Balancing Vectorized Query Execution with Bandwidth-Optimized Storage

Marcin Żukowski

Balancing Vectorized Query Execution with Bandwidth-Optimized Storage

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Agnietenkapel
op vrijdag 11 september 2009, te 10:00 uur

door

Marcin Żukowski
geboren te Białystok, Polen

Promotiecommissie:

Promotor:	prof. dr. M.L. Kersten
Copromotor:	dr. P.A. Boncz
Overige leden:	dr. G. Graefe
	prof. dr. A. Ailamaki
	prof. dr. ir. A.W.M. Smeulders
	prof. dr. C.R. Jesshope

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The research reported in this thesis was carried out at CWI, the Dutch national research centre for mathematics and computer science, within the theme *Database Architectures and Information Access*, a subdivision of the research cluster *Information Systems*.

The research reported in this thesis was partially carried out and is being continued at VectorWise B.V., a CWI spin-off company.

SIKS Dissertation Series No. 2009-30
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis was partially funded by the MultimediaN N3 project.

Cover design: Bartek Wąsiel, Fabryka BeWu, Poznań, Poland

ISBN: 978-90-9024564-5

Wszystkim moim nauczycielom,
zwłaszcza tym pierwszym i najlepszym –
Rodzicom

To all my teachers,
especially the very first and the best ones –
My Parents

Contents

Acknowledgments

I had two supervisors at CWI. Martin Kersten always kept an eye on what this new student from Poland was doing, and teased me with hard problems and tricky questions when appropriate.

Still, it was Peter Boncz who had to live with all my questions, ideas, problems and complaints almost every day. The discussions we had were one of the most thought-stimulating moments of my life, even when we disagreed on something. Many of the ideas described in this book were conceived in these meetings. I think Peter also enjoyed working together, as he wanted to continue our cooperation when it was time for me to leave CWI. Thanks to his persistence and energy, together we started a spin-off company, where we can now keep discussing (and yes, Martin still keeps an eye on us).

This thesis would not exist as it is without two other people: Sándor Héman and Niels Nes. We created a small team, which worked hard on many cool ideas and produced some nice papers. We also missed some deadlines after sleepless nights, but even that was an interesting experience.

I was very lucky to convince truly great researchers to take a closer look at this thesis. I am honored to thank Goetz Graefe, Anastasia Ailamaki, Arnold Smeulders and Chris Jesshope for their remarks on the content and agreeing to become the members of my PhD committee.

The "insane" database group at CWI was a great place to work (and live). I made good friends, and learned a lot from very smart people. With Roberto Cornacchia it was great fun to work with, but also to go out to "the windmill" (yes, the order should be different here, but this is a PhD thesis!). Stefan Manegold always helped, even when he was overloaded, and also provided priceless advice on beer brands. Sjoerd Mullender provided constructive critique on technical ideas and (especially!) on my writing. Arjen de Vries added a bit of less-hackerish flavor to some of the research we did. So, it was always a "work and fun" combination – big "thank you" to all of you: Eefje, Erietta, Jennie,

Marja, Milena, Nina, Theodora, Alex, Arjen, Fabian, Henning, Johan, Lefteris, Matthijs, Nan, Romulo, Stratos, Wouter and, last but not least, Thijs (we will do a 7A one day!).

Work described in this thesis resulted in VectorWise – a spin-off company of CWI. I would like to thank Michał, Giel, Cecilia, Miriam, Dick, Bill, Roger, Fred, Dan and Doug for making it possible and for your continued involvement in this project.

During my PhD I did two internships, first at Microsoft and then at Google. These were both great adventures, where I worked with extremely smart people and made some really good friends. I would especially like to thank Craig Freedman, Paul Yan and Gregory Eitzmann who were my mentors. Many others made these internships fun and interesting: Bart van den Berg, Yuxi Bai, Greg Teather, Sarah Rowe, Florian Waas, Wey Guy, Gargi Sur, Mark Callaghan, Stephan Ellner, Ela Iwaszkiewicz, Robin Zueger, Marius Renn, Gary Huang, Tom Bennett, Alex Chitea and the "Brasilian invasion": Gustavo Moura and Sam Goto (he *is* a programmer with this name!).

Life in Amsterdam was not only work. I would like to thank many people that made these years a great adventure. Elena, Elisa, Dorina, Simona, Simone, Sonja, Christian, Krzysztof, Stefano, Katja, Peter, Maya, Volker, Stephanie, Wouter, David, Alexandra – thanks! Special thanks go to the "Polish mafia": Ania (cmok!), Agnieszka (A. and K.!), Dorota, Gosia, Ghosia, Kasia, Ewa, Iza, Magda, Ola, Justynka, Asia, Ela, Michaś, Koris a.k.a. Misza, Banan, Radziu, Bartek, Miłosz, Klaudiusz, Michał, Paweł (Z. and G.!), Adam, Wojtek (B., R. and W.!), Daniel, Marcin, Zbyszek, Guma, Łukasz. With you, sometimes I felt like I never left Poland.

When moving to Amsterdam, I left many great friends in Poland. We do not meet that often now, but when we do, it is great to feel there are people like them, people I can always return to. I won't list all of them by name... as they all have cool nicknames! Kluczyk, Zenzka, Qleczka, Agmyszka, AngeLinka, P00h, Fox, X-Ray, BeWu, Rzóg, Seban, Kwaz, Bulba – hope to see you soon!

Finally, I would like to thank my Family: my parents, Basia and Marek, my sisters, Agnieszka and Marta, and their close ones, Maciek, Bartek, Antoś and Darek. I know it was not always easy, but I am thankful that you were watching over me, supported all my choices, and were always there for me. Love you all.

Chapter 1

Introduction

The continuous evolution of computer hardware in the past decades has resulted in a rapid increase of available computing power. However, not all application areas benefited from this improvement to the same extent – most new hardware features are targeted at computation-intensive tasks, including computer games, multimedia applications and scientific computing. On the other hand, general-purpose database systems have been shown to have problems with fully exploiting today's hardware performance potential [ADHW99].

Over the last two decades, CPUs evolved from relatively simple, single-pipeline in-order devices that were easy to program into highly complex elements. These new processors introduce technologies like superscalar out-of-order execution, SIMD instructions and multiple cores. To achieve optimal performance on such hardware, the application code needs to follow new hardware-conscious patterns and be amenable to compiler optimizations. Furthermore, the increase of CPU frequencies resulted in an increasing imbalance between the processor speed and memory latency. As a result, computers depend more and more on multi-level *cache memories* that improve the memory access time, but, again, often require the programmer to tune data access patterns in the program.

In disk storage two trends can be observed that introduce new challenges for system designers. First, random disk access latency improves significantly more slowly than sequential disk bandwidth. Secondly, both disk latency and bandwidth improve more slowly than the computing power of modern processors, especially with the advent of multi-core CPUs.

1.1 Problem statement

Database engines have been shown to adapt poorly to the hardware developments presented above, in both query processing and storage layers.

Query execution. In this layer, many database systems continue to follow the *tuple-at-a-time* pipelined model working with N-ary tuples. This makes the CPU spend most time not on the actual data processing, but on traversing the query operator tree. Such program behavior causes problems for modern processors, since it can lead to poor instruction-cache performance and frequent branch mispredictions, significantly reducing the performance. Even worse, the tuple-at-a-time execution model makes it impossible for compilers to apply many performance-critical optimization techniques such as loop-unrolling and SIMDization. This is in contrast with other application areas, such as scientific computing, where *data-intensive* approaches, spending most time on the actual data processing, can be optimized into highly efficient programs. Some of the database performance problems can be partially solved with techniques that have been published in the area of architecture-conscious query processing. However, most of the previous work in this field concentrates on improving isolated problems within an existing execution framework, often limiting the achieved performance gains to single operations.

An alternative approach to query execution has been presented in the MonetDB system. Here, instead of working on single tuples, the system uses *column-at-a-time* materializing operators, internally working as simple operations on arrays of values. This results in *bulk processing*, improving performance by removing the per-tuple interpretation overhead and exposing multiple compiler optimization opportunities. However, the full materialization implied by this model often results in large intermediate results. This causes extra memory or disk traffic and degrades the performance of this system when working with large data volumes. Also, a processing unit of a full column is often too large to apply some of the existing optimization techniques such as memory prefetching. Finally, the column algebra used in this model makes it hard to implement multi-attribute operations, resulting in extra processing steps.

Storage. Also in this layer database systems do not fully adapt to the changing hardware properties. While the relative performance of random I/O with respect to sequential I/O gets worse, many database systems still often rely on random-access methods, such as unclustered indices, even in data-intensive operations. To keep this method efficient, database systems use storage facilities that contain more and more disks to provide enough throughput of random-access operations. This approach is unsustainable in the long run, but it is not always clear how scan-based strategies could replace random-access ones. Additionally, while random-access methods easily scale to handle heavy query loads by using RAID systems and request batching, scalability of scan-based approaches requires more investigation.

Another problem is that with computing power increasing at a faster pace than disk performance, data delivery becomes a bottleneck even with sequential-access approaches. This is especially visible in systems using the *N-ary storage model*, where entire tuples need to be fetched from disk, even if only a small subset of attributes is actually used. An alternative to this model are *column stores*, which only read relevant attributes, requiring lower disk bandwidth. In both storage models disk performance can also be improved with *data compression*. Here, it is crucial that the decompression is highly efficient, so it does not dominate the actual data processing. Yet another challenge is making a system efficiently support multiple concurrent users. In such scenarios, the performance of current systems often degrades due to queries competing for resources, instead of benefiting from the potential of performing the common tasks once for many users.

1.2 Research direction

The above analysis leads to the general research question addressed in this thesis:

> *How can various architecture-conscious optimization techniques be combined to construct a coherent database architecture that efficiently exploits the performance of modern hardware, for both in-memory and disk-based data-intensive problems?*

As the stated research question is very general, the research track presented in this thesis originally focused on improvements to the MonetDB query execution layer. In the author's master's thesis on parallel query execution [Zuk02], the fully materializing approach was identified as a significant performance and scalability problem, and a more iterative approach was proposed, still working within the MonetDB framework.

This idea evolved into a completely new *vectorized in-cache execution* approach that became the core of the MonetDB/X100 system [BZN05, ZBNH05, Zuk05a]. This new approach extends the pipelined model by making the operators work on a set of tuples, represented by *vectors*, each consisting of hundreds or thousands of values of a single attribute. The execution is divided into generic operator logic and specialized, highly efficient data processing primitives similar to the MonetDB operators. This allows the system to achieve the high performance that bulk-processing delivers, without sacrificing system scalability.

After obtaining very high in-memory performance results on the 100GB TPC-H benchmark it became clear that the high processing bandwidth of the query execution layer (reaching over one gigabyte per second on a single CPU core) is hard to match with typical disk systems. This resulted in shifting the focus of this research to the storage layer, with the goal of researching new disk storage techniques and disk access strategies able to satisfy these high requirements [Zuk05b]. Consecutively, this led to the development of a number of methods that improve data delivery performance for scan-based applications, resulting in a *ColumnBM* storage system.

The techniques proposed in this thesis have been evaluated in two application areas: data warehousing and decision support, represented by the TPC-H benchmark [Tra06], and large-scale information retrieval, represented by the Terabyte TREC benchmark [CSS].

1.3 Research questions

The research directions presented above reflect the following set of the underlying research questions:

1. Is it possible to combine the benefits of the tuple-at-a-time model and bulk-processing in a coherent query execution model?

2. What techniques allow database engines to rely more on sequential scans instead of on random I/O?

3. What techniques can improve database I/O performance for individual queries?

4. What techniques can improve database I/O performance for heavy query loads?

This thesis tries to provide answers to these questions. However, it does not look at proposed improvements in isolation, but rather investigates how different optimizations, both new and existing ones, can cooperate within a coherent database architecture. Additionally, it has a goal of making the proposed improvements readily applicable to database systems.

1.4 Research results and thesis outline

The research presented in this book leads to the following thesis statement:

> *With the* ***vectorized execution model*** *database systems can minimize the instructions-per-tuple cost on modern CPUs and achieve high in-memory performance, but* ***bandwidth-optimizing improvements*** *in the storage layer are required to scale this performance to disk-based datasets.*

This thesis statement is supported with the following scientific contributions:

Vectorized in-cache execution model. *Addresses research question 1, parts published in CIDR'05 [BZN05], DAMON'06 [ZHB06] and DAMON'08 [ZNB08], discussed in Chapters 4-5.*
The thesis proposes a new execution model that combines the best properties of the previously applied approaches. Benchmarks have demonstrated that it brings numerous performance benefits, including reduced interpretation overhead and multiple performance optimization opportunities. However, the strict separation of the relational operator logic and the actual data processing, which is the key feature of this model, makes it hard to provide fully vectorized relational operator implementations. This thesis proposes various methods of tackling this problem, presenting how typical processing tasks can be efficiently vectorized. It also introduces new hardware-conscious techniques, for example improved hash-based processing. The resulting execution engine efficiently exploits modern CPUs and cache-memory systems and achieves in-memory performance often one or two orders of magnitude higher than the existing approaches.

Bandwidth-optimizing disk access model. *Addresses research question 2, parts published in CIDR'05 [BZN05], BNCOD'05 PhD Workshop [Zuk05b] and VLDB'07 [ZHNB07], discussed in Chapter 4.*
With the high performance of the vectorized execution kernel, it becomes hard to provide sufficient data delivery bandwidth from disk. With the increasing imbalance between disk latency and bandwidth, strategies based on random disk access are infeasible for most applications processing large data volumes. This thesis discusses a number of approaches that avoid random accesses and allow a scan-mostly query execution model. Additionally, even with scan-based approaches, it is crucial to optimize the use of available disk bandwidth. The DSM storage model, improved with *lightweight compression*, can reduce data volumes that need to be transferred. Additionally, intelligent data sharing be-

tween queries minimize the number of times the same data needs to be fetched from disk.

Ultra-lightweight data compression. [1] *Addresses research question 3, published in ICDE'06 [ZHNB06], discussed in Chapter 6.*
This thesis introduces a set of compression algorithms that allow trading some CPU power for an increased perceived disk bandwidth. This approach is especially useful in column-stores, as the contiguously stored data from the same domain offers good compression opportunities, and only the used columns need to be decompressed. The proposed algorithms are carefully tuned for modern CPUs, achieving decompression bandwidth in the order of gigabytes per second, which is one or two orders of magnitude higher than popular compression solutions. Moreover, they are optimized for the vectorized in-cache execution pipeline: data is decompressed on a vector granularity, and it is materialized only in the CPU cache, from where it is immediately consumed for processing. These two techniques make the decompression overhead minimal, leaving enough CPU time to process the decompressed data. As a result the query performance for disk-based datasets is significantly improved.

Cooperative scans. *Addresses research question 4, published in VLDB'07 [ZHNB07], discussed in Chapter 7.*
Since sequential data access is the preferred access method for data intensive workloads, it is important to optimize scenarios with multiple concurrent queries performing scans at the same time. The introduced "cooperative scans" technique extends the traditional buffer manager by dynamically managing query activity, buffer content and I/O operations to maximize data sharing between queries and minimize disk activity. It outperforms existing *shared scans* methods both in query latency and system throughput, as demonstrated for various scenarios on PAX and DSM datasets.

[1] Joint work with Sándor Héman, parts of this research might appear in his PhD thesis.

Chapter 2

Computer hardware evolution

This chapter presents the aspects of computer hardware that are the most important for the query processing techniques presented in this thesis. A computer-architecture expert might consider the material presented here as only scratching the surface, and he or she might just skim through this chapter. On the other hand, we hope that it provides enough background for database researchers unfamiliar with the low-level hardware details to allow understanding the rationale behind the techniques presented later in this book.

As described in Chapter 1, this thesis focuses on two major aspects of query execution: efficient in-memory query processing and high-performance storage facilities. These two areas map directly on the hardware features described in this chapter. First, in Sections 2.1 and 2.2, we analyze the features of modern CPUs and the hierarchical memory infrastructure. Then, in Section 2.3, we discuss the features of the storage systems. For all discussed areas we provide a short description of the evolution of a particular area, the current state of the art, and the future trends.

Naturally, the material presented in this chapter is in no way exhaustive. For readers interested in more details, we recommend the following resources:

- "Inside the Machine" [Sto07] by Jon Stokes – this book provides an easy to understand introduction to the CPU architecture, including the description of the crucial aspects of the way CPUs process the data: pipelined execution, superscalar execution, SIMD instructions and hierarchical memory infrastructure. It uses the examples from the history of two of the most important CPU families: Intel's Pentium (and the following Core) architectures, and Apple/IBM/Motorola PowerPC line. However, it omits the CPU architecture contributions coming from AMD and SUN processors, and does not discuss the new hybrid processor architectures such as STI's Cell [IBM07] and Intel's Tera-scale [ACJ$^+$07].

- "Computer Architecture – A Quantitative Approach" [HP07] by John L. Hennessy and David A. Patterson – this book provides a wider overview of computer architecture, including not only the CPU internals but also storage facilities. The book is significantly more technical than [Sto07], so it is recommended for readers with some background in these areas. Additionally, the most recent, 4th edition, provides an extensive discussion of the multi-core generation of modern CPUs.

- journals: ACM Transactions on Storage (TOS), ACM Transactions on Architecture and Code Optimization (TACO), ACM Transactions on Computer Systems (TOCS).

- conference proceedings: International Symposium om Computer Architecture (ISCA), International Conference on Supercomputing (ICS), International Conference on Computer Design (ICCD).

2.1 Modern CPU architecture

Over the last few decades, the architecture of CPUs evolved greatly and became extremely complex. Table 2.1 and Figure 2.1 demonstrate the milestones in that evolution, using Intel's CPU line as an example. This section describes the architectural features of the modern CPUs that are directly related to the research presented in this thesis.

2.1.1 Basic CPU computing model

A basic model of a CPU is presented in the left-most side of Figure 2.2. This model is highly simplified – the right-most side of the figure presents a more complex architecture of Pentium 4 CPU. Looking at the basic CPU architecture, the following major components can be identified:

Data is a set of operands for the CPU.

Processor	16-bit address/, bus, micro-coded	32-bit address/ bus, micro-coded	5-stage pipeline, on-chip I&D caches FPU	2-way super-scalar, 64-bit bus	Out-of-order, 3-way super-scalar	Out-of-order, super-pipelined, on-chip L2 cache	Multi-core
Product	80286	80386	80486	Pentium	PentiumPro	Pentium4	CoreDuo
Year	1982	1985	1989	1993	1997	2001	2006
Transistors (thousands)	134	275	1,200	3,100	5,500	42,000	151,600
Latency (clocks)	6	5	5	5	10	22	12
Bus width (bits)	16	32	32	64	64	64	64
Clock rate (MHz)	12.5	16	25	66	200	1500	2333
Bandwidth (MIPS)	2	6	25	132	600	4500	21000
Latency (ns)	320	313	200	76	50	15	5

Table 2.1: Milestones in the CPU development, looking at the Intel's CPU line (based on [Pat04], with the 2006 milestone added)

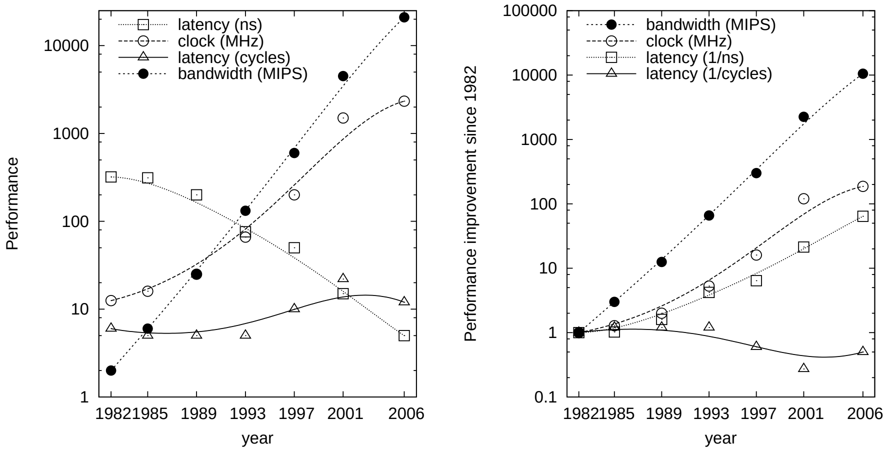


Figure 2.1: Evolution of CPU characteristics

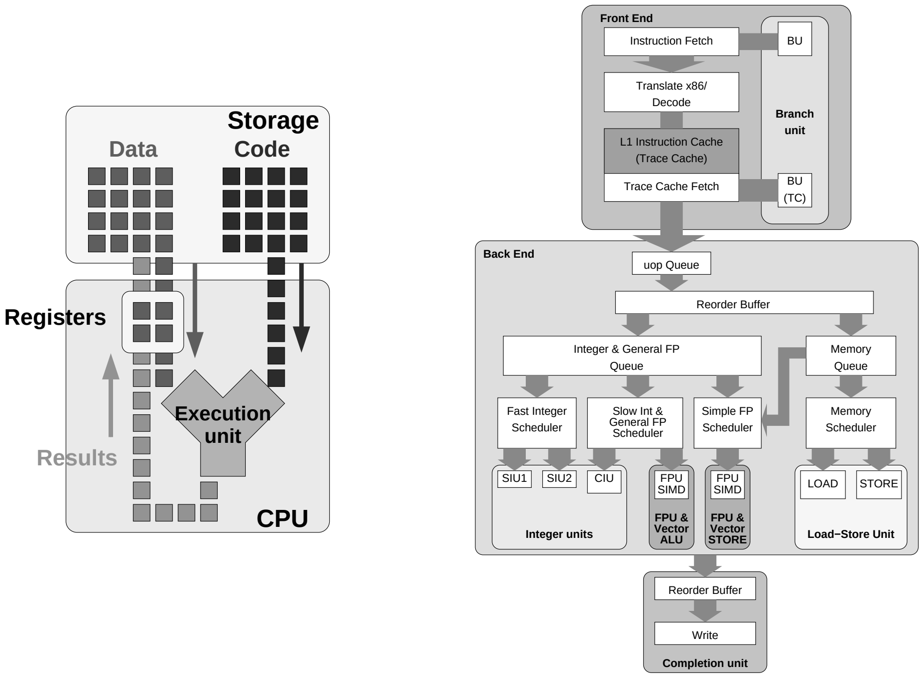


Figure 2.2: A simplified CPU model (left) and a diagram of Pentium 4 CPU (right, from [Sto07])

Code is a set of commands for the CPU, describing what to do with the data

Storage is a container where the data and the code are stored. For now, we can assume it is the main memory. CPUs typically cannot directly work on the data stored there.

Registers are the local CPU storage, used to keep the data the CPU is currently working on. Data can be transferred between registers and storage with special instructions.

Execution unit is a part of a CPU that performs requests tasks on given data, for example addition of two elements. Typically, it operates on data taken from registers, and it also saves the results in a register.

CPU Cycles. CPU operates in *cycles*, which are synchronized by *clock* pulses, issued by an external *clock generator*. The frequency of the clock corresponds with the frequency of the CPU. In our simplified model, in each *CPU cycle* the processor takes a single instruction from the code stream and performs a requested instruction on the execution unit.

ISA. The code that CPU executes consists of a sequence of *instructions*, having different forms, depending on the *instruction set architecture* (ISA) a given CPU supports. Currently, the most popular ISA is *x86* (and its 64-bit extension *x64*) present in CPUs such as Intel's Pentium or the AMD Athlon. While the ISA serves as an interface to a CPU, different CPUs can internally implement instructions from an ISA in different ways. Usually this is performed by translating the ISA opcodes into internal *microcodes* (also known as *microops* or *uops*), which are the actual commands executed by the CPU. This translation is especially important in CISC (complex instruction set computing) CPUs, which often need to translate complex ISA instructions (e.g. string operations in x86) into a sequence of microcodes.

Execution stages. In the basic computing model, every cycle the processor executes the next instruction. This execution can be decomposed into multiple sequentially performed stages. The exact number and the nature of these stages is different for various processors, but usually they follow this general set of stages, presented in the top part of Figure 2.3:

- *Instruction Fetch (IF)* – get the next instruction to execute.
- *Instruction Decode (ID)* – decode the instruction from the binary opcode form into the internal representation. Translation from ISA into microcodes also happens here.
- *Execute (EX)* – perform the requested task. While in a simplified CPU model we assumed one execution unit, this can involve multiple different devices, including arithmetic logic unit (ALU), floating-point unit (FPU), load-store unit (SPU) and more.
- *Write-back (WB)* – save the result of the execution.

Usually, each of these stages is performed by a specialized CPU unit. Since the stages execute in a fully sequential manner, this means that at a given time only one part of the CPU is busy. As a result, the computational resources of the

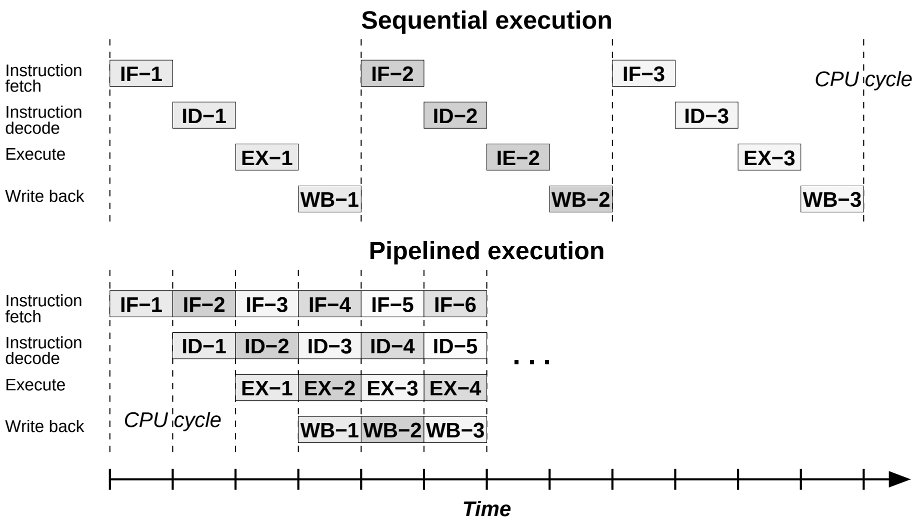


Figure 2.3: Comparison of sequential (top) and pipelined (bottom) instruction execution

CPU are not fully used. For example, when an instruction is being fetched, the *execute* unit is idle. Additionally, if an instruction is to be executed in a single CPU clock cycle, the time of a cycle needs to be long enough to allow all stages to execute, making it harder to increase the frequency of a CPU.

2.1.2 Pipelined execution

To improve the utilization of the CPU resources, the classical sequential execution has been replaced with a *pipelined* execution, presented in the bottom part of Figure 2.3. In this model, different instructions are executed at the same time, performing different processing stages. This keeps units responsible for different stages busy all the time.

Since every stage takes only a fraction of time required by the entire sequence, this also allows shorter CPU clock cycle lengths, and hence higher CPU frequencies. In Figure 2.3 the cycle length decreased perfectly to be one fourth of the original length. However, in real CPUs the lengths of the stages are not exactly the same, making the cycle length higher than the length expected from a simple division of the original length by the number of stages.

Pipelined execution, while not improving the execution latency of a single instruction, significantly improves the *instruction completion rate*, or instruction throughput, of a CPU. In our (simplified) example, this rate is increased four times.

2.1.3 SIMD instructions

Execution units (e.g. ALU) typically work with instructions that perform a given operation for a single set of operands (e.g. a single addition of two integers). This follows the Single-Instruction-Single-Data (SISD) [Fly72] execution model. However, there are many cases in which exactly the same operation needs to be performed for a large number of elements. A simple example is negating an 8-bit grayscale image, stored in memory as a sequence of bytes, each representing a pixel. A straightforward SISD implementation would look as follows:

```
for (i = 0; i < num_pixels; i++)
    output[i] = 255 - input[i];
```

A different approach is to use Single-Instruction-Multiple-Data (SIMD) execution model, where a single instruction can perform the same operation on multiple elements at once. For example, imagine a CPU that has a special SIMD ALU that can perform a subtraction of 8 bytes from a constant with one instruction. Then the code becomes:

```
for (i = 0; i < num_pixels; i += 8 )
    simd_sub_const_vector(output + i, 255, input + i);
```

In this case, thanks to using a SIMD instruction, the loop needs to have 8 times fewer iterations, significantly improving the performance.

SIMD instructions are very useful in areas processing large data volumes, including multimedia processing, 3D modeling and scientific computing. To improve the performance in these areas, most modern CPUs provide some form of SIMD instructions. In particular, the most popular *x86* processors provide a set of SIMD extensions, including MMX, 3DNow! and SSE (versions 1 to 4). This computational model is also a base for GPU-based processing and some specialized processors, e.g. SPUs in a Cell processor (see Section 2.1.7.3).

2.1.4 Superscalar execution

In a classical pipelined execution, only one instruction can be at a given stage of the processing pipeline. To further increase the CPU instruction throughput, modern CPUs use multiple execution units, allowing a “wider” pipeline, with different instructions working on the same stage of processing. This is achieved by extending the number of operands a given execution unit can process (e.g. by making the instruction-fetch unit fetch 4 instructions at once), or by introducing multiple execution units working on the same stage. In our simplified CPU model, the latter can be achieved by having more than one ALU. In modern CPUs there are not only multiple ALUs, but also other execution units for different types of operations, including floating-point units (FPU), memory load-store units (LSU) and SIMD units, possibly few of each.

Typically, the “width” of a superscalar CPU is measured as the number of instruction that can enter the “execute” stage every cycle – this number is usually smaller than the total number of all available execution units.

2.1.5 Hazards

Pipelined and superscalar execution only achieve their full efficiency if the processing pipelines are filled at every moment. To do so, at every CPU cycle the maximum available number of instructions should be dispatched for execution. However, to dispatch an instruction all the prerequisites for it should be matched, including availability of code, data and internal CPU resources. When one of the conditions is not met, the instruction needs to be delayed, and a *pipeline-bubble*, or a no-op instruction, enters the pipeline instead. A bubble in a pipeline causes a suboptimal resource utilization, and in effect reduces the CPU instruction completion ratio.

In this section we discuss various events, usually called “hazards”, that can result in instruction delays and pipeline bubbles.

2.1.5.1 Data hazards

Data hazards are situations when an instruction cannot be executed because some of the inputs for it are not ready. Let us look at the following code snippet.

```
c = a + b;
e = c + d;
```

In this case, the computation of `e` cannot start before the result of `c` is computed. As a result, the second addition is delayed.

CPUs try to limit the impact of data hazards by various techniques. In *forwarding*, the output of a computation from one ALU can be directly passed back as an input to this (or different) ALU, bypassing the register-write phase.

Another technique is *register renaming* that can improve the performance in case of *false register conflicts*. It exploits the fact that processors usually have more physical registers than visible through the ISA. Here is an example of a false register conflict.

```
c = a + b;
a = d + e;
```

At a first glance, the first instruction *has to* read the content of the `a` register, before the second one writes its result to it. However, we can see that these instructions are completely independent. As a result, the CPU can map the `a` register in the first instruction to one physical register, and to another one in the second instruction. Thanks to that, both instructions can execute simultaneously.

While typically not considered data hazards, *cache-misses* also cause execution units to wait for data delivery. This problem is described in Section 2.2.3.

2.1.5.2 Control hazards

One of the major problems in superscalar pipelines is making sure that the CPU knows what instructions will be executed next. In case of a program without any conditional statements and function calls, the code is just a well-defined sequence of statements. However, if the sequence of instructions is hard to determine in advance, it might be impossible to schedule the next instructions, causing pipeline delays.

Branch prediction. Branch instructions are a typical example of a control hazard. Usually, CPUs use *branch prediction* [McF93] to guess the outcome of the involved predicate, and uses *speculative execution* to follow the expected path. This prediction process comes in two variants: static and dynamic. *Static prediction* for a given branch always assumes the same output. It is relatively efficient in some special cases, for example in backward-branches that usually correspond to program loops and are taken in the majority of cases. *Branch hints* are another related technique, where a programmer or a compiler can annotate the code with the most likely branch outcome. *Dynamic prediction* is a scheme that analyzes the history of a given predicate, and uses it to guess the outcome of the next computation. Additionally, some dynamic prediction schemes not only store the expected branch result, but also the instruction that is to be executed when branch is taken, reducing the need to fetch/decode it, allowing it to enter the execution pipeline immediately. While beneficial, prediction techniques and

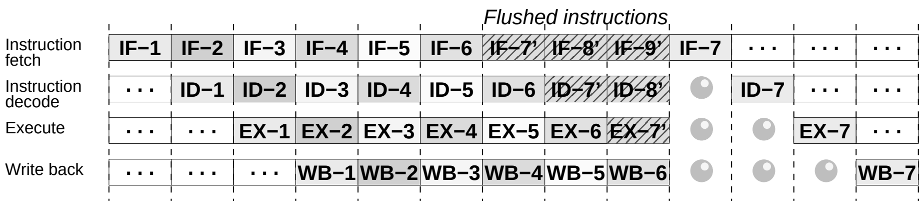


Figure 2.4: Branch misprediction: pipeline flushing and pipeline bubbles

branch hints are not perfect. When a *branch misprediction* happens, the entire pipeline (or large part of it) needs to be *flushed* (cleared), and the computation needs to start from the beginning.

For example, in the following code, computing the sum of numbers from 1 to N, the loop branch can be predicted as taken:

```
; input: a is 0, b is N; output: sum of 1..N in b
loop:
    add a, a, b   ; a = a + b
    dec b         ; decrease b, set flag 'zero' if it became zero
    bnz loop      ; branch to 'loop' if the zero flag is not set
    mov b, a      ; b = a
```

This code, when executed for $N = 2$, will result in the following sequence of instructions:

```
                  ;      a == 0, b == 2
    add a, a, b   ; I-1, a == 2, b == 2
    dec b         ; I-2, a == 2, b == 1
    bnz loop      ; I-3, a == 2, b == 1, branch correctly predicted as taken
    add a, a, b   ; I-4, a == 3, b == 1
    dec b         ; I-5, a == 3, b == 0
    bnz loop      ; I-6, a == 3, b == 0, branch incorrectly predicted as taken
    mov b, a      ; I-7, a == 3, b == 3
```

Figure 2.4 demonstrates CPU activity when executing this sequence of instructions. We see that after fetching I-3, the CPU correctly predicts the branch will be taken, and fetches the proper instruction I-4. However, after fetching I-6, the CPU assumes an incorrect code flow, and starts executing a wrong sequence of instructions I-7', I-8', I-9'. Only when I-6 is fully evaluated, the CPU detects that the taken sequence is incorrect, and it needs to flush the pipeline and start executing from the proper I-7 instruction. However, when I-7 enters the "fetch" phase, no other instructions can be in the latter phases, and pipeline bubbles occur, wasting system resources. Note that the delay between the properly predicted I-3 and I-4 is a single cycle, while the delay between I-6 and I-7 is four cycles. This demonstrates how dangerous a branch misprediction can be for the CPU efficiency.

Indirect branches. Branch prediction addresses the problem of *direct* branches, i.e. branches where the jump address is encoded in the instruction. However, there is a class of situations where the branch is *indirect*, with the jump address stored in a register or in memory. Typical examples include calling a function from a function array, or polymorphic method calls in object-oriented languages. Such cases are often handled by *branch-target-buffers* (BTB), where for a given originating address the last target address is stored. This is an efficient solution if the indirect branch target is relatively static. However, in many cases simple BTBs are not efficient enough, and more sophisticated methods are necessary [DH98].

Predication. Another technique that overcomes branch problems is *predication*. Let us analyze the following code:

```
    if (a < b)
        a++;
    else
        b++;
```

On a traditional CPU, that does not use predication, it would compile into assembly using a conditional branch instruction similar to this:

```
    cmp a,b    ; compare a and b, set the status flags accordingly
    blt lower  ; branch to 'lower' if a < b
    inc b      ; a >= b, increase b
    j   end    ; unconditional jump to 'end'
lower:
    inc a      ; a < b, increase a
end:
```

On CPUs that provide predication instructions can be annotated by a predicate that defines if a particular instruction should be executed. For example, the predicated code could look as follows:

```
    cmp   a,b  ; compare a and b, set the flags accordingly
    inclt a    ; if lower-than (LT) flag is set, increase a
    incge b    ; if greater-equal (GE) flag is set, increase b
```

In this scenario, the result of only one instruction will be used, without any conditional branching. Predication is provided by e.g. ARM and IA-64 (Itanium) architectures. It often provides a significant speed improvement, and, while single instruction codes can get longer because of extra bits needed for the predicate definition, the overall code size can get reduced due to a smaller number of instructions.

2.1.5.3 Structure hazards

Another type of hazards are the *structure hazards*, related to the computational limits of modern CPUs. For example, on many architectures it is possible to fetch/decode more than one instruction in one CPU cycle, but only one instruction per-cycle can use load-store units. In such a situation, if two memory accessing instructions are decoded at the same time, one of them needs to be delayed, due to insufficient LSU resources.

2.1.6 Deepening the pipeline

As presented so far, the execution pipeline is relatively simple, and only consists of a few stages. However, modern CPUs use a full bag of tricks that improve performance and try to limit the negative effect of the discussed hazards. As a result, the pipeline needs to be broken into more logical steps. Additionally, the steps are getting more and more complicated, and with an increasing CPU frequency, they often cannot execute in a single clock cycle. For this reason, the stages need to be broken into smaller sub-stages, further increasing the pipeline depth, and resulting in *super-pipelined* CPUs.

Since the clock frequency used to be the most distinguishable CPU feature, with higher frequencies positively influencing sales, processor companies, especially Intel, for a long time had this parameter as the focus of their CPU architecture design. An extreme example are Pentium 4 Prescott CPUs that have a pipeline of 31-stages. Such long pipelines allowed for very high CPU frequency, resulting in great performance on CPU-friendly code. On the other hand, the hazard-induced penalties in such pipelines become even higher. This increasing penalty has led to a reverse in the trend, with recent CPUs having shorter pipelines (e.g. 14 stages in Intel Core2). Still, even with such “short” pipelines the impact of pipeline bubbles is significant, stressing the importance of generating code that contains as little hazards as possible.

2.1.7 Development trends and future architectures

For decades the major focus of the CPU designers was the performance of a single CPU. Typical methods of improving this performance are increasing clock frequency, super-scalar CPUs, out-of-order execution, and larger cache sizes. However, further improvements in these already highly-sophisticated areas result in relatively small gains, significantly increasing system complexity and power consumption at the same time. Furthermore, many application areas, among them database systems, have problems with fully exploiting such complex architectures. As a result, in the last few years new trends in CPU architectures become popular.

2.1.7.1 Simultaneous multithreading

In many cases, a single executing thread has problems with full utilization of the available computational resources in modern superscalar CPUs. This is because of instruction-issue delays related to data-dependencies, memory waits (see Section 2.2.3) etc. *Simultaneous multithreading* (SMT) [TEL95] improves this situation by allowing *multiple* threads to be executing at the same time. This is achieved by having a per-thread copy of some of the CPU sections, e.g. registers, but sharing a single instance of the main execution resources between the threads. Multiple hardware threads provide more instructions every cycle, and also allow hiding delays in one thread by executing instructions from another.

SMT is a relatively cheap technique in terms of incorporating into CPUs, as the added per-thread CPU infrastructure is small. It has been implemented in some of Intel's Pentium 4 and Core i7 CPUs (as *hyper-threading* [MBH+02]), IBM Power5 [SKT+05] and Sun Microsystems UltraSparc chips [Sunb] that allow even 8 simultaneous threads in UltraSparc T2 CPUs.

2.1.7.2 Chip multiprocessors

With advances in the chip miniaturization process, more and more transistors become available on a die [Moo65]. This allows not only to create more sophisticated CPU cores, but also, to put multiple fully functional cores on a single chip. Typically, cores have both designated private memory (usually L1 cache), as well as shared cache memory (usually L2). This technology, known as *chip multiprocessors* (CMP), recently became a de-facto standard, being available both in mainstream CPUs, including Intel's Core and AMD's Athlon and Phenom chips (up to 4 cores), as well as server processors, including Sun's UltraSparc (up to 8 cores).

Quick widespread adoption of CMP CPUs puts new challenges on developers. To utilize the performance of the underlying hardware, parallel programming techniques, previously applied to a relatively limited number of applications, now need to be used in most types of programs. Furthermore, optimizing software for the new architecture becomes increasingly hard with higher degrees of parallelism. Another complication factor occurs in situations where CMP and SMT are combined in a single chip, as is the case e.g. in Sun's UltraSparc T2 CPU that can have 8 cores, each with 8-way SMT, resulting in 64 concurrently available hardware threads.

2.1.7.3 Heterogeneous computation platforms

Previously discussed developments in processor technology assumed that applications are running on top of one or more identical general purpose processors. However, in modern computers, more and more computational tasks are off-loaded to designated specialized units. A typical example are *graphics processing units* (GPUs), optimized for processing 2D and 3D graphics, and available in almost every new computer, either as dedicated graphics cards, or integrated in the motherboard chipset. Other examples include devices specialized for *digital signal processing* (DSP), video en- and de-coding, network traffic handling, physics simulation and data encryption. While in most cases these additional processors are used only for their designated task, in many cases they can be used for other applications. Again, a typical example are graphics cards that provide pure computational power often exceeding the CPU, as demonstrated with a record-breaking sort performance [GGKM06]. Thanks to this high speed, as well as the continuously improving programming flexibility of these devices (e.g. NVIDIA CUDA [NVI08]), GPUs became a very popular platform for numeric-intensive tasks.

Multiple computational units with different properties are also possible even on a single chip. For example, the STI Cell Broadband Engine [IBM07] consists of a single general-purpose processor and 8 additional cores specialized for streaming applications (e.g. multimedia). Another example is the future Intel Tera-Scale platform [HBK06, ACJ$^+$07] that envisions 10s to 100s different cores with different functionality on a single chip. Also in *systems-on-chip* [Wol04] (SoC) designs multiple functional units are combined on a single chip. For example, Sun's UltraSparc T2 processor [Sunb] combines traditional general-purpose cores with a network controller and a cryptographic unit. SoC devices are es-

Memory module	DRAM	Page mode DRAM	Fast page mode DRAM	Fast page mode DRAM	Synchronous DRAM	Double data rate SDRAM	DDR2 SDRAM
Year	1980	1983	1986	1993	1997	2000	2006
Module width (bits)	16	16	32	64	64	64	64
Mbits per DRAM chip	0.06	0.25	1	16	64	256	1024
Bandwidth (MBit/sec)	13	40	160	267	640	1600	8533
Latency (ns)	225	170	125	75	62	52	36

Table 2.2: Milestones in the DRAM development (adapted from [Pat04], with the 2006 milestone added)

pecially popular in embedded environments, and include e.g. Intel IXP series, Philips Nexperia and Texas Instrument OMAP chips.

The heterogeneous nature of the discussed platforms brings additional challenges for software developers. For optimal performance, applications need to be designed to exploit the available hardware, e.g. by performing a particular task using computational units best suited for it. However, with increasing heterogeneity of the computers, optimizing an application for every single configuration is not economically feasible. This calls for applications that can dynamically *adapt* to the available computing resources. An example of such approach is the OpenCL framework [Khr09], where CPUs, GPUs and other computing devices can be transparently used by the application.

2.2 Memory system

So far in our discussion, we have focused on the internal details of the CPU execution pipeline, assuming both data and code come from an abstract external "storage". For a long time this "storage" was just main-memory, typically consisting of DRAM chips. Table 2.2 shows the major evolution steps of DRAM over last decades, and the trends are visualized in Figure 2.5. Comparing to Figure 2.1 we see that over time the memory latency improves significantly more slowly than the CPU frequency. This means that a modern CPU, when performing a memory-access instruction, needs to wait a significant amount of

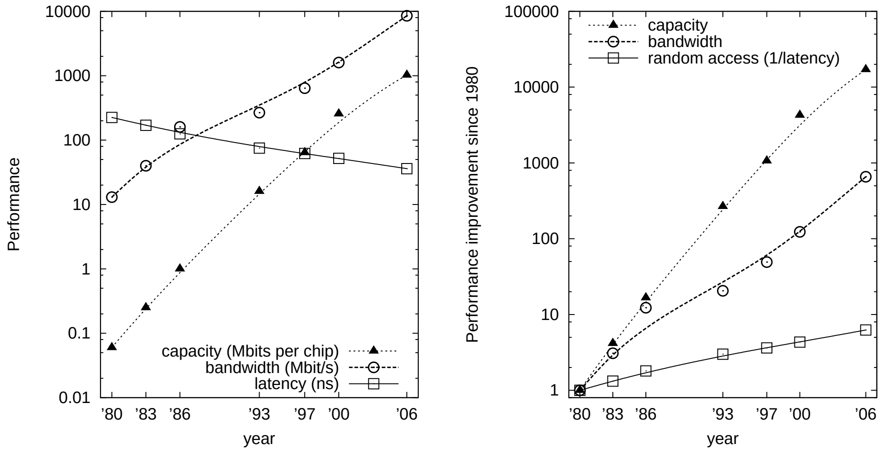


Figure 2.5: Memory chips characteristics evolution

time before the data is actually delivered from memory. This imbalance is actually significantly higher than the raw numbers in Tables 2.1 and 2.2 suggest, as the actual cost of the memory-access stage is only a fraction of the CPU latency, since other pipeline stages are included in this number. In reality, to satisfy the CPU data needs, the memory should deliver the data with the latency of only a few CPU cycles. Since commonly used *dynamic-RAM* (DRAM) memory chips cannot provide such performance, accessing them directly is very expensive.

2.2.1 Hierarchical memory system

To overcome the problem of expensive main memory access, a simple main-memory + CPU architecture has been extended with *cache memories* – small, but fast specialized memories, designed to keep the most recently accessed data, typically built with *static-RAM* (SRAM) chips [Smi82]. Cache memories hold both the process data as well as program instructions – this leads to distinguishing between *D-cache* and *I-cache*, respectively. With proper application design, most memory accesses can use this fast memory, minimizing the main-memory latency overhead. Over time, with further advancements of chip manufacturing techniques, multiple cache-memory levels have been introduced, resulting in a *hierarchical memory system*. An example of such a system is presented in Figure 2.6. Typically, a modern computer memory hierarchy is a combination of the following levels, ordered by the increasing latency[1]:

- registers – CPU registers can be seen as the closest storage for the CPU, and often the only storage that CPU can perform computation on. Typically, there are 4-256 registers and accessing them takes 1-3 cycles.
- L1 cache – small (ca. 16-128KB) and fast (2-10 cycles) memory, on-chip, typically divided into I-cache and D-cache
- L2 cache – larger (ca. 256-8192KB) but slower (10-30 cycles) memory, usually on-chip, typically shared by instructions and data
- L3 cache – relatively large (1MB+) but slow cache, either on-chip or on a motherboard. Only on some platforms.
- main memory – large (gigabytes) but relatively slow (50-300 cycles) storage.
- solid-state disk - large (tens or hundreds of gigabytes) but moderately slow (tens to hundreds of thousands of cycles).
- magnetic disk – very large (hundreds of gigabytes or terabytes) but very slow (millions of cycles) storage.

2.2.2 Cache memory organization

Cache memory is typically divided into a set of fixed-size *cache lines*, usually ranging from 16 to 128 bytes. To simplify cache management, each cache line holds data from an area in main memory aligned to the cache line size. Additionally, when data is transferred into cache, typically the entire cache line is filled. This means that even when asking for a single byte, the memory subsystem will deliver e.g. 64 bytes, and that amount of cache will be used. As a result, small data requests lead to poor cache utilization, promoting the use of large data transfers.

Cache lines are usually organized as two-dimensional arrays, where one dimension is the *set* and the other is the *set associativity* [PHS99]. For a given memory address, its *set id* is typically determined by a function on the address bits. Within a set, the *line id* is determined by matching the reference address

[1]Disk storage is accessible as a "normal" memory through virtual memory facilities (see Section 2.2.4)

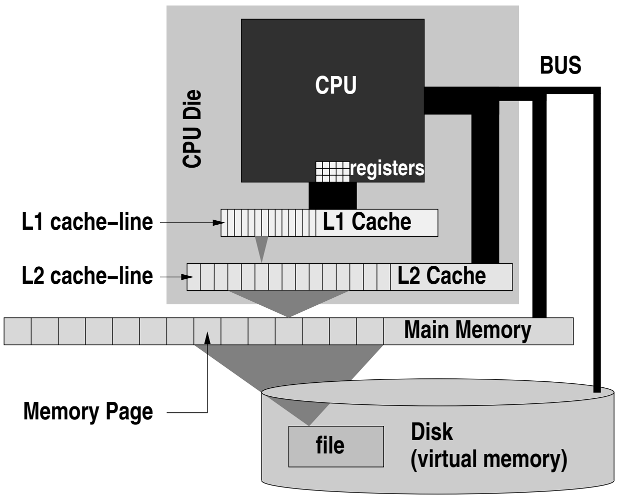


Figure 2.6: Hierarchical memory structure: registers, caches, main memory, virtual memory on disk (from [Bon02])

with the address tags of the stored data. If there is only a single set (all addresses map onto the same set id), the cache is referred to *fully associative*. On the other extreme, caches with a set associativity of 1 are called *directly mapped*. Typically, the associativity of the caches is small, in range of 1..16, since an increased number of potential lines for a referenced address can negatively influence the cache access time. Note that in a hierarchical memory system caches at different levels may vary in size, cache-line size, associativity, etc.

2.2.3 Cache memory operation and control

When CPU refers to a particular memory location, the set id for the referenced address is computed and the cache lines in this set are checked. If one of the address tags matches the requested address, a *cache hit* occurs, and the cache line can be delivered to the CPU immediately. Typically, this line is also marked as referenced, to influence the replacement policy. If the address is not in any of the locations, a *cache miss* occurs. In this situation, one of the lines is evicted, using some fast replacement policy (e.g. *LRU*), and a request to fetch the needed memory area is sent to the memory controller (or a higher cache level). Once that request is completed, the cached line is sent to the CPU, and processing can continue.

This simple behavior is intuitive, but there are situations where extra cache functionality is beneficial. A typical example is sequential memory access: in this situation, modern CPUs can predict that after fetching a particular cache line, soon the next line will be needed, and prefetch it. For example, the Intel Core architecture provides two components, DCU and DPL [Int07a], responsible for prefetching data into L1 and L2 cache levels, respectively. This allows overlapping the memory access latency with the current CPU activity. Similarly, if an application has the up-front knowledge about which memory locations will be referenced next, it can use *software prefetching* instructions [Int07b, Adv05] to issue requests for these locations. Since prefetching can lead to increased CPU activity and memory traffic, if not used properly, it can have an adversary effect, especially in case of more expensive and error-prone software prefetching.

Special instructions are also available for other cache-controlling tasks. In situations when some computed data is known not to be needed for now, it can be saved directly into main-memory, without polluting the cache. Similarly, if some previously read memory is known to be of no use, it can be flushed from the cache to reduce evictions of other, more useful data.

Since cache memories serve as a view of a subset of main memory, it is crucial that the changes done to main memory are reflected in the cache content. This is especially important in multi-CPU and multi-core systems, where different processing units have private caches. In such architectures, special *cache-coherence protocols* are being used [Ste90][HP07, Section 4.2] to guarantee data consistency. The overhead of these protocols can be significant, therefore it is important to design parallel algorithms such that the need of applying these consistency mechanisms is minimized.

2.2.4 Virtual memory

Another important aspect of the memory infrastructure is the difference between the *physical* and the *virtual* memory. In early computer generations, the entire computer memory was directly accessible to the program. In modern computers, however, the address space of an application is explicitly managed by the operating system. Typically it does not constitute a single contiguous area, but instead, it consists of a set of *pages*, where each virtual page refers to some physical memory area. Page sizes are typically in range of few kilobytes (4KB on *x86* platforms), but some systems allow multiple different page sizes, reaching 256MB on Intel Itanium architecture [Int06]. Having virtual pages allows multiple improvements to a vanilla memory system, including enhanced security, sharing memory between applications, on-demand physical memory allocation, copy-on-write memory areas, memory-mapped files and more.

One performance problem that the virtual memory systems introduce is the need of translating a virtual address as seen by an application into a physical memory area. Typically, an operating system stores a *page table* that provides this translation. Since accessing this table for each address translation can be expensive, CPUs typically use a highly specialized cache, called *translation lookaside buffer* (TLB), that stores translations for the recently accessed pages. Like cache memory, it can also be hierarchical (L1, L2), and independent for instructions and data (I-TLB, D-TLB).

The TLB capacity can be small, e.g. L1 D-TLB on Athlon 64 can keep track of 32 recently accessed 4KB pages [Adv05]). Since *TLB-misses* can be as expensive as cache-misses, the applications need to take special care of avoiding them. Typically, sequential access patterns do not incur these problems, but random memory accesses can easily result in frequent misses. One of the methods of reducing the TLB misses is to explicitly use larger pages, available in some operating systems. This allows to have fast translation for larger memory area (e.g. 8 entries for 2MB pages on Athlon 64 [Adv05]), but may result in an increased memory fragmentation.

2.2.5 Future trends

Continuous increase in the cache sizes allows larger datasets to be quickly accessed, but also results in an increased access latency. This is especially visible in L2 caches, where the latency increased by a factor of 3 in the last decade [HPJ$^+$07]. As a result, further increases in the L2 sizes can have a detrimental effect for applications with a working set already fitting in the cache. To overcome this problem, manufacturers limit the L2 cache sizes, and introduce additional L3 cache levels. While this solution used to be applied mostly in the server market, in CPUs like Intel Itanium 2 (6MB on-chip L3) and IBM Power6 (32MB off-chip L3), AMD Phenom and Core i7 chips make this solution mainstream, with 2MB and 8MB on-chip L3 cache, respectively.

The evolution of the cache memory hierarchy is also heavily influenced by multi-core CPUs, since typically parts of the hierarchy are private to each core, while other parts are shared. For example, Intel Core2 CPU uses private L1 caches and shared L2, while AMD Phenom uses private L1 and L2 caches and shared L3. This variety of configurations makes it continuously more challenging to provide solutions optimally exploiting all of them.

RPM	3600	5400	7200	10000	15000	15000
Product	CDC Wrenl 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	Seagate ST3450856
Year	1983	1990	1994	1998	2003	2008
Capacity (GB)	0.03	1.4	4.3	9.1	73.4	450
Bandwidth (MB/sec)	0.6	4	9	24	86	166
Latency (msec)	48.3	17.1	12.7	8.8	5.7	5.4

Table 2.3: Milestones in the hard-drive technology development (from [Pat04], with the 2008 milestone added)

With the increasing popularity of multi-core chips, and hence parallel programs, the synchronization techniques between processes become increasingly important. Traditional locking mechanisms, while known for decades, are often hard to use, expensive and error-prone. As a result, recently, hardware mechanisms for *lock-free* operations have been proposed. *Transactional memory* [HM93] introduces a mechanism that allows a set of memory operations to execute atomically, with an explicit commit at the end of a code block. Since this commit can succeed or fail, the software needs to check for the result and adapt to it. Another solution that does not require any software modifications has been proposed with *transactional lock removal* [RG02]. Here, the hardware can identify the transactions looking at the locks acquired by the program, speculatively execute them without acquiring a lock, and apply conflict resolution schemes in case of conflicts. Both proposals reduce the need of locking and hence can significantly improve performance.

2.3 Hard-disk storage

The previous two sections discussed the major features of modern CPUs and memory subsystems, crucial for high-performance in-memory data processing. Since this thesis focuses on large-scale data sets, we now focus on the characteristics of typical storage systems.

The most popular medium for large-volume data storage are magnetic disks, with an example disk presented on the left-most side of Figure 2.7. In this solution, data is stored on *platters* with a magnetic surface and accessed with a moving *head*. Each platter is divided into *tracks*, and each track consists of *sectors*. Platters are attached to rotating *spindles* that perform thousands of

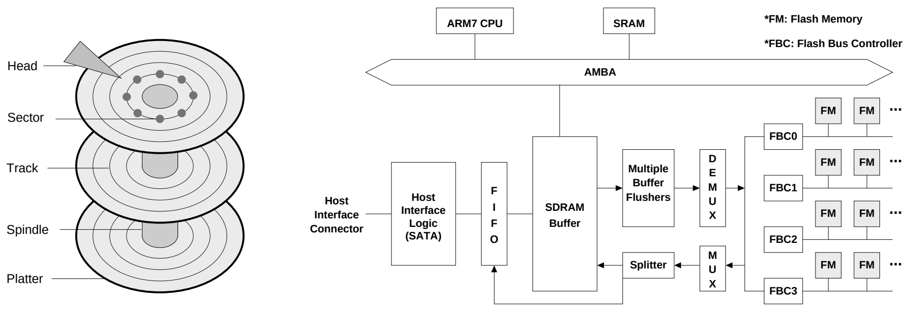


Figure 2.7: Hard-drive diagram (left) and an example solid-state disk diagram (Mtron SLC, right)

rotations per minute. To read a particular data unit, the head needs to perform a *seek* to a proper track, and wait for the platter to rotate to a proper position to read a given sector. As a result, disk latency can be computed as a sum of the seek time, rotational delay and the transfer time.

Table 2.3 presents the major development milestones for the hard-drive technology over last 25 years. The performance trends, visualized in Figure 2.8, are similar to those of memory chips, presented in Figure 2.5: while the capacity grows at a very rapid pace, the bandwidth, while steadily improving, lags behind it, and the latency improves very slowly. In terms of bandwidth, disks are typically in order of 100 times slower than memory, and in terms of latency, this difference is in order of $10,000$ to $100,000$. As a result, efficient data delivery from disk is a significantly harder problem than memory access.

The other factor that distinguishes hard drive performance from main memory is the difference between the cost of random access and sequential bandwidth. Reading a single byte from memory takes in range of 100ns if random access is used and a fraction of a nanosecond if sequential access is used,[2] a 100 to 1000 times difference. On disk, reading a single byte takes ca. 5ms with random access, and ca. 20ns with sequential access, resulting in a difference factor of $100,000$ to $1,000,000$. As a result, on disk it is even more important to use sequential access methods.

[2]assuming multi-gigabyte sequential RAM bandwidth possible with prefetching mechanisms

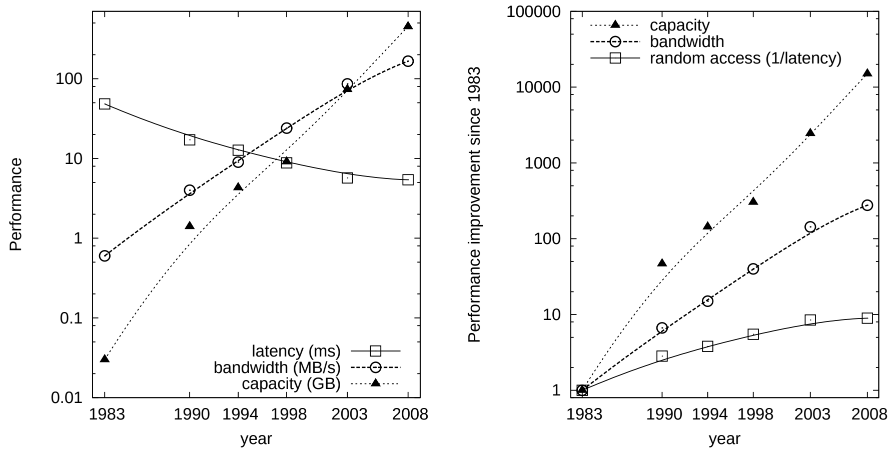


Figure 2.8: Hard-drive characteristics evolution

2.3.1 Disk performance improvements

One of the methods of improving access to disk is caching the most recently accessed disk areas in the available main memory. This plays a similar role as cache-memories in hierarchical memory systems. Caching can be also performed on the disk itself – modern disks can have cache memories of several megabytes.

Another improvement, performed in both operating systems as well as in the disk controller, is request scheduling. In many cases, especially with random-access-oriented applications, it is common to have multiple outstanding read or write requests at the same time. These pending accesses can be re-ordered to match the movement of a disk arm and platter rotation, reducing the average request latency [TP72]. It also allows for prefetching data from disk, similarly as in memory prefetching.

Relatively high sequential disk bandwidth is only possible if the high cost of moving the disk head and waiting for the platter rotation is paid once for a large unit of data. This is especially important in scan-intensive applications, where multiple processes perform sequential data access. In such cases, it is important to use large, isolated I/O operations, to amortize the random seek cost. This is presented in Figure 2.9 with an experiment that measures the bandwidth of the disk system by issuing a sequence of sequential or random disk accesses, with a varying I/O unit size, and not using the caching and prefetching facilities

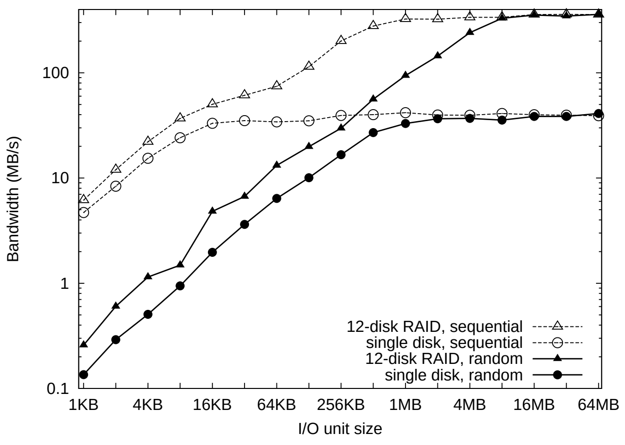


Figure 2.9: Disk read bandwidth depending on the I/O unit size, using sequential and random access patterns

provided by the operating system. This experiment shows that to get a good bandwidth with random accesses, access granularity needs to be in range of a few megabytes. Note that databases and operating systems typically use much smaller disk pages (4-64KB).

2.3.2 RAID systems

To improve the performance of the storage layer, it is common to use multiple disks, typically in some form of a RAID system [PGK88]. While many different RAID configurations are possible, they typically exploit three basic concepts: *mirroring*, which stores the same data on multiple drives; *striping*, which partitions data across different drives; and *error correction* (or *fault tolerance*), which allows detecting (e.g. using CRC [PB61]) and possibly fixing (e.g. using error-correcting codes [RS60]) problems related to data corruption and hardware failures. These three concepts are used to build various *RAID levels* (e.g. *RAID-0* or *RAID-6*), including *nested* RAID configurations (e.g. *RAID 0+1*). Depending on the used configuration, a RAID system can improve the storage layer in areas of capacity, performance and reliability.

While RAID levels can significantly improve performance for both random and sequential access scenarios, using them often requires careful tuning, and can have some detrimental effects. Typically, the data is spread between disk not on a single-byte basis, but using larger blocks, e.g. 64KB in size. As a result, using I/O units of sizes smaller than the block size, involves just a single disk, and only with larger I/O units multiple disks are used, resulting in an improved bandwidth, as seen in Figure 2.9 for RAID systems. This figure also shows that the I/O size at which the random-IO performance approaches that of the sequential access is larger for the RAID systems. This is caused by the fact that in RAID systems multiple disks are used to serve an I/O request, resulting in a per-disk bandwidth being only a fraction of the original size, reducing the benefit of large I/Os. In such situations, to improve the sequential performance, the I/O unit size needs to be scaled proportionally to the number of used disks, quickly reaching tens of megabytes, as shown in Figure 2.9. With such transfer sizes, the memory consumption of the I/O layer can be very high, especially when multiple large requests are served at the same time.

2.3.3 Flash storage

The most visible alternative to magnetic disks are *NAND flash memories* [MA95, GT05], currently the storage medium of choice in small, portable computers and multimedia devices. In this solution, multiple flash chips are combined into a single device, typically visible to the system as a regular drive, as presented in the right-most side of Figure 2.7. Current generations of flash drives excel over traditional disks in sequential access, random-read performance [LM07, SHWG08], power consumption and failure rates, as demonstrated in Table 2.4. Flash memories are less attractive in the price/capacity dimension, as their per-byte price is significantly higher. To overcome this difference, flash memories are also applied in *hybrid drives* – traditional magnetic drives with an integrated flash-drive, used to store most frequently accessed data.

One particularly interesting aspect of NAND devices is that in any given area on the flash memory all the bits are by default set to 1. Clearing a bit to 0 is a relatively fast process. However, to set a 1 bit again, the entire area needs to be *erased* to its previous state, making this process expensive. To optimize performance for this behavior, a set of algorithms similar to those used previously for write-only storage has been proposed (see Section 3.5.7). However, currently available flash drive interfaces typically imitate standard disks, not exposing the low-level functionality of setting particular bits and explicit erasing. As a result, any write to a flash device typically causes an erase operation, resulting in a significant difference between the random read and random write performance, as presented in Table 2.4.

	NATA Disk	USB Flash	IDE Flash	FC Flash
GB	500	4	32	146
$/GB	0.20	5.00	15.62	-
Watts (W)	13	0.5	0.5	8.4
seq. read (MB/s)	60	26	28	92
seq. write (MB/s)	55	20	24	108
rnd. read (IO/s)	120	1,500	2,500	54,000
rnd. write (IO/s)	120	40	20	15,000
IO/s/$	1.2	75	5	-
IO/s/W	9.2	3,000	5,000	6,430

Table 2.4: Disk and Flash characteristics (from [SHWG08]))

2.3.4 Future trends

Current developments in magnetic hard-drive technology follow the trends observed over the last decades: capacity and bandwidth increase at a rapid pace, while latency improves very little. This tendency will most likely continue, since the first two parameters are related to density of data on disk platters, while the third one depends on mechanical factors – head seek time and platter rotation speed – which are much harder to improve. In this situation, flash-based devices, with their rapidly improving performance parameters and, at the same time, rapidly decreasing prices, are quickly becoming a feasible storage solution for a large class of applications.

Another possible direction are micro-electro-mechanical store (MEMS) devices [Sch04]. In these devices, data is organized on a rectangular surface, and accessed by thousands of heads. Comparing to traditional magnetic disks, these devices provide a few-times improvement in terms of sequential access and ca. 10 times improvement in random-access [Ail05]. Additionally, the large available number of heads allows for performance improvements impossible for standard disks [SSAG03]. First, heads not used by the priority tasks can be exploited to provide data for background processes. Secondly, fine-grained data organization allows both row-order and column-order data access for two-dimensional data structures.

2.4 Conclusions

This chapter provided the overview of the most important aspects of modern hardware, concentrating on three areas: processor architecture, memory system and disk technology. Advances in the processor architecture resulted in highly efficient chips, but programs need to be carefully designed to fully exploit the new features (multiple execution units, SIMD) that deliver this performance. On the memory level, a hierarchy of CPU caches forces developers to re-design their in-memory data storage strategies. Finally, on disk level, the increasing imbalance between latency and bandwidth requires applications to operate with large I/O units, with sequential scans becoming a preferable access method. These features of modern hardware have direct impact on the research described in the remainder of this thesis.

Chapter 3

Databases on modern hardware

Database management systems (DBMSs) provide application developers with a high-level abstraction of data management tasks. They expose generic interfaces that allow accessing and manipulating data, while at the same time providing features like concurrency control, transaction management, failure recovery and consistency checks. Additionally, they hide the hardware details of a used machine, helping applications to run on various platforms.

This chapter discusses the major characteristics of a DBMS, focusing on elements crucial for this thesis. In Section 3.1, we briefly describe the relational data model and relational algebra that are the fundamentals of most existing database engines. Section 3.2 describes the typical architecture of a DBMS, shortly discussing the most important components. Two of these components – the query executor and the storage manager – are of most interest for this thesis, and in this chapter we present different approaches of implementing them, both significantly influencing the research presented in the following chapters of this thesis. First, Section 3.3 discusses the most commonly used implementation approach, based on an *iterator execution model* working on top of an *N-ary storage model.* Then, Section 3.4 presents a completely different approach, found in the already existing MonetDB system, which is using a *fully-materialized algebra* based on the *decomposed (column) storage model.* Both architectures, while having benefits in some areas over the other one, do not make a full use of the possibilities of modern computer hardware. To improve this situation, multiple optimization techniques have been proposed, as presented in Section 3.5

3.1 Relational model

Since late 1970s, the relational model is the most popular model in database systems. In this model, the data is stored as a set of *N-ary relations*, where each relation is a subset of a Cartesian product of *N domains*. A relation consists of a set of *tuples*, each containing *N attribute values*, one for each *attribute*. Typically, relations are visually represented as *tables*, where tuples are *rows* and attributes are *columns*, as presented in the left-most side of Figure 3.1. Still, the model itself does not impose any particular physical data representation. In particular, the relations by definition are *unordered*.

The operations on relations are defined in *relational algebra*, consisting of a number of *operators*. The basic operators include *projection* (π), *selection* (σ), *aggregation* ($\mathcal{G}$), *Cartesian product* ($\times$) and various types of *join* ($\bowtie$). For example, using relation *People* from Figure 3.1, to compute the age-bonus for all people older than 30, one could use the following relational query:

$$\pi_{Id,Name,Age,Bonus=(Age-30)*50}\ (\sigma_{Age>30}\ (People)) \tag{3.1}$$

Similarly as its underlying model, the algebra does not discuss how particular operations should be performed, but only what is the outcome of a given operator.

While the relational model is the most popular approach in the database world, other solutions exist. For example, object-oriented [Bar96], hierarchical [Bla98] or semi-structured [BGvK$^+$06] databases are all being used in specialized data management tasks. In this thesis we focus on the relational databases and query processing in these systems, but some of the techniques can be applied within other paradigms.

3.1.1 Relational model implementation

When proposed, the relational model was an abstract mathematical concept, without an existing physical implementation. In the second part of 1970s and early 1980s, real-world realizations of this idea have been implemented, e.g. System R [CAB$^+$81] and Ingres [SHWK76]. These systems introduced multiple concepts and designs that often can still be found in the existing relational databases.

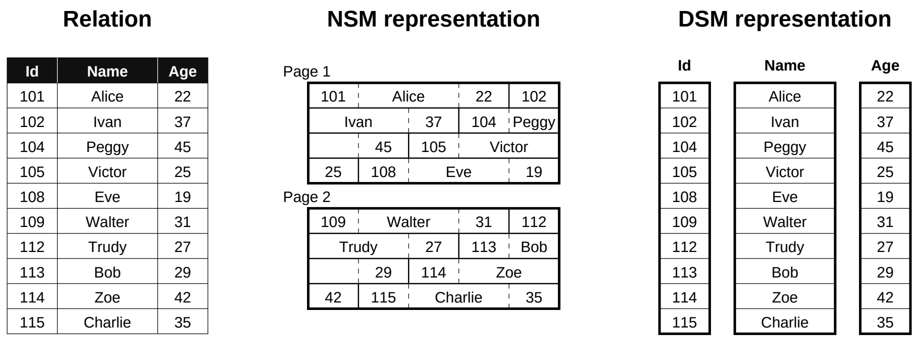


Figure 3.1: Relational table and its representation in the N-ary storage model (NSM) and the decomposed storage model (DSM)

3.1.1.1 Physical relation representation

The most common data representation in relational databases is to keep a relation as a collection of *rows*, each corresponding to a tuple. These rows are typically stored as *records* one after another in one *table* per relation, consisting of disk pages, each storing multiple records. This representation, known as the *N-ary storage model* (NSM), is presented in the central part of Figure 3.1. An alternative representation is the *decomposed storage model* (DSM [CK85]) presented in the right-most part of Figure 3.1. Here, every attribute is stored as a separate area on disk.

3.1.1.2 Query execution plans

To provide the functionality of the relational algebra in a physical world, databases commonly include a set of *physical* operators, roughly corresponding with their logical counterparts. Typically, it is not a one-to-one mapping, as the same logical operator can be implemented in various ways. For example, a logical *join* operator can be executed with a *merge-join* or a *hash-join*, depending on data properties, available resources etc. Various operators are combined into a *query execution plan* – a physical representation of a user query. A good overview of the implementation techniques for the physical query plans is presented in [Gra93].

Within a query plan, typically two execution methods are used [SKS02, Chapter 13.7]: pipelining and materialization. These two approaches are discussed in more detail in Sections 3.3 and 3.4.

```
SELECT   l_returnflag,
         l_linestatus,
         sum(l_quantity) AS sum_qty,
         sum(l_extendedprice) AS sum_base_price,
         sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
         sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
         avg(l_quantity) AS avg_qty,
         avg(l_extendedprice) AS avg_price,
         avg(l_discount) AS avg_disc,
         count(*) AS count_order
FROM     lineitem
WHERE    l_shipdate <= date '1998-09-02'
GROUP BY l_returnflag,
         l_linestatus
ORDER BY l_returnflag,
         l_linestatus;
```

Figure 3.2: TPC-H Query 1

3.1.1.3 Query language

DBMSs typically hide the *imperative* nature of the relational algebra by providing some high-level language that is converted into the actual query plan. The de-facto standard for the relational databases is *Structured Query Language* (SQL) [CB74, EKM$^+$04] that expresses the queries in a declarative syntax close to the natural English language. For example, the relational query from Section 3.1 can be expressed with this SQL statement:

```
SELECT Id, Name, Age, (Age - 30) * 50 AS Bonus
FROM   People
WHERE  Age > 30
```

Figure 3.2 presents a more complicated SQL example: Query 1 from the TPC-H benchmark that is often used as an example throughout this thesis.

3.2 DBMS architecture

The components of a typical relational DBMS are presented in Figure 3.3. In fact, the architectures of state-of-the-art DBMS products are significantly more complex and often include dozens of cooperating modules. Still, generally, database consists of the following components:

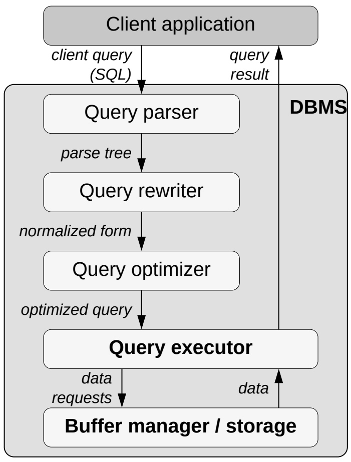


Figure 3.3: A simplified architecture of a DBMS

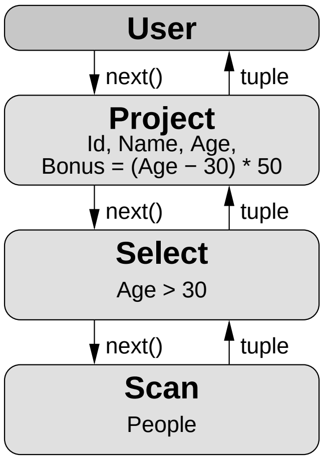


Figure 3.4: An operator tree for a simple SQL query in a tuple-at-a-time execution model

client application – before a query enters a DBMS, it needs to be provided by a client. A query is typically expressed in a high-level query language, e.g. SQL. A client can connect to a DBMS directly, using some DBMS-specific low-level communication protocols, or by exploiting a general-purpose high-level connection infrastructure, such as ODBC [Mic] or JDBC [Suna]. Furthermore, additional components can be used between the actual application and the DBMS, for example specialized utilities for load balancing or query result caching.

query parser – the syntax of the client query is analyzed, and a *parse tree* is built, providing internal representation of a query.

query rewriter – this component checks the parse tree for its semantic correctness (e.g. existence of used table names or proper access rights) and converts it into some *normalized form.* It is typically a tree of logical operations, often close to the relational algebra. This module may perform some additional tasks, for example expansion of user-defined views into the underlying queries.

query optimizer – the major task of this component is to rearrange the query tree in such a way that the expected execution time of the result query is minimal. It also prepares the physical query plan, with the logical operations (e.g. a generic join) replaced with their physical counterparts (e.g. a hash-join). This module is usually highly complex, and has a tremendous impact on the total query execution time. For example, a wrong order of operations or a bad choice of an operator can result in a computational blow-up at some stage of processing. For such a bad plan, even the fastest query executor cannot process a given plan in satisfying time.

query executor – is the core component of query processing. It accepts a physical query plan, and performs all specified processing steps on data that it receives from the storage layer. The computed results are returned to the client.

buffer manager / storage – takes care of storing data on persistent media, accessing it and buffering it in memory. Typically, it also takes care of handling updates, managing transactions, performing disaster recovery, logging, locking, and more. However, these issues are not the focus of this thesis, and we only concentrate on data storage and access.

Of these components, two are of most importance for this thesis: query executor and storage layer. In the next two sections we discuss two approaches of implementing the query execution layer, based on two different principles: pipelining and materializing [SKS02, Chapter 13.7]. First, in Section 3.3 we analyze a typical query processor using a pipelined iterator interface and working on top of the N-ary tuple storage. Then, in Section 3.4 we discuss the architecture of MonetDB, concentrating on its fully-materialized in-memory query execution model and the use of column-based storage.

3.3 Tuple-at-a-time iterator model

Most database engines internally use the *iterator model* for their query execution layers [Gra94]. In this model, a query plan consists of a set of relational operators, connected in some topology. Typically, it is a tree, but operators can also compose a direct acyclic graph (e.g. in parallel execution plans) or even a graph with cycles [Waa02]. Operators communicate in a "pipeline" manner following the interface based on three major functions: *open()* initializes the operator and its children, *next()* makes operator return the next part of data to the caller, and finally *close()* finishes processing and frees the resources. Typically, in the next() call, a single tuple is returned, using the NSM-based records. This "pull-based" model is known as *tuple-at-a-time* iterator model.

Figure 3.4 presents an example of an operator tree for the query from Section 3.1. The query execution proceeds as follows. First, the user (client application etc.) asks the top operator (Project) for the next result tuple. Project asks its child (Select) and it in turns asks its child (Scan). Scan retrieves the next tuple from the table, and sends it back to Select. Select checks if the tuple passes its predicate and, if so, sends it back to Project. If not, it asks Scan for the next tuple. Project, for each input tuple, computes an additional column and returns a new tuple to the user. When Scan determines that there are no more tuples in the underlying relation, it sends the end-of-stream identifier throughout the pipeline, which finishes the processing.

To better demonstrate what is happening within an operator, this is a pseudocode for the next() function in the Select operator:

```
Tuple Select::next() {
    while (true) {
        Tuple candidate = child->next();
        if (candidate == EndOfStream)
            return EndOfStream;
        if (condition->check(candidate))
            return candidate;
    }
}
```

3.3.1 Tuple-at-a-time model performance characteristics

The tuple-at-a-time approach is elegant, simple to understand, and relatively easy to implement. Performance-wise the most important characteristics of this model is that for every tuple there are multiple function calls performed. In our example, these include at least multiple next() calls, and calls to evaluate the condition in Select as well as to compute a new attribute value in Project. As a result, the state of each operator, as well as the code used by it, are accessed frequently. These properties result in a set of important performance drawbacks in a number of areas:

CPU instruction cache – if the query plan consists of many different types of operators, their combined instruction-memory footprint can be too large for the CPU I-cache to hold. Since the CPU changes its context between operators every tuple, if the I-cache is not large enough, cache-misses might occur every time a given part of code is accessed.

plan-data cache – each instance of a relational operator consumes some memory to keep its state, necessary for executing the *next()* call. With complex plans (even consisting of very few types of operators), or operators with large state, this data might not fit in the CPU D-cache, resulting in cache-misses.

function call overhead – the communication between operators, as well as many data operations are performed by calling appropriate functions or object methods. Per each operator iteration, multiple such calls are performed. Since a cost of performing a function call, in particular to a dynamically-dispatched (e.g. data-dependent) function, can be in range of tens of CPU cycles, especially when multiple parameters are passed, this overhead can be significant.

tuple manipulation – since tuples are organized as records of attributes, getting a particular value often requires extra steps to determine its position in the record. This record navigation is often repeated for each tuple.

superscalar CPUs utilization – as discussed in Section 2.1.4, modern CPUs have multiple execution units that allow performing multiple operations at the same time. Database engines, performing the same operations for a large number of tuples, seem naturally suited to exploit this feature. Unfortunately, with the tuple-at-a-time approach in each function call only a single operation on a single tuple is performed, not exposing enough work to keep multiple execution units busy. Also, heavy branching and multiple function cause frequent stalls in the pipeline. As a result, typical database code achieves very low instructions-per-cycle (IPC) performance [ADHW99].

compiler optimizations – many compile-time optimizations are impossible with the interpreted tuple-at-a-time approach. For example, due to the dynamic method dispatching, function inlining cannot be applied. Also, processing a single value at a time does not allow application of many performance-critical loop optimizations including loop unrolling, loop pipelining, strength-reduction and automatic SIMDization.

data volume – the commonly used N-ary tuple representation requires all table attributes to be stored in memory and transferred from disk. This might result in a waste of both memory and disk bandwidth, as well as the CPU cache, if a query does not use all attributes. Additionally, records

cumm. time (sec)	**excl. time (sec)**	**calls**	**avg. instr. / call**	avg. IPC	function name
11.9	11.9	846M	6	0.64	ut_fold_ulint_pair
20.4	8.5	0.15M	27K	0.71	ut_fold_binary
26.2	5.8	77M	37	0.85	memcpy
29.3	**3.1**	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
32.3	3.0	6M	247	0.83	row_search_for_mysql
35.2	**2.9**	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
37.8	2.6	108M	11	0.60	rec_get_bit_field_1
40.3	2.5	6M	213	0.61	row_sel_store_mysql_rec
42.7	2.4	48M	25	0.52	rec_get_nth_field
45.1	2.4	60	19M	0.69	ha_print_info
47.5	2.4	5.9M	195	1.08	end_update
49.6	2.1	11M	89	0.98	field_conv
51.6	2.0	5.9M	16	0.77	Field_float::val_real
53.4	1.8	5.9M	14	1.07	Item_field::val
54.9	1.5	42M	17	0.51	row_sel_field_store_in_mysql..
56.3	1.4	36M	18	0.76	buf_frame_align
57.6	**1.3**	**17M**	**38**	**0.80**	**Item_func_mul::val**
59.0	1.4	25M	25	0.62	pthread_mutex_unlock
60.2	1.2	206M	2	0.75	hash_get_nth_cell
61.4	1.2	25M	21	0.65	mutex_test_and_set
62.4	1.0	102M	4	0.62	rec_get_1byte_offs_flag
63.4	1.0	53M	9	0.58	rec_1_get_field_start_offs
64.3	0.9	42M	11	0.65	rec_get_nth_field_extern_bit
65.3	**1.0**	**11M**	**38**	**0.80**	**Item_func_minus::val**
65.8	**0.5**	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

Table 3.1: MySQL gprof trace of TPC-H Q1: `+,-,*,SUM,AVG` takes <10%, low IPC of 0.7 (from [BZN05])

representing tuples typically include some meta-data, leading to suboptimal disk usage.

The above properties of the iterator model lead to two major inefficiencies in the traditional database performance. We demonstrate them with an experiment in which TPC-H Query 1 is executed on MySQL. This query scans a single relation consisting of a large number of tuples, performs some simple computations, and finally generates a few aggregate values. The query plan is very simple, and does not include any sophisticated operators such as joins or disk-spilling aggregations. In this situation, one could expect that most of the processing time is spent in data-manipulating functions. Table 3.1, presenting a detailed profiling of the benchmark, shows otherwise. Functions performing the actual operations on data (in bold) consume less than 10% of total time. This demonstrates the first inefficiency – there is a lot of instructions related to query interpretation and tuple manipulation, causing a high *instructions-per-tuple* ratio. Additionally, the *instructions-per-cycle* factor is significantly lower than achievable on super-scalar processors. This is caused by the inability of the tuple-at-a-time algorithms to exploit multiple processing units, SIMD instructions and many of the crucial compiler optimizations. These two inefficiencies combined result in a very high *cycles-per-tuple* ratio which, even for simple operations, can reach hundreds or thousands of CPU cycles.

The tuple-at-a-time model also brings challenges in the areas of program profiling and optimization. This is caused by the fact that most of the CPU time is spread over a relatively large volume of code, including data processing functions, operator methods, tuple navigation etc. In this situation, it is hard to identify performance bottlenecks and hence introduce significant performance optimizations.

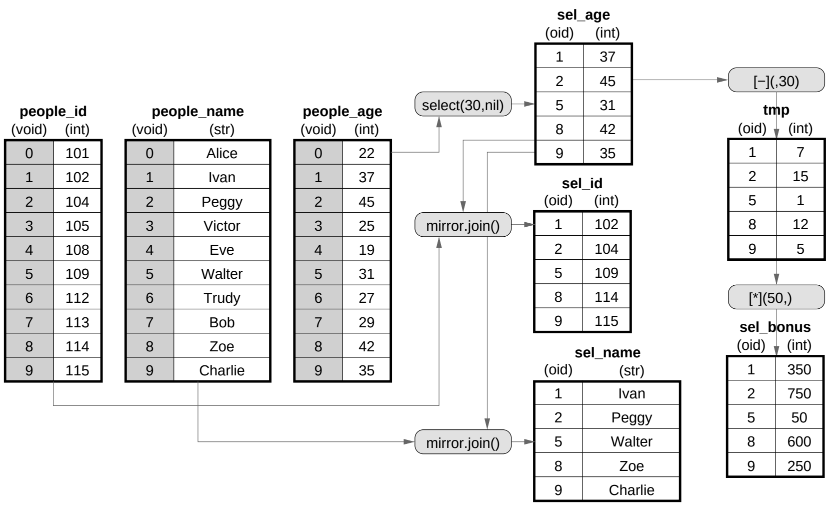
While the pipelined model often suffers in raw processing performance, it has a major benefit over the materializing approach discussed in the following section – scalability. Since in each next() call only a single tuple is passed, as long as there are no large intermediate results inside the query plan, the pipelined model can efficiently process arbitrarily large volumes of data. Maintaining this property is one of the crucial design goals of the new iterator model presented in Section 4.2.

3.4 Column-at-a-time execution in MonetDB

The MonetDB system [Bon02] was designed specifically for analytical data processing. In these scenarios, the query load typically consists of a relatively small number of queries, but the queries are complex and process large amounts of data. To achieve high performance in such scenarios, MonetDB proposed alternative solutions in various layers of the database system.

The crucial difference between MonetDB and traditional systems is in the way data is processed. Instead of using the N-ary tuple model, MonetDB follows the ideas presented in the *decomposition storage model* (DSM) [CK85] and uses an algebra entirely based on *Binary Association Tables* (BATs) [BK99]. This influences the storage layer, query language and the execution layer implementation.

In the storage layer, BATs are simply two-column tables, where *head* and *tail*



people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

sel_id

(oid)	(int)
1	102
2	104
5	109
8	114
9	115

sel_name

(oid)	(str)
1	Ivan
2	Peggy
5	Walter
8	Zoe
9	Charlie

tmp

(oid)	(int)
1	7
2	15
5	1
8	12
9	5

sel_bonus

(oid)	(int)
1	350
2	750
5	50
8	600
9	250

Figure 3.5: Execution of a simple SQL query (see Section 3.1.1.3) in the MonetDB column-at-a-time execution model

columns can contain different data types, as presented in the left-most side of Figure 3.5. A similar data organization has been proposed before in the context of database machines, specifically for vector processors [TKK$^+$88]. Different attributes of the same tuple in an N-ary table are connected by using the value of object-id (*oid*) column, equivalent to the *surrogate* columns in [CK85]. For persistent data, this column typically contains a continuously increasing dense sequence of numbers, and is stored using a special *virtual-oid* (*void*) column type [BK99] that is not physically materialized. As a result, a BAT is often stored using a single column. For fixed-width data this format is equivalent to a simple data array. For variable-width types the storage is separated into two elements: a heap containing the actual data, and a fixed-width array of per-tuple positions in the heap.

The column-based approach in the storage layer has a significant impact on the I/O performance as well as the memory consumption. Since only columns that are actually used by a given query are fetched from disk, the volume of the transferred data becomes a fraction of what a system based on the N-ary storage would use. This is especially important with tables having a large number of columns, as is the case e.g. in data mining applications.

In the processing layer, MonetDB implements its binary algebra using the *column-at-a-time* approach: every operator is executed at once for all the tuples in the input columns, and its output is fully materialized as a set of columns. As a result, the query plan is not a pipeline of operators, but instead a series of sequentially executing statements, consuming and producing columns of data. For example, the execution of the example SQL query from Section 3.1.1.3 in Monet Interpreter Language (MIL) [BK99], is as follows:

```
sel_age   := people_age.select(30, nil);
sel_id    := sel_age.mirror().join(people_age);
sel_name  := sel_age.mirror().join(people_name);
tmp       := [-](sel_age, 30);
sel_bonus := [*](50, tmp);
```

The data flow for this query plan is presented in Figure 3.5. The resulting sel_* BATs constitute the final result. A more complex MIL example is presented in the left-most column of Figure 3.6, which shows the MIL code for the TPC-H Query 1.

The implementation of MonetDB operators is based on a principle of *no degree of freedom*. For every combination of task (e.g. select, sort), input data types (e.g. integer, string) and properties (e.g. sorted, nullable) a single specialized routine is created. Note that this approach would not be feasible in the N-ary model, as the number of possible combinations is too high, but it is maintainable in the binary model. When an operator is called, the version matching the input data types and properties is chosen and executed. Since the operator input is typically stored directly as arrays of values, and the entire input is processed at once, many operations boil down to simple loops over arrays. For example, a simplified code for a routine that selects from a `[void,int]` BAT identifiers of tuples bigger than a given constant and produces an `[oid,void]` result would look as follows:

```
int uselect_bt_void_int_bat_int_const(oid *output, int *input, int value, int size) {
    oid i;
    int j = 0;
    for (i = 0; i < size; i++)
        if (input[i] > value)
            output[j++] = i;
    return j;
}
```

Naturally, it is infeasible to manually implement and maintain all possible combinations of operators. MonetDB uses aggressive macro expansion using the Mx tool [KSvdBB96] that converts operator templates into dozens or even hundreds

MIL statement	SF=1				SF=0.001	
	data volume (MB)	result size (tuples)	time (ms)	band-width (MB/s)	time (usec)	band-width (MB/s)
s0 := select(l_shipdate).mark	45	5.9M	127	352	150	305
s1 := join(s0,l_returnflag)	68	5.9M	134	505	113	608
s2 := join(s0,l_linestatus)	68	5.9M	134	506	113	608
s3 := join(s0,l_extprice)	114	5.9M	235	483	129	887
s4 := join(s0,l_discount)	114	5.9M	233	488	130	881
s5 := join(s0,l_tax)	114	5.9M	232	489	127	901
s6 := join(s0,l_quantity)	68	5.9M	134	507	104	660
s7 := group(s1)	45	5.9M	290	155	324	141
s8 := group(s7,s2)	45	5.9M	329	136	368	124
s9 := unique(s8.mirror)	0	4	0	0	0	0
r0 := [+](1.0,s5)	91	5.9M	206	440	60	1527
r1 := [-](1.0,s4)	91	5.9M	210	432	51	1796
r2 := [*](s3,r1)	137	5.9M	274	498	83	1655
r3 := [*](s12,r0)	137	5.9M	274	499	84	1653
r4 := {sum}(r3,s8,s9)	45	4	165	271	121	378
r5 := {sum}(r2,s8,s9)	45	4	165	271	125	366
r6 := {sum}(s3,s8,s9)	45	4	163	275	128	357
r7 := {sum}(s4,s8,s9)	45	4	163	275	128	357
r8 := {sum}(s6,s8,s9)	22	4	144	151	107	214
r9 := {count}(s7,s8,s9)	22	4	112	196	145	157
TOTAL / average:	1361 (MB)		3724 (ms)	365 (MB/s)	2327 (usec)	584 (MB/s)

Figure 3.6: MIL code and performance profile of the TPC-H Query 1 (see Figure 3.2) running on MonetDB, scale factors 1 and 0.001 (from [BZN05])

of versions differing with input data types, properties etc. While this approach significantly increases the program size, it has highly useful properties influencing the execution performance:

instruction cache – Even though the overall code size is large, the cost of loading a given function is amortized over the entire input of an operator, making it negligible in most cases.

plan-data cache – Since only one operator is executed at any given time (within a single process), its state can make full use of the CPU cache. MonetDB was a platform used for some pioneering work in the area of cache-conscious databases, and provides a number of cache-conscious algorithms [MBK02, MBNK04].

function call overhead – For most operators, there are no per-tuple function calls happening, making the function call overhead negligible.

tuple manipulation – The data is typically stored as contiguous set of tuples, equivalent to e.g. C arrays. As a result, value access is direct, and does not require any interpretation.

superscalar CPUs utilization – The code inside the MonetDB operators usually has no function calls, has significantly fewer branches, and as a result works much better on modern CPUs. However, expensive main memory accesses often hinder the performance.

compiler optimizations – MonetDB operators code is simple, and many automatic compiler optimization techniques can be applied.

data volume – Since MonetDB uses columnar data representation, only the used columns are being transferred from disk and stored in memory. Additionally, since data is packed in dense arrays, the overhead of record structure present in NSM is avoided, resulting in smaller storage requirements.

Thanks to the above properties, the MonetDB execution model reduces the need of the expensive query plan interpretation and makes the CPU spend most of the time on the actual data processing. With the CPU-efficient operator code, this allows to achieve low cycles-per-tuple cost for large-volume data processing.

The materializing operator model has also benefits in the area of extensibility, profiling and query optimization. Since operators are fully independent, and internally typically perform simple array-processing tasks, new operators can be easily added. Also, query execution time is clearly divided into a sequence of consecutively executing steps. This allows easy profiling and determining possible optimization areas. Finally, before executing each operator, there is more information available than in the tuple-at-a-time model (e.g. the exact relation cardinality), exposing multiple runtime optimization opportunities.

While MonetDB in many areas demonstrates a significant performance improvement over the traditional tuple-at-a-time strategy, it also suffers from a number of problems. The most important one is related to the intermediate result materialization. Even during in-memory processing, writing the results by each operator can cause high memory traffic, making the operators not CPU-bound, but memory-bound. This can be observed e.g. by comparing the TPC-H Query 1 results for scale factors 1 (1GB) and 0.001 (1MB) presented in Figure 3.6 [1]. For SF=0.001 the data is small enough to fit the intermediate results in the CPU cache, and as a result, the per-operator bandwidth can be even 3 times higher than the memory-bound SF=1 case, and the overall execution is almost two times faster (2327 usec on a 1MB dataset versus 3724 ms on a 1GB dataset). The impact of the result materialization is even more visible on multi-CPU machines, where the memory bandwidth is shared among different CPUs [Zuk02]. Avoiding this problem is one of the crucial goals of the execution model introduced in Section 4.2.

The overhead of the intermediate result materialization is additionally increased in MonetDB by its use of a column-at-a-time algebra. With this approach, tasks involving *multiple* columns often become complicated. For example, an aggregation with multi-attribute keys needs to be decomposed into a number of steps. Similar problems occur in multi-attribute joins or sorting. Binary algebra also enforces *attribute post-projection* [MBNK04] – after an operation, all attributes "carried" through it need to be materialized, as is the case with the `id` and `name` columns in Figure 3.5. In all these cases, additional computational steps causes extra data materialization. These MonetDB problems suggest that, while the columnar storage is good for reducing the I/O cost and for allowing high processing performance, query plans should be expressed using the N-ary approach.

3.4.1 Breaking the column-at-a-time model

The high memory-bandwidth problem of MonetDB has been analyzed in [Zuk02] that focused on the in-memory execution on SMP machines. In such scenarios, the computational benefits of parallelizing queries are often seriously hindered by the relatively poor per-CPU memory bandwidth. In contrast, since cache memories are often dedicated to each CPU, the cost of accessing cache (assuming no cache coherency protocol overheads) does not increase with multiple CPUs.

The imbalance between the main-memory and CPU-cache bandwidth, especially visible in the multi-CPU environments, has led to the idea of *partitioned execution* [Zuk02, Section 4.3.3]. Here, instead of executing a sequence of column-at-a-time statements, columns are broken into smaller *slices*, and the operators execute on them in the pipelined fashion. With the slice size chosen

[1]2005 results from [BZN05]

such that the intermediate results fit in the CPU cache, in-memory materialization is avoided. As a result, performance is significantly improved, especially during parallel execution. In single-CPU execution, the performance benefits were relatively small, mostly due to the fact that the interpretation mechanisms were implemented in the MIL language, which is relatively inefficient for scripting. With the slice size typically in range of a few hundreds or a few thousands of tuples, the high overhead of MIL interpretation could not be well amortized.

While the performance improvement of partitioned execution was relatively limited, this research has hinted that combining the high performance of column-at-a-time operations with the pipelined execution strategy can both improve the already high performance of MonetDB, as well as allow it to reduce its memory-consumption related problems. This gave a motivation for the research presented in the following chapters of this thesis.

3.5 Architecture-conscious database research

In the previous sections we have discussed two existing execution models that are the base for the research presented in this thesis. This section broadens the picture by presenting the research from the area of *architecture-conscious database systems*. In this field, the performance of database systems on modern hardware is analyzed and improved, both in terms of the modifications to the discussed execution models, as well as new implementations of various data processing tasks.

3.5.1 Analyzing database performance on modern hardware

The detailed analysis of CPU performance on database workloads was first performed by the computer-architecture community [MDO94, CB94, BGB98, LBE$^+$98, KPH$^+$98]. In [MDO94] authors compared a few types of multi-user commercial workloads, including TPC-A [Tra94] and TPC-C [Tra07] benchmarks, simulating OLTP scenarios, with a set of numeric-intensive applications, mostly from the area of scientific computing. One of the observations was that the transaction-processing systems typically use significantly larger instruction footprint, with most instructions executing a relatively small number of times. This is related to the fact that these systems typically do not spend a lot of time in tight loops, as is the case in e.g. numeric processing. This directly impacts the

L1 I-cache performance – the percentage of I-cache misses for TPC-A is a few times higher than for scientific applications. Interestingly, for the L1 D-cache, the results were opposite – multi-user applications suffered from a significantly lower number of misses. For the L2 misses, the situation is slightly different – transaction workloads not only suffer from significant number of I-cache misses, but also D-cache misses are on a par with numeric workloads. Finally, it has been observed that the multi-user workloads, due to their event-based nature, spend a significant amount of time in the kernel space – 40% for TPC-A versus 7% for used non-database workload.

Similar experiments have been presented in [CB94], where the authors additionally analyze the performance of the sort operation, and provide more insight into the exploitation of the superscalar nature of the Alpha AXP CPUs. This paper demonstrates that already in 1994 transaction-processing workloads resulted in significantly higher cycles-per-instruction (CPI), and made inefficient use of the multi-pipeline architecture of the CPUs. Also, the impact of branch mispredictions has been demonstrated to be higher than in most numeric-intensive problems. As a result, it has been shown that transaction-processing programs can spend as little as 20% on actual computation, wasting rest of the time on various stalls, comparing to 30% in sort, and 80% in the Linpack benchmark. These problems have also been identified in [KPH$^+$98], where authors confirm OLTP problems with utilizing out-of-order execution, superscalar issues and branch prediction.

A comparable analysis of the OLTP and decision-support systems (DSS) workloads has been presented in [BGB98, LBE$^+$98, ADHW99]. In [BGB98], the authors demonstrate that the CPI of DSS scenarios is significantly better (factor 4) than in OLTP. Also, DSS systems have better code and data locality, resulting in significantly fewer cache misses. These results are confirmed in [LBE$^+$98], where authors present higher OLTP instructions footprint, and hence an increased number of I-cache misses. Similar conclusions are drawn in [ADHW99], where the authors identify L2 data-cache misses and L1 instruction-cache misses as crucial for performance.

Interestingly, recent analysis of the large-scale OLTP experiments [SK06] presents slightly different conclusions. On the Itanium platform, with the used transaction load, the instruction-cache stalls only contributed to less than 10% of the total time. On the other hand, data-cache misses, particularly expensive L3 misses, consumed ca.60% of total time. These results show that the exact characteristics of the performance highly depends on the used hardware platform, system architecture and query loads.

While this collection of papers is not exhaustive, it demonstrates that the performance of database systems is far from optimal on modern hardware, especially comparing to numeric-intensive scientific programs. As a result, there is an ongoing activity in the database research community to improve the database architecture by addressing the most important performance problems.

3.5.2 Improving data-cache

Perhaps the most visible inefficiency of classical database algorithms is related to the suboptimal use of the hierarchical memory systems. The impact of the non-uniform access cost has been identified quite early with a pioneering work of Shatdal et al [SKN94]. While in 1994 the difference of the cache-access and memory-access (2-4 cycles versus 15-25 cycles) was an order of magnitude smaller than now, even then cache-conscious algorithms allowed up to 200% performance improvement. Authors proposed a set of techniques that frequently reoccur in the following research, and include: *blocking* – reuses chunks of data that fit in the cache; *partitioning* – a variant of blocking, divides data into cache-sized segments; *key extraction* – reduce the volume of processed data by using only attributes relevant to the current operation; *loop fusion* – combine multiple operations in a single pass over data to reduce memory traffic; *clustering* – rearrange attributes to improve spatial data locality. An early example of using some of these ideas has been presented in the AlphaSort [NBC$^+$95] algorithm, which minimized the number of cache misses. It was achieved by using cache-sized data units with cache-friendly QuickSort [Hoa61], processing {key-prefix, pointer} pairs instead of full records to allow more elements to fit in the cache, and using a cache-friendly replacement-selection tree for merging the runs.

Improving the performance by cache-conscious data reorganization has been proposed in [CHL99], where the authors rearrange elements of pointer-based data structures to increase locality of references. Rao and Ross address similar problem focusing on tree structures [RR99, RR00]. First, in [RR99], they suggest organizing a tree in a read-only array, eliminating the need of storing data pointers. Internal nodes are chosen to fit the cache-line size, optimizing the number of cache-line references, and highly-optimized in-node search routines are used for faster lookup. This work is extended to update-enabled *cache-sensitive B^+ Trees* (CSB^+-Trees) [RR00], which modify the B^+-Tree organization such that all the children of a given node are stored contiguously. This reduces the volume of data and hence the number of cache misses.

Vertical data representation in DSM [CK85], has been identified by Boncz et al. [BMK99] to have beneficial impact on cache behavior of applications, due to reduced memory traffic and an increased spatial locality. While this research has been performed in scope of column-stores, Ailamaki et al. [ADHS01] took this idea further, by proposing a hybrid data storage model, called PAX. In this model, the layout of a disk page is modified to store the same attribute from different tuples contiguously, like in DSM. PAX has the I/O characteristics of NSM, and cache-characteristics of DSM, and has been shown to improve query performance by as much as 48%. The *data-morphing* technique [HP03] identifies the in-memory performance differences between DSM and NSM and generalizes the PAX approach by dynamically dividing a relation into a set of vertical partitions, stored in a PAX-like, mini-page based manner. In all these solutions the model used for storage and for processing is the same. Clotho [SSS$^+$04] decouples these two issues by transparently choosing the storage model most suited for the current workload.

Another approach to optimize cache-memory behavior has been proposed in [ZR03b], where the accesses to nodes in tree-based data structures are buffered, and tree-traversal operations are performed in bulk. This increases the *temporal locality*, as the same cache-lines are used multiple times in one operation. As a result, the throughput of the operations on both cache-conscious and traditional tree-structures is significantly improved, at a cost of an increased response time of single operations.

The early approaches on using the in-memory partitioned hash-join [SKN94] has been extended in work of Boncz et al. [BMK99, MBK00, MBK02], This research found the high-fanout partitioning necessary to split large data volumes into cache-sized sub-relations to result in poor cache and TLB behavior. The proposed multi-pass *radix-cluster* partitioning strategy eliminates this problem, and while performing more work, results in a better overall performance. This work also proposes optimizations to the partitioning code, as well as highly accurate cost-models for choosing the best partitioning method and predicting its performance. This work is further extended in [MBNK04], where authors introduce a cache-friendly *radix-decluster* attribute post-projection algorithm.

Another approach to address the imbalance between the cache and memory latency is to use *data prefetching.* As discussed in Section 2.2.3, modern CPUs provide both implicit and explicit cache prefetching capabilities. Explicit prefetching has been suggested for the general problem of recursive, pointer-chasing, data-structures [LM96, LM99]. In the database community, a series of papers by Chen et al. [CGM01, CGMV02, CAGM04, CAGM05, CAGM07] discusses how these techniques can be applied to database operations. [CGM01] presents how prefetching can be exploited to increase the width of the nodes in a B^+-trees without paying the latency cost of fetching the additional cache-lines, resulting in a reduced tree-depth and hence *lookup* performance boost of up to 55%. Additionally, the authors discuss how prefetching can improve *range-scanning* performance, where the performance gains can be as high as a factor 8.7. This technique has been further extended to disk-resident *fractal prefetching* $B^+ - Trees$ that use cache-conscious in-page tree organization and processing for good performance, as well as I/O prefetching for improved range scans. [CAGM04, CAGM07] discuss how *group prefetching* and *software-pipelined prefetching* techniques can be applied when performing hash-join operation on a set of tuples. These methods were shown to perform better than cache-partitioning and also be more resistant to interference by other programs. Memory prefetching has also been applied to optimize various data accesses in the *inspector join* algorithm [CAGM05].

It is interesting to see that most of the techniques proposed for improving memory performance by using cache memory have a direct correspondence with similar techniques for improving disk performance by using main memory: multi-pass partitioning is necessary e.g. for external hash-join with a high number of partitions; disk prefetching is commonly applied for scans and index lookups; reducing I/O volume with vertical storage is the major beneficial feature of column stores; and so on. This shows that the problems addressed by the research presented in this section are relatively generic, and only particular hardware constants slightly differ. Since the memory hierarchy continuously becomes more complex, it is possible that in the future similar approaches might need to be applied at different levels of it. This observation gave birth to the area of *cache-oblivious algorithms* [FLPR99, HL07], where the exact characteristics of the hardware are ignored, and algorithms and data structures are designed to maximize the spatial and temporal locality, and hence work well on any cache configurations.

3.5.3 Improving instruction-cache

Instruction cache misses have been identified as a major problem for database performance, especially for OLTP workloads [BGB98, LBE$^+$98, ADHW99]. The main reason for it is poor temporal locality of the accessed instructions – most database systems change position in code very frequently, especially with tuple-at-a-time processing.

Harizopoulos and Ailamaki [HA04] observed that while at a given moment different concurrently running transactions typically use different parts of the program, their overall code paths overlap significantly. To exploit this, they suggest organizing multiple queries needing a particular operator into *execution teams*, and evaluate this operator for all members of the team one after another.

This makes the queries share the instruction-cache, which significantly reduces the number of misses, up to 96.7%. Additionally, better temporal locality of the executed code increases the efficiency of branch predictor, allowing for reduction of mispredicted branches by up to 64%. In the result, this technique has been shown to provide an overall transaction-processing speedup in a live system of up to 31%.

Zhou and Ross [ZR04] observed that the instruction-cache misses are also a significant problem in analytical processing queries, even within a single query. This happens when the instruction footprint of the entire query plan exceeds the CPU I-cache size. To reduce this problem [ZR04] introduced a new *Buffer* operator that saves the tuples coming from the child, and sends then to the parent once the number of tuples reaches a given threshold. This effectively decomposes a query plan into a set of partially-materializing sub-plans, and if the buffering happens in the proper locations in the plan, each of the sub-plans is small enough to fit in the I-cache. As a result, this method can reduce the I-misses by 80% and improve the overall performance by 15%. This approach is especially useful in OLAP queries, where the same operation is performed for large number of tuples in one query.

3.5.4 Exploiting superscalar CPUs

Database systems have not only been identified to work sub-efficiently with cache memories, but also have been shown to poorly utilize the superscalar capabilities of modern CPUs [CB94, ADHW99]. This is especially visible in the *cycles-per-instruction (CPI)* ratio, which for databases is typically in range of 1.2 1.8 for TPC-D benchmark and 2.5 to 4.5 for TPC-C benchmark [ADHW99]. For comparison, modern CPUs can issue a few instructions per cycle, and optimized programs can achieve a CPI of 0.5 or less. The high CPI of database loads can be partially explained by instruction and data stalls, but other factors influence it as well. One of them is the highly branching nature of the traditional tuple-at-a-time model: for every tuple multiple comparisons (e.g. check for data types or computation errors) and various function calls need to be performed. Also, since there is only a single unit of computation at a time (a tuple), there are relatively little *independent* instructions that can be executed in parallel.

Improving the performance by reducing the number of comparisons was proposed in [Aga96], where authors propose the comparison-free *radix-sort* in place of commonly applied, comparison-based *quick-sort* and *merge-sort* algorithms. In [Ros02], Ross analyzed the impact of branch mispredictions when evaluating selection conditions, and presented techniques to replace control-hazards resulting from conditional branches with data-hazards that have smaller impact on performance. The reduction of the number of function calls and branches is also one of the effects of the block-oriented processing [PMAJ01].

Another related functionality of modern CPUs are the SIMD instructions. In the database community, there have been relatively little research in this area. In [ZR02], Zhou and Ross apply SIMD instructions to a subset of database operations. The results show that with proper implementation, this method can give speedup close to the number of elements SIMD instructions process, or even higher, if SIMD instructions allow to reduce the number of branches. In [Ros07] SIMD instructions are used to optimize the processing of the hash tables. SIMD potential can also be exploited to improve parallel predicate evaluation, as presented in [JRSS08]. Finally, SIMD-processing is used heavily on many alternative hardware platforms, discussed in Section 3.5.6.

3.5.5 Intra-CPU parallelism

For a long time parallel execution in databases has been exploiting *inter-node parallelism*, using different machines, and *intra-node parallelism*, using multiple CPUs in one machine. Recently, with the increased popularity of the SMT and CMP architectures, *intra-CPU parallelism* becomes an important research area.

The first analysis of the SMT potential for database system was presented in [LBE$^+$98], where the SMT performance was modeled using database logs. The experiments showed that with SMT significantly more instructions are issued every cycle, and memory-access latencies can be tolerated to some extent. As a result, the performance on 8-context SMT processor improved 3-fold for OLTP and 1.5-fold for DSS. [ZCRS05] introduces two SMT-specific methods of executing relational operators. In the *bi-threaded* implementation, the work of an operator is distributed evenly among concurrently executing threads. In the *working-set* version, the “helper thread” knows what data will be needed by the “main thread” and preloads it in advance. Both strategies have been evaluated in row-wise and column-wise record layouts, and shown up to ca. 20% improvement over a standard, SMT-oblivious parallel algorithm.

The potential of CMP architectures for databases and the impact of various CPU-architectural choices have been presented in [HPJ$^+$07]. The authors discussed two types of CMP designs: complex wide-issue, out-of-order, deep-pipelined *fat-camp* (FC) chips, and simple in-order, multi-threaded, simple-pipelined *lean-camp* (LC) chips. The results show that looking at the query response times LC can be worse by up to 70% in DSS loads and up to 12% in OLTP loads. Still, in multi-query saturated scenarios, LC chips can achieve up to 70% higher overall system throughput. This paper also analyzes how the growing L2 cache sizes increase the L1-hit times, and as a result can have detrimental effect on the overall system performance.

The technical challenges of exploiting CMP machines have been investigated in [CRG07] and [CR07]. In [CRG07] authors propose a parallel buffer structure that enables sharing input and output between the concurrently working operators, providing load balancing at the same time. This work is continued in [CR07], where various parallel aggregation algorithms are proposed for the UltraSPARC T1 chip. The results show that the new architectures require new algorithm designs that make good use of shared L2 cache and exploit available atomic instructions to reduce the need of locking.

3.5.6 Alternative hardware platforms

After the unsuccessful attempts of creating database-specialized hardware in the field of *database machines* (e.g. [AvdBF+92]), research in the high-performance database processing has been focused on exploiting the traditional disk–memory–CPU data flow path and CPU-based processing. However, recent technology developments result in computer architectures where the computational capabilities of a machine are divided between more than one type of devices, resulting in research targeting completely new hardware platforms.

The most visible example of shifting processing to an alternative hardware are *graphics processing units* (GPUs, see Section 2.1.7.3). In [GLW+04] Govindaraju et al. proposed GPU-based implementations of a set of database operations, including predicate evaluation and aggregations. The used GPU processing model is significantly different from traditional CPUs, as it depends heavily on SIMD instructions and cannot efficiently perform random memory accesses. This makes the implementation of various database operations challenging. Still, the presented performance improvement of 2–4 times suggests a huge computational potential of GPUs, as demonstrated in further research on sorting [GGKM06] and joins [HYF+07]. GPUTeraSort [GGKM06] is a GPU-based sorting algorithm that provided the top sorting performance in the PennySort benchmark [NBC+95]. The comparison of CPU-based versus GPU-based join algorithm presented in [HYF+07] shows that also for this operation GPUs can provide order of 2-20 improvement. Combining these standalone techniques into a coherent GPU-based system has been suggested in the GPUQP system [FHL+07]. The rapid improvements in the GPU technology, in terms of both performance as well as programming flexibility (e.g. with the recent NVIDIA

CUDA project [NVI08]), suggests that this approach might become increasingly important in the future of high-performance database systems.

Other alternative hardware architectures have also been used as platforms for database processing. In [GAHF05] a set of relational operators have been evaluated on an Intel IXP2400 network processor with 8 simple low-frequency microengines, each supporting 8 thread contexts, and using an explicitly controlled memory architecture. While exploiting this architecture introduces various implementation challenges, the results show that the high on-chip parallelism allows from 1.9 to over 3 times improvement over a general-purpose CPU. A similar hardware architecture is present in the STI Cell chip [IBM07] that uses 1 general-purpose PPU core and 8 SIMD-specialized SPU cores with a different ISA and no cache. Full utilization of these specialized cores often requires careful algorithm redesign. This has been presented in [Ros07], where Ross demonstrated how a different hash structure results in a 10 times performance improvement. Similarly, in [GBY07] and [GYB07] Gedik et al. presented that properly engineered algorithms on Cell can provide the performance improvement of a factor 4 for sorting (using 16 SPUs) and a factor 8.3 for stream joins, comparing to a dual Intel Xeon machine.

Another hardware area in which database technology is being used are embedded devices. For example, in [BBPV00] the authors analyze the challenges of implementing a database on a smartcard. These devices have significantly different hardware characteristics: slow writes, very little memory etc. As a result, the proposed PicoDBMS system needs to introduce new ideas in the areas of data organization (optimizing for data compactness) and query execution (processing with limited RAM).

3.5.7 Analyzing and improving database I/O performance

While most databases follow the traditional NSM data storage format, there has been a substantial amount of research investigating other solutions. For example, NSM is known to have poor performance for scan-intensive applications, where only a subset of relation attributes is accessed. To improve this, the *decomposition storage model* (DSM) has been proposed [CK85], where all attributes are stored in separate files, and only the used attributes are scanned. This format is also known as *column storage*. Originally, DSM introduced the idea of an extra *surrogate column*, containing key values used to identify tuples in different files. Modern column stores do not require such a column, and use the natural order (possibly involving compression) to reconstruct tuples [SAB$^+$05, ZR03a, BZN05]. However, DSM introduces a need to access multiple disk locations if a single tuple needs to be accessed, which can be a significant problem for sparse lookups and updates – this problem is absent in NSM.

The analysis of NSM and DSM presented in [HLAM06] has shown that DSM can provide a performance advantage for queries reading only a relatively small number of attributes (up to ca. 30% with settings used in the paper), and NSM is better for queries reading a large fraction of the table attributes. These results are somewhat surprising, as, using large enough I/O units (achieved in [HLAM06] with prefetching), the I/O cost of the operations should be proportional to the width of the scanned attributes, making DSM beneficial for all scans accessing a subset of columns. The reason for the low tuple width percentage at which NSM becomes beneficial in [HLAM06] is that both storage models are executing as a data source for an N-ary execution model. For the DSM model, this results in the tuple-reconstruction phase consuming extra CPU time.

The different characteristics of DSM and NSM resulted in a substantial number of approaches that combine different features of these two models. The PAX storage model [ADHS01] and the data morphing technique [HP03] discuss the in-memory properties and both models, and propose hybrid solutions (see Section 3.5.2). The *fractured-mirrors* approach [RDS02] suggests keeping two copies of data on two disks, like in RAID-1 [PGK88], but storing one copy in NSM and another in DSM. This allows using the best model depending on the task, as well as combining both mirrors in a single query for even better performance. The *multi-resolution block storage model* (MBSM) [ZR03a] investigates the physical placement of the DSM-based data. In this approach, the good scan performance of DSM is preserved, while the cost of tuple reconstruction is reduced, since values of different attributes from the same tuple are stored closer on disk than in the naive DSM implementation.

A series of papers discusses the "five-minute rule" [GP87, GG97, Gra07], which says that if an item is accessed roughly every five minutes, it should be main-memory resident for good cost efficiency. Still, the page size at which this break-even interval is applicable continuously increases (1KB in 1987, 8KB in 1997, 64KB in 2007). This trend is caused by the imbalance between the improvements in disk bandwidth and latency.

Another research direction in the I/O optimization area is related to the increasing popularity of solid-state storage devices (see Section 2.3.3). Two major features of these systems are typically analyzed: low random-read latency, and the performance difference between slow general updates and fast 1-to-0 updates. Graefe [Gra07] not only revisits the RAM-disk five-minute rule, but also discusses how low flash latency introduces similar rules for RAM-flash and flash-disk transfers. Shah et al [SHWG08] investigate a new join algorithm that depends more than traditional approaches on random accesses to disk, efficiently supported by flash. Ross [Ros08] presents a number of algorithms tuned for “write-once-then-erase” nature of flash device. This work is related to the previous research on write-only storage systems [Mai82, RS82], but differs due to the “erase” functionality of flash.

While MEMS devices (see Section 2.3.4) are not yet publicly available, they have been already investigated for database applications. In [YAA03] authors exploit the two-dimensional nature of these devices, and propose a storage model mapping two-dimensional relational tables on these. In related work, Schlosser et al. [SSAG03] demonstrate how the inherent MEMS parallelism can be exploited to provide efficient execution of both row-oriented and column-oriented requests.

3.6 Conclusions

This chapter presented an overview of the database technology, concentrating on the differences between the traditional tuple-at-a-time and the MonetDB column-at-a-time processing model. Both architectures provide a unique set of benefits and problems. Additionally, a set of hardware-related database optimization techniques has been discussed. Combining the best elements from all these areas is the goal of the new approach to the query execution, proposed in the next chapter.

Chapter 4

MonetDB/X100 overview

The analysis of database architectures presented in the previous chapter has shown how existing execution models fail to fully exploit the performance potential of modern computers on all layers of the hardware stack: super-scalar CPUs, cache memories, main memory and disk. This motivates the research on a new database architecture presented in this chapter.

MonetDB/X100 is a prototype query processing engine that allows us to investigate new techniques in the field of efficient data processing for data-intensive applications. The techniques introduced in this engine focus on the areas of CPU-efficient execution and bandwidth-optimized storage, corresponding directly with the main thesis of this book, defined in Chapter 1:

> *With the* ***vectorized execution model*** *database systems can minimize the instructions-per-tuple cost on modern CPUs and achieve high in-memory performance, but* ***bandwidth-optimizing improvements*** *in the storage layer are required to scale this performance to disk-based datasets.*

Vectorized execution model. In the execution layer, we try to overcome the inefficiencies of the existing approaches described in Chapter 3. The tuple-at-a-time pipelined model requires expensive query-plan interpretation, resulting in a high *instructions-per-tuple* number. Additionally, this model is unfriendly for modern CPUs, causing poor utilization of super-scalar CPU features and cache memories, resulting in poor *cycles-per-instruction* ratios. The bulk-processing model of MonetDB overcomes these problems, and achieves high CPU efficiency. However, this comes at a cost of intermediate result materialization, hindering the overall performance and limiting scalability. As a result, both models, especially the tuple-at-a-time one, can suffer from a high *cycles-per-tuple* cost during in-memory processing. In Section 4.2, we introduce a new *vectorized in-cache execution model* that combines the best features of the tuple-at-a-time model and bulk processing, providing performance often orders of magnitude higher than the existing solutions.

Bandwidth improvements. In the storage layer, systems following the NSM approach result in suboptimal exploitation of the disk bandwidth and available buffer memory. Moreover, even for data-intensive applications, databases often rely heavily on random-access methods that require many disk arms per CPU-core to provide sufficient data delivery rates, leading to installations with hundreds of disks even for small data warehouses [ZHNB06]. This approach is unsustainable due to the imbalance in the evolution of hard disk latency and bandwidth, as discussed in Chapter 2. Section 4.3 proposes an alternative approach, where data-intensive tasks are implemented using mostly scan-based methods, and presents ColumnBM – a new storage system designed for this type of processing. While this system reduces the bandwidth requirements by using column-based storage, providing enough data for the highly efficient execution layer is still an important problem, especially since fast improvements in the CPU technology allow more and more queries to be executing at the same time. As a result, further bandwidth-improving optimizations are necessary, and ColumnBM introduces two such techniques: lightweight compression and cooperative scans.

This chapter presents the major features of the proposed architecture, and serves as a foundation for detailed discussion of the key introduced techniques, presented further in Chapters 5, 6 and 7.

4.1 MonetDB/X100 architecture

Two major design goals of MonetDB/X100 is to *(i)* execute high-volume queries at high CPU efficiency, and *(ii)* scale this efficiency to large, disk-resident data volumes. To achieve these goals, MonetDB/X100 fights performance bottlenecks throughout the entire computer architecture, as visualized in Figure 4.1:

CPU To avoid the overhead present in the tuple-at-a-time model, MonetDB/X100 follows the bulk-processing idea of MonetDB. It performs all operations on data using simple *primitive* functions that operate on multiple values in one call (see Section 4.2.1.3). This reduces the number of function calls,

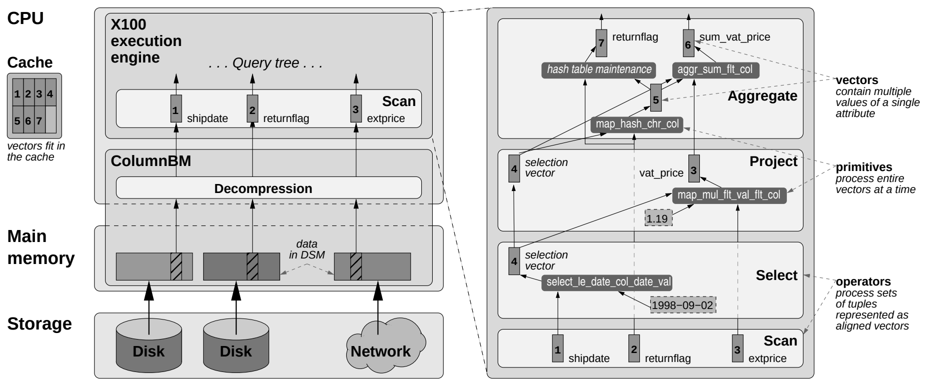


Figure 4.1: MonetDB/X100: architecture overview (left) and the query execution layer (right)

improving the instructions-per-tuple ratio and increasing the code locality. Additionally, primitives typically consist of simple loops over multiple input values. This exposes multiple compiler-level optimization opportunities, and allows efficient execution on super-scalar CPUs.

Cache MonetDB/X100 avoids the intermediate result materialization overhead of MonetDB by combining the bulk-processing approach with the pipelined iterator model. Instead of single tuples or full columns, the operators exchange data in the form of *vectors* – small (ca. 100-1000 elements) arrays of input values (see Section 4.2.1.1). These vectors are fully cache-resident, removing the need of expensive memory accesses. Furthermore, relational operators are internally implemented using cache-efficient algorithms.

RAM The buffered disk data is stored in a vertical layout, often compressed, maximizing the amount of useful information that can be kept in memory. Main-memory bandwidth is minimized by only reading relevant attributes and by decompressing the buffered compressed data on the boundary between RAM and cache. Additionally, RAM access is seen in many cases as an input-output operation, and is carried out through explicit memory-to-cache and cache-to-memory routines.

Disk The ColumnBM I/O subsystem of X100 is geared towards efficient sequential data access. To reduce bandwidth requirements, it uses a vertically fragmented data layout, in some cases enhanced with lightweight data compression (see Chapter 6). Furthermore, it coordinates the scan operators of different queries to maximize data sharing and minimize disk accesses (see Chapter 7). Finally, it minimizes the need for physical data reorganization on updates by using in-memory delta update structures.

The major components of the system, the query execution and storage layers, are discussed in the following sections. Other components typically found in databases, such as the query optimizer or logging and recovery facilities, are not implemented in the current MonetDB/X100 prototype, due to its clear focus on the efficient query execution techniques. These components are in many cases orthogonal to the techniques presented in this thesis, and might become a part of the system in the future.

4.1.1 Query language

MonetDB/X100 uses a simple relational algebra as the query language. The algebra is defined on the *physical* level of operations, so, for example, different physical types of an aggregation operator are expressed explicitly. A simplified version of the TPC-H query, used as an example in the right-hand side of Figure 4.1, is expressed in SQL as:

```
SELECT   returnflag, sum(extprice * 1.19) as sum_vat_price
FROM     lineitem
WHERE    shipdate <= date '1998-09-02'
GROUP BY returnflag
```

This can be translated into the following MonetDB/X100 query:

```
HashAggr(
   Project(
      Select(
         Scan([ 'shipdate', 'returnflag', 'extprice' ]),
         <=( shipdate, date('1998-09-02'))),
      [ vat_price = *( extprice, flt('1.19') ])),
   [ returnflag ],
   [ sum_vat_price = sum(vat_price) ])
```

The MonetDB/X100 version of the full TPC-H Query 1 is presented in Figure 4.2.

Unlike in MIL (MonetDB query language), MonetDB/X100 operators are *N-ary*, i.e. can have an arbitrary number of input attributes, making the plans highly similar to the ones used by traditional RDBMSs. All the computations, including simple arithmetic, are expressed as functions using prefix notation.

```
Sort(
   Project(
      HashAggr(
         Project(
            Select(
               Scan(
                  [ 'l_shipdate',             // input columns
                    'l_returnflag',
                    'l_linestatus',
                    'l_quantity',
                    'l_extendedprice',
                    'l_discount',
                    'l_tax' ] ),
               <=( l_shipdate, date('1998-09-02') )),
            [ l_returnflag,                   // pre-aggregation projections
              l_linestatus,                   // and computations
              l_quantity,
              l_extendedprice,
              l_discount,
              discountprice = *( -( flt('1.0'), l_discount), l_extendedprice),
              charge = *( +( flt('1.0'), l_tax), discountprice) ]),
         [ l_returnflag,                      // group-by attributes
           l_linestatus ],
         [ sum_qty = sum(l_quantity),         // aggregate functions
           sum_base_price = sum(l_extendedprice),
           sum_disc_price = sum(discountprice),
           sum_charge = sum(charge),
           sum_disc = sum(l_discount),
           count_order = count() ]),
      [ l_returnflag,                         // post-aggregation projections
        l_linestatus,                         // and computations
        sum_qty,
        sum_base_price,
        sum_disc_price,
        sum_charge,
        avg_qty = /( sum_qty, cnt=dbl(count_order)),
        avg_price = /( sum_base_price, cnt),
        avg_disc = /( sum_disc, cnt),
        count_order ]),
   [ l_returnflag ASC,                        // final sort parameters
     l_linestatus ASC ] ))
```

Figure 4.2: TPC-H Query 1 (see Figure 3.2) in MonetDB/X100 algebra

These functions map directly onto *primitives* described later. The output attributes are implicitly defined for each operator. For example, `Select` propagates all the attributes from its child, `Project` explicitly defines which child attributes are propagated and what new attributes are introduced, and `HashAggr` returns the input columns used in the group-by list and the results of the aggregate functions.

MonetDB/X100 operators can process two types of input data: *dataflows* and *tables*. A dataflow contains tuples delivered in a pipelined fashion. Tables are a special version of dataflows, where the entire input is provided in a single iteration step. This allows operators that e.g. perform random accesses into the data or need to read the same data multiple times. The list of the major operators currently available in MonetDB/X100 is presented in Table 4.1.

4.2 Vectorized in-cache execution model

To provide a high-efficiency query execution engine for data-intensive applications, we propose a new execution model, used in MonetDB/X100, that keeps the benefits of the existing approaches, while avoiding their problems. To maintain scalability, this new model needs to work in a pipelined fashion, avoiding expensive materialization. On the other hand, to achieve high CPU performance, it needs to avoid expensive per-tuple interpretation by processing multiple values at once. This analysis has led to the formulation of an execution model presented in this chapter, based on the concept of *vectorized in-cache execution*.

4.2.1 Vectorized iterator model

Like in the traditional pipelined model, MonetDB/X100 query plans consist of a tree (or a DAG) of *operators*. However, instead of single tuples stored in the N-ary format, they operate on *vectors* of values. Operators provide generic high-level logic, and use *primitives* – specialized, type-specific functions – for actual data processing. These concepts, visualized in the right-hand side of Figure 4.1, are the basic components of the MonetDB/X100 execution layer.

4.2.1.1 Vectors

Vectors are the basic data manipulation and transfer units in the vectorized execution model. Each vector contains a simple, one-dimensional array of values, similar to the `[void,type]` BATs found in MonetDB. The number of elements in a vector may vary, and typically ranges from 100 to 1000 tuples.

Operator	Definition
	Basic operators
`*Scan`	A family of operators providing a dataflow of tuples coming from different sources: ColumnBM storage (`ColumnScan`), MonetDB BATs (`BatScan`), simple binary files (`RawScan`), PAX-formatted [ADHS01] binary files (`PaxScan`).
`*Save`	A family of operators that save the input, fully symmetric to `*Scan`. Includes `ColumnSave` (with optional compression), `BatSave`, `RawSave` and `PaxSave`.
`Print`	Displays the input values.
	Unary (one input relation) operators
`Select`	Returns tuples that match a specified predicate.
`Project`	Performs column projection and introduces new columns with values computed from the input columns. Does not perform duplicate elimination.
`HashAggr`	Performs aggregation on the input. If the aggregate keys are not provided, global aggregates are computed. If the aggregate functions are not provided, it performs duplicate elimination.
`MergeAggr`	Similar to `HashAggr`, but assumes the input to be clustered by the key values.
`FixedAggr`	Similar to `MergeAggr`, but assumes the input to be clustered in groups of a predefined size.
`TopN`	Selects top tuples according to some ordering criteria.
`ExternSort`	Performs a partitioned external radix-sort.
	Binary (two input relations) operations
`Hash*`	Hash-based in-memory operations. Provide different variants of joins (N-1, N-N, inner, outer etc.) and set operations (union, diff etc).
`Merge*`	Similar to `Hash*`, but assumes both inputs are sorted by the key values.
`FetchJoin*`	Joins based on join-indices. Requires a table as the right input.
`CartProd`	Produces a Cartesian product of two inputs. Requires a table as the right input.
	Other operations
`Window`	Returns tuples from a pre-specified range (counted as the position in the dataflow).
`FlowMat`	Materializes an arbitrary dataflow into a table.
`Reuse`	Allows multiple consumers to read from the same input data. It adapts to situations when the consumers are not synchronized, and buffers the tuples accordingly.
`Chunk`	Creates a dataflow with a new vector size. Additionally condenses the data if the input has a selection vector.
`BEP`	Performs Best-Effort Partitioning on the input (see Section 5.4).
`Array`	A dataflow generating a sequence of all indices in the N-dimensional space, similar to *grid* in [BS92].

Table 4.1: Main operators available in the MonetDB/X100 algebra

Vectors are typically *transient* – during consecutive iterations of a given operator they contain different data. Internally, the pointer to the actual data can be dynamic, allowing e.g. zero-cost access to a large sequential data structure (e.g. a disk page in the buffer pool). For variable-size data types, a vector can have an associated *heap* structure. This allows temporary storage of e.g. string values.

Vectors are used to pass data between operators, but they are also used heavily inside the operators. For example, the hash-join implementation discussed in Section 5.3 uses vectors to keep the internally computed hash-values of input tuples.

In traditional block-oriented processing, tuples are typically stored in the N-ary form [PMAJ01]. The rationale of using vectors holding vertically decomposed data comes from two observations. First, with this solution operations that add or remove columns are trivial and do not need to copy any data. Second, as discussed further in Section 5.2.2, dense data packing in vectors removes the need for tuple navigation, provides better sequential memory access and exposes SIMD opportunities.

4.2.1.2 Operators

Operators are the building blocks of the query plan. They operate in a pipelined, pull-driven fashion, as in the Volcano model. The difference is that instead of single tuples, operators exchange data using vectors. A collection of per-attribute vectors represents a set of tuples with values of a given tuple stored at the same position in all vectors. To reduce the need for data copying, it is possible to define that only a subset of the tuples needs to be processed. This is achieved by using an optional *selection vector* – an array containing offsets of the relevant tuples.

Operators provide generic logic, independent of the input data types, properties etc. To perform the actual work, they use primitive functions, described below. For example, the pseudo code for the `next()` method of the `Select` operator looks like this:

```
int Select::next() {
    while (true) {
        n = child->next();      // see how many tuples are produced by the child
        if (n == EndOfStream)
            return EndOfStream;
        n = condition->evaluate(n);
        if (n > 0)              // something was selected
            return n;           // return the number of selected tuples
    }
}
```

This logic is very similar to the tuple-at-a-time model (see Section 3.3), but instead of operating on a single tuple, it is vector-based. In this code, `condition` may be an arbitrary *expression* that computes the desired predicate. Expressions are components that glue the primitives with their associated input and result vectors. In this case, the result vector of the `condition` expression is being used as the selection vector of the parent operator. Expressions can be nested – for example, evaluation of `condition` may trigger the evaluation of its sub-expressions. The leaves in an expression tree can be expressions exported by the child operators, or intermediate expressions produced by a given operator (e.g. `discountprice` in Figure 4.2).

As mentioned, the operators in MonetDB/X100 are N-ary, allowing an arbitrary number of input attributes. However, the primitives used to perform the operations can only work on a limited combination of input types. As a result, internally, operators need to introduce per-attribute logic. For example, the pseudo code for the `next()` method of the `Project` operator is:

```
int Project::next() {
    n = child->next();
    if (n == EndOfStream)
        return EndOfStream;
    for (i = 0; i < expressions->size(); i++)
        expressions[i]->evaluate(n);
    return n;                 // return the number of selected tuples
}
```

The operator logic, trivial in case of `Project`, becomes significantly more complex in operators such as hash-join and merge-join, especially if there are multiple key columns. The problems with providing vectorized versions of such operators, as well as implementation techniques and examples for some typical operations are discussed in Chapter 5.

4.2.1.3 Primitives

While operators provide high-level generic logic, primitives are the components that perform all the operations on the actual data. Similar to operators in MIL, primitives are highly specialized for a given task and data type combination. However, the granularity of operation is much smaller. For example, in MIL an operator would perform the entire process of hash-aggregation for the entire column, involving hash number computation, finding a position in a hash table, updating aggregates etc. In MonetDB/X100, each of these tasks is implemented by one or more primitives, with operator logic gluing them together. As a result, the total number of primitives is even higher than in MonetDB, but they are significantly simpler than the full operators, making the code footprint relatively small. Like MonetDB, MonetDB/X100 relies on the Mx tool [KSvdBB96] with heavy macro expansion to implement the primitives. This results in a large number of short and efficient routines, usually performing a single loop over the data e.g.:

```
int map_add_sint_col_sint_val(int n, sint *result, sint *param1, sint *param2) {
    for (int i = 0; i < n; i++)
        result[i] = param1[i] + *param2;
    return n;
}
```

Each primitive is identified with a *signature*, defining the function of the primitive, its input and output types etc. For example, a primitive adding a signed-integer vector to a constant has a signature `map_add_sint_col_sint_val`, with `col` representing a column of values (vector), and `val` representing a single value (constant). Primitives are organized into groups defining their functionality. For example, "map" primitives produce a single value for each input tuple (e.g. math operations); "select" primitives generate a selection vector according to a predicate; "aggr" primitives take care of aggregation-specific operations etc. This allows determining which primitive needs to be chosen for a specific operation. For example, different primitives that check if float values are greater than a constant will be used if it is used to select tuples in the `Select` operator (`select_gt_flt_col_flt_val`) or if it is used to compute a Boolean value in the `Project` operator (`map_gt_flt_col_flt_val`). Currently, for every signature there is exactly one primitive. However, it is possible to extend this scheme to allow multiple primitive implementations, with one dynamically chosen depending on e.g. a CPU type, similarly as is done in LibOIL [Lib]. A choice can also be based on query and data properties, e.g. depending on predicate selectivity, a different `select_*` implementation can be more efficient [Ros02].

Another possible extension is to allow *compound primitives* – primitives that perform complex operations. For example, the computation of the Mahanalobis distance, a performance-critical operation in some multimedia retrieval tasks [CvBdV04], can be defined for some cases as follows:

$$D_M(x, y, \sigma) = \sqrt{\frac{(x-y)^2}{\sigma}} \tag{4.1}$$

In MonetDB/X100, this can currently be expressed as an explicit computation:

```
    sqrt(/(sqr(-(x, y))), sigma)
```

This will construct a tree of expressions, with one primitive for each mathematical operator. In this tree, the intermediate results at all evaluation steps need to be materialized, which introduces a small, yet visible overhead. Another approach is to manually implement a `map_mahanalobis` primitive containing the entire evaluation, and call it explicitly in the query plan. This can be done either with manual C code or by using a primitive generator based on signature requests, as discussed in [BZN05]. The final option would be to automatically detect often used expression patterns and dynamically compile, link and use the optimized primitives.

MonetDB/X100 primitives, like MonetDB operators, are based on the principle of *bulk processing*. In one function call, a large set of values is processed, without the need of per-tuple decisions based on data type, input properties etc. This way all the performance benefits of MonetDB discussed in Section 3.4 can be maintained. Additionally, the next section discusses how MonetDB/X100 manages to avoid the in-memory materialization problem that hinders the performance in the original MonetDB model.

The implementation of primitives is a relatively straightforward task for simple operations like projection or selection. However, special care needs to be taken to guarantee their high performance on modern CPUs. The discussion of how this can be achieved even for complex operations is presented in Chapter 5.

4.2.2 In-cache execution

MonetDB/X100 primitives are similar to MonetDB operators in many ways: they use highly specialized code, achieve high CPU efficiency, and consume and produce arrays of values. In MonetDB this last property can lead to the materialization of large intermediate results, which results in a significant performance degradation, as discussed in Section 3.4 – the main memory bandwidth is often simply too low to sustain the data hunger of the CPU-efficient primitives. To avoid this problem, MonetDB/X100 exploits the fact that cache-memory bandwidth is significantly higher than that of RAM, and introduces a principle of *in-cache execution*. The idea is presented in the left-hand side of Figure 4.1: vectors are organized in such a way that their entire memory footprint is small enough to fit in the CPU cache. This allows keeping the materialized results on the CPU die, without the need of expensive writes into main memory.

To demonstrate the importance of keeping the data in the CPU cache, we

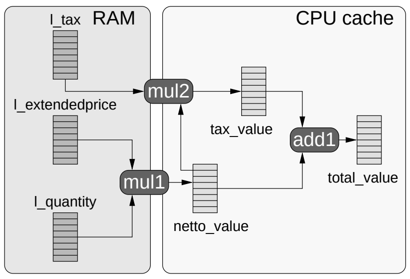


Figure 4.3: Plan of a simple query in MonetDB/X100, demonstrating in-RAM or in-cache data placement

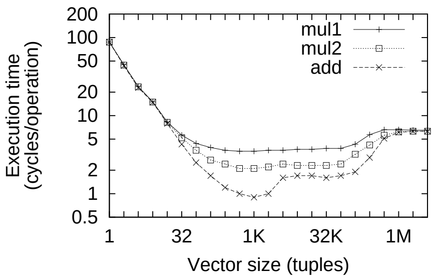


Figure 4.4: Impact of data location and vector size on primitive performance, using a query from Figure 4.3

analyze the performance of a simple X100 query, equivalent to this SQL statement:

```
SELECT l_quantity * l_extendedprice AS netto_value,
       netto_value * l_tax AS tax_value,
       netto_value + tax_value AS total_value
FROM   lineitem
```

The simplified plan for this computation is presented in Figure 4.3. It shows that the base columns are read from main memory (storage manager buffers), while the intermediate results stay in the CPU cache. Figure 4.4 demonstrates the speed of the primitives depending on the used vector size. For small vector sizes, the execution time is dominated by the per-vector logic. On the other hand, for large vector sizes, the vectors do not fit in the CPU cache anymore, causing expensive main-memory traffic.

The performance of the individual primitives depends on the location of their input data. As Figure 4.4 shows, the *mul1()* primitive is significantly slower than *add1()* – the reason is that it needs to read a large amount of data from main memory. In this case, with the optimal vector size, it spends 3.5 CPU cycles to add two 4-byte wide values. On a 2.16GHz machine, this results in a memory bandwidth of ca. 4.8 GB/sec. On the other hand, *add1* only spends 0.9 cycles on the equivalent computation (on this platform multiplication and addition execute at the same speed), because it can quickly access all its input data in the CPU cache. The performance of *mul2* lands in between (2.1 cycles), as only one of its inputs is RAM-resident. This experiment demonstrates that even with sequential access keeping the data in-cache can result in a significant performance improvement and explains the performance benefit of MonetDB/X100 over the MonetDB-style processing.

4.2.2.1 Cache interference

Many cache-conscious data processing techniques implicitly assume that during the operation of a given algorithm it has an exclusive ownership of the cache [MBK02]. However, in real-life scenarios, it is often the case that multiple queries are active at the same time in the system, and the execution context of the CPU changes quite frequently. As a result, the cache content useful for one query can be replaced with other data. This problem has been demonstrated e.g. in [CAGM07, Section 8.2.5] where the performance of the hash-join algorithm based on cache-partitioning has been shown to deteriorate significantly (up to 78%) when the cache content is periodically flushed.

To analyze the impact of cache interference on the performance of MonetDB/X100 in-cache processing, two parts of the execution layer need to be considered: the iterator pipeline and the individual operators. The impact on the operators is algorithm-specific and not directly related to the chosen execution model. For example, the cache-partitioned hash-join will suffer from the cache interference problems in all the execution models: tuple-based, column-based, and vector-based. Therefore, we postpone the discussion of cache-conscious operator implementations to Chapter 5, and for now focus on the execution model itself.

An execution context change in the CPU can occur when an application performs a system call, e.g. by requesting an I/O operation. Also, the kernel can decide to preempt the current process, e.g. when an interrupt happens or the quantum of time assigned to this process is finished. In the scenarios MonetDB/X100 is targeted at, the system is typically performing relatively few and large I/Os, hence the interrupts do not occur often. With the typical kernel scheduling frequency in the range of 100 times per second (10ms), it is reasonable to assume that an executing process has an uninterrupted slice of time in the range of 0.1 to 10 ms.

To evaluate the robustness of the vectorized in-cache execution model proposed in MonetDB/X100, we analyze the performance of the TPC-H Query 1 with enforced cache-flushes happening at different intervals. Query 1 is a good candidate to demonstrate the impact on the iterator pipeline itself, as all the operators except for the aggregation are stateless, and the aggregation uses a very small hash-table, as it only computes 4 output values. As a result, only the

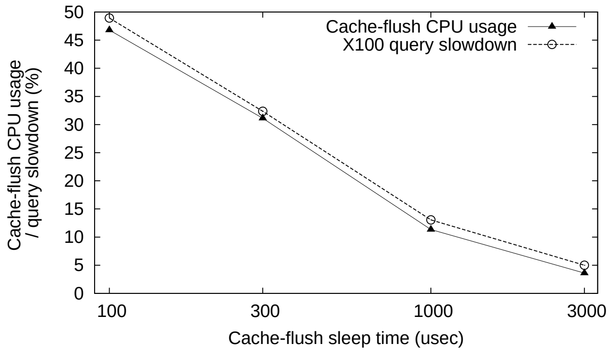


Figure 4.5: Impact of the cache-flushing on the TPC-H Query 1 performance in MonetDB X100 (2.4 GHz Athlon64, 512KB L2 cache)

vectors containing the data passed between the operators use the CPU cache. In the experiment presented in Figure 4.5, a series of queries is running, and, for part of the queries, a separate program is forcing the cache-flush by touching all the cache lines in a separate cache-sized main-memory area.

As the results show, the slowdown of the queries is in line with the CPU time taken by the flushing program, and not significantly higher. This demonstrates that the vectorized in-cache model is relatively robust to cache interference. One of the reasons the impact of the cache flushing is so marginal, compared to the results from [CAGM07], is that for most operations in this query the data accesses are fully sequential. As a result, if the cache-misses start to occur at the beginning of the vector, the hardware-prefetching mechanisms available in most modern CPUs (see Section 2.2.3) will be triggered and will start loading the remaining part of the data. Additionally, even with a 0.1ms time slice, tens or hundreds of primitives can execute, exploiting the exclusive access to the cache during this period.

4.2.2.2 Vector size and allocation

For high execution efficiency, it is important that the vector sizes in the vectorized execution model are on the one hand as large as possible, to minimize the interpretation overhead, and on the other hand not too large, to fit in the CPU cache. Two aspects of this issue need to be analyzed: how many vectors need to be allocated, and the size of each vector.

In the existing MonetDB/X100 implementation, every primitive uses its input vectors and introduces a new result vector, with vector sizes fixed in the entire query plan. The only optimization currently applied is that if one of the primitive input vectors has only one parent, and shares the same datatype with the result vector, it can be *reused*. This is similar to the *register allocation* problem, found in compiler optimization (see e.g. [HKMW66, AC76]). It is possible to see physical vectors as *vector registers* and then map logical vectors used by the primitives to them. Such an optimization can reduce the number of vectors within a given query.

An interesting point in vector allocation optimization is that different vectors can have different datatype widths. As such, for optimal results, the classical register allocation scheme needs to be extended to be able to e.g. reuse a 64-bit wide vector space as two 32-bit vectors. Another possible optimization exploits the fact that query graphs can often be decomposed into a collection of pipelinable sub-graphs (separated by e.g. blocking operators). Then, vector allocation for each sub-graph can be performed separately.

Once the number of vectors for a given query sub-graph is known, we can find the vector size by dividing the available D-cache size by the number of vectors. Typically, not the entire cache can be used, as there is space overhead related to operator state, especially for blocking operators with a potentially large state (e.g. hash aggregation). Another problem is related to the trade-offs between targeting different CPU cache levels. While, as presented earlier, L1 can provide much better performance than L2, it can require significantly smaller vectors. The choice should be based on the complexity of the query: for very simple queries L1 should be targeted, and for queries with more vectors, it should be L2.

4.2.3 Execution layer performance

To evaluate the performance of the proposed execution model, this section demonstrates the performance of TPC-H Query 1 on a 1GB database running on top of MonetDB/X100 with a varying vector size. The results, presented in Figure 4.6, follow the trend already observed in Figure 4.4: the speed improves very quickly with an increasing vector size, until it reaches the optimum where the vectors still fit in the CPU cache and the interpretation overhead is maximally amortized. Then, once the vectors exceed the cache size, the speed deteriorates again. For comparison, the same figure presents the results for two “traditional” tuple-at-a-time systems (MySQL and a commercial one), MonetDB column-at-a-time processing, and hand-written optimized code. Interest-

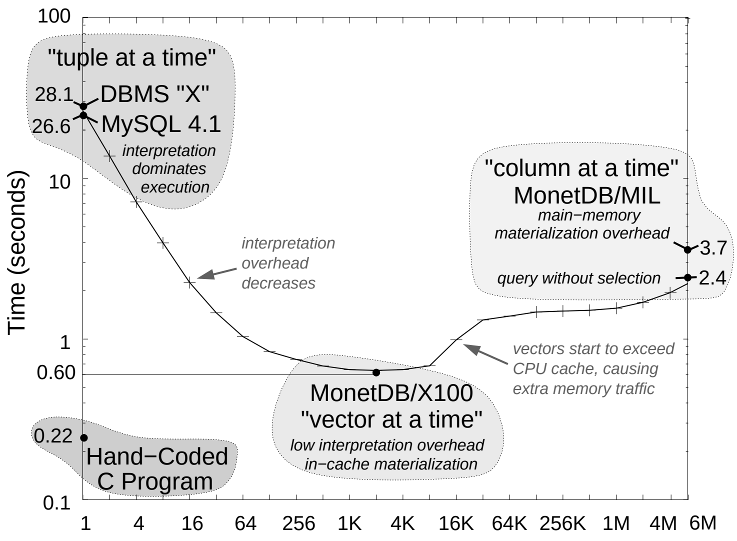


Figure 4.6: TPC-H Query 1 performance in X100 with varying vector size (in tuples, horizontal axis, 2005 results from [BZN05])

ingly, MonetDB/X100 results with vector size of 1 match almost exactly the performance of the tuple-at-a-time systems. Similarly, with very large vector sizes, MonetDB/X100 execution boils down to the column-at-a-time model, but achieves performance almost two times better than MonetDB. That difference is caused by the cost of after-selection projection of tuple attributes in Query 1 on MonetDB, visible in a series of `join` operations in Figure 3.6. In MonetDB/X100 this projection is avoided thanks to the selection vector. For a fair comparison, Figure 4.6 additionally includes the MonetDB results without the selection statement – the computation cost for the extra tuples is negligible, as the predicate selects almost all tuples. MonetDB/X100 performance almost exactly coincides with the performance of this approach.

Table 4.2 presents a detailed analysis of Query 1 performance with MonetDB/X100, showing per-primitive and per-operator properties. Since in Query 1 the aggregation produces only 4 tuples, post-aggregation processing time is negligible and is ignored. The results show that the primitives achieve very good per-tuple performance, spending only a few CPU cycles on each operation, and deliver extremely high data throughput (for primitive bandwidth, we count both

input count	total MB	time (us)	BW MB/s	avg. cycles	**X100 primitive**
6.0M	68.7	7294	9418	3.1	`select_<_date_col_date_val`
5.9M	135.4	11349	11930	4.9	`map_-_slng_val_slng_col`
5.9M	135.4	12112	11178	5.3	`map_*_slng_col_slng_col`
5.9M	135.4	11537	11736	5.0	`map_+_slng_val_slng_col`
5.9M	135.4	9219	14687	4.0	`map_*_slng_col_slng_col`
5.9M	95.9	6138	15623	2.7	`map_uidx_uchr_col`
5.9M	141.1	8508	16584	3.7	`map_directgrp_uidx_col_uchr_col_uint_val`
5.9M	135.4	12025	11259	5.2	`aggr_count_uidx_col`
5.9M	158.0	15741	10037	6.9	`aggr_sum_sint_col_uidx_col`
5.9M	180.6	16588	10887	7.2	`aggr_sum_slng_col_uidx_col`
5.9M	180.6	16513	10936	7.2	`aggr_sum_slng_col_uidx_col`
5.9M	180.6	16462	10970	7.2	`aggr_sum_slng_col_uidx_col`
5.9M	180.6	16575	10875	7.2	`aggr_sum_slng_col_uidx_col`
					X100 operator
0		2363			ColumnScan
6.0M		8244			Select
5.9M		46709			Project
5.9M		112699			Aggr(DIRECT)

Table 4.2: TPC-H Query 1 performance trace with MonetDB/X100 (Core2 2.4GHz, SF=1, vector size 1024 tuples, post-aggregation processing ignored)

input and output data volume). This is possible thanks to performing the execution on cache-resident data.

4.3 Bandwidth-optimized storage

Performance improvements achieved with the vectorized query execution model result in a demand for very high data delivery rates from persistent storage. In memory, MonetDB/X100 can process hundreds of megabytes, or even over a gigabyte, of columnar data on a single CPU core per second, leading to data consumption of 0.1 to 1 bytes per CPU cycle. Providing such data bandwidths, even for a single core system, requires relatively expensive solutions, and scaling for multi-core systems is even more challenging. Additionally, Figures 2.1 and 2.8 demonstrate that CPU performance improves significantly more rapidly than disk bandwidth and latency, suggesting that efficient data delivery will continue

processing			**disks**			**throughput**	
#	CPU	RAM	#	totsize	cost	single	5-way
4	Xeon 3.0GHz dual-core	64GB	124	4.4TB	47%	19497	10404
2	Opteron 2GHz	48GB	336	6.0TB	80%	12941	11531
4	Xeon 3.0GHz dual-core	32GB	92	3.2TB	67%	11423	6768
2	Power5 1.65GHz dual-core	32GB	45	1.6TB	65%	8415	4802

Table 4.3: Official 2006 TPC-H 100GB results

to be a problem in the future. As a result, software approaches attempting to reduce the data demand from the physical storage become increasingly important.

We address the problem of providing high data bandwidth to the execution layer by introducing a new storage layer, called *ColumnBM*. It is optimized to fully exploit the available disk bandwidth, and proposes a number of techniques for reducing data volumes that need to be shipped from disk. This section presents the main design choices of this layer, while Chapters 6 and 7 discuss in more details two major optimization techniques ColumnBM introduces: *light-weight compression* and *cooperative scans*.

4.3.1 Scan-based processing

Database systems are addicted to random disk I/O, caused by non-clustered index lookups, and hardware trends are pushing this model to the limit of sustainability. Foreign-key joins, as well as selection predicates executed using unclustered indices, both may yield large streams of row-IDs for looking up records in a target table. If this target table is large and the accesses are scattered, the needed disk pages will have to be fetched using random disk I/O. To optimize performance, industry-strength RDBMSs make good use of asynchronous I/O to farm out batches of requests over multiple disk drives, both to achieve I/O parallelism between the drives, and to let each disk handle multiple I/Os in a single arm movement (amortizing some access latency).

Massive numbers of unclustered disk accesses occur frequently in benchmarks like TPC-H, and it is not uncommon now to see benchmark configurations that use hundreds or thousands of disks. For example, Table 4.3 shows that four representative TPC-H submissions of even the smallest 100GB data size used an average of 150 disks with total storage capacity of 3.8 terabyte. All these disks are less than 10% full, and the main reason for their high number is to get more disk-arms, allowing for a higher throughput of random I/O requests.

Note from Table 4.3 that a high number of disks seems especially crucial in the concurrent (5 stream) query scenario. The underlying hardware trends of the past decades, namely sustained exponential growth of CPU power as well as a much faster improvement of disk-bandwidth than of I/O latency, are expected to continue. Thus, to keep each next CPU generation with more cores busy, the number of disks will need to be doubled to achieve system balance.

This exponential trend is clearly unsustainable, and one can argue that in the real world (i.e. outside manufacturer benchmarking projects) it is already no longer being sustained, and database servers are often configured with fewer disks than optimal. The main reason for this is cost, both in terms of absolute value of large I/O subsystems (nowadays taking more than two thirds of TPC-H benchmark systems cost, see Table 4.3), but also maintenance costs. In a multi-thousand disk configuration, multiple disks are expected to break each day [SG07], which implies the need for full-time attendance by a human system administrator.

We argue that the only way to avoid efficiency problems caused by random I/O is to rely more on sequential scans of tables or clustered indices, which depend on disk bandwidth rather than latency. Achieving scan-mostly data plans is possible thanks to a number of techniques:

1. storing relations redundantly in *multiple orders* [AMDM07], such that more query patterns can use a clustered access path. To avoid the cost of updating multiple such tables, updates in such systems are buffered in RAM in differential lists and are dynamically merged with persistent data.

2. exploiting *correlated orderings*. In MonetDB/X100, a min- and max-value is kept for each column per large disk block (see Section 4.3.3). Such meta-data, similar to “small materialized aggregates” [Moe98] and also found e.g. in the Netezza system as “zonemaps” [Net], allows avoiding reading unneeded blocks during an (index) scan, even if the data is not ordered on that column, but on a correlated column. For example, in the `lineitem` table in the TPC-H schema it allows avoiding I/O for almost all non-relevant tuples in range selections on any date column in the TPC-H schema, as dates in the fact tables of a data warehouse tend to be highly correlated. This technique can sometimes result in a scan-plan that requires a set of non-contiguous table ranges.

3. using *multi-table clustering* or *materialized views*, to exploit index range-scans even over foreign-key joins.

4. exploiting *large RAMs* to fully buffer small (compressed) relations, e.g. the dimensions of a star schema.

5. reducing scan I/O volume by offering *column storage* (DSM) as an option [ZHNB06, SAB$^+$05] to avoid reading unused columns. The same remark as made in (1) on handling updates applies here.

6. using data *compression*, where the reduced I/O cost due to size reduction outweighs the CPU cost of decompression. It has been shown that with column storage, (de-)compression becomes less CPU intensive and achieves better compression ratios (Chapter 6, [AMF06]).

After applying (1-2), data warehousing queries use (clustered index) scans for their I/O, typically selecting ranges from the fact tables. Other table I/O can be reduced using (3-4) and I/O bandwidth can be optimized to its minimum cost by (5-6).

4.3.2 ColumnBM storage format

ColumnBM stores data in fixed-size *blocks*. To reduce the cost of moving the disk head when performing multiple scan operations at the same time (which happens in DSM even with a single active scan operator), the blocks from the same column are grouped into large *chunks*. The size of a chunk is large enough to amortize the disk latency, as presented in Figure 2.9. When performing a scan, if the system detects multiple blocks from a single chunk are useful for a given query, they are all read in one operation. Similarly, when saving the data, the writing process for a block can be delayed to possibly save multiple blocks in one go.

In the storage layer, most database systems follow either NSM or DSM as their data organization format, as discussed in Section 3.1.1. A choice of the storage model has an impact not only on the I/O performance, but also on the query processing efficiency. If the storage format does not match the internal execution layer format, an extra translation phase is necessary. This reduces the performance, as discussed in Section 3.5.7 for the [HLAM06] results.

The PAX storage format [ADHS01], where data is organized in columns, like in DSM, but all attributes are grouped in a single I/O block, like in NSM, has demonstrated that using different data representations on disk and in memory can be beneficial. ColumnBM applies this idea by keeping in-block data in the columnar format, matching the MonetDB/X100 execution layer. However, it follows and extends the PAX approach by allowing arbitrary vertical fragmentation of a table on disk, similar to the *data-morphing* technique [HP03] for in-memory data. In this approach, a table is decomposed into non-overlapping *groups* of attributes, with each group stored as PAX disk blocks. As Figure 4.7 demonstrates, both DSM and NSM can be expressed as special cases, allowing having a uniform infrastructure for different storage layouts.

The flexible partitioning in ColumnBM allows application-specific data organization. For large scans involving a subset of columns, DSM can be applied. For tasks that read relatively small numbers of tuples and use most of the columns, NSM is a better choice. An example of such an application is inverted-list processing in information retrieval [HZdVB06], where in each query a subset of a three-column table (*term-doc-score*) is scanned for each term. Since many terms have a relatively small number of occurrences, the scan times for these are dominated by the random-access cost. Arbitrary partitioning is applicable if the access pattern of an application is well known – then attributes often accessed together can be grouped on disk.

Having both PAX and DSM facilities also allows applying the *fractured mirrors* [RDS02] storage strategy. In this approach, the data is stored in two copies: one NSM (PAX in MonetDB/X100) and one in DSM, distributed among multiple disks. This allows both efficient scan-based processing, using the DSM copy, and fast random-access lookups, using the NSM/PAX copy. This functionality, useful for query execution, is also very important for the update strategies currently being developed for ColumnBM, as discussed in Section 4.3.4.

4.3.3 Index structures

To allow efficient range lookups on various attributes, databases typically apply non-clustered B-tree indices. However, if a given index does not cover all the query attributes, extra random accesses to the base table are necessary. This makes this approach useful only if a very small percentage of tuples is selected. To increase the applicability of scan-based approaches, we allow extra indices to carry an arbitrary collection of base table columns, allowing satisfying more queries. This is similar to the *projections* proposed in [SAB$^+$05], but ColumnBM always keeps one full copy of a table with all the columns sorted in the same order.

To reduce the volume of scanned data even in the absence of ordering on a range-predicate attribute, ColumnBM provides per-block value-range information, which is a simple case of *small materialized aggregates* [Moe98]. This information is especially useful with *correlated* columns, as is the case e.g. in

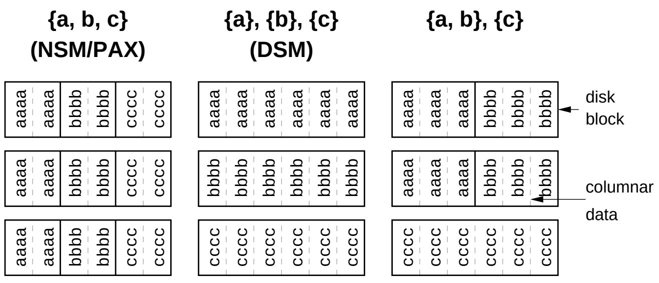


Figure 4.7: ColumnBM data organization, with NSM, DSM and arbitrary vertical fragmentation strategies for a 3-column table

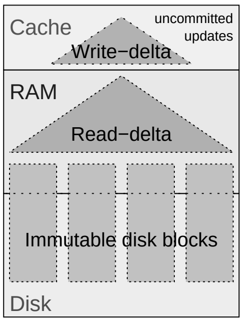


Figure 4.8: Multi-level delta structures for handling updates in ColumnBM

different date attributes in the "lineitem" table of the TPC-H benchmark. Then, if a table is sorted on one date attribute, and a query has a range predicate on a different date column, per-block statistics can be used to filter-out many disk blocks.

Another index structure used in ColumnBM are *join-indices* [Val87], which provide direct access to the matching tuple in the *parent* relation for each tuple in the *child* table in a foreign-key relationship. This can significantly reduce the join cost when the inner relation is a rarely-updated, memory-resident table, as is often the case with the dimension tables in typical *star* or *snowflake* data warehousing schemas.

4.3.4 Updates

ColumnBM provides a novel approach to handling updates which allows combining high performance of scan-based queries with high update throughput. This functionality is the topic of ongoing research [HNZB08], and this section provides only an overview of the design goals and features.

4.3.4.1 Delta-based updates

Most database systems perform updates by directly modifying related disk pages. A stream of such updates results in a high load of random disk accesses in NSM storage, which is even higher in DSM, where multiple columns need to be accessed. Large disk blocks and in-block compression, often used in analytics-oriented systems, make in-place updates even more expensive. To overcome this problem, MonetDB/X100 uses in-memory *deltas*, similar to *differential files* [SL76]. In this approach, updates are buffered in a delta structure, which is transparently merged during query processing. These structures are periodically merged into persistent storage, but the updates are visible even before this process occurs.

Multiple delta structures can be stacked to allow transaction isolation. For example, Figure 4.8 shows a simple two-level approach to implement the *read-committed* isolation level. Here, the active transaction reads previous updates from a global *read-delta*, but writes all changes to its private *write-delta*, keeping them invisible to other users. Finally, when the transaction commits, the write-delta is merged into the read-delta, making the changes visible to other transactions.

4.3.4.2 Positional delta trees

In the previous delta-based approaches [SL76, RDS02], updates in the delta structures are stored in the same order as the underlying table, and the update merging is performed by looking at the values of the key attributes. This process requires all the key attributes of a table to be scanned, even if a query does not need to process all of them, potentially reducing the performance advantage provided by DSM. Additionally, finding a position where the deltas need to be applied in the persistent data stream requires a relatively expensive value-based search, similar to a single phase in *merge-sort*.

An alternative to this *value-based* approach is a newly introduced *Positional Delta Tree* (PDT) structure [HNZB08, Section II-C]. PDTs store updates ordered by their *position* in the table, dynamically adjusting these positions with incoming updates. Having positional access allows the merging process to be performed even if not all key attributes are fetched from the disk, allowing better performance in column stores. Additionally, merging becomes faster since PDTs provide the direct location in the data stream where the updates need to be applied.

PDTs introduce one challenge not present in the value-based approaches. When a tuple is inserted, its exact position needs to be found in the current image of the data, which might consist of both persistent storage and previous updates stored in a PDT. With DSM, if the table key consists of multiple columns, this may require multiple disk accesses for each tuple. To avoid this extra cost, ColumnBM update facilities exploit an optional *fractured mirrors* layout discussed in Section 4.3.2. In this approach, an extra NSM/PAX copy of the table is stored, allowing finding the tuple position with just one I/O access. This optimization is especially important when multiple indices or table replicas are present. In such scenarios, any update to the base table needs to be propagated to all the auxiliary structures. This requires getting all the key attributes of these structures from the base table, which, again, may cause multiple I/Os in the plain DSM storage scheme, and can be avoided using the NSM/PAX copy.

4.4 Conclusions

This chapter presented the overview of the MonetDB/X100 system architecture, focusing on the execution and storage layers. In the execution layer a new *vectorized in-cache execution model* was proposed. By minimizing the interpretation overhead found in most database systems and efficiently exploiting modern CPU features, it provides execution performance often orders of magnitude higher than existing solutions. This model is further investigated in Chapter 5.

The high performance of the execution layer requires new approaches in the storage layer that would scale this performance to disk-resident datasets. The proposed ColumnBM system achieves this by introducing a number of strategies for analytical queries: vertical partitioning, scan-optimized storage and index structures, as well as efficient updates. Two major optimizations in this component, *lightweight compression* and *cooperative scans* are discussed in more detail in Chapters 6 and 7.

Chapter 5

Vectorized execution model

This chapter discusses in detail the vectorized execution model introduced in Chapter 4. First, Section 5.1 analyzes the properties of this model, comparing it to the previously proposed tuple-at-a-time and column-at-a-time models. Later, Section 5.2 discusses the implementation of data processing operations in this model, first identifying the requirements of efficient implementations, and then providing a set of implementation techniques. Additionally, Section 5.2.2 discusses different possible choices of data organization during processing. All these techniques are synthesized in the description of an example implementation of one of the crucial database operators: a hash join. A simple vectorized implementation is initially presented in Section 5.3, and Section 5.4 discusses a set of techniques improving its performance. Finally, to complete the chapter, Section 5.5 provides a set of vectorized implementations of other interesting processing tasks.

5.1 Properties of the vectorized execution model

The goal of researching a new execution model was to overcome the problems found in the previously proposed tuple-at-a-time and column-at-a-time models. This section analyzes the properties of the new model and compares them with the existing solutions.

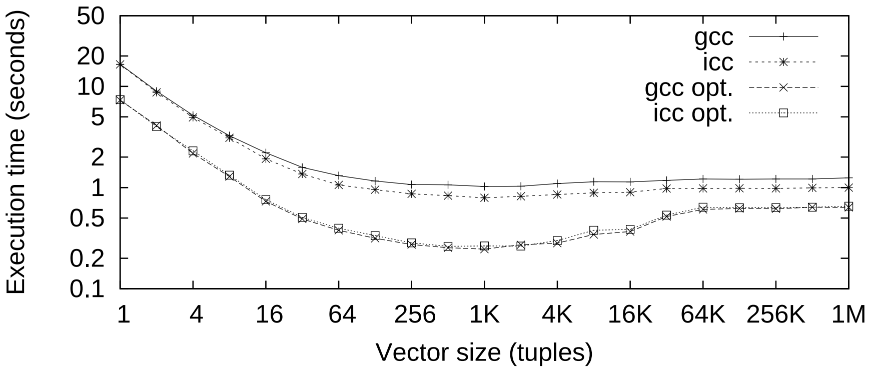


Figure 5.1: TPC-H Query 1 benchmark on MonetDB/X100 using different compilers, optimization options and vector sizes (Athlon64)

5.1.1 Interpretation overhead

In the traditional Volcano model, the data is processed in a 'pull' fashion, where the consuming operators ask their children for the next tuple. As a result, at least one `next()` call is performed for every tuple in every operator. Also within a single relational operator, multiple functions are called. For example, a Project operator that computes a sum of two columns needs to call an addition primitive for each tuple it processes. Note that these calls cannot be inlined by the compiler, as they are query-specific, hence the cost of passing the parameters and modifying the program counter is always present. Also, the actual addresses of the functions to call need to be read from memory, making it hard for a CPU to speculate ahead of the call. Finally, complex operator logic is performed for every tuple, causing the interpretation overhead to dominate the overall execution time.

With vectorized processing, in both scenarios the function call can be amortized over a large set of tuples. Figure 5.1 presents the results for TPC-H query 1 (scale factor 1) executed on MonetDB/X100 running on a 2-GHz Athlon64. The first observation is that using the optimal vector size can give a performance improvement of as much as 30 times. The second observation is on the influence of optimization settings in the compiler: with optimization, both using the gcc and icc compilers, performance gains are much bigger for larger vector sizes. This is because the actual data processing code can be efficiently optimized by the compiler, unlike the branch-heavy control logic found in operators. This aspect is further analyzed in Section 5.2.1.3.

Comparing to the column-at-a-time model used in MonetDB, the vectorized model can result in a slightly higher overhead, as the interpretation occurs for every vector. Still, with vector size in range of hundreds of tuples, this overhead is so small that its impact on the overall performance is negligible, as seen with an almost flat line in Figure 5.1. This is confirmed by the left-most side of the Figure 5.2, which shows that for large vector sizes the number of the CPU instructions stays virtually constant. Note that MonetDB suffers from main-memory materialization overhead, which degrades its performance, as discussed in section 3.4.

5.1.2 Instruction cache

The impact of instruction-cache misses on the performance of the tuple-at-a-time model has been identified for both OLTP query loads [ADHW99, HA04, HSA05], as well for OLAP-like queries [ADHW99, ZR04]. They can constitute even 40% of the entire query execution time [HA04]. Techniques to reduce this overhead include grouping different queries performing the same operation [HA04, HSA05] and buffering tuples within a single query [ZR04]. The latter technique is slightly similar to vectorized processing, since it passes multiple tuples between the operators. However, it causes an additional data copying cost, and the data processing is still performed in a tuple-at-a-time fashion.

To demonstrate the impact of the instruction cache on vectorized processing, we analyze the performance of three queries: TPC-H $Q1$, and two variants of it, $Q1'$ and $Q1''$ that use roughly 2- and 3- times more different primitives, increasing the amount of used instruction memory. In Figure 5.2, for all three queries we provide both the total number of executed instructions, as well as the number of L1 instruction cache misses. As the results show, for $Q1$ instruction misses are negligible. For $Q1'$, the number grows somewhat, but still is relatively low. With $Q1''$, the size of separate code paths finally exceeds the size of the instruction cache, and we see that although the code size increased over 3 times over $Q1$, the number of misses can be as much as 1000 times higher, even though the total number of instructions grew only two-fold. Luckily, even for this complex query, the number of instruction-misses decreases linearly with a growing vector size, and the instruction-cache-miss overhead can be alleviated.

Similarly to the interpretation overhead, the overhead of the instruction misses in the vectorized model can be slightly higher than in MonetDB. Still, since it is amortized among multiple tuples in a vector, it is typically negligible.

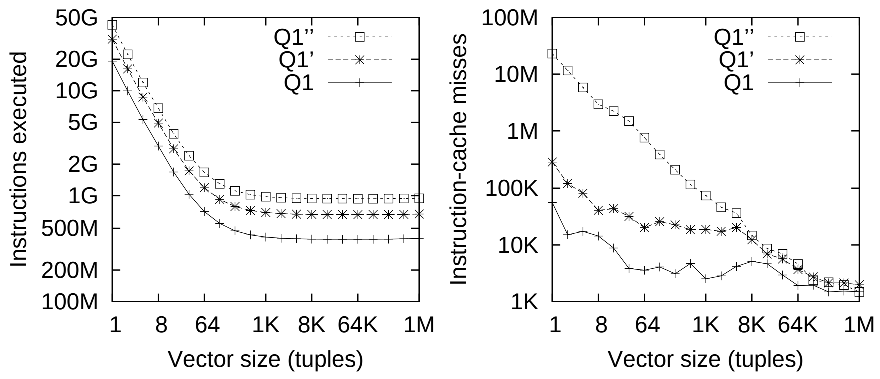


Figure 5.2: Impact of the vector size on the number of instructions and L1 instruction-cache misses (Athlon64)

5.1.3 Processing unit size

Comparing to the tuple-at-a-time and column-at-a-time models, the vectorized model provides a granularity of operation that falls between these two extremes. As a result, there are situation in which some logic that is usually executed for every tuple, can be executed on a per-vector base. A simple example is data partitioning, when the result partition sizes are not known in advance. The code for dividing a vector of N tuples into P partitions using the hash values could be as follows:

```
for (i = 0; i < N; i++) {
    group = hash_values[i] % P;
    *(part[group]++) = values[i];
    if (part[group] == part_end[group])
        overflow(group);
}
```

Note that the overflow check is necessary for each tuple if we do not know the partition sizes in advance. While this check is usually false, we can still remove it from the loop, by exploiting the fact that in most cases the buffers for the destination groups are much larger than the size of the vector. As a result, we can check if every group buffer still contains enough tuples before processing each vector.

```
for (i = 0; i < P; i++)
    if (part[i] >= part_sentinel[i])
```

```
            overflow(i);
    for (i = 0; i < n; i++) {
        group = hash_values[i] % P;
        *(part[group]++) = values[i];
    }
```

In this situation, we check for a buffer overflow not N times, but P times. It is also possible to perform such check every few vectors, to further reduce its cost. This solution requires some extra 'sentinel' space left in the buffer, but this waste should be marginal (e.g. 1024 elements out of 128 thousands). We compared both solutions implemented using optimization techniques described in Section 5.2.4, and the second version gave a ca.15% improvement of the partitioning step (using 64 partitions and 1024-tuple vectors). Note that this optimization cannot be applied by the compiler automatically, since it requires modifications to the underlying data organization.

Another case where vectors can be a useful extra computational unit are exception situations. An example is handling of an arithmetic overflow. Typically, an overflow is checked for each performed operation. However, on some architectures, it is possible to check if an overflow occurred over a large set of computations (e.g. by using *summary overflow* bit in PowerPC CPUs [PMAJ01]). A different vectorized solution to overflow checking is proposed in Section 5.5.1.

A natural intermediate processing unit can also be helpful for data routing in a query plan. For example, the `exchange` operator [Gra90] can distribute tuples for parallel processing using vectors. Also, in dynamic query optimization, for example in Eddies [AH00], adapting the plan every vector, and not every tuple, is beneficial.

Removing logic from the per-tuple loop has an additional benefit – the resulting code is typically simpler, allowing better optimizations by a compiler and more efficient execution on a CPU.

5.1.4 Code efficiency

In the vectorized model, operator functionality is decomposed into small processing units that we call primitives. As hinted before, thanks to their high specialization they provide code that is easy to optimize for the compiler and efficiently executed on modern CPUs. As an example, let us analyze the following simple routine that adds two vectors of integers:

```
void map_add_int_vec_int_vec(int *result, int *input1, int *input2, int n) {
    for (int i = 0; i < n; i++)
```

```
        result[i] = input1[i] + input2[i];
}
```

We can identify the following properties of this routine: it does not contain any control dependencies, hence does not suffer from branch prediction misses; it does not contain any data dependencies, hence there are no stalls in the processing pipeline; a simple loop allows easy unrolling, reducing the loop overhead; data access is direct, there is no overhead in attribute extraction; data access is fully sequential, hence does not suffer from random cache misses and hardware prefetching can be applied; performed operations are simple and allow easy SIMDization. The last property already provides a 2x-8x speedup for various operations on many common data types, and with growing widths of SIMD units (e.g. 256-bits in Intel AVX [Int08]) this performance benefit of this technique will increase.

The described routine is a perfect example of how efficient vectorization can be – on a Core2Duo machine it spends only 0.92 cycles per single iteration. Comparing to an interpreted tuple-at-a-time approach, the performance benefit can be even two orders of magnitude.

While providing primitives having all described properties for all types of operations found in databases is probably impossible, efficient solutions can be developed even for complex problems. In Section 5.2.4 we will discuss a set of techniques helpful in the implementation of such routines.

5.1.5 Block algorithms

Processing multiple tuples does not only allow efficiently compiled and executed code. It also enables applying algorithms that require a set of tuples to work. For example, in software data prefetching, two major approaches are used: pipelined prefetching and group prefetching [CAGM04]. In the tuple-at-a-time model, they both require tuple buffering, while being directly applicable in the vectorized model. On the other hand, in the column-at-a-time model, the effective block size (full column) is typically too large to exploit the benefit of the prefetching – the data prefetched at the beginning of the column will be most likely evicted at the end.

Another technique that requires multiple tuples is efficient computation of selection predicates [Ros02]. With a block of tuples for which the same predicate needs to be evaluated, different approaches (*binary AND*, *logical AND* or *no-branch*) are optimal. The choice of the used method can be performed using a cost model [Ros02] at query compilation time, but also dynamically during the query execution – information about selectivity in the previous vector is typically a good indicator for the current one.

Finally, having an opportunity to work with multiple tuples allows various programming tricks. For example, during processing data after selection, where selection result is a Boolean bitmap, we can exploit the knowledge of high predicate selectivity, similarly as in [ZR02]. Then the bitmap consists mostly of zeros, and we can check multiple bits in one go, speculating that they are not set, and handle the non-zero cases in an extra step. Similar tricks can be used to detect a zero in the vector that is a parameter for the division operation, handle NULL values in mostly non-NULL data, and more.

5.1.6 Scalability

In the column-at-a-time model, every operator fully materializes its result, making it ineffective for queries requiring disk-based intermediate results. The Volcano model, thanks to its pipelined nature, can process datasets larger than available memory, using on-disk materialization only for blocking operators (sort, hash-join). This property is directly inherited by the vectorized model.

A newly introduced aspect of the vectorized model is its scalability with respect to the complexity of the query and the cache size. With complex query plans that internally keep a large number of vectors, the vector size needs to be reduced to fit the data in the CPU cache, diminishing the benefits of the reduced interpretation overhead. As discussed in Section 4.2.2.2, depending on the query complexity, the vector size should be chosen such that all data fits in either L1 or L2 cache. Since the L2 caches of modern CPUs are in order of megabytes, vectors can be sufficiently large to remove the interpretation overhead (hundreds of tuples) and still fit in the cache even for queries with hundreds of vectors.

5.1.7 Query plan complexity and optimization

The query plans for the vectorized model usually match the plans of the tuple-at-a-time model: they are trees of N-ary operators working in a pipelined fashion. As a result, the vectorized model can benefit from decades of research on the traditional query plans optimization.

In the column-at-a-time model, query plans are significantly more complex, mostly due to the used binary algebra – for the same task, multiple per-column operations need to be performed. However, due to the materializing nature of this model, the query plans are closer to an imperative programming language, and many optimizations from that area are additionally applicable. For example, common subexpression elimination is straightforward here, while being potentially non-trivial in the pipelined model [DSRS01].

5.1.8 Implementation complexity

Implementation of relational operators in the tuple-at-a-time model has been studied over the last 3 decades and is well understood. Still, typical solutions provide code that needs to be very generic, making the implementation often highly complex. In the column-at-a-time model, every operator both consumes and produces simple data arrays of known data types. This makes the implementation of most operators relatively straightforward.

The vector-at-a-time model brings a new challenge of decomposing the processing into independent vectorized primitives. While for some operators (e.g. projections) it is easy, for some it is significantly more challenging, as discussed later in this chapter. An interesting option in this model is that it is possible to emulate both tuple- and column-at-a-time models internally in the operators, allowing easy system prototyping.

5.1.9 Profiling and performance optimization

In the tuple-at-a-time model, the processing thread continuously switches between all operators inside the active plan segment, performing both control logic and the actual data processing. As a result, profiling the execution of individual processing steps is relatively complex: putting a time (or e.g. hardware event) counter for every step inside the operator is often too expensive, and sampling process activity or simulations can be imprecise. Even when the performance bottleneck is localized, improving the performance of that part is often hard, as the involved code is typically large and complex.

In the column-at-a-time model, profiling is straightforward – each operator is fully independent and hence it is trivial to measure its cost. This allows easy detection of bottleneck operators. Still, within the operator it is unclear how much time different operations take. For example, in a hash-join operator, the operator-level profiling does not provide information on the cost of the build- and probe-phase separately.

The vectorized model lands, again, in between. Since the profiling overhead is amortized among multiple tuples, it is possible to precisely measure the performance of every operator and every primitive. As a result, it is easy to spot fine-grain performance bottlenecks. Additionally, once a bottleneck is located,

	Tuple	**Column**	**Vector**
query plans	**simple**	complex	**simple**
instruction cache utilization	poor	**extremely good**	**very good**
plan-data cache utilization	poor	**extremely good**	**very good**
function calls	many	**extremely few**	**very few**
attribute access	complex	**direct**	**direct**
time mostly spent on ...	interpretation	**processing**	**processing**
CPU utilization	poor	**good**	**very good**
compiler optimizations	limited	**applicable**	**applicable**
implementation	medium	**easy**	medium
profiling and optimization	hard	**medium**	**simple**
materialization overhead	**very cheap**	expensive	**cheap**
scalability	**good**	limited	**good**
volume of accessed data	large	**small**	**small**

Table 5.1: Comparison of the N-ary tuple-at-a-time (Tuple), MonetDB column-at-a-time (Column) and vectorized in-cache (Vector) execution models

the code involved is small (typically a single primitive, often only a few lines), making it relatively easy to optimize. Finally, some dynamic optimizations, like choosing one of the possible implementations of the same primitive, are easy in this model.

5.1.10 Comparison summary

To sum up the comparison of the execution models, Table 5.1 shows their properties in all discussed areas[1]. Clearly, the vectorized model combines the best properties of the previous approaches. Still, the question remains, how to actually implement a full system based on the principles of this model. The following sections try to address this issue.

[1]Note that for very complex query plans, in the vectorized model either the vector size shrinks and the model starts to suffer from some of the "Tuple" problems, or the vector size exceeds the cache capacity, causing some of the "Column" inefficiencies

5.2 Implementing the vectorized model

5.2.1 Efficient implementation requirements

Since the major part of time in the vectorized execution model tends to be spent in the data processing primitives, it is important to provide efficient implementation of these. For optimal performance, the vectorized primitives need to meet a set of requirements described in this section. While not every data processing operation can have all the described features, the following sections introduce a set of optimization techniques that make it possible for most operations to get many of these benefits.

5.2.1.1 Bulk processing

To achieve the computational efficiency described in Section 5.1.4, data processing primitives should follow the idea of *bulk processing* – performing the same operation for multiple tuples independently. To achieve this, the primitives need to meet some criteria that can be seen as task independence at various levels of processing:

primitive independence - the first step is to make the primitives process multiple data items in one function call, without the need to communicate with other primitives.

operation independence - if processing of one tuple is independent from other ones, the same computation can be in parallel executed for multiple tuples. This has benefits for super-scalar execution on modern CPUs, and provides SIMD opportunities.

CPU instruction independence - when processing a given tuple, it is important that separate CPU instructions performing the operation are independent. Otherwise, it is possible that execution hazards described in Section 2.1.5 cause“pipeline bubbles”, damaging the performance.

5.2.1.2 Data location and organization

The location of data that a primitive processes can have a significant impact on the execution performance, as demonstrated in Section 4.2.2. Even with fully sequential access, reading and writing data to main memory is significantly more expensive than performing the operation in the CPU cache. Therefore, it is crucial to minimize RAM accesses and focus on in-cache execution.

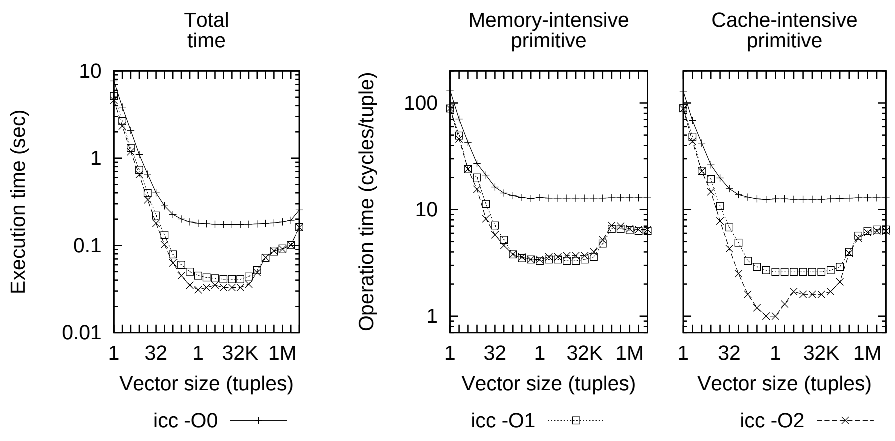


Figure 5.3: Impact of compiler optimizations on execution

Another issue is data organization. While MonetDB/X100 uses column-based structures for passing the data between operators, for some tasks row-based organizations are beneficial, as presented in Section 5.2.2. As a result, for different cases, varying in the type of operation, but also in data properties, different data organizations should be used for optimal performance.

5.2.1.3 Compiler optimization amenability

Another important factor for a high-performance primitive is its amenability to compiler optimizations. As mentioned in Section 5.1.1, the computation-intensive primitives in MonetDB/X100 result in a larger higher benefit of compiler optimizations higher than in the interpretation-intensive code found in the traditional database engines.

For better analysis of this issue, Figure 5.3 demonstrates the performance of the query from Section 4.2.2 dissected into total time (left), *mul1* performance (middle) and *add1* performance (right). Three *icc* compiler optimization levels have been used: `-O0`, with optimizations disabled; `-O1`, with basic optimizations; `-O2`, with more optimizations, including exploiting SIMD instructions. First observation is that compiler optimization adds very little performance for small vector sizes – even with `-O2` the benefit is less than 50% improvement. The reason for this is that in this situation the execution time is dominated by function calls, which are hard to optimize, as they cannot be inlined. For larger vector sizes, the time spent in data-intensive primitives is relatively longer, and, since these primitives are more amenable to the compiler optimizations, the optimization impact increases.

Detailed analysis of the per-primitive optimizations effect shows that for memory-intensive *mul1* primitive the use of SIMD instructions does not improve performance. This is caused by this primitive being memory-bound. On the other hand, for the cache-intensive *add1* primitive, SIMD instructions provide a significant performance improvement, especially visible when the data stays in the L1 cache – the per-tuple cost can be even below a single CPU cycle. As a result, with optimal vector sizes, the properly compiled code can be over 10 times faster. However, the code needs to provide a relatively simple access pattern to allow such level of improvement.

5.2.1.4 Conclusion

The results in this section show that single improvements provide only a limited benefit. With small vector sizes, the benefits of bulk-processing are minimal, also reducing the impact of compiler optimization. In-cache data placement does not result in an improvement if the executing code is unoptimized, as the data-access cost is not the dominating factor anymore. And finally, the benefit of the bulk processing is significantly smaller for the non-cached, unoptimized code. As a result, a combination of all discussed properties is required for highly efficient code.

5.2.2 Choosing the data organization models

As discussed in Section 4.2.1.1, MonetDB/X100 uses single-dimensional vectors for data exchange between the operators. This section demonstrates that this layout is beneficial for sequential data access, which is an approach typically used by operators to consume and produce (but not necessarily process) data. It also discusses a number of other scenarios, where, depending on the operation and data location, either DSM or NSM can be beneficial. Finally, we outline the possibility of combining both models during the execution of a single query for an additional performance improvement. For more details on the issues described in this section, the reader is referred to [ZNB08].

5.2.2.1 Block-data representation models

When discussing the performance between NSM and DSM, it is important to define the used implementation of both models. The internal structure of systems following the same general model can vary significantly, by using different approaches to variable-width datatype storage, NULLs, compression etc. Following the block-oriented processing model of MonetDB/X100, we focus on the representation of entire blocks of tuples.

DSM representation. Traditionally, the Decomposed Storage Model [CK85] proposed for each attribute column to hold two columns: a *surrogate* (or *object-id*) column and a *value* column. Modern column-based systems [BZN05, SAB$^+$05] choose to avoid the former column, and use the natural order for the tuple reorganization purposes. As a result, the table representation is a set of simple value arrays, each containing consecutive values from a different attribute. This format is sometimes complicated e.g. by not storing NULL values and other forms of data compression [ZHNB06, AMF06], but we assume that on the query execution level data is normalized into a contiguous sequence of values. This results in the following simple code to access a specific value in a block:

```
    value = attribute[position];
```

NSM representation. The exact tuple format in NSM can be highly complex, mostly due to storage considerations. For example, NULL values can be materialized or not, variable-width fields result in non-fixed attribute offsets, values can be stored explicitly or as references (e.g. dictionary compression or values from a hash table in a join result). Even fixed-width attributes can be stored using variable-width encoding, e.g. length encoding [WKHM00] or Microsoft's Vardecimal Storage Format [AD07].

Most of the described techniques have a goal of reducing the size of a tuple, which is crucial for disk-based data storage. Unfortunately, in many cases, such tuples are carried through into the query executor, making the data access and manipulation complex and hence expensive. In traditional tuple-at-a-time processing, the cost of accessing a value can be acceptable compared to other overheads, but with block processing, handling complex tuple representations can consume the majority of time.

To analyze the potential of NSM performance, we define a simple structure for holding NSM data, which results in a very fast access to NSM attributes. Tuples in a block are stored contiguously one after another. As a result, tuple offset in a block is a result of the multiplication of the tuple width and its index. Attributes are stored ordered by their widths (wider first). Assuming attributes with widths of power of 2, this makes every value naturally aligned to its datatype within the tuple. Additionally, the tuple is aligned at the end to make its width a multiple of the widest stored attribute. This allows accessing a value of a given attribute at a given position in the table with this simple code:

```
    value = attribute[position * attributeMultiplier];
```

Direct vs. Indirect Storage. Variable-width datatypes such as strings cannot be stored directly in arrays. A solution is to represent them as memory pointers into a heap. In MonetDB/X100, a tuple stream containing string values uses a list of heap buffers that contain concatenated, zero-separated strings. As soon as the last string in a buffer has left the query processing pipeline, the buffer can be reused.

Indirect storage can also be used to reduce value copying between the operators in a pipeline. For instance, in MonetDB/X100, the Select operator leaves all tuple-blocks from the data source operator intact, but just attaches an array of selected offsets, called the *selection vector*. All primitive functions support this optional index array:

```
     value = attribute[selection[position]];
```

Other copy-reduction mechanisms are possible. For example, MonetDB/X100 avoids copying result vectors altogether if an operator is known to leave them unchanged (i.e. columns that just pass through a Project or the left side of an N-1 Join).

Note that the use of index arrays (selection vectors) is not limited to the Select operator. Other possibilities include e.g. not copying the build-relation values in a HashJoin, but instead storing references to them. In principle, each column could have a different (or no) selection vector. This brings multiple optimization opportunities and challenges. For example, a single primitive can be implemented assuming fully independent selection vectors, or provide optimized code for cases where some of the selection vectors are shared. This might provide extra performance, but can significantly increase code size and complexity. For this reason, these optimizations are not yet exploited in MonetDB/X100: all columns in a dataflow share the same selection vector information.

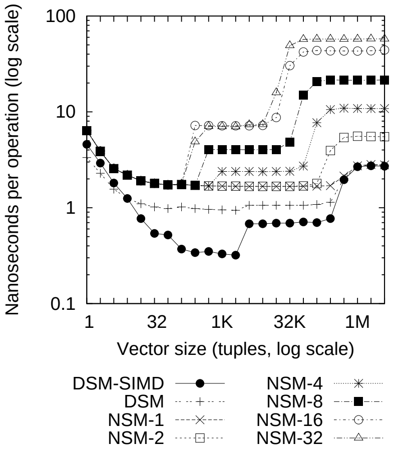


Figure 5.4: Sequential access benchmark: an ADD routine using DSM and NSM with varying tuple widths

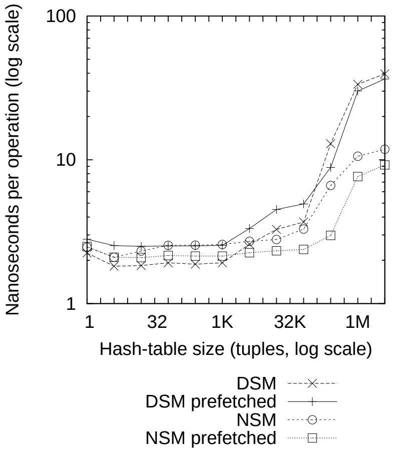


Figure 5.5: Random access benchmark: 4 aggregations using a varying number of aggregation groups

5.2.2.2 NSM and DSM in-memory performance

This section demonstrates how the choice of storage model influences the performance of a given operation. The experimental platform used in the microbenchmarks is a Core2 Quad Q6600 2.4GHz CPU with 8GB RAM running on Linux with kernel 2.6.23-15. The per-core cache sizes are: 16KB L1 I-cache, 16KB L1 D-cache and 4MB L2 cache (shared among 2 cores).

Sequential data access. Figure 5.4 present the results of the experiment in which a SUM aggregate of a 4-byte integer column is computed repeatedly in a loop over a fixed dataset. The size of the data differs, to simulate different block sizes, which allows identifying the impact of the interpretation overhead, as well as the location (cache, memory) in block-oriented processing. We used GCC, using standard (SISD) processing, and additionally ICC to generate SIMD-ized DSM code (NSM does not benefit from SIMD-ization since the values to operate on are not adjacent). In the NSM implementation, we use tuples consisting of a varying number of integers, represented with *NSM-x*.

To analyze the impact of the data organization on CPU efficiency, we look at the performance of *NSM-1*, which has exactly the same memory access pattern and requirements as the DSM implementations. The results show that DSM, thanks to a simpler access code, can provide a significant performance benefit, especially in the SIMD case.

The other aspect of this benchmark is the impact of the interpretation overhead and data location. While for small block sizes the performance is dominated by the function calls[2], for larger sizes, when the data does not fit in the L1 cache anymore, the data location aspect becomes crucial.

Looking at the performance of wider NSM tuples, we see that the performance degrades with increasing tuple width. As long as the tuples are in L1, the performance of all widths is roughly equal. However, for NSM-16 and higher (64 byte tuples or longer) once the data shifts to L2, the impact is immediately visible. This is caused by the fact that only a single integer from the entire cache-line is used. For NSM-2 to NSM-8, the results show that the execution is limited by the L2 bandwidth: when a small fraction of a cache-line is used (e.g. NSM-8) the performance is worse than when more integers are touched (e.g. NSM-2). Similar behavior can be observed for the main-memory datasets.

We see that if access is purely sequential, DSM outperforms NSM for multiple reasons. First, the array-based structure allows simple value-access code. Second, individual primitive functions (e.g. SUM, ADD) use cache lines fully in DSM, and L2 bandwidth is enough to keep up. As mentioned before, during query processing, all tuple blocks used in a query plan should fit the CPU cache. If the target for this is L2, this means significantly larger block sizes than if it were L1, resulting in a reduced function call overhead. Finally, the difference in sequential processing between DSM and NSM can be huge if the operation is expressible in SIMD, especially when the blocks fit in L1, and is still significant when in L2.

Random data access. Figure 5.5 demonstrates an experiment investigating the random-access performance. An input table consists of a single *key* column and 4 *data* columns, contains 4M tuples, and is stored in DSM for efficient sequential access. The range of the *key* column differs from 1 to 4M. We perform an experiment equivalent to this SQL query:

```
SELECT SUM(data1), ..., SUM(dataN)
FROM TABLE GROUP BY key;
```

[2]In a real DBMS the overhead of function calls and other interpretation is significantly larger [BZN05] – this was a hard-coded micro-benchmark.

To store the aggregate results, we use a simple array with the *key* column as a direct index into it. In DSM, the result table is just a collection of arrays, one for each *data* attribute. In NSM, it is a single array of a size equal to the number of tuples multiplied by 4 (the number of *data* attributes). In each iteration, all values from different *data* attributes are added to the respective aggregates, stored at the same index in the table.

The faster access code of the DSM version makes it slightly (up to 10%) faster than NSM as long as the aggregate table fits in the L1 cache. However, once the data expands into L2 or main-memory, the performance of DSM becomes significantly worse than that of NSM. This is caused by the fact that in DSM every memory access is expected to cause a cache-miss. In contrast, in NSM, it can be expected that a cache-line accessed during processing of one *data* column, will be accessed again with the next *data* column in the same block, as all the columns use the same key position.

Figure 5.5 also shows experiments that use software prefetching: we interspersed SUM computations with explicit prefetch instructions on the next tuple block. The end result is that prefetching does improve NSM performance when the aggregate table exceeds the CPU caches, however in contrast to [CAGM07] we could not obtain a straight performance line (i.e. hide all memory latency). In general, our experience with software prefetching indicates that it is hard to use, machine-dependent, and difficult to tune, which makes it hard to apply it in generic database code.

5.2.2.3 Choosing the data model

The results from the previous section suggest that DSM should be used for all sequentially-accessed data as well as for randomly-accessed data that fits in the L1 cache, and NSM should be used for randomly-accessed data that does not fit in L1. Other model-specific optimizations might influence the choice of the used date layout. For example, in [HNZB07, ZNB08] the authors demonstrate an NSM-based technique that uses SIMD instructions to perform aggregation of values from *different* columns at the same time. Row-storage has also been exploited in [JRSS08] to compute multiple predicates on different columns in parallel. These optimizations demonstrate that the choice of a particular data layout while enabling some optimizations, might make other ones impossible. This problem can be partially reduced by on-the-fly format conversion, implemented either as a side-effect of performing some operation (e.g. a SUM routine reading NSM and producing DSM), or as an explicit phase [ZNB08]. Still, this approach increases the complexity of the query plan significantly and incorporating it inside an operator pipeline is an interesting challenge.

In MonetDB/X100 DSM is currently used as the only data exchange format between the operators. This is motivated by the observation that operators typically consume and produce their outputs in a sequential manner. Internally, the operators have a flexibility to choose a storage model most fitting the needs of a used algorithm. Currently, it is typically DSM, but it is expected that the future versions of e.g. HashJoin operator will be able to work with both NSM- and DSM-based data structures.

5.2.3 Decomposing data processing

The core of the vectorized system architecture is the separation of the control logic performed by the operators and the raw data processing performed in primitives. As a result, a methodology to convert a traditional algorithm implementation into a vectorized form is necessary. This problem is close to query compilation for the binary algebra of MonetDB [BK99], but it is different in the following aspects: since it needs to be adapted to the pipelined model, it goes even deeper in the operator decomposition, and additionally needs to handle the N-ary nature of the operators. As a result, *expressing* complex relational operators in a vectorized model is a challenge in itself.

5.2.3.1 Multiple-attribute processing

One of the main benefits of vectorized processing is the high efficiency of the primitives. To achieve this efficiency, however, the primitives are allowed very little (or no) degree of freedom - a single routine can only perform one specific task on a defined set of input types. As a result, usually a primitive is applied to perform a given function on just one or two attributes. This is enough in many cases, e.g. in the `Project` operator, which only adds new columns without the need for manipulating existing ones. However, in many operators, e.g. in aggregation and joins, multiple attributes need to be handled.

A typical approach to this problem is to separate the processing into two phases: one that computes some form of an index structure that is common for all attributes, and the second that uses this structure to perform some computation per attribute. For example, in hash-aggregation [ZHB06], first a position in the hash-table for each tuple is computed using all aggregate keys, and then each aggregate function is computed using it. A similar approach can be used in other operators: in hash-join and merge-join two aligned index-vectors are created, defining matching pairs of tuples in the input relations; in radix-sort the bucket id is computed for each tuple and used to copy non-key attributes; etc.

5.2.3.2 Phase separation

Tuple-at-a-time implementations of most operators contain control logic that is hard or impossible to embed in a single efficient primitive. An example for aggregation using cuckoo-hashing has been presented in [ZHB06]. Here we will present a vectorization process for a different operator: Top-N. Let us take a look at the pseudocode for each tuple in a heap-based Top-N implementation:

```
if (tuple.key > heap.minimum) {
    position = heap.insert(tuple.key);
    heap.values.copy(tuple, position);
}
```

Here, `heap` has a separate `values` section that contains tuple attributes not taking part in the actual heap processing. This code can be decomposed into two separate vectorized parts:

```
selected = select_bigger(input[key], heap.minimum);
heap.process(selected)
```

The initial selection can easily be vectorized for multi-attribute keys. This approach can result in false-positives – tuples that will not enter the heap because a tuple earlier in the same vector increased the heap minimum. Still, in most cases, a large majority of tuples is filtered out with a highly efficient `select_bigger` function, making the cost of an additional check in the later phase negligible. The next step is to decompose `heap.process` into separate primitives (ignoring false-positives for simplicity):

```
positions = heap.insert(input[key], selected);
foreach attribute
    heap.values[attribute].copy(input[attribute], positions);
```

Here, `heap.insert` for each input tuple returns its position in the value area (freed by the expelled tuple), and the `copy()` routine copies all values for a given attribute into their positions in the `values` section.

5.2.3.3 Branch separation

The next issue in operator decomposition is handling situations where different processing steps are taken for each tuple. An example of such a situation is hash-aggregation using a bucket-chained hash-table. Code for each tuple looks as follows:

```
key = tuple.values[KEY];
hash = computeHash(key);
group = hash % num_groups;
idx = buckets[group];
while (idx != NULL) {
    if (key_values[idx] == key)
        break;
    idx = next[idx];
}
if (idx == NULL)
      idx = insert(group, key);
foreach aggregate
    compute_aggr(aggregate, idx, tuple);
```

The presented code is branch-intensive, making it hard for bulk processing. This problem has been identified in the context of software memory prefetching for hash-table processing, where authors annotate each tuple with a special state-identifier, and later combine stages at the same positions in different code-paths into a single stage, using tests on the tuple states to determine the actual code to execute [CAGM04]. A related technique allowing handling this issue is separating the input tuples that are at the same stage of processing into groups. Such a technique can be applied to our aggregation code, resulting in the following vectorized version:

```
keys = input.columns[KEY];
hashes = map_hash(n, keys);
groups = map_modulo(n, hashes, num_groups);
idxs = map_fetch(n, groups, buckets);
searched = vec_sequence(n);  // 0,1,...,n-1
misses = vec_empty();
found = vec_empty();
do {
    separate_misses(searched, misses, idxs);
    separate_found(searched, found, key_values, keys, idxs);
    follow_list(searched, next, idxs);
} while (searched.not_empty());
insert_misses(misses, idxs)
foreach aggregate
    compute_aggr(aggregate, idxs, input);
```

Here, `separate_misses()` extracts all tuples from `searched` for which `idxs` points to the end of the list, and saves them in `misses`. Then, `separate_found()` extracts all tuples, for which the bucket has been found (key matches). Finally, `follow_list()` updates bucket pointers in `idxs` with the next bucket in the linked list for all tuples that are neither determined as nulls nor found. This process repeats while there is some tuple that needs to follow the list. Finally, all tuples in `misses` are inserted into the hash table, and their bucket indices are saved in `idxs` (we omit the details of this phase, but it needs to take care of duplicate keys in `misses`). Such code is beneficial for performance for two reasons: there are fewer but longer loops, and the loop code is simpler, allowing efficient execution. A vectorized hash-join implementation presented in Section 5.3 follows this approach, achieving performance comparable with a hand-written solution.

5.2.4 Primitive implementation

Once the data processing functionality is separated into primitives, the next task is to provide efficient implementations of these. In this section we analyze how to approach the problem of primitive implementation and discuss the programming techniques that allow development of CPU-friendly routines.

5.2.4.1 Primitive development and management

Due to the high primitive specialization along the data type, data representation and other dimensions, the number of different routines can be very high, making the manual implementation impractical. As a result a special language for describing primitives should be used. For example, in the MonetDB/X100 kernel, ca. 3000 lines of the *Mx* macro language are expanded into ca. 185.000 lines of C code implementing almost 3000 different functions.

The approach described in Section 4.2.1.3 results in a single function for each primitive signature. However, it is very well possible that on various hardware and compilation platforms different implementations of the same task can provide different performance, without a single multi-platform winner. For example, one CPU family can provide SIMD operations of some type, another can allow explicit memory prefetching, and yet another can have both capabilities – all these platforms might require different implementation approaches for optimal performance. This problem is addressed e.g. in an open-source LibOIL library [Lib] that provides multiple specialized implementations for a small set of typical data processing tasks, and at runtime determines which one to use based on CPU capabilities and micro-benchmarks. A vectorized DBMS can follow this approach to optimize performance of primitives that are execution bottlenecks. This idea can be extended even further to exploit dynamic data properties. For example, in the `Select` operator different approaches can be optimal depending on the selectivity [Ros02], and the runtime optimizer can dynamically choose the best primitive implementation.

5.2.4.2 Control dependencies

Deep execution pipelines in modern CPUs cause severe performance degradation in case of branch mispredictions. In the example code in Section 5.1.4, the only branch taken is the loop control. While this branch is easy to predict, hence relatively cheap, compilers usually further reduce its cost by applying loop unrolling.

Let us look at another routine that selects out the indices of tuples bigger than a given constant:

```
for (i = 0, found = 0; i < n; i++)
    if (input[i] > val)
        result[found++] = i;
return found;
```

As analyzed in [Ros02], such code is efficient only for very low or very high selectivities due to branch mispredictions. In this case, and in many others, the control dependency can be replaced with a data dependency, resulting in the following routine:

```
for (i = 0, found = 0; i < n; i++) {
    result[found] = i;
    found += (input[i] > val);
}
return found;
```

While issuing more instructions, this approach does not have a hard-to-predict 'if', and results in a significant performance improvement, as discussed in [Ros02] and confirmed in Figure 5.6. Another possible approach for complex, branch-heavy routines, is to separate tuples going into different code paths, as discussed in Section 5.2.3.3.

5.2.4.3 Data dependencies

Some of the control-dependency solutions involve replacing them with data-dependencies. Such dependencies can also be inherent to a data processing task. A typical case is aggregation – for example, a routine that increases the `COUNT` values stored in `result` looking at the group identifiers from `groupids` looks like this:

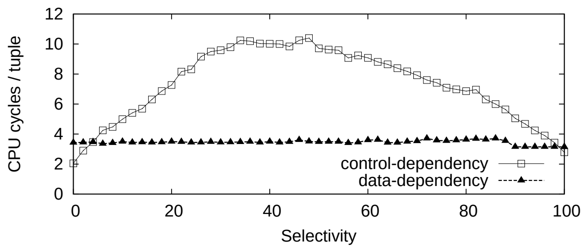


Figure 5.6: Performance of control-dependency based and data-dependency based selection routines (Core 2 Duo)

```
void aggr_count_int_vec_int_vec(int *result, int *groupids, int n) {
    for (int i = 0; i < n; i++)
        result[groupids[i]] += 1;
}
```

In this code, each tuple depends on the previous one, causing data stalls in the CPU pipeline. One approach to reduce these stalls, is to use multiple copies of the `result` array, and make different tuples update different versions of it.

```
void aggr4_count_int_vec_int_vec(int **result, int *groupids, int n) {
    for (int i = 0; i < n; i += 4) {
        result[0][groupids[i+0]] += 1;
        result[1][groupids[i+1]] += 1;
        result[2][groupids[i+2]] += 1;
        result[3][groupids[i+3]] += 1;
    }
}
```

The latter solution, while minimizing data dependencies between iterations, increases the memory consumption for 'result' arrays by a factor 4. Still, if such extra cost is acceptable, this approach allows for a significant performance improvement. For example, on our Core2Duo test machine it improved the performance from already very good 2.76 cycles/tuple (with 256 groups) to 2.05 cycles/tuple. On some architectures this difference can be significantly larger.

Another solution to the data dependency problem is to combine multiple operations into one primitive. For example, in some scenarios, multiple aggregates are computed at one processing stage – such a situation occurs in TPC-H query 1 [HNZB07]. Then, it is possible to compute e.g. 4 aggregates in one primitive:

```
void multiaggr_sum_int_vec4_int_vec(int **result, int **values, int *groupids, int n) {
    for (int i = 0; i < n; i++) {
        result[0][groupids[i]] += values[0][i];
        result[1][groupids[i]] += values[1][i];
        result[2][groupids[i]] += values[2][i];
        result[3][groupids[i]] += values[3][i];
    }
}
```

This solution, similarly to the previous routine, reduces the data dependencies and improves the performance. One of the major problems here is the use of the same data type for all 4 aggregations, which limits its applicability. Still, in scenarios like data mining, with queries often computing dozens of aggregates at once, this technique can be beneficial.

5.2.4.4 SIMDization

SIMD instructions allow processing multiple elements with one CPU instruction. Originally, they were designed to improve multimedia processing and scientific computing, but they have also been suggested for the databases [ZR02]. While having a large potential, SIMD instructions suffer from two limitations important for database processing. First, usually SIMD instructions can only operate on a set of values sharing the same data type, and the data types are usually limited to 32/64 bit integers and floats. Secondly, in most ISAs, SIMD write/load instructions usually do not have scatter/gather functionality, making them only useful for fully sequential data processing.

Overcoming the problem of datatypes is sometimes possible by casting a column to a different datatype (e.g. a character into an integer, or a float into a double). As for the strict sequential data locality, one of the solutions is to use an alternative data representation. In the previous example we used data storage known in the SIMD world as Structure-of-Arrays (SOA). It is possible to further extend it to use 'Array-of-Structures' (AOS), as presented in Figure 5.7. Note a parallel between SOA-AOS and DSM-NSM. AOS can be seen as a subset of NSM that holds data tightly packed for efficient SIMD processing. This approach has been previously presented in the context of database processing on the Cell processor [HNZB07]. As a result, our multi-aggregation code from the previous section becomes:

```
void multiaggr_sum_int4_vec_int_vec(int4 *result,
        int4 *values, int *groupids, int n) {
    for (int i = 0; i < n; i++)
        result[groupids[i]] = SIMD_add(result[groupids[i]], values[i]);
}
```

```
struct int_vec4 {              struct int4 {
    int attrA[1024];               int attrA;
    int attrB[1024];               int attrB;
    int attrC[1024];               int attrC;
    int attrD[1024];               int attrD;
};                             };
int_vec4 data;                 int4 data[1024];
```

Figure 5.7: 'Structure-of-Arrays' (SOA, left) and 'Array-of-Structures' (AOS, right)

5.3 Case study: Hash-Join

This section demonstrates how presented techniques can be used to implement a vectorized version of a hash-join, one of the most important database algorithms. Initially, we present a relatively straightforward hash-join implementation – the next section will introduce a set of additional optimizations.

5.3.1 Problem definition

Hash-join is one of the physical implementations of the relational equi-join operator, which is a specialization of the generic join operator. Formally, any join between relations R and S can be represented as: $R \bowtie_{\varphi} S = \sigma_{\varphi}(R \times S)$. Here, φ is a join condition, for equi-join represented as: $\varphi \equiv (rkey_1 = skey_1 \wedge ... \wedge rkey_n = skey_n)$, where $rkey_i$ and $skey_i$ are the *key* attributes from R and S respectively. The most often used version of an equi-join is an *N-1* join, where keys in S are unique, and for every tuple in R there is exactly 1 matching tuple in S. We will assume this type of join in the remainder of this section.

5.3.2 Standard implementation

In the hash-join, first a *build* relation S is used to construct a *hash-table* containing all the tuples from S indexed on the key of S. In the second phase, the key of every tuple from the *probe* partition R is looked up in that hash-table, and the result tuples are constructed. The following code performs the described process for a simple case, where the input relations `build` and `probe`, each with three `attributes` are joined, where the first two attributes (`0` and `1`) constitute the key. The data is stored as simple arrays, and a new, two-column `result` relation is produced containing only the values of the non-key attribute from

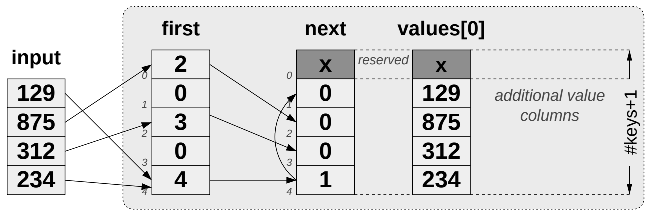


Figure 5.8: Simple bucket-chained hash table, using *modulo 5* as a hash function

both inputs. We use a simple bucket-chained hash-table, presented in Figure 5.8. Here, the `next` array represents the linked list of all tuples falling into a given bucket, with a value `0` reserved for the end of the list.

```
// Build a hash table from "build"
for (i = 0; i < build.size; i++) {
    bucket = rehash(hash(build.values[0][i]), build.values[1][i]) & mask;
    hashTable.values[0][i + 1] = build.values[0][i];
    hashTable.values[1][i + 1] = build.values[1][i];
    hashTable.values[2][i + 1] = build.values[2][i];
    hashTable.next[i + 1] = hashTable.first[bucket];
    hashTable.first[bucket] = i + 1;
}
// Probe the "probe" relation against the hash table
for (i = 0; i < probe.size; i++) {
    bucket = rehash(hash(probe.values[0][i]), probe.values[1][i]) & mask;
    current = hashTable.bucket[bucket];
    while (hashTable.values[0][current] != probe.values[0][i] ||    // assume eventual hit
            hashTable.values[1][current] != probe.values[1][i]) {
        current = hashTable.next[current];
    }
    result.values[0][i] = probe.values[2][i];
    result.values[1][i] = hashTable.values[2][current];
}
```

Note that this is a hard-coded implementation for double-key, single-value relations with known attribute data types. A real system needs to be able to perform a join on any combination of relations, including multi-key attributes with different data types, as well as different numbers of attributes. Clearly, even using macro expansions, providing the hard-coded version for all the input combinations is impossible. The following section will demonstrate how, looking at this algorithm, a generic high-performance vectorized operator can be realized.

5.3.3 Vectorized implementation

The vectorized implementation of the hash-join should be able to consume entire vectors with tuples and process them following the principles discussed in Section 5.2.1. The implementation in MonetDB/X100 provides most of the desired properties, and is based on the following observations:

- During the build phase, the processing for different tuples in a vector is not fully independent. If multiple keys fall into the same bucket, they need to be processed one after another. This can cause some data dependency, but it is not possible to avoid it with this hash table organization.

- During the probe phase, processing of different tuples is fully independent, thanks to the assumption of the *N-1* join: each probe tuple generates exactly one result tuple, hence the location of each result tuple is known.

- Finding the position in the hash table is a one-time investment for every tuple, during both build and probe phases. Once done, it allows quick insertion or lookup of multiple attributes.

- Following the linked list in the inner loop during the probe phase might take different number of steps for different tuples. Also, it introduces data and control dependencies, which are bad for modern CPUs, and makes it impossible for this code to overlap the cache misses that might occur during the linked list traversal.

5.3.3.1 Build phase

The vectorized implementation of the build phase follows closely the hard-coded version presented above, but uses vectors of size `n` as input and allows arbitrary column combinations in the input. The simplified code is as follows:

```
// Input: build relation with N attributes and K keys
// 1. Compute the bucket number for each tuple, store in bucketV
for (i = 0; i < K; i++)
    hash[i](hashValueV, build.keys[i], n); // type-specific hash() / rehash()
modulo(bucketV, hashValueV, numBuckets, n);
// 2. Prepare hash table organization, compute each tuple position in groupIdV
hashTableInsert(groupIdV, hashTable, bucketV, n)
// 3. Insert all the attributes
for (i = 0; i < N; i++)
    spread[i](hashTable.values[i], groupIdV, build.values[i], n);
```

The first task during the build phase is to find the bucket number for each build tuple. To support processing of arbitrary number and combination of key attributes, this phase is decomposed into a set of steps, as follows:

- Compute the `hashValueV` vector using a `hash*` (e.g. `hash_slng`) primitive computing a type-specific hash-function, using the first key column as a parameter.
- Adjust the `hashValueV` vector by applying a type-specific `rehash*` primitive that combines an existing hash value with a hash value for the second key column. Repeat for the remaining key columns.
- Compute the `bucketV` vector containing the bucket number for each tuple using a `modulo` (or `and`) primitive.

The resulting `bucketV` is the vectorized equivalent of the `bucket` variable in the previous section. Having this, it is possible to apply the insertion process to all tuples, In step 2 in the algorithm, the hash-table organization is prepared by adjusting the `first` and `next` arrays:

```
hashTableInsert(groupIdV, hashTable, bucketV, n) {
    for (i = 0; i < n; i++) {
        groupIdV[i] = hashTable.count++;
        hashTable.next[groupIdV[i]] = hashTable.first[bucketV[i]];
        hashTable.first[bucketV[i]] = groupIdV[i];
    }
}
```

At the same time, the `groupIdV` vector is computed, holding for each input tuple its position in the hash table. In step 3, all the input attributes are inserted into the matching positions in the hash table with type specific `spread` functions:

```
spread(hashTableValues, groupIdV, inputValues, n) {
    for (i = 0; i < n; i++)
        hashTableValues[groupIdV[i]] = inputValues[i];
}
```

5.3.3.2 Probe phase

The probe phase has two problems making it especially challenging. First, during the linked list traversal, equality comparisons can be arbitrarily complex, depending on the key structure. Secondly, the linked list traversal seems to require a per-tuple loop that would internally need to perform this complicated equality check.

In the MonetDB/X100 implementation of this phase we exploit the fact that while the inner loop length for different tuples can significantly differ, the number of steps is limited, and most tuples need to check only one or two elements in the hash table. This allows us to modify the way the linked list is traversed for all the tuples. We first find the first element in the list for every tuple. Then, we compare if these elements match our probe keys. For tuples that have a value difference, we find the next element in the list and repeat the process.

```
// Input: probe relation with M attributes and K keys, hash-table containing
//        N build attributes
// 1. Compute the bucket number for each probe tuple.
// ... Construct bucketV in the same way as in the build phase ...
// 2. Find the positions in the hash table
// 2a. First, find the first element in the linked list for every tuple,
//     put it in groupIdV, and also initialize toCheckV with the full
//     sequence of input indices (0..n-1).
lookupInitial(groupIdV, toCheckV, bucketV, n);
m = n;
while (m > 0) {
    // 2b. At this stage, toCheckV contains m positions of the input tuples for
    //     which the key comparison needs to be performed. For each tuple
    //     groupIdV contains the currently analyzed offset in the hash table.
    //     We perform a multi-column value check using type-specific
    //     check() / recheck() primitives, producing differsV.
    for (i = 0; i < K; i++)
        check[i](differsV, toCheckV, groupIdV, hashTable.values[i], probe.keys[i], m);
    // 2c. Now, differsV contains 1 for tuples that differ on at least one key,
    //     select these out as these need to be further processed
    m = selectMisses(toCheckV, differV, m);
    // 2d. For the differing tuples, find the next offset in the hash table,
    //     put it in groupIdV
    findNext(toCheckV, hashTable.next, groupIdV, m);
}
// 3. Now, groupIdV for every probe tuple contains the offset of the matching
//    tuple in the hash table. Use it to project attributes from the hash table.
//    (the probe attributes are just propagated)
for (i = 0; i < N; i++)
    gather[i] (result.values[M + i], groupIdV, hashTable.values[i], n);
```

5.3.3.3 Performance

We have experimentally analyzed the performance of the presented algorithm by comparing it with the hard-coded routines presented in the previous section. The performance of the vectorized implementation is tested with 2 vector sizes: 1 tuple, which simulates tuple-at-a-time approach, and 1024 tuples. Two 2- and 3-attribute relations were used, with 1- and 2-attribute keys, respectively. The probe relation always contains 4M tuples, all having a matching key in the build

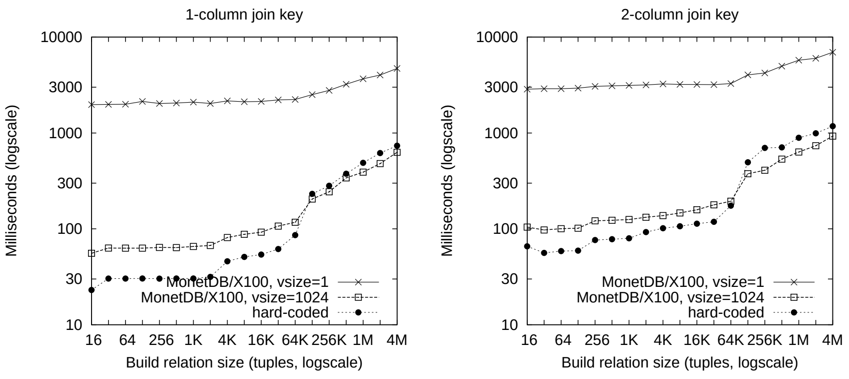


Figure 5.9: Comparison of a hard-coded hash-join implementation with the generic vectorized implementation in MonetDB/X100 (Core 2 Quad, 2.4GHz)

relation. The build relation, and hence the hash table, contains from 16 to 4M tuples with unique keys.

As Figure 5.9 shows, for cache-resident hash tables the performance of the generic MonetDB/X100 version is only ca. 2 times slower than hard-coded, specialized implementation. Surprisingly, once the hash table does not fit in the cache anymore, MonetDB/X100 implementation is *faster* than the hard-coded one. This is caused by the fact that all the operations in the vectorized version are independent, allowing e.g. overlapping of main-memory accesses. In the hard-coded version, control- and data-dependencies do not allow it, making the impact of cache-misses higher. The tuple-at-a-time implementation suffers from significant interpretation overhead, but is also less sensitive to the hash-table size. As a result, while the vectorized version provides a 30-times improvement for cache-resident data, this improvement goes down to factor 7 on memory-resident data. This demonstrates the importance of combining CPU-efficient vectorized execution with cache-optimized data placement, discussed in the next section.

5.4 Optimizing Hash-Join

The vectorized hash-join implementation demonstrated in the previous section achieves high in-cache efficiency, but suffers from significant performance degradation when working on RAM-resident data, caused by random memory accesses related to the linked list traversal. This problem can be reduced by using hashing methods that do not need a linked list, for example *cuckoo hashing* [PR04], as discussed in [ZHB06]. Still, even with this improvement the overhead of cache-misses can dominate the cost of per-tuple processing. Two main techniques were previously proposed to address this problem.

The first technique, proposed by Chen et al., uses explicit *memory prefetching* instructions inside the hash lookup routine [CAGM04]. This transforms hash-lookup throughput from a memory latency-limited into a memory bandwidth-limited workload, which can strongly improve overall hash-join performance. Our CPU-optimized hashing, however, has become too fast for memory bandwidth. Optimized cuckoo-hashing implementation from [ZHB06] spends only 7 CPU cycles per lookup and touches at least two cache lines. On a 1.3GHz CPU this implies bandwidth usage of 24GB/s, which exceeds the available RAM bandwidth. For that reason, we employ the second technique, based on *hash-table partitioning*. This idea was originally introduced for I/O based hashing in Grace Join [FKT86] and Hybrid Hash Join [DKO$^+$84] algorithms. More recently, with Radix-Cluster [MBK02], this work has been extended to hash-partitioning into the CPU cache.

The problem with these partitioned hashing techniques is that all the data needs to be first fully partitioned, and only then processed [Gra93]. This works fine in the disk-based scenario, as the temporary space for the partitions is usually considered unlimited. Main memory capacity, however, cannot be assumed to be unlimited, meaning that if the data does not fit in RAM during partitioning, it has to be saved to disk. Since using the disk when optimizing for in-cache processing is reasonable only in extreme scenarios, we propose a new hash partitioning algorithm that, while providing in-cache processing, prevents spilling data to disk.

5.4.1 Best-Effort Partitioning

Best-effort partitioning (BEP) is a technique that interleaves partitioning with execution of hash-based query processing operators without using I/O. The key idea is that if the available partition memory is filled, data from one of the partitions is passed on to the processing operator (aggregation, join), freeing space for more input tuples. In contrast to conventional partitioning, BEP is a pipelinable operator that merely reorders the tuples in a stream so that many consecutive tuples come from the same partition. Operators that use BEP, like Partitioned Hash Join and Partitioned Hash Aggregation, create a separate

```
**InitBuffers**($numBuffers$)
**while** $tuple$ = GetNextTuple($child$)
    $p$ = Hash($tuple$) mod $numPartitions$
    **if** MemoryExhausted($p$)
        **if** NoMoreBuffers()
            $maxp$ = ChooseLargestPartition()
            ProcessPartition($maxp$)
            FreeBuffers($maxp$)
        AddBuffer($p$)
    Insert($p$, $tuple$)
**for** $p$ **in** $0..numPartitions - 1$
    ProcessPartition($p$)
    FreeBuffers($p$)
```

Figure 5.10: Best-Effort Partitioning (BEP) algorithm

hash table per partition, and detect which hash table should be used at a given moment looking at the input tuples. When one of the hash tables is active, the operations on it are performed for many consecutive tuples, hence the cost of loading the hash-table into the cache is amortized among them.

Interestingly, the consuming operator can still benefit from BEP even with a single hash table, because of improved temporal locality of accesses. Still, the benefit will be significantly smaller, as memory related to the current partition is not "dense", and some space in fetched cache lines might be wasted by data of the other partitions.

An algorithm from Figure 5.10 presents an implementation where each partition consists of multiple *buffers*. When no more buffers are available, we choose the biggest partition to be processed, for two reasons. Firstly, it frees most space for the incoming tuples. Secondly, with more tuples passed for processing, the time of loading the hash-table is better amortized due to increased cache-reuse

5.4.2 Partitioning and cache associativity

The main-memory performance of data partitioning algorithms, with respect to the number of partitions, number of attributes, sizes of the CPU cache and TLB has been studied in [MBK02] and [MBNK04]. However, to our knowledge, one other important property of modern cache memories has been ignored so far: cache associativity. As discussed in Section 2.2.2, cache memories typically are not *fully associative*, but rather *N-way associative*. As a result, for different

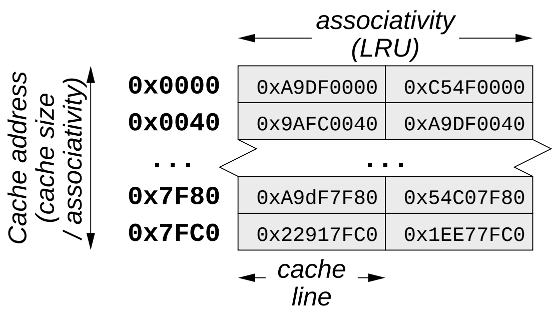


Figure 5.11: Organization of a 64 kilobyte 2-way associative cache memory with 64-byte cache-lines

addresses with the same bits used to determine the set id there are only N possible locations in the cache. For example, Figure 5.11 presents a 2-way associative 64KB cache with 64-byte cache lines – there are 512 sets, determined by bits 6..14 (mask `0x7fc0` of the memory address, and 2 cache-lines in each set.

This limitation on the number of possible locations in the cache can significantly influence the partitioning performance. This can be demonstrated by the analysis of this simple partitioning function:

```
for (i = 0; i < n; i++) {
    partno = HASH_TYPE(src[i]) & PARTITION_MASK;
    dst[partno][counts[partno]++] = src[i];
}
```

It is a common situation that the addresses of the `dst` buffers are aligned to the page size. As a result, using the cache from Figure 5.11 and a page size of 8KB, all these addresses will map onto only 4 separate cache addresses, each holding 2 cache-lines. This means that if we partition into more than 8 buffers, there is a high probability that, when we refer to a buffer that has been recently used, the cache-line with its data has already been replaced, possibly causing a cache-miss. Since the partitioning phase is usually performed using hash-values, data is roughly uniformly distributed among partitions. As a result, this *cache associativity thrashing* may continue during the entire execution of this primitive. Since the previous experiments with Radix-Cluster [MBK02] were primarily performed on a computer architecture where high fan-out partitioning deteriorated due to slow (software) TLB miss handling, these issues had previously not been detected.

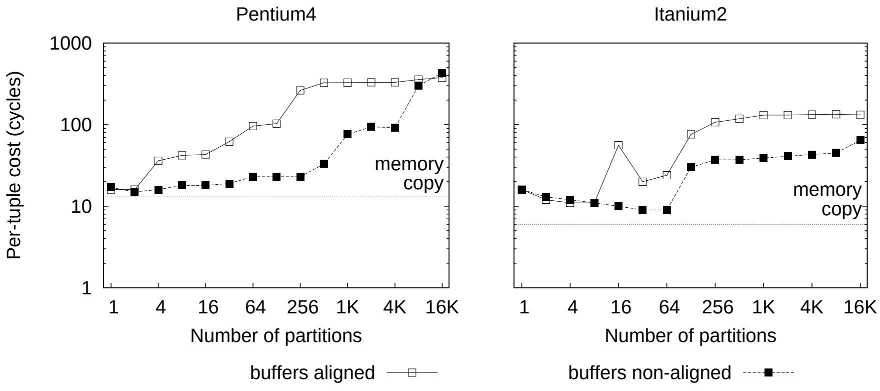


Figure 5.12: Impact of number of partitions and buffer allocation method on partitioning performance on various hardware architectures

A simple solution for the cache associativity problem is to shift each buffer address with a different multiple of a cache line size, such that all map to different cache offsets. Figure 5.12 presents the performance of the partitioning phase with both aligned and non-aligned buffers on Pentium Xeon and Itanium2 CPUs. As the number of partitions grows, the performance of aligned buffers goes down, quickly approaching the cost of random-memory access per each tuple. The non-aligned case, on the other hand, manages to achieve speed comparable to simple memory-copying even for 256 partitions. When more partitions are needed, it is possible to use a multi-pass partitioning algorithm [MBK02]. BEP can be easily extended to handle such a situation.

5.4.3 BEP performance

Performance of hash processing with best-effort partitioning is influenced by a number of factors presented in Table 5.2. The first group, data and query properties define the number of tuples stored in a hash table and their width, determining a size of the hash table. The second group, partitioning settings, determine the size of per-partition hash tables. Finally, the hardware factors influence the recommended size of the small hash tables, hence the partitioning fan-out. Moreover, cache and memory latencies influence the desirable cache-reuse factor, which determines the amortized cost of data access.

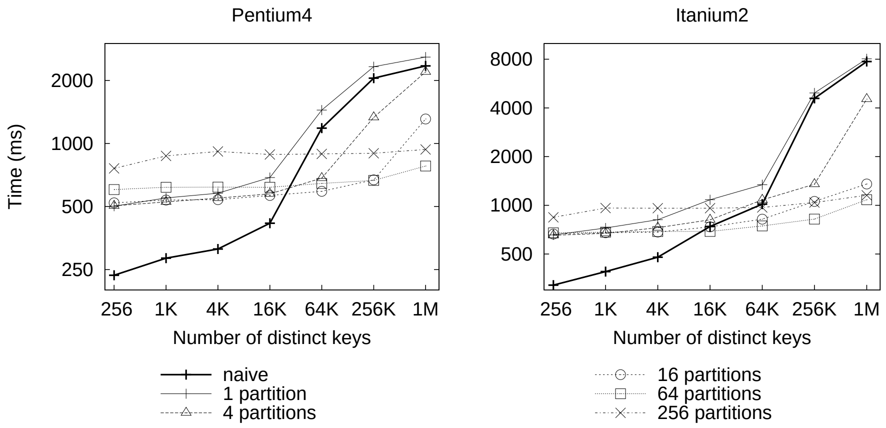


Figure 5.13: Aggregation performance with varying number of partitions and distinct keys (20M tuples)

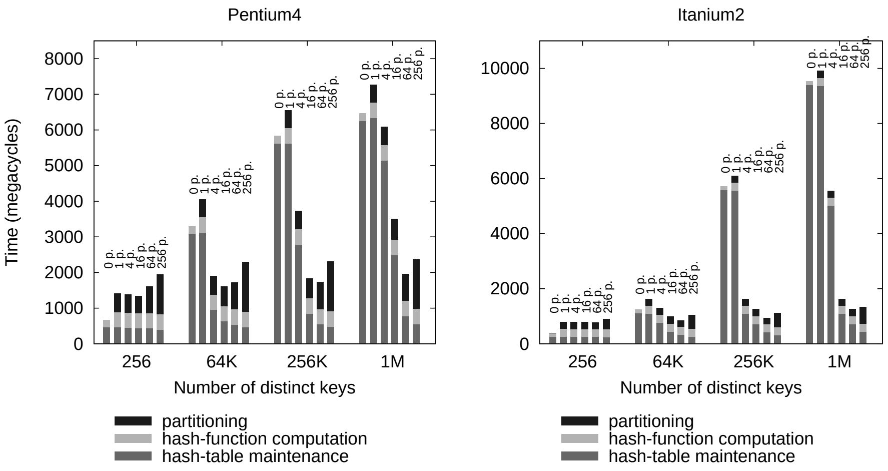


Figure 5.14: Execution profiling with varying number of partitions and distinct keys (20M tuples)

Table 5.2: Best-Effort Partitioning parameters

Description	Symbol	Example
Query properties		
Number of distinct values	D	1 M
Number of tuples	T	20 M
Input width	$\widehat{i}$	4 B
Hash-table: data width	$\widehat{h_d}$	4 B
Hash-table: buckets width	$\widehat{h_b}$	8 B
Hash-table: per-key memory = $\widehat{h_d} + 2 \cdot \widehat{h_b}$ (Cuckoo, 50% fill ratio)	$\widehat{h_w}$	20 B
Hash-table: size = $D \cdot \widehat{h_w}$	$\|H\|$	20 MB
BEP settings		
Available buffer memory	$\|M\|$	30 MB
Number of partitions	P	16
Partition: size = $\frac{\|M\|}{P}$	$\|M_p\|$	1.875 MB
Partition: tuples buffered = $\frac{\|M_p\|}{\widehat{i}}$	T_p	480 K
Partition: hash-table size = $\frac{\|H\|}{P}$	$\|H_p\|$	1.25 MB
Number of per-lookup random accesses (Cuckoo)	a	4
Hardware properties (Example = Itanium2)		
Cache size	$\|C\|$	3 MB
Cache line width	$\widehat{C}$	128 B
Cache latency	l_C	14 cycles
Main-memory latency	l_M	201 cycles

We now discuss in detail one particular scenario of using BEP for partitioned hash aggregation. This setting is later used in experiments on our Itanium2 machine. The relevant hardware and algorithm parameters are listed in Table 5.2, which in its rightmost column also contains the specific hardware characteristics of Itanium2. Note that Itanium2 has a large and fast L3 cache, which is the optimization target (in case of Pentium4, it is best to optimize for L2).

Example Scenario. Assume we need to find 1M unique values in a 20M single-attribute, 4-byte long tuples using 50MB of RAM on our Itanium2 machine with a 3MB L3 cache with 128-byte cache-lines. A hash table with a load factor of 0.5 occupies 20MB using optimized single-column Cuckoo Hashing [ZHB06]: 16MB for the bucket array and 4MB for the values. Using 16 partitions will divide

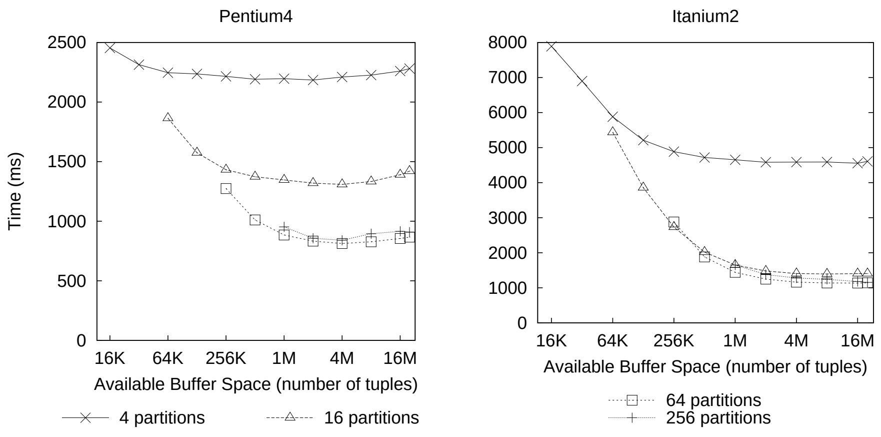


Figure 5.15: Impact of available buffer space (20M tuples, 1M unique values)

it into 1.25MB (cache-resident) hash-tables. There will be 30MB of RAM left for partitions, and assuming uniform tuple distribution (which is actually the worst case scenario for our algorithm), the largest partition during overflow occupies 1.875MB, holding 480K 4-byte tuples. Thus, when this partition is processed, 480K keys are looked-up in a hash-table, using 4 random memory accesses per-tuple, resulting in 1875K accesses. Since the hash table consists of 10240 128-byte cache lines, each of them will be accessed 188 times. With main-memory and (L3) cache latencies of 201 and 14 cycles, respectively, this results in an average access cost of 15 cycles.

Experiments. Figure 5.13 compares in a micro-benchmark naive (non-partitioned) and best-effort partitioning hash aggregation, in a `"SELECT DISTINCT key FROM table"` query on a 20M 4-byte wide tuples table, with a varying number of distinct keys. When this number is small, the hash table fits in the CPU cache, hence the partitioning only slows down execution. When the number of keys grows, the hash table exceeds the cache size, and best-effort partitioned execution quickly becomes fastest. Figure 5.14 shows a performance break-down into partitioning cost, hash table maintenance (lookup and inserts) and hash function computation. With more partitions, the data locality improves, making the hash table maintenance faster. On the other hand, more partitions result in a slower partitioning phase. Finally, we see that with partitioned execution the cost of the hash-function is two times higher, as it is computed both in partitioning and lookup phases. Depending on the cost of computing this function (especially when it is computed over multiple attributes), it can be more beneficial to store it during partitioning and reuse it during lookup.

The performance of partitioned execution depends highly on the cache-reuse ratio during one processing phase, which in turn depends on the amount of buffer space. As Figure 5.15 shows, with an increasing number of buffered tuples, performance improves since more tuples hit the same cache line. If the number of partitions is big enough to make the hash table fit in the cache, adding more partitions does not change performance given the same buffer space. Finally, we see that the performance curve quickly flattens, showing that the performance can be close to optimal with significantly lower memory consumption. In this case, processing time with a buffer space of only 2M tuples is the same as with 20M tuples (which is equivalent to full partitioning). We see this reduced RAM requirement as the main advantage of best-effort partitioning.

Cost Model. We now formulate a cost model to answer the question "what is the amount of buffer memory that should be given to BEP to achieve (near) optimal performance?"

The cost of the amortized average data access cost during hash-table lookup depends on the cache-reuse factor:

$$access\_cost = l_C + \frac{l_M}{reuse\_factor}$$

The cache-reuse factor is the expected amount of times a cache line is read while looking up in the hash table all tuples from a partition. It can be computed looking at the query, partitioning and hardware properties from Table 5.2:

$$reuse\_factor = \frac{T_p \cdot a \cdot \widehat{C}}{|H_p|} = \frac{|M| \cdot a \cdot \widehat{C}}{\widehat{i} \cdot D \cdot \widehat{h_w}}$$

A good target for the cache-reuse factor is to aim for an amortized RAM latency close to the cache performance, for example 25% higher:

$$\frac{l_M}{reuse\_factor} = \frac{l_C}{4}$$

This, in turn, allows us to compute the required amount of memory BEP needs:

$$|M| = \frac{l_M \cdot 4 \cdot \widehat{i} \cdot D \cdot \widehat{h_w}}{l_C \cdot a \cdot \widehat{C}}$$

In the case of our Itanium2 experiments we arrive at:

$$|M| = \frac{201 \cdot 4 \cdot 4 \cdot 1M \cdot 20}{14 \cdot 4 \cdot 128} = 9{,}409{,}096 \text{ B} = 2{,}352{,}274 \text{ tuples}$$

and in case of Pentium 4:

$$|M| = \frac{370 \cdot 4 \cdot 4 \cdot 1M \cdot 20}{24 \cdot 4 \cdot 128} = 10{,}103{,}464 \text{ B} = 2{,}525{,}866 \text{ tuples}$$

This prediction is confirmed in Figure 5.15, where a buffer of 2M tuples results in the optimal performance.

As a final observation, it is striking that the amount of partitions does not play a role in the formula. The cost model does assume, though, that the hash table fits in the CPU cache. This once again is confirmed in Figure 5.15, which shows that once partitions are small enough for them to fit in the CPU cache, performance does not change. Note that on Pentium4, the 16 partition line is in the middle, because at that setting the hash-tables (20MB/16 = 1.25MB) are just a bit too large to fit L2, but average latency has gone down with respect to pure random access.

5.4.4 BEP discussion

Best-effort partitioning can be easily applied to various relational operations. In aggregation, the ProcessPartition() function simply incrementally updates the current aggregate results. In joins and set-operations, the regular partitioning can first be used for the build relation, and then BEP can be applied for the probe relation. This allows, for example, cache-friendly joining of two relations if only one of them fits in main memory. This can be further extended to multi-way joins using *hash teams* [GBC98].

The flexibility of BEP memory requirements is useful in a scenario where the memory available for the operator changes during its execution. If the memory manager provides BEP with extra memory, it can be simply utilized as additional buffer space. If, on the other hand, available memory is reduced, BEP only needs to pass some of the partitions to the processing operator and free the buffers they occupied.

The ideas behind BEP can be applied in a scenario with more storage levels, e.g. in a setup with a fast flash drive and a slow magnetic disk. If the hash-table does not fit in main memory, and the partitioned data is too large to fit on a flash drive, BEP can be used to buffer the data on a flash device and periodically process memory-size hash tables, possibly again using BEP to make it cache-friendly. This scenario raises the question whether it is possible to build a *cache-oblivious* data structure [FLPR99] with properties similar to those of BEP.

BEP is related to a few other processing techniques besides vanilla data partitioning. *Early aggregation* [Lar97] allows computing aggregated results for part of the data and later join combine them. In parallel *local-global aggregation* [Gra93], tuples can be distributed using hash-partitioning among multiple nodes. If the combined memory of these nodes is enough to keep the whole hash table, I/O-based partitioning is not necessary. In *hybrid hashing* [DKO$^+$84], the effort is made to keep as much data in memory as possible, spilling only some of the partitions to disk. While there are clearly similarities between BEP and these techniques, BEP provides a unique combination of features: *(i)* it allows efficient processing if the data does not fit in the first-level storage (cache), *(ii)* it optimizes data partitioning for a limited second-level storage (main memory), *(iii)* it allows a non-blocking partitioning phase, and, finally, *(iv)* it can be easily combined with dynamic memory adjustments.

5.5 Extending the vectorized world

One of the concerns related to vectorized processing is that originally it has been limited to pure numeric processing, ignoring many issues crucial to database performance, but often neglected in research. In this section we discuss how vectorized processing can be applied in some of these areas.

5.5.1 Overflow checking

Arithmetic overflows are rarely analyzed in the database literature, but they are a necessity in a production-quality system. While CPUs do detect overflows, many programming languages (e.g. C++) do not provide mechanisms to check for them. As mentioned in Section 5.1.3, on some platforms, a special *summary overflow* processor flag can be checked to detect an overflow over a large set of computations. Still, mainstream Intel and AMD CPUs do not have such capabilities, and software solutions need to be applied to this problem. One of the approaches is to cast a given datatype into a larger one, and check if the result of the arithmetic operation fits into the smaller datatype range. A simple overflow-handling addition primitive for unsigned integers could then look like this

```
int map_add_int_vec_int_vec(uint *result, uint *input1, uint *input2, int n) {
    for (int i = 0; i < n; i++) {
        ulong l1 = input1[i];
        ulong l2 = input2[i];
        ulong res = l1 + l2;
        if (res > 0xFFFFFFFFUL)
            return STATUS_ERROR;
        result[i] = (uint)res;
    }
    return STATUS_OK
}
```

Note again that the overflow check is performed for every tuple. An optimized vectorized version could look like this:

```
int map_add_int_vec_int_vec(int *result, uint *input1, uint *input2, int n) {
    ulong tmp = 0;
    for (int i = 0; i < n; i++) {
        ulong l1 = input1[i];
        ulong l2 = input2[i];
        ulong res = l1 + l2;
        tmp |= res;
        result[i] = (int)res;
    }
    if (tmp > 0xFFFFFFFFUL)
        return STATUS_ERROR;
    return STATUS_OK;
}
```

While the check in the first version can be perfectly predicted, it still causes some overhead. As a result, removing it in the second version gives us a 25% boost on our test Core2Duo machine.

5.5.2 NULL handling

NULL handling is another often ignored issue in performance-focused research. There are different options for NULL representation in data, including a reserved NULL value, a list of positions with (or without) NULL values and a bitmap with a bit set for every NULL value. In this section we demonstrate an example where vectorization improves the NULL handling with the bitmap representation.

Figure 5.16 demonstrates the performance of different possible implementations of a NULL-handling integer addition primitive. Both inputs use a data representation with a "value" vector holding all the values (including undefined values for NULLs), and a "bitmap" vector holding the NULL bitmap. The first, "iterative" version, for every tuple checks if the proper bit in either input bitmap is set. If so, it sets the bit in the destination NULL bitmap, and

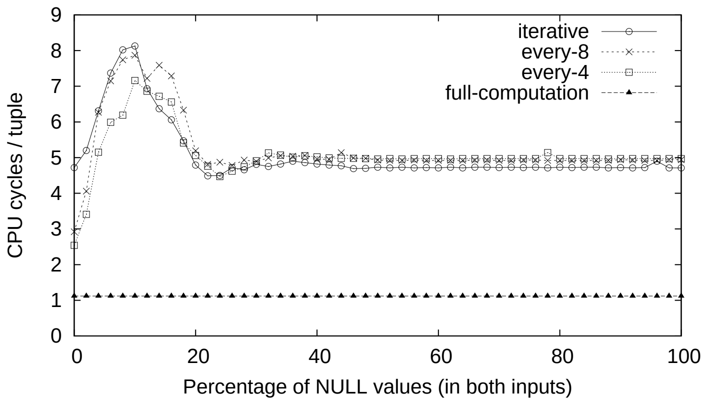


Figure 5.16: NULL-handling addition using the standard approach, per-8-bit and per-4-bit checks, and a full-computation approach (Core2Duo)

if not, performs the actual addition. The “every-8” and “every-4” versions performs a trick mentioned in Section 5.1.5: they check 8 or 4 bits at once in both inputs, and if either is non-zero, they perform the slow “iterative” code. This can be beneficial with a very small number of NULL values. Finally, the “full-computation” version exploits the fact that in many cases performing the computation for the NULL values does not cause an error, as long as the result value is marked as NULL. First, it creates a destination bitmap by simply binary-ORing the source bitmaps. Then, it performs the addition for *all* tuples. Both loops can be very efficiently optimized by the compiler, resulting in a 3 to 8 times faster implementation. Clearly, this aggressive approach is not applicable in every case – for example, dividing by a NULL tuple that has a zero in the value vector can cause an error. Also, when the performed computation is expensive, the extra performed operations can outweigh the benefit of removing the comparison. Still, it is a nice example of the improvement possible with vectorized processing.

5.5.3 String processing

An obvious application of vectorized processing for strings is in areas where there exist some (possibly approximate) fixed-width string representations, for example hash values or dictionary keys for strings from limited domains. In

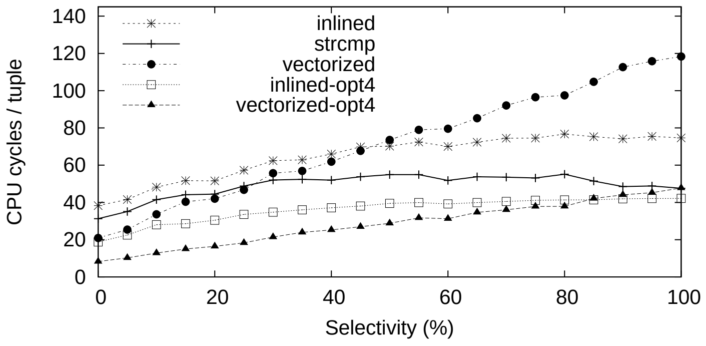


Figure 5.17: Performance of string-equality selection using `strcmp()`, inlined comparison and vectorized version (Core 2 Duo)

such a case, for example in a string equality-search, vectorized processing can be used to perform efficient filtering-out of the non-matching strings using integer processing, possibly followed by the expensive full-string comparison for a subset of tuples.

Even when working on real strings, it turns out there are cases when vectorization can help. Figure 5.17 presents an experiment in which we perform string-equality search, using a 20-byte long key, and a collection in which half of the non-matching strings share some prefix with the key. We use a standard zero-terminated C string representation. We compare three types of implementations: one based on the standard `strcmp()` function, one in which the `strcmp()` functionality is inlined, and a vectorized one. In the vectorized implementation, instead of comparing the entire strings one after another, we first select out strings with a matching first character, then from these we select strings with a matching second character, and so on. Additionally, for the inlined and vectorized versions, we applied an additional optimization ("-opt4"), in which we compare 4 bytes in one step as integers (except for the last few bytes of the pattern). This technique is safe, as long as it is known that the 3 bytes after each string are safe to be addressed by the user process, which can be guaranteed by the memory allocation and buffer management facilities.

Figure 5.17 demonstrates that the standard `strcmp()` function is quite efficient and can beat the inlined version. Still, it loses with the vectorized solution for selectivities ranging from 0% to 25%. The main reason of this difference comes from the fact that the vectorized version performs fewer but longer loops over the data, reducing the loop management overhead. If the 4-byte comparison trick is applied, the vectorized version becomes a clear winner, providing as much as 2- and 4- times improvement over optimized inlined and default `strcmp()` implementation, respectively.

While not conclusive, this simple experiment shows that vectorization can be efficiently applied to some string processing problems. We believe that this research area, while relatively unexplored, has a potential for more significant improvements.

5.5.4 Binary search

Finding an element in an ordered sequence is a problem present in many data processing tasks. In databases, it occurs e.g. when searching for an element in a sorted dictionary, finding the next pointer during the B-tree traversal, performing merge join and more. Typically, *binary search* is used to implement this task. It is a very well researched problem, often being used as an example of algorithm design and analysis [Ben99]. Let us define this problem as finding an element `key` in an ordered array `data` of size `N`, returning the value `p` equal to a position in `data` of a found element, or `-1` if the element does not occur in the array. We extend this problem to a vectorized case, when we need to perform multiple binary searches for all values in a `keys` arrays of size `M`, and store the positions in the `result` array. This "bulk" implementation based on the binary search version from [Ben99] looks as follows:

```
// BINARY SEARCH - SIMPLE
for (i = 0; i < M; i++) {
    key = keys[i];
    l = -1;
    u = N;
    while (l + 1 != u) {
        m = (l + u) / 2;
        if (data[m] < key)
            l = m;
        else
            u = m;
        p = u;
    }
    result[i] = p;
}
```

One possible optimization of this `SIMPLE` algorithm, discussed in [Ben99], exploits the fact that if the array size is a power of 2, the code can be simpler, as division by two never results in rounding errors. We used this technique to

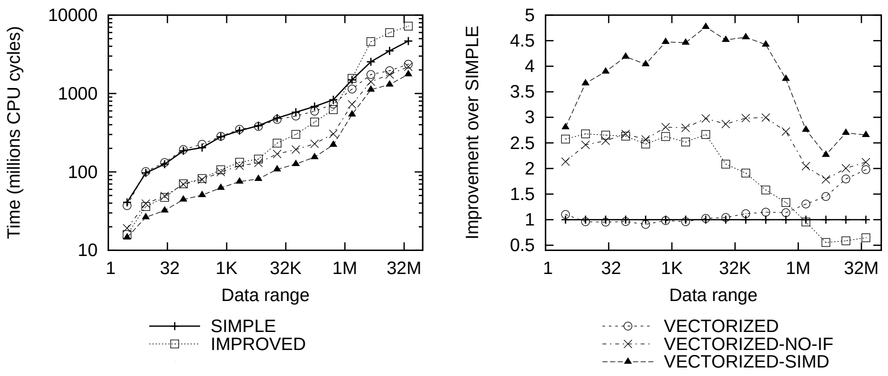


Figure 5.18: Performance of different binary-search implementations (Core 2 Quad 2.4 GHz, 16KB L1 D-cache, 4MB L2 cache)

implement a solution for a simplified binary search problem, when the `key` is guaranteed to be in the `data`:

```
// BINARY SEARCH - IMPROVED
powerOfTwo = pow(2, floor(log2(N - 1))); // biggest power of 2 smaller than N
splitIndex = N - powerOfTwo;        // used to divide a problem into two problems
splitValue = data[splitIndex];      // with sizes guaranteed to be powers of 2
for (i = 0; i < M; i++) {
    key = keys[i];
    if (key >= splitValue)
        p = splitIndex
    else
        p = 0;
    for (j = powerOfTwo / 2; j >= 1; j = j / 2)
        if (key >= data[p + j])
            p = p + j;
    result[i] = p;
}
```

Figure 5.18 presents results of an experiment, in which we perform a search of 2M keys in a `data` array of unique values, guaranteed to include the keys. The size of `data` increases from 3 to ca. 43 millions, with step 3. As Figure 5.18 shows, the `IMPROVED` version provides a significant boost, mostly related to its simpler code. Still, this boost is only visible when `data` fits in the CPU cache - once the lookups start to cause main memory accesses, the time gets dominated by the cache misses.

Looking at the code in the IMPROVED version, we can observe that the iterations in the inner loop are not independent. This is caused by the nature of the binary tree traversal – to find the next node, the previous node needs to be fully compared with. As a result, the code cannot fully exploit superscalar CPU features and cache misses cannot be overlapped between iterations. To improve this situation, we can exploit the fact that the inner loops are independent for different searched keys (values of i). Also, the value that is added or not to the current position p at a given level (j) is the same for different keys. This leads to the following, VECTORIZED implementation of our problem:

```
// BINARY SEARCH - VECTORIZED
VSIZE=256;  // vector size
// Perform computation vector by vector
for (processed = 0; processed < M; processed += VSIZE) {
    int *result_vector = result + processed;
    int *keys_vector = keys + processed;
    // First, for each vector, perform the first phase of the search
    for (i = 0; i < VSIZE; i++)
        result_vector[i] = splitIndex * (keys_vector[i] >= splitValue);
    // Then, for each search phase, perform it for all the elements in a vector
    for (j = powerOfTwo / 2; j >= 1; j = j / 2) {
        int *data_shifted = data + j;
        for (i = 0; i < VSIZE; i++)
            if (keys_vector[i] >= data_shifted[result_vector[i]])
                result_vector[i] += j;
    }
```

This version follows the idea of *phase separation* from Section 5.2.3.2. The state of each element in a given phase is saved in result_vector. As a result, each iteration in the inner-most loop is fully independent. Also, some optimizations become possible, e.g. introducing the data_shifted variable, which allows to remove the addition of p during each comparison. While this version achieves a relatively poor performance, as seen in Figure 5.18, it is much more resistant to the data not fitting in the CPU cache. This is because the cache misses occurring in the lookup phase are independent, and can be overlapped by the memory controller. This motivates further optimizations of this routine. One problem found in this routine is the if statement in the inner-most loops. This *control dependency* can be easily converted into *data dependency* with this simple code:

```
// BINARY SEARCH - VECTORIZED-NO-IF
...
        for (i = 0; i < VSIZE; i++)
            result_vector[i] += (keys_vector[i] >= data_shifted[result_vector[i]]) * j;
...
```

This version significantly improves the performance, by matching the IMPROVED version for in-cache data ranges and being more resistant to cache misses for larger data ranges.

The final optimization comes from the observation that since all the operations for different keys are independent, SIMD instructions can be used to further improve the performance. This SIMD-ized version of the binary search is as follows (using Intel ICC SIMD intrinsics [Int07c]):

```
// BINARY SEARCH - VECTORIZED-SIMD
VSIZE=256;  // vector size
for (processed = 0; processed < M; processed += VSIZE) {
    int *result_vector = result + processed;
    int *keys_vector = keys + processed;
    __m128i xm_idxvec = _mm_set_epi32(splitIndex, splitIndex, splitIndex, splitIndex);
    __m128i xm_valvec = _mm_set_epi32(splitValue, splitValue, splitValue, splitValue);
    // Prepare the first index in the search
    for (i = 0; i < VSIZE; i += 4) {
        __m128i xm_vals = _mm_load_si128((__m128i*)(keys_vector + i));
        xm_vals = _mm_andnot_si128(
                _mm_cmplt_epi32( xm_vals, xm_valvec ),
                xm_idxvec);
        _mm_store_si128((__m128i*)(result_vector + i), xm_vals);
    }

    // Then, for each search phase, perform it for all the elements in a vector
    for (j = powerOfTwo / 2; j >= 1; j /= 2) {
        int *data_shifted = data + j;
        __m128i xm_jvec = _mm_set_epi32(j, j, j, j);
        for (i = 0; i < VSIZE; i += 4) {
            __m128i xm_idxvec = _mm_load_si128((__m128i*)(result_vector + i));
            int cmpval0 = data_shifted[result_vector[i + 0]];
            int cmpval1 = data_shifted[result_vector[i + 1]];
            int cmpval2 = data_shifted[result_vector[i + 2]];
            int cmpval3 = data_shifted[result_vector[i + 3]];
            __m128i xm_cmpvalvec = _mm_set_epi32(cmpval3, cmpval2, cmpval1, cmpval0);
            __m128i xm_valvec = _mm_load_si128((__m128i*)(keys_vector + i));
            xm_idxvec = _mm_add_epi32(
                    xm_idxvec, _mm_andnot_si128(
                            _mm_cmplt_epi32(xm_valvec, xm_cmpvalvec),
                            xm_jvec));
            _mm_store_si128((__m128i*)(result_vector + i), xm_idxvec);
        }
    }
}
```

This VECTORIZED-SIMD version, while more complicated, allows to significantly reduce the computation costs by executing the same operations for multiple keys (4 in this case). Note that this implementation cannot be fully SIMD-ized due to ISA limitations. For example, Intel SSE instructions do not have single “gather” memory access instructions, allowing filling in a single SIMD register with data from multiple memory locations – this leads to explicit creation of the `xm_cmpvalvec` variable. Even with this non optimal implementation, Figure 5.18 shows that `VECTORIZED-SIMD` beats all other implementations by a large margin, especially when the data is in the CPU cache. Keep in mind that the non-vectorized versions cannot be SIMD-ized in a similar manner, since the computations there are not independent.

This section demonstrates again that vectorized processing, when applied, allows many optimization techniques impossible in classical approaches. We demonstrated how *phase-separation* can be used to provide independent operations, allowing better overlapping of cache misses. Then, by removing the *control dependency* we improved the performance on superscalar CPUs. Finally, the introduction of SIMD instructions allowed to amortize the computation cost among multiple elements. This gave a 2-3 times improvement over an optimized non-vectorized version, both for in-cache and in-memory data.

5.6 Conclusions

This chapter demonstrates that the vectorized execution model, proposed in Chapter 4 has numerous advantages over both the traditionally applied tuple-at-a-time model and the column-at-a-time model of MonetDB. However, the new approach results in new challenges, especially in the area of expressing relational operators in a vectorized way. Implementation techniques proposed in this chapter can make this process easier, allowing generic implementations of different data processing tasks that are often approaching the performance of hard-coded solutions. This high performance is achieved not only by removing the interpretation overheads found in an iterative approach, but also thanks to the ability to exploit various features of modern CPUs: superscalar processing, SIMD instructions and cache memories.

Chapter 6

Light-weight data compression

The previous chapter demonstrated how the vectorized execution model can be used to achieve high performance for main-memory processing tasks. As discussed in Section 4.3, scaling this performance to disk-resident datasets poses a challenge. The main reason is the imbalance between the continuously increasing CPU power and the relatively stagnated disk performance. Even with a scan-based processing approach, presented in Section 4.3.1, the bandwidth of a decent RAID system cannot cope with the high performance of the MonetDB/X100 execution layer.

To obtain high query performance, even with modest disk configurations, we propose new forms of *lightweight data compression* that reduce the I/O bandwidth need of database and information retrieval systems. Our work differs from previous use of compression in databases and information retrieval in the following aspects:

Super-scalar Algorithms. We contribute three new compression schemes (PDICT, PFOR and PFOR-DELTA) that are specifically designed for the super-scalar capabilities of modern CPUs. In particular, these algorithms lack any *if-then-else* constructs in the performance-critical parts of their compression and decompression routines. Also, the absence of dependencies between values being (de)compressed makes them fully *loop-pipelinable* by modern compilers and allows for *out-of-order* execution on modern CPUs that achieve high *instructions per cycle* (IPC) efficiency. As a result, PFOR, PFOR-DELTA and PDICT

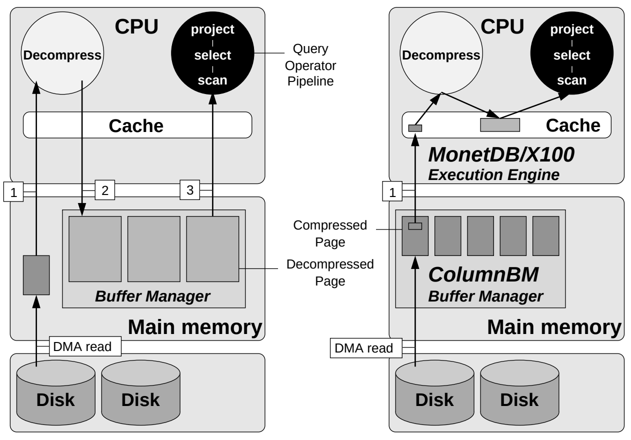


Figure 6.1: I/O-RAM vs RAM-CPU compression

spend as little as 1-2 CPU cycles per 1 byte of source data, and a fraction of that during decompression. This makes them even 10 times faster than previous speed-tuned compression algorithms and allows them to improve I/O bandwidth even on RAID systems that read and write data at rates of hundreds of MB/s.

Improved Compression Ratios. PDICT and PFOR are generalizations of respectively dictionary and Frame-Of-Reference (FOR) or prefix-suppression (PS) compression that were proposed previously [NCR02, GRS98, WKHM00]. In contrast to these schemes, our new compression methods can gracefully handle data distributions with outliers, allowing for a better compression ratio on such data. We believe this makes our algorithms also applicable to information retrieval. In particular, we show that PFOR-DELTA compression ratios on the TREC dataset approach that of a recently proposed high-speed compression method tuned for inverted files [AM05] ("carryover-12"), while retaining a 7-fold compression and decompression speed advantage.

RAM-CPU Cache Compression. We make a case for compression/decompression to be used on the boundary between the CPU cache and RAM storage levels. This implies that we also propose to cache pages in the buffer manager (i.e. in RAM) in compressed form. Tuple values are decompressed at a small granularity (such that they fit the CPU cache) just-in-time, when the query processor needs them.

Previous systems [Syb] use compression between the RAM and I/O storage levels, such that the buffer manager caches decompressed disk pages. Not only does this mean that the buffer manager can cache less data (causing more I/O), but it also leads the CPU to move data three times in and out of the CPU cache during query processing. This is illustrated by the left-most side of Figure 6.1: first the buffer manager needs to bring each recently read disk block from RAM to the CPU for decompression, then it moves it back in uncompressed form to a buffer page in RAM, only to move the data a third time back into the CPU cache, when it is actually needed by the query. As buffer manager pages are compressed, a crucial feature of all our new compression schemes is fine-grained decompression, which avoids full page decompression when only a single value is accessed.

We implemented PDICT, PFOR and PFOR-DELTA as an integral part of ColumnBM. Our experiments show that on the 100GB TPC-H benchmark, our compression methods can improve performance by a factor equal to the compression ratio in I/O constrained systems, and eliminate I/O as the dominant cost factor in most cases. We tested our compression methods both using DSM column-wise table storage [CK85] as well as a PAX layout, where data within a single disk page is stored in a vertically decomposed fashion [ADHS01]. While the TPC-H scenario favors the column-wise approach, PAX storage also strongly benefits from our compression, extending its use to scenarios where the query mix contains more OLTP-like queries.

The outline of this chapter is as follows. In Section 6.1 we relate our algorithms to previous work on database compression. Section 6.2 then introduces our new PFOR, PFOR-DELTA and PDICT compression algorithms. We use CPU performance counters on three different hardware architectures to show in detail how and why these algorithms achieve multi GB/s (de)compression speeds. We evaluate the effectiveness of our techniques using MonetDB/X100 on TPC-H in Section 6.3, as well as on information retrieval datasets from TREC and INEX in Section 6.4. Section 6.5 concludes this chapter and outlines future work.

6.1 Related work

Previous work on compression in database systems coincides with our goal to save I/O, which requires *lightweight* methods (compared with compression that minimizes storage size), such that decompression bandwidth clearly outruns I/O bandwidth, and CPU-bound queries do not suffer too great a setback by additional decompression cost. In the following, we describe a number of previously proposed database compression schemes [WKHM00, GS91, GRS98]:

Prefix Suppression (PS) compresses by eliminating common prefixes in data values. This is often done in the special case of zero prefixes for numeric data types. Thus, PS can be used for numeric data if actual values tend to be significantly smaller than the largest value of the type domain (e.g. prices that are stored in large decimals).

Frame Of Reference (FOR), keeps for each disk block the minimum min_C value for the numeric column C, and then stores all column values $c[i]$ as $c[i] - min_C$ in an integer of only $\lceil log_2(max_C - min_C + 1)\rceil$ bits. FOR is efficient for storing clustered data (e.g. dates in a data warehouse) as well as for compressing node pointers in B-tree indices. FOR resembles PS if $min_C = 0$, though the difference is that PS is a variable-bitwidth encoding, while FOR encodes all values in a page with the same amount of bits.

Dictionary Compression, also called "enumerated storage" [Bon02], exploits value distributions that only use a subset of the full domain, and replaces each occurring value by an integer code chosen from a dense range. For example, if gender information is stored in a `VARCHAR` and only takes two values, the column can be stored with 1-bit integers (`0="MALE"`, `1="FEMALE"`). A disadvantage of this method is that new value inserts may enlarge the subset of used values to the point that an extra bit is required for the integer codes, triggering recompression of all previously stored values.

Several commercial database systems use data compression; especially node pointer prefix compression in B-trees is quite prevalent (e.g. in DB2). Teradata's Multi-Valued Compression [NCR02] uses dictionary compression for entire columns, where the DBA has the task of providing the dictionary. Values not in the dictionary are encoded with a reserved *exception value*, and are stored elsewhere in the tuple. Oracle also uses dictionary compression, but on the granularity of the disk block [PP03]. By using a separate dictionary for each disk block, the overflow-on-insert problem is easy to handle (at the price of additional storage size).

The use of compressed column-wise relations in our approach strongly resembles the Sybase IQ product [Syb]. Sybase IQ stores each column in a separate set of pages, and each of these pages may be compressed using a variety of schemes, including dictionary compression, prefix suppression and LZRW1 [Wil91]. LZRW1 is a fast version of common LZW [Wel84] Lempel-Ziv compression, which uses a hash table *without* collision list to make value lookup during compression and decompression simpler (typically achieving a reduced compression ratio when compared to LZW). While faster than the common Lempel-Ziv compression utilities (e.g. gzip), we show in Section 6.2 that LZRW1 is still an order of magnitude slower than our new compression schemes. Another major difference with our approach is that the buffer manager of Sybase IQ caches decompressed pages. This is unavoidable for compression algorithms like LZRW1 that do not allow for fine-grained decompression of values. Page-wise decompression fully hides compression on disk from the query execution engine, at the expense of additional traffic between RAM and CPU cache (as depicted in Figure 6.1).

An interesting research direction is to adaptively determine the data compression strategy during query optimization [CGK01, GS91, WKHM00]. An example execution strategy that optimizes query processing by exploiting compression may arise in queries that select on a dictionary-compressed column. Here, decompression may be skipped if the query performs the selection directly on the integer code (e.g. on `gender=1` instead of `gender="FEMALE"`), which both needs less I/O and uses a less CPU-intensive predicate. Another opportunity for optimization arises when (arithmetic) operations are executed on a dictionary compressed column. In that case, it is sometimes possible to execute the operation only on the dictionary, and leave the column values unchanged [Syb] (called "enumeration views" in [Bon02]). Optimization strategies for compressed data are described in [CGK01], where the authors assume page-level decompression, but discuss the possibility to keep the compressed representation of the column values in a page in case a query just copies an input column unchanged into a result table (unnecessary decompression and subsequent compression can then be avoided).

Finally, compression to reduce the volume of transferred and memory-resident data has received significant attention in the information retrieval community, in particular for compressing inverted lists [WMB99]. Inverted lists contain all positions where a term occurs in a document (collection), always yielding a monotonically increasing integer sequence. It is therefore effective to compress the *gaps* rather than the term positions (*Delta Compression*). Such compression is the prime reason why inverted lists are now commonly considered superior to signature files as an IR access structure [WMB99]. Early inverted list compression work focused on exploiting the specific characteristics of gap distributions to achieve optimal compression ratio (e.g. using Huffman or Golomb coding tuned to the frequency of each particular term with a local Bernoulli model [Huf52]). More recently, attention has been paid to schemes that trade compression ratio for higher decompression speed [Tro03]. In Section 6.4, we show that our new PFOR compression scheme compares quite favorably with a recent proposal in this direction, the word-aligned compression scheme called "carryover-12" [AM05].

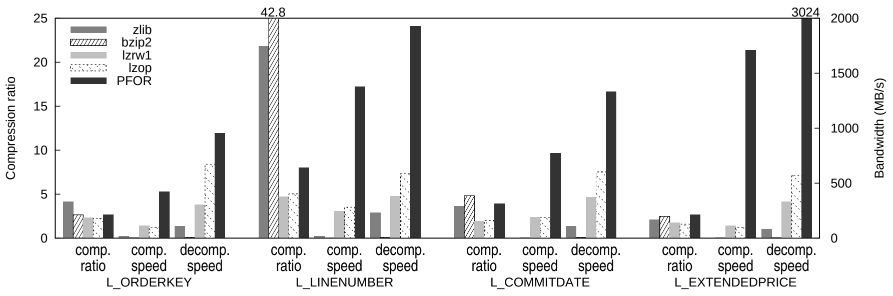


Figure 6.2: Comparison of various compression algorithms on a subset of TPC-H columns

6.2 Super-scalar compression

In this section we describe how insight in extracting high IPC (Instructions Per Cycle) efficiency from super-scalar CPUs led us to the design of PFOR, PFOR-DELTA and PDICT. Figure 6.2 shows that state-of-the-art “fast” algorithms such as LZRW1 or LZOP usually obtain 200-500MB/s decompression throughput on our evaluation platform (a 2.0GHz Opteron processor). However, we aim for 2-6GB/s.

Let us first motivate the need for such speed with the following simple model (all bandwidths in GB/s):

B	= I/O bandwidth
r	= compression ratio
Q	= query bandwidth
C	= decompression bandwidth
R	= result tuple bandwidth

Our goal with compression is to make queries that are I/O bound (i.e. $Q > B$) faster:

$$R = \begin{cases} Br & : \quad \frac{Br}{C} + \frac{Br}{Q} \leqslant 1 \qquad \text{(I/O bound)} \\ \frac{QC}{Q+C} & : \quad \frac{Br}{C} + \frac{Br}{Q} \geqslant 1 \qquad \text{(CPU bound)} \end{cases} \tag{6.1}$$

Many datasets in e.g. data warehouses and information retrieval systems can be compressed considerably [GS91, GRS98]. Section 6.3 shows that even the synthetic TPC-H dataset, with its uniform distributions, allows for good compression ratios. With these ratios, we often have $B < Q < Br$, such that the query becomes CPU bound using compression. Also, modern high-end RAID controllers deliver $B > 0.6$GB/s [App06], so with $r = 4$ one needs $C = 2.4$GB/s just to keep up with that. As we desire to spend only a minority of CPU time on decompression, we need $C = 4.8$GB/s to keep overhead to 50% of CPU time, and $C = 12$GB/s to get it down to 20%. Since disk bandwidth is typically shared among multiple CPU cores, these numbers motivate our goal of $C = 2-6$GB/s.

We must point out that achieving such high decompression bandwidth is hard. If we assume the decoded values to be 64-bit integers, e.g. $C = 3$GB/s means that 400M integers must be decoded per second, such that we can spend at most *five cycles per tuple* on our 2.0GHz machine! This motivates our interest in getting high IPC out of modern CPUs.

6.2.1 Design guidelines

Our approach to super-scalar data (de)compression is similar to that of our CPU-efficient arithmetic primitives of MonetDB/X100, namely to create *vectorized* compression and decompression algorithms that follow these guidelines:

1. (small) arrays of values should be compressed/decompressed in a tight loop.
2. `if-then-else` inside the loop should be avoided;
3. the loop iterations should be kept independent.

The computational complexity of generic compression algorithms (e.g. LZW) makes it very challenging to adhere to these guidelines, while the new algorithms we propose are specifically designed to meet this challenge.

An additional guideline for our compression strategies is to allow them to work efficiently with the update mechanisms of MonetDB/X100, described in Section 4.3.4. In this architecture, the modifications are stored in in-memory delta structures, similar to differential files [SL76]. The tables on disk are treated as “immutable” objects that are only updated in a batched manner. During the scan, data from disk and delta structures are merged, providing the execution layer with a consistent state. As depicted in Figure 6.1, ColumnBM stores disk pages in a compressed form and decompresses them just before execution on a per-vector granularity. Thus (de)compression is performed on the boundary between CPU cache and main memory, rather than between main memory and disk, saving both cache misses and allowing more data to be cached in RAM. This approach nicely fits the delta-based update mechanism, as merging the deltas can be applied after decompression, and blocks (see 6.2.3) need to be re-compressed only periodically.

6.2.2 PFOR, PFOR-DELTA and PDICT

All our compression methods classify input values as either *coded* or *exception* values. Coded values are represented as small integers of arbitrary bit-width b, with $1 \leqslant b \leqslant 24$. The bit-width used for code values is kept constant within a disk block. Exception values are stored in uncompressed form, thus they should be infrequent in order to achieve a good compression ratio.

Our compression schemes are defined as follows:

PFOR Patched Frame-of-Reference: the small integers are positive offsets from a base value. One (possibly negative) base value is used per disk block. Unlike standard FOR, the base value is not necessarily the minimum value in the block, as values below the base can be stored as exceptions.

PFOR-DELTA PFOR on value deltas: it encodes the *differences* between subsequent values in the column. Decompression consists of first applying PFOR and then computing the running sum on the result.

PDICT Patched Dictionary Compression. Integer codes refer to a position in an array of values (the dictionary). Not all values need to be in the dictionary; there can be exceptions. A disk block can contain a new dictionary but can also re-use the dictionary of a previous block.

The microbenchmarks presented throughout this section all compress 64-bit data items into 8 bits codes, but we implemented and tested our algorithms for all (applicable) datatypes and bit-widths b. In general, we found that, (de)compression bandwidth varies proportionally with the compression ratio.

Datasets encountered in practice are often skewed, both in terms of value distribution and frequency distribution. However, the existing FOR and dictionary compression cannot cope well with this. FOR compression needs $\lceil log_2(max - min + 1)\rceil$ bits, and is thus vulnerable to outliers if the data (i.e. value) distribution is skewed. In contrast, our new PFOR stores outliers as exceptions, such that the $[max_{coded}, min_{coded}]$ range is strongly reduced. Similarly, dictionary compression always needs $\lceil log_2(|D|)\rceil$ bits, even if the frequency distribution of the domain D is highly skewed. In PDICT, however, infrequent values become exceptions, such that the size $|D_{coded}|$ of the frequent domain is strongly reduced on skewed frequency distributions.

6.2.3 Disk storage

As discussed in Section 4.3, ColumnBM stores data in blocks, grouped in large chunks. Blocks contain one or more *segments*. In case of column-wise storage, a segment is identical to a block. In case of PAX [ADHS01], a block contains a segment for each column, and all segments in the block contain the same number of values, which implies that these segments may have different byte-sizes (that sum to a number close to the block size).

Uncompressed fixed-width data types are stored in a segment as a simple array of values.[1] Figure 6.3 shows the structure of a *compressed segment* that divides the segment in four *sections*:

- a fixed-size *header* that contains compression-method specific info as well as the sizes and positions of the other sections.
- the *entry point section* that allows for fine-grained tuple access. For every 128 values, it contains an offset to the next exception in the code section, and a corresponding offset in the exception section.
- the *code section* is a forward-growing array with one small integer code for each encoded value. This section takes the majority of the space in the block.
- the *exception section*, growing backwards, stores non-compressed values that could not be encoded into a small integer code.

6.2.4 Decompression

A pre-processing step in decompression is *bit-unpacking*: the transformation of b bits-wide code patterns in the disk block into an array of machine-addressable integers. Symmetrically, we use *bit-packing* as post-processing for compression. These phases take only a moderate fraction of our algorithms cost, thanks to using highly optimized routines that are *loop-unrolled* to handle 32 values each iteration:

```
/* example: routine to unpack 12-bits codes into integers */
void UNPACK12(unsigned int* out, unsigned int *in, int n) {
    for(int i = 0, j = 0; i < n; i += 8, j += 3) {
        out[i + 0] = ((in[j + 0] & 0x00000fff) >>  0);
        out[i + 1] = ((in[j + 0] & 0x00fff000) >> 12);
```

[1]Variable-width data types such as strings are stored in two segments: one byte-array that contains all values concatenated and a segment with integer offsets to their start positions.

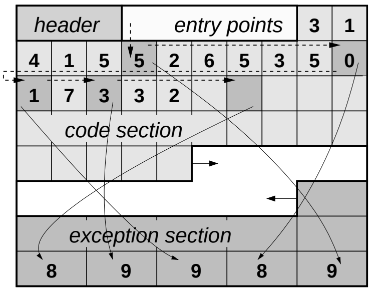


Figure 6.3: Compressed Segment Layout (encoding the digits of π: 31415*9*26535*8979*32 using 3-bit PFOR compression)

```
        out[i + 2] = ((in[j + 0] & 0xff000000) >> 24) | (in[j + 1] & 0x0000000f);
        out[i + 3] = ((in[j + 1] & 0x0000fff0) >>  4);
        out[i + 4] = ((in[j + 1] & 0x0fff0000) >> 16);
        out[i + 5] = ((in[j + 1] & 0xf0000000) >> 28) | (in[j + 2] & 0x000000ff);
        out[i + 6] = ((in[j + 2] & 0x000fff00) >>  8);
        out[i + 7] = ((in[j + 2] & 0xfff00000) >> 20);
    }
}
```

The naive way to implement any decompression scheme that distinguishes between *coded* and *exception* values, is to use a special code (`MAXCODE`) for exceptions, and continuously test for it while decompressing:

```
/* NAIVE approach to decompression */
for (i = j = 0; i < n; i++)
    if (code[i] < MAXCODE)
        output[i] = DECODE(code[i]);
    else
        output[i] = exception[--j]);
```

The above decompression kernel is applicable to both PFOR and PDICT, though the way they encode/decode values differs. In our pseudo code, we abstract from these differences using the following macros: *(i)* `int ENCODE(ANY)`, which transforms an input value into a small integer, and *(ii)* `ANY DECODE(int)`, which produces the encoded input value given a small integer code.

The problem with the NAIVE approach is that it violates our guideline to avoid `if-then-else` in the inner loop. This hinders loop pipelining by the compiler, and also causes branch mispredictions when the else-branch is taken (assuming exceptions are the less likely event). The bottom-left part of Figure 6.4

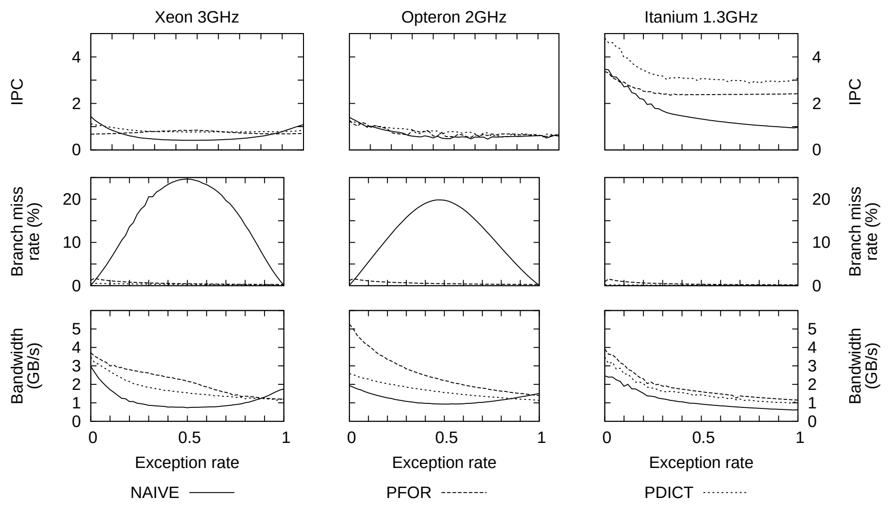


Figure 6.4: Decompression bandwidth, branch miss rate and instructions-per-cycle depending on the exception rate

demonstrates most clearly on Pentium4 how NAIVE decompression throughput rapidly deteriorates as the exception rate gets nearer to 50%. The cause are branch mispredictions[2] on the `if-then-else` test for an exception, that becomes impossible to predict. In the graph on top, we see that the IPC takes a nosedive to 0.5 at that point, showing that branch mispredictions are severely penalized by the 31 stage pipeline of Pentium4.

To avoid this problem, we propose the following alternative "patch" approach:

```
int Decompress<ANY>( int n, int bitwidth,
        ANY   *__restrict__ output,
        void  *__restrict__ input,
        ANY   *__restrict__ exception,
        int   *next_exception)
{
    int next, code[n], cur = *next_exception;

    UNPACK[bitwidth](code, input, n); /* bit-unpack the values */

    /* LOOP1: decode regardless of exceptions */
```

[2]We collected IPC, cache misses, and branch misprediction statistics using *CPU event counters* on all test platforms.

```
    for (int i = 0; i < n; i++) {
        output[i] = DECODE(code[i]);
    }

    /* LOOP2: patch it up */
    for (int i = 1; cur < n; i++, cur = next) {
        next = cur + code[cur] + 1;
        output[cur] = exception[-i];
    }

    *next_exception = cur - n;
    return i;
}
```

Different from the NAIVE method, decompression is now split in two tight loops without any `if-then-else` statements, which both can be efficiently optimized by a compiler and executed on superscalar CPUs.

Figure 6.3, depicting the integer sequence of π stored using 3-bit PFOR with $min_{coded} = 0$, shows that all exception values (i.e. digits $\geqslant 8$) use their code value to store an offset to the next exception, forming a linked list.

The first loop simply decodes all values, which will generate wrong values for the exceptions. The second loop then *patches up* the incorrect values by walking the linked exception list and copying the exception values into the output array. The idea of patching rather than escaping exception values is central to our new algorithms, hence the "P" in their name derives from it.

Following the linked list during patching violates our guideline that one iteration should be independent of the previous one. Iterating the list poses a data hazard to the CPU, however, and not a control hazard, such that it is not very expensive. Moreover, the second loop processes only a small percentage of values, and the data it updates is in the CPU cache. This makes its overhead easily amortized by the performance improvement of the first loop.

The results in Figure 6.4 show that the performance of our patching algorithms decreases monotonically with increasing exception rates. Contrary to the NAIVE approach, decompression bandwidth degrades roughly proportionally with the compression ratio, or the size of the compressed data, as one would expect. The relatively flat IPC lines suggest that the overhead of the data dependency in `LOOP2` is negligible with respect to the increase in memory traffic.

This does not hold for the NAIVE kernel, for which on Pentium4 and Opteron we observe a clear increase in decompression bandwidth towards an exception rate of one. This suggests that its performance is not determined by the size of the compressed data, but by branch mispredictions in the CPU, as both decompression bandwidth and IPC follow the inverse of the bell-shaped branch misprediction curve.

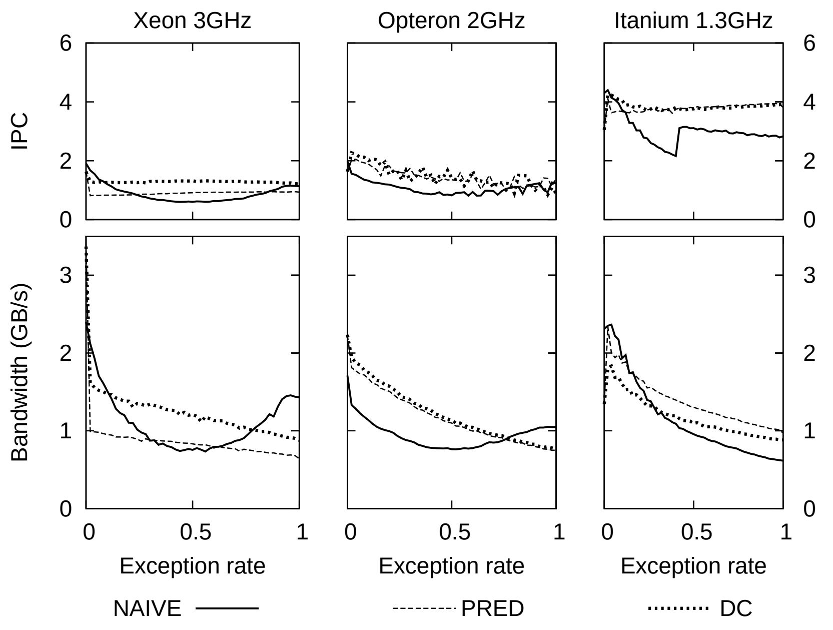


Figure 6.5: PFOR compression bandwidth as a function of exception rate, using an `if-then-else` (NAIVE), predication (PRED) and double-cursor predication (DC)

On Itanium2, the branch mispredictions are avoided thanks to branch predication explained in Section 2.1.5.2. As a result, the performance of the NAIVE kernel closely tracks that of PFOR and PDICT, as presented in the rightmost graph in Figure 6.4. Overall, the patching schemes are clearly to be preferred over the NAIVE approach, as they are faster on all tested architectures.

6.2.5 Compression

Previous database compression work mainly focuses on decompression performance, and views compression as a one-time investment that is amortized by repeated use of the compressed data. This is caused by the low throughput of compression, often an order of magnitude slower than decompression (see Figure 6.2), such that compression bandwidth is clearly lower than I/O write bandwidth. In contrast, our super-scalar compression *can* be used to accelerate I/O bound data materialization tasks. In OLAP and data mining environments, such materialization happens quite frequently for sorting, ad-hoc joins that require partitioning, or (view) materialization of intermediate results that are re-used by a subsequent query batch. Efficient compression is also important for re-compression of data chunks occurring in case of updates. Note that I/O write bandwidth tends to be considerably lower than read bandwidth, especially on RAID devices with mirroring. Therefore, the design goal of compression throughput can be lower than for decompression, e.g. 1-2GB/s. The bottom graphs in Figure 6.5 show that PFOR compression meets this target on all our test platforms.

To achieve such high throughput, we again use the principle of avoiding `if-then-else` in the inner loop. The first loop uses a temporary array `miss` to make a list of exception positions. The second loop constructs the linked patch list and copies the exception values.

```
int Compress<ANY>( int n, int bitwidth,
        ANY  *__restrict__ input,
        void *__restrict__ code,
        ANY  *__restrict__ exception,
        int  *lastpatch)
{
    int miss[N], data[N], prev = *lastpatch;

    /* LOOP1: find exceptions */
    for (int i = 0, j = 0; i < n; i++) {
        int val = ENCODE(input[i]);
        data[i] = val;
        miss[j] = i;
        j += (val > MAXCODE);
    }

    /* LOOP2: create patchlist */
    for (int i = 0; i < j; i++) {
        int cur = miss[i];
        exception[-1 - i] = input[cur];
        data[prev] = (cur - prev) - 1;
        prev = cur;
    }

    PACK[bitwidth](code, data, n); /* bit-pack the values */
    *lastpatch = prev;
    return j; /* #exceptions */
}
```

Appending a position to the `miss` list without `if-then-else` uses a technique similar to the one used in [Ros02] for selection computation: the current position is always copied to the end of the list, and the list pointer is incremented with a Boolean value. This technique transforms a control dependency into a data dependency, which is more efficient. Still, the presence of a data dependency on the variable j in the first, performance-critical loop, violates our guideline that iterations should be independent. Data dependencies cause delay slots in the CPU pipeline. The left-upper graph of Figure 6.5 shows that Pentium4 has an IPC of < 1. We can try to improve IPC by offering it more independent work using a technique called *double-cursor*. It runs two cursors through the to-be-encoded values, one from the start, and one from halfway. Two independent miss lists are used to detect exceptions, processed one after the other in the sequel (omitted):

```
/* LOOP1a: find exceptions */
int m = n / 2;
for(int i = 0, j_0 = 0, j_m = 0; i < m; i++) {
    int val_0 = ENCODE(input[i + 0]);
    int val_m = ENCODE(input[i + m]);
    code[i + 0] = val_0;
    code[i + m] = val_m;
    miss_0[j_0] = i + 0;
    miss_m[j_m] = i + m;
    j_0 += (val_0 > MAXCODE);
    j_m += (val_m > MAXCODE);
}
```

Double-cursor is not the same as loop-unrolling, and cannot be introduced automatically by the compiler.

Figure 6.5 shows that double-cursor significantly improves the IPC and throughput of PFOR on Pentium4, while it behaves the same as single-cursor PFOR on Opteron. On Itanium, where single-cursor already achieved a very high IPC (4), performance degrades somewhat. As the gains on Pentium4, which is also the more prevalent, outweigh the loses on Itanium, double-cursor can be considered the overall winner.

6.2.6 Fine-grained access

While we anticipate that most performance-intensive queries will decompress *all* values in a compressed segment sequentially, some queries may perform random value accesses. A random lookup in the buffer manager will likely cause a CPU cache miss, so if decompression overhead stays in the same ballpark as DRAM access (i.e. 150-400 CPU cycles per cache miss), we deem it efficient enough.

It is easy to randomly access the `code` section at any position x, but we should also know whether position x is an exception and if so, where in the exception section the real value is stored. For this purpose, the *entry point* *section* keeps a pointer to the next exception, as well as its position in the exception section, for each position that is an exact multiple of 128. Each entry point, stored once every 128 values, is a combination of a 7-bits patch `start_list` and a 25-bits `start_exception`, hence the storage overhead of fine-grained access is $32/128 = 0.25$ bits per value. Note that 25-bits exception codes limit our segments to a maximum of 32MB, which is more than sufficient for now to obtain high sequential bandwidth on any RAID system. We can obtain the value at position x in the block, as follows:

```
ANY finegrained_decompress(int position,
        int*__restrict__ code,
        ANY*__restrict__ exception,
        entry_t*__restrict__ entry)
{
    int i = entry[position >> 7].start_list + position & ~127;
    int j = entry[position >> 7].start_exception;
    while(i < x) {
        i += code[i];
        j--;
    }
    return (i == x) ? exception[j] : DECODE(code[x]);
}
```

This tight pipelinable loop that walks the linked list takes 8, 9 and 11 cycles per iteration on respectively the Opteron, Itanium2 and Pentium4 CPUs. Even in the worst realistic case of 30% exceptions, it thus takes on average only a limited ($128 * 0.3/2 = 19$) number of iterations on average, such that random access decoding takes around 200 CPU work cycles per value.

In case of PFOR-DELTA, we must also store the current running total for each entry point. Sticking with 64-bit integers, this induces an additional storage overhead of 0.75 bit per value. Also, fine-grained PFOR-DELTA access requires decompressing a vector of 128 values (which usually causes one cache miss in both the code and exception sections, bringing memory access cost to 300-800 cycles). Since our decompression algorithms typically spend between 3-6 cycles per value, uncompressing 128 values is in the same order of cost.

6.2.7 Compulsory exceptions

A complication of patching is that the compressed integer codes only have a range from [0,2^b-1]; hence the maximum distance between elements in the linked list of exceptions is 2^b. If gaps exceeding this distance occur, so-called *compulsory exceptions* must be introduced. A compulsory exception is a value that can be

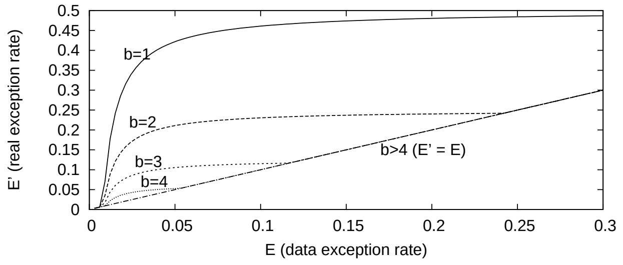


Figure 6.6: Impact of the compulsory exceptions on the real exception rate E' for $b \leqslant 4$

compressed but is represented as an exception anyway, just in order to use its code value to keep the exception list connected.

We do not always have to insert compulsory exceptions if the gap is larger than 2^b though. Each *entry point* starts a new exception list, and these lists need not be connected to each other. Thus, gaps between exceptions at the start and end of each 128-value sequence never need compulsory exceptions. This effectively reduces the area in the code section that must be "covered" by a linked exception list per 128 values by $1/E$, where E is the exception rate caused by the data distribution. From this, we can compute E', which is the effective exception rate after taking into account compulsory exceptions as $E' = MAX(E, \frac{128E-1}{128E}2^{-b})$. Figure 6.6 shows that with bit-width $b = 1$ for miss rates $E > 0.01$, the effective exception rate E' quickly increases to a rather useless 0.47. With $b = 2$, it goes to an already more usable $E' = 0.22$, while for all bit-widths $b > 4$, the effect of compulsory exceptions is negligible.

6.2.8 RAM-RAM vs. RAM-cache decompression

Figure 6.7 presents the results of a micro-benchmark conducted to evaluate our choice for fine-grained, into-cache decompression, as opposed to decompression on the granularity of disk pages. Into-cache decompression is achieved by decompressing a page on a per-vector basis, always storing the result in the same cache-resident result vector, overwriting any previous results. In the page-wise approach, the full, uncompressed page is materialized in RAM.

Results show that RAM-cache decompression is much more efficient than RAM-RAM decompression. Performance of the former approach degrades with the exception rate, and thus the size of the compressed data. The flat shape

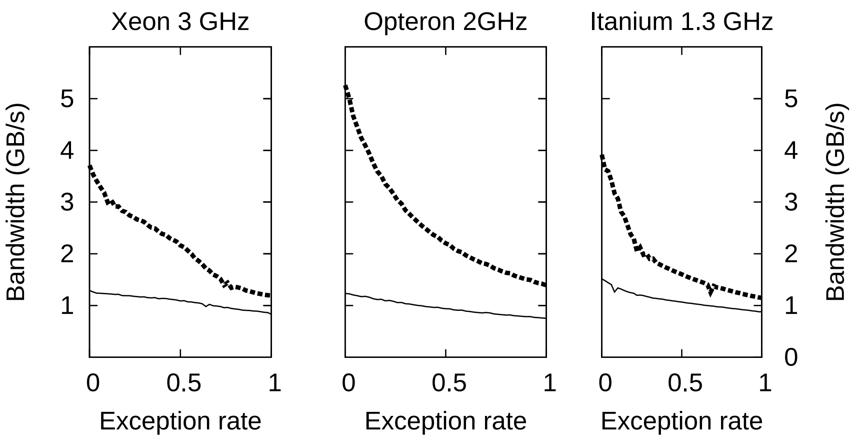


Figure 6.7: RAM-RAM (thin) versus RAM-cache PFOR decompression (thick)

of the latter approach suggests that performance is constrained by the need to materialize the uncompressed result, which is always constant in size.

Another benefit of the RAM-cache approach is that the cache-resident result vector can be fed directly into an operator pipeline. In the RAM-RAM approach, the uncompressed page needs to be read back into the CPU, presenting an additional overhead which is not even incorporated in the RAM-RAM results from Figure 6.7.

6.2.9 Improving memory bandwidth on multi-core CPUs

Algorithms presented in this chapter were originally designed to improve the delivery speeds for disk-resident data. However, disk is not the only media where bandwidth becomes the bottleneck. Even in main memory, with high data consumption rates, as is the case in MonetDB/X100, there are cases when RAM bandwidth limits the performance. This is becoming especially important with the recent popularity of multi-core CPUs (see Section 2.1.7.2). There, memory bandwidth needs to be shared among multiple cores, and with the number of cores continuously increasing, providing enough bandwidth for each core becomes a problem. To demonstrate it, we have performed an experiment in which we run two versions of the TPC-H Query 6 on a dual-chip Core2 Quad 2GHz machine (8 cores in total), using both uncompressed and compressed table rep-

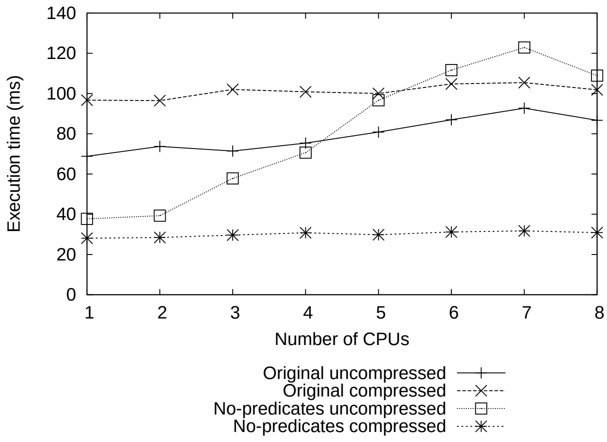


Figure 6.8: Performance of two versions of TPC-H Query 6 (with and without predicates), on a dual-chip Core2 Quad 2GHz (8 cores in total), using uncompressed and compressed table representation

resentations. The first version (A) is the original query, scanning 4 columns, the second (B) is the query with all the predicates removed, scanning only two columns. We used scale-factor 1, making the data fully RAM-resident. In this setup, the original 4 columns occupy 144MB uncompressed and 37.5MB compressed (4-byte wide l_shipdate and l_quantity and 8-byte wide l_extendedprice and l_discount). The two columns used in version B are 96MB and 21MB, respectively. The original Query 6 is as follows:

```
SELECT sum(l_extendedprice * l_discount) AS revenue
FROM lineitem
-- The following predicates are only present in version A.
WHERE l_shipdate >= date '1994-01-01'
  AND l_shipdate < date '1995-01-01'
  AND l_quantity < 24;
  AND l_discount BETWEEN 0.05 and 0.07
```

Figure 6.8 presents the results for benchmarks in which we run from 1 to 8 parallel streams of 300 queries, and measure the average query time of the middle 100 queries. For the original query, the uncompressed version always beats the compressed one. This is because the used combination of predicates selects only ca. 2% of the original data, and the compressed run always decompresses all the values, even the ones not used in the actual computation. Also, the impact of performance on multiple cores is relatively small in the original case without compression. This is again related to the low selectivity of predicates: only a part of the data is actually touched by the query. Also, in this query, the memory-intensive primitives are overlapped with primitives reading the data from the CPU cache, allowing better use of the bus.

The situation changes dramatically when we look at the version B, which does not include predicates and really consumes a full 96MB. Profiling shows that, in the uncompressed case, the majority of time is spent in the memory-intensive multiplication primitive, computing `l_extendedprice * l_discount`. This computation for 6 millions tuples takes on average 33ms. Since the primitive consumes 16 bytes for each tuple, its bandwidth requirement is almost 3 GB/s (per core), which is too high for the memory infrastructure. In the compressed case, most of the time is spent in the decompression phase, which takes 16 ms, resulting in the RAM consumption rate of 1.7 GB/s, and the decompression rate of 6 GB/s. The uncompressed data is in the CPU cache, and the following multiplication primitive only takes 8ms (12 GB/s data input, only available in the CPU cache). This allows the compressed version to improve the performance even on a single core. With more cores, the no-predicate version suffers even more from the insufficient memory bandwidth, making the improvement thanks to the data compression as high as 400%.

This experiment demonstrates that date compression can significantly improve query performance by reducing the main-memory bandwidth requirements. With future CPU generations having dozens, if not hundreds of cores, compressing data in RAM might become more and more important.

6.2.10 Choosing compression schemes

The table materialization operator in MonetDB/X100 should automatically decide which compression method to use for each disk block, and with what parameters. The idea is to first gather a sample (e.g. s=64K values) and look for the best settings for all applicable schemes. For numeric data types (e.g. integers, decimals) all three schemes apply. Otherwise, only PDICT is usable.[3]

When a column is being compressed, the compression ratio can be easily monitored at the granularity of a disk block or chunk. When it strongly deteriorates, we could re-run the compression mode analysis to adapt the parameters for the next block or even choose another compression scheme. The complexity

[3]In the near future, we plan to add new super-scalar compression algorithms targeted at floating point data and text.

of choosing a compression mode is $O(s \log s)$ to the size of the sample s, because it must be sorted as a preprocessing step. We now discuss for each method, how the optimal parameters are found using the sorted sample.

In PFOR, we can determine in one pass through the sorted sample where the longest stretch of values starts, such that the difference between first and last is representable in `b` bits.

```
PFOR_ANALYZE_BITS(int n, ANY *V, int b) {
    int len = 0, min = 0, range = 1 << b;

    for (int lo = 0, hi = 0; hi < n; hi++)
        if (V[hi] - V[lo] >= range) {
            if (hi - lo > len) {
                min = lo; len = hi-lo;
            }
        while (V[hi] - V[lo] >= range)
            lo++;
        }
    return (min, len + 1);
}
```

We simply invoke this function for all relevant bit-widths b and choose the setting that yields best compression, i.e. $1 \leqslant b < 8 * \texttt{sizeof}(\texttt{V})$ where $b + E_{PFOR(b)} * 8 * \texttt{sizeof}(\texttt{V})$ is minimal. In this equation the exception rate $E_{PFOR(b)} = \frac{s-len_b}{s}$, where len_b is returned by the above function, when invoked on the sample with parameter b.

The parameters for PFOR-DELTA are derived by running this same algorithm on the sorted differences of the sample.

For PDICT, we use once again the sorted sample, to create a (smaller) frequency histogram h, which we re-sort descending on frequency. PDICT will encode the first (i.e. largest) 2^b buckets of this histogram such that the exception rate $E_{PDICT(b)} = 1 - \sum_{i=1}^{2^b} \frac{h[i]}{s}$. Again, by trying all relevant settings of b, we can quickly determine the b that yields the highest compression rate. The first 2^b values from the histogram are subsequently used to create a super-scalar *perfect hash function* (*memorized probing* in [Hem05]) that is used during PDICT compression to compute the integer codes for values that must be compressed. In all, it achieves PDICT compression bandwidth of $> 1GB/s$ on all our three test platforms.

6.3 TPC-H experiments

Table 6.1 shows the performance of compression algorithms in MonetDB/X100 running the TPC-H benchmark [Tra06] with scale factor 100 on two different hardware platforms. Low-end servers are represented by an Opteron 2GHz machine with a 4-disk RAID system delivering around 80 GB/s. The example of a middle-end system is a Pentium4 machine with 12-disk RAID delivering around 350GB/s. Both machines are dual CPU systems with 4GB memory, but MonetDB/X100 currently uses only one CPU.

We used the same data clustering and index structures as in the previous in-memory MonetDB/X100 TPC-H SF-100 experiments [BZN05]. Only a subset of TPC-H queries is presented, since the X100 execution layer currently misses some of the features necessary to run the remaining ones in a disk-based scenario.

While ColumnBM by default uses the DSM storage model [CK85], we also present the results for PAX storage [ADHS01]. I/O-wise they are comparable to an NSM system running DB2, for which the last column lists the official TPC-H scores. This system uses eight Pentium4 Xeon CPUs (2.8GHz), 16GB RAM and 142 SCSI disks. Thus, while the CPU used is roughly equivalent to our middle-end server, it has 4-12x more hardware resources across the board.

The TPC-H data was compressed using PFOR, PFOR-DELTA and PDICT (enum) compression schemes. The second and third columns of Table 6.1 show the compression ratios achieved per query. Note that since the "comment" fields could not be compressed with our algorithms, the PAX queries achieve significantly lower compression ratios. Columns 4 and 10 show that in most cases we reach our decompression speed target of > 2 GB/s.

On the Opteron system, the speedup for most of the DSM queries is in line with the compression ratio. As the left-most side of Figure 6.9 shows, this is related to the fact that the low-end disk system makes the queries I/O-bound even with compression. The middle part of Figure 6.9 shows that on the Pentium4 system with a faster RAID the situation is different with much higher CPU usage in the uncompressed case. As a result, after increasing the perceived I/O bandwidth with decompression, all the queries become CPU-bound, such that the performance gain is less than the compression ratio. With the PAX storage model and its increased I/O requirements, the CPU processing impact is reduced again, resulting in better speedups than in the DSM case.

We also implemented the possibility to perform full-page decompression in ColumnBM, as described in Section 6.2.8. Column 7 of Table 6.1 shows that such decompression from memory into memory is significantly slower than the fine-grained decompression between RAM and the CPU Cache, presented in col-

TPC-H query	compression ratio		Pentium4 Xeon 3GHz 12 disks 4GB RAM					
	DSM	PAX	dec.speed	DSM			PAX	
			MB/sec	unc.	$M \Rightarrow C$	$M \Rightarrow M$	unc.	$M \Rightarrow C$
1	2	3	4	5	6	7	8	9
01	4.33	2.30	4502	65.9	50.9	63.8	265.0	103.0
03	3.04	1.66	2306	8.9	6.0	7.1	45.5	27.0
04	8.15	1.82	3709	4.8	1.8	2.3	30.2	16.5
05	3.81	2.24	2421	17.2	16.2	16.7	81.2	36.7
06	4.39	2.25	2200	10.8	4.6	6.1	51.0	22.5
07	1.71	2.01	1457	34.4	40.8	48.3	158.0	76.5
11	2.14	1.08	4084	18.8	18.5	19.4	38.8	35.6
14	1.91	1.94	3688	5.8	4.9	5.4	22.1	11.5
15	2.70	2.13	2584	30.3	31.2	31.3	49.6	40.0
18	3.56	2.75	4315	38.9	13.6	21.3	419.9	151.5
21	4.11	2.12	2600	43.2	24.2	32.1	338.0	157.6

TPC-H query	compression ratio		Opteron 2GHz 4 disks 4GB RAM					8 x P4 Xeon 2.8GHz 142 disks 16GB RAM
	DSM	PAX	dec.speed	DSM		PAX		IBM DB2
			MB/sec	unc.	$M \Rightarrow C$	unc.	$M \Rightarrow C$	UDB 8.1
1	2	3	10	11	12	13	14	15
01	4.33	2.30	3736	307.2	69.6	1098.9	480.3	111.9
03	3.04	1.66	2546	35.0	11.3	183.5	113.6	15.1
04	8.15	1.82	3018	18.2	2.4	115.5	65.9	12.5
05	3.81	2.24	2119	54.3	15.3	300.1	155.9	84.0
06	4.39	2.25	2031	48.2	10.7	232.7	104.3	17.1
07	1.71	2.01	1251	119.8	72.0	614.2	349.4	86.5
11	2.14	1.08	3225	27.0	14.6	180.9	162.2	19.5
14	1.91	1.94	2888	23.7	12.2	90.6	46.9	10.9
15	2.70	2.13	2464	44.9	22.4	209.8	97.1	21.6
18	3.56	2.75	3833	181.9	50.6	1379.7	704.9	318.2
21	4.11	2.12	2520	197.6	46.6	1423.5	759.2	374.9

Legend: *unc.* – uncompressed data,
$M \Rightarrow C$ – memory-to-cache decompression, $M \Rightarrow M$ – memory-to-memory decompression

Table 6.1: TPC-H SF-100 experiments on MonetDB/X100 (except DB2 results, taken from `www.tpc.org`)

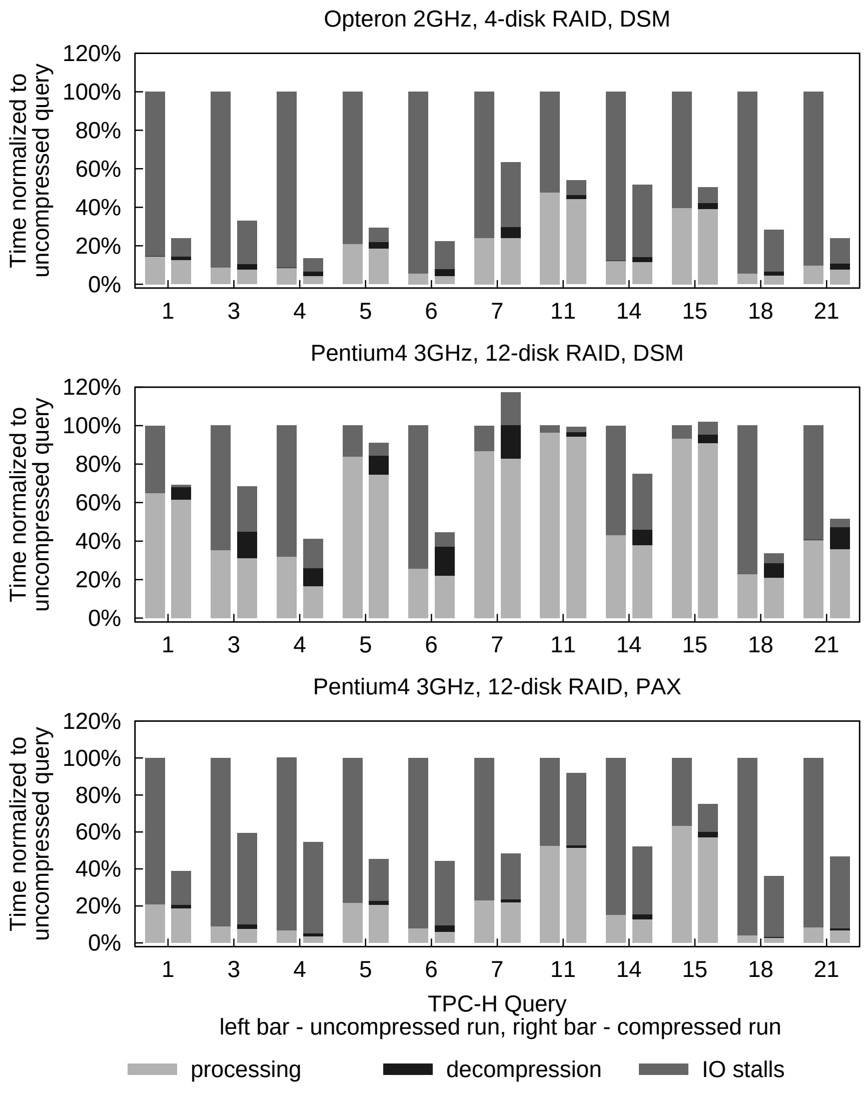


Figure 6.9: Profiling details for the TPC-H SF-100 results from Table 6.1

	PFOR-DELTA			**carryover-12**			**shuff**		
	comp ratio	comp MB/s	dec MB/s	comp ratio	comp MB/s	dec MB/s	comp ratio	comp MB/s	dec MB/s
INEX	1.75	679	3053	2.12	49	524	2.45	3.5	82
TREC fbis	3.47	788	3911	4.26	98	740	5.11	190	164
TREC fr94	3.12	682	3196	3.49	84	689	4.65	149	154
TREC ft	3.13	761	3443	3.47	84	704	4.89	178	157
TREC latimes	2.99	742	3289	3.30	79	683	4.61	164	153

Table 6.2: PFOR-DELTA on Inverted Files

umn 6. The main reason for this is the high-cost of in-memory materialization of decompressed data. Another interesting feature of our fine-grained decompression can be observed in Figure 6.9, where the processing in the compressed case is slightly faster. This is caused by the fact that the main-memory access is performed by the decompression routines, and the query execution layer reads the data directly from the CPU cache.

6.4 Inverted file compression

Compression of *inverted files* to improve I/O bandwidth and sometimes latency (due to reduced seek distances on the compressed file) is important for the performance of information retrieval systems [WMB99]. In this area, there is a trend to use lightweight-compression schemes rather than the classical storage-optimal schemes [Tro03, AM05].

We evaluated the performance of PFOR-DELTA with respect to both compression ratio and speed on inverted file data derived from the INEX and TREC document collections, and compared it with the implementation of the recently proposed *carryover-12* compression scheme [AM05], which was designed for high decompression speeds. Furthermore, performance was compared to that of a semi-static Huffman coder, which is commonly used for inverted file compression. Table 6.2 summarizes the results on our 3GHz Pentium 4 machine, and shows that PFOR-DELTA improves decompression bandwidth of *carryover-12* 6.5 times, while only reducing the compression ratio by 15%.

To verify the need for such decompression speeds, we measured the raw query bandwidth of a typical retrieval query that looks up the top-N documents in which a given term from the TREC fbis dataset occurs most frequently (a merge-join of the postings table with the document offsets, followed by ordered aggregation and heap-based top-N). Within our MonetDB/X100 system, this query was able to process a list of d-gaps at 580MB/s, which implies that even on our 350MB/s RAID system it would remain I/O-bound. Using equation 6.1 to compute the decompression bandwidth C that achieves an equilibrium between CPU time spent on query processing and decompression, yields $\frac{580 \times C}{580 + C} = 350$, which leads to $C = 883$MB/s. Table 6.2 shows that decompression bandwidths from *shuff* and even *carryover-12* are below this point, hence only make the query slower, while PFOR-DELTA accelerates it from 350MB/s to 504MB/s.

The benefit of using PFOR-DELTA on IR tasks has been evaluated with Terabyte-TREC experiments discussed in Section 8.2.2. Additionally, further research by Zhang et al [ZLS08] has demonstrated that this technique is competitive against a large class of other methods, and the ideas behind it can also be used to improve other compression algorithms.

6.5 Conclusions and future work

This chapter presented our work on using data compression to scale the high performance of MonetDB/X100 engine to disk-based datasets. We proposed a new set of *super-scalar* compression algorithms. Their "patching" approach allows these algorithms to handle outliers gracefully while still exploit the pipelined features of modern CPUs. Additionally, we introduced the idea of decompressing between RAM and the CPU Cache, rather than the common idea to apply it between I/O and RAM. Our results show that this not only allows the buffer manager to store more (compressed) data, but is also faster to (de)compress. As a result, our algorithms provide decompression speeds in the range of $> 2GB/s$. This is an order of magnitude faster than conventional compression algorithms, making decompression almost transparent to query execution. By using these techniques in TPC-H, TREC and INEX datasets, we managed to significantly reduce or completely eliminate the I/O bottleneck. In the future, we plan to extend the applicability of our system by introducing additional compression algorithms specialized for other data types and distributions.

Chapter 7

Cooperative scans

The previous chapter focused on improving the perceived disk bandwidth from the point of view of a single scan-based query. Another possible direction is to exploit the fact that with multiple queries running at the same time in a system, there are opportunities to share the data between them. This is typically achieved through special buffering policies in the storage layers. While there has been a lot of previous research in this area [CD85, SS86], disk scans were mostly considered trivial, and simple LRU or MRU buffering policies were proposed for them [CR93, SS86]. We show that if scans start at different times, these policies achieve only a low amount of buffer reuse. To improve this situation, some systems support the concept of *circular scans* [Col98, Coo01, NCR02, HSA05] which allows queries that start later to *attach* themselves to already active scans. As a result, the disk bandwidth can be shared between the queries, resulting in a reduced number of I/O requests. However, this strategy is not efficient when queries process data at different speeds or a query scans only a range of records instead of a full table.

In this chapter we analyze the performance of existing scan strategies, identifying three basic approaches: *normal*, *attach* and *elevator*. In *normal*, a traditional LRU buffering policy is employed, while in both *attach* and *elevator* incoming queries can join an ongoing scan in case there is overlap in data need, with the main difference that *elevator* employs a single, strictly sequential scan cursor, while *attach* allows for multiple (shared) cursors. Benchmarks show that they provide sharing of disk bandwidth and buffer space only in a limited set of scenarios. This is mostly caused by the fact that the disk access order is predefined when a query enters the system, hence it cannot be adjusted to optimize

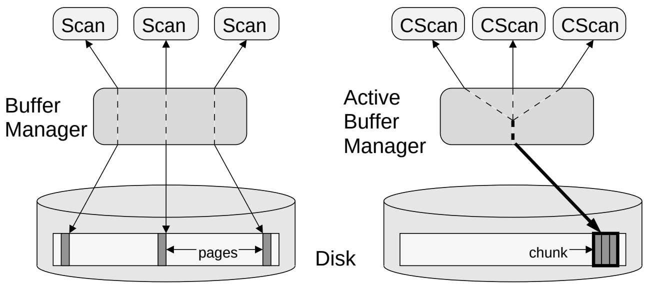


Figure 7.1: Normal scans (left) versus Cooperative Scans (right)

performance in dynamic multi-query scenarios.

To overcome these limitations, we introduce the *Cooperative Scans* framework, depicted in Figure 7.1. It involves `CScan` – a modified (index) `Scan` operator that announces the needed data ranges upfront to an *active buffer manager* (`ABM`). The `ABM` dynamically optimizes the order of disk accesses, taking into account all current `CScan` requests on a relation (or a set of clustered relations). This framework can run the basic *normal*, *attach* and *elevator* policies, but also a new policy, *relevance*, that is central to our proposal. Besides optimizing throughput, the *relevance* policy also minimizes latency. This is done by departing from the strictly sequential access pattern as present in *attach* and elevator. Instead, *relevance* makes page load and eviction decisions based on per-page *relevance functions*, which, for example, try to evict pages with a low number of interested queries as soon as possible, while prioritizing page reads for short queries and pages that have many interested queries.

To further illustrate the need for a more flexible approach to I/O scheduling, consider the following example. Assume that a system has to execute two queries, Q_1 and Q_2, which enter the system at the same time and process data at the same speed. Q_1 needs to read 30 pages and is scheduled first, while Q_2 needs 10 different pages. If those queries get serviced in a round-robin fashion, as in the *normal* policy, Q_2 finishes after 20 pages are loaded, and Q_1 after 40, giving an average query latency of 30. The *elevator* policy may perform better, by first fully servicing Q_2 and then Q_1, reducing the average waiting time from 30 to 25. Still, *elevator* can choose the opposite order, resulting in waiting times of 30 and 40, hence actually increasing the average time. With *relevance*, we aim to get close to the optimal average query latency, without relying on the sequential scan order, by making flexible I/O scheduling decisions.

The outline of this chapter is as follows. First, Section 7.1 analyzes existing approaches to scan processing. In Section 7.2 we introduce the Cooperative Scans framework for row stores and we validate its performance in Section 7.3. In Section 7.4 we extend Cooperative Scans to column stores. The incorporation of ABM into an existing DBMS is discussed in Section 7.5, where we also explore possibilities of adapting order-aware query processing operators to handle out-of-order data delivery. We discuss related work in Section 7.6 before concluding in Section 7.7.

7.1 Traditional scan processing

With multiple scans running concurrently, in a naive implementation sequential requests from different queries can interleave, causing frequent disk-arm movements and resulting in a semi-random access pattern and low overall disk throughput. To avoid this problem, most database systems execute such scans using large isolated I/O requests spanning over multiple pages, together with physical clustering of table pages. As a result, the overhead of shifting the disk-arm is amortized over a large chunk of data, resulting in an overall bandwidth comparable to a standalone scan.

Even when using bandwidth-efficient chunk-based I/O, different scheduling policies are used for concurrent scans. The most naive, called ***normal*** in the rest of this chapter, performs scans by simply reading all disk blocks requested by a query in a sequential fashion, using an LRU policy for buffering. The disadvantage of LRU is that if one query starts too long after the other, the loaded pages will already be swapped out before they can be reused. As a result, assuming there is no buffer reuse between the queries, and queries are serviced in a round-robin fashion, the expected number of I/Os performed in the system until a new query Q_{new} that reads C_{new} chunks finishes can be estimated by: $C_{new} + \sum_{q \in queries} MIN(C_{new}, C_q)$.

The major drawback of the *normal* policy is that it does not try to reuse data shared by different running queries. In a dynamic environment, with multiple partial scans running at the same time, it is likely that the buffer pool contains some data that is useful for a given query. With a table consisting of C_T chunks, a query that needs C_Q chunks and a buffer pool of C_B chunks, the probability of finding some useful data in the randomly-filled buffer is:

$$P_{reuse} = 1 - \prod_{i=0}^{C_B-1} \frac{C_T - C_Q - i}{C_T - i} \qquad (7.1)$$

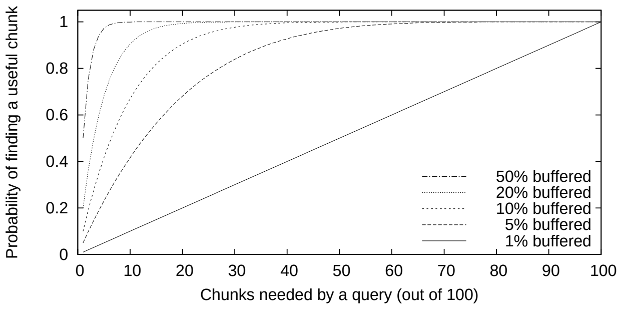


Figure 7.2: Probability of finding a useful chunk in a randomly-filled buffer pool, with varying buffer pool size and query demand

As Figure 7.2 shows, even for small scanned ranges and buffer sizes, this probability can be high, e.g. over 50% for a 10% scan with a buffer pool holding 10% of the relation. Unfortunately, the *normal* policy, by enforcing a sequential order of data delivery, at a given time can use only a single page, reducing this probability to C_B/C_T.

In many cases it is possible to relax the requirement of sequential data delivery, imposed by *normal*. Even when using a clustered index for attribute selection, consuming operators often do not need data in a particular order. This allows for scheduling policies with “out-of-order” data delivery.

A simple idea of sharing disk access between the overlapping queries is used in the ***attach*** strategy. When a query Q_{new} enters the system, it looks at all other running scans, and if one of them (Q_{old}) is overlapping, it starts to read data at the current Q_{old}'s position. To optimize performance, *attach* should choose a query that has the largest remaining overlap with Q_{new}. Once Q_{new} reaches the end of its desired range, it starts from the beginning until reaching the original position. This policy, also known as “circular scans” or “shared scans”, is used among others in Microsoft SQLServer [Coo01], RedBrick [Col98], and Teradata [NCR02], and allows significant performance improvement in many scenarios. The *attach* policy, however, may suffer from three problems. First, if one query moves much faster than the other, the gap between them may become so large that pages read by the fastest query are swapped out before the slower reaches them (they “detach”). Second, if queries are range scans, it is possible that one of the queries that process data together finishes, and the other continues by itself, even though it could attach to another running query. Also, if a full scan is underway but not yet in its range, *attach* misses this sharing opportunity. Finally, when exploiting per-block meta-data, the scan request can consist of multiple ranges, making it even harder to benefit from sharing a scan with a single query. As a result, the upper bound on the number of I/Os performed by *attach* is the same as in *normal.*

The ***elevator*** policy is a variant of *attach* that addresses its problems by enforcing *strict* sequential reading order of the chunks for the entire system. This optimizes the disk latency and minimizes the number of I/O requests, and thus leads to good disk bandwidth and query throughput. However, the problem here is that query speed degenerates to the speed of the slowest query, because all queries wait for each other. Also, range queries often need to wait a long time before the reading cursor reaches the data that is interesting for them. In principle, in the worst case the number of I/Os performed by a system before a fresh query Q_{new} finishes can be $MIN(C_T, C_{new} + \sum_{q \in queries} C_q)$, where C_T is the number of chunks in the entire table.

7.2 Cooperative Scans

The analysis in the previous section, further confirmed by results in Section 7.3, demonstrates that existing scan-processing solutions that try to improve over the *normal* policy still suffer from multiple inefficiencies. In this section we propose a new "Cooperative Scans" framework that avoids these problems. As Figure 7.1 presents, it consists of a cooperative variant of the traditional (index) `Scan` operator, named `CScan`, and an Active Buffer Manager (`ABM`).

The new **CScan** operator registers itself as an *active scan* on a range or a set of ranges from a table or a clustered index. `CScan` has the same interface as the normal `Scan` operator, but it is willing to accept that data may come in a different order. Note that some query plans exploit column ordering present on disk. We discuss integration of such queries in our framework in Section 7.5.

The **Active Buffer Manager** (`ABM`) extends the traditional buffer manager in that it keeps track of `CScan` operators and which parts of the table are still needed by each of them, and tries to *schedule* disk reads such that multiple concurrent scans reuse the same pages. The overall goal of `ABM` is to minimize the average query cost, keeping the maximum query execution cost reasonable (i.e. ensuring "fair" treatment of all queries). As discussed in Section 4.3, in MonetDB/X100 scan processing is usually performed with large I/O units we call *chunks*, to achieve good bandwidth with multiple concurrent queries. Note that a chunk in memory does not have to be contiguous, as it can consists of

```
CScan process
selectChunk($q_{trigger}$)
| if finished($q_{trigger}$)
| | return NULL
| else
| | if abmBlocked()
| | | signalQueryAvailable()
| | chunk = chooseAvailableChunk($q_{trigger}$)
| | if (chunk == NULL)
| | | chunk = waitForChunk($q_{trigger}$)
| | return chunk

chooseAvailableChunk($q_{trigger}$)
| $c_{available}$ = NULL, $U = 0$
| foreach $c$ in interestingChunks($q_{trigger}$)
| | if chunkReady($c$) and useRelevance($c$) > $U$
| | | $U$ = useRelevance($c$)
| | | $c_{available} = c$
| return $c_{available}$
```

Figure 7.3: Pseudo-code for the Relevance policy: CScan operator

multiple pages filled in with a single *scatter-gather* I/O request. In our framework there are two more reasons for using chunks. First, the number of chunks is usually one or two orders of magnitude smaller than the number of pages, thus it becomes possible to have chunk-level scheduling policies that are considerably more complex than disk-block-level policies. Secondly, it is possible to extend chunks to be logical entities whose boundaries may not even correspond exactly to disk block boundaries, a feature that will be exploited in the more complex scenarios with column-based storage.

In our system, the Cooperative Scans framework implements the traditional scan-processing policies: *normal*, *attach* and *elevator*. However, its main benefit comes from a newly introduced ***relevance*** policy that takes scheduling decisions by using a set of *relevance functions*. Both the `CScan` and `ABM` processes, as well as the relevance functions used by them, are described in Figures 7.3, 7.4 and 7.5, respectively.

As Figure 7.3 illustrates, the `CScan` process is called on behalf of a certain query, $q_{trigger}$, that contains a `CScan` operator in its query plan. Each time $q_{trigger}$ needs a chunk of data to process, *selectChunk* is called. This triggers a search over all buffered chunks that still need to be processed by the query, in *chooseAvailableChunk*, and returns the most relevant one, as governed by

```
ABM process
main()
    while (true)
        $query$ = chooseQueryToProcess()
        if $query$ == NULL
            blockForNextQuery()
            continue
        $chunk$ = chooseChunkToLoad($query$)
        $slot$ = findFreeSlot($query$)
        loadChunk($chunk$, $slot$)
        foreach $q$ in $queries$
            if (chunkInteresting($q$, $chunk$) and queryBlocked($q$)
                signalQuery($q$, $chunk$)

chooseQueryToProcess()
    $relevance = -\infty$, $query$ = NULL
    foreach $q$ in $queries$
        $qr$ = queryRelevance($q$)
        if ($query$ == NULL or $qr > relevance$)
            $relevance = qr$
            $query = q$
    return $query$

chooseChunkToLoad($q_{trigger}$)
    $c_{load}$ = NULL, $L = 0$
    foreach $c$ in interestingChunks($q_{trigger}$)
        if (not chunkReady($c$)) and loadRelevance($c$) $> L$
            $L$ = loadRelevance($c$)
            $c_{load} = c$
    return $c_{load}$

findFreeSlot($q_{trigger}$)
    $s_{evict}$ = NULL, $K = \infty$
    foreach $s$ in $slots$
        if empty($s$)
            return $s$
        $c$ = chunkInSlot($s$)
        if (not currentlyUsed($s$)) and (not interesting($c$, $q_{trigger}$))
          and (not usefulForStarvedQuery($c$))
          and keepRelevance($c$) $< K$
            $K$ = keepRelevance($c$)
            $s_{evict} = s$
    freeSlot($s_{evict}$)
    return $s_{evict}$
```

Figure 7.4: Pseudo-code for the Relevance policy: Active Buffer Manager

```
NSM Relevance Functions
queryRelevance($q$)
| if not queryStarved($q$)
| | return $-\infty$
| return - chunksNeeded($q$) +
| | waitingTime($q$) / runnningQueries()

useRelevance($c$, $q_{trigger}$)
| return $Q_{max}-$ numberInterestedQueries($c$)

loadRelevance($c$)
| return numberInterestedStarvedQueries($c$) * $Q_{max}$
|   + numberInterestedQueries($c$)

keepRelevance($c, q_{trigger}$)
| return numberInterestedAlmostStarvedQueries($c$) * $Q_{max}$
|   + numberInterestedQueries($c$)

queryStarved($q_{trigger}$)
| return numberOfAvailableChunks($q_{trigger}$) $< 2$
```

Figure 7.5: Pseudo-code for the Relevance policy: Relevance functions

useRelevance. If no such chunk is available, the operator blocks until the `ABM` process loads a chunk that is still needed by $q_{trigger}$. Our *useRelevance* function promotes chunks with the smallest number of interested queries. By doing so, the less interesting chunks will be consumed early, making it safe to evict them. This also minimizes the likelihood that less interesting chunks will get evicted before they are consumed.

The `ABM` thread continuously monitors all currently running queries and their data needs. It schedules I/O requests on behalf of the query with the highest priority, considering the current system state. For this query, it chooses the most relevant chunk to load, possibly evicting the least relevant chunk present in the buffer manager. Once a new chunk is loaded into the `ABM`, all blocked queries interested in that chunk are notified. This is the core functionality of `ABM`'s main loop, as found in the middle part of Figure 7.4.

The *chooseQueryToProcess* call is responsible for finding the highest priority query, according to *queryRelevance*, to load a chunk for. This *queryRelevance* function considers non-starved queries (i.e. a queries that have 2 or more available chunks, including the one they are currently processing) equal, assigning them the lowest priority possible. Starved queries are prioritized according to the amount of data they still need, with shorter queries receiving higher priorities. However, to prevent the longer queries from being starved forever, the priorities are adjusted to also promote queries that are already waiting for a long time. By prioritizing short queries, `ABM` tries to avoid situations where chunks are assigned to queries in a round-robin fashion, as this can have a negative impact on query latency. Besides, a chunk loaded for a short query has a higher chance of being useful to some large query than the other way around. In case `ABM` does not find a query to schedule a chunk for, it blocks in *blockForNextQuery*, until the `CScan` operator wakes it up again using *signalQueryAvailable*.

Once `ABM` has found a query to schedule a chunk for, it calls the *chooseChunkToLoad* routine to select a not yet loaded chunk that still needs to be processed by the selected query. The *loadRelevance* function determines which chunk will actually be loaded, not only by looking at what is relevant to the current query, but also taking other queries needs into consideration. To maximize sharing, it promotes chunks that are needed by the highest number of starved queries, while at the same time slightly adjusting priorities to prefer chunks needed by many non-starved queries.

If there are no free slots in the buffer pool, `ABM`'s *findFreeSlot* routine needs to swap out the chunk with the lowest *keepRelevance*. This function is similar to *loadRelevance*, except that when looking at queries, we treat queries on the border of starvation as being starved, to avoid evicting their chunks, which would make them starved, hence schedulable, immediately.

The *relevance* policy tries to maximize buffer pool reuse without slowing down fast queries. Thus, a slow query will re-use some of the chunks loaded by a fast query, skipping over chunks that it was too slow to process. These are read again later in the process, when the fast query might already be gone. The access pattern generated by this approach may be (quasi-) random, but since chunks consist of multiple sequential pages, disk (arm) latency is still well amortized.

7.3 Row-wise experiments

Benchmark system: We carried out row storage experiments using the PAX storage model, which is equivalent to NSM in terms of I/O demand, but better suited for the MonetDB/X100 execution layer. The chunk size used was 16MB, and the ABM buffer-pool size was set to 64 chunks (1GB), unless stated otherwise. Direct I/O was used, to avoid operating system buffering. Our test machine was a dual-CPU AMD Opteron 2GHz system with 4GB of RAM. The

storage facility was a 4-way RAID system delivering slightly over 200 MB/s.
Benchmark dataset: We used the standard TPC-H [Tra06] benchmark data with scale factor 10. In this setting the *lineitem* table consumes over 4GB of disk space. The other tables are fully cached by the system.
Queries: To allow flexible testing of our algorithms we have chosen two queries based on the TPC-H benchmark. Query *FAST* (F) is TPC-H Q6, which is a simple aggregation. Query *SLOW* (S) is TPC-H Q1 with extra arithmetic computations to make it more CPU intensive. For all queries we allow arbitrary scaling of the scanned table range. In this section we use the notation QUERY-PERCENTAGE, with QUERY representing the type of query, and PERCENTAGE the size of the range being scanned. For example, with F-10 we denote query *FAST*, reading 10% of the full relation from a random location. We use multiple query streams, each sequentially executing a random set of queries. There is a 3 second delay between starting the streams, to better simulate queries entering an already-working system.

7.3.1 Comparing scheduling policies

Table 7.1 shows the results for all scheduling policies when running 16 streams of 4 queries. We used a mix of slow and fast queries with selectivity of 1%, 10%, 50% and 100%. The two major system-wide results are the average stream running time, representing the system throughput, and the average normalized latency of a query (running time in this benchmark divided by the base time, when the query runs by itself with an empty buffer), representing the system latency. Additionally we provide the total execution time, CPU-utilization, and the number of issued I/Os. The difference between the total time and the average stream time comes from the random distribution of queries in the streams, resulting in a significant variance of stream running times. For each query type and policy we provide the average latency, normalized latency and number of I/Os issued when scheduling this query type. Additionally, Figure 7.6 presents a detailed analysis of the I/O requests issued by each policy.

As expected, the *normal* policy achieves the worst performance. As Figure 7.6 shows, it maintains multiple concurrent sequential scans, which leads to the largest number of I/O requests and a minimal buffer reuse. Since the query load is relatively CPU-intensive, it still manages to use a significant fraction of the CPU time.

The *attach* policy allows merging requests from some queries. As a result, it consistently improves the performance of all query types and the system throughput. Still, in Figure 7.6 we see that there are multiple (albeit fewer than

System statistics	**Normal**	**Attach**
Avg. stream time	283.72	160.81
Avg. normalized latency	6.42	3.72
Total time	453.06	281.19
CPU use	53.20%	81.31%
I/O requests	4186	2325

Query statistics			**Normal**				**Attach**			
query	count	standalone	latency(sec)		norm.	I/Os	latency(sec)		norm.	I/Os
		cold time	avg	stddev	lat.		avg	stddev	lat.	
F-01	9	0.26	1.71	1.02	6.58	2	1.02	0.49	3.92	2
F-10	7	2.06	13.97	5.69	6.78	23	6.23	2.56	3.02	18
F-50	6	10.72	103.59	14.96	9.66	78	58.77	10.96	5.48	67
F-100	9	20.37	192.82	31.56	9.47	153	96.98	23.33	4.76	69
S-01	13	0.38	1.67	1.25	4.39	2	1.19	0.65	3.13	3
S-10	6	3.55	21.58	5.11	6.08	19	15.12	4.08	4.26	24
S-50	6	17.73	78.23	29.07	4.41	95	46.98	16.82	2.65	79
S-100	8	35.27	179.35	59.04	5.09	177	105.51	33.40	2.99	60

System statistics	**Elevator**	**Relevance**
Avg. stream time	138.41	99.55
Avg. normalized latency	13.52	1.96
Total time	244.45	238.16
CPU use	90.20%	93.94%
I/O requests	1404	1842

Query statistics			**Elevator**				**Relevance**			
query	count	standalone	latency(sec)		norm.	I/Os	latency(sec)		norm.	I/Os
		cold time	avg	stddev	lat.		avg	stddev	lat.	
F-01	9	0.26	5.31	7.33	20.42	-	0.52	0.36	2.00	2
F-10	7	2.06	15.17	8.63	7.36	-	3.30	1.30	1.60	18
F-50	6	10.72	44.87	7.92	4.19	-	18.21	6.64	1.70	43
F-100	9	20.37	59.60	19.57	2.93	-	29.01	8.17	1.42	69
S-01	13	0.38	15.01	15.04	39.50	-	0.55	0.29	1.45	2
S-10	6	3.55	20.29	23.93	5.72	-	11.30	5.98	3.18	22
S-50	6	17.73	37.39	14.23	2.11	-	37.77	15.66	2.13	48
S-100	8	35.27	79.39	24.37	2.25	-	98.71	29.89	2.80	44

Table 7.1: Row-storage experiments (PAX) with a set of FAST and SLOW queries scanning 1%, 10%, 50% and 100% of a table (16 streams of 4 random queries, all times in seconds)

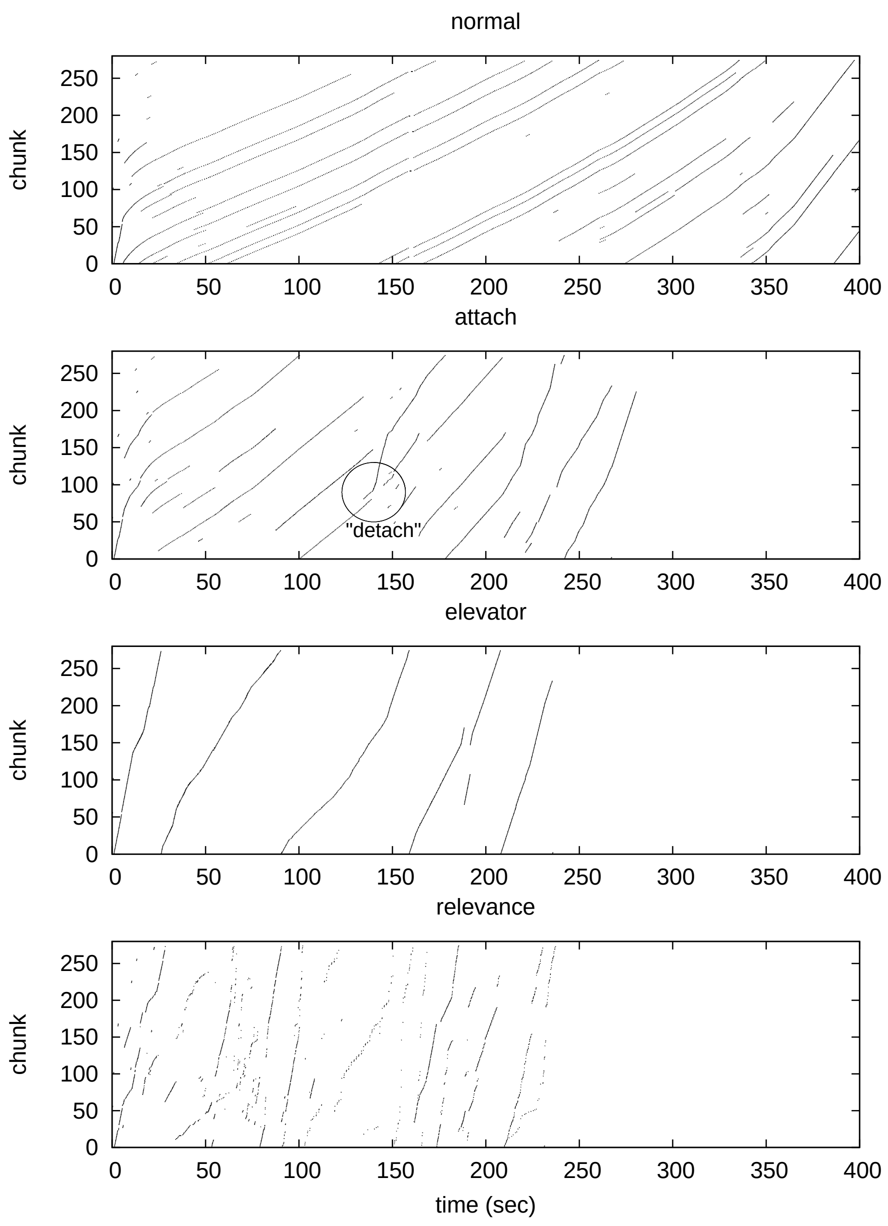


Figure 7.6: Behavior of different scheduling algorithms: disk accesses over time

in *normal*) concurrent scans, since not all queries can share the same chunk sequence. Additionally, we can see that a faster query can detach from a slower one (circled), resulting in a split of a reading sequence, further reducing the performance.

The *elevator* policy shows a further reduction of the I/O requests and improvement of system throughput. This is a result of its simple I/O pattern seen in Figure 7.6. However, we see that the average normalized latency is very bad for this policy. This is caused by the short queries that suffer from a long waiting time, and achieve results even worse than in *normal.* This blocking of queries also degrades the overall system time, since it delays the start moment of the next query in a given stream. We also see that fast and slow queries differ little in performance - this is caused by the fast queries waiting for the slow ones.

Our new *relevance* policy achieves the best performance, both in terms of global system parameters, as well as in most query times. As Figure 7.6 shows, its I/O request pattern is much more dynamic than in all the other policies. Interestingly, although *relevance* issues more I/Os than *elevator*, it still results in a better throughput. This is because the system is mostly CPU-bound in this case, and extra available I/O time is efficiently used to satisfy further query requirements. Also, *relevance* differs from other policies by significantly improving the performance of I/O bound queries. Average query latency is three times better than *normal* and two times better than *attach* (I/O bound queries like F-100 can even be three times faster than *attach*).

7.3.2 Exploring many different query mixes

To provide more than accidental evidence of the superior performance of *relevance*, we conducted experiments with the same basic settings as in the previous section: 16 streams of 4 queries, TPC-H table with scale factor 10 and buffer size of 1GB. However, we changed the set of queries to explore two dimensions: range size and data-processing speed. Figure 7.7 shows the results, where we compare throughput as the average stream running time (y axis) and average normalized query latency (x axis) of *normal*, *attach* and *elevator*, with respect to our *relevance* policy. Each point represents a single run for one policy. The point labels describe runs in a format "SPEED-SIZE", where SPEED defines what query speeds were used (FS - mix of fast and slow queries, F - only fast ones, FFS - 2 times more fast queries than slow ones etc.), and SIZE represents the size of the range being scanned: S - short (mix of queries reading 1, 2, 5, 10 and 20% of a table), M - mixed (1,2,10,50,100) and L - long (10,30,50,100).

From this experiment we conclude that indeed *relevance*, representing the

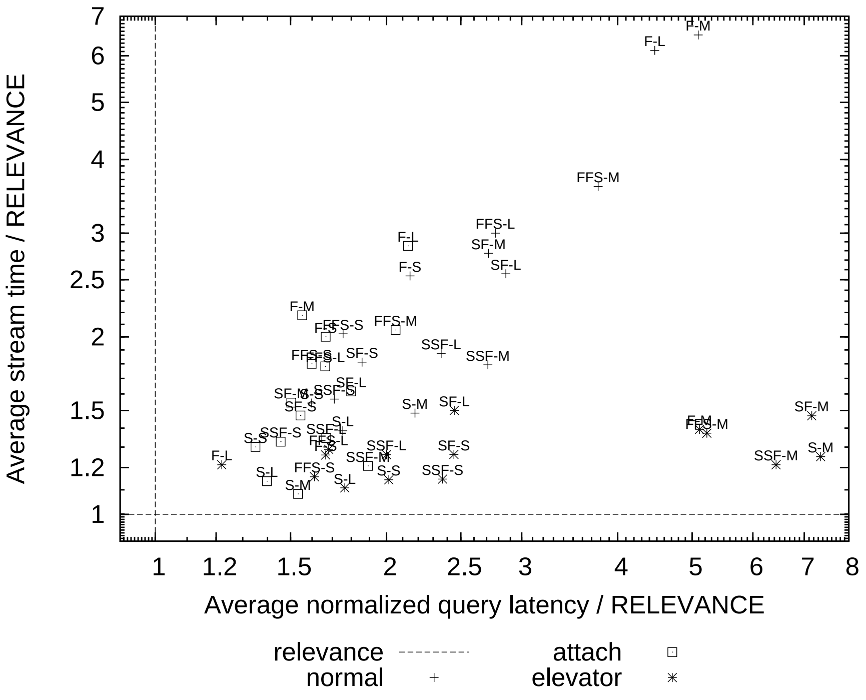


Figure 7.7: Performance of various scheduling policies for query sets varying in processing speed and scanned range

(1,1) point in this scatter plot, is consistently better than all other policies. Recall that our objective was to find a scheduling policy that works well both on the query throughput and on the query latency dimension, and this is exactly what is shown in Figure 7.7. This scatter plot also allows us to better understand the other policies. We see that *normal* is inferior on both dimensions, whereas *elevator* gets close to *relevance* on throughput, but its performance is significantly hindered by poor query latencies. As for the *attach*, it does find a balance between throughput and latency, but it is consistently beaten by *relevance* in both dimensions.

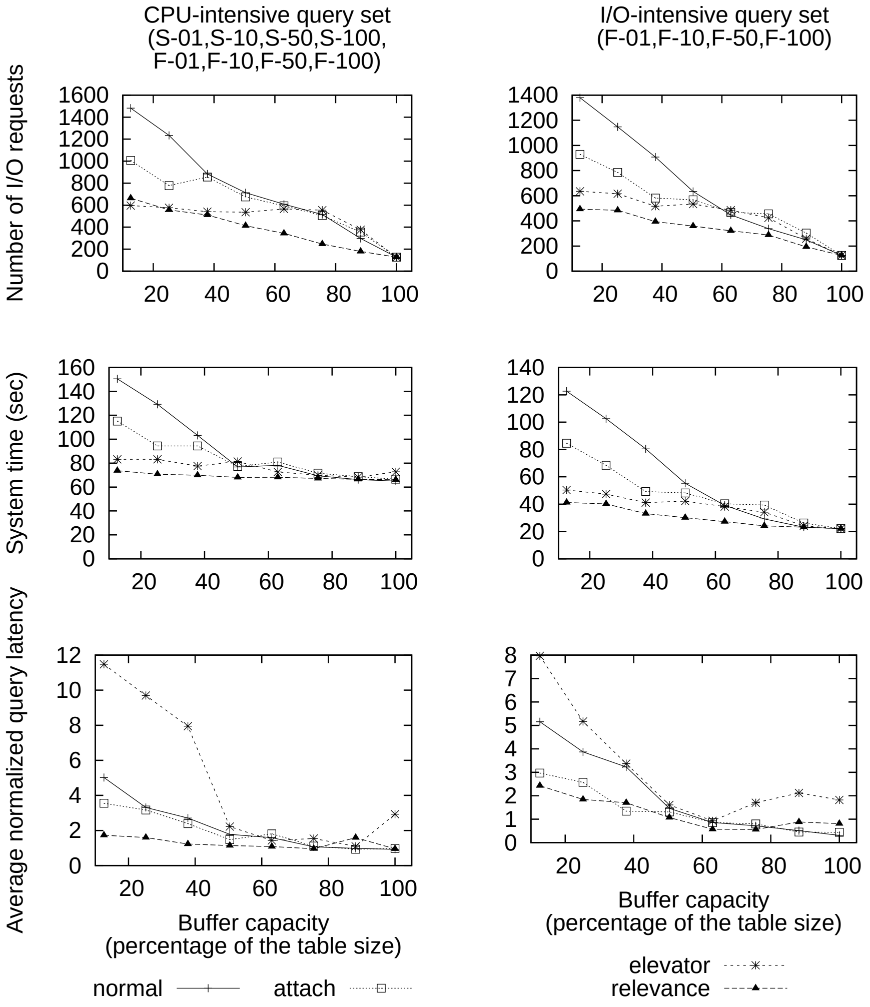


Figure 7.8: Behavior of all scheduling policies under varying buffer pool capacities

7.3.3 Scaling the data volume

With growing dataset sizes, the percentage of a relation that can be stored inside the buffer pool decreases. To simulate this, we tested the performance of different policies under varying buffer size capacities, ranging from 12.5% to 100% of the full table size. This allows us to observe how different scheduling policies would behave under growing relation sizes, when a smaller fraction of a table can be buffered. In this experiment we used a version of our relation trimmed-down to 2 GB, that can be fully cached in the memory of our benchmark machine. Figure 7.8 shows the results of a benchmark with two sets of queries, one disk-intensive, consisting only of fast queries, and a CPU-intensive one, consisting of a mix of fast and slow queries. We used 8 streams of 4 queries.

As expected, the number of I/Os is decreasing with increasing buffer size. In the disk-intensive case, the absolute system performance is directly influenced by this number, because the system is never CPU-bound. Still, thanks to better request scheduling, the *relevance* policy manages to improve the performance, issuing significantly fewer I/O requests even when using a 87.5% buffer capacity. In the CPU-intensive scenarios, the number of I/Os influence the absolute time only partially. This is because most algorithms manage to make a system CPU-bound with some buffer capacity. For *relevance* even a small buffer size of 12.5% of the full table is enough to achieve this, as we can see by its mostly constant performance.

Interestingly, Figure 7.8 shows that the performance advantages of *relevance* over the other policies as observed in Figure 7.7 are maximized when tables get bigger (i.e. at low buffered percentages). When looking at *attach*, the most viable competitor, we see that throughput in I/O bound situations, as well as latency in CPU bound queries deteriorate strongly, and we expect the advantage of *relevance* to grow even more if table sizes become huge.

7.3.4 Many concurrent queries

With more concurrent queries the opportunities for data-reuse increase. In Figure 7.9 we present how the average query time changes when an increasing number of concurrent queries reads 5, 20 and 50% of our relation, using a buffer pool of 1GB. As expected, the benefit of *relevance* over *normal* grows with larger scans and more concurrency. We see that *relevance* also enhances its advantage over *attach* when more queries run concurrently, even exceeding the factor two observed in Figures 7.8 and 7.7, when scan ranges are very or moderately selec-

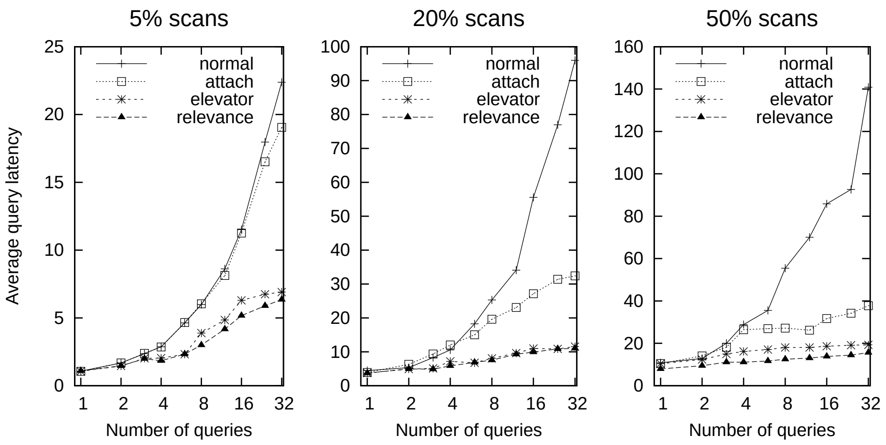


Figure 7.9: Performance comparison with varying number of concurrent queries and scanned ranges

tive. As this query set is uniform in terms of range sizes, *elevator* can score close to *relevance*, but we know from previous experiments that on combinations of short and long ranges it is not a viable competitor.

7.3.5 Scheduling-cost scalability

The cost of scheduling in *relevance* is significantly higher than for other policies. For example, the *loadRelevance* function needs to check every query for every table chunk, and do this for each chunk a query requests. Figure 7.10 presents the average times spent on scheduling when running 16 concurrent streams of 4 I/O-bound queries, each with the same relation stored in a different number of chunks of varying sizes. As expected, the overhead grows super-linearly - with smaller chunks, every query needs to scan more of them, and the decision process for each data request needs to consider more chunks. Still, even with the largest tested number of chunks, the scheduling overhead in the worst case does not exceed 1% of the entire execution time. In situations when such overhead is not acceptable, e.g. with relations consisting of hundreds of thousands of chunks, slightly less complex policies can be considered. Also, our *relevance* implementation is rather naive, leaving opportunities for optimizations that can significantly reduce the scheduling cost.

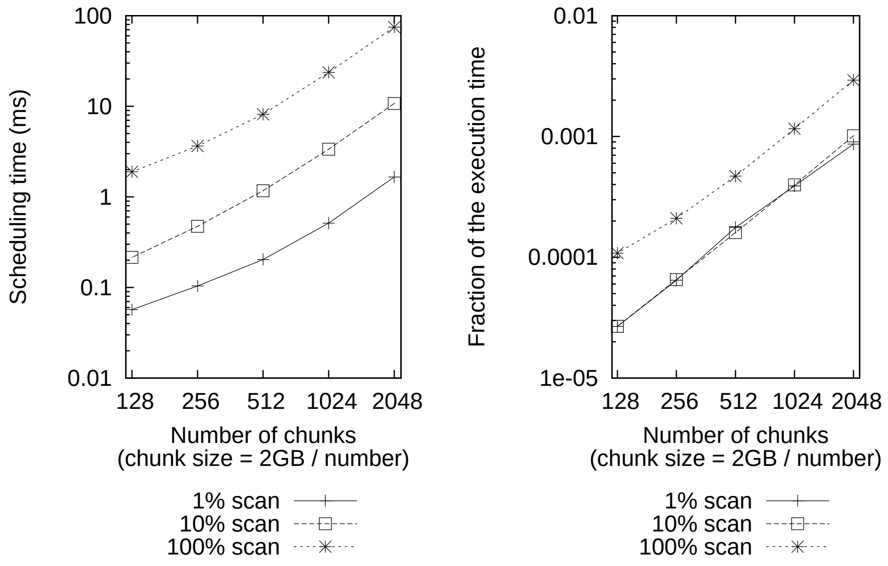


Figure 7.10: Scheduling time and fraction of execution time when querying 1%, 10% and 100% of a 2GB relation with varying chunk size / number

7.4 Improving DSM scans

After our successful experiments with *relevance* in row-storage, we now turn our attention to column-stores. The decomposed storage model (DSM) has recently gained popularity for its reduced disk-bandwidth needs, faster query processing thanks to improved data locality [ADHS01, HP03], possibility of vectorized processing (Section 5.2.2) and additional compression opportunities (Chapter 6, [AMF06]). While we will show that *relevance* can also be successful here, we first discuss why DSM is much more complex than NSM when it comes to scans in general and I/O scheduling in particular, and how this influenced our Cooperative Scans framework.

7.4.1 DSM challenges

Table columns stored using DSM may differ among each other in physical data representation width, either because of the data types used, or because of com-

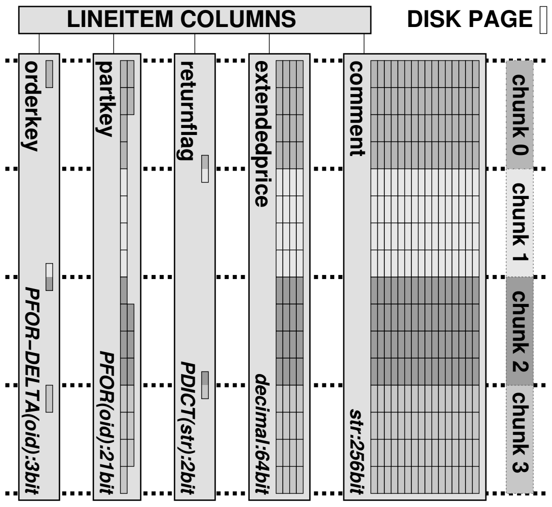


Figure 7.11: Compressed column storage: more complex logical chunk – physical page relationships

pression. For example, Figure 7.11 depicts column storage of a part of the TPC-H `lineitem` table, with some columns compressed with techniques presented in Chapter 6. This shows that we can not assume a fixed number of tuples on a disk page, even within a single column. As a result, a chunk cannot consist of a fixed number of disk pages as in NSM. Instead, chunks are logical concepts, i.e. a horizontal partitioning of the table on the tuple granularity. For example, one may divide a table in conceptual chunks of a 100.000 tuples, but it is also possible to use variable-size chunks, e.g. to make the chunk boundary always match some key boundary. This implies that chunk boundaries do not align with page boundaries. The underlying storage manager should provide an efficient means to tell which pages store data from a chunk. Depending on the physical data representation, a single logical chunk can consist of multiple physical pages, and a single physical page can contain data for multiple logical chunks.

This logical-physical mismatch present in DSM becomes even more problematic when using large physical blocks of a fixed size for I/O, a technique introduced in NSM for good concurrent bandwidth. The first problem here is that when loading a block for one chunk, a potentially large amount of data from a neighboring chunk can be loaded at the same time. In NSM this does

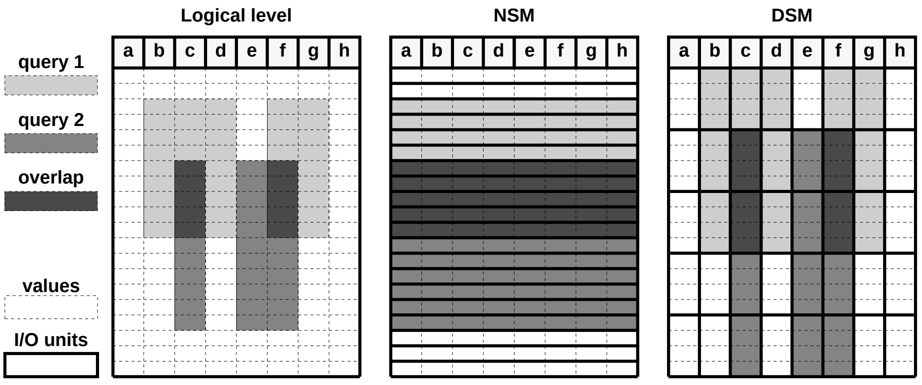


Figure 7.12: Data overlapping in NSM and DSM

not occur, as chunks and blocks are equivalent. In DSM, however, ABM needs to take special care to minimize situations in which this extra data is evicted before it could be used by a different chunk. Also, keeping a full physical block in memory to provide it to another query in the near future may result in a sub-optimal buffer usage. The second issue, buffer-space demand, is a general problem for DSM I/O scheduling. In NSM, for Q concurrent queries the system requires memory for $2 * Q$ (factor 2 because of prefetching) blocks, which is usually acceptable, e.g. 512MB for 16 concurrent scans using 16MB chunks/blocks. In DSM, however, to process a set of rows, data from multiple columns needs to be delivered. While some optimizations are possible, e.g. performing selections on some columns early [HLAM06], in general all the columns used by a query need to be loaded before the processing starts. As a result, a separate block is needed for every column used in a query, increasing the buffer demand significantly for multi-column scans, e.g. to 4GB for 16 scans reading 8 columns each. This can be improved (in both models) by analyzing which pages from blocks are already processed, and re-using them as-soon-as-possible for I/Os for different queries (with scatter-gather I/O). Both problems demonstrate that in DSM the performance and resource demand can be significantly improved by making algorithms page-aware. However, such solutions significantly complicate the implementation, and our current system currently handles only chunk-level policies.

The final DSM problem for Cooperative Scans is the reduced data reuse opportunity between queries. Figure 7.12 shows two queries reading a subset of a table and their I/O requirements in both NSM and DSM. Comparing the logical data need to the physical data demand in both models, we see that the vertical expansion present in DSM is usually significantly smaller than the horizontal expansion present in NSM. As a result, fewer disk blocks are shared between the queries, reducing the chance of reusing the same block for different scans. In NSM, for a block fetched by some query Q_1, the probability that another query Q_2, reading T_2 tuples, will use it is proportional to $\frac{T_2}{T_T}$, where T_T is the number of tuples in the entire table. In DSM, we need to take into account both vertical (as in NSM) and horizontal overlap, reducing this probability to $\frac{T_2}{T_T} * P_{overlap}(Q_1, Q_2)$, where $P_{overlap}(Q_1, Q_2)$ is the probability of a column from Q_1 also being used in Q_2.

7.4.2 Cooperative Scans in DSM

The DSM implementation of the traditional policies is straightforward. In *normal*, the order of I/Os is strictly determined by the query and LRU buffering is performed on a (chunk,column) level. DSM *attach* joins a query with most overlap, where a crude measure of overlap is the number of columns two queries have in common. A more fine-grained measure would be to get average page-per-chunk statistics for the columns of a table, and use these as weights when counting overlapping columns. Just like in NSM, the DSM *elevator* policy still enforces a global cursor that sequentially moves through the table. Obviously, it only loads the union of all columns needed for this position by the active queries.

The framework for *relevance* in DSM is similar to that in NSM, with a few crucial differences, caused by the challenges discussed in the previous section:
avoiding data waste – as discussed, with I/O based on large physical blocks, it is possible that a block loaded for one logical chunk contains data useful for neighboring chunks. When the first chunk is freed, this data would be evicted. To avoid that, the choice of the next chunk for a given query is performed before the query blocks for a fully available chunk. The already-loaded part of the chunk is marked as used, which prohibits its eviction.
finding space for a chunk – in DSM it is possible that a subset of columns in a buffered chunk is not useful for any query. `ABM` first evicts blocks belonging to such columns. Then, it starts evicting useful chunks, using the *keepRelevance* function to victimize the least relevant chunk. Note that, unlike in NSM, this eviction process is *iterative*, since due to different physical chunk sizes, possibly

```
useRelevance($c, q_{trigger}$)
|  $cols$ = queryColumns($q_{trigger}$)
|  $U$ = |interestedOverlappingQueries($c$, $cols$)|
|  $P_u$ = numberCachedPages($c$, $cols$)
|  return $P_u/U$

loadRelevance($c, q_{trigger}$)
|  $query\_cols$ = queryColumns($q_{trigger}$)
|  $queries$ = overlappingStarvedQueries($c$, $query\_cols$)
|  $cols$ = columnsUsedInQueries($queries$)
|  $L = |queries|$
|  $P_l$ = |columnPagesToLoad($c, cols$)|
|  return $L/P_l$

keepRelevance($c, q_{trigger}$)
|  $starved$ = almostStarvedQueries($c$)
|  $cols$ = columnsUsedInQueries($starved$)
|  $E$ = |starved|
|  $P_e$ = |cachedColumnPages($c, cols$)|
|  return $E/P_e$
```

Figure 7.13: DSM Relevance Functions

multiple chunks need to be freed.[1]

column-aware relevance functions – Figure 7.13 shows that the DSM relevance functions need to take into account the two-dimensional nature of column storage and the varying physical chunk sizes. Like in NSM, *useRelevance* attempts to use chunks needed by few queries, to make them available for eviction. However, it also analyzes the size of a chunk, to additionally promote chunks occupying more buffer space. The *loadRelevance* function looks at the number of starved queries that overlap with a triggering query and are interested in a given chunk. It also estimates the cost of loading a given chunk by computing the number of cold pages required for all needed columns. The returned score promotes chunks that benefit multiple starved queries, and require a small amount of I/O. The DSM *keepRelevance* function promotes keeping chunks that occupy little space in the buffer pool and are useful for many queries.

column loading order – a final issue in DSM is the order of columns when loading a chosen chunk. If some queries depend only on a subset of the columns, it may be beneficial to load that subset first. Our current crude approach is to

[1]If multiple chunks need to be freed, the dependency between them should be taken into account, something missed by the greedy iterative approach used here. Choosing the optimal set of chunks to free is a good example of a *knapsack problem* surfacing in DSM I/O scheduling.

		Normal	**Attach**	**Elevator**	**Relevance**
System statistics					
avg stream time		536.18	338.24	352.35	264.82
avg norm. lat.		7.05	4.77	15.11	2.96
total time		805	621	562	515
CPU use		61 %	77 %	82 %	92 %
I/O requests		6490	4413	2297	3639
Query statistics					
query	cold latency	avg. latency	avg. latency	avg. latency	avg. latency
F-01	0.92	6.12	4.68	26.95	3.17
F-10	2.99	21.01	16.39	45.64	10.19
F-50	15.88	191.12	108.53	141.84	64.97
F-100	26.53	364.33	198.86	145.81	90.16
S-01	1.90	6.92	5.07	54.75	3.33
S-10	8.15	47.93	37.96	103.12	21.93
S-50	36.28	148.19	126.20	134.19	88.19
S-100	71.25	346.65	259.14	184.60	231.38

Table 7.2: Column-storage experiments with a set of FAST and SLOW queries scanning 1%, 10%, 50% and 100% of a table (16 streams of 4 random queries, all times in seconds)

just load column chunks in increasing size (in terms of pages), which maximizes the number of "early" available columns, allowing queries to be awoken earlier. Another approach could prioritize columns that faster satisfy some query needs. Finally, if data is compressed on disk but kept decompressed in the buffer manager (like in SybaseIQ), it might be valuable to first load compressed columns, so their decompression is interleaved with loading the remaining ones.

7.4.3 DSM results

Table 7.2 presents DSM results for an experiment similar to the one presented in Table 7.1 for NSM/PAX. One difference is that we used a faster "slow" query, since, due to the faster scanning achieved by DSM, with the original query the system was completely CPU bound, making it impossible to demonstrate performance differences of different policies with these queries. Also, we increased the *lineitem* size from factor 10 (60Mtuples) to 40 (240Mtuples), to compensate for the lower data-volume demand of DSM. Finally, since our current implementation requires reserved memory for each active chunk, and there are

Queries (used columns)	**Normal**			**Relevance**		
	number of I/Os	**query latency avg.**	**query latency stddev**	**number of I/Os**	**query latency avg.**	**query latency stddev**
Non-overlapping queries						
ABC	5094	100.58	20.71	1560	24.27	5.24
ABC,DEF	6215	121.83	24.83	3254	57.87	14.54
Partially-overlapping queries						
ABC	5094	100.58	20.71	1560	24.27	5.24
ABC,BCD	5447	107.86	21.28	2258	39.69	10.34
ABC,BCD,CDE	5791	113.26	27.39	2918	52.94	14.02
ABC,BCD,CDE DEF	6313	125.14	22.35	3299	60.20	12.50

Table 7.3: Performance of DSM queries when scanning different sets of columns of a synthetic table

more chunks in DSM (one for each column), we had to increase the buffer size from 1GB to 1.5GB to allow concurrent execution of 16 queries.

The results confirm that also in DSM *relevance* is clearly the best scheduling approach. All policies behave as observed earlier in NSM: *normal* performs bad in both dimensions, while *attach* and *elevator* both improve the system throughput, with the former additionally improving query latencies. The *relevance* policy beats the competitors in both dimensions, only losing slightly to *elevator* on the slow full-table scan.

7.4.3.1 Overlap-ratio experiments

While so-far we have seen another success story of *relevance*, in DSM there is the caveat of column overlap. If queries have a significant percentage of overlapping columns, DSM provides good I/O reuse opportunities, which are then best exploited by *relevance*. In the following experiment, however, we investigate to what extent decreasing column overlap affects performance. We have performed a synthetic benchmark, where we run various queries against a 200M-tuple relation, consisting of 10 attributes (called A to J), each 8 bytes wide. The buffer size is 1GB. We use 16 streams of 4 queries that scan 3 adjacent columns from the table. In different runs, corresponding queries read the same 40% subset of the relation, but may use different columns. The total number of chunks for the entire run is 7680 chunks.

The first part of Table 7.3 shows the performance changes when query types do not have any overlapping columns. With 16 parallel queries and one resp. two query types, we thus have 16 resp. 8 queries of the same type (but with randomly chosen 40% scan ranges). Due to the rather large size of the scans, *normal* can still re-use quite a few blocks in case of a single query type (around 33% of the 7680 chunks), but about half of that is lost when two column-disjunct queries are used. As for *relevance*, very good re-use is achieved using a single query type, with *relevance* beating *normal* by a factor 4. With two query types, the average query latency doubles, which corresponds to the 0.5 reduction of sharing opportunities, but *relevance* still beats *normal* by a factor two there.

With non-overlapping query families, numbers are somewhat harder to understand, but the general trend is that I/O reuse drops with decreasing column overlap. As *relevance* normally benefits more from bandwidth sharing, it is hit more, relative to *normal*, but we still observe *relevance* beating *normal* by a factor two in these situations. These results confirm that the benefit of the *relevance* policy does depend on the columns used in the queries. This knowledge can be exploited by applications. For example, when looking for correlations in data mining, assuming thousands of queries are issued in the process and but only few are executing at the same time, it may be beneficial to schedule the queries such that the column overlap is maximized.

7.5 Cooperative Scans in a RDBMS

In this section we outline how existing DBMSs can be extended with Cooperative Scans, focusing on the `ABM` implementation and adapting order-aware operators to out-of-order data delivery.

7.5.1 ABM implementation

The most practical and least intrusive way to integrate Cooperative Scans into an existing RDBMS is to put `ABM` on top of the standard buffer manager. We successfully created an early `ABM` prototype in PostgreSQL [ZBK04]. Here, to load a chunk, `ABM` requests a range of data from the underlying buffer manager. This request is fulfilled by reading multiple pages occupying random positions in the standard buffer pool. These pages, locked by `ABM` after reading, are provided to all interested `CScan` operators and finally are freed when `ABM` decides to evict them. An additional benefit is that `ABM` can dynamically adjust its buffer size in situations when the system-wide load changes , e.g. when the number of active `CScan` operators decreases. Also, if the buffer manager provides an appropriate interface, it is possible to detect which pages of a chunk are already buffered and promote partially-loaded chunks in `ABM`.

Though the original focus of `CScan` was improving the scans of a single table, a production-quality implementation of `CScan` should be able to keep track of multiple tables, keeping separate statistics and meta-data for each (large) table in use. As our approach targets I/O bound situations, for small tables `CScan` should simply fall back on `Scan`.

Finally, `ABM` only improves performance on clustered scans. For unclustered data access, `CScan` should not be used. Still, `ABM` can exploit the queue of outstanding page requests generated by the normal buffer manager to prioritize chunks more as they intersect more with this queue. When the *chooseChunkToLoad()* decides to load a chunk, any intersecting individual page requests should be removed from the normal page queue.

7.5.2 Order-aware operators

In this section, we discuss the impact of the out-of-order delivery of tuples by `CScan` on query processing. In its purest form, the relational algebra is order-unaware, and this holds true for many physical operators (e.g. nested-loop join, scan-select, hash-based join and aggregation, etc.). However, query optimizers make a clear distinction between order-aware and unaware physical operators (e.g. by enumerating sub-plans that preserve certain "interesting orders"). The two major order-aware physical operators are ordered aggregation and merge-join.

Ordered aggregation exploits the key-ordering of the input for efficient computation of per-key aggregate results that can be immediately passed to the parent once a key change is detected. With Cooperative Scans, ordered aggregation can still exploit the fact that the per-chunk data is internally sorted. We pass the chunk number of a tuple as a virtual column via the Volcano-like operator interface of MonetDB/X100. When processing a given chunk, the operator performs inside-chunk ordered aggregation, passing on all the results except for the first and the last one in the chunk, as these aggregates might depend on the data from other chunks. These border values are stored on a side, waiting for the remaining tuples that need to be included in that computation. A key observation is that chunks are large, so not huge in number, and the number of boundary values to keep track of is limited by the number of chunks. Looking at the chunk sequence, it is also possible to detect the "ready" boundary values early and pass them to the parent immediately, which is especially useful with multiple consecutive chunks delivered.

Merge Join can be handled in the *attach* and *elevator* policies as follows [HSA05]: at a moment when a scan starts on one table, a matching position in the other table is found, and join processes until the end of table. Then, the scan on both tables starts from the beginning, processing until the original position.

Since *relevance*'s data delivery pattern is much more dynamic, a more complex approach is necessary. In case the inner table fits main memory, it is enough to switch to a proper position in this table (using search or index lookup) whenever a chunk in the outer table changes. Similarly, with a multi-level storage hierarchy, including both fast but small solid state memory and large but slow disks, similar strategy can be applied if the inner table fits on a flash drive and the cost of finding a matching position in it is small.

Situation complicates when both tables need to be scanned from disk. After loading data from the outer table, it is necessary to enforce loading a matching range in the inner table, and synchronize their delivery to the reading operators. Since a chunk in the inner table can be useful for multiple chunks in the outer relation, the *loadRelevance* function for the outer table should be extended to take into account the availability of matching data in the inner table, optimizing its reuse. Clearly, this approach significantly complicates implementation, especially in multi-join scenarios, suggesting using traditional `Scan` operators in such cases.

There is one special, yet valuable, scenario where `CScan` can be applied on *both* sides of a merge join. MonetDB/X100 uses *join indices* for foreign-key relationships. For example, the join index over `orderkey` between `lineitem` and `order` in TPC-H adds the physical row-id `#order` as an invisible column to `lineitem`. By storing the `lineitem` table sorted on `#order` (and `order` itself sorted on `orderdate`), we get *multi-table clustering*, where tuples are stored in an order corresponding to the foreign key join relationship.

Within MonetDB/X100, there is ongoing work on *Cooperative Merge Join* (CMJ) that works on top of such clustered tables, fully accepting out-of order data as it is delivered by `CScan`. The key observation is that multi-table clustered DSM tables can be regarded as a single, joined, DSM table on the level of `ABM`, as it already has to deal with the fact that in DSM columns have widely varying data densities and chunk boundaries never coincide with page boundaries. Thus, `ABM` views the physical representation of the clustered `order` and `lineitem` table as the physical representation of the already joined result, even though the data density in the `order` columns is on average six times lower than in `lineitem`. Using the freedom to choose the boundaries of logical chunks at will, it makes sure that matching tuples from `order` and `lineitem` always belong to the same chunk. Thus, a single `CScan` operator can deliver matching column data from both `order` and `lineitem` tables and the special CMJ merge-join reconstructs joined tuples from these.

7.6 Related work

Disk scheduling policies are a topic that originated from operating systems research [TP72]. Various such policies have been proposed, including First Come First Served, Shortest Seek Time First, SCAN, LOOK and many others. Most relevant for our work is SCAN, also known as the "Elevator" algorithm. In this approach, a disk head performs a continuous movement across all the relevant cylinders, servicing requests it finds on its way. Other related operating system work is in the area of virtual memory and file system paging policies, for which generally LRU schemes are used. Note that these solutions are mostly targeted at optimizing the disk seek time with multiple random disk accesses. In case of large sequential scans, these policies will offer very little improvement.

Previous research in DBMS buffer management [CD85, SS86, CR93, FNS91] usually considered large table scans trivial and suggested a simple LRU or MRU policy, which minimized the possibility of inter-query data reuse. To overcome this problem, the concept of circular-scans has been introduced in some commercial DBMSs, e.g. Teradata, RedBrick and Microsoft SQLServer [NCR02, Col98, Coo01]. A variation of this idea was suggested in [KSR01], where authors issue a massive number of concurrent request to the buffer manager and serve them in a circular fashion. It was also discussed as a part of the Q-Pipe architecture [HSA05]. All these approaches follow either the *attach* or *elevator* policies, which in Section 7.3 have been shown as inferior to the new proposed *relevance* policy. Recently, a modified version of the *attach* policy has been suggested for table scans [LBM$^+$07] and index scans [LBMW07] it the IBM DB2 system. This solution introduces slight improvements to *attach*, by adding explicit group control and allowing a limited throttling of faster queries, but still suffers from the main *attach* problems.

Most previous work regarding table scans has focused on row storage only, ignoring scans over column-oriented data. A recent paper by Harizopoulos et al. [HLAM06] provides a detailed analysis of the I/O behavior differences between DSM and NSM. However, this paper concentrates on single-query scenarios and does not analyze the problem of the DSM buffer demand, which we found important, especially in a concurrent environment.

Scheduling is also important in real-time database systems [KGM95], where transactions need to be scheduled to meet certain time critical constraints. This involves making scheduling decisions based on the availability of certain resources, such as CPU, I/O, and buffer-manager space. Our work differs from such scheduling, in that we do not treat buffer-manager space as a resource being competed for, but rather schedule in a way to maximize sharing opportunities.

In multi-query optimization [SSB00], the optimizer identifies common work units in concurrent queries and either materializes them for later use [MPK00] or creates pipelined plans where operators in multiple queries directly interact with each other [DSRS01]. The concept of shared scans is often a base for such query plans [DSRS01]. When compared with our work, multi-query optimization is performed on a higher level, namely on the level of query processing operators that may be shared. Such operator sharing is even the cornerstone of the Q-Pipe architecture [HSA05].

A related approach is multi-query *execution* (rather than optimization). The NonStop SQL/MX server [Cle99] introduced a special SQL construct, named 'TRANSPOSE', that allows explicitly specifying multiple selection conditions and multiple aggregate computations in a single SQL query, which is executed internally as a single scan.

Ideas close to our algorithms have been explored in research related to using tertiary storage. Sarawagi and Stonebraker [SS96] present a solution that reorders query execution to maximize data sharing among the queries. Yu and DeWitt [YD97] propose pre-executing a query to first determine the exact access pattern of a query and then exploit this knowledge to optimize the order of reads from a storage facility. Moreover, they use query batching to even further improve performance in a multi-query environment. Shoshani et al. [Sho99] explore the multi-dimensional index structure to determine files interesting for queries and apply a simple file weighting based on the number of queries interested in it. This is a special-purpose system, while we attempt to integrate Cooperative Scans in (compressed column) database storage and query processing architectures.

Ramamurthy and DeWitt recently proposed to use the actual buffer-pool content in the query optimizer for access path selection [RD05]. This idea can be extended for Cooperative Scans, where the optimizer could adjust the estimated scan cost looking at the currently running queries.

All research discussed so far focused on improving disk data delivery performance. Modern multi-core CPUs with complex memory hierarchies result in RAM bandwidth becoming a bottleneck, as discussed e.g. in Section 6.2.9. As a result, data-sharing algorithms are also applied on the CPU level. For example, in [QRR+08] queries running on different cores that share some level of cache reuse the data stored there, minimizing main-memory traffic.

7.7 Conclusions and future work

This chapter motivated and described the *Cooperative Scans* framework that significantly enhances existing I/O scheduling policies for query loads that perform concurrent (clustered index) scans. One area where this is highly relevant is data warehousing, but (index) scan-intensive loads are found in many more application areas, such as scientific databases, search, and data mining.

The Active Buffer Manager (`ABM`) coordinates the activities of multiple Cooperative Scan (`CScan`) queries in order to maximize I/O bandwidth reuse, while ensuring good query latency. We compared a number of existing scheduling policies (LRU, *attach*, *elevator*), and have shown that our new *relevance* policy outperforms them consistently.

We have shown the benefit of our approach in experiments using both row-wise storage (NSM or PAX) and column-wise storage (DSM). While column-stores have gained a lot of interest in recent years, we are not aware of significant previous work on I/O scheduling for column stores. One of our findings here is that DSM scheduling is much more complex, and efficient DSM I/O requires considerably more buffer space than NSM. Our new policy performs progressively better when buffer space is scarce, which plays to its advantage in DSM.

We described how ABM can be implemented on top of a classical buffer manager and also discussed order-aware query processing despite out-of-order data delivery, which is a topic of ongoing research.

Chapter 8

Conclusions

The research presented in this thesis, summarized in Section 8.1, introduces a number of techniques that improve the query execution efficiency in databases by taking advantage of modern hardware trends. The resulting MonetDB/X100 system, used to validate our ideas, achieves high performance in database processing, as demonstrated with the TPC-H benchmarks presented throughout this thesis and summarized in Section 8.2.1. Additionally, thanks to its efficiency, it makes it possible for database technology to be successfully applied in other areas, as discussed in Section 8.2.2. Finally, many ideas presented in this thesis lead to interesting research problems, discussed in Section 8.3.

8.1 Contributions

This thesis introduces a number of techniques in the field of data-intensive query processing, focusing on in-memory query execution and processing of disk-resident data. The proposed methods are presented in the context of a balanced database architecture.

8.1.1 Improving in-memory query processing

The major contribution in this area is the *vectorized execution model* introduced in Section 4.2. This model combines the best features of the previously proposed tuple-at-a-time and column-at-a-time models, taking the scalability from the former, and the raw processing performance from the latter. This high performance is achieved by spending most of the CPU time in data processing primitives designed and implemented to execute efficiently on modern superscalar CPUs. Additionally, in-cache execution allows good use of hierarchical memory structures. As a result, the proposed model has been shown to provide performance often one or two orders of magnitude better than the existing solutions.

Research on the efficient implementation of the vectorized execution model, presented in Chapter 5, resulted in a number of algorithms and implementation methods beneficial to a wide range of applications. First, a number of algorithm design and implementation techniques for the vectorized model has been presented. These techniques demonstrate how to decompose algorithms into a sequence of vectorized steps and how to efficiently implement primitives performing these steps. Additionally, we discuss how different data organization models have different performance characteristics for different processing tasks, and introduce a novel idea of dynamic data reorganization inside the query processing pipeline to better exploit these properties. These techniques are synthesized in a fully vectorized implementation of a generic hash-join, which is demonstrated to achieve performance comparable with a hand-written, specialized implementation. A proposed technique of cache-efficient best-effort partitioning additionally allows applying algorithms using hash-tables to data sizes exceeding the capacity of the CPU cache. Finally, we propose new implementations of a set of important, but less-frequently researched, data processing algorithms, such as exception handling, string processing and binary search, showing how the vectorized versions can provide significant performance improvement over traditional approaches.

8.1.2 Improving processing of disk-resident data

For disk-resident data, this thesis proposes to remove the need for random disk accesses by following the idea of *scan-only* query processing. In this way large volumes of data can be efficiently provided to the query execution layer. To reduce the impact of the increasing imbalance between the CPU and disk performance, this thesis additionally introduces two techniques that allow better exploitation of the available disk bandwidth.

The first technique follows the idea of using data compression to reduce the volume of data that needs to be fetched from disk. This thesis introduces a set of new compression algorithms that are focused on efficiently exploiting the properties of modern CPUs to achieve high compression and decompression speeds. Additionally, we propose keeping the data compressed in the buffer manager and decompressing it on the boundary between RAM and the CPU cache. This technique allows caching more data, avoids in-memory materialization, and can even improve the performance of the memory bandwidth on multi-core machines.

The second major contribution is the research on improving the performance of concurrently running scan queries. Within the general framework of *cooperative scans* we investigate the properties of various scheduling algorithms in databases. We also introduce a new *relevance* policy that dynamically schedules I/O requests by analyzing the entire system activity and, thanks to dynamic adaptation to changing conditions, achieves significant performance improvements for a large class of queries. Finally, this work discusses why efficient scanning is more complex in the DSM world, with queries scanning not only a subset of rows but also a subset of columns.

8.1.3 Balanced database system architecture

Performance benefits of the vectorized execution model can only be observed if the data delivery layer can match its processing speed. Similarly, improvements to the I/O infrastructure will not be visible if the performance of the processing layer is low and the system is not I/O bound.

The design of the MonetDB/X100 architecture is based around the idea of different optimizations cooperating to achieve high performance. An example of this approach is the compression layer: it decompresses data per-vector saving it into the CPU cache, perfectly matching the vectorized execution model.

The importance of balancing both processing and storage optimizations is presented with an experiment in Figure 8.1. Here, TPC-H Query 6 (1GB scale factor) is executed on a 2.8GHz Intel Nehalem system with an efficient disk subsystem. We compare the performance using different vector sizes, I/O buffer sizes, and two queries: one operating on raw data (ca. 160 MB) and one operating on compressed data (ca. 36 MB)[1]. With fully buffered data the overhead of decompression adds a small, but visible overhead (left plot). However, compression provides two benefits. First, since the data volume is reduced, a smaller buffer size is required to have data fully cached (middle plot). Secondly, even if the data needs to be fully read from the disk, the I/O overhead is significantly smaller for the compressed data (right plot). In all cases, small vector sizes cause the execution to be dominated by the interpretation overhead, making the impact of data format and buffer size negligible. This experiment demonstrates that compression can scale vectorized execution to larger data sets, and

[1]both queries perform a fully sequential scan, hence do not benefit from the partially buffered data

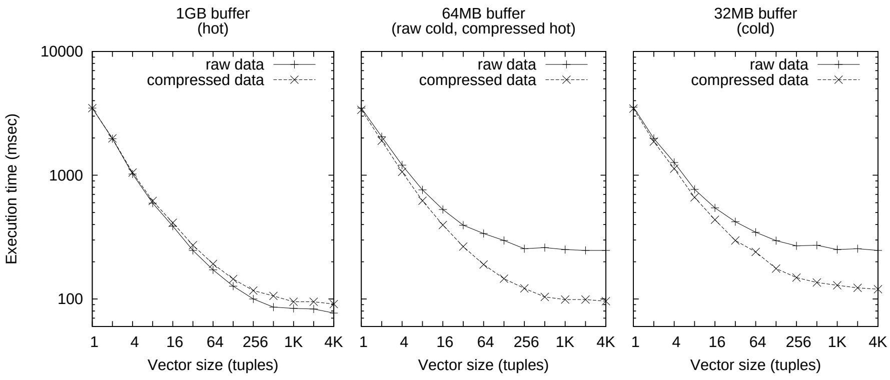


Figure 8.1: Performance of the TPC-H Query 6 with raw and compressed data using different I/O buffer sizes

at the same time high-performance execution is needed to make the benefit of compression visible.

8.2 Evaluation

The techniques used in the MonetDB/X100 system allowed it to achieve very high efficiency in data-analysis tasks, as demonstrated with the TPC-H results that are summarized in Section 8.2.1. This performance, often close to hand-written solutions, has led to the investigation if the proposed architecture can also be applied in other fields where databases were traditionally not used due to their poor performance. Section 8.2.2 discusses how MonetDB/X100 successfully competed with specialized solutions in one of such areas: large-scale information retrieval.

8.2.1 TPC-H performance

TPC-H is currently the standard benchmark suite for decision support systems [Tra06]. It simulates systems that process large volumes of data and execute highly complex queries to answer critical business questions. This is a typical example of *data-intensive* problems. Since research presented in this thesis focuses on improving database performance in such applications, TPC-H has

TPC-H query	MonetDB/X100			IBM DB2 UDB 8.1
	Itanium2 1.3GHz 12GB RAM	AMD Opteron 2GHz, 4GB RAM 4 disks		8 x P4 Xeon 2.8GHz 16GB RAM
	in-memory	raw data	compressed	142 disks
01	30.25	307.2	69.6	111.9
03	3.77	35.0	11.3	15.1
04	1.15	18.2	2.4	12.5
05	11.02	54.3	15.3	84.0
06	1.44	48.2	10.7	17.1
07	29.47	119.8	72.0	86.5
11	1.66	27.0	14.6	19.5
14	2.64	23.7	12.2	10.9
15	14.36	44.9	22.4	21.6
18	10.37	181.9	50.6	318.2
21	17.61	197.6	46.6	374.9

Table 8.1: TPC-H SF-100 results on MonetDB/X100: memory-resident benchmarks (from [BZN05]), disk-resident benchmarks (from [ZHNB06]). DB2 results for comparison (2006 results from `www.tpc.org`)

been used throughout this thesis as a leading example evaluating the proposed techniques.

Table 8.1 demonstrates the performance achieved by MonetDB/X100 on a subset of the TPC-H benchmark for both memory- and disk-resident data. It also present the results of the IBM DB2 benchmark running on a significantly more powerful hardware configuration. While these systems are not fully comparable, as MonetDB/X100 at the moment of these benchmarks did not provide all the capabilities required by TPC-H and the query plans were hand-crafted, the results indicate that MonetDB/X100 performs significantly better in terms of the absolute performance. This performance advantage becomes even more visible when taking into account the difference in the available processing and storage power.

8.2.2 Information retrieval with MonetDB/X100

Integration of information retrieval and databases is seen as one of the major goals of the database community [AY05, CRW05]. Still, these two areas have developed independently from each other, even though many IR tasks can be performed using relational systems [GFHR97, GBS04]. One of the reasons for not applying database technology to information retrieval is the poor performance of standard database solutions. For example, only one attempt has been made to participate in the Terabyte TREC benchmark using a database system [CCS04], and the performance of the presented solution was significantly worse than that of specialized IR systems. Since one of the goals of MonetDB/X100 was to bridge the gap between general-purpose database technology and task-specialized applications, we decided to evaluate the performance of our system in this area. This section provides only an overview of the results, more details can be found in [HZdVB06, CHZ+08, HZdVB07].

8.2.2.1 Expressing IR tasks as relational queries

For our experiments, we investigated a set of techniques applied in keyword search, including Boolean OR and AND queries as well as the popular Okapi BM25 formula [RWB98]. In these models, data is typically stored in *inverted files* [ZM06], where for each term a sorted list of documents with this term is stored, either containing all individual occurrences, or simply a number of occurrences within the document. For this storage model, a keyword search can be expressed as a combination of these lists with some arithmetic on top, typically followed by Top-N computation. This general structure can be applied to both Boolean queries as well as to BM25. Interestingly, the task of combining inverted lists is the exact equivalent of performing a sequence of *merge-join* operations (inner or outer) on the matching data in a database. This allows easy mapping of these query models onto relational queries, as presented in [CHZ+08, HZdVB06, HZdVB07].

8.2.2.2 Performance on Terabyte TREC benchmark

A simple mapping of IR tasks onto MonetdDB/X100 queries provides performance already in the same ballpark as specialized systems [HZdVB07]. However, many of the optimization techniques used in IR can also be applied in a relational system. For example, the high-performance data compression algorithms of MonetDB/X100 provide a significant performance benefit when querying disk-resident data, and allow fitting more data in RAM, reducing the number of machines needed to store the entire inverted index in a distributed setting [HZdVB07]. Also, IR techniques such as two-pass querying [BCH+03] and score materialization are easily expressible in a relational system [HZdVB07].

Run	**Index size (GB)**	**p@20**	**CPUs**	**Time per query (ms)**
Indri	100	0.5610	1	1724
Wumpus	14	0.5310	1	91
Zettair	44	0.4770	1	390
MonetDB/X100	9	0.5470	1	117

Table 8.2: MonetDB/X100 performance compared to custom IR systems on a single-CPU disk-resident Terabyte TREC data (from [CHZ+08])

The discussed techniques allow MonetDB/X100 to achieve performance directly comparable with the specialized IR systems, as demonstrated with the Terabyte-TREC results presented in Table 8.2. In this benchmark, a collection of 25 millions documents with a total size of 426 GB is queried using simple keyword-search. System efficiency (speed) is measured by executing 50,000 queries, while effectiveness (accuracy) is evaluated by early precision (*p@20*) on a subset of 50 preselected queries for which relevance judgments are available. Systems competing in this benchmark are typically specialized IR solutions optimized for executing this type of queries. In our approach, search requests are expressed fully as relational queries and executed on MonetDB/X100 without any specific IR optimizations. The results show that, surprisingly, MonetDB/X100 easily competes with the best participating systems, achieving high efficiency without hurting the effectiveness. Additional general-purpose optimizations, e.g. using PAX storage[2] to reduce random disk access cost, would make this comparison even more favorable for MonetDB/X100.

More detailed experiments presented in [CHZ+08], demonstrate that our approach is also highly flexible. Thanks to its efficient compression, MonetDB/X100 reduces the dataset size from 29GB to 9GB, making it memory-resident on a 12GB machine. This removes the need for I/O and brings the average execution time down to 21ms. Additionally, remote query execution allows easy setup of distributed experiments, where MonetDB/X100 achieves amortized query time of just 3.2ms (using 8 nodes, each with 2GB of RAM).

These results demonstrate that MonetDB/X100 can be successfully applied not only to database tasks, but to a wider class of problems where large volumes of data need to be processed fast. In the future, we hope to investigate other such areas, including scientific data processing and data mining.

[2]PAX storage was not implemented in MonetDB/X100 at the time of running these experiments

8.3 Future research directions

The material discussed in this thesis provides valuable insights in many areas of query execution on modern hardware. Still, the dynamic nature of computer evolution, as well as the wide scope of discussed areas, lead to a number interesting problems for future research.

8.3.1 Improving the vectorized execution model

In this thesis the vectorized execution model has been shown to have good performance characteristics. Still, multiple improvements are possible.

One of the crucial parameters of the vectorized execution model is the vector size, discussed in Section 4.2.2.2. Currently, it is fixed for the entire query. However, with blocking operators inside the query tree, the query plan is effectively decomposed into semi-independent processing stages with potentially different characteristics. In such a scenario, the vector sizes can be different for various operators, to better balance the available cache size and query complexity.

In Section 5.2.2 we discussed how the storage model during query execution influences performance. While currently only DSM and NSM models were discussed, it is possible to generalize these models to a setup where different attributes are clustered into multiple groups, each represented as DSM (single-attribute groups) or NSM (multi-column groups), as discussed in [HP03]. In some situations, this can provide performance improvements. Furthermore, the research presented in [ZNB08] demonstrates that in-query on-the-fly data conversion can provide extra performance benefits. However, such a flexible data organization introduces an additional dimension to the query optimizer, especially with dynamic data reorganization. Also, efficient storage of variable-width data types, as well as efficient handling of selection (and other types of tuple-indirection) in this model need to be researched.

8.3.2 Storage-layer improvements

Light-weight compression methods discussed in Chapter 6 can significantly reduce the I/O bandwidth hunger. Still, the presented set of algorithms is relatively limited, as it does not allow efficient compression and fast decompression of many data sets. For example, similarly performing algorithms for strings or floating-point numbers need to be researched. Additionally, with the increased number of possible compression schemes (and their parameters), more efficient and flexible automatic compression algorithms need to be investigated.

The cooperative-scans technique presented in Chapter 7 currently focuses on optimizing single-table scans, as found in a typical star schema. Extending this concept to a multi-table scenario, especially in a situation when individual queries scan multiple tables, is an interesting challenge. Similarly, applying this technique to complex merge-join scenarios poses a problem. Finally, with the continuously increasing imbalance between memory and cache speeds, it is possible that some of the discussed concepts can be applied to coordinate memory accesses by concurrently running queries.

Another research direction is related to the rapidly increasing popularity of solid-state storage (see Section 2.3.3). Since this form of storage has significantly different parameters, new algorithms to access the data need to be devised [Ros08, SHWG08].

The final topic currently being researched in the storage layer of MonetDB/X100 are efficient updates for analytical applications. Since data organization for such scenarios (clustering, DSM storage, compression) is often different than for transaction processing, new techniques that allow efficient updating of such structures need to be investigated.

8.3.3 Parallel execution

The vectorized execution model provides interesting opportunities for parallel execution. With parallelism implemented e.g. with *exchange* operators [Gra90], the larger size of the data transfer unit leads to less overhead of the inter-process communication. Additionally, vertical fragmentation minimizes the volume of RAM-to-CPU data transfer, reducing the problem of insufficient memory bandwidth on multi-core machines.

A large processing unit in vectorized processing leads to the idea of parallelizing on the level of a single data-processing primitive, similarly to the *horizontal parallelism* for relational operators. With large L2 and L3 caches, holding tens of thousands of values, the overhead of synchronization might be amortized well and this approach might provide an efficient solution with relatively limited system changes.

8.3.4 Alternative hardware platforms

Previous research [HNZB07] demonstrated that the vectorized execution model is a good foundation for database processing kernels on the hybrid STI Cell processor. Interestingly, a lot of emerging computing architectures share some characteristics with Cell. SIMD-like processing units constitute a consistently growing fraction of the entire system computing power. For example, for many tasks modern GPUs easily provide better performance than price-comparable general purpose CPUs. Multiple processing cores already are a mainstream solution, and the number of cores in future generations of CPUs will probably grow [HBK06]. In some architectures, e.g. GPUs, parallelism is an inherent property of the computing units. Also, the heterogeneity of the computing units is increasing, with systems consisting of multiple types of cooperating devices (CPUs, GPUs etc), potentially also internally heterogeneous (e.g. Cell). Finally, the hard restrictions on code and data size, as well as explicit management of both, appear in many new architectures. We believe that the vectorized model provides multiple opportunities to address these issues, and can be a good foundation for developing data processing kernels on these new architectures.

Bibliography

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[ACJ+07] Mani Azimi, Naveen Cherukuri, D. N. Jayasimha, Akhilesh Kumar, Partha Kundu, Seungjoon Park, Ioannis Schoinas, and Aniruddha S. Vaidya. Integration Challenges and Tradeoffs for Tera-scale Architectures. *Intel Technology Journal*, 11(3):173–184, August 2007. Special issue on Tera-scale Computing.

[AD07] Sunil Agarwal and Hermann Daeubler. *Reducing Database Size by Using Vardecimal Storage Format.* Microsoft, 2007.

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, Rome, Italy, 2001.

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. VLDB*, Edinburgh, 1999.

[Adv05] Advanced Micro Devices Inc. *Software Optimization Guide for AMD64 Processors*, September 2005.

[Aga96] Ramesh C. Agarwal. A Super Scalar Sort Algorithm for RISC Processors. In *Proc. SIGMOD*, Montreal, Canada, 1996.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. SIGMOD*, Dallas, USA, 2000.

[Ail05] Anastassia Ailamaki. Database architectures for new hardware. In *Proc. ICDE*, Tokyo, Japan, 2005.

[AM05] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[AMDM07] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proc. ICDE*, 2007.

[AMF06] Daniel Abadi, Sam Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proc. SIGMOD*, 2006.

[App06] Applied Micro Circuits Corporation. *3ware 9650SE Datasheet*, 2006.

[AvdBF$^+$92] P. Apers, C. van den Berg, J. Flokstra, P. Grefen, M. Kersten, and A. Wilschut. PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):541–554, December 1992.

[AY05] S. Amer-Yahia. Report on the DB/IR Panel at Sigmod 2005. *SIGMOD Record*, 34(4), 2005.

[Bar96] Dirk Bartels. ODMG 93 - The Emerging Object Database Standard. In *Proc. ICDE*, pages 674–676, New Orleans, LA, USA, 1996.

[BBPV00] Christophe Bobineau, Luc Bouganim, Philippe Pucheral, and Patrick Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *VLDB*, 2000.

[BCH$^+$03] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, 2003.

[Ben99] Jon Bentley. *Programming Pearls*. ACM Press, 2nd edition, 1999.

[BGB98] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proc. International Symposium on Computer Architecture*, pages 3–14, Barcelona, Spain, 1998.

[BGvK$^+$06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. SIGMOD*, pages 479–490, Chicago, IL, USA, 2006.

[BK99] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal*, 8(2):101–119, 1999.

[Bla98] Kenneth R. Blackman. IMS celebrates thirty years as an IBM product. *IBM Systems Journal*, 37(4), 1998.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. VLDB*, pages 54–65, Edinburgh, 1999.

[Bon02] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.* Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[BS92] C. R. Banger and D. B. Skillicorn. Flat arrays as a categorical data type. Technical report, Queen's University, Kingston, Canada, 1992.

[BZN05] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, 2005.

[CAB$^+$81] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[CAGM04] Shimin Chen, Anastassia Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance through Prefetching. In *Proc. ICDE*, Boston, MA, USA, 2004.

[CAGM05] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Inspector joins. In *Proc. VLDB*, Trondheim, Norway, 2005.

[CAGM07] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured English query language. In *Proc. SIGFIDET*, pages 249–264, Ann Arbor, Michigan, 1974.

[CB94] Z. Cvetanovic and D. Bhandarkar. Characterization of alpha AXP performance using TP and SPEC workloads. In *Proc. International Symposium on Computer Architecture*, pages 60–70, Chicago, IL, USA, 1994.

[CCS04] C. L. A. Clarke, N. Craswell, and I. Soboroff. Overview of the TREC 2004 Terabyte Track. In *Proc. TREC*, 2004.

[CD85] H.-T. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proc. VLDB*, 1985.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. *SIGMOD Rec.*, 30(2):271–282, 2001.

[CGM01] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proc. SIGMOD*, Santa Barbara, CA, USA, 2001.

[CGMV02] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B+-Trees: optimizing both cache and disk performance. In *Proc. SIGMOD*, Madison, USA, 2002.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proc. SIGPLAN PLDI*, Atlanta, GA, USA, 1999.

[CHZ$^+$08] Roberto Cornacchia, Sandor Heman, Marcin Zukowski, Arjen P. de Vries, and Peter Boncz. Flexible and Efficient IR using Array Databases. *The VLDB Journal*, 17(1):151–168, January 2008.

[CK85] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, 1985.

[Cle99] J. Clear et al. NonStop SQL/MX primitives for knowledge discovery. In *Proc. KDD*, 1999.

[Col98] L. S. Colby et al. Redbrick vista: Aggregate computation and management. In *Proc. ICDE*, 1998.

[Coo01] C. Cook. *Database Architecture: The Storage Engine*, July 2001. `http://msdn.microsoft.com/library`.

[CR93] C.-M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proc. VLDB*, 1993.

[CR07] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Proc. VLDB*, 2007.

[CRG07] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel Buffers for Chip Multiprocessors. In *Proc. SIGMOD DaMoN Workshop*, Beijing, China, 2007.

[CRW05] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *Proc. CIDR*, 2005.

[CSS] Charles Clarke, Falk Scholer, and Ian Soboroff. TREC Terabyte Track. `http://www-nlpir.nist.gov/projects/terabyte/`.

[CvBdV04] Roberto Cornacchia, Alex van Ballegooij, and Arjen P. de Vries. A case study on array query optimisation. In *Proc. CVDB*, Paris, France, 2004.

[DH98] Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. *SIGARCH Comput. Archit. News*, 26(3):167–178, 1998.

[DKO$^+$84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proc. SIGMOD*, 1984.

[DSRS01] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *Proc. PODS*, 2001.

[EKM$^+$04] Andrew Eisenberg, Krishna Kulkarni, Jim Melton, Jan-Eike Michels, and Fred Zemke. SQL:2003 Has Been Published. *SIGMOD Record*, 33(1):119–126, March 2004.

[FHL$^+$07] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. GPUQP: query co-processing using graphics processors. In *Proc. SIGMOD*, Beijing, China, 2007.

[FKT86] S. Fushimi, M Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *Proc. VLDB*, August 1986.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proc. FOCS*, 1999.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *EEE Trans. Comput*, C-21(9):948–960, 1972.

[FNS91] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. VLDB*, 1991.

[GAHF05] Brian T. Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating Database Operations Using a Network Processor. In *Proc. SIGMOD DaMoN Workshop*, Baltimore, MD, USA, 2005.

[GBC98] Goetz Graefe, Ross Bunkera, and Shaun Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *Proc. VLDB*, August 1998.

[GBS04] Torsten Grabs, Klemens Bhoem, and Hans-Jorg Schek. PowerDB-IR: scalable information retrieval and storage with a cluster of databases. *Knowledge and Information Systems*, 6(4):465–505, 2004.

[GBY07] Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *VLDB*, 2007.

[GFHR97] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating structured data and text: A relational approach. *JASIS*, 48(2), 1997.

[GG97] Jim Gray and Goetz Graefe. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. SIGMOD*, Chicago, IL, USA, 2006.

[GLW$^+$04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proc. SIGMOD*, Paris, France, 2004.

[GP87] Jim Gray and Gianfranco R. Putzolu. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *Proc. SIGMOD*, San Francisco, CA, USA, 1987.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. SIGMOD*, pages 102–111, 1990.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[Gra07] Goetz Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. 2007.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proc. ICDE*, 1998.

[GS91] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, 1991.

[GT05] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[GYB07] Bugra Gedik, Philip S. Yu, and Rajesh Bordawekar. Executing Stream Joins on the Cell Processor. In *VLDB*, Vienna, Austria, 2007.

[HA04] Stavros Harizopoulos and Anastassia Ailamaki. STEPS towards Cache-resident Transaction Processing. In *Proc. VLDB*, Toronto, Canada, 2004.

[HBK06] Jim Held, Jerry Bautista, and Sean Koehl. *From a Few Cores to Many: A Tera-scale Computing Research Overview.* Intel Corporation, 2006.

[Hem05] Sandor Heman. Super-Scalar Database Compression Between RAM and CPU-Cache. Masters thesis, Universiteit van Amsterdam, July 2005.

[HKMW66] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, 1966.

[HL07] Bingsheng He and Qiong Luo. Cache-oblivious query processing. In *CIDR*, Asilomar, CA, USA, 2007.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel Abadi, and Samuel Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proc. VLDB*, 2006.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. International Symposium on Computer Architecture*, San Diego, CA, USA, 1993.

[HNZB07] Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *Proc. SIGMOD DaMoN Workshop*, Beijing, China, 2007.

[HNZB08] Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. Positional Delta Trees to reconcile updates with read-optimized data storage. Technical Report INS-E0801, CWI, August 2008.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[HP03] Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *Proc. VLDB*, pages 417–428, Berlin, Germany, 2003.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitive Approach.* Morgan Kaufmann Publishers, 4th edition, 2007.

[HPJ+07] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Proc. CIDR*, Asilomar, CA, USA, 2007.

[HSA05] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: a simultaneously pipelined relational query engine. In *Proc. SIGMOD*, Baltimore, MD, USA, 2005.

[Huf52] D.A. Huffman. A method for construction of minimum redundancy codes. In *Proceedings of the IEEE*, volume 40, pages 1098–1101, 1952.

[HYF+07] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Joins on Graphics Processors. Technical Report HKUST-CS07-06, Department of Computer Science and Engineering, HKUST, March 2007.

[HZdVB06] S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. MonetDB/X100 at the 2006 TREC TeraByte Track. In *Proceedings of the Text REtrieval Conference (TREC-2006)*, Gaithersburg, MD, USA, November 2006.

[HZdVB07] Sandor Heman, Marcin Zukowski, Arjen P. de Vries, and P. Boncz. Efficient and Flexible Information Retrieval Using MonetDB/X100. In *Proc. CIDR*, 2007.

[IBM07] IBM Corporation. *Cell Broadband Engine Architecture*, 2007.

[Int06] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*, January 2006.

[Int07a] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, November 2007.

[Int07b] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2007.

[Int07c] Intel Corporation. *Intel C++ Intrinsic Reference*, 2007.

[Int08] Intel Corporation. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*, March 2008.

[JRSS08] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. In *Proc. VLDB*, Auckland, New Zealand, 2008.

[KGM95] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In Sang H. Song, editor, *Advances in Real-Time Systems*, pages 463–486, 1995.

[Khr09] Khronos Group. *OpenCL - The Open Standard for Heterogeneous Parallel Programming*, February 2009.

[KPH$^+$98] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. In *Proc. International Symposium on Computer Architecture*, pages 15–26, Barcelona, Spain, 1998.

[KSR01] Yannis Kotidis, Yannis Sismanis, and Nick Roussopoulos. Shared index scans for data warehouses. In *Proc. DaWaK*, 2001.

[KSvdBB96] Martin L. Kersten, F. Schippers, Carel A. van den Berg, and Peter A. Boncz. *Mx documentation tool*, January 1996.

[Lar97] Per-Ake Larson. Grouping and duplicate elimination: Benefits of early aggregation. Technical Report MSR-TR-97-36, Microsoft, December 1997.

[LBE$^+$98] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. International Symposium on Computer Architecture*, pages 39–50, Barcelona, Spain, 1998.

[LBM$^+$07] Christian A. Lang, Bishwaranjan Bhattacharjee, Timothy Malkemus, Sriram Padmanabhan, and Kwai Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE*, Istanbul, Turkey, 2007.

[LBMW07] Christian A. Lang, Bishwaranjan Bhattacharjee, Tim Malkemus, and Kwai Wong. Increasing Buffer-Locality for Multiple Index Based Scans through Intelligent Placement and Index Scan Speed Control. In *Proc. VLDB*, Vienna, Austria, 2007.

[Lib] LibOIL. `http://liboil.freedesktop.org`.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *ASPLOS*, pages 222–233, Cambridge, MA, USA, 1996.

[LM99] Chi-Keung Luk and Todd C. Mowry. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Trans. Computers*, 48(2):134–141, 1999.

[LM07] Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In *Proc. SIGMOD*, Beijing, China, 2007.

[MA95] Petro Estakhri Mahmud Assar, Siamack Nemazie. Flash memory mass storage architecture. US patent 5,388,083, 1995.

[Mai82] David Maier. Using write-once memory for database storage. In *Proc. PODS*, Los Angeles, CA, USA, 1982.

[MBH$^+$02] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[MBK00] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Proc. VLDB*, Cairo, Egypt, 2000.

[MBK02] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Eng.*, 14(4):709–730, 2002.

[MBNK04] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. Cache-Conscious Radix-Decluster Projections. In *Proc. VLDB*, Toronto, Canada, 2004.

[McF93] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Equipment Corporation, June 1993.

[MDO94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *SIGOPS Oper. Syst. Rev.*, 28(5):145–156, 1994.

[Mic] Microsoft Corporation. *ODBC Programmer's Reference*. `http://msdn.microsoft.com/en-us/library/ms714177.aspx`.

[Moe98] Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proc. VLDB*, 1998.

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MPK00] S. Manegold, A. Pellenkoft, and M. Kersten. A Multi-Query Optimizer for Monet. In *Proc. BNCOD*, 2000.

[NBC$^+$95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A Cache-Sensitive Parallel External Sort. *VLDB J.*, 4(4):603–627, 1995.

[NCR02] NCR Corp. Teradata Multi-Value Compression V2R5.0. 2002.

[Net] Netezza Inc. *Netezza*. `http://www.netezza.com`.

[NVI08] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture. Reference Manual. Version 2.0*, June 2008.

[Pat04] David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47:71–75, October 2004.

[PB61] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49:228–235, 1961.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. SIGMOD*, pages 109–116, Chicago, IL, USA, 1988.

[PHS99] Jih-Kwon Peir, Windsor W. Hsu, and Alan Jay Smith. Functional Implementation Techniques for CPU Cache Memories. *IEEE Transactions on Computers*, 48(2):100–110, 1999.

[PMAJ01] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proc. ICDE*, Heidelberg, Germany, 2001.

[PP03] Meikel Pöss and Dmitry Potapov. Data Compression in Oracle. In *Proc. VLDB*, 2003.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[QRR$^+$08] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-Memory Scan Sharing For Multi-Core CPUs. In *Proc. VLDB*, Auckland, New Zealand, 2008.

[RD05] R. Ramamurthy and D. DeWitt. Buffer pool aware query optimization. In *Proc. CIDR*, 2005.

[RDS02] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *Proc. VLDB*, pages 430–441, Hong Kong, 2002.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. ASPLOS*, San Jose, CA, USA, 2002.

[Ros02] Kenneth A. Ross. Conjunctive Selection Conditions in Main Memory. In *Proc. PODS*, Washington, DC, USA, 2002.

[Ros07] Kenneth A. Ross. Efficient Hash Probes on Modern Processors. In *Proc. ICDE*, Istanbul, Turkey, 2007.

[Ros08] Kenneth A. Ross. Modeling the Performance of Algorithms on Flash Memory Devices. 2008.

[RR99] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proc. VLDB*, Edinburgh, 1999.

[RR00] Jun Rao and Kenneth A. Ross. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proc. SIGMOD*, Philadelphia, PA, USA, 2000.

[RS60] Irving Reed and Gustave Salomon. Polynomial Codes over Certain Finite Fields. *SIAM Journal ofApplied Mathematics*, 8:300–304, 1960.

[RS82] Ronald L. Rivest and Adi Shamir. How to reuse a ẅrite - once m̈emory (Preliminary Version). In *Proc. STOC*, San Francisco, CA, USA, 1982.

[RWB98] S. E. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track. In *Proc. TREC*, 1998.

[SAB$^+$05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *Proc. VLDB*, 2005.

[Sch04] Steven W. Schlosser. *Using MEMS-based Storage Devices in Computer Systems.* Ph.d. thesis, Carnegie Mellon University, May 2004.

[SG07] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proc. FAST*, 2007.

[Sho99] A. Shoshani et al. Multidimensional indexing and query coordination for tertiary storage management. In *Proc. SSDBM*, 1999.

[SHWG08] Mehul A. Shah, Stavros Harizopoulos, Janet L. Wiener, and Goetz Graefe. Fast Scans and Joins using Flash Drives . 2008.

[SHWK76] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of ingres. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[SK06] Gerrit Saylor and Badriddine Khessib. Large scale Itanium 2 processor OLTP workload characterization and optimization. In *Proc. SIGMOD DaMoN Workshop*, Chicago, IL, USA, 2006.

[SKN94] A. Shatdahl, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. VLDB*, Edinburgh, 1994.

[SKS02] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 4th edition, 2002.

[SKT$^+$05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM J. RES. and DEV.*, 49(4/5):505–512, July/September 2005.

[SL76] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3), 1976.

[Smi82] Alan J. Smith. Cache Memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[SS86] G. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. Database Syst.*, 11(4), 1986.

[SS96] S. Sarawagi and M. Stonebraker. Reordering query execution in tertiary memory databases. In *Proc. VLDB*, 1996.

[SSAG03] Steven W. Schlosser, Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Exposing and exploiting internal parallelism in MEMS-based storage. Technical Report CMU-CS-03-125, Carnegie Mellon University, mar 2003.

[SSB00] Pand S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. SIGMOD*, 2000.

[SSS$^+$04] Minglong Shao, Jiri Schindler, Steven W. Schlosser, Anastassia Ailamaki, and Gregory R. Ganger. Clotho: decoupling memory page layout from storage organization. In *Proc. VLDB*, Toronto, Canada, 2004.

[Ste90] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, 1990.

[Sto07] Jon Stokes. *Inside the Machine.* No Starch Press, 2007.

[Suna] Sun Microsystems Inc. *JDBC Overview.* `http://java.sun.com/products/jdbc/overview.html`.

[Sunb] Sun Microsystems Inc. *Ultrasparc processors.* `http://www.sun.com/processors/`.

[Syb] Sybase Inc. *Sybase IQ.* `http://www.sybase.com`.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proc. International Symposium on Computer Architecture*, S. Margherita Ligure, Italy, 1995.

[TKK+88] Shun'ichi Torii, Keiji Kojima, Yasusi Kanada, Akiharu Sakata, Seiichi Yoshizumi, and Masami Takahashi. Accelerating Nonnumerical Processing by an Extended Vector Processor. In *Proc. ICDE*, Los Angeles, CA, USA, 1988.

[TP72] T. Teorey and T. Pinkerton. A comparative analysis of disk scheduling policies. *Commun. ACM*, 15(3), 1972.

[Tra94] Transaction Processing Performance Council. *TPC Benchmark A, Revision 2.0*, June 1994.

[Tra06] Transaction Processing Performance Council. *TPC Benchmark H, Revision 2.6.1*, June 2006.

[Tra07] Transaction Processing Performance Council. *TPC Benchmark C, Revision 5.9*, June 2007.

[Tro03] Andrew Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[Val87] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[Waa02] Florian Waas. Extending Iterators for Advanced Query Execution. In *Proc. ADC*, Melbourne, Australia, 2002.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[Wil91] Ross N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference*, pages 362–371, 1991.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, September 2000.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers, 1999.

[Wol04] Wayne Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, San Diego, CA, USA, 2004.

[YAA03] Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. Tabular Placement of Relational Data on MEMS-based Storage Devices. In *VLDB*, pages 680–693, Berlin, Germany, 2003.

[YD97] J.-B. Yu and D. DeWitt. Query pre-execution and batching in paradise: A two-pronged approach to the efficient processing of queries on tape-resident raster images. In *Proc. SSDBM*, 1997.

[ZBK04] Marcin Zukowski, Peter A. Boncz, and Martin L. Kersten. Cooperative scans. Technical Report INS-E0411, CWI, December 2004.

[ZBNH05] Marcin Zukowski, Peter Boncz, Niels Nes, and Sandor Heman. MonetDB/X100: A DBMS In The CPU Cache. *Data Engineering Bulletin*, 28(2), 2005.

[ZCRS05] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. Improving Database Performance on Simultaneous Multithreading Processors. In *Proc. VLDB*, Trondheim, Norway, 2005.

[ZHB06] Marcin Zukowski, Sandor Heman, and Peter Boncz. Architecture-Conscious Hashing. In *Proc. SIGMOD DaMoN Workshop*, Chicago, IL, USA, 2006.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*, 2006.

[ZHNB07] Marcin Zukowski, Sandor Heman, Niels Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc. VLDB*, 2007.

[ZLS08] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, Beijing, China, 2008.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. In *Proc. SIGMOD DaMoN Workshop*, 2008.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *Proc. SIGMOD*, Madison, USA, 2002.

[ZR03a] Jingren Zhou and Kenneth A. Ross. A Multi-Resolution Block Storage Model for Database Design. In *Proc. IDEAS*, Hong Kong, 2003.

[ZR03b] Jingren Zhou and Kenneth A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *Proc. VLDB*, Berlin, Germany, 2003.

[ZR04] Jingren Zhou and Kenneth A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *Proc. SIGMOD*, Paris, France, 2004.

[Zuk02] Marcin Zukowski. Parallel Query Execution in Monet on SMP Machines. Masters thesis, Vrije Universiteit Amsterdam and Warsaw University, August 2002.

[Zuk05a] Marcin Zukowski. Hardware Conscious DBMS Architecture for Data-Intensive Applications. In *Proc. VLDB PhD Workshop*, 2005.

[Zuk05b] Marcin Zukowski. Improving I/O Bandwidth for Data-Intensive Applications. In *Proc. BNCOD Doctoral Consortium*, 2005.

Summary

With the performance of modern computers improving at a rapid pace, database technology has problems with fully exploiting the benefits that each new hardware generation brings. This has caused a significant performance gap between general-purpose databases and specialized, application-optimized solutions for large-volume computation-intensive processing problems, as found in areas including information retrieval, scientific data management and decision support.

This thesis attempts to enhance the state-of-the-art in architecture-conscious database research, both in the query execution layer as well as in the data storage layer, and in the way these work together. Thus, rather than focusing on an isolated problem or algorithm, the thesis presents a new database system architecture, realized in the *MonetDB/X100* prototype, that combines a coherent set of new architecture-conscious techniques that are designed to work well together.

The motivation for the new query execution layer comes from the analysis of the problems of two popular approaches to query processing: *tuple-at-a-time* operator pipelining, used in most existing systems, and *column-at-a-time* materializing operators, found in MonetDB. MonetDB/X100 proposes a new *vectorized in-cache* execution model that exploits ideas from both approaches and combines the scalability of the former with the high-performance *bulk processing* of the latter. This is achieved by modifying the traditional *operator pipeline* model to operate on cache-resident *vectors* of data using highly optimized *primitive* functions. Additionally, within this architecture, a set of hardware-conscious design and programming techniques is presented, enabling efficient execution of typical data processing tasks. The resulting query execution layer efficiently exploits modern super-scalar CPUs and cache-memory systems and achieves in-memory performance often one or two orders of magnitude higher than the existing approaches.

In the storage area there are two hardware trends that significantly influence

database performance. First, the imbalance between sequential disk bandwidth and random disk latency continuously increases. As a result, access methods that rely on random I/O become less attractive, making various forms of sequential access the preferred option. MonetDB/X100 follows this idea with *ColumnBM* – a bandwidth-optimized column store. Secondly, both disk bandwidth and latency improve significantly more slowly than the computing power of modern CPUs, especially with the advent of multi-core CPUs. ColumnBM introduces two techniques that address this issue. *Lightweight in-cache compression* allows trading some processor time for an increased perceived disk bandwidth. High decompression performance is achieved by applying the decompression on the RAM-cache boundary, providing cache-resident data directly to the execution layer. Additionally, the introduced family of compression methods provides performance an order of magnitude higher than previous solutions. *Cooperative scans* observe current system activity and dynamically schedule I/O operations to exploit overlapping demands of different queries. This amortizes the cost of disk access among multiple consumers, and also better utilizes the available buffer space, providing much better performance with many concurrently executing queries.

By combining CPU-efficient processing with a bandwidth-optimized storage facility, MonetDB/X100 has been able to achieve its high in-memory raw query execution power also on huge disk-resident datasets. We evaluated its performance both on TPC-H decision support data sets as well as in the area of large-volume information retrieval (the Terabyte TREC task), where it successfully competed with the specialized solutions, both for in-memory and disk-based tasks.

Samenvatting

De prestaties van moderne computers nemen in hoog tempo toe. Database technologie toont echter problemen bij het benutten van de verbeteringen die iedere hardware generatie brengt. Dit heeft geleid tot een significante discrepantie tussen de prestaties van traditionele database engines en gespecialiseerde, applicatie-geoptimaliseerde oplossingen voor data- en reken-intensieve verwerkingsproblemen, zoals voorkomen binnen de 'information retrieval', het beheren van wetenschappelijke data, en 'decision support'.

In dit proefschrift wordt gepoogd de state-of-the-art op het gebied van hardwarearchitectuur-bewust database onderzoek te verbeteren, binnen zowel de software lagen voor executie als voor opslag, maar ook binnen de manier waarop deze twee samenwerken. In plaats van zich te richten op een enkel probleem of algoritme in isolatie, presenteert dit proefschrift een nieuwe database architectuur, gerealiseerd in het *MonetDB/X100* prototype, die een coherente verzameling van hardware-bewuste technieken omhelst, ontworpen om goed samen te kunnen werken.

De motivatie voor de nieuwe queryexecutielaag komt voort uit de analyse van de problemen die inherent zijn aan twee populaire queryexecutie methoden: *tuple-voor-tuple* operator pipelining, zoals gebruikt in de meeste bestaande systemen, en *kolom-voor-kolom* materialiserende operatoren, zoals in MonetDB. MonetDB/X100 stelt een nieuw *gevectoriseerd in-cache* executiemodel voor, dat ideeën uit beide methoden gebruikt, en de schaalbaarheid van de eerste combineert met de hoge prestaties bij *bulk verwerking* van de tweede. Dit wordt bereikt door het traditionele *operator pipeline* model aan te passen, zodat het opereert op *vectoren* van data in de CPU cache, gebruikmakend van sterk geoptimaliseerde *primitive* functies. Tevens wordt binnen deze architectuur een verzameling hardware-bewuste ontwerp- en programmeertechnieken gepresenteerd, die de efficiënte executie van typische dataverwerkingopdrachten bewerkstelligen. De resulterende queryexecutielaag maakt efficiënt gebruik van moderne 'super-scalar CPUs' en cache geheugens, en bereikt binnen het werkgeheugen prestaties die één of twee ordes van grootte sneller zijn dan die van bestaande systemen.

Op het gebied van gegevensopslag zijn er twee hardware trends die een significante invloed hebben op database prestaties. Allereerst blijft het gebrek aan balans tussen sequentiële disk bandbreedte en de toegangstijd bij willekeurige schijf zoekopdrachten verslechteren. Als resultaat daarvan worden toegangsmethoden die op willekeurige I/O berusten minder aantrekkelijk, waardoor sequentiële toegang meer voor de hand komt te liggen. MonetDB/X100 haakt in op dit idee met *ColumnBM* – een kolom opslagsysteem geoptimaliseerd voor sequentiële I/O. Ten tweede, nemen zowel bandbreedte als toegangstijd van hardeschijven aanzienlijk trager toe dan de rekenkracht van moderne CPUs, vooral sinds de komst van CPUs met meerdere verwerkingseenheden. ColumnBM introduceert twee technieken die dit probleem aanpakken. *Lichtgewicht in-cache compressie* maakt het mogelijk om processorkracht te verruilen tegen een toename in waargenomen schijf bandbreedte. Hoge decompressiesnelheid wordt bereikt door te decomprimeren op de RAM-cache grens, en daarbij de in de cache aanwezige data direct aan te bieden aan de executie laag. Tevens leveren voorgestelde compressie methoden snelheidsprestaties die een orde van grootte hoger zijn dan voorgaande oplossingen. *Coöperatieve Scans* observeren lopende activiteiten binnen het systeem, en plannen I/O operaties dusdanig dat overlap tussen lopende queries wordt uitgebuit. Dit maakt het mogelijk om de kosten van een I/O operatie te verdelen over meerdere consumenten, en tevens buffer ruimte beter te benutten, waardoor prestaties tijdens het gelijktijdig uitvoeren van meerdere queries aanzienlijk toenemen.

Door CPU-efficiënte verwerking en opslag geoptimaliseerd voor bandbreedte te combineren, is MonetDB/X100 in staat zijn hoge snelheden in dataverwerking binnen het werkgeheugen ook te bewerkstelligen in geval van gegevens die van de harde schijf moeten komen. Wij hebben de prestaties geëvalueerd zowel op de TPC-H 'decision support' dataset als op een grote 'information retrieval' dataset (de Terabyte TREC data), waar het systeem succesvol concurreerde met gespecialiseerde oplossingen, zowel binnen het werkgeheugen als in scenario's waarbij de data van schijf moest komen.

Streszczenie

Wydajność nowoczesnych komputerów zwiększa się w błyskawicznym tempie. Niestety, systemy bazodanowe mają problemy z pełnym wykorzystaniem możliwości pojawiających się z każdą kolejną generacją sprzętu komputerowego. To zjawisko doprowadziło do znaczącej różnicy wydajności pomiędzy bazami danych a wyspecjalizowanymi rozwiązaniami zoptymalizowanymi do intensywnego obliczeniowo przetwarzania dużych ilości danych, takich jak symulacje naukowe, wyszukiwanie informacji czy przetwarzanie multimediów.

Niniejsza praca proponuje nowe technologie bazodanowe koncentrujące się na efektywnym wykorzystaniu nowoczesnych architektur komputerów. Przedstawione są nowe rozwiązania zarówno w warstwie przetwarzającej dane w pamięci, jak również w warstwie przechowywania danych na dysku. Dodatkowo, kładziony jest nacisk na współpracę różnych rozwiązań wewnątrz spójnej architektury, zrealizowanej na przykładzie prototypowego systemu *MonetDB/X100.*

W warstwie przetwarzania danych w pamięci, proponujemy nową architekturę *wektorowego przetwarzania w pamięci podręcznej procesora* (*vectorized in-cache processing*). Ta technika pozwala na połączenie skalowalności tradycyjnego modelu *potokowego* (*pipeline*) z wysoką wydajnością *przetwarzania hurtowego* (*bulk-processing*) zaproponowanego w systemie MonetDB. Dodatkowo, proponujemy zestaw metod pozwalających efektywne wykonywanie standardowych zadań bazodanowych w ramach tej architektury. Wynikowy system skutecznie wykorzystuje nowoczesne procesory osiągając wydajność w pamięci o jeden lub dwa rzędy wielkości wyższą niż istniejące rozwiązania.

W warstwie przechowywania danych, dwie tendencje w rozwoju sprzętu wpływają na wydajność baz danych. Po pierwsze, nierównowaga pomiędzy sekwencyjnym i losowym dostępem do dysku się systematycznie zwiększa. W efekcie, rozwiązania bazujące na losowym dostępie stają się mniej atrakcyjne, czyniąc różne formy dostępu sekwencyjnego preferowanymi metodami. MonetDB/X100 idzie w tym kierunku, proponując *ColumnBM* - kolumnowy system przechowywa-

nia danych optymalizujący przepustowość transferu danych. Po drugie, zarówno przepustowość jak i czas dostępu dysku polepszają się znacząco wolniej niż moc obliczeniowa nowoczesnych procesorów. ColumnBM wprowadza dwie techniki, które starają się rozwiązać ten problem. *Szybka dekompresja w pamięci podręcznej* (*light-weight in-cache compression*) pozwala poświęcenie części czasu procesora w celu zwiększenia efektywnego transferu danych. W metodzie *współpracujących odczytów* (*cooperative scans*) dostępy do dysku są dynamicznie szeregowane dostosowując się do aktywności w systemie, pozwalając zamortyzować koszt dostępu do dysku na wiele równolegle wykonywanych zapytań.

Dzięki połączeniu wydajnego modelu przetwarzania danych w pamięci z sytemem przechowywania danych optymalizującym sekwencyjny transfer z dysku, MonetDB/X100 pozwala na skalowanie swojej wysokiej wydajności również dla dużych rozmiarów danych, nie mieszczących się w pamięci operacyjnej. Prezentowany system uzyskał wysoką wydajność zarówno w standardowym teście wydajności analitycznych baz danych TPC-H, jak i w teście wyszukiwania informacji na dużą skalę (*Terabyte TREC*), gdzie MonetDB/X100 skutecznie konkurował z rozwiązaniami wyspecjalizowanymi do tego typu zadań.

SIKS Dissertation Series

1998-1 Johan van den Akker (CWI). DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM). Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD). A Contribution to the Linguistic Analysis of Business Conversationswithin the Language/Action Perspective

1998-4 Dennis Breuker (UM). Memory versus Search in Games

1998-5 E.W.Oskamp (RUL). Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU). Physiology of Quality Change Modelling; Automated modelling ofQuality Change of Agricultural Products

1999-2 Rob Potharst (EUR). Classification using decision trees and neural nets

1999-3 Don Beal (UM). The Nature of Minimax Search

1999-4 Jacques Penders (UM). The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB). Empowering Communities: A Method for the Legitimate User-DrivenSpecification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU). Re-design of compositional systems

1999-7 David Spelt (UT). Verification support for object database design

1999-8 Jacques H.J. Lenting (UM). Informed Gambling: Conception and Analysis of a Multi-Agent Mechanismfor Discrete Reallocation.

2000-1 Frank Niessink (VU). Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE). Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UvA). Sociaal-organisatorische gevolgen van kennistechnologie;een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU). ETAG, A Formal Model of Competence Knowledge for User InterfaceDesign

2000-5 Ruud van der Pol (UM). Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU). Programming Languages for Agent Communication

2000-7 Niels Peek (UU). Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coupe (EUR). Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI). Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI). Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI). Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU). Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU). Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA). Learning as problem solving

2001-4 Evgueni Smirnov (UM). Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU). Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU). Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU). Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU). A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL). Towards Distributed Development of Large Object-Oriented Models,Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA). Modeling and Simulating Work PracticeBRAHMS: a multiagent modeling and simulation language forwork practice analysis and design

2001-11 Tom M. van Engers (VUA). Knowledge Management:The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU). Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT). Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT). Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU). The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU). The Private Cyberspace Modeling ElectronicEnvironments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL). Applied legal epistemology; Building a knowledge-based ontology ofthe legal domain

2002-07 Peter Boncz (CWI). Monet: A Next-Generation DBMS Kernel For Query-IntensiveApplications

2002-08 Jaap Gordijn (VU). Value Based Requirements Engineering: Exploring InnovativeE-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB). Integrating Modern Business Applications with Objectified LegacySystems

2002-10 Brian Sheppard (UM). Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU). Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva). Processing XML in Database Systems

2002-13 Hongjing Wu (TUE). A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU). Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT). Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU). The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UvA). Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU). Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU). Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD). Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT). Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UvA). Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT). Development and specification of virtual environments

2003-07 Machiel Jansen (UvA). Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM). Repair Based Scheduling

2003-09 Rens Kortmann (UM). The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT). Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT). Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT). Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM). Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN). Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD). Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI). Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT). Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM). Learning Search Decisions

2004-01 Virginia Dignum (UU). A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT). Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU). A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UvA). Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR). Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD). The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM). Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM). Politie en de Nieuwe Internationale Informatiemarkt, Grensregionalepolitiele gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU). For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UvA). Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU). Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT). Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT). Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU). Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU). Multi-Relational Data Mining

2004-16 Federico Divina (VU). Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM). Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA). Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT). Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode). Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UvA). Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)). AI techniques for the game of Go

2005-03 Franc Grootjen (RUN). A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT). Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UvA). Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM). Adaptive Game AI

2005-07 Flavius Frasincar (TUE). Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE). A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU). Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UvA). Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU). Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR). Distributed Simulation in Industry

2005-13 Fred Hamburg (UL). Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU). Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU). Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU). Usability of XML Query Languages

2005-17 Boris Shishkov (TUD). Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU). Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM). Situated Representation

2005-20 Cristina Coteanu (UL). Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT). Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE). Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU). Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UvA). The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU). Building Web Service Ontologies

2006-05 Cees Pierik (UU). Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU). Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT). XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT). Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM). Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU). Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT). Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU). Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU). Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU). Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU). CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU). Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU). User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UvA). Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU). Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT). Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN). Aptness on the Web

2006-22 Paul de Vrieze (RUN). Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU). Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU). Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU). Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT). Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI). Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UvA). Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT). Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG). Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU). Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU). Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL). Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UvA). Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT). To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU). Modeling of Change in Multi-Agent Organizations

2007-09	David Mobach (VU). Agent-Based Mediated Service Negotiation
2007-10	Huib Aldewereld (UU). Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols
2007-11	Natalia Stash (TUE). Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System
2007-12	Marcel van Gerven (RUN). Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
2007-13	Rutger Rienks (UT). Meetings in Smart Environments; Implications of Progressing Technology
2007-14	Niek Bergboer (UM). Context-Based Image Analysis
2007-15	Joyca Lacroix (UM). NIM: a Situated Computational Memory Model
2007-16	Davide Grossi (UU). Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems
2007-17	Theodore Charitos (UU). Reasoning with Dynamic Networks in Practice
2007-18	Bart Orriens (UvT). On the development an management of adaptive business collaborations
2007-19	David Levy (UM). Intimate relationships with artificial partners
2007-20	Slinger Jansen (UU). Customer Configuration Updating in a Software Supply Network
2007-21	Karianne Vermaas (UU). Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005
2007-22	Zlatko Zlatev (UT). Goal-oriented design of value and process models from patterns
2007-23	Peter Barna (TUE). Specification of Application Logic in Web Information Systems
2007-24	Georgina Ramirez Camps (CWI). Structural Features in XML Retrieval
2007-25	Joost Schalken (VU). Empirical Investigations in Software Process Improvement
2008-01	Katalin Boer-Sorbán (EUR). Agent-Based Simulation of Financial Markets: A modular, continuous-time approach
2008-02	Alexei Sharpanskykh (VU). On Computer-Aided Methods for Modeling and Analysis of Organizations
2008-03	Vera Hollink (UvA). Optimizing hierarchical menus: a usage-based approach
2008-04	Ander de Keijzer (UT). Management of Uncertain Data - towards unattended integration
2008-05	Bela Mutschler (UT). Modeling and simulating causal dependencies on process-aware information systems from a cost perspective
2008-06	Arjen Hommersom (RUN). On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
2008-07	Peter van Rosmalen (OU). Supporting the tutor in the design and support of adaptive e-learning
2008-08	Janneke Bolt (UU). Bayesian Networks: Aspects of Approximate Inference
2008-09	Christof van Nimwegen (UU). The paradox of the guided user: assistance can be counter-effective
2008-10	Wauter Bosma (UT). Discourse oriented summarization
2008-11	Vera Kartseva (VU). Designing Controls for Network Organizations: A Value-Based Approach
2008-12	Jozsef Farkas (RUN). A Semiotically Oriented Cognitive Model of Knowledge Representation
2008-13	Caterina Carraciolo (UvA). Topic Driven Access to Scientific Handbooks
2008-14	Arthur van Bunningen (UT). Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT). The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU). Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD). Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM). Adaptive Active Vision

2008-19 Henning Rode (UT). From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UvA). Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.

2008-21 Krisztian Balog (UvA). People Search in the Enterprise

2008-22 Henk Koning (UU). Communication of IT-Architecture

2008-23 Stefan Visscher (UU). Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU). Using background knowledge in ontology matching

2008-25 Geert Jonker (UU). Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT). Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU). Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN). On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT). Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU). Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM). Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT). Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UvA). Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU). Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT). Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN). Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU). Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT). A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN). Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN). Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU). Understanding Classification

2009-07 Ronald Poppe (UT). Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU). Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN). Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UvA). Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UvA). Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin). Operating Guidelines for Services

2009-13 Steven de Jong (UM). Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU). From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA).

2009-15 Rinke Hoekstra (UvA). Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT). New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT). Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI). Armada, An Evolving Database System

2009-19 Valentin Robu (CWI). Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU). Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM). Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT). Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU). Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA). Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI). RAM: Array Database Management through Relational Mapping

2009-26 Fernando Koch (UU). An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU). Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT). Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT). Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI). Balancing Vectorized Query Execution with Bandwidth-Optimized Storage

2009-31 Sofiya Katrenko (UvA). A Closer Look at Learning Relations from Text