra. We plan to integrate the compiler as a component into the *HYRISE* system.

Based on the database schema and the relational operator trees, the **Layout-Generator** generates one or many, partitioned or unpartitioned layouts. The costs of each generated layout is evaluated and the result used to iteratively generate better layouts until the optimal layout is found.

The costs of a layout are evaluated using the **Cost-Estimator**. It takes a layout, the queries (as operator trees) and the cardinality of every attribute (see Section 3.1.2) as input and generates an integer value that represents the overall costs of a workload on a layout.

A detailed description of each of the components will be given in the following.

4.2 The SQL Compiler

To make *Spades* capable of processing SQL queries, an SQL compiler component was needed. To keep the integration effort to a minimum a compiler that is implemented in C++ was required. Our first approach to this problem was to investigate into compilers of existing DBMSs.

4.2.1 Existing Compilers

Finding a suitable compiler that is freely available turned out to be difficult. There are some open-source DBMS that are implemented in C++, but all the considered options had at least one of three major problems: they were too complex, too tightly integrated into the rest of the DBMS or simply too immature.

Too complex The SQL compilers that are part of the established open source databases, like MySQL, PostgreSQL, Firebird or Ingres, are very complex. They do compile to an intermediate language that is sophisticated enough to allow Data Manipulation Queries, Data Definition Queries as well as custom extensions for the definition of indices, views, etc.. Dealing with this rich intermediate language would have meant considerable additional work without real benefit.

Too tightly integrated Some of the simpler DBMS, like Drizzle and SQLite, contain compilers that are simple enough to be used without too much effort. The simplicity, however, comes with the problem of limited generality. They are very tightly integrated into the rest of the DBMS to support, e.g., multi user operation. A big part of the DBMS would have to be *mocked up* to get the compiler working.

Too immature The third problem, that seems to be mainly a problem of academic DBMS is maturity. SystemJ, an academic DBMS implementation was considered. Since it was implemented in a semester project by students, the quality of the implementation was a serious issue. The ammount of work to even compile the source code was unacceptable.

Since none of the available options proved adequate, an SQL compiler was implemented from scratch.

4.2.2 Spades' SQL Compiler

The compiler consists of two components: one component, that reads the schema into the database schema catalog and another, that compiles the SQL queries on that schema.

The Catalog

To store the database schema, a very simple `catalog` was implemented. The catalog is implemented as a singleton [15] map from a `qualified_attribute` to a pair of type (as string) and the cardinality (as integer). The `qualified_attribute` is a unique pair of `strings` (the relation and the attribute) that identifies an attribute in a relational database.

The schema is read from a file like the one shown in Figure 1. It holds a qualified attribute, it's type and extension (the number of unique values of the attribute) per line. The cardinality (number of stored tuples) of the relation is the extension of the `id`-column of the relation.

```
adrc.id          int 500
adrc.addrnumber  int 500
adrc.kunnr       int 500
```

Listing 1: Sample Schema Input

compile sql to relational algebra (parsed queries)

- A scan+projection of the first requested table is constructed for. It is stored as the root of the operator tree
- For each remaining table
  - A join with no conditions (carthesian product) is created with the former root of the operator-tree as left and a scan+project of the table as right child. It is stored as the root of the operator tree.
- For each condition
  - A new selection is created as new root of the tree and the former root as child
- If the number of group-by attributes is greater than 0
  - One group_by is added as new root

Listing 2: Pseudocode of the SQL Compiler

The Compiler

Spades' SQL parser was implemented using the standard compiler generation tools *GNU Flex* and *GNU Yacc*. The input grammar for *Yacc* is shown in Appendix B.1, the *Flex* token definitions in Appendix B.2. The implementation is not given in Appendix B.2 but is planed to be released as part of the *HYRISE* implementation. The grammar captures `INSERT` as well as simple `SELECT` queries. `GROUP BY`s, `ORDER BY`s and arbitrary conditions (test of equality to a constant or another requested attribute) are supported.

The parser returns a parsed representation of the query, not the finished relational operator tree. Though possible, this would have increased the amount of code in the parser which is harder to maintain. The parsed query is simply an object that contains all relevant information about the entered query. The `parsed_query` class contains

- all requested attributes as strings (not necessarily qualified yet)
- all requested tables as strings
- all parsed conditions as triple of attribute, comparison and attribute/constant
- all attributes that the result is grouped by as strings.

The operator tree is constructed in the next step by a factory [15] , the `sqlcompiler`, from the parsed query (for a class diagram of the relational algebra see appendix B.3). The tree is constructed from the leaves to the root according to the code in figure 2.

The evaluation of this initial operator tree would, although correct, be very inefficient. Especially the large intermediate results that originate from the products are a major shortcoming. As an example, consider the operator tree in Figure 4.2, which is the unoptimized version of the tree seen in Figure 3.1 in Section 3.1. The selection operators are executed very late in the evaluation, the size of intermediate results may, therefore, be unreasonably high. To avoid this, a rule based optimizer was implemented. The following rules are applied repeatedly until none of the rules is applicable.

- All selections are pushed as far as possible to the leafs of the operator tree by repeatedly exchanging them with their child. If the child is a join, the selection is stored as a child of the join. The child of the join that exports the attributes that are used in the selection becomes it's new child. A selection cannot be pushed further if the attributes that are exported by any of its grandchild do not suffice to evaluate the conditions of the selection.
- If a selection ends up directly above a projection they are merged into a combined project_selection to avoid unnecessary materialization of tuples or attributes. This is done until no more selections are directly above a projection.
- If a selection ends up directly above a join it is merged into the join by adding it's conditions as join-condition. This is done until no more selections are directly above a join.

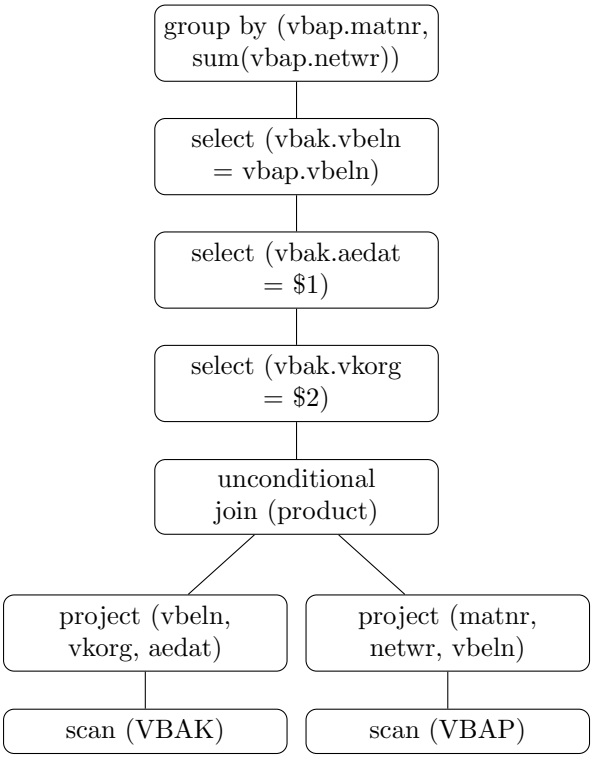


Figure 4.2: An Example of a Relational Operator-Tree before optimization

After this optimization the operator tree, though not optimal in every case, avoids the most important mistakes in query evaluation. Adding cost-based optimization (potentially using Manegold's model [2]) is considered future work. Based on the operator tress of the queries, the costs of their execution on a layout can be calculated.

4.3 The Cost Calculator

Calculating the costs of a layout consists of two steps. In the first step, the access pattern of each query on the schema is calculated and weighted to construct the costs function. In the second step, the costs of the overall workload is estimated.

4.3.1 Constructing the Cost Function

The access pattern that is performed to evaluate a query can be derived from the relational operator tree and the layout. The construction of the access pattern from the operator tree is done very much like a DBMS would evaluate the query: by traversing the operator tree from the leaves to the root and mapping each operator to an access pattern as defined in Table 3.1 in Section 3.1.5 . These are concatenated in the order the operators would be executed. Multiple children of joins are executed sequentially. This is done in a factory called `cost_function_calculator`.

Manegold's cost model (see Section 3.1.3) has not been implemented completely. The goal of the described implementation is the modeling of the query execution in *HYRISE*, not providing a generic implementation of Manegold's model. Therefore, only the necessary patterns of Manegold's model were implemented. Furthermore, the parameters for their instantiation are sometimes less generic than they could be (e.g. `random_traversal` doesn't allow gaps between the accessed values because this pattern doesn't occur in the current *HYRISE* implementation). The implementation will be discussed in the following.

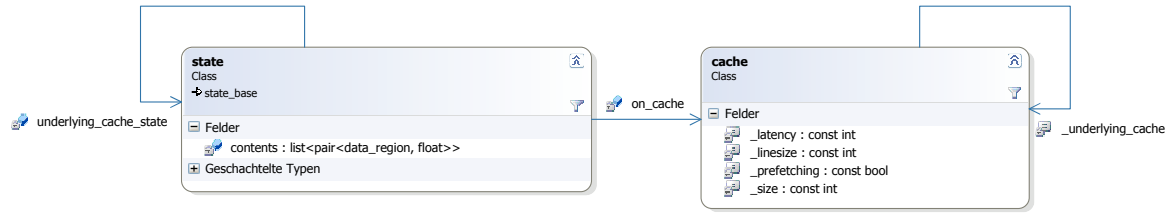


Figure 4.3: The UML diagram of the classes modeling the cache hierarchy and it's state

4.3.2 Evaluation of the Cost Function

Representing Caches and their State

Figure 4.3 shows the model for the cache hierarchy and it's state which are used as parameters of the atomic access patterns

Cache Every layer of cache is defined by it's parameters (`size`, `linesize` and `latency`). The memory hierarchy is modeled through a reference to the `underlying_cache` that every `cache` has. It also contains a boolean that specifies if this cache does `prefetching` (see Section 2.2.3).

State As defined in the model, access patterns may leave the cache in a state and/or benefit from state left by another pattern. This state is implemented using a `list` of `pairs` of `data_regions` and a `float` to represent the percentage of the region that is held in the cache. To represent the state of a cache, an `stl::map` could not be used because it relies on a sorted dictionary for lookups. Since data regions do not have an inherent order, defining and implementing an artificial ordering seemed unnecessary. The uniqueness of the `state`-keys is asserted manually. To represent the state of all caches in one state instance a reference to the state of the underlying cache is held as well.

Cost Function

Figure 4.4 shows an UML diagram of the data structures that were used to implement Manegold's model. The evaluation of the costs is built around the class `cost_function`. It is an abstract class that defines the function `costs()`. `costs()` returns an integer representing the estimated costs as defined in Section 3.1.3. The implementation of Manegold's cost model is built around the class `access_pattern` that is derived from `cost_function`. In addition to the costs, it also defines a method to get the resulting cache state. As illustrated in Section 3.1.3 this is important to estimate the number of cache misses induced by a `sequential_execution`.

Atomic Access Patterns

On the left side of Figure 4.4 are the classes that implement the atomic access patterns defined in Manegold's model. Since all of them are performed on an input data region, each contains a reference to an instance of `data_region`.

Data_region Manegold's model evaluates the costs of an atomic access pattern based on the area of memory it operates on (see Section 3.1.3). In *spades*, an area of memory is represented by an instance of `data_region`. It holds a list of `qualified_attributes` and an extension (the number of tuples). To test for equality of `data_regions` we test for mutual inclusion of the list of `qualified_attributes` and equality of the extension. This can lead to illogical identity[1] because multiple `data_regions` may be defined with the same attributes and extensions but represent different areas in memory. In *Spades*, this does not occur due to the way `data_regions` are used:

[1] two class instances are considered equal even though they represent different real-world objects

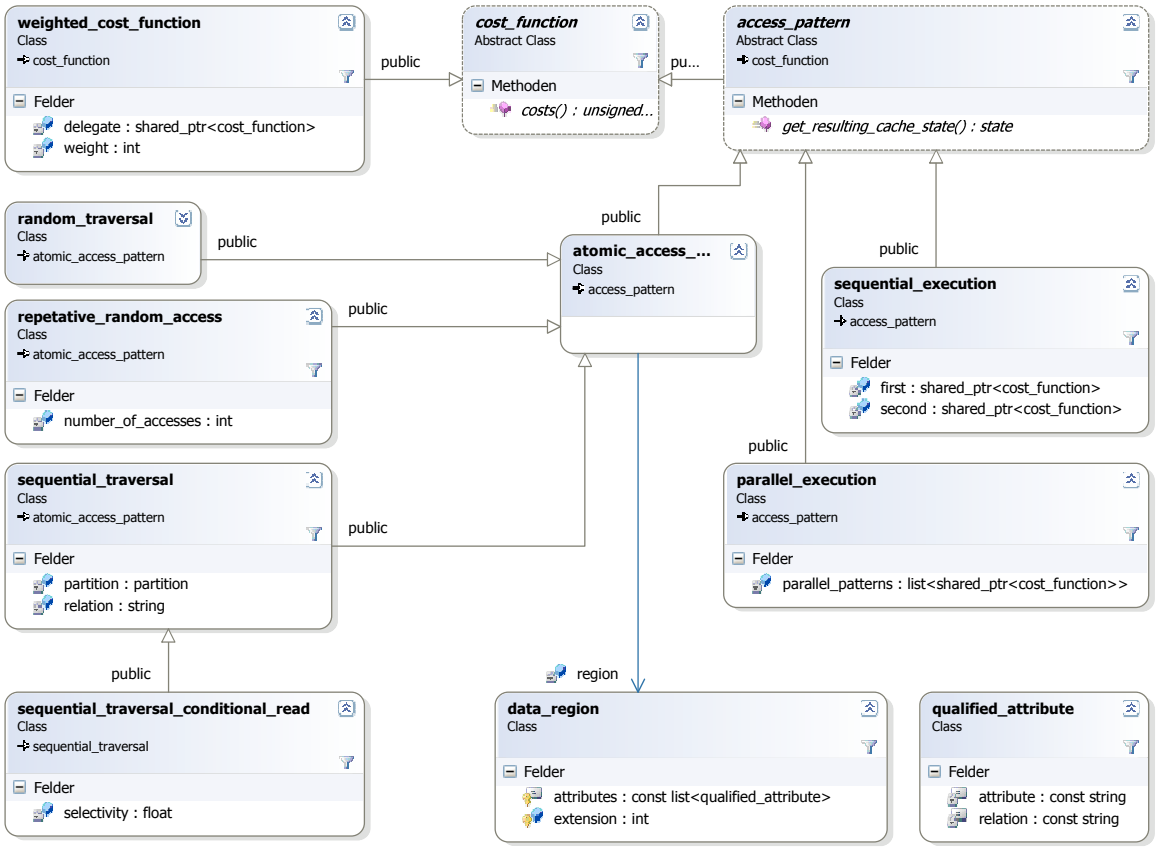


Figure 4.4: The UML Diagram of the Classes Related to the Cost Function

- as an attribute of atomic access patterns, used merely to hold parameters that are relevant for the estimation of the number of cache misses, and

- as a part of the state that is left by an access pattern. Since the state is only carried forward one sequential execution step, an illogical identity can only occur if two concurrent patterns leave seemingly identical states. According to Table 3.1 in Section 3.1.5 this only happens if input and output of an operator are indeed identical (e.g. a selection with selectivity 1). In the `state` object, the data region will only be stored once and the following operator will (correctly) benefit from the left state once.

The problem of illogical identity of `data_regions`, even though a problem in general, does not affect the correctness of our implementation. A generic solution would be to assign an identifier to every data region and use it to test for identity. This would, however, increase the complexity of the `cost_function_calculator` because it would have to make sure that the output of an access pattern is identical to the appropriate input of another access pattern. Without the identifier, it suffices that every operator has a reference to it's input data region since the output data region can be derived from it.

Random_traversal The estimation of the costs of a random traversal in Manegold's model is dependent of the gap between the accessed tuples. According to Table 3.1 in Section 3.1.5, `random_traversal` is only used with a gap of width 0. Since all values of the tuple are accessed, `random_traversal` does not contain a field for the accessed values of a tuple.

Repetative_random_access To estimate the costs of the *rr_acc*, the number of accesses is needed. For the same reasons as `random_traversal`, `repetative_random_access` does not contain an attribute for the accessed attributes.

Sequential_traversal the costs of a sequential traversal in the model are estimated depending on the gap between the accessed value. In *Spades*, this only happens if values are accessed from a stored relation. Thus, when the gap between tuples is greater 0, the `sequential_traversal` is instantiated with a `string` that defines the accessed `relation` and the list of accesses `attributes`. The extension of the `data_region` is the same as the extension of the relation that is traversed. The parameters of the relation are retrieved when evaluating the `costs()` using the `catalog`. An analogue case is the traversal of a `partition` (a partition is a typedef of a `set` of `qualified_attributes`).

Sequential_traversal_conditional_read Our main addition to the model is implemented as a class that is derived from `sequential_traversal`. This ensures a loose coupling of this extension to the model. Thus, `s_trav_cr` is considered a special case of a s_trav. The difference is the `selectivity` and an implementation of the `costs()`-function according to section 3.1.4.

Complex Access Patterns

As defined in the cost model, our implementation contains two complex access patterns.

Sequential_execution The sequential execution is a pair of `access_patterns`. A pair suffices because the sequential execution operator of the access pattern algebra is commutative (equation 4.1).

$$\oplus(ABC) = ((A \oplus B) \oplus C) = (A \oplus (B \oplus C)) \tag{4.1}$$

Thus the sequential execution of n access patterns can be modeled using n nested `sequential_execution` patterns.

Parallel_execution In contrast to the `sequential_execution` pattern, a `parallel_execution` is not commutative (equation 4.2).

$$\odot(ABC) \neq ((A \odot B) \odot C) \tag{4.2}$$

This is due to the way the cache is split between concurrently executed access patterns: it is divided using the footprint of the nested patterns. To illustrate this, consider the case that A, B, C are all instances of `s_trav` working on the regions a, b, c. The footprint of a *s_trav* is, as defined in Section 3.1.3, 1 (thus, Equation 4.3 holds).

$$f(A) = f(B) = f(C) = 1 \tag{4.3}$$

The state that results from the right side of Equation 4.2 can be seen in Equation 4.4.

$$\mathbf{S}((A \odot B) \odot C) = \left\{ \left(a, \frac{1}{4}\mathbf{C}_i\right), \left(b, \frac{1}{4}\mathbf{C}_i\right), \left(c, \frac{1}{2}\mathbf{C}_i\right) \right\} \tag{4.4}$$

This would mean that some access patterns are using a greater portion of the cache than others even though they have the same footprint. The expected and correct state would be the one seen in Equation 4.5.

$$\mathbf{S}(\odot(ABC)) = \left\{ \left(a, \frac{1}{3}\mathbf{C}_i\right), \left(b, \frac{1}{3}\mathbf{C}_i\right), \left(c, \frac{1}{3}\mathbf{C}_i\right) \right\} \tag{4.5}$$

That is why `parallel_execution` holds a reference to a list of `access_patterns` that are executed in parallel and the cache is divided correctly amongst them.

Special Cost Functions

To support the weighting of queries according to their relative frequency (see Section 3.2.1), an additional `cost_function` was implemented as a wrapper [15]. A `weighted_cost_function` merely contains the weight and a delegate `cost_function` that would generally be an `access_pattern`.

4.4 The Layouter

As described in Section 3.2, two layouting algorithms were implemented. They are not data centric, which is why their data model is not shown here. Their implementation will be described in the following.

4.4.1 The Simplex Layouter

The simplex based layouting algorithm is very simple[2]. It starts with an arbitrary layout, toggles the orientation of each of the relations (from row to column or vice versa) and selects the layout with the minimal costs for the next iteration. This is repeated until no improvement can be made by toggling the orientation of any of the relations. Figure 3 shows the implementation in pseudocode.

4.4.2 The Partitioned Layouter

The partitioned layouter is based on the extended Optimal Binary Partitioning algorithm (see Section 3.2.4). To calculate the optimal partitioning, the layouter performs two steps. In step one the extended transactions are calculated (see Section 3.2.4), which are used to find the optimal layout in step two.

The implementation of step one is straight forward (the pseudocode is shown in Listing 4). In step two, each table of the schema is recursively partitioned using the extended transactions (see the implementation in pseudocode in Listing 6). As discussed in Section 3.2.2, we assume that the costs that are induced by an access pattern on one relation do not affect the costs induced on another relation. Under this assumption, the optimal partitioning of each relation can be calculated separately [18]. Violation of this assumption may spoil the optimality of the solution.

The recursion starts with an unpartitioned layout (the *layout before cut* spans the whole relation). In each recursion step, one of the *remaining transactions* is a possible cut for the *layout before cut*. Since the cut may only be beneficial for one of the possible oriented layouts, the costs of the cut have to be evaluated for each of the oriented layouts (see Listing 5 for the pseudocode to calculate the oriented layouts for a partitioning). If the cut yields an improvement, the algorithm branches into one branch that includes the transaction in the cut and one that does not. If the cut did not yield an improvement, the branch that includes the cut is pruned. The best resulting partitioning and it's costs are returned.

Note that to calculate the costs of a partitioning, the complete workload has to be considered, not just the transaction that is currently considered for the cut. Whilst the overall costs may take longer to evaluate, their evaluation is necessary since improvements for one transaction may result in performance degradation for another.

Based these methods for database schema partitioning Manegold's generic cost model, *Spades* can automatically propose an analytically optimal, partitioned or unpartitioned, layout for a given workload on a database and hardware configuration. In the following chapter, the accuracy of the model will be evaluated and the performance benefit of the proposed layout over all-row and all-column layouts evaluated.

[2]The method is named after the geometrical shape, not the fact that the method is simple

simplex (schema, workload)

- Start with any layout (e.g. all row), store it as *current optimum*
- calculate the cost function for the layout and estimate the costs, store it as *current optimal costs*
- set *further optimization is possible* to true
- while *further optimization is possible*
  - store the *current optimal costs* as *best adjacent costs*
  - for every *relation* in the *schema*
    - store the *current optimum* as *adjacent layout*
    - toggle the orientation of the *relation* in the *adjacent layout*
    - calculate and evaluate the cost function of the workload on the *adjacent layout*
    - if the costs are less than the *current optimal costs*
      - store the *adjacent layout* as *best adjacent layout*
      - store the costs as *best adjacent costs*
  - if the *best adjacent costs* are less than the current optimal costs
    - store the *best adjacent layout* as *current optimum*
    - store the *best adjacent costs* as *current optimal costs*
  - else
    - set *further optimization is possible* to false

Listing 3: Simplex Algorithm to Calculate the Optial Unpartitioned Layout

Calculate Extended Transactions (schema, workload)

- the *extended transactions* is an empty set of partitions (sets of attributes)
- for each *query* in the workload
  - for each *operator* in the query
    - if the *operator* is a join, a selection or a projection+selection
      - for each *condition* in the operator
        add the *attributes used in the condition* as a transaction to the *split transactions*
      - add *all exported attributes* that are not used in any condition as a partition to the set of *split transactions*
      - add all combinations of the *split transactions* to the *extended transactions*
    - else
      - add *all attributes* that are accessed by the operator to the *extended transactions*

Listing 4: Pseudocode to calculate the Extended Transactions

calculate the possible oriented partitionings (partitioning)

- store an empty set of oriented partitionings as *resulting oriented partitioning*
- add the first partition with the orientation columns of the *partitioning* to the *resulting oriented partitioning*
- if the number of attributes of the first partition of the *partitioning* is greater than 1
  - add the first partition with the orientation rows of the *partitioning* to the *resulting oriented partitioning*
- for each but the first of *the partition*s in the *partitioning*
  - for each *oriented partitioning* in the *resulting oriented partitioning*
    - add the union of the *oriented partitioning* and *the partition* with the orientation columns to the *resulting oriented partitioning*
    - if the number of attributes of *the partition* is greater than 1
      - add the union of the *oriented partitioning* and *the partition* with the orientation rows to the *resulting oriented partitioning*

Listing 5: Calculating the Possible Oriented Partitionings for a Partitioning

obp(queries, layout before cut, costs before cut, remaining transactions)

- store the first of the *remaining transactions* as *selected transaction*
- remove the *selected transaction* from the *remaining transactions*
- store an empty set of patitions as *new partitioning*
- for every *existing partition* in the *layout before cut*
  - add all attributes that are contained in the *selected transaction* and the *existing partition* as a new partition to the *new partitioning*
  - add all attributes that are contained in the *selected transaction* and not the *existing partition* as a new partition to the *new partitioning*
- store the *costs before cut* as *best costs so far*
- store the *layout before cut* as *best layout so far*
- calculate the *possible oriented partitionings* from the *new partitioning*
- for each of the *possible oriented partitionings*
  - calculate *the costs* of the *oriented partitioning*
  - of *the costs* are less than the *best costs so far*
    - store the *oriented partitioning* as *best layout so far*
    - store *the costs* as *best costs so far*
- store the *best costs so far* as *best costs in this branch*
- store the *best layout so far* as *best layout in this branch*
- if the *best costs so far* are less than *costs before cut*
  - call obp with *queries*, *best layout so far*, *best costs so far*, *remaining transactions* and store the result as *best of left branch*
  - if the cost of the *best of left branch* are less than the *best costs so far*
    - store the costs of the *best of left branch* as *best costs in this branch* and the layout of *best of left branch* as *best layout in this branch*
  - call obp with *queries*, *layout before cut*, *costs before cut*, *remaining transactions* and store the result as *best of right branch*
  - if the cost of the *best of right branch* are less than the *best costs so far*
    - store the costs of the *best of right branch* as *best costs in this branch* and the layout of *best of right branch* as *best layout in this branch*
- else
  - call obp with *queries*, *layout before cut*, *costs before cut*, *remaining transactions* and store the result as *best of right branch*
  - if the cost of the *best of right branch* are less than the *best costs so far*
    - store the costs of the *best of right branch* as *best costs in this branch* and the layout of *best of right branch* as *best layout in this branch*
- return *best partitioning in this branch* and *best costs in this branch*

Listing 6: Oriented Optimal Binary Partitioning in Pseudocode

Chapter 5

Evaluation

In this section, we want to evaluate the accuracy of our cost model in simple cases and illustrate the benefit of our approach in a mixed workload scenario. We will start by giving a brief introduction to *performance counters* which we used to measure CPU cycles as well as misses induced on some the cache layers. Following that, we want to evaluate the accuracy of the cost model in a some common cases. In the end of this section we present the results of our methods for hybrid data layouting in a complex case.

5.1 Performance Counters

To support the profiling of applications, most modern processors contain special registers called *Performance Counters*. These are registers in the CPU core that can be configured to count events that occur during the execution of program code. The Intel Core 2 Architecture Specification [1] gives an introduction into performance counters. Documentation on their configuration and usage, as well as a detailed description of their semantics on different platforms can be found in the Apple Shark User Guide [66].

All experiments were conducted a processor of the Intel Core 2 class. The Core 2 specification [1] describes a total of 116 different events that can be counted in five counters. Two of these are general purpose counters that can be configured and three are special counters that can only count a predefined events. The dedicated register that is most interesting to us counts the event

CPU_CLK_UNHALTED.REF, the total number of elapsed cycles at the reference CPU clock frequency (we always ran the CPU at the reference clock frequency).

The two general purpose registers can be used to count various events. The events we counted are related to the data cache:

DCU_LINES_IN, the number of cache lines that were loaded into the Level 1 Data Cache.

L2_LINES_IN, the number of cache lines that were loaded into the Level 2 Cache. This includes cache lines that contain data as well as those that contain instructions.

The Performance Application Programming Interface

The complexity of programming the performance counters as well as the fact that every processor architecture supports different events has lead to several frameworks that try to unify access to the performance counters. Of these, we investigated into *Rabbit*[1], *PCL*[2] and *PAPI (Performance Application Programming Interface)*[3]. The later was used in our experiments. It supports a number of platforms and a unified set of events.

[1] available at `http://www.scl.ameslab.gov/Projects/Rabbit`
[2] available at `http://www.fz-juelich.de/jsc/PCL`
[3] available at `http://icl.cs.utk.edu/papi/`

The PAPI-Library allows to count some of the events that our cost-model predicts: Level 1 and Level 2 cache misses. Even though we are not really interested in the number of cache misses but in the spent CPU-cycles, the accuracy of the model for predicting the cache misses can be evaluated. We also used PAPI to measure our target metric: the time spend evaluating a query in CPU cycles.

Amongst the events that can be counted by PAPI are some that cannot be counted directly. It does, e.g., support counting the Level 2 Data Cache Misses (L2_DCM) which is calculated by subtracting the number of instruction cache lines (BUS_TRANS_IFETCH) from the total number of lines loaded into the Level 2 Cache (L2_LINES_IN). This has the drawback that counting the Level 2 Data Cache Misses "blocks" two performance counters. Since the Intel Core 2 processors only have two performance counters, counting any other event. Since the number cache lines containing instructions is expected to be low in our data centric case, we counted L2_LINES_IN and accepted the inaccuracy that comes with counting the cache lines containing instructions as well.

5.2 Cost Model Evaluation

The cost model was evaluated through a set of hand coded experiments. This eliminates the overhead of a full blown DBMS and allows us to evaluate the data access performance in isolation from the rest of the system. All experiments were conducted on an *IBM BladeCenter HS21 XM* with an Intel Xeon E5450 Processor (3 GHz) and 32 GB RAM.

5.2.1 Calibration of the Model

```
Calibrator v0.9e
(by Stefan.Manegold@cwi.nl, http://www.cwi.nl/~manegold/)
81985010 47530282340368 4096     16
81985fff 47530282344447 4096   4095
81986000 47530282344448 4096      0

MINTIME = 10000

...

CPU loop + L1 access:        1.00 ns =   3 cy
              ( delay:       0.00 ns =   0 cy )

caches:
level   size     linesize    miss-latency          replace-time
  1      32 KB    64 bytes     4.06 ns =  12 cy     4.06 ns =  12 cy
  2       6 MB    64 bytes   111.42 ns = 334 cy   111.67 ns = 335 cy

TLBs:
level #entries   pagesize   miss-latency
```

Listing 7: Output of the Calibrator

For an accurate prediction of the memory access costs, the properties of the various memory layers (latency, size and block size) are needed as input parameters to the cost model. The first step to predicting the costs of a workload is, therefore, the determination of the relevant parameters. Stefan Manegold developed the *Calibrator v0.9e*[4] for this. When executed, the Calibrator conducts experiments similar to the ones we used for the initial investigation into memory access time in Section 2.2.1.

After we disabled the Level 2-prefetching, the Calibrator produced the output seen in Listing 7. It recognized the Level 1 data cache and Level 2 cache but no TLB. To validate the sizes/line sizes of the detected caches we used *cpuinfo_x86*[5]. Cpuinfo_x86 reads the values for various system parameters from the `CPUID` of the CPU. The output of cpuinfo_x86 is displayed in Listing 8. Cpuinfo_x86 confirms the parameters the Calibrator detected for Level 1 and Level 2 size and line size. Contrary to the Calibrator, however, cpuinfo_x86 reports that there is a TLB (note that only the Data TLB is of interest to us). This

[4]available at `http://homepages.cwi.nl/~manegold/Calibrator/`
[5]available at `http://www.osxbook.com/blog/2009/03/02/retrieving-x86-processor-information/`

```
# Identification
Vendor                : GenuineIntel
Brand String          : Intel(R) Xeon(R) CPU           E5450  @ 3.00GHz
Model Number          : 23 (Penryn)
Family Code           : 6
Extended Model        : 1
Extended Family       : 0
Stepping ID           : 10
Signature             : 67194

...

# Caches
...

## L1 Data Cache
Size                  : 32K
Line Size             : 64B
Sharing               : dedicated per processor thread
Sets                  : 64
Partitions            : 1
Associativity         : 8

## L2 Unified Cache
Size                  : 6M
Line Size             : 64B
Sharing               : shared between 2 processor threads
Sets                  : 4096
Partitions            : 1
Associativity         : 24

# Translation Lookaside Buffers
Instruction TLBs      : 8 large, 128 small
Data TLBs             : 32 large, 256 small

...
```

Listing 8: Output of the cpuinfo_x86

inaccuracy of the Calibrator led us to conduct an experiment of our own to determine the latency of the different caches (for the capacity and line size we used the parameters that were given by cpuinfo_x86).

The Calibrating Experiment: Increasing Stride

The experiment we used for calibration has already been introduced in Section 2.2.1 to motivate the initial investigation in varying memory access time: A constant number of values is summed and the distance of their addresses increased. The access pattern for this experiment is a sequential traversal (*s_trav*) with increasing gaps (constant u and $R.n$, increasing $R.w$). The experiment was conducted with disabled prefetching and the results plotted in Figure 5.1. It shows that the prediction of the costs of the experiment using the parameters of the Calibrator (dashed line) exceeds the measured values for small strides. The predictions deviates from the measured values by a factor up to 5.6. We believe that this is because the calibrator does not consider a memory layer with a block size as large as 32Kb, the block size of the page table (see Section 2.2.3).

To gather more accurate parameters, we used the results of the experiment and *Gnuplot*[6] to fit the cost function of the access pattern to the data points. The cost function for the increasing stride experiment is given in Equation 5.1. The size and line size of the Level 1 and Level 2 caches were taken from the cpuinfo_x86 output. B_0, the size of a data word of the CPU is taken from the documentation of the processor [1]. The size of a TLB page, B_3, can be determined on a UNIX system using the system call `getpagesize()` [67]. The line size of the data page table, B_4, can be calculated from the size of a TLB page B_3 and the number of page table references that can be stored in a cache line. The number of page table references per cache lines is determined by the size of an address (our system uses 64 bit address values) and the size of a Level 2 cache line (64 Byte). The size of a data page block is, thus,

[6] available at `http://www.gnuplot.info/`

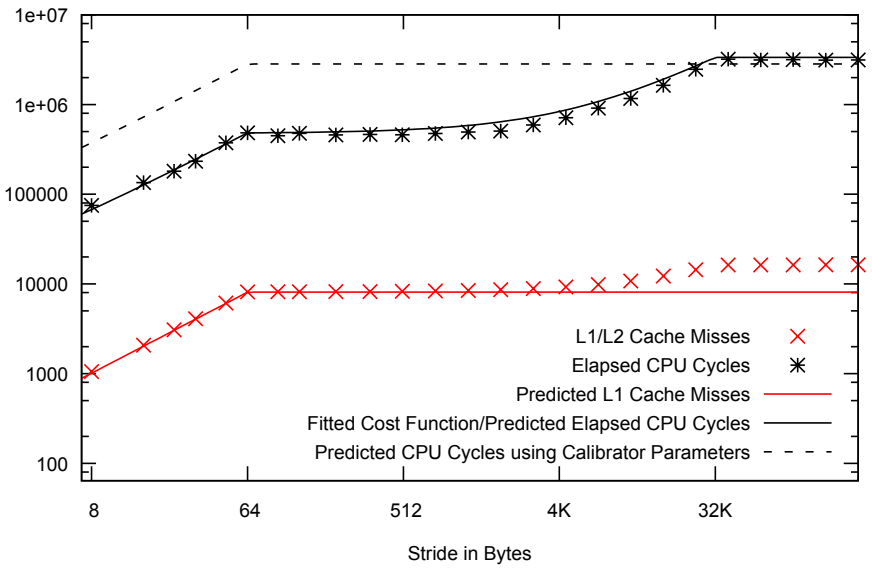


Figure 5.1: Prediction and measured values for the increasing stride experiment

$B_4 = \frac{B_2}{B_0} \cdot B_3 = 32\text{Kbyte}$.

$$T_{Mem} = T_4 \cdot \frac{s}{B_4} + T_3 \cdot \frac{s}{B_3} + T_2 \cdot \frac{s}{B_2} T_1 \cdot \frac{s}{B_1} + T_0 \cdot \frac{s}{B_0} \tag{5.1}$$

Fitting the function of Equation 5.1 to the data points, we determined the parameters displayed in Table 5.1.

The reader may notice that Figure 5.1 shows an unpredicted increase of the Level 2 cache misses as the stride approaches 32Kbyte. This increase reflects the additional Level 2 misses that are induced by the increasing number of entries of the Page Table that have to be read from memory.

Since Level 1 and Level 2 Cache have the same line size, we cannot distinguish Level 2 Cache and memory access latency using the increasing stride experiment. In this experiment Level 1 and Level 2 misses do always occur together. To determine the individual latency of the caches we use the second initial experiment: the increasing unique items experiment (multiple sequential traversals, constant $R.w$ and $u = R.w$, varying $R.n$). The results of this experiment (see Figure 5.2) show that as long as less than 32KByte are accessed (the dataset fits into the Level 1 cache) each value is processed in 1 CPU cycle. When the size of the dataset exceeds 32KByte, the processing time per value increases to 3 CPU cycles per value. The increase in the costs are caused by the induced Level 2 misses which cost 2 CPU cycles each. This experiment allows us to determine the Level 2 access latency independently from the memory access latency. Using the results from this experiment in combination with the data gathered in the previous experiment, we can determine the memory access latency of our system to be 56 cycles.

Having determined values for the necessary parameters, we can evaluate the predictive performance of our model.

5.2.2 Evaluation of the Model

We evaluated the accuracy of the model for the atomic access patterns using hand coded microbenchmarks. The first two, random traversal and repetitive random access are access patterns of the original generic cost model as defined by Manegold et. al [2]. Their accuracy was already evaluated in the original

Variable	Description	Value
B_0:	Size of a General Purpose Register of the CPU	1 word (64 bit)
l_0:	Access Latency of the Level 1 Cache (including processing time)	1 cycle
C_0:	Capacity of a General Purpose Register of the CPU	1 word
B_1:	Size of a cache line of the Level 1 cache	8 words
$l_1 + l_2$:	Access Latency of the Level 2 Cache Access Latency of the main memory	58 cycles
C_1:	Capacity of the Level 1 Cache	4096 words
B_2:	Size of a cache line of the Level 2 cache	8 words
C_2:	Capacity of the Level 2 Cache	786432 words
B_3:	Size of a Memory Page	512 words
l_3:	Lookup time in the Page Table	1 cycle
C_3:	Number of Memory Pages in the TLB multiplied with the Page size	131072 words
B_4:	Size of a Page Table Block	4096 words
l_4:	Loading time of a Page Table Block	340 cycles
C_4:	Number of TLB Page references that can be stored in the Level 2 Cache	3221225472 words

Table 5.1: Memory Access Parameters of the Test System

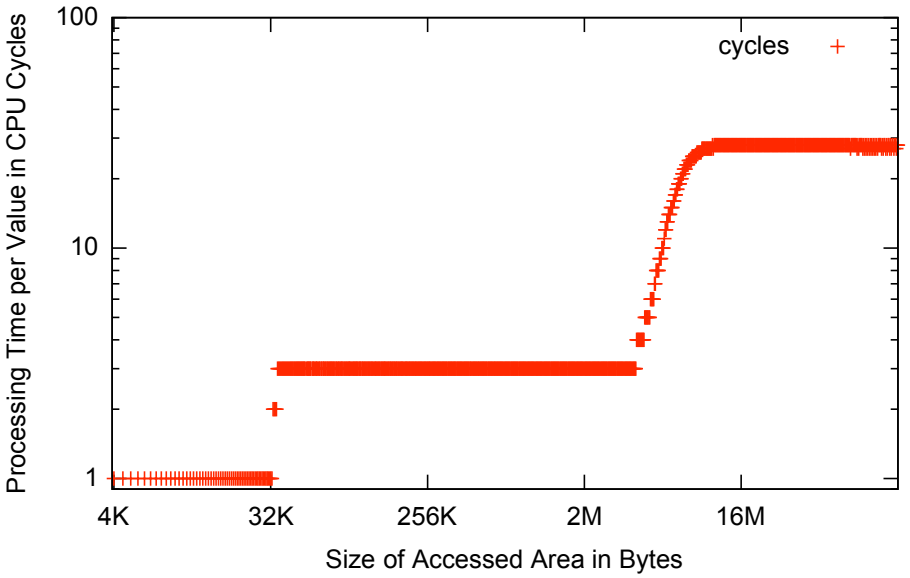


Figure 5.2: Costs of a Data Access to an Area of Varying Size

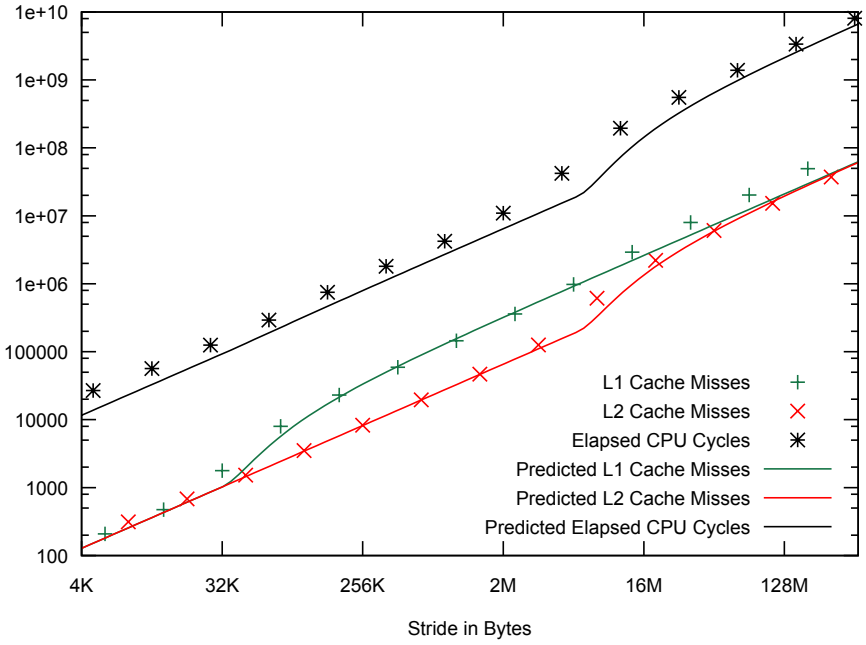


Figure 5.3: Costs of Hash Building (Parallel Sequential and Random Traversal)

work and we only report the results of our experiments to demonstrate the impact of our extensions to the model on the predictive performance. The third access patter, sequential traversal with conditional reads was defined in this thesis and is, consequently, inspected more accurately. All the experiments consist of the reading of an input region and accesses to an output region. They are to reflect operations that are performed by operators of the relational algebra.

Random Traversal

A random traversal is performed by a hash join operator in the hash building phase. An input relation is sequentially (`s_trav`) and a temporary buffer randomly traversed (`r_trav`) to build the hash. Thus, the access pattern is $s\_trav \odot r\_trav$ with equal tuple widths ($u = R.w$) and number of tuples ($R.n$). In our experiment (see Appendix A.3 for the source code), we filled the input relation with randomly distributed unique integer values ($R.w = 1$). The maximal value of these was the size of the input field. When performing the hash build, we used the value of the integer as hash value, and inserted it at a position in the temporary buffer accordingly.

Figure 5.3 shows the predicted and measured values for the L1 and L2 misses as well as the CPU costs for a varying number of values ($R.n$). The figure shows a non-linear increase of the respective costs when the size of the input exceeds the size of a cache. All three depicted measures show that the increase comes earlier than predicted. This can be explained by the higher number of evictions through the prefetching.

Repetitive Random Access

The probing phase of a hash join is characterized by a repetitive random accesses (as is the aggregation in a group by). For our experiment (see Appendix A.4 for the source code), we, again, filled an input relation with random values. They were, however, not required to be unique, but completely independent random

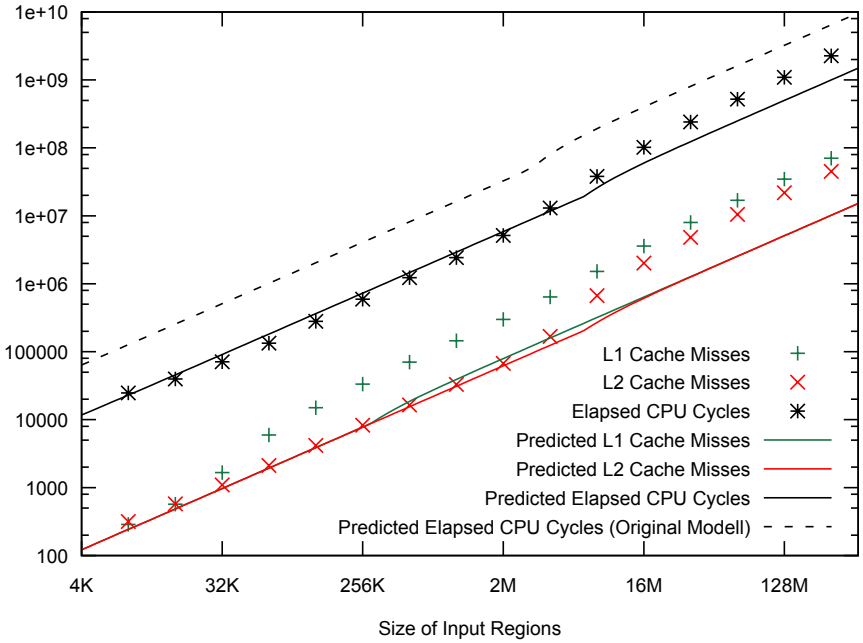


Figure 5.4: Costs of Hash Probing (Parallel Sequential and Random Traversal)

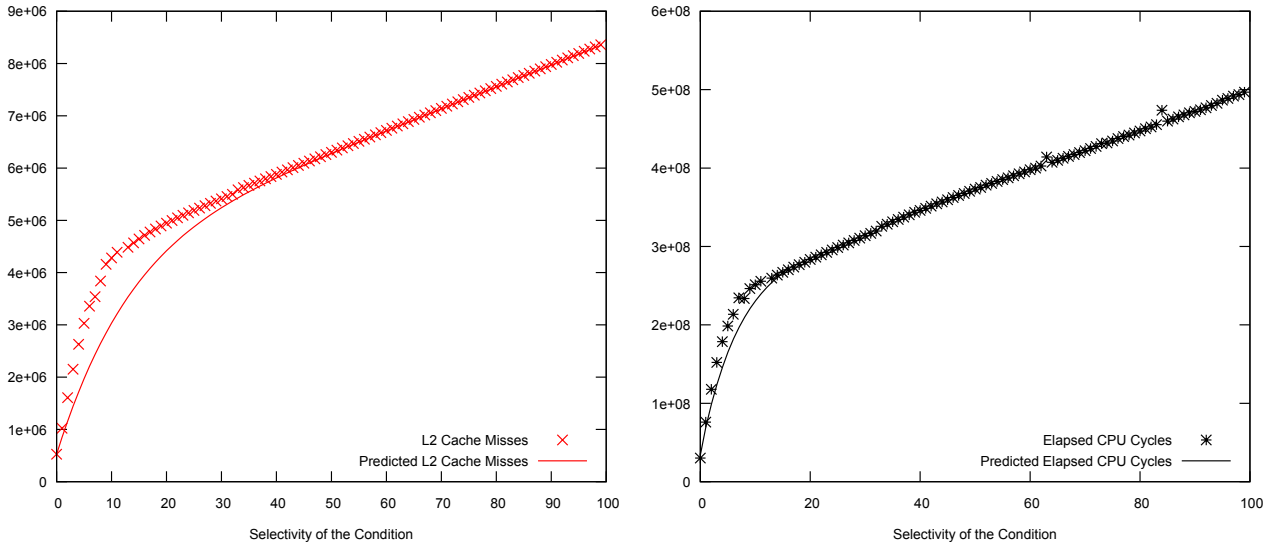
values (the maximal value was, again, the field size). In the experiment the input relation (`s_trav`) was traversed sequentially and the values used to access the temporary buffer that was build in the last experiment. The access pattern is, therefore, $s\_trav \odot rr\_acc$. Figure 5.4 shows the resulting L1/L2 misses as well as the elapsed CPU cycles as well as the prediction for this experiment.

Similar to the last experiment, the measured values show an early and stronger increase than the prediction. This can, again, be explained by the increased number of cache line evictions due to prefetching. The impact of this effect is, however, higher than it is on a random traversal. This is due to the difference in cache line reuse: incorrectly prefetched cache lines in a random traversal have a relatively high chance of being accessed later on because every cache line is accessed. In a repetitive random access, not all cache lines are accessed which decreases the probability that an incorrectly prefetched cache line is of use later on. Due to this the additional evictions, the costs of a repetitive random access to a large input relation are underestimated in our model.

As reported in Section 3.1.3, we used a different weighting for the probability of multiple data items being stored in the same cache line. To report the impact of this modification of Manegold's model, Figure 5.4 also shows the predicted costs using the original weighting (dashed line).

Sequential Traversal Conditional Read

The sequential traversal with conditional reads is our main extension to the generic cost model. It is performed for the reconstruction of tuples in a column-store. For the evaluation (see Appendix A.5 for the source code) we used a column oriented representation of tuples with 8 attributes. The condition, a check of equality to a constant, was applied to the first attribute and the whole tuple reconstructed if the condition held true. Thus, the access pattern is $s\_trav \odot s\_trav\_cr \odot s\_trav\_cr \odot s\_trav\_cr \odot s\_trav\_cr \odot s\_trav\_cr \odot s\_trav\_cr \odot s\_trav\_cr$. The relation was filled with $R.n = 2^{22}$ random integer values ($R.w = 1$). We varied the number of distinct values, thus varying the selectivity of the predicate. Figure 5.5a shows the resulting cache misses for a varying selectivity. For a low selectivity (few tuples



(a) Induced Cache Misses of a `s_trav_cr`

(b) Elapsed CPU Cycles of a `s_trav_cr`

Figure 5.5: Costs of a Sequential Traversal Conditional Read

match), the predicted number of cache misses is below the measured number of misses. This is due to the fact that requested as well as prefetched cache lines are counted in the experiment which is not reflected in the prediction. The prediction of the elapsed CPU cycles (see Figure 5.5b) is, however, more accurate because the different costs of random and sequential misses are taken into account through their different weighting (see Section 3.1.4).

Having evaluated the accuracy of the model, we will report the findings of our efforts in applying our method to a complex scenario in the following section.

5.3 Optimization Performance

We will start this section by defining our benchmark, followed by a discussion of the impact of our method on the costs of the benchmark. We will conclude this section by describing "access pattern compression", an improvement of the implementation that could reduce the optimization costs.

5.3.1 Benchmark Definition

To the best of our knowledge, no benchmark exists that resembles a mixed workload scenario. The TPC-C benchmark targets an OLTP-only scenario while the TPC-H and TPC-DS benchmarks focus on OLAP-only scenarios. In addition, all of them are complex benchmarks that are hard to implement, especially given the early stage of the HYRISE development efforts.

Thus, we evaluated our method based on OLTP queries that were taken from the *SAP Sales and Distribution (SD)* benchmark, which is intended to reflect the typical workload of a sales scenario in an SAP R/3-System. We added two analytical queries to model an Operational Reporting (OR) scenario. The SQL queries, a short description and their relative frequency in our benchmark are displayed in Table 5.3. The queries were executed on the schema that is displayed in Table 5.2. Formalizing this derivate of the SD benchmark as a full benchmarking suite is planned for the future.

While our model can deal with integers (4 Byte) as well as longer fixed length data types, the *HYRISE* prototype can not. Thus, we used an integer-only database schema. Even though this is a

practical limitation, it does not diminish the theoretical value of our approach. Longer fixed length data types can be mapped to multiple columns and queries rewritten accordingly. Data types of variable length can be stored in a pointer-based dictionary. For the benchmark, the tables were filled with randomly distributed integer values. All values, except the foreign keys, are unique within their column.

5.3.2 Experiments

The queries of our benchmark were evaluated by the HYRISE prototype. HYRISE does not have an integrated query processor yet. Thus, the query plans that were generated by the compiler of Spades' SQL Compiler were implemented by hand. We plan to integrate the Spades SQL compiler into the HYRISE system.

Figure 5.6 shows the simulated costs as well as the costs that were measured when executing the workload on a physical system (the data is also displayed in Tables 5.5 and 5.6). The figure shows that in all cases, the real costs are higher than the costs in the simulation. This indicates shortcomings of the HYRISE implementation. Since the calibrations and model evaluation experiments were very carefully implemented, the compiler was abled to optimize the code by, e.g., unrolling loops or using loop vectorization. A real DBMS is, however, more complex and harder to implement in a way that allows sophisticated optimization by the compiler.

Rows versus Columns A good example of the potential for optimization of HYRISE is query 4. In the simulation, the costs decrease by a factor of 3.1[7] when switching from a row to a column oriented layout. In the experiment, the costs merely drop by a factor of 1.2. We believe that the reason for this is that the table `MAKT`, which is the input to query 4, is very narrow. The table has 5 columns, which means that even in a rowstore, a L1/L2 cache line contains on average $\frac{16}{5} = 3.2$ values that have to be processed. These values can not be processed by HYRISE as fast as they come in which makes this query CPU bound even in a rowstore. Our experiments earlier in this chapter indicate that this is not a principle problem but a problem in the implementation of HYRISE.

OLAP queries that are executed on wide tables, however, do benefit from column oriented storage. The costs of query 12, e.g., are reduced by a factor of 8.4 by storing `VBAP` column oriented. This even exceed the predicted improvement of 6.1. Search queries like query 1, that involve attribute scans, also benefit from column storage. In the case of query 1, the costs are reduced by a factor 5.5 (simulated: 5.6). OLTP Queries, like query 5, however, suffer a performance loss when executed on a column oriented layout. Evaluating query 5 takes 4.7 times longer in a columnstore than in a rowstore (simulated: 2.7). Overall, the costs of this workload are reduced by a factor 1.3 by switching from a rowstore to a columnstore.

Unpartitioned Hybrid Calculating the optimal unpartitioned layout (see Table 5.4a) using the simplex based layouter on a MacBook Pro (2.26 Hz Intel Core 2 Duo, 4 GB 1067 Mhz DDR3 RAM) took 63 seconds. As displayed in Figure 5.6, the unpartitioned layout reduced the execution costs of our benchmark by a factor $1,68$ (simulated: $2,75$) in comparison to a rowstore and $1,26$ (simulated: $1,15$) in comparison to a columnstore. The improvement in comparison to a column store is mainly due to the queries 2 and 5. These are the evaluated on the respective tables `KNA1` and `MARA`. The simplex layouter decided to store these row oriented, which is appropriate to the OLTP nature of the queries 2 and 5. These were relatively easy decisions because both tables are not accessed by other queries. For all other tables, the simplex based layouter decided to store them column oriented because the benefit of row oriented storage for OLTP does not outweigh the performance loss for OLAP queries. For these tables, a partitioned layout may by a suitable option.

Partitioned Hybrid Calculating the partitioned layout (see Table 5.4b) took 424 seconds. Figure 5.6 shows a decrease of the costs by a factor 2.3 compared to a row- and 1.8 compared to a columnstore. The improvement compared to an unpartitioned hybrid layout is by a factor 1.4. The largest improvement is in the OLTP queries (6 to 10) because they benefit from the mostly row layout. This layout allows the

[7] All factors have been rounded to the first decimal place

Table Name	Primary Key	Description	Number of Entries	Number of Attributes	Foreign Keys
ADRC	ADDRNUMBER	Business Partner Address	15000	85	ADRC.KUNNR →KNA1.KUNNR
KNA1	KUNNR	Business Partner	12000	165	
VBAK	VBELN	Sales Document Header	300000	123	VBAK.KUNNR →KNA1.KUNNR
VBAP	VBELN, MATNR	Sales Document Items	1200000	214	VBAP.VBELN →VBAK.VBELN, VBAP.MATNR →MARA.MATNR
MARA	MATNR	Material	50000	204	
MAKT	MATNR	Material Text	50000	5	MAKT.MATNR →MARA.MATNR

Table 5.2: The Tables used in the benchmark

	Query Description and SQL	Relative Frequency
Q1	Search for a customer by it's name `select addrnumber, name_co, name1, name2, kunnr from adrc where name1 like 'x' or name2 like 'y'`	500
Q2	Show the details for this customer `select * from kna1 where id = $1`	500
Q3	Show all addresses of this customer `select * from adrc where kunnr = $1`	500
Q4	Search for a material by it's description `select matnr, maktx from makt where maktx like 'x'`	2500
Q5	Show the details of this material `select * from mara where id = $1`	2500
Q6	Create a new order `insert into vbak values ($1,$2,...)`	500
Q7	Create five line items for this order `insert into vbap values ($1)`	2500
Q8	Display the created order `select * from vbak where id = $1`	500
Q9	Display the line items of the created order `select * from vbap where id = $1`	2500
Q10	Show the last 30 created orders `select top 30 * from vbak where order by vbeln`	10
Q11	Show the turnover for customer KUNNR in the last month `select sum(vbap.netwr), kunnr from vbap, vbak where vbap.vbeln = vbak.vbeln and month(vbak.audat) = $1 and vbak.kunnr = $2`	10
Q12	Show the number of sold units of material MATNR for last two month `select edatu, sum(kwmeng) from vbap where matnr = $1 and (month(aedat) = $2 or month(aedat) = $3)`	10

Table 5.3: Queries of the modified SAP SD Benchmark

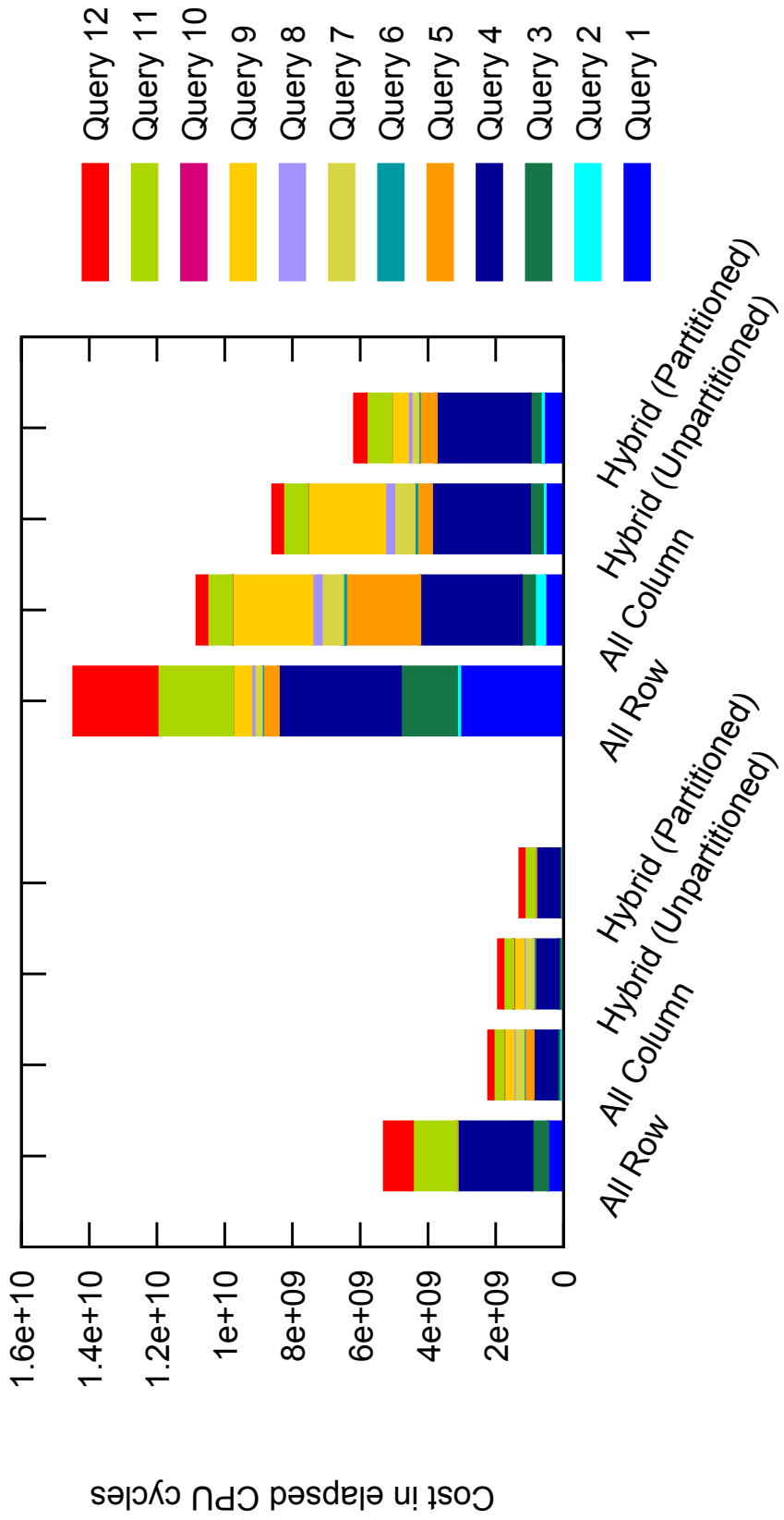


Figure 5.6: Simulated and Measured Costs of different Layouts

Table	Orientation
ADRC	columns
KNA1	rows
MAKT	columns
MARA	rows
VBAK	columns
VBAP	columns

(a) The (analytically) optimal unpartitioned layout

Table	Oriented Partitions
ADRC	(kunnr): C, (name1, name2): R, (...): R
KNA1	(...): R
MAKR	(MAKTX): C, (...): R
MARA	(...): R
VBAK	(VBELN): C, (AEDAT, VKORG): R, (...): R
VBAP	(AEDAT, MATNR): R, (...): R

(b) The (analytically) Optimal Partitioned Layout

Table 5.4: The layouts generated by Spades

OLAP queries to benefit from the column oriented partitions withhout hurting the OLTP performance. Query 3, a search query with tuple reconstruction, also benefits from this layout (factor 1.3 compared to unpartitioned). The partitioning, while expensive to calculate, brings an additional performance benefit over the unpartitioned layout. The relatively high calculation costs do, however, may deem the partitioned layouter unsuited for an optimization at runtime. In the last section we want to present an option for reducing the optimization effort: *Access Pattern Compression*.

Options for improving the layouting performance

Evaluating *Spades*, we noticed that the layouting of schemas with wide tables takes much longer than the optimization of schemas with narrower tables. This is due to the increasing complexity of the access patterns. Even for a fixed workload, the complexity of the access pattern may increase with the widths of the tables. To illustrate this, consider the query

```
select * from adrc where name like $1.
```

When evaluated on a column store, the access pattern for this query is a sequential traversal of `name` and a sequential traversal with conditional reads for all other attributes. Assuming that all attributes have the same datatype, the access patterns on all of these attributes, and therefore also their costs are identical. *Spades*, however evaluates the costs for each of the patterns individually. For wide tables, the computation effort increases accordingly. We believe that this problem can be circumvented by "compressing" the access patterns: multiple identical access patterns without dependencies on each other can be evaluated once and the costs multiplied by their number. This was, however, not implemented in *Spades*.

Even though there is room for improvement in the derivate of *Spades*, we could reduce the costs of our implementation of the *SAP Sales and Distribution* benchmark by 134% compared to a row- and 75% compared to a columnstore in an optimization time of minutes.

Layout	Row	Column	Unpartitioned	Partitioned
Query 1	474749000	84690000	84690000	84184000
Query 2	1619500	41978500	1619500	1619500
Query 3	449022500	68885500	68885500	42899000
Query 4	2217492500	697330000	697330000	696080000
Query 5	9527500	259287500	9527500	9527500
Query 6	1324000	31597500	31597500	1546500
Query 7	9892500	271950000	271950000	11005000
Query 8	1324000	31597500	31597500	1546500
Query 9	9892500	271950000	271950000	11005000
Query 10	794400	18958500	18958500	927900
Query 11	1286024500	299997780	299997780	299381450
Query 12	825301470	134127100	134127100	133056320
Sum	5286964370	2212349880	1922230880	1292778670

Table 5.5: Simulated Costs

Layout	Row	Column	Unpartitioned	Partitioned
Query 1	3057717500	547021300	542125735	588737465
Query 2	97386535	307390700	80636800	96067065
Query 3	1661614835	386983900	376918765	301295100
Query 4	3598084325	2999605175	2896466675	2768905175
Query 5	462536000	2188097825	436721500	499058000
Query 6	40419300	88004965	79768665	44914635
Query 7	215806675	632137000	595904175	202250825
Query 8	93123865	274834865	264345600	103605000
Query 9	529782175	2357982500	2273861500	476284325
Query 10	1665284	5232181	5074078	1808724
Query 11	2239759510	740689724	738673890	739500652
Query 12	2456126884	291709684	291479098	345380690
Sum	14454022888	10819689819	8581976481	6167807656

Table 5.6: Real Costs

Chapter 6

Conclusion and Future Work

In this section we want to revise our findings and present ideas for future work.

6.1 Conclusion

Neither row nor column oriented storage is the optimal storage layout to maximize the database performance for all applications. Hybrid storage is an alternative that allows storing each piece of data in the most appropriate layout.

In this thesis, we developed a methodology to automatically select an appropriate hybrid storage layout for data in an in-memory database. Which layout is appropriate is, naturally, depending on the queries that are executed on the database, the workload.

We found that the most important factor for the data access performance to be the number of the number cache misses, i.e., blocks that have to be transferred from one memory layer to another. Our method, therefore, aims at minimizing the number of cache misses.

We investigated into different models to capture the data access costs in dependence of the workload and data layout and found the generic cost model developed by Manegold et al. to fit our needs best. In the generic cost model the workload is represented as the data access pattern that is exposed when executing each of the queries on a given layout. The costs of the workload on the layout can be derived from it's access pattern. To support our use case, we extended the model with a new access pattern that is exposed in column stores: the sequential traversal with conditional reads. We also extended the model to take the effects of Level 2 Cache prefetching into account. We evaluated the accuracy of the extended generic cost model using a set of microbenchmarks.

Based on this cost model, we developed two algorithms that aim at selecting the most appropriate layout from different search spaces. The first is based on the *Simplex Method* for solving linear problems. It is, therefore, called the *Simplex Based Layouter*. It selects the analytically optimal unpartitioned layout, i.e., it assigns an orientation, row or column, to every relation. The second algorithm, the *Oriented Optimal Binary Partitioning*, is based on the *Optimal Binary Partitioning* algorithm. It divides each relation into partitions and selects the most appropriate orientation for each partition. To ensure optimality, we had to limit the workload to queries with one or no join at all.

Both algorithms were implemented in a tool called *Spades*. *Spades* takes parameters of the target system, an SQL workload, a database schema and the distribution of the values in the database as input. Based on these data, *Spades* generates an analytically optimal partitioned or unpartitioned layout.

We evaluated the performance improvements of both algorithms using a benchmark that is based on the *SAP Sales and Distribution Benchmark*. Besides the queries of the original SD benchmark, our benchmark also contained analytical queries. The benchmark was executed using HYRISE, a prototype of a hybrid in-memory database. We found that using our method, we could reduce the time to run the benchmark by 134% in comparison to a rowstore and 75% compared to a columnstore. Despite these improvements for our benchmark, we see several open fields for future work.

6.2 Future Work

An untackled limitation of our approach is the assumption of single- or no-join queries. In our case this limitation did not spoil analytical optimality but for more complex workloads the usage of methods of nonlinear optimization would have to be investigated.

The extended generic cost model allows a prediction of the execution costs depending on a number of parameters. This thesis is focused on the influence of the storage layout on the execution costs. Alternatively, the model could be used to optimize any of the other parameters with respect to a workload. Amongst the parameters that could be investigated in the future are

- the compression of values within a column,
- the compression of values within a row,
- the prefetching strategy,
- the writing strategy of the caches.

Especially the last two are also interesting in a multi CPU context because they could be varied on a per CPU basis. This would allow tuning each CPU to a different part of the target workload and distributing the queries to the most appropriate CPU.

We also focused strictly on non-redundant layouting. Indices or materialized views have not been considered. Especially indices, that are sometimes seen as a competitor to column oriented storage, are of interest. On the one hand, indices come with a performance benefit that is expected to be superior to that of hybrid partitioning for read-only workloads. On the other hand, indices introduce additional costs for their maintenance in workloads that contain modifying queries. Automatically selecting hybrid storage or indices whenever each is appropriate is another interesting challenge for the future.

Appendix A

Sourcecode for Experiments

A.1 increasingstride.cpp

```
#include <iostream>
#include <string>
#include <papi.h>
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: %s \n", retval,__FILE__,__LINE__
    , errstring); }
#define CACHESIZE_IN_MB 6

void inline clear_cache(){
  int sum;
  int * dummy_array = new int[1024*1024*CACHESIZE_IN_MB];
  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array[address] = address+1;
  }
  int * dummy_array2 = new int[1024*1024*CACHESIZE_IN_MB];
  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array2[address] = address+1;
  }
  for(int repetition = 0; repetition < 3; repetition++){
    for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
      sum += dummy_array[address];
    }
  }
  delete dummy_array;
  delete dummy_array2;
}

int sum = 0;
main(){
  unsigned int size = 4*1024*1024*1024;
  int* field = new int[size];

  int randomvalue = 0;
  for(int i =0; i< size;i++){
    field[i] = randomvalue = (randomvalue+104729)%48611;
  }

  int Events[1] = {PAPI_TOT_CYC};
  int num_hwcntrs = 0;
  int retval;
  char errstring[PAPI_MAX_STR_LEN];
  long_long values[1] = {0};

  unsigned int stride = 0;
   for(float fstride=1;fstride<524288;fstride*=1.0108892860517f){
    stride=((unsigned int)fstride);
      sum = 0;
      clear_cache();
      if ( (retval = PAPI_start_counters(Events, 1)) != PAPI_OK)
        ERROR_RETURN(retval);

      unsigned int position=0;
      for(int i = 0; i < 8192; i++)
        sum += field[i*stride];

      if ((retval=PAPI_stop_counters(values, 1)) != PAPI_OK)
        ERROR_RETURN(retval);

      std::cout << stride*4 << " " << values[0]/8096 << std::endl;
  }

  return 0;
}
```

A.2 increasinguniqueitems.cpp

```
#include <iostream>
#include <string>
#include <sstream>
#include <papi.h>
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: %s \n", retval,__FILE__,__LINE__
    , errstring); }
#define CACHESIZE_IN_MB 6

void inline clear_cache(){
  int sum;
  int * dummy_array = new int[1024*1024*CACHESIZE_IN_MB];

  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array[address] = address+1;
  }


  int * dummy_array2 = new int[1024*1024*CACHESIZE_IN_MB];
  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array2[address] = address+1;
  }

  for(int repetition = 0; repetition < 3; repetition++){
    for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
      sum += dummy_array[address];
    }
  }
  delete dummy_array;
  delete dummy_array2;
}

void fill_field_with_random_values(int field[], int size){
  int randomvalue = 0;
  for(int i =0; i< size;i++){
    field[i] = randomvalue = (randomvalue+104729)%48611;
  }
}

int sum = 0;
int* field;
main(){
  unsigned int size = 4*32*1024*1024/4;
  field = new int[size];

  fill_field_with_random_values(field,size);

  int Events[1] = {PAPI_TOT_CYC};
  int num_hwcntrs = 0;
  int retval;
  char errstring[PAPI_MAX_STR_LEN];
  long_long values[1] = {0};
  std::ostream& out = std::cout;

  unsigned int numberofuniqueitems = 0;
  for(float numberofuniqueitemsfloat = 1; numberofuniqueitemsfloat < size; numberofuniqueitemsfloat *=
      1.0108892860517f){
    if(numberofuniqueitems!=((unsigned int)numberofuniqueitemsfloat))
      {
        numberofuniqueitems=((unsigned int)numberofuniqueitemsfloat);
        clear_cache();
        sum = 0;
        if ( (retval = PAPI_start_counters(Events, 1)) != PAPI_OK)
          ERROR_RETURN(retval);
        int maxwith = numberofuniqueitems * 16;
        for(int i = 0; i < size; i+=16)
          sum += field[i%maxwith];

        if ((retval=PAPI_stop_counters(values, 1)) != PAPI_OK)
          ERROR_RETURN(retval);

        out << numberofuniqueitems << " " << values[0]/numberofuniqueitems << std::endl;
      }
  }


  return 0;
}
```

A.3 hash_build.cpp

```
#include <iostream>
#include <string>
#include <sstream>

#ifdef USE_PAPI_TRACE
```

```
#include <papi.h>
#endif

#include <vector>
#include <map>
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: %s \n", retval,__FILE__,__LINE__
    , errstring); }
#define CACHESIZE_IN_MB 6

struct papi_triple{
  long cycles;
  long l2_total;
  long l1_data;
};



namespace cache2{
  int * dummy_array;
  int * dummy_array2;
  int sum;

void inline clear(){
 dummy_array = new int[1024*1024*CACHESIZE_IN_MB];
 dummy_array2 = new int[1024*1024*CACHESIZE_IN_MB];

  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array[address] = address+1;
  }


  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array2[address] = address+1;
  }

  for(int repetition = 0; repetition < 3; repetition++){
    for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
      sum += dummy_array[address];
    }
  }
  delete dummy_array;
  delete dummy_array2;
}
}

namespace hash_build_ns{
  int field_size;
  long * field;
  long * source_field;
  bool initialized = false;
  bool target_field_initialized = false;
  void clear_target_field(){
    if(!target_field_initialized){
      target_field_initialized = true;
      field = new long[50000000];
    }
    for(int i = 0; i<50000000;i++){
      field[i] = 0;
    }
  };
}

void holger_malloc(int size_in_ints){
  if(!hash_build_ns::initialized){
    hash_build_ns::source_field = new long[50000000];
    hash_build_ns::initialized = true;
  }
  hash_build_ns::field_size = size_in_ints;
  std::cout << "created_field_of_size_" << hash_build_ns::field_size  << std::endl;
}
void holger_free(){
  std::cout << "deleting_field_of_size_" << hash_build_ns::field_size  << std::endl;
  if(hash_build_ns::initialized){
    delete(hash_build_ns::field);
    hash_build_ns::initialized = false;
  }
}
void holger_set_in_allocated_field(int position, long value){
  hash_build_ns::source_field[position] = value;
}



papi_triple hash_build(){

#ifdef USE_PAPI_TRACE
  hash_build_ns::clear_target_field();


  int Events[3] = {PAPI_TOT_CYC, PAPI_L2_TCM, PAPI_L1_DCM};
  int num_hwcntrs = 0;
```

```
  int retval;
  char errstring[PAPI_MAX_STR_LEN];
  long_long papi_values[3] = {0,0,0};

  cache2::clear();
  if ( (retval = PAPI_start_counters(Events, 3)) != PAPI_OK)
    ERROR_RETURN(retval);

  for(int i =0; i< hash_build_ns::field_size;i++){
    hash_build_ns::field[hash_build_ns::source_field[i]] = hash_build_ns::source_field[i];
  }

  if ((retval=PAPI_stop_counters(papi_values, 3)) != PAPI_OK)
    ERROR_RETURN(retval);



  papi_triple thing;
  thing.cycles = papi_values[0];
  thing.l2_total = papi_values[1];
  thing.l1_data = papi_values[2];

#endif
  return thing;
}
```

A.4 hash_probe.cpp

```
#include <iostream>
#include <string>
#include <sstream>

#ifdef USE_PAPI_TRACE
#include <papi.h>
#endif

#include <vector>
#include <map>
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: %s \n", retval,__FILE__,__LINE__
    , errstring); }
#define CACHESIZE_IN_MB 6

struct papi_triple{
  long cycles;
  long l2_total;
  long l1_data;
};



namespace cache3{
  int * dummy_array;
  int * dummy_array2;
  int sum;

void inline clear(){
 dummy_array = new int[1024*1024*CACHESIZE_IN_MB];
 dummy_array2 = new int[1024*1024*CACHESIZE_IN_MB];

  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array[address] = address+1;
  }


  for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
    dummy_array2[address] = address+1;
  }

  for(int repetition = 0; repetition < 3; repetition++){
    for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
      sum += dummy_array[address];
    }
  }
  delete dummy_array;
  delete dummy_array2;
}
}

namespace hash_probe_ns{
  long sum = 0;
  int field_size;
  long * field;
  long * source_field;
  long * target_field;
  bool initialized = false;

}
```

```
#define SOURCE_FIELD_SIZE 50000000

void holger_malloc_hp(int size_in_ints){
  if(!hash_probe_ns::initialized){
    hash_probe_ns::field = new long[50000000];
    hash_probe_ns::source_field = new long[SOURCE_FIELD_SIZE];
    hash_probe_ns::target_field = new long[SOURCE_FIELD_SIZE];
    hash_probe_ns::initialized = true;
    for(int i = 0; i<50000000; i++){
      hash_probe_ns::field[i] = 50000000-i;
    }
  }
  hash_probe_ns::field_size = size_in_ints;
  std::cout << "created_field_of_size_" << hash_probe_ns::field_size  << std::endl;
}
void holger_free_hp(){
  std::cout << "deleting_field_of_size_" << hash_probe_ns::field_size  << std::endl;
  if(hash_probe_ns::initialized){
    delete(hash_probe_ns::field);
    hash_probe_ns::initialized = false;
  }
}
void holger_set_in_allocated_field_hp(int position, long value){
  hash_probe_ns::source_field[position] = value;
}



papi_triple hash_probe(){

#ifdef USE_PAPI_TRACE


  int Events[3] = {PAPI_TOT_CYC, PAPI_L2_TCM, PAPI_L1_DCM};
  int num_hwcntrs = 0;
  int retval;
  char errstring[PAPI_MAX_STR_LEN];
  long_long papi_values[3] = {0,0,0};

  cache3::clear();
  if ( (retval = PAPI_start_counters(Events, 3)) != PAPI_OK)
    ERROR_RETURN(retval);
  long sum = 0;

  for(int i =0; i< hash_probe_ns::field_size;i++){
    hash_probe_ns::target_field[i] = hash_probe_ns::field[hash_probe_ns::source_field[i]];
  }

  if ((retval=PAPI_stop_counters(papi_values, 3)) != PAPI_OK)
    ERROR_RETURN(retval);



  papi_triple thing;
  thing.cycles = papi_values[0];
  thing.l2_total = papi_values[1];
  thing.l1_data = papi_values[2];

#endif
  return thing;
}
```

A.5 selection_with_varying_selectivity.cpp

```
#include <iostream>
#ifdef USE_PAPI_TRACE
#include <papi.h>
#endif
#include <testing/papi_triple.h>
#include <stdlib.h>
#include <map>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define DATATYPE long

#define ROWS 4194304
#define COLUMNS 8
#define TOTAL_SIZE 33554432

#define CACHESIZE_IN_MB 6

#define ERROR_RETURN(retval) { fprintf(stderr, "Error_%d_%s:line_%d:,_%s_\n", retval,__FILE__,__LINE__
    , errstring);  }

namespace selection_with_varying_selectivity_ns{
  typedef long_long PAPI_LONG_LONG;
```

```
  const int CARDINALITY = 5000;

  int * dummy_array;
  int * dummy_array2;
  int sum;
  DATATYPE * target;


  void inline clear_cache(){
    dummy_array = new int[1024*1024*CACHESIZE_IN_MB];
    dummy_array2 = new int[1024*1024*CACHESIZE_IN_MB];

    for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
      dummy_array[address] = address+1;
    }


    for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
      dummy_array2[address] = address+1;
    }

    for(int repetition = 0; repetition < 3; repetition++){
      for(int address = 0; address < 1024*1024*CACHESIZE_IN_MB; address++){
        sum += dummy_array[address];
      }
    }
    delete dummy_array;
    delete dummy_array2;
  }


  inline papi_triple rowstore(const DATATYPE * table, float selectivity){

#ifdef USE_PAPI_TRACE
    int Events[3] = {PAPI_TOT_CYC,PAPI_L2_TCM,PAPI_TLB_DM};
    int num_hwcntrs = 0;
    int retval;
    char errstring[PAPI_MAX_STR_LEN];
#endif

    PAPI_LONG_LONG values[3];

    int threshold = selectivity*CARDINALITY;

    clear_cache();
#ifdef USE_PAPI_TRACE
    if ( (retval = PAPI_start_counters(Events, 3)) != PAPI_OK)
      ERROR_RETURN(retval);
#endif

#define row_process \
    target[target_position++] = table[i];                \
    target[target_position++] = table[i+1];      \
    target[target_position++] = table[i+2];      \
    target[target_position++] = table[i+3];      \
    target[target_position++] = table[i+4];      \
    target[target_position++] = table[i+5];      \
    target[target_position++] = table[i+6];      \
    target[target_position++] = table[i+7];      \


    int target_position = 0;
    for(int i=0; i<TOTAL_SIZE; i+=COLUMNS)
      if(table[i] <= threshold){
        row_process
      }

#ifdef USE_PAPI_TRACE
    if ((retval=PAPI_stop_counters(values, 3)) != PAPI_OK)
      ERROR_RETURN(retval);
#endif

    papi_triple result;
    result.cycles = values[0];
    result.l2_total = values[1];
    result.l1_data = values[2];
    return result;
  }


  inline papi_triple columnstore(DATATYPE * table, float selectivity){

#ifdef USE_PAPI_TRACE
    int Events[3] = {PAPI_TOT_CYC,PAPI_L1_DCM,PAPI_L2_TCM};
    int retval;
    char errstring[PAPI_MAX_STR_LEN];
#endif
    PAPI_LONG_LONG values[3];

    const DATATYPE threshold = selectivity*CARDINALITY;
```

```
    clear_cache();
#ifdef USE_PAPI_TRACE
    if ( (retval = PAPI_start_counters(Events, 3)) != PAPI_OK)
      ERROR_RETURN(retval);
#endif

    int target_position = 0;

#define process                                                 \
    target[target_position++] = table[i];                       \
    target[target_position++] = table[1*ROWS+i];                \
    target[target_position++] = table[2*ROWS+i];                \
    target[target_position++] = table[3*ROWS+i];                \
    target[target_position++] = table[4*ROWS+i];                \
    target[target_position++] = table[5*ROWS+i];                \
    target[target_position++] = table[6*ROWS+i];                \
    target[target_position++] = table[7*ROWS+i];                \

    for(int i=0; i<ROWS;){
      if(table[i++] <= threshold){
        process
          }
      if(table[i++] <= threshold){
        process
          }
      if(table[i++] <= threshold){
        process
          }
      if(table[i++] <= threshold){
        process
          }
    }

#ifdef USE_PAPI_TRACE
    if ((retval=PAPI_stop_counters(values, 3)) != PAPI_OK)
      ERROR_RETURN(retval);
#endif
    papi_triple result;
    result.cycles = values[0];
    result.l2_total = values[1];
    result.ll_data = values[2];
    return result;
  }


  inline void fill_table_column_layout(DATATYPE * table){
    for(int column=0; column < COLUMNS; column++){
      for(int row=0; row<ROWS; row++){
        table[column*ROWS + row] = ((int)(((float)rand())*((float)CARDINALITY)/((float)RAND_MAX)))+1;
      }
    }
  }

  inline void fill_table_row_layout(DATATYPE * table){
    fill_table_column_layout(table);
  }

}



std::map<int,papi_triple> selection_with_varying_selectivity_row(){
  //select addrnumber, name_co, name1, name2, kunnr from adrc where name1 = 'x' or name2 = 'y';

  std::map<int,papi_triple> result;
  int retval;
#ifdef USE_PAPI_TRACE
  char errstring[PAPI_MAX_STR_LEN];

  if((retval = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_CURRENT )
    {
      fprintf(stderr, "Error: %d %s\n",retval, errstring);
    }
#endif


  DATATYPE * adrc = new DATATYPE[ROWS*COLUMNS];
  selection_with_varying_selectivity_ns::fill_table_row_layout(adrc);
  selection_with_varying_selectivity_ns::target = new DATATYPE[ROWS*COLUMNS];
  selection_with_varying_selectivity_ns::fill_table_row_layout(selection_with_varying_selectivity_ns::
      target);

  for(float i = 0;i<=1;i+=.01){
    selection_with_varying_selectivity_ns::clear_cache();
    result[(int)(i*100)]=  selection_with_varying_selectivity_ns::rowstore(adrc,i);
  }
  free(adrc);
  free(selection_with_varying_selectivity_ns::target);

  return result;
}
```

```
std::map<int,papi_triple> selection_with_varying_selectivity_column(){
  //select addrnumber, name_co, name1, name2, kunnr from adrc where name1 = 'x' or name2 = 'y';

  std::map<int,papi_triple> result;

  int retval;
#ifdef USE_PAPI_TRACE
  char errstring[PAPI_MAX_STR_LEN];

  if((retval = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_CURRENT ) {
    fprintf(stderr, "Error: %d %s\n",retval, errstring);
  }
#endif


  DATATYPE * adrc = new DATATYPE[ROWS*COLUMNS];
  selection_with_varying_selectivity_ns::target = new DATATYPE[ROWS*COLUMNS];
  selection_with_varying_selectivity_ns::fill_table_column_layout(adrc);

  for(float i = 0.0;i<=1.0;i+=.01){
    selection_with_varying_selectivity_ns::clear_cache();
    result[(int)(i*100)] = selection_with_varying_selectivity_ns::columnstore(adrc,i);
  }
  free(adrc);
  free(selection_with_varying_selectivity_ns::target);
  return result;
}

main(){
  selection_with_varying_selectivity_row();
}
```

Appendix B

Sourcecode of the Spades Implementation

B.1 parser.ypp

```
%token <a_char> A_COMPARATOR
%token FROM SELECT WHERE COMMA AND GROUPBY AS TOP OR INSERTINTO VALUES ORDERBY ORDER_DIRECTION
%token <an_object> AGGREGATION VALUE_FUNCTION
%token <number> NUMBER
%left <an_object> TABLENAME CONSTANT
%right <an_object> ATTRIBUTENAME
%%
input: statement
| statement input;

statement: selectstatement ';'
| insertstatement ';'
| selectstatement ':' NUMBER ';'
| insertstatement ':' NUMBER ';';

insertstatement: INSERTINTO TABLENAME '(' atomicattributes ')' VALUES '(' constants ')'

selectstatement: SELECT attributes FROM tables optionalwhere optionalgroupby optionalorderby
| SELECT TOP NUMBER attributes FROM tables optionalwhere optionalgroupby optionalorderby;;

constants: CONSTANT ',' constants
| CONSTANT;

optionalgroupby: | GROUPBY atomicattributes;

optionalorderby: | ORDERBY orderedatomicattributes;

tables: tables ',' TABLENAME
| TABLENAME
| tables ',' TABLENAME AS TABLENAME
| TABLENAME AS TABLENAME;

attributes: valueattribute ',' attributes
| valueattribute
| AGGREGATION '(' ATTRIBUTENAME ')' ',' attributes
| AGGREGATION '(' ATTRIBUTENAME ')'
| AGGREGATION '(' ATTRIBUTENAME ')' AS ATTRIBUTENAME ',' attributes
| AGGREGATION '(' ATTRIBUTENAME ')' AS ATTRIBUTENAME ;

atomicattributes: valueattribute ',' atomicattributes
| valueattribute ;

orderedatomicattributes: valueattribute ORDER_DIRECTION ',' orderedatomicattributes
| valueattribute ORDER_DIRECTION
| valueattribute ',' orderedatomicattributes
| valueattribute ;

optionalwhere: | WHERE conditions;

conditions: conditions AND onecondition
| conditions OR onecondition
| onecondition ;

valueattribute: VALUE_FUNCTION '(' ATTRIBUTENAME ')'
| ATTRIBUTENAME ;

onecondition: valueattribute A_COMPARATOR valueattribute
| valueattribute A_COMPARATOR CONSTANT
```

```
| '(' onecondition OR onecondition ')' ;
%%
```

B.2 lexer.lpp

```
%{
#include "y.tab.h"
#include <algorithm>
#include <iostream>
#include <string>
        using std::cout;
        using std::endl;
        using std::string;


        extern int yyerror();

#undef YY_INPUT
#define YY_INPUT(b,r,ms) (r = my_yyinput(b, ms))

        extern const char input[] ;
        extern const char *inputptr;
        extern const char *inputlength;

        int my_yyinput(char *buff, int max_size){
                int n = std::min((int) max_size,(int)(inputlength - inputptr));
                if(n > 0){
                        memcpy(buff,inputptr,n);
                        inputptr += n;
                }
                return n;
        }

        bool moreInput(){
                return inputptr < inputlength;
        }

        enum identifiertypes {
                TABLE,ATTRIBUTE,NONE,OPERATOR
        };

        enum numbertypes {
        A_NUMBER,A_CONSTANT
        };

        enum query_types {
                SELECT_QUERY,INSERT_QUERY
        };

        enum query_types query_type = SELECT_QUERY;

        enum identifiertypes next_identifier_type = NONE;
        enum numbertypes next_number_type = A_NUMBER;

%}

%option noyywrap
%s select_query
%%
select  {
        next_identifier_type = ATTRIBUTE;
        query_type = SELECT_QUERY;
        BEGIN(select_query);
        return SELECT;
}

top                     {
        next_number_type = A_NUMBER;
        return TOP;
}

"//"[^\n]* {
}

"--"[^\n]* {
}

[0-9]+  {
        if(next_number_type == A_CONSTANT) {
                yylval.an_object = new string(yytext);
                next_number_type = A_NUMBER;
                return CONSTANT;
        }
        else{
                yylval.number = atoi(yytext);
                return NUMBER;
        }
}
```

```
from            {next_identifier_type = TABLE; return FROM;}

and                     {return AND;}

or                      {return OR;}

where           {next_identifier_type = ATTRIBUTE; return WHERE;}

sum|avg {
        yylval.an_object = new string(yytext);
        return AGGREGATION;
}

year|month {
        yylval.an_object = new string(yytext);
        return VALUE_FUNCTION;
}

as {
        return AS;
}


group\ by {
        next_identifier_type = ATTRIBUTE;
        return GROUPBY;
}

order\ by {
        next_identifier_type = ATTRIBUTE;
        return ORDERBY;
}


asc|desc {
        yylval.an_object = new string(yytext);
        return ORDER_DIRECTION;
}

insert\ into {
        next_identifier_type = TABLE;
        query_type = INSERT_QUERY;
        return INSERTINTO;
}

values {
        return VALUES;
}

like    {
        yylval.a_char = '=';
        next_number_type = A_CONSTANT;
        return A_COMPARATOR;
}

<select_query>[a-zA-z0-9]+(\.[a-zA-z0-9]+)?\ *\*\ *[a-zA-z0-9]+(\.[a-zA-z0-9]+)? {
        switch (next_identifier_type) {
                case ATTRIBUTE:
                        yylval.an_object = new string(yytext);
                        return ATTRIBUTENAME;
                default:
                        break;

        }
}


[a-zA-z0-9]+(\.[a-zA-z0-9]+)?\ *-\ *[a-zA-z0-9]+(\.[a-zA-z0-9]+)? {
        switch (next_identifier_type) {
                case ATTRIBUTE:
                        yylval.an_object = new string(yytext);
                        return ATTRIBUTENAME;
                default:
                        break;

        }
}


\* {
        if(next_identifier_type == ATTRIBUTE) {
                yylval.an_object = new string(yytext);
                return ATTRIBUTENAME;
        }

}

[a-zA-z0-9]+(\.[a-zA-z0-9]+)? {
        switch (next_identifier_type) {
                case TABLE:
                        yylval.an_object = new string(yytext);
```

```
                        if(query_type == INSERT_QUERY){
                                next_identifier_type = ATTRIBUTE;
                        }
                        return TABLENAME;
                case ATTRIBUTE:
                        yylval.an_object = new string(yytext);
                        return ATTRIBUTENAME;
                default:
                        break;

        }
}


'[^']*'|\$[0-9]+        {
yylval.an_object = new string(yytext);
return CONSTANT;
}

=       {
yylval.a_char = yytext[0];
next_number_type = A_CONSTANT;
return A_COMPARATOR;
}

\<|\>   {
yylval.a_char = yytext[0];
next_number_type = A_CONSTANT;
return A_COMPARATOR;
}



\; {
BEGIN(INITIAL);
return yytext[0];
}

: {
next_number_type = A_NUMBER;
return yytext[0];
}
, {return yytext[0];}
[()] {
return yytext[0];
}

\r\n {}
\n {}
. {}
%%
```

B.3 Relational Algebra Data Model

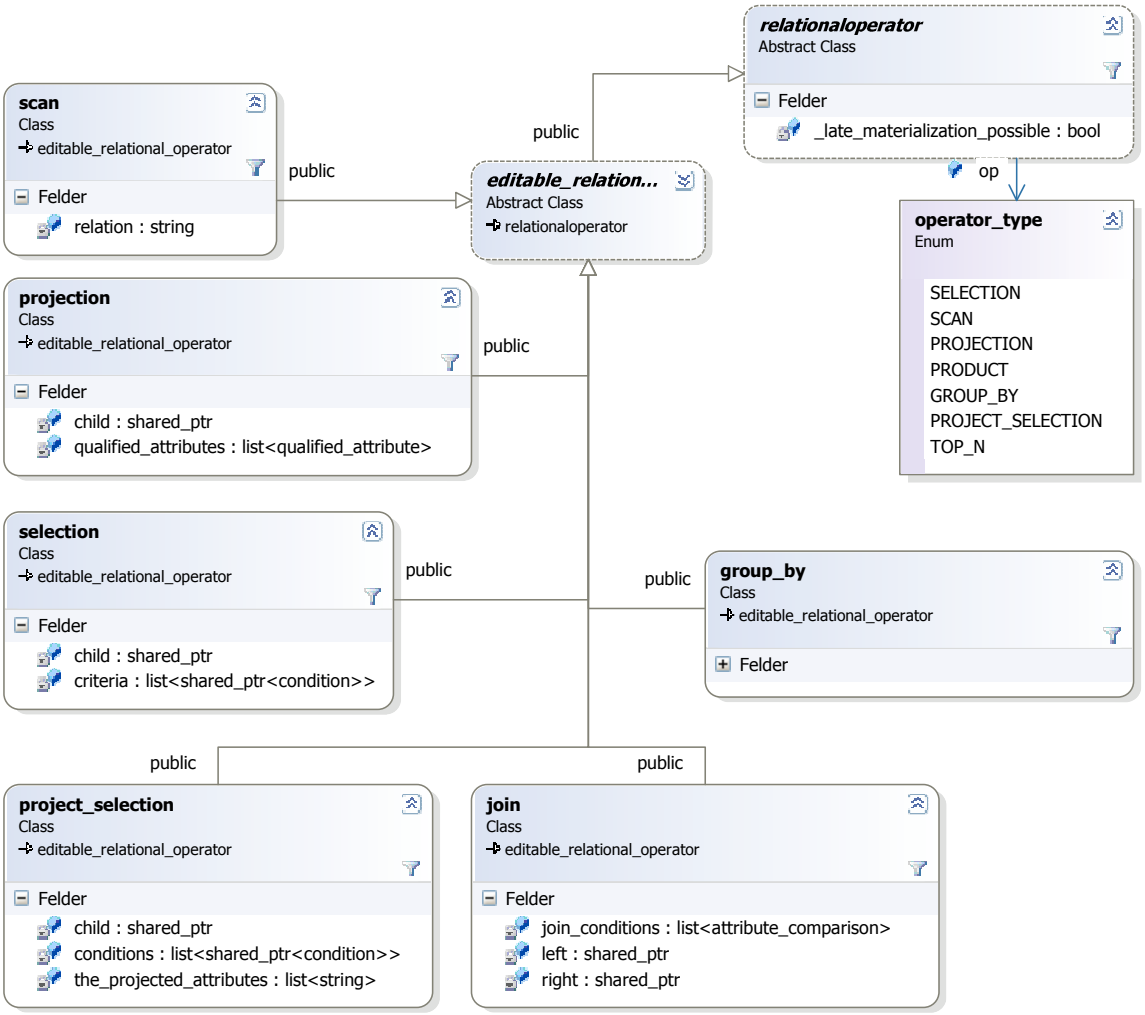


Figure B.1: The UML Diagram of the Classes of the Relational Algebra

B.4 Benchmark Schema

```
CREATE TABLE "ADRC" (
    "CLIENT" character varying(3) NOT NULL,
    "ADDRNUMBER" character varying(10) NOT
        NULL,
    "DATE_FROM" character varying(8) NOT
        NULL,
    "NATION" character varying(1) NOT NULL,
    "DATE_TO" character varying(8),
    "TITLE" character varying(4),
    "NAME1" character varying(40),
    "NAME2" character varying(40),
    "NAME3" character varying(40),
    "NAME4" character varying(40),
    "NAME_TEXT" character varying(50),
    "NAME_CO" character varying(40),
    "CITY1" character varying(40),
    "CITY2" character varying(40),
    "CITY_CODE" character varying(12),
    "CITYP_CODE" character varying(8),
    "HOME_CITY" character varying(40),
    "CITYH_CODE" character varying(12),
    "CHCKSTATUS" character varying(1),
    "REGIOGROUP" character varying(8),
    "POST_CODE1" character varying(10),
    "POST_CODE2" character varying(10),
    "POST_CODE3" character varying(10),
    "PCODE1_EXT" character varying(10),
    "PCODE2_EXT" character varying(10),
    "PCODE3_EXT" character varying(10),
    "PO_BOX" character varying(10),
    "DONT_USE_P" character varying(4),
    "PO_BOX_NUM" character varying(1),
    "PO_BOX_LOC" character varying(40),
    "CITY_CODE2" character varying(12),
    "PO_BOX_REG" character varying(3),
    "PO_BOX_CTY" character varying(3),
    "POSTALAREA" character varying(15),
    "TRANSPZONE" character varying(10),
    "STREET" character varying(60),
    "DONT_USE_S" character varying(4),
    "STREETCODE" character varying(12),
    "STREETABBR" character varying(2),
    "HOUSE_NUM1" character varying(10),
    "HOUSE_NUM2" character varying(10),
    "HOUSE_NUM3" character varying(10),
    "STR_SUPPL1" character varying(40),
    "STR_SUPPL2" character varying(40),
    "STR_SUPPL3" character varying(40),
    "LOCATION" character varying(40),
    "BUILDING" character varying(20),
    "FLOOR" character varying(10),
    "ROOMNUMBER" character varying(10),
    "COUNTRY" character varying(3),
    "LANGU" character varying(1),
    "REGION" character varying(3),
    "ADDR_GROUP" character varying(4),
    "FLAGGROUPS" character varying(1),
    "PERS_ADDR" character varying(1),
    "SORT1" character varying(20),
    "SORT2" character varying(20),
    "SORT_PHN" character varying(20),
    "DEFLT_COMM" character varying(3),
    "TEL_NUMBER" character varying(30),
    "TEL_EXTENS" character varying(10),
    "FAX_NUMBER" character varying(30),
    "FAX_EXTENS" character varying(10),
    "FLAGCOMM2" character varying(1),
    "FLAGCOMM3" character varying(1),
    "FLAGCOMM4" character varying(1),
    "FLAGCOMM5" character varying(1),
    "FLAGCOMM6" character varying(1),
    "FLAGCOMM7" character varying(1),
    "FLAGCOMM8" character varying(1),
    "FLAGCOMM9" character varying(1),
    "FLAGCOMM10" character varying(1),
    "FLAGCOMM11" character varying(1),
    "FLAGCOMM12" character varying(1),
    "FLAGCOMM13" character varying(1),
    "ADDRORIGIN" character varying(4),
    "MC_NAME1" character varying(25),
    "MC_CITY1" character varying(25),
    "MC_STREET" character varying(25),
    "EXTENSION1" character varying(40),
    "EXTENSION2" character varying(40),
    "TIME_ZONE" character varying(6),
    "TAXJURCODE" character varying(15),
    "ADDRESS_ID" character varying(10),
    "LANGU_CREA" character varying(1)
);


CREATE TABLE "KNA1" (
    "MANDT" character varying(3) NOT NULL,
    "KUNNR" character varying(10) NOT NULL,
    "LAND1" character varying(3),
    "NAME1" character varying(35),
    "NAME2" character varying(35),
    "ORT01" character varying(35),
    "PSTLZ" character varying(10),
    "REGIO" character varying(3),
    "SORTL" character varying(10),
    "STRAS" character varying(35),
    "TELF1" character varying(16),
    "TELFX" character varying(31),
    "XCPDK" character varying(1),
    "ADRNR" character varying(10),
    "MCOD1" character varying(25),
    "MCOD2" character varying(25),
    "MCOD3" character varying(25),
    "ANRED" character varying(15),
    "AUFSD" character varying(2),
    "BAHNE" character varying(25),
    "BAHNS" character varying(25),
    "BBBNR" character varying(7),
    "BBSNR" character varying(5),
    "BEGRU" character varying(4),
    "BRSCH" character varying(4),
    "BUBKZ" character varying(1),
    "DATLT" character varying(14),
    "ERDAT" character varying(8),
    "ERNAM" character varying(12),
    "EXABL" character varying(1),
    "FAKSD" character varying(2),
    "FISKN" character varying(10),
    "KNAZK" character varying(2),
    "KNRZA" character varying(10),
    "KONZS" character varying(10),
    "KTOKD" character varying(4),
```

```
    "KUKLA" character varying(2),
    "LIFNR" character varying(10),
    "LIFSD" character varying(2),
    "LOCCO" character varying(10),
    "LOEVM" character varying(1),
    "NAME3" character varying(35),
    "NAME4" character varying(35),
    "NIELS" character varying(2),
    "ORT02" character varying(35),
    "PFACH" character varying(10),
    "PSTL2" character varying(10),
    "COUNC" character varying(3),
    "CITYC" character varying(4),
    "RPMKR" character varying(5),
    "SPERR" character varying(1),
    "SPRAS" character varying(1),
    "STCD1" character varying(16),
    "STCD2" character varying(11),
    "STKZA" character varying(1),
    "STKZU" character varying(1),
    "TELBX" character varying(15),
    "TELF2" character varying(16),
    "TELTX" character varying(30),
    "TELX1" character varying(30),
    "LZONE" character varying(10),
    "XZEMP" character varying(1),
    "VBUND" character varying(6),
    "STCEG" character varying(20),
    "DEAR1" character varying(1),
    "DEAR2" character varying(1),
    "DEAR3" character varying(1),
    "DEAR4" character varying(1),
    "DEAR5" character varying(1),
    "GFORM" character varying(2),
    "BRAN1" character varying(10),
    "BRAN2" character varying(10),
    "BRAN3" character varying(10),
    "BRAN4" character varying(10),
    "BRAN5" character varying(10),
    "EKONT" character varying(10),
    "UMSAT" numeric(8,2),
    "UMJAH" character varying(4),
    "UWAER" character varying(5),
    "JMZAH" character varying(6),
    "JMJAH" character varying(4),
    "KATR1" character varying(2),
    "KATR2" character varying(2),
    "KATR3" character varying(2),
    "KATR4" character varying(2),
    "KATR5" character varying(2),
    "KATR6" character varying(3),
    "KATR7" character varying(3),
    "KATR8" character varying(3),
    "KATR9" character varying(3),
    "KATR10" character varying(3),
    "STKZN" character varying(1),
    "UMSA1" numeric(15,2),
    "TXJCD" character varying(15),
    "PERIV" character varying(2),
    "ABRVW" character varying(3),
    "INSPBYDEBI" character varying(1),
    "INSPATDEBI" character varying(1),
    "KTOCD" character varying(4),
    "PFORT" character varying(35),
    "WERKS" character varying(4),
    "DTAMS" character varying(1),
    "DTAWS" character varying(2),
    "DUEFL" character varying(1),
    "HZUOR" character varying(2),
    "SPERZ" character varying(1),
    "ETIKG" character varying(10),
    "CIVVE" character varying(1),
    "MILVE" character varying(1),
    "KDKG1" character varying(2),
    "KDKG2" character varying(2),
    "KDKG3" character varying(2),
    "KDKG4" character varying(2),
    "KDKG5" character varying(2),
    "XKNZA" character varying(1),
    "FITYP" character varying(2),
    "STCDT" character varying(2),
    "STCD3" character varying(18),
    "STCD4" character varying(18),
    "XICMS" character varying(1),
    "XXIPI" character varying(1),
    "XSUBT" character varying(3),
    "CFOPC" character varying(2),
    "TXLW1" character varying(3),
    "TXLW2" character varying(3),
    "CCC01" character varying(1),
    "CCC02" character varying(1),
    "CCC03" character varying(1),
    "CCC04" character varying(1),
    "CASSD" character varying(2),
    "KNURL" character varying(132),
    "J_1KFREPRE" character varying(10),
    "J_1KFTBUS" character varying(30),
    "J_1KFTIND" character varying(30),
    "CONFS" character varying(1),
    "UPDAT" character varying(8),
    "UPTIM" character varying(6),
    "NODEL" character varying(1),
    "DEAR6" character varying(1),
    "/VSO/R_PALHGT" numeric(13,3),
    "/VSO/R_PAL_UL" character varying(3),
    "/VSO/R_PK_MAT" character varying(1),
    "/VSO/R_MATPAL" character varying(18),
    "/VSO/R_I_NO_LYR" character varying(2),
    "/VSO/R_ONE_MAT" character varying(1),
    "/VSO/R_ONE_SORT" character varying(1),
    "/VSO/R_ULD_SIDE" character varying(1),
    "/VSO/R_LOAD_PREF" character varying(1
        ),
    "/VSO/R_DPOINT" character varying(10),
    "ALC" character varying(8),
    "PMT_OFFICE" character varying(5),
    "PSOFG" character varying(10),
    "PSOIS" character varying(20),
    "PSON1" character varying(35),
    "PSON2" character varying(35),
    "PSON3" character varying(35),
    "PSOVN" character varying(35),
    "PSOTL" character varying(20),
    "PSOHS" character varying(6),
    "PSOST" character varying(28),
    "PSOO1" character varying(50),
    "PSOO2" character varying(50),
    "PSOO3" character varying(50),
    "PSOO4" character varying(50),
    "PSOO5" character varying(50)
);

CREATE TABLE "MAKT" (
    "MANDT" character varying(3) NOT NULL,
    "MATNR" character varying(18) NOT NULL,
```

```
    "SPRAS" character varying(1) NOT NULL,
    "MAKTX" character varying(40),
    "MAKTG" character varying(40)
);


CREATE TABLE "MARA" (
    "MANDT" character varying(3) NOT NULL,
    "MATNR" character varying(18) NOT NULL,
    "ERSDA" character varying(8),
    "ERNAM" character varying(12),
    "LAEDA" character varying(8),
    "AENAM" character varying(12),
    "VPSTA" character varying(15),
    "PSTAT" character varying(15),
    "LVORM" character varying(1),
    "MTART" character varying(4),
    "MBRSH" character varying(1),
    "MATKL" character varying(9),
    "BISMT" character varying(18),
    "MEINS" character varying(3),
    "BSTME" character varying(3),
    "ZEINR" character varying(22),
    "ZEIAR" character varying(3),
    "ZEIVR" character varying(2),
    "ZEIFO" character varying(4),
    "AESZN" character varying(6),
    "BLATT" character varying(3),
    "BLANZ" character varying(3),
    "FERTH" character varying(18),
    "FORMT" character varying(4),
    "GROES" character varying(32),
    "WRKST" character varying(48),
    "NORMT" character varying(18),
    "LABOR" character varying(3),
    "EKWSL" character varying(4),
    "BRGEW" numeric(13,3),
    "NTGEW" numeric(13,3),
    "GEWEI" character varying(3),
    "VOLUM" numeric(13,3),
    "VOLEH" character varying(3),
    "BEHVO" character varying(2),
    "RAUBE" character varying(2),
    "TEMPB" character varying(2),
    "DISST" character varying(3),
    "TRAGR" character varying(4),
    "STOFF" character varying(18),
    "SPART" character varying(2),
    "KUNNR" character varying(10),
    "EANNR" character varying(13),
    "WESCH" numeric(13,3),
    "BWVOR" character varying(1),
    "BWSCL" character varying(1),
    "SAISO" character varying(4),
    "ETIAR" character varying(2),
    "ETIFO" character varying(2),
    "ENTAR" character varying(1),
    "EAN11" character varying(18),
    "NUMTP" character varying(2),
    "LAENG" numeric(13,3),
    "BREIT" numeric(13,3),
    "HOEHE" numeric(13,3),
    "MEABM" character varying(3),
    "PRDHA" character varying(18),
    "AEKLK" character varying(1),
    "CADKZ" character varying(1),
    "QMPUR" character varying(1),
    "ERGEW" numeric(13,3),
    "ERGEI" character varying(3),
    "ERVOL" numeric(13,3),
    "ERVOE" character varying(3),
    "GEWTO" numeric(3,1),
    "VOLTO" numeric(3,1),
    "VABME" character varying(1),
    "KZREV" character varying(1),
    "KZKFG" character varying(1),
    "XCHPF" character varying(1),
    "VHART" character varying(4),
    "FUELG" character varying(2),
    "STFAK" smallint,
    "MAGRV" character varying(4),
    "BEGRU" character varying(4),
    "DATAB" character varying(8),
    "LIQDT" character varying(8),
    "SAISJ" character varying(4),
    "PLGTP" character varying(2),
    "MLGUT" character varying(1),
    "EXTWG" character varying(18),
    "SATNR" character varying(18),
    "ATTYP" character varying(2),
    "KZKUP" character varying(1),
    "KZNFM" character varying(1),
    "PMATA" character varying(18),
    "MSTAE" character varying(2),
    "MSTAV" character varying(2),
    "MSTDE" character varying(8),
    "MSTDV" character varying(8),
    "TAKLV" character varying(1),
    "RBNRM" character varying(9),
    "MHDRZ" character varying(3),
    "MHDHB" character varying(3),
    "MHDLP" character varying(2),
    "INHME" character varying(3),
    "INHAL" numeric(13,3),
    "VPREH" character varying(3),
    "ETIAG" character varying(18),
    "INHBR" numeric(13,3),
    "CMETH" character varying(1),
    "CUOBF" character varying(18),
    "KZUMW" character varying(1),
    "KOSCH" character varying(18),
    "SPROF" character varying(1),
    "NRFHG" character varying(1),
    "MFRPN" character varying(40),
    "MFRNR" character varying(10),
    "BMATN" character varying(18),
    "MPROF" character varying(4),
    "KZWSM" character varying(1),
    "SAITY" character varying(2),
    "PROFL" character varying(3),
    "IHIVI" character varying(1),
    "ILOOS" character varying(1),
    "SERLV" character varying(1),
    "KZGVH" character varying(1),
    "XGCHP" character varying(1),
    "KZEFF" character varying(1),
    "COMPL" character varying(2),
    "IPRKZ" character varying(1),
    "RDMHD" character varying(1),
    "PRZUS" character varying(1),
    "MTPOS_MARA" character varying(4),
    "BFLME" character varying(1),
    "MATFI" character varying(1),
    "CMREL" character varying(1),
    "BBTYP" character varying(1),
```

```
    "SLED_BBD" character varying(1),
    "GTIN_VARIANT" character varying(2),
    "GENNR" character varying(18),
    "RMATP" character varying(18),
    "GDS_RELEVANT" character varying(1),
    "WEORA" character varying(1),
    "HUTYP_DFLT" character varying(4),
    "PILFERABLE" character varying(1),
    "WHSTC" character varying(2),
    "WHMATGR" character varying(4),
    "HNDLCODE" character varying(4),
    "HAZMAT" character varying(1),
    "HUTYP" character varying(4),
    "TARE_VAR" character varying(1),
    "MAXC" numeric(15,3),
    "MAXC_TOL" numeric(3,1),
    "MAXL" numeric(15,3),
    "MAXB" numeric(15,3),
    "MAXH" numeric(15,3),
    "MAXDIM_UOM" character varying(3),
    "HERKL" character varying(3),
    "MFRGR" character varying(8),
    "QQTIME" character varying(2),
    "QQTIMEUOM" character varying(3),
    "QGRP" character varying(4),
    "SERIAL" character varying(4),
    "PS_SMARTFORM" character varying(30),
    "LOGUNIT" character varying(3),
    "CWQREL" character varying(1),
    "CWQPROC" character varying(2),
    "CWQTOLGR" character varying(9),
    "/BEV1/LULEINH" character varying(8),
    "/BEV1/LULDEGRP" character varying(3),
    "/BEV1/NESTRUCCAT" character varying(1)
        ,
    "/DSD/VC_GROUP" character varying(6),
    "/VSO/R_TILT_IND" character varying(1),
    "/VSO/R_STACK_IND" character varying(1)
        ,
    "/VSO/R_BOT_IND" character varying(1),
    "/VSO/R_TOP_IND" character varying(1),
    "/VSO/R_STACK_NO" character varying(3),
    "/VSO/R_PAL_IND" character varying(1),
    "/VSO/R_PAL_OVR_D" numeric(13,3),
    "/VSO/R_PAL_OVR_W" numeric(13,3),
    "/VSO/R_PAL_B_HT" numeric(13,3),
    "/VSO/R_PAL_MIN_H" numeric(13,3),
    "/VSO/R_TOL_B_HT" numeric(13,3),
    "/VSO/R_NO_P_GVH" character varying(2),
    "/VSO/R_QUAN_UNIT" character varying(3)
        ,
    "/VSO/R_KZGVH_IND" character varying(1)
        ,
    "MCOND" character varying(1),
    "RETDELC" character varying(1),
    "LOGLEV_RETO" character varying(1),
    "NSNID" character varying(9),
    "IMATN" character varying(18),
    "PICNUM" character varying(18),
    "BSTAT" character varying(2),
    "COLOR_ATINN" character varying(10),
    "SIZE1_ATINN" character varying(10),
    "SIZE2_ATINN" character varying(10),
    "COLOR" character varying(18),
    "SIZE1" character varying(18),
    "SIZE2" character varying(18),
    "FREE_CHAR" character varying(18),
    "CARE_CODE" character varying(16),
    "BRAND_ID" character varying(4),
    "FIBER_CODE1" character varying(3),
    "FIBER_PART1" character varying(3),
    "FIBER_CODE2" character varying(3),
    "FIBER_PART2" character varying(3),
    "FIBER_CODE3" character varying(3),
    "FIBER_PART3" character varying(3),
    "FIBER_CODE4" character varying(3),
    "FIBER_PART4" character varying(3),
    "FIBER_CODE5" character varying(3),
    "FIBER_PART5" character varying(3),
    "FASHGRD" character varying(4)
);

CREATE TABLE "VBAK" (
    "MANDT" character varying(3) NOT NULL,
    "VBELN" character varying(10) NOT NULL,
    "ERDAT" character varying(8),
    "ERZET" character varying(6),
    "ERNAM" character varying(12),
    "ANGDT" character varying(8),
    "BNDDT" character varying(8),
    "AUDAT" character varying(8),
    "VBTYP" character varying(1),
    "TRVOG" character varying(1),
    "AUART" character varying(4),
    "AUGRU" character varying(3),
    "GWLDT" character varying(8),
    "SUBMI" character varying(10),
    "LIFSK" character varying(2),
    "FAKSK" character varying(2),
    "NETWR" numeric(15,2),
    "WAERK" character varying(5),
    "VKORG" character varying(4),
    "VTWEG" character varying(2),
    "SPART" character varying(2),
    "VKGRP" character varying(3),
    "VKBUR" character varying(4),
    "GSBER" character varying(4),
    "GSKST" character varying(4),
    "GUEBG" character varying(8),
    "GUEEN" character varying(8),
    "KNUMV" character varying(10),
    "VDATU" character varying(8),
    "VPRGR" character varying(1),
    "AUTLF" character varying(1),
    "VBKLA" character varying(9),
    "VBKLT" character varying(1),
    "KALSM" character varying(6),
    "VSBED" character varying(2),
    "FKARA" character varying(4),
    "AWAHR" character varying(3),
    "KTEXT" character varying(40),
    "BSTNK" character varying(20),
    "BSARK" character varying(4),
    "BSTDK" character varying(8),
    "BSTZD" character varying(4),
    "IHREZ" character varying(12),
    "BNAME" character varying(35),
    "TELF1" character varying(16),
    "MAHZA" character varying(2),
    "MAHDT" character varying(8),
    "KUNNR" character varying(10),
    "KOSTL" character varying(10),
    "STAFO" character varying(6),
    "STWAE" character varying(5),
```

```
    "AEDAT" character varying(8),
    "KVGR1" character varying(3),
    "KVGR2" character varying(3),
    "KVGR3" character varying(3),
    "KVGR4" character varying(3),
    "KVGR5" character varying(3),
    "KNUMA" character varying(10),
    "KOKRS" character varying(4),
    "PS_PSP_PNR" character varying(8),
    "KURST" character varying(4),
    "KKBER" character varying(4),
    "KNKLI" character varying(10),
    "GRUPP" character varying(4),
    "SBGRP" character varying(3),
    "CTLPC" character varying(3),
    "CMWAE" character varying(5),
    "CMFRE" character varying(8),
    "CMNUP" character varying(8),
    "CMNGV" character varying(8),
    "AMTBL" numeric(15,2),
    "HITYP_PR" character varying(1),
    "ABRVW" character varying(3),
    "ABDIS" character varying(1),
    "VGBEL" character varying(10),
    "OBJNR" character varying(22),
    "BUKRS_VF" character varying(4),
    "TAXK1" character varying(1),
    "TAXK2" character varying(1),
    "TAXK3" character varying(1),
    "TAXK4" character varying(1),
    "TAXK5" character varying(1),
    "TAXK6" character varying(1),
    "TAXK7" character varying(1),
    "TAXK8" character varying(1),
    "TAXK9" character varying(1),
    "XBLNR" character varying(16),
    "ZUONR" character varying(18),
    "VGTYP" character varying(1),
    "KALSM_CH" character varying(6),
    "AGRZR" character varying(2),
    "AUFNR" character varying(12),
    "QMNUM" character varying(12),
    "VBELN_GRP" character varying(10),
    "SCHEME_GRP" character varying(4),
    "ABRUF_PART" character varying(1),
    "ABHOD" character varying(8),
    "ABHOV" character varying(6),
    "ABHOB" character varying(6),
    "RPLNR" character varying(10),
    "VZEIT" character varying(6),
    "STCEG_L" character varying(3),
    "LANDTX" character varying(3),
    "XEGDR" character varying(1),
    "ENQUEUE_GRP" character varying(1),
    "DAT_FZAU" character varying(8),
    "FMBDAT" character varying(8),
    "VSNMR_V" character varying(12),
    "HANDLE" character varying(22),
    "PROLI" character varying(3),
    "CONT_DG" character varying(1),
    "CRM_GUID" character varying(70),
    "SWENR" character varying(8),
    "SMENR" character varying(8),
    "PHASE" character varying(11),
    "MTLAUR" character varying(1),
    "STAGE" character varying(4),
    "HB_CONT_REASON" character varying(2),
    "HB_EXPDATE" character varying(8),
    "HB_RESDATE" character varying(8),
    "LOGSYSB" character varying(10),
    "KALCD" character varying(6),
    "MULTI" character varying(1)
);

CREATE TABLE "VBAP" (
    "MANDT" character varying(3) NOT NULL,
    "VBELN" character varying(10) NOT NULL,
    "POSNR" character varying(6) NOT NULL,
    "MATNR" character varying(18),
    "MATWA" character varying(18),
    "PMATN" character varying(18),
    "CHARG" character varying(10),
    "MATKL" character varying(9),
    "ARKTX" character varying(40),
    "PSTYV" character varying(4),
    "POSAR" character varying(1),
    "LFREL" character varying(1),
    "FKREL" character varying(1),
    "UEPOS" character varying(6),
    "GRPOS" character varying(6),
    "ABGRU" character varying(2),
    "PRODH" character varying(18),
    "ZWERT" numeric(13,2),
    "ZMENG" numeric(13,3),
    "ZIEME" character varying(3),
    "UMZIZ" character varying(5),
    "UMZIN" character varying(5),
    "MEINS" character varying(3),
    "SMENG" numeric(13,3),
    "ABLFZ" numeric(13,3),
    "ABDAT" character varying(8),
    "ABSFZ" numeric(13,3),
    "POSEX" character varying(6),
    "KDMAT" character varying(35),
    "KBVER" character varying(2),
    "KEVER" character varying(2),
    "VKGRU" character varying(3),
    "VKAUS" character varying(3),
    "GRKOR" character varying(3),
    "FMENG" character varying(1),
    "UEBTK" character varying(1),
    "UEBTO" numeric(3,1),
    "UNTTO" numeric(3,1),
    "FAKSP" character varying(2),
    "ATPKZ" character varying(1),
    "RKFKF" character varying(1),
    "SPART" character varying(2),
    "GSBER" character varying(4),
    "NETWR" numeric(15,2),
    "WAERK" character varying(5),
    "ANTLF" character varying(1),
    "KZTLF" character varying(1),
    "CHSPL" character varying(1),
    "KWMENG" numeric(15,3),
    "LSMENG" numeric(15,3),
    "KBMENG" numeric(15,3),
    "KLMENG" numeric(15,3),
    "VRKME" character varying(3),
    "UMVKZ" character varying(3),
    "UMVKN" character varying(3),
    "BRGEW" numeric(15,3),
    "NTGEW" numeric(15,3),
    "GEWEI" character varying(3),
    "VOLUM" numeric(15,3),
```

```
"VOLEH" character varying(3),
"VBELV" character varying(10),
"POSNV" character varying(6),
"VGBEL" character varying(10),
"VGPOS" character varying(6),
"VOREF" character varying(1),
"UPFLU" character varying(1),
"ERLRE" character varying(1),
"LPRIO" character varying(2),
"WERKS" character varying(4),
"LGORT" character varying(4),
"VSTEL" character varying(4),
"ROUTE" character varying(6),
"STKEY" character varying(1),
"STDAT" character varying(8),
"STLNR" character varying(8),
"STPOS" character varying(3),
"AWAHR" character varying(3),
"ERDAT" character varying(8),
"ERNAM" character varying(12),
"ERZET" character varying(6),
"TAXM1" character varying(1),
"TAXM2" character varying(1),
"TAXM3" character varying(1),
"TAXM4" character varying(1),
"TAXM5" character varying(1),
"TAXM6" character varying(1),
"TAXM7" character varying(1),
"TAXM8" character varying(1),
"TAXM9" character varying(1),
"VBEAF" numeric(5,2),
"VBEAV" numeric(5,2),
"VGREF" character varying(1),
"NETPR" numeric(11,2),
"KPEIN" character varying(3),
"KMEIN" character varying(3),
"SHKZG" character varying(1),
"SKTOF" character varying(1),
"MTVFP" character varying(2),
"SUMBD" character varying(1),
"KONDM" character varying(2),
"KTGRM" character varying(2),
"BONUS" character varying(2),
"PROVG" character varying(2),
"EANNR" character varying(13),
"PRSOK" character varying(1),
"BWTAR" character varying(10),
"BWTEX" character varying(1),
"XCHPF" character varying(1),
"XCHAR" character varying(1),
"LFMNG" numeric(13,3),
"STAFO" character varying(6),
"WAVWR" numeric(13,2),
"KZWI1" numeric(13,2),
"KZWI2" numeric(13,2),
"KZWI3" numeric(13,2),
"KZWI4" numeric(13,2),
"KZWI5" numeric(13,2),
"KZWI6" numeric(13,2),
"STCUR" numeric(9,5),
"AEDAT" character varying(8),
"EAN11" character varying(18),
"FIXMG" character varying(1),
"PRCTR" character varying(10),
"MVGR1" character varying(3),
"MVGR2" character varying(3),
"MVGR3" character varying(3),
"MVGR4" character varying(3),
"MVGR5" character varying(3),
"KMPMG" numeric(13,3),
"SUGRD" character varying(4),
"SOBKZ" character varying(1),
"VPZUO" character varying(1),
"PAOBJNR" character varying(10),
"PS_PSP_PNR" character varying(8),
"AUFNR" character varying(12),
"VPMAT" character varying(18),
"VPWRK" character varying(4),
"PRBME" character varying(3),
"UMREF" character varying(32),
"KNTTP" character varying(1),
"KZVBR" character varying(1),
"SERNR" character varying(8),
"OBJNR" character varying(22),
"ABGRS" character varying(6),
"BEDAE" character varying(4),
"CMPRE" numeric(11,2),
"CMTFG" character varying(1),
"CMPNT" character varying(1),
"CMKUA" numeric(9,5),
"CUOBJ" character varying(18),
"CUOBJ_CH" character varying(18),
"CEPOK" character varying(1),
"KOUPD" character varying(1),
"SERAIL" character varying(4),
"ANZSN" character varying(4),
"NACHL" character varying(1),
"MAGRV" character varying(4),
"MPROK" character varying(1),
"VGTYP" character varying(1),
"PROSA" character varying(1),
"UEPVW" character varying(1),
"KALNR" character varying(12),
"KLVAR" character varying(4),
"SPOSN" character varying(4),
"KOWRR" character varying(1),
"STADAT" character varying(8),
"EXART" character varying(2),
"PREFE" character varying(1),
"KNUMH" character varying(10),
"CLINT" character varying(10),
"CHMVS" character varying(3),
"STLTY" character varying(1),
"STLKN" character varying(8),
"STPOZ" character varying(8),
"STMAN" character varying(1),
"ZSCHL_K" character varying(6),
"KALSM_K" character varying(6),
"KALVAR" character varying(4),
"KOSCH" character varying(18),
"UPMAT" character varying(18),
"UKONM" character varying(2),
"MFRGR" character varying(8),
"PLAVO" character varying(4),
"KANNR" character varying(35),
"CMPRE_FLT" character varying(32),
"ABFOR" character varying(2),
"ABGES" character varying(32),
"J_1BCFOP" character varying(10),
"J_1BTAXLW1" character varying(3),
"J_1BTAXLW2" character varying(3),
"J_1BTXSDC" character varying(2),
"WKTNR" character varying(10),
"WKTPS" character varying(6),
```

```
    "SKOPF" character varying(18),
    "KZBWS" character varying(1),
    "WGRU1" character varying(18),
    "WGRU2" character varying(18),
    "KNUMA_PI" character varying(10),
    "KNUMA_AG" character varying(10),
    "KZFME" character varying(1),
    "LSTANR" character varying(1),
    "TECHS" character varying(12),
    "MWSBP" numeric(13,2),
    "BERID" character varying(10),
    "PCTRF" character varying(10),
    "LOGSYS_EXT" character varying(10),
    "J_1BTAXLW3" character varying(3),
    "/BEV1/SRFUND" character varying(2),
    "FERC_IND" character varying(4),
    "KOSTL" character varying(10),
    "FONDS" character varying(10),
    "FISTL" character varying(16),
    "FKBER" character varying(16),
    "GRANT_NBR" character varying(20)
);
```

Bibliography

[1] *Intel© 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, Intel Corporation, June 2009.

[2] S. Manegold, P. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 191–202.

[3] M. S. et al, "C-store: A column-oriented dbms," in *Proceedings of the 31st international conference on Very large data bases*, 2005.

[4] M. Stonebraker, "The Design of the POSTGRES Storage System," in *Proceedings of the 13th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1987, p. 300.

[5] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009, pp. 1–2.

[6] D. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008.

[7] D. Abadi, D. Myers, D. DeWitt, and S. Madden, "Materialization strategies in a column-oriented DBMS," in *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007*, 2007, pp. 466–475.

[8] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[9] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis, *Fundamentals of data warehouses*. Springer Verlag, 2003.

[10] W. J. Labio, R. Yerneni, and H. Garcia-molina, "Shrinking the warehouse update window," in *In Proceedings of SIGMOD*, 1998, pp. 383–394.

[11] B. Inmon, "Operational and informational reporting," DM Review Magazine, 2000.

[12] S. Brobst and A. Venkatesa, "Active Warehousing," *Teradata Magazine*, vol. 2, no. 1, 1999.

[13] J. Kiviniemi, A. Wolski, A. Pesonen, and J. Arminen, "Lazy aggregates for real-time OLAP," *Lecture notes in computer science*, pp. 165–172, 1999.

[14] R. Ramamurthy, D. J. DeWitt, and Q. Su, "A case for fractured mirrors," in *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 430–441.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns*. Addison-Wesley Reading, MA, 1995.

[16] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 359–370.

[17] R. Hankins and J. Patel, "Data morphing: An adaptive, cache-conscious storage technique," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 417–428.

[18] W. W. Chu and I. T. Ieong, "A transaction-based approach to vertical partitioning for relational database systems," *IEEE Trans. Softw. Eng.*, vol. 19, no. 8, pp. 804–812, 1993.

[19] S. Navathe and M. Ra, "Vertical partitioning for database design: a graphical algorithm," *ACM SIGMOD Record*, vol. 18, no. 2, pp. 440–450, 1989.

[20] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," in *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1985, pp. 268–279.

[21] J. Hoffer and D. Severance, "The use of cluster analysis in physical data base design," in *Proceedings of the 1st International Conference on Very Large Data Bases*. ACM New York, NY, USA, 1975, pp. 69–86.

[22] J. A. Hoffer, "A clustering approach to the generation of subfiles for the design of a computer data base." Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 1975.

[23] J. A. Hoffer and D. G. Severance, "The use of cluster analysis in physical data base design," in *VLDB '75: Proceedings of the 1st International Conference on Very Large Data Bases*. New York, NY, USA: ACM, 1975, pp. 69–86.

[24] B. He, Y. Li, Q. Luo, and D. Yang, "EaseDB: a cache-oblivious in-memory query processor," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 2007, pp. 1064–1066.

[25] P. Boncz, S. Manegold, and M. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, 1999, pp. 54–65.

[26] C. Team, "In-memory data management for consumer transactions the timesten approach," *ACM SIGMOD Record*, vol. 28, no. 2, pp. 528–529, 1999.

[27] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, 1992.

[28] T. Lehman and M. Carey, "Query processing in main memory database management systems," in *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*. ACM, 1986, p. 250.

[29] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2005, pp. 225–237.

[30] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman, "MonetDB/X100 - A DBMS In The CPU Cache," *IEEE Data Engineering Bulletin*, vol. 28, no. 2, pp. 17–22, June 2005.

[31] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood, "DBMSs on a modern processor: Where does time go?" in *Proceedings of the International Conference on Very Large Data Bases*. Citeseer, 1999, pp. 266–277.

[32] H. Messmer and K. Dembowski, *PC-Hardwarebuch*. Addison-Wesley Bonn etc, 1995.

[33] J. Hennessy and D. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[34] R. Prasad and C. Dovrolis, "Bandwidth estimation: metrics, measurement techniques, and tools," *IEEE network*, vol. 17, no. 6, pp. 27–35, 2003.

[35] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proceedings of the 26th annual international symposium on Computer architecture*. IEEE Computer Society Washington, DC, USA, 1999, pp. 222–233.

[36] I. AMD, "AMD64 Architecture Programmers Manual–Volume 2: System Programming," *Rev*, vol. 3, p. 168, 2007.

[37] W. Lin, S. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society Washington, DC, USA, 2001, p. 301.

[38] R. Hedge, "Optimizing application performance on Intel Core microarchitecture using hardware-implemented prefetchers," 2007.

[39] I. Corporation, "Ibm soliddb universal cache," IBM Corporation, Tech. Rep., 2009.

[40] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation techniques for main memory database systems," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 1984, pp. 1–8.

[41] D. Mortensen and J. Sheth, "Burst mode data block transfer system," Sep. 17 1985, uS Patent 4,542,457.

[42] P. Zagar, B. Williams, and T. Manning, "Burst EDO memory device," Jun. 11 1996, uS Patent 5,526,320.

[43] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw-Hill, 2003.

[44] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 671–682. [Online]. Available: http://dx.doi.org/http://doi.acm.org/10.1145/1142473.1142548

[45] A. Ailamaki, D. DeWitt, and M. Hill, "Data page layouts for relational databases on deep memory hierarchies," *The VLDB Journal The International Journal on Very Large Data Bases*, vol. 11, no. 3, pp. 198–215, 2002.

[46] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis, "Weaving relations for cache performance," *The VLDB Journal*, pp. 169–180, 2001.

[47] T. Lehman and M. Carey, "A study of index structures for main memory database management systems," in *Conference on Very Large Data Bases*, vol. 294, 1986.

[48] A. Cárdenas, "Analysis and performance of inverted data base structures," *Communications of the ACM*, vol. 18, no. 5, May 1975.

[49] M. Grund, J. Krueger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "Hyrise - a main memory hybrid storage engine," 2009, unpublished Manuscript. Submitted for Publication.

[50] L. Getoor, B. Taskar, and D. Koller, "Selectivity estimation using probabilistic models," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 461–472, 2001.

[51] V. Poosala, P. Haas, Y. Ioannidis, and E. Shekita, "Improved histograms for selectivity estimation of range predicates," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 294–305, 1996.

[52] Y.-W. Huang, N. Jing, and E. A. Rundensteiner, "A cost model for estimating the performance of spatial joins using r-trees," *Scientific and Statistical Database Management, International Conference on*, vol. 0, p. 30, 1997.

[53] J. McHugh and J. Widom, "Query optimization for XML," in *Proceedings of the International Conference on Very Large Data Bases*. Citeseer, 1999, pp. 315–326.

[54] S. Listgarten and M. Neimat, "Modelling Costs for a MM-DBMS," in *Proc. of the Intl. Workshop on Real-Time Databases, Issues and Applications*, 1996, pp. 72–78.

[55] K. Whang, "Query optimization in a memory-resident domain relational calculus database system," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 1, pp. 67–95, 1990.

[56] K. McKinley and S. Carr, "Improving data locality with loop transformations," *ACM Transactions on Programming Languages and Systems*, 1996.

[57] G. Diehr and A. Saharia, "Estimating block accesses in database organizations," *IEEE Transactions on Knowledge and Data Engineering*, Jan 1994.

[58] S. Yao, "Approximating block accesses in database organizations," *Communications of the ACM*, vol. 20, no. 4, Apr 1977.

[59] T.-Y. Cheung, "Estimating block accesses and number of records in file management," *Communications of the ACM*, vol. 25, no. 7, Jul 1982.

[60] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Cooperative scans: dynamic bandwidth sharing in a dbms," *Proceedings of the 33rd international conference on Very large data base*, Jan 2007.

[61] F. Hillier and G. Lieberman, *Introduction to operations research*. McGraw-Hill, 2005.

[62] H. J. Greenberg, "Klee-minty polytope shows exponential time complexity of simplex method," 1997, unpublished Manuscript.

[63] D. Spielman and S. Teng, "Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time," *Journal of the ACM*, vol. 51, no. 3, pp. 385–463, 2004.

[64] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, 1984.

[65] J. Clausen, "Branch and bound algorithms-principles and examples," *Parallel Computing in Optimization*, pp. 239–267, 1997.

[66] *Shark User Guide*, Apple Inc., 1 Infinite Loop, Cupertino, CA 95014, April 2008, available at: http://developer.apple.com/documentation/DeveloperTools/Conceptual/SharkUserGuide/SharkUserGuide.pdf.

[67] D. Mosberger and D. Dugger, "IA-64 Linux kernel internals," *URL http://www. linuxia64. org*, 2000.