The case for holistic query evaluation

Konstantinos Krikellas

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

In this thesis we present the holistic query evaluation model. We propose a novel query engine design that exploits the characteristics of modern processors when queries execute inside main memory. The holistic model (*a*) is based on template-based code generation for each executed query, (*b*) uses multithreading to adapt to multicore processor architectures and (*c*) addresses the optimization problem of scheduling multiple threads for intra-query parallelism.

Main-memory query execution is a usual operation in modern database servers equipped with tens or hundreds of gigabytes of RAM. In such an execution environment, the query engine needs to adapt to the CPU characteristics to boost performance. For this purpose, holistic query evaluation applies customized code generation to database query evaluation. The idea is to use a collection of highly efficient code templates and dynamically instantiate them to create query- and hardware-specific source code. The source code is compiled and dynamically linked to the database server for processing. Code generation diminishes the bloat of higher-level programming abstractions necessary for implementing generic, interpreted, SQL query engines. At the same time, the generated code is customized for the hardware it will run on. The holistic model supports the most frequently used query processing algorithms, namely sorting, partitioning, join evaluation, and aggregation, thus allowing the efficient evaluation of complex DSS or OLAP queries.

Modern CPUs follow multicore designs with multiple threads running in parallel. The dataflow of query engine algorithms needs to be adapted to exploit such designs. We identify memory accesses and thread synchronization as the main bottlenecks in a multicore execution environment. We extend the holistic query evaluation model and propose techniques to mitigate the impact of these bottlenecks on multithreaded query evaluation. We analytically model the expected performance and scalability of the proposed algorithms according to the hardware specifications. The analytical performance expressions can be used by the optimizer to statically estimate the speedup of multithreaded query execution.

Finally, we examine the problem of thread scheduling in the context of multithreaded query evaluation on multicore CPUs. The search space for possible operator execution schedules scales fast, thus forbidding the use of exhaustive techniques. We model intra-query parallelism on multicore systems and present scheduling heuristics that result in different degrees of schedule quality and optimization cost. We identify cases where each of our proposed algorithms, or combinations of them, are expected to generate schedules of high quality at an acceptable running cost.

Acknowledgements

There is an ancient Greek saying, according to which we owe our life to our parents and our welfare to our teachers. During the studies towards my Ph.D. at Edinburgh my professors and colleagues, my friends and my family members have firmly supported and eased my efforts. I feel lucky to have pursued my studies with these people by my side; they helped me focus on and enjoy my research work and get the most out of the time I spent as a Ph.D candidate.

Back to 2006, when I decided to move to Edinburgh for my Ph.D. studies, little did I know about my supervisor, Stratis D. Viglas. It soon became clear to me that I could not have made a better choice. I was expecting that Stratis would help me get a better understanding of my research area and teach me how to innovate and communicate my ideas. I was expecting that, being a phenomenal researcher himself, he would push me hard to conduct research at the highest level, with no compromises or excuses. What I never expected was that my supervisor would also be a father and a close friend to me. He spent infinite hours chatting with me, wisely bringing me down to earth when I thought I had conquered the world and standing by me when I was disappointed from bad results or paper rejections. He almost spent as much time working on our papers as I did – will never forget June 2009, when I was at Cambridge and we kept exchanging emails until (or even after) midnight for days, to catch the conference deadline. He tried hard to absorb my stress, to patiently correct my mistakes and to transfer his research skills to me. No student can ask for more.

Marcelo Cintra, my second supervisor, worked hard to provide me with a background in an area I had never studied before. He taught me to be realistic and patient with my research work. He kept challenging me to persuade him that my ideas were sound; that way, he helped me improve my communication skills and identify new research directions. Marcelo has always been supportive and optimistic about our work; this may seem simple but sometimes proves invaluable.

Peter Buneman, leader of the Database Group, has closely monitored my progress during the past three years. His advice and guidance have always been greatly appreciated; however, I was more impressed by his humbleness and openness, considering that he is a legend in the Computer Science community. I also want to thank the School of Informatics and the Engineering and Physical Sciences Research Council (EPSRC) for funding my studies; having no financial worries allowed me to focus on my work. I hope the outcomes justify their decision to sponsor my postgraduate studies.

My doctoral studies were interrupted twice because I interned with Microsoft Research at Cambridge. During my internships, I had the chance to conduct research in an astonishing environment and collaborate with senior researchers of the calible of Foula Vagena, Sameh Elnikety, Orion Hodson and Ant Rowstron. Sameh was the mastermind behind our research work. He spent most of his time transferring his knowledge to me, both on his research area and on how a researcher and an engineer should behave. He was my mentor and will definitely remain a good friend. Orion gave me an insight of what constitutes a skillful software engineer and how successful software products are built and extended. He taught me that code maintainability is sometimes as important as performance and encouraged me to focus on both these aspects during software development.

Foula has been my third supervisor. She helped me apply to Microsoft Research and supported me throughout my internship by any means, wisely guiding me on how to get the most out of it. She spent (and spends) lots of her precious time discussing about my future occupation alternatives, the professional skills I need and my approach to the job-hunting process. It is not common to meet people who are willing to help without asking for any return, who are kind, friendly and supportive under all circumstances.

Moving to Edinburgh brought a change to my living environment, as I had never lived abroad before. However, I was lucky enough to join such an amazing social circle of Greek PhD students. My flatmate and officemate, Giannis, has been one of the few people who can withstand my presence for so many hours a day. Besides being a gifted engineer and researcher, his vivid and spontaneous sense of humour helped me forget the stress of work. Savvas was the oldest Greek student in our office, but this did not stop us from teasing him all the time. Probably the most kind, polite and outgoing student I have met in the UK. Paul “Holeras” was the newby in our office, always full of energy and ready to hit the pubs. Manos and Kostas were so unbelievably crazy guys; when they met Gianni, chaos and disorder prevailed – beer showers, mermaid statue feeding, stolen bike saddles and awful public karaoke performances are only a small sample of their actions. Iro “Rourou” and Giorgia “Jojoka” patiently tolerated boys’ behaviour and offered their place for our gatherings and our crazy parties – rumors say that pieces of broken plates are still recovered from the fireplace. Jerry “worm” was as crazy and humouristic as only a Thessalonikian can be. Panagiotis, Anastasia, Kostas “barbas” and Ifigenia also joined our gatherings and contributed to our company’s careless spirit.

While staying at Cambridge, I often visited Niko at Cranfield. I first met him during the first year of our undergraduate studies; he has been a true friend ever since. I also had the chance to meet his flatmates, Elias and Takis, the twins from Tripoli. They always made me feel at home and I enjoyed barbecueing and having all these cheerful chats with them. I am sure our friendship will last for long.

During the past three years, I have spent some time with a few girls that helped me relax and take my mind away from work; this was crucial for my internal balance and mental outlook. I want to thank Anna, Alex and Mihaela for the time they spent with me and the patience they showed. When I first met Jane at Cambridge, I would never expect to spend so much time with her. She has been by my side from then on, enduring distance, my focus at work and my sharp and selfish behaviour. She will always have a place in my heart, even when we separate.

Finally, all my family members have been more than supportive during my studies. I am blessed to be the last of five children: Ino, Dimitris, Maria and Gianna. They have all been there when I needed them and alleviated any homesickness feelings. My siblings in law, Nektarios, Nektarios and Nektaria, have been as close to me as my blood siblings. I already have seven nephews and nieces, more to come in short. I regret for not being present when my nieces, Elpida, Theodora and Stella, were born – will try to compensate in the future. Above all, I would like to thank my parents, Odysseas and Stella for everything they have done for me. They have sacrificed every luxury from their life so as to make sure that all their children get the highest level of education they could afford. Through their lives, they set us an example of how to stand on our knees, to work hard, to always opt to excel and improve our skills. They taught us that, if we trust our abilities and spare no pains to work, everything is feasible. Nothing makes them more happy than learning about their children' progress and prosperity. I dedicate this dissertation to them.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Konstantinos Krikellas*)

To my parents
Odysseas and Stella

“Αἰέν ἀριστεύειν καί ὑπείροχον ἔμμεναι ἄλλων, μηδέ γένος πατέρων αἰσχυνέμεν.”
(Ever to excel, to do better than others, and to bring glory to your forebears).

Homer's Iliad

“῝Εν οἶδα ὅτι οὐδὲν οἶδα.”
(I know one thing, that I know nothing).

Socrates

Table of Contents

Chapter 1

Introduction

In this thesis we study issues in the areas of query evaluation and optimisation. Our focus is on query execution when the query dataset fits inside main memory. Efficient query execution in this environment requires the query engine to adapt to the characteristics of modern hardware.

The architecture of most commercial database systems dates from two to three decades back [72]. In that era, query processing algorithms focused on minimising disk I/O, since the RAM was too limited to hold even the dataset of short-running queries while main memory accesses were much faster than accesses to hard disk drives. This environment has steadily changed over time. The capacities of both hard disk drives and main memory have increased by almost three orders of magnitude within the last fifteen years. For contemporary servers with large amounts of memory it is certainly conceivable for a large portion of the on-disk data – or even the entire database – to fit in main memory. In such cases, the difference in access latency between the processor's registers and main memory leads to processor stalls until data is fetched to the CPU pipeline. In data-intensive applications, memory accesses have been shown to become the performance bottleneck [3].

To mask the slow access rate to main memory, current processors use multiple levels of cache memory that buffer recently accessed data closer to the CPU. The capacity of each level of cache memory is typically orders of magnitude smaller than the query dataset. For data-intensive workloads it is imperative to adjust the data flow to take advantage of the buffering effect of the cache memories. This, however, is a complex process, as data transfer inside the cache hierarchy is controlled by hardware. It is therefore impossible for any application to have absolute control of the cache content.

Moreover, the advent of multicore processors has not brought any dramatic improvements to the efficiency of query engines in main-memory query execution. Multicore architectures provide hardware support for the parallel execution of multiple threads. This is achieved by replicating execution pipelines and caches in separate cores on the same die. Though this design increases the processor's raw computational power, it also introduces contention when all cores attempt to simultaneously access main memory; this is a usual case in data-intensive workloads. Moreover, accesses and updates of shared data structures require synchronisation, further restricting scalability in multithreaded query execution. Parallel query execution has extensively been studied before for shared-nothing and shared-memory parallel systems (*e.g.*, [25, 34]). However, the presence of cache levels that are either private or shared by multiple cores requires revisiting the design of query engines to fine-tune parallelism for multicore CPUs.

To improve the performance of contemporary query engines during main-memory query execution, we introduce the *Holistic Query Evaluation* model. We propose a novel query engine architecture that is globally optimised with regard to the characteristics of modern hardware. We term our model "holistic" because query execution is optimised both for the query operations included in each query and the specifications of the host hardware platform. To reduce processor stalls during query evaluation, the query engine generates query-specific source code, which is then compiled, linked to and executed by the database back-end to produce the query results. To exploit the multicore design of modern processors, the holistic model introduces a framework for multithreaded query execution that incurs minimal synchronisation overhead and favours cache locality to boost scalability. Finally, to enhance intra-query parallelism, the query engine uses heuristics for distributing the available budget of hardware-supported threads to query operators and generating execution schedules of high quality.

We present the various aspects of the holistic model in three parts. In the first part, we describe the proposed query engine architecture and give our template-based approach for code generation, focusing on single-threaded execution. In the second part, we provide a uniform framework to extend query operations for multithreading so as to exploit multicore architectures. Furthermore, we present our methodology to analytically estimate the speedup from multithreaded execution, based on hardware parameters and performance metrics. The proposed framework and analytical model are not restricted to query engines using code generation; they can directly be applied to any query engine and hardware platform. Finally, we investigate how our speedup estimations for multithreaded execution can be used by the query optimiser to produce an execution schedule of high optimality. This is particularly interesting in (but not restricted to) systems with multicore CPUs, since the design of the latter allows for fine-grained intra-query parallelism.

The database community has realised that the generic design of contemporary query engines proves inadequate under common workloads and hardware setups [71, 72] due to their OLTP-oriented architecture. Previous work has argued that efficiency in main-memory query execution requires changes to the storage layer, so as to reduce the size of the data required to be transferred to the cache hierarchy. We believe that changing the storage layer is a radical departure from conventional database design, that may have side-effects on concurrency control and recovery. On the contrary, we focus on the established implementation model, namely the *iterator* model, that is extensively used to build interpreted query engines for SQL. Interpretation means genericity and composability – precisely what the iterator model provided when it was introduced (and what it still provides). However, that comes at a cost on modern CPUs due to the iterators' inflated needs for both executed instructions and processed data.

Our objective is to introduce a novel query engine design that would prove efficient in main-memory execution and also maintain the compositional aspects of the iterator model. Moreover, we have decided to use the established storage layout of sequentially storing tuples inside pages, *i.e.*, the N-ary Storage Layout (NSM), so as not to affect orthogonal modules of the database system. To that end, we propose the use of per-query code generation, compilation and dynamic linking to the database back-end. Code generation allows the customisation of the executed code for the characteristics of each query in particular, using succinct and highly efficient code segments. Furthermore, during code generation and compilation it is possible to optimise the code for the hardware platform it will run on, thus having another optimisation step on top of traditional query optimisation.

The idea of on-the-fly code generation is not new; a primitive form of this technique was introduced in System-R [9], though its use resembled more of modern query interpretation. However, back then compiler optimisations and hardware-friendliness of the native code were not as advanced as they are now and the cost of code generation and dynamic linking outweighed their benefits. Moreover, on-the-fly generation and compilation of query-specific code was tricky and error-prone since there was no common interface to interconnect different query operations, as in the case of iterators.

This situation has radically changed during the past decades. Modern database servers and compilers are powerful enough to generate and compile code for each executed query at an acceptable cost. The issue that remains is how different query operators can be instantiated and interconnected inside the generated source code, using a uniform and simple to implement framework. To mitigate this, we present a template-based framework that uses composition to produce query-specific code. The templates use a common nested-loops code segment, that is appropriately modified to implement various sort- or partition-based query evaluation algorithms. We do not use memory-resident hash tables, so as to avoid uncontrollable random access patterns; instead we present algorithms that combine partitioning with sorting to exploit cache locality. Operator interconnection is done using temporary intermediate tables, that are handled by the storage manager and therefore can be either disk- or memory-resident.

Code generation improves the efficiency of the executed code in the following ways: (*a*) the number of function calls during query evaluation is minimised, (*b*) the code layout fits the superscalar design of modern processors, and (*c*) data locality is enhanced by making optimal use of cache-resident data. Moreover, code compilation allows the use of compiler optimisation techniques targeting each individual query, an extra optimisation level on top of conventional query optimisation. The proposed framework is easily extendable and flexible enough to accommodate sophisticated query evaluation algorithms, such as multi-way join evaluation, with only moderate changes.

Using this framework, we have developed a prototype query engine that we term *Holistic Integrated Query Engine* (HIQUE). We have studied its performance extensively by comparing it to both iterator-based solutions and existing database systems, using a variety of workloads. The results (*a*) quantify the advantage of per-query code generation over generic query operator implementations, and (*b*) demonstrate a superiority of the holistic approach over both iterator-based and hardware-conscious systems, therefore proving its viability as an alternative query engine design.

The next challenge in the development of the holistic model is its extension for multicore architectures. According to previous work in parallel query processing, the starting point is to split the input of each query operator to disjoint partitions and process the latter independently. However, the presence of the cache hierarchy and the high latency of uncached memory accesses complicate parallel query execution and restrict the scalability of naïve implementations when ported on multicores. Furthermore, parallelism is lightweight since it is based on threads instead of processes, as it is common in shared-nothing and shared-memory systems. This means that the overhead for thread synchronisation and scheduling should be limited, otherwise it will become comparable to the time spent for query execution, thus reducing the performance gains from parallel execution.

Based on these observations, we develop a uniform and widely applicable framework that extends query evaluation algorithms for multithreading. The framework divides execution to two stages: partitioning and processing. During partitioning, the input is broken to disjoint splits that are processed independently. The number of splits is equal to the number of the threads that can efficiently be supported by hardware inside the cores. That way, the scheduling overhead is negligible; threads are started and run with minimal synchronisation needs, until they have fully processed their input split. The output of the first stage is a number of partitions (equal to or higher than the number of threads used) that contain tuples with range- or hash-partitioned predicate values. The next step is to run the actual query processing algorithm (*e.g.*, join evaluation or aggregation) over the partitions that the previous step produced. Again, the number of threads utilised is tightly connected to the hardware support capabilities for multithreading. Each thread processes a disjoint range of partitions, so no synchronisation is required during this step.

After developing our multithreading framework and integrating it to HIQUE by extending the code templates and the generator accordingly, we were concerned with the measured performance. The question that arose is whether the speedup we measured was good and, if not, what factor was restricting performance. Our framework was carefully designed to have negligible synchronisation and scheduling overhead, so our intuition was that concurrent accesses to the main memory incur inflated cost, as it has also been shown in [57]. To model this, for each query operation we define the *multithreaded utility ratio* as the ratio of the time spent for fetching data to the total processing time that corresponds to each input unit (split, partition or page). High values of this ratio denote fetch-dominated operations; in this case memory accesses become effectively serialised, thus restricting scalability. On the contrary, low values of the utility ratio show that there is sufficient computational load to overlap with data fetching, so thread contention for the memory bus is limited; this leads to almost linear speedups. Note that the definition of the multithreaded utility ratio is not tied to the holistic model; this ratio can be calculated for the algorithms of any query engine implementation running on any hardware platform.

Using the methodology above, we have devised analytical cost expressions and speedup estimations for the supported query evaluation algorithms. These formulas are based on hardware parameters and performance metrics. We then used HIQUE to experiment with different query parameters, such as input tuple size and predicate selectivity, for the supported query evaluation algorithms and compared the measured performance with the estimated one. The results verify that our scalability model adequately captures the performance trends, giving reliable estimations of the expected speedup from multithreaded query execution. Furthermore, our proposed framework shows substantial performance merits, achieving considerable scalability in all cases and even approximating linear behaviour in certain ones.

The last part in the development of the holistic model addresses the optimisation problem of thread scheduling for intra-query parallelism. Multicore processors provide hardware support for parallel execution of multiple threads. Our framework for multithreading allows the efficient use of many threads for each query operator. In a complex query, there are many possible ways of allocating threads to the various query operators: each operator can be assigned from one to all available threads, while its execution can start as soon as its input has (partially or totally) been generated. Since the holistic model can give static estimations of the speedup from multithreaded execution of each query operator, we study how these estimations can be leveraged by the query optimiser to produce an execution schedule of high quality.

Our analysis starts with modelling the optimisation problem. We show how a query task tree can be built from a bushy query execution plan when various forms of intra- and inter-operator parallelism are taken into consideration. Each query task comprises one or more (pipelined) query operators that can be individually scheduled to one or more threads, so the query task tree is the input of the scheduler. We avoid tying our modelling approach to HIQUE or any other query engine design by explaining how different implementation alternatives can be modelled to exploit the characteristics of multicore processors.

The number of possible query operator combinations in the execution schedule is limited by data and resource restrictions. However, the use of bushy execution plans and the increased thread budget of modern multicore CPUs counterbalance these restrictions and increase the size of the search space. It has been shown in [27] that the number of possible execution schedules grows exponentially to the number of available threads and the number of query operators. This renders the use of exhaustive techniques practically inapplicable. To that end, we introduce a family of heuristics-based techniques examining subplans that are likely to give schedules of high optimality. Our algorithms search opportunities for parallel execution of either (*a*) the descendants of a single node until a certain depth, or (*b*) all the nodes of the same level of the query task tree. In both cases, scheduling uses a parallel execution window of width equal to the number of hardware-supported threads. The objective is to specify groups of tasks that, when scheduled together and following a specific thread distribution, (*a*) minimise delays due to data and resource dependencies and (*b*) achieve the highest degree of parallelism for the scalability potential of all tasks, as given by our analytical scalability model for multithreading.

To evaluate the optimality and the execution cost of the proposed techniques, we simulated the execution of our scheduling algorithms for a wide range of bushy query plans using different thread budgets and scalability estimations. The results show that our heuristics produce schedules of high quality while incurring acceptable optimisation cost and, hence, they are practically applicable. Finally, we present cases where each of the proposed algorithms, or combinations of them, are expected to give schedules of high quality at an acceptable execution cost.

The rest of this thesis is organised as follows: in Chapter 2 we elaborate on modern processor architecture, describe the design of modern query engines and present previous work in main-memory query execution. In Chapter 3 we present our template-based approach to per-query code generation, along with the proposed algorithmic extensions for main-memory query evaluation. In Chapter 4 we introduce our framework for multithreaded query evaluation and present our modelling methodology for estimating scalability. In Chapter 5 we model the optimization problem of scheduling threads to query operators and introduce a family of heuristic-based scheduling algorithms. Finally, we draw our conclusions and identify future work directions in Chapter 6.

Chapter 2

Background

2.1 Hardware primer

Modern CPUs process multiple instructions simultaneously through pipelining and superscalar execution [41]. Though individual processor architectures vary widely, most modern CPUs follow a common design, as depicted in Figure 2.1. The key difference between CPU core architectures is whether their pipeline follows *out-of-order* or *in-order* execution. In the first case, if one instruction has to wait for a data transfer, or for another instruction to be executed first, other instructions following it in the pipeline can take its turn and execute. This allows the CPU to (partially) cover memory access and instruction dependency stalls. Though out-of-order execution hides stalls on single-threaded execution through *Instruction Level Parallelism* (ILP), it offers limited advantages when the executed code includes consecutive memory requests and long data and control dependency chains; the latter are common in database workloads. Out-of-order execution also requires complex circuits occupying valuable chip surface, which could be otherwise exploited to include more cache memory, or more registers. On the contrary, processors using in-order execution have a simpler pipeline that stalls on branches or memory operations. For those types of processor it is common practice to adopt hardware support for multithreading, in the sense that they switch to a thread ready to execute when the running one is stalled. An in-depth study of the differences of these two designs on database workloads can be found in [39]. That report can be used as indicative of trends; it is based on simulation results using a database system not optimised for main-memory execution, while it does not take into account the effect of other major processor subsystems, such as the hardware prefetchers.

The superscalar design proves very efficient in performing independent calcula-

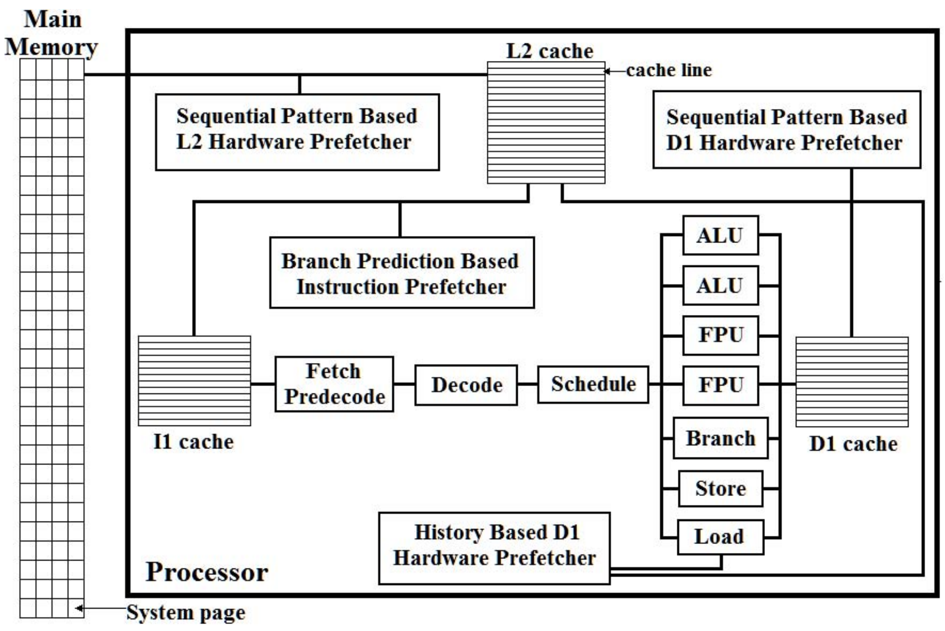

Main Memory
L2 cache
cache line
Sequential Pattern Based L2 Hardware Prefetcher
Sequential Pattern Based D1 Hardware Prefetcher
Branch Prediction Based Instruction Prefetcher
ALU
ALU
FPU
FPU
Branch
Store
Load
I1 cache
Fetch Predecode
Decode
Schedule
D1 cache
History Based D1 Hardware Prefetcher
Processor
System page

Figure 2.1: The pipeline architecture of modern CPUs

tions. It leads, however, to undesirable processor stalls when the execution pipeline contains dependent instructions, such as branching conditions or successive uses of the same data. For branching, the usual countermeasure is to speculatively continue executing the instruction stream indicated by one of the targets of the branch. The choice of target is based on the history of previous outcomes for a small number of the most recent instantiations. In case of a branch misprediction, all instructions that follow the mispredicted branch in the pipeline have to be flushed and the new and correct instructions loaded, leading to wasted CPU cycles. This used to be a significant source of stalls, especially for processors that adopted a deep pipeline, like the Intel Pentium 4.[1] Branch misprediction stalls, however, have a limited impact on the performance of modern CPUs, as the hardware community has abandoned deep pipeline designs.

Although most processors have multiple execution units for arithmetic and logical operations, they only have one or two units for memory operations. Therefore, they can perform only a few memory operations simultaneously, thus inevitably restricting superscalar operation in data-intensive workloads. The large difference in latency between accessing the main memory and the processor's registers is countered with multiple levels of cache memory. The levels closer to the processor are smaller but faster to access than the ones closer to the main memory. Cached data is organised in

[1]The Intel Pentium 4 processor had the deepest pipeline of its class – 31 stages.

fixed-length chunks, each termed a *cache line*, that are also the data exchange unit with the main memory. Most modern processors incorporate a very fast to access level-1 (L1) cache of 32-128KB divided in the Instruction (I1) and the Data (D1) cache; a moderately fast to access 0.5-12MB level-2 (L2) cache; and, in some models, an even larger but much slower to access level-3 (L3) cache.

Caches work by exploiting both temporal locality (data tend to be repeatedly accessed within a short period) and spatial locality (contiguously allocated data tend to be accessed in unison). Their non-blocking operation, combined with superscalar execution, allows for multiple pending memory operations, thus overlapping fetch latencies. Data-intensive workloads, however, restrict this operation. The cache controller can only serve a limited number of concurrent requests and therefore becomes saturated. While caches can serve a large percentage of the processor's instruction and data requests, the remaining ones (cache misses) are extremely expensive and can become a performance bottleneck.

To assist the caches, modern processors incorporate hardware prefetching units that identify the instructions and data likely to be accessed shortly and prefetch them into the appropriate cache level. That way, it is possible to avoid many cache misses and reduce processor stalls. A closer look at the operating principles of hardware prefetchers ([4, 26]) shows that they can identify future accesses when data is linearly accessed in fixed strides of cache lines. The most sophisticated designs, as depicted in Figure 2.1, employ multiple prefetching units tightly coupled with the cache hierarchy. The *sequential pattern based* prefetchers for the D1- and the L2-cache in Figure 2.1, are capable of detecting purely sequential patterns and can prefetch the next line upon a cache miss. The *history based* prefetcher for the D1-cache in Figure 2.1 closely monitors the addresses touched by the processor. It is capable of identifying more complex-strided access patterns by (*a*) keeping the history of accesses for a small number of the most frequently accessed entries, and (*b*) tracking the distance (stride) between successive fetches. Modern hardware prefetchers can identify and prefetch data exactly when it is to be accessed, without polluting the cache with data not immediately needed. There is usually a startup penalty until the prefetcher "locks on" the strided data access pattern.

In Figure 2.2 we show the impact of hardware prefetching for an Intel Core 2 Duo 6300 processor, operating at 1.86 GHz. We present the data access latency for variable examined block sizes, for sequential (forward and backward) and random access patterns. The measurements were extracted by the RightMark Memory Analyzer [65] program. While all accesses to the D1-cache have a uniform cost of 3 cycles, there is

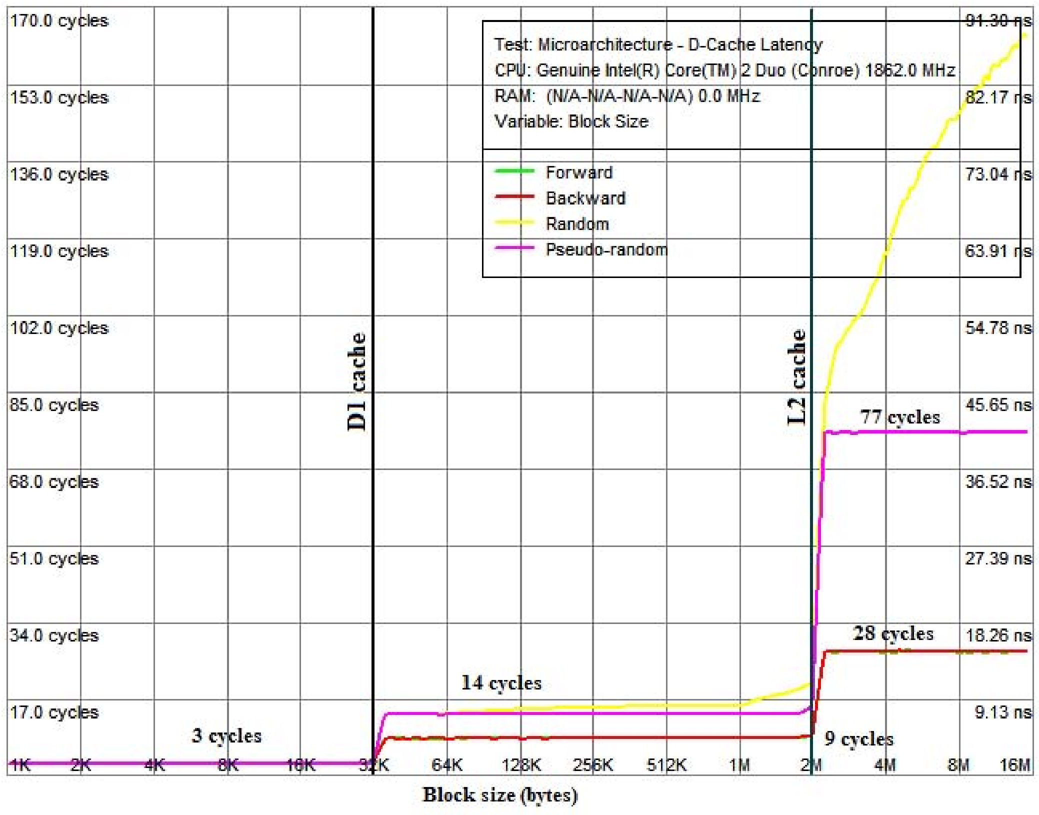


Figure 2.2: Memory access latency for the Intel Core 2 Duo processor

a significant difference when switching from sequential to random access in the L2-cache: the former takes 9 cycles and the latter 14 cycles. This gap grows further when a data access cannot be served from caches, as sequential access in main memory costs 28 cycles, while random access costs 77 cycles or more. Note that if we provide the CPU with sufficient computational load for out-of-order execution, the time the prefetcher needs to fetch a cache line can be masked by independent computations so the CPU stall time is minimised. This operation, however, can only take place in the D1-cache: the execution pipeline of the CPU is too short to fully hide an L2-cache miss through out-of-order execution. Furthermore, in many architectures, speculative accesses outside the cache hierarchy are inhibited, to avoid the penalty of performing expensive memory operations that might be dropped at a later stage.

The latest trend in processor architecture is the integration of multiple processing cores on the same die. Termed *chip multiprocessors* (CMPs), multicore chips natively support parallel execution, while combining scalability with energy efficiency [41]. Multicore chips have been implemented in various ways. The main difference is the

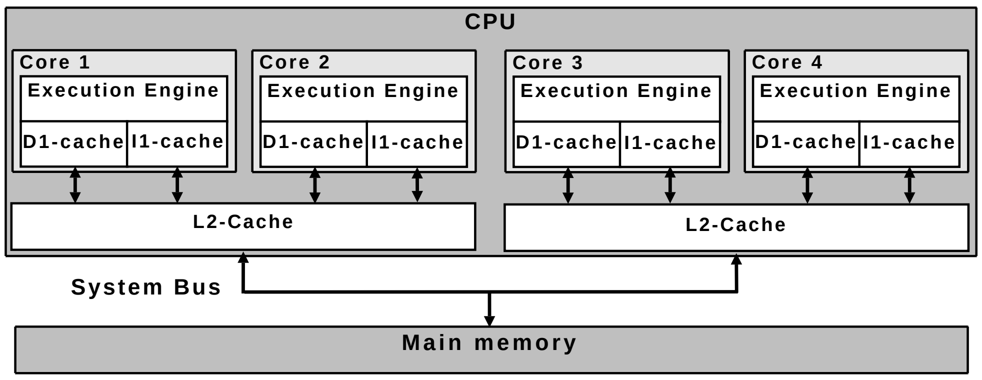


Figure 2.3: The architecture of the Intel Xeon E5420

type of parallelism supported by each core. Some processor designs, *e.g.*, the Intel Quad Core and the AMD Phenom, support out-of-order execution and ILP; alternatively, the pipelines of the Sun UltraSPARC T2 and the IBM Power 6 support only in-order execution but there is in-core support for TLP. There are also hybrid designs, *e.g.*, the Intel Core i7 CPU, which combine out-of-order execution with hardware supported multithreading inside the cores, similar to Simultaneous Multithreading [42].

Multicore designs also differ in terms of the memory hierarchy, specifically whether on-chip caches are shared between all or some of the cores. In Figure 2.3 we sketch the Intel Xeon E5420 quad-core processor: each pair of cores shares a common L2-cache and cores from different pairs communicate through the memory bus. In other designs, *e.g.*, the AMD Phenom and the Intel Core i7, each core has its own L1- and L2-caches, while all cores share a common on-chip L3-cache. The salient challenge in multicore CPUs is to keep all cores processing data at rates close to their clock. To do so, manufacturers improve memory throughput by integrating memory controllers inside the chip and using multiple memory banks. Still, if the caches and the memory are concurrently accessed by all cores, contention for their utilisation may increase the latency of memory operations and degrade performance.

As multiple cores share main memory but not necessarily individual caches, it is common practice to replicate data inside the caches of different cores to enhance parallelism. Cache coherency involves the propagation of data writes from one core to the others. When one cache line is shared between cores and is updated by one of them, the other cores invalidate their cached copy and refetch the cache line on the next access. Invalidation takes place on true sharing, *i.e.*, cores access the same data of the cache line, or on false sharing, *i.e.*, when one core updates a part of the cache line that no other core accesses. Coherency protocols "snoop" updates to all cores or use directories to maintain data sharing information [41].

Concurrent execution at the hardware level (*i.e.*, processing independently scheduled threads) does not imply synchronisation at the software level. The latter is achieved by providing hardware support for atomic operations through mutexes and *spin locking*. Each mutex is a memory word set to 0 when free and 1 when locked; to operate on the mutex, a core must have it in its D1-cache. To acquire a lock, a core continuously probes the mutex (*i.e.*, the core "spins") using the atomic *compare-and-swap* instruction. Once the lock is acquired the core executes the synchronised code and resets the lock. Each core spins on a locally cached copy of the mutex without affecting other cores. Whenever the mutex is released, cache coherency requires that the cache line containing it be invalidated and refetched. The first core to refetch the cache line will acquire the lock; other cores waiting for the lock will continue to spin.

2.2 Modern query engine design

The query engines of most database systems are based on the iterator model [33]. This model provides an abstract interface used for streaming tuples across query operators, in the form of three functions: (*a*) `open()`, designating the start of information exchange and initialisation of internal operator state, (*b*) `get_next()`, for propagating tuples between operators, and (*c*) `close()`, denoting the end of processing and allowing the operators to free up their resources. Query plans are then organised as pipelined binary trees of operators communicating through iterator calls.

Iterators are used to implement various query operators. The most commonly applied and, at the same time, demanding ones are join evaluation and aggregation [30]. Efficient execution of these operations requires the input to first be either sorted or hash-partitioned. Sorting is used in merge join and in sort-based aggregation; the input tables are first sorted using the external sorting algorithm [49] and then scanned sequentially to evaluate the join predicate or calculate the aggregates. Hash join algorithms appear in numerous variants [5, 23, 48, 68, 79]: GRACE, hybrid, radix-cluster and cuckoo hash join, to name a few. Although in some variants both inputs are partitioned, this is not necessary. The usual practice is to first build a hash table for the smaller table. This hash table maps join attribute values to tuples using an appropriate hash function [49]. The larger table is then scanned and the values of its join attribute are used to probe the previously built hash table, evaluate the join predicate and retrieve the matching tuples of the smaller table.

Hash join requires only the values of the smaller input to be inserted into a hash table, while merge join requires sorting both inputs. In addition, the building phase of hash join requires one single pass over the input so its complexity is $O(N)$, while the complexity of sorting is $O(Nlog(N))$. However, the use of a hash table introduces random access patterns during probing that incur an order of magnitude higher fetching cost outside the L1-cache (see also Section 2.1 and Figure 2.2), compared to the sequential access patterns that the sort-based algorithms use extensively. Moreover, in case the hash table cannot fit in the available memory, multiple building and probing steps are required so the cost of hash join gets closer or exceeds the one of merge join in practice, despite its lower complexity. This effect is also important in main-memory execution. Unless the hash table fits in the cache hierarchy, every lookup will most likely trigger expensive cache misses. Note that the hardware prefetcher cannot prevent these cache misses since they stem from accesses due to pointer dereferencing. The cost of each such memory access will range close to $250-300$ CPU cycles, so the probing phase will proceed through massive processor stalls. Similar conclusions apply to hash-based aggregation.

In Online Analytical Processing (OLAP) and Decision Support System (DSS) applications it is a common practice to use star or snowflake schemata [10], where multiple tables connect to a common root table using primary key – foreign key constraints. In this case the workloads contain multi-way join queries with join teams, *i.e.*, sets of tables joined on a common key. For instance, the conjunction $R.a = S.b$ **and** $S.b = T.c$ **and** $T.d = A.x$ **and** $A.x = B.y$ contains two join teams, one with tables R, S, and T and another with tables T, A, and B; table T is common between the two teams. Modern query engines can identify join teams and use sophisticated algorithms such as hash teams [35] and interesting orders [67] to evaluate them. These techniques allow the pipelined evaluation of multiple joins without materialising intermediate results, thus radically reducing data fetching and processing. Furthermore, the authors of [45] generalised the idea of join teams for generic join predicates, without requiring a common interconnecting key between the joined tables. This approach proves efficient when the number of distinct values of the join attribute is low for all tables.

2.3 Previous work in main-memory query execution

It has long been known that processors are designed for complex numerical workloads over primitive data types and are not well-tailored for database workloads. After the seminal paper of [3], the database community obtained a better understanding of how databases behave on modern processors. A quantitative analysis of processor stalls was presented in [54], along with the necessary analytical tools to describe query evaluation algorithms using hardware performance terms. Ongoing research in the area has shown that, to make database systems more processor-friendly, one needs to either change the data flow of the evaluation algorithms or the storage layout, in ways conducive to the type of processing the CPU has been designed for.

In [69], performance was not only measured in terms of response time, but also in terms of processor metrics (*e.g.*, cache misses and branch mispredictions). Various modifications were proposed to improve the behaviour of join algorithms on contemporary processors. Along the same lines, the *Alphasort* algorithm [58] proposed a sorting implementation with the goal of minimising cache misses during the internal sorting and merging phases.

In the context of the iterator model, a buffering operator was proposed in [78] to increase the tuple granularity in inter-operator communication. This resulted in a measurable reduction in the number of iterator calls across the operator pipeline, but had no effect on the number of evaluation function calls in the body of the operator. In [60] it was proposed that multiple aggregation operations can be combined in a single blocking operator that executes these operations over a sequence of tuples through array computations. Common computations across the aggregation functions are performed only once and stored as intermediate results; array computations are used to evaluate aggregates, a technique more in line with the superscalar design of modern processors.

Regarding the storage layer, it was soon realised that the established N-ary Storage Model (NSM) penalised execution for the common case of only a small number of fields in each tuple being necessary during query evaluation. This led to the introduction of vertical partitioning and the Decomposed Storage Model (DSM) [20], where each tuple field is separately stored. This reduces the amount of data touched during query evaluation. At the same time, it allows the use of array computations when implementing the operators. This change of storage layout, however, implied revisiting all query evaluation algorithms. It also affected not only the design of the query engine, but other orthogonal aspects of a DBMS as well, *e.g.*, concurrency control.

Vertical partitioning was revisited in [2] with the introduction of the Partition Attributes Across, or PAX, storage model. The idea is that although pages still provide a tuple-level interface, the tuples within a page are vertically partitioned, thus greatly enhancing cache locality. This hybrid approach combines the benefits of NSM and DSM while requiring only limited changes to the database system. Though its performance is inferior to DSM's because of the intervention of the buffer manager for page manipulation, it appears to be the most promising storage model since it does not affect other parts of the system. PAX can be used in conjunction with holistic query evaluation.

A paradigm of a DBMS optimised for main-memory query execution is MonetDB [5, 55]. In addition to vertical decomposition, its entire query engine is built on the notion of array manipulation, with sophisticated query processing techniques (*e.g.*, radix-cluster hash join) having been developed in that context. Though MonetDB's engine employs a different data flow than that of traditional DBMSs, it still is an operator-based approach, tightly connected to the DSM. It also requires materialising all intermediate results, thus reducing opportunities for exploiting cache locality across separate query operators. These restrictions led to the introduction of MonetDB/X100 [6, 80, 81], where the idea of the block operator [60] was coupled with a column-wise storage layout. The use of compound vectorised primitives for performing all computations achieved performance comparable to hard-coded programs. Still, these primitives cannot be used for computation across multiple vectors in parallel, something that would better exploit the processor's registers. In addition, the introduced buffer manager, ColumnBM, can use both DSM and PAX for storage (though if PAX is used, performance is penalised).

Prefetching has been another area that has received attention, with [12, 13] presenting ways of employing software prefetching in hash join evaluation. Though this approach may improve response times, it introduces the need for dynamically calculating the prefetching distance according to the CPU's frequency, cache latencies, and the runtime load. Inaccuracies result in failing to prefetch the required data on time, or polluting the cache with not immediately needed data. In addition, the cache controller considers software prefetching instructions as hints and may ignore them if there are pending fetch requests. Most papers concerning software prefetching for database systems [12, 13, 14, 15] either provided measurements on processors without hardware prefetchers, or presented simulation results. This was unfortunate, since in [29] it was shown that modern processors mask software prefetching advantages, due to the efficiency of their hardware prefetchers. In the same vein, [17] verified that using software prefetching on a SUN UltraSparc T1 offered no gains in multithreaded aggregation, despite this architecture's lack of hardware prefetching units. Our early experiments on processors employing the x86-64 architecture (with both Intel and AMD processors) corroborated that using software prefetching is very complicated, highly dependant on the specific architecture, and offers only modest speedup under very specific circumstances. We have therefore chosen not to employ software prefetching but solely rely on the hardware prefetching mechanisms that almost most modern processors incorporate.

Chapter 3

Generating code for holistic query evaluation

3.1 Introduction

This chapter presents the application of customised code generation for the purpose of efficient database query processing. Our approach stems from template-based programming. The idea is to use code templates for the various query processing algorithms and then dynamically instantiate them and compose them in a single piece of source code that can evaluate the query. Dynamic template instantiation removes most high-level abstractions that are necessary for implementing generic query evaluators in current query engine designs. Moreover, since the code is dynamically generated, it can be customised to exploit the architectural characteristics of the hardware it will execute on. The resulting performance advantage in main-memory execution is substantial; for instance, it reaches a factor of 167 over established database technology in TPC-H Query 1. The novelty we claim and demonstrate is that template-based code generation can be generalised to efficiently process any type of query without affecting orthogonal aspects of the database system.

3.1.1 Motivation

Traditionally, query processing algorithms have focused on minimising disk I/O while their in-memory efficiency has been considered to be a secondary priority. For contemporary servers with large amounts of memory, it is conceivable for a large portion of the on-disk data – or even the entire database – to fit in main memory. In such cases, the difference in access latency between the processor's registers and main memory becomes the performance bottleneck [3]. To optimise such workloads, one needs to carefully "craft" the executed code so as to minimise the processor stall time during query execution.

Existing work has identified the data layout as the main bottleneck that prevents contemporary processor designs with multiple levels of cache memories from exploiting their full potential in database workloads. We argue that changing the storage layer is a radical departure from existing designs. We identify the biggest problem with the design of a query engine to be the compilation of SQL queries in operator plans and the generality of the common operator interface, namely the *iterator model*. The latter results in a poor utilisation of CPU resources. Its abstract implementation and the frequent use of function calls inflate the number of instructions and memory accesses required for query evaluation. The use of generic code does not permit its customisation according to the characteristics of both the executed queries and the hardware platform. SQL and query processing in main memory, however, exhibit a strong potential for exploiting just-in-time compilation. We take this idea to the extreme.

3.1.2 Code generation for query evaluation

Ideally, query processing code should optimally use the cache hierarchy and reduce the number of instructions needed for query evaluation. At the same time, one would want to keep the compositional aspects of the iterator model and not affect separate system modules. To that end, we introduce a novel query evaluation technique that we term *holistic query evaluation*. The idea is to inject a *source code* generation step in the traditional query evaluation process. The system should look at the entire query and optimise it holistically, by generating query- and hardware-specific source code, compiling it, and executing it.

Our approach has multiple benefits: (*a*) the number of function calls during query evaluation is minimised; (*b*) the generated code exhibits increased data locality, therefore making optimal use of cache-resident data; (*c*) code generation and compilation allow the use of compiler optimisation techniques targeting each individual query, an extra optimisation level on top of conventional query optimisation; and (*d*) the generated code approaches the performance of hard-coded evaluation plans. The model is flexible enough to incorporate sophisticated query evaluation algorithms and does not affect other orthogonal system aspects, such as storage management and concurrency control.

Using this framework, we have developed a prototype holistic query engine and compared its performance to both iterator-based solutions and existing database systems. The results (*a*) quantify the advantage of per-query code generation over generic query operator implementations, and (*b*) demonstrate a superiority of the holistic approach over both iterator-based and hardware-conscious systems in a variant of the Wisconsin benchmark and a subset of the TPC-H benchmark, therefore proving its viability as an alternative query engine design.

The rest of this chapter is organised as follows: in Section 3.2 we discuss the reasons why iterator-based query engines prove inefficient in main-memory execution and why their implementations cannot be optimised by the compiler. In Section 3.3 we present the design of a system based on the concept of holistic evaluation. We describe the code generation process in Section 3.4 and the holistic query evaluation algorithms in Section 3.5. In Section 3.6 we show why our template-based framework facilitates the implementation of a query engine employing per-query code generation, while query optimisation in the context of the suggested model is presented in Section 3.7. We evaluate the model in a detailed experimental study in Section 3.8, where we also compare its performance to both established and hardware-friendly designs. Finally, we present work for further reading in the area of customised code generation in Section 3.9.

3.2 Query evaluation in main memory

3.2.1 Limitations of current query engines

The query engines of most database systems are based on the iterator model [36], as described in Section 2.2. Though generic, the iterator model suffers from a large number of function calls. For each in-flight tuple the system needs to make at least two calls: one for the calling operator to request it and one for the called operator to propagate it. The number of function calls is further increased as iterators need to be generic. Their functions may be virtual to be dynamically bound to the data types they process, which implies that all field accesses and comparisons may require a function call. Each function call updates the stack register and saves/restores the contents of the processor's registers to the stack. As modern processors have tens of registers, frequent function calls may lead to significant execution overhead. This translates to a substantial percentage of the CPU time being spent without any actual contribution to result computation. Moreover, since a function call represents a jump in the executed code, it forces a new instruction stream to be loaded in the execution pipeline, thus limiting superscalar execution.

In addition to stack interaction, there is also overhead at the data level. Each iterator maintains internal state; iterator calls require several memory operations for accessing and updating the iterator state, each call potentially triggering cache misses. Moreover, iterator state manipulation interferes with data stream accesses. Even if the data access pattern is sequential it will be frequently interrupted, thus reducing the efficiency of hardware prefetching. Note that the iterator interface does not control the data flow of pipelined operators, as each operator implementation is independent. Consequently, pipelined iterator calls may introduce cache contention and evict cache lines from each other's dataset, leading to cache thrashing.

The performance of contemporary query engines is further affected by the operators being optimised for I/O and the cost model employed by the query optimiser being an I/O-based one. Since I/O is what hurts performance, most query evaluation algorithms strive to have sequential access patterns for disk-resident data without worrying so much about the access pattern in main memory; the latter may well be random, especially for some highly efficient algorithms like hash join. However, sequential access patterns in main memory exhibit cache locality and can drammatically improve performance; but they are not widely adopted in the traditional data flow of a database system.

3.2.2 Can the compiler help?

Developers typically rely on the compiler to perform a variety of code transformations that reduce processor stalls during execution. Since the compiler generates the executable code, it can optimise it for the target architecture and hardware platform. Current compilers transform the code layout in ways that (*a*) keep the execution pipeline full of independent instuctions, (*b*) distribute variables to registers in ways that encourage their reuse, and (*c*) group together accesses to the same data. All these result in increased parallelism, reduced memory accesses and maximised cache locality, thus limiting processor stalls and increasing the processor's throughput. Applicable compiler optimisations include [46] loop interchanging, peeling, skewing, unrolling, blocking and fusion, scalar expansion and renaming, and strip mining, to name a few.

Most aforementioned code transformations are loop-oriented, so one would expect the iterative access patterns in the data flow of a DBMS to be a perfect fit. This is far from true: all transformations are designed for computation over fixed-length arrays of primitive data types, like the ones found in scientific workloads. Employing the iterator model in the design of a DBMS prevents the compiler from performing these transformations. Each iterator call triggers a chain of function calls that will eventually produce a single tuple. The compiler cannot factor out this process to identify the (possibly iterative) access pattern over the input tables, as interprocedural analysis and optimisations are much more limited than intraprocedural ones. Moreover, conditions and jumps in the source code, as the ones caused by function calls, disrupt the instruction sequence and reduce the range of code the compiler can examine for optimisation opportunities. This is aggravated by information about the executed code (*e.g.*, predicate value types, offsets inside tuples, *etc.*) being only specified at run-time for each query. These ambiguities refrain the compiler from applying a substantial part of its code optimisation techniques on iterator implementations. Furthermore, most query operators are either unary or binary, so there are at most two loops within the context of a single operator. The effectiveness of code transformations, however, increases with the number of loops, as this provides more ground for parallelism and data reuse.

3.3 System overview

We argue that to overcome the problems caused by memory stalls and make database code more hardware friendly, one should take a holistic approach to query evaluation. In this section, we shall present the high-level architecture of our system, that we have named HIQUE, standing for *Holistic Integrated Query Engine*. In doing so, we shall also present the layout of the generated query-specific code and the primitives used during query evaluation. The system has been implemented in C/C++ and compiled using the GNU gcc compiler, over the GNU Linux operating system. It adopts the traditional client-server model. Multiple queries are submitted to the same back-end, which generates the query-specific code and evaluates the query.

3.3.1 Storage layer

We have adopted the N-ary Storage Model (NSM) as a storage layout, with tuples consecutively stored in pages of 4096 bytes. As we shall see in subsequent sections, the system is not tied to the NSM in any way; any other storage model, such as the Decomposed Storage Model (DSM) or the Partition Attributes Across (PAX) model, can be used and our proposals will still be applicable. Each table resides in its own file on disk, and the system's storage manager is responsible for maintaining information on table/file associations and schemata.

The buffer manager is responsible for buffering pages from and to disk and for providing concurrency control. The adopted page replacement strategy is LRU. It follows a slightly non-standard approach, as it favours entire files used by a single query, be they primary tables or intermediate results, to be kept in the buffer pool. The goal is to have complete control of when evaluation is over memory-resident or disk-resident data. That would not be feasible if memory-mapping was employed, or if the buffer manager followed a query-agnostic approach. We should note, however, that even with this bias in place the efficiency of our storage manager is not to be compared with that of commercial database systems. As we shall see in Section 3.8, the efficiency of our system is due to its holistic evaluation model, and not any other orthogonal aspects of its design.

In addition to standard files, the system employs indexes adhering to the principles of memory-resident evaluation [64]. Our indexes are fractal B$^+$-trees, with each physical page divided in four tree nodes of 1024 bytes each, as in [15]. This design results in reducing each index lookup by as much as 20% over each index node spanning the entire physical page (*i.e.*, an index node size of 4096 bytes). The nodes are still large enough to store sufficient entries and maintain a wide fan-out. Early experiments revealed that software prefetching did not exhibit any substantial performance increase – at least not for the x86-64 architecture that we experimented with. This was due to index and table pages spanning multiple cache lines. Software prefetching in such an environment quickly saturates the cache controller with multiple fetch requests. We therefore decided not to include any software prefetching instructions in the index code.

3.3.2 Query processing

The route a query follows through the system is shown in Figure 3.1. The first module is the SQL parser, which accepts an SQL query conforming to the following grammar (with the obvious semantics):

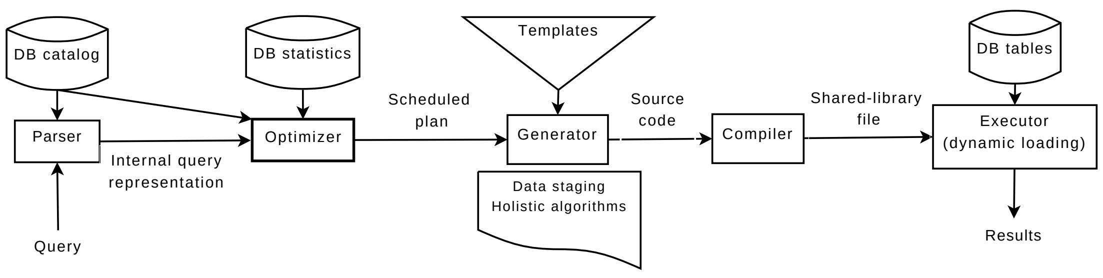


Figure 3.1: Holistic query engine overview

Query	::=	SC FC (WC)? (AC)? (OC)?
SC	::=	**select** *elem* (, *elem*)*
elem	::=	function(*arithmetic*) $\mid$ *table.attribute*
function	::=	*sum* $\mid$ *avg* $\mid$ *min* $\mid$ *max* $\mid$ *count*
arithmetic	::=	*table.attribute* $\mid$ *number* $\mid$ (*arithmetic*) $\mid$
		arithmetic arithm-op arithmetic
FC	::=	**from** *table* (, *table*)*
WC	::=	**where** P (**and** P)*
P	::=	*table.attribute* op *constant* $\mid$ *table.attribute* = *table.attribute*
AC	::=	**group by** *table.attribute* (, *table.attribute*)*
OC	::=	**order by** *table.attribute* (, *table.attribute*)*
op	::=	$> \mid < \mid = \mid \leq \mid \geq$
arithm-op	::=	$+ \mid - \mid * \mid /$

The grammar represents conjunctive queries with equi-joins and arbitrary groupings and sort orders. We do not support (*a*) complex mathematics in predicates and projection lists, (*b*) statistical functions in aggregate values, and (*c*) nested queries. We believe, however, that all these can be straightforward extensions without restricting the generality of the holistic evaluation model. The supported type system includes single and double precision integer and floating-point numbers, dates, ASCII characters and strings of fixed length.

The SQL parser checks the query for validity against the system catalogue and outputs an internal query representation, that is then passed to the optimiser. The latter uses the following table statistics: minimum and maximum values, and distinct value cardinalities for all fields of all tables. The first step in optimising a query is to identify the presence of join teams, as defined in Section 2.2. Join teams are kept as a single block of query execution, with the system generating code for each such block. Before code generation, however, the optimiser reorders the blocks of the query in the best way according to the system's statistics. As we shall see in Section 3.7, query plan selection is based on heuristics rather than dynamic programming, the main heuristic being the minimisation of the size of intermediate results.

The output of the optimiser is a topologically sorted list O of operator descriptors o_i. Each o_i has as input either primary table(s), or the output of $o_j, j < i$. The descriptor contains the algorithm to be used in the implementation of each operator and additional information for initialising the code template of this algorithm. Effectively, this list describes a scheduled tree of physical operators since there is only one root operator. It is organised so that the first elements describe the joins (binary operators or join teams) of the query, followed by any aggregation and sorting operations (unary operators, at most one descriptor for each). The optimiser keeps track of interesting orders and join teams by grouping together join operations in a single descriptor where possible to avoid intermediate result materialisation and processing.

The code generator will then traverse the topologically sorted list and emit a set of functions containing the source code for each operator. This is done in two steps per operator:

1. *Data staging*: all input tables are scanned, all selection predicates are applied, and any unnecessary fields are dropped from the input to reduce tuple size and increase cache locality on subsequent processing. Any pre-processing needed by the following operator, *e.g.*, sorting or partitioning, is performed by interleaving the pre-processing code with the scanning code. The output is then materialised (though not on disk, if the staged input is small enough to fit in main memory).

2. *Holistic algorithm instantiation*: the source code that implements each operator is generated. This code is an instantiation of the appropriate *holistic* algorithm template, as described in Section 3.5.

By looking at the inputs of each operator the code generator composes the operator implementations to generate a final function. This function evaluates the entire query and is to be called by the query engine. The final step of the code generation process is to insert all generated functions into a new C source file.

Once the query-specific source code has been generated, a system call invokes the compiler to compile the source file into a shared library file. This step allows the application of aggressive compiler optimisations that target the code of the specific query. The shared library file is then dynamically linked and loaded by the query executor. The latter calls the dynamically loaded function to evaluate the query and redirects the output to the client.

To make the process above more concrete, we describe what happens when the user submits the following query:

```
select  R.a, R.b, S.c, S.d
from    R, S
where   R.a = S.c
```

Initially, the parser traverses the query and builds an internal representation of the it, containing the scan operators for R and S, the following join operator and the final projection. The optimizer then decides on the join operator and outputs a list containing the sole join descriptor. This descriptor contains the access pattern for each input table, the data types and offsets of the fields `R.a`, `R.b`, `S.c` and `S.d` (projections are pushed down), the schemata of the temporary tables holding the staged versions of R and S, the data types and offsets of join attributes `R.a` and `S.c` inside the temporary tables, the output schema and the mapping of the fields `R.a`, `R.b`, `S.c` and `S.d` from the temporary tables to the ones of the output schema. Then, the code generator traverses the output of the optimizer and processes the join descriptor in two steps: (*a*) it builds two C source functions that access R and S accordingly to fill the staged temporary tables, and (*b*) it builds a C source function for implementing the selected join algorithm and generating the final result. It finally builds a C source function containing the instatiation of the temporary ables, the call of the staging functions followed by the joining one, the redirection of the final result to the client and the release of all allocated resources, including the temporary tables.

3.4 Code generation

In this section we present the implementation of the code generator. The code generator uses a template-based approach. Each algorithm is represented as an abstract template, which is instantiated according to the execution plan.

The code generator accepts as input the output of the optimiser (*i.e.*, a topologically sorted list O of operator descriptors) and produces a C source file of query-specific code. The generation algorithm is shown in Algorithm 1. As mentioned, each descriptor contains the algorithm to be implemented, along with the necessary parameters for

Algorithm 1: The code generation algorithm

```
Input: Topologically sorted list of operators $O$, allocated memory budget $M$.
Output: Code templates for data staging ($TS$), join evaluation ($TJ$) and
        aggregation ($TA$).
1  foreach join operator $j_m \in O$ do
2      retrieve code template $ts_m \in TS$ to stage $j_m$'s inputs ;
3      foreach input $i_n$ of $j_m$ do
4          instantiate $ts_m$ for $i_n$ ;
5          generate C function $cs_{mn}$ for staging $i_n$ ;
6          retrieve code template $tj_m \in TJ$ for $j_m$'s algorithm ;
7          instantiate $tj_m$ for $j_m$ ;
8          generate C function $cj_m$ to evaluate join ;
9  if $\exists$ aggregate operator $a \in O$ then
10     retrieve code template $ts_a \in TS$ to stage $a$'s input ;
11     instantiate $ts_a$ for $a$ ;
12     generate C function $cs_a$ for staging $a$ ;
13     retrieve code template $ta \in TA$ for $a$'s algorithm ;
14     instantiate $ta$ for $a$ ;
15     generate C function $ca$ to compute aggregate values ;
16 if $\exists$ ordering operator $s \in O$ then
17     retrieve code template $ts \in TS$ for sorting ;
18     instantiate $ts$ and generate sorting C function $cs$ ;
19 traverse $O$ to compose the main function $cm$ calling all functions ;
20 write all generated functions to a new source file $F$ ;
21 return $F$;
```

instantiating code templates. These parameters include the predicate data type(s), information about the operator's inputs, be they primary tables or intermediate results, and the output schema.

Code generation progresses as follows: the generator traverses the operator descriptor list processing the join operators first (Lines 1 to 8 in Algorithm 1) and moving on to any aggregation (Lines 9 to 15) and ordering operators (Lines 16 to 18). For each operator the generator emits functions that (*a*) stage the input (one function per input table), and (*b*) execute the operator's algorithm. These functions are built by retrieving the appropriate code template (*e.g.*, Lines 2 and 6 for joins) and instantiating it according to the parameters of the operator's descriptor (*e.g.*, Lines 4 and 7). Given that operator descriptors in O contain information about how operators are connected, the last bit of code generation is to traverse O and generate a main (composing) function that calls all evaluation functions in the correct order, ensures the correct (de)allocation of resources and sends the output to the client (Line 19). Finally, all generated functions are put into a new C source file, in the same order as they have been generated.

3.5 Holistic query evaluation algorithms

In this section we present the main algorithms of holistic query evaluation, with the goal of adapting well-known algorithms to main memory execution. The holistic framework builds on two premises:

1. NSM is employed at the storage layer. Though NSM has been shown to have sub-optimal performance in main-memory evaluation, most commercial and research database systems use it. We therefore use NSM in the interest of keeping the same storage layer and not affecting orthogonal system aspects.

2. Code is generated on a per-query basis. This technique exhibits opportunities for fine-grained and query-specific optimisations.

The above are more design decisions than assumptions. All proposed algorithms can be used over DSM- or PAX-based systems, while they can be implemented using different evaluation models, even iterators, as we show in Section 3.8.3. We believe, however, that code generation offers an elegant and compact platform for adapting database code to in-memory query evaluation, permitting query-specific code optimisations. Our algorithms perform the following optimisations: (*a*) since code is generated on a per-query basis, attribute types are known *a priori*, which means we can revert separate function calls for data accessing and predicate checking to pointer casts and primitive data comparisons respectively; and (*b*) fixed-length tuples inside each page can be accessed in an array-like mode through pointer arithmetic and direct referencing.

As an example of performing the above optimisations, Listings 3.1 and 3.2 show how the C code for a simple table scan-select operator can be generated in a more hardware-friendly way. By employing type information (`int` in this case) and using

Listing 3.1: Generic table scan-select

```
// loop over pages
for (int p = start_page; p <= end_page; p++) {
 page_struct *page = read_page(p, table);
 // loop over tuples
 for (int t = 1; t <= page->num_tuples; t++) {
  tuple_struct *tuple = read_tuple(t, page);
  if (!(matches(tuple, predicate_value, predicate_offset)) continue;
  add_to_result(tuple);
}}
```

Listing 3.2: Type-specific table scan-select

```
1  // loop over pages
2  for (int p = start_page; p <= end_page; p++) {
3   page_struct *page = read_page(p, table);
4   // loop over tuples
5   for (int t = 0; t < page->num_tuples; t++) {
6    void *tuple = page->data + t * tuple_size;
7    int *value = tuple + predicate_offset;
8    if (*value != predicate_value) continue;
9    memcpy(..);
10 }}
```

array accesses, we can eliminate all function calls (but the unavoidable for loading pages and generating the output) from the loop over the tuples of each page, saving a large number of CPU cycles. We also reduce the number of instructions executed, as we evaluate predicates over primitive data types. Moreover, the use of array computations allows the code to exploit the processor's superscalar design. The lack of function calls in the inner loop, in combination with directly accessing tuples and their fields by reference, further aids the compiler in optimising the generated code in ways that efficiently distribute data to registers and favour cache reuse.

All holistic algorithms build upon the code template of Listing 3.2 and extend it to include more tables (*e.g.*, in join algorithms), perform any necessary predicate(s) evaluation (Line 8) and manipulate the retrieved tuples as needed (Line 9). For completeness, we shall provide samples of the C code emitted by the code generator. This will highlight the efficiency of the generated code, as well as the common code patterns across different algorithms.

Throughout the subsequent analysis, one must keep in mind the difference in latencies for accessing each level of the memory hierarchy. Recall from Figure 2.2 that switching from sequential to random access may even double the latency on accesses outside the D1-cache; moving one layer down the memory hierarchy increases latency by one order of magnitude. The proposed algorithms therefore (*a*) examine the input in blocks that fit inside the D1- or the L2-cache, (*b*) maximise reuse by performing multiple operations over cache-resident data, and (*c*) strive for random access patterns appearing only inside the D1-cache, as this is the only level where the fetch cost is the same for both sequential and random accesses.

3.5.1 Data staging

Excluding the trivial case of all data needed to execute a query fitting the D1-cache, data caches can hold a very small portion of the input tables. As mentioned in Section 3.3, before executing any complex query operation, we stage the input by evaluating local predicates, dropping unnecessary fields, and reformatting it (using sorting and/or partitioning), according to the execution plan. In this section we shall describe the data staging primitives of our system and show how these have been implemented in a hardware-friendly fashion.

3.5.1.1 Sorting

Sorting builds on the standard sort-merge algorithm. The table to be sorted is partitioned into L2-cache-fitting partitions. Each partition is sorted in memory; all partitions are then merged in a single or multiple merging phases (depending on the input size and the amount of available memory) to generate the result. We have chosen to use non-recursive quicksort as the internal sorting algorithm due to the algorithm's cache locality merits [58]. Each page is treated as an array of tuples (as in Listing 3.2), while the code generator inlines all comparisons and performs the appropriate type-specific pointer casts. Given this implementation, internal sorting has a sequential access pattern (therefore having better cache locality and aiding the hardware prefetcher in its predictions) and does not make any function calls, enabling further compile-time code optimisations.

The sorted partitions are then merged in a standard multi-level merge-tree. Note here that although tuples are randomly accessed across different partitions during merging, the access pattern *within* each partition is sequential. This means that one can raise the same arguments regarding spatial locality and hardware prefetching as before, though the impact of prefetching decreases as the number of partitions and merging stages increase. Finally, the code generator injects type information and eliminates

R.a

value	*id*
x	0
y	1
z	2

(a) A single mapping table

R.a

value	*id*
x	0
y	1
z	2

R.b

value	*id*
A	0
C	1
B	2

R.c

value	*id*
10	0
30	1
40	2
20	3

(b) Multiple mapping tables

$$\begin{aligned}\text{partition-ID}(R.a = y, R.b = \text{B}, R.c = 20) &= R.a[y] \cdot |R.b| \cdot |R.c| + R.b[\text{B}] \cdot |R.c| + R.c[20] \\ &= 1 * 3 * 4 + 2 * 4 + 3 = 23\end{aligned}$$

(c) Partitioning across three attributes

Figure 3.2: Fine-grained partitioning

all function calls.

3.5.1.2 Partitioning

The system employs two types of partitioning, termed *fine-grained* and *coarse-grained* partitioning. The choice of partitioning depends on the number of distinct values of the partitioning attribute, with a small number of values favouring fine-grained partitioning and a large number of values favouring coarse-grained partitioning. We shall examine each in turn.

Fine-grained partitioning. This is applicable if the number of distinct values of the partitioning attribute is small enough so that a directory mapping values to partitions can be kept inside the cache hierarchy, as shown in Figure 3.2(a). This implies that no hash function is used, but rather that attribute values are mapped to partitions as the input table is scanned. For each attribute value read either (*a*) the value does not exist in the map, so a new entry is inserted, or (*b*) the value already exists in the map, so the identifier for the corresponding partition is retrieved. The tuple is then inserted into that partition. The directory is implemented as a sorted array of attribute values, with lookups using binary search. Hash-based solutions are also possible, though imbalancies in bucket sizes complicate their implementation and may lead to random memory accesses outside the D1-cache.

This technique can be adjusted to work when partitioning across more than one attributes, each with a small number of distinct values. To do so, we create a mapping table per partitioning attribute (see also Figure 3.2(b)) following the same principles as before and use a formula to identify the partition each combination of values belongs to. Assuming that M_i is the map for attribute i and $M_i[v]$ gives the identifier for value v of attribute i, one can then reduce the multi-dimensional mapping of tuple $(v_1, v_2, \ldots, v_n)$, for partitioning across n attributes, to the scalar $\sum_{i=1}^{n} (M_i[v_i] \prod_{j=i+1}^{n} |M_j|)$, where $|M_i|$ is the size of the mapping table for attribute i. The previous formula maps each combination of values to a unique partition offset, for a total number of $(\prod_{i=1}^{n} |M_i|)$ partitions. An example of applying the formula is shown in Figure 3.2(c). The constraint when using multiple attributes to partition the input is that the mapping tables for all attributes fit inside the cache hierarchy, else insertions and lookups will trigger expensive cache misses in random patterns that cannot be prevented by hardware prefetching.

Fine-grained partitioning is useful when joining on an attribute with a small number of distinct values for one of the joined tables. In the latter case, there is no partition probing, but rather the entire partition is scanned to generate matches (see Section 3.5.2.3). In addition, it is useful for aggregations resulting in a small number of groups (see Section 3.5.3).

Coarse-grained partitioning. The goal is to ensure that each partition fits in the L2-cache. To do so, we generate as many partitions as necessary and for each tuple read we apply multiplicative hashing [49]. Applying the hash function is inlined by the code generator to avoid a function call. Note that with coarse-grained partitioning there is no mapping table built, thus the partitioning fan-out might be wider than for fine-grained partitioning. If there are more partitions than the cache-line capacity of the D1-cache (*e.g.*, for a D1-cache size of 32KB and a cache-line size of 64B, if the fan-out is more than 512), the spatial locality of partitions will gradually degrade, leading to cache misses for each partition entry. Note that multiple attribute values will be hashed to the same partition (hash collisions), so subsequent processing is still necessary if matches are to be retrieved (*e.g.*, in the case of a hash join); this is not the case for fine-grained partitioning.

In both partitioning techniques we do not store tuple references in each partition, but rather the actual tuples. This approach reduces the number of tuples that fit within a partition, while copying whole tuples is certainly more expensive than storing references of 4-8 bytes. Still, the use of tuples inside partitions ensures that we do not pay the penalty of having another level of indirection. The latter would result in random access patterns touching data outside the L2-cache and subsequent cache misses. The query optimiser decides whether fine-grained or coarse-grained partitioning should be used, depending on the input statistics, as we shall show in Section 3.7.

3.5.1.3 Hybrid hash-sort

We have also implemented an alternative way of storing the tuples within each partition, which is useful as a staging method in certain join evaluation and aggregation scenarios. The idea is to keep the tuples of a partition sorted by applying the hardware-friendly implementation of quicksort. The efficiency of the latter is assured in case the partitioning fan-out is wide enough to allow the biggest partition to fit in the L2-cache. Though the sorting step seems redundant, it greatly improves the efficiency of the joining and grouping phases, as we shall see in Sections 3.5.2.3 and 3.5.3, by exhibiting sequential access patterns when processing the partitions.

3.5.1.4 Index scanning

Our system employs fractal B$^+$-trees for indexing. If there is a B$^+$-tree index on the attribute of interest and it is deemed the best access method for a table, the code generator interleaves index scanning with tuple retrieval from the primary table in a single code segment. In such a case the only function calls are the unavoidable ones to the buffer manager for retrieving pages from the primary table. Moreover, we can inject any other staging operations in the same block of generated code. For instance, the retrieved tuple can be hashed and inserted in a partition, or extra predicates may be evaluated. As usual, the code generator will perform all appropriate pointer casts to emit type-specific code.

3.5.2 Join evaluation

Join evaluation is based on nested loops, *i.e.*, it progresses in nested iterations over two tables (or more, as we shall see in Section 3.5.2.5). At first glance, join evaluation using variants of nested loops seems sub-optimal in terms of performance. As we shall see, however, it better utilises the resources of the CPU and improves response times.

The general form of nested-loops join is shown in Listing 3.3 for a query joining tables R and S.[1] We first iterate over the tables and load a page from each table; we then iterate over the tuples of each page, injecting the appropriate join predicate tests

[1]We do not present any data staging operations to avoid cluttering the code.

Listing 3.3: Naïve nested loops join

```
// loop over pages
for (int p_R = start_page_R; p_R <= end_page_R; p_R++) {
 page_struct *page_R = read_page(p_R, R);
 for (int p_S = start_page_S; p_S <= end_page_S; p_S++) {
  page_struct *page_S = read_page(p_S, S);

  // loop over tuples
  for (int t_R = 1; t_R <= page_R->num_tuples; t_R++) {
   tuple_struct *tuple_R = read_tuple(t_R, page_R) ;
   for (int t_S = 1; t_S <= page_S->num_tuples; t_S++) {
    tuple_struct *tuple_S = read_tuple(t_S, page_S);
    if (!(matches(tuple_R, offset_R, tuple_S, offset_S))) continue;
    add_to_result(tuple_R, tuple_S);
}}}}
```

Listing 3.4: Holistic nested loops join

```
// loop over pages
for (int p_R = start_page_R; p_R <= end_page_R; p_R++) {
 page_struct *page_R = read_page(p_R, R);
 for (int p_S = start_page_S; p_S <= end_page_S; p_S++) {
  page_struct *page_S = read_page(p_S, S);

  // loop over tuples
  for (int t_R = 0; t_R < page_R->num_tuples; t_R++) {
   void *tuple_R = page_R->data + t_R * tuple_size_R;
    for (int t_S = 0; t_S < page_S->num_tuples; t_S++) {
    void *tuple_S = page_S->data + t_S * tuple_size_S;
    int *attr_R = tuple_R + offset_R;
    int *attr_S = tuple_S + offset_S;
    if (*attr_R != *attr_S) continue;
    add_to_result(tuple_R, tuple_S);    /* inlined */
}}}}
```

(abstracted by the `matches()` function call in the code, where `offset_R` and `offset_S` are the offsets of the join predicate attributes within tuples from `R` and `S` respectively) and proceeding to the next loop only if a match is obtained. We now move on to presenting how this principle can be applied in conjunction with other join algorithms.

3.5.2.1 Holistic nested loops join

Naïve nested loops can be greatly optimised if the code is generated in a hardware-friendly way. This is shown in Listing 3.4, where, for simplicity, we assume the join attributes are integers: (*a*) since the code is generated per query, the field types are known *a priori*, which means we can revert separate function calls to pointer casts and primitive type comparisons (Lines 12 to 14). (*b*) Tuples of fixed length (denoted in the code by the values of `tuple_size_R` and `tuple_size_S` in Lines 9 and 11) can be accessed in an array mode through direct referencing. Both improvements eliminate all function calls (apart from the unavoidable calls to store the output) from the two inner loops, saving CPU cycles. Moreover, the array-like access pattern favours the utilisation of the hardware prefetcher on the first iteration over each page's tuples. Subsequent tuple iterations will be performed over D1-cache-resident pages, thus without any cache misses. This fits modern multicore processor designs, as it allows for D1-cache-resident processing with the D1-cache being separate for each core. At the same time it reduces resource contention for the L2-cache. All of the above are at the runtime; since the generated code will be compiled, loop nesting, lack of function calls, and array computations, allow the compiler to apply code optimisations targeting the specific join.

3.5.2.2 Holistic merge join

The nested loops block can be adjusted for merging sorted inputs, as shown in Listing 3.5. In the code, we continuously update the bounds of the loops (both in terms of starting and ending pages per table, and in terms of starting and ending tuples per page) as the merging process progresses. This is controlled by the condition variable `match_found`, which can take one of three values: a value of `no_match` means that there is no match between the current tuples; a value of `first_match` means that at least one match has been found and we should continue scanning inner tuples for matches; and a value of `in_group` means that the group of inner matching tuples has been exhausted, so we need to advance the outer tuple and backtrack to the beginning of the group of matching inner tuples. If each tuple of the outer loop matches at most once with tuples from the inner loop, we have a linear access pattern for both inputs, while backtracking to the beginning of a group of matching inner tuples is quite likely to result in cache hits (as small groups will tend to be resident in the L2-, or even the D1-cache).

In case there exist indexes on the join attribute for both input tables, we can use an algorithm that combines *index nested-loops* join with the *multi-predicate merge* join of [76]. We term this *cooperative staging* and its sketch is presented in Figure 3.3. The two indexes are scanned in a "zig-zag" mode, by testing the join predicate on their keys. For example, if the current key value of the outer index is 10, the inner index is linearly scanned until a value equal to, or greater than 10 is found. Then, the latter inner value

Listing 3.5: Holistic merge join

```
/* Code for staging R and S by sorting them on the join attributes */

int match_found = no_match;

for (int p_R = start_page_R; p_R <= end_page_R; p_R++) {
 page_struct *page_R = read_page(p_R, R);
 for (int p_S = start_page_S; p_S <= end_page_S; p_S++) {
  page_struct *page_S = read_page(p_S, S);

   for (int t_R = 0; t_R < page_R->num_tuples; t_R++) {
    void *tuple_R = page_R->data + t_R * tuple_size_R;
    for (int t_S = 0; t_S < page_S->num_tuples; t_S++) {
     void *tuple_S = page_S->data + t_S * tuple_size_S;

     int *attr_R = tuple_R + offset_R;
     int *attr_S = tuple_S + offset_S;
     if (*attr_R < *attr_S) {
      if (match_found == first_match) {
       /* set end page & tuple bounds for S to current offsets */
       match_found = in_group;
      }
      break;
     }
     if (*attr_R > *attr_S) {
      if (match_found == in_group) {
        /* set current page & tuple offsets for S right after their end bounds */
        /* set end page & tuple bounds for S to table & page end respectively */
        match_found = no_match;
      }
      continue;
     }
     if (*attr_R == *attr_S) {
      if (match_found == no_match) {
        /* set start page & tuple bounds for S to current offsets */
        match_found = first_match;
      }
     }
     add_to_result(tuple_R, tuple_S);   /* inlined */
}}}}
```

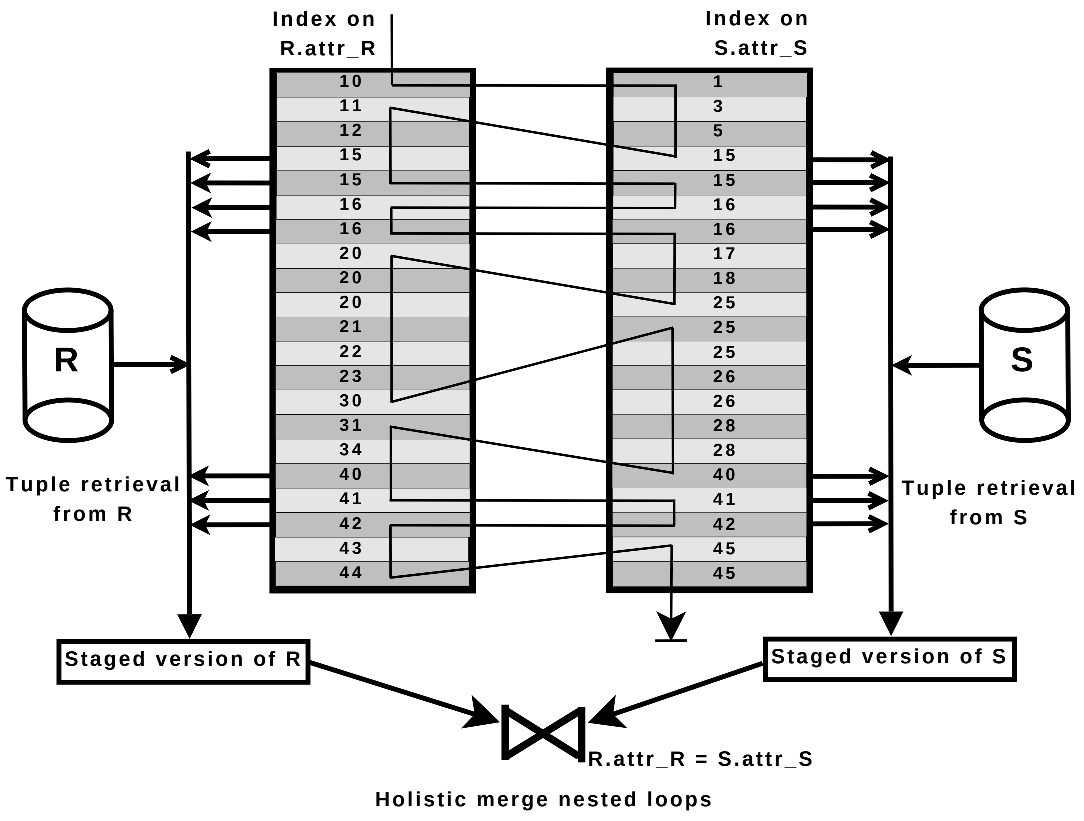


Figure 3.3: Cooperative staging

is used for scanning the outer index accordingly. For matching join attribute values, the tuples of each table are retrieved from the primary tables and any other selection predicates are applied. If all predicates evaluate to *true*, the tuples are stored in the staged versions of the corresponding tables. That way we store to temporary files only the input tuples that contribute to the join result, along with any other fields necessary for subsequent operations. The generated temporary files are then joined according to the holistic merge join algorithm.

Cooperative staging ensures that we have only matching tuples in the staged input. Even when there are multiple matches between the tuples of the inner and the outer tables, they will be retrieved once from the primary tables. Furthermore, instead of probing the index, the algorithm linearly scans the leaves of both indexes, thus exploiting spatial cache locality and hardware prefetching. Through code generation we can apply cooperative staging in a single code construct. The generated code integrates (a) interleaved scanning of both indexes, (b) testing of the join predicate (and any other selection predicates), and (c) storing of tuples in the corresponding temporary files.

Merge join bears a higher than normal branch misprediction overhead. Since loop bounds are updated according to the current value of the join predicate, successive predicate evaluations will yield widely differing results, except for the case where the join selectivity is either very small or very high. This is an inherent disadvantage of the merging phase. Modern processors, however, can adapt to the workload and minimise stalls through speculative execution, as we shall show in Section 3.8. Finally, by using the same ideas as before (*i.e.*, using a sequential array-like access pattern over tuples and eliminating function calls) hardware prefetching and compiler optimisations are applicable.

3.5.2.3 Holistic partitioned join

We have used Grace hash join [48] as the starting point for our partitioned join implementation. The input tables are partitioned using either fine-grained, or coarse-grained partitioning, as described in Section 3.5.1. In case fine-grained partitioning has been chosen, both inputs are partitioned using the same value-to-partition mapping. The next step is to join corresponding partitions using the optimised nested loops implementation of Listing 3.4, *i.e.*, no in-memory hash table is built for the outer table's partition. We chose to do so since we wanted to have sequential access patterns within each partition and avoid random ones (which would have been the case if a hash table was built). In the case of fine-grained partitioning this results in a cross-product of partitions. For coarse-grained partitioning, however, and since no hash table is built for the outer table's partitions, we effectively perform a "mini" nested loops join across corresponding partitions (as in Listing 3.4); this is sub-optimal and the reason why we have implemented the hybrid hash-sort partitioning strategy.

3.5.2.4 Holistic hybrid hash-sort-merge join

Recall from Section 3.5.1.3 that hybrid hash-sort, after applying coarse-grained partitioning, sorts each partition with a single quicksort call. When data is staged using this technique, we can apply merge-join across corresponding table partitions, by extending the *ImprovedSort* algorithm [69] to the *hybrid hash-sort-merge* join. The efficiency of the hybrid staging algorithm is combined with the increased cache locality of the merge join phase, as exhibited by the latter's linear access patterns. Note that if the size of the partitions is smaller than half that of the L2-cache, by sorting pairs of corresponding partitions just before joining them (instead of during data staging), we ensure that they are L2-cache-resident during join evaluation. The result is a generally applicable and globally optimised algorithm for join evaluation, as we shall show in Section 3.8.

3.5.2.5 Join teams

We have so far focused on binary joins to aid the understanding of the holistic evaluation model. The nested-loops template, however, allows for pipelined evaluation of multiple joins without materialising intermediate results, thus radically reducing memory operations and processor stalls. This is applicable in multi-way join queries with join teams, as per the definition given in Section 2.2. Notions such as hash teams and interesting orders, are translated to our model by increasing loop nesting. Listing 3.6 provides the generic code layout for a join team of m tables. For each input table the code generator emits one loop over its pages and one over its tuples, with the page loops preceding the tuple loops and following the same table order. The form of the code resembles the loop-blocking code optimisation technique, which is known to increase cache locality.

The presented algorithms will have to be adjusted for operating over more than two inputs. Depending on the join algorithm, different code paths will be generated. In case of holistic merge join, we sort all input tables (Line 1), omit the code block concerning partitions (Lines 2 to 5) and inject the code for holistic merge join with the necessary statements for updating the loop bounds of all tables (Lines 22 and 31 – see also Listing 3.5), to perform multi-way merging. The latter is controlled by a set of control variables `match_found_i`, `i` $\in [0, m-1]$, one per join predicate. For holistic hash join, we partition all input tables on the join attribute (Line 1) into `M` partitions, using the same mapping table (fine-grained partitioning) or hash-function (coarse-grained partitioning), and then examine corresponding partitions (Lines 2 to 4) using holistic nested loops. The loop bounds are not updated inside the page and tuple loops (*i.e.*, no code is generated for Lines 22 and 31). For the hybrid hash-sort-merge algorithm, we first partition all input tables as in hash join (Line 1) and then sort the corresponding partitions (Lines 2 to 5). The next step is to merge the sorted partitions using holistic merge join, as described before.

When we are not dealing with a join team, generalised hash teams [45] can be adapted to holistic evaluation when data has been staged using fine-grained partitioning. The code generator uses multiple maps over a set of fields for each table, as shown in Figure 3.2. The generated number of partitions is equal to the product of

Listing 3.6: Generic holistic template for join teams

```
/* Code to hash-partition or sort inputs */
hash: // examine corresponding partitions together
for (k = 0; k < M; k++) {
 /* update page bounds for all tables, for their k-th partition values */
 /* sort partitions - only in hybrid hash-sort-merge join */

 for (p_1 = start_page_1; p_1 <= end_page_1; p_1++) {
  page_struct *page_1 = read_page(p_1, partition_1[k]);
  for (p_2 = start_page_2; p_2 <= end_page_2; p_2++) {
   page_struct *page_2 = read_page(p_2, partition_2[k]);
   ...
   for (p_m = start_page_m; p_m <= end_page_m; p_m++) {
    page_struct *page_m = read_page(p_m, partition_m[k]);

    for (t_1 = 0; t_1 < page_1->num_tuples; t_1++) {
     void *tuple_1 = page_1->data + t_1 * tuple_size_1;
     for (t_2 = 0; t_2 < page_2->num_tuples; t_2++) {
      void *tuple_2 = page_2->data + t_2 * tuple_size_2;
      int *t1 = tuple_1 + offset_1;
      int *t2 = tuple_2 + offset_2;
      if (*t1 != *t2) {
        merge: // update bounds for all loops
        continue;
      }
      ...
      for (t_m = 0; t_m < page_m->num_tuples; t_m++) {
       void *tuple_m = page_m->data + t_m * tuple_size_m;
       t1 = tuple_k + offset_k;
       t2 = tuple_m + offset_m;
       if (*t1 != *t2) {
         merge: // update bounds for all loops
         continue;
       }
       add_to_result(tuple_1, ... , tuple_m);   /* inlined */
} ...}}}...}}}
```

distinct values over all fields used in join predicates. The produced disjoint partitions will contain tuples with specific values for these fields. These are then directly joined with holistic nested loops. That way, generalised hash teams can be implemented with only modest changes to the data staging process. The overhead is an increase in memory requirements for storing the attribute maps, one per join attribute, during staging. Hence, this approach is only applicable in the case where the number of partitions is kept small, *i.e.*, if the number of distinct values for all join attributes is limited. If this is true, generalised hash teams can be used to evaluate general join predicates with no intermediate results being materialised and, hence, evaluation proceeding in a fully pipelined fashion.

3.5.2.6 Observations and special provisions

The algorithms we presented favour the use of a small page size. This allows the D1-cache to simultaneously hold a sufficient number of pages when joining the tables of one group of nested loops. It is essential to perform tuple iterations over D1-cache resident pages, or else this process will result in numerous cache misses and degraded performance. In addition, when multiple pages from different tables co-exist in the D1-cache, our algorithms exploit cache locality even for wide join teams. In our system the page size is set to 4096 bytes, enabling eight pages to be simultaneously resident in a 32KB D1-cache, like the one we used for our experiments (see also Section 3.8.1.)

If the tuple size of the temporary input file is smaller than a cache line, the cache line fetch pattern will be sequential. Otherwise, the stride between successive cache line fetches will be greater that unary. Current hardware prefetchers can identify linear non-unary strided accesses and act accordingly. When dealing with variable-sized tuples we can replace the variable-sized fields with references to separate tables, if the fields are too large (*e.g.*, Character Large Objects), or allocate the maximum required space for this field on every tuple, thus maintaining the ability to access tuples in an array-like mode.

3.5.3 Aggregation algorithms

Multiple aggregate functions can be computed in a single block of code emitted by the code generator. As in the case of join evaluation, any technique used for staging the data affects the aggregation computation. Moreover, if the aggregation query does not contain any join predicates, aggregation and staging can be coupled and combined in the same code segment to generate highly efficient code (as we shall also see in the TPC-H experiments of Section 3.8.6). We shall provide C code samples for holistic aggregation using the following query:

select	`T.A`, `T.B`, *sum*(`T.C`), *avg*(`(1-T.C)*T.D`)
from	`T`
group by	`T.A`, `T.B`

This query is a variant of Query 1 of the TPC-H benchmark. It is also quite useful in showing that our code generator is capable of identifying computational dependencies as the ones between *sum* and *avg* in the query. In such cases the generator re-uses common parts of the computation by storing them in temporary variables (which the compiler will most likely exploit to increase register reuse).

3.5.3.1 Holistic sort aggregation

Sort-based aggregation implies that the input has already been sorted on the grouping attributes. The code generated for the example query, and for integer grouping attributes, is shown in Listing 3.7. The input is scanned linearly and the generated code has two control variables per grouping attribute to keep track of groups (variables `prev_i` and `curr_i`). Whenever a new group is encountered, the aggregate result is output and aggregate computation is reset (Lines 18 to 24). Note that for any other aggregate functions the code would have minimal differences: only the specific aggregate function computation would need to change (*e.g.*, Lines 5, 21, and 26 for computing *sum* in Listing 3.7).

As usual, we perform only sequential scans over tuples thus aiding the hardware prefetcher to lock on to the access pattern. Moreover, the lack of function calls is especially important in aggregation as it allows the compiler to generate executable code that widely reuses registers in a computationally-intensive operation, therefore reducing the number of data accesses per tuple. For example, in Listing 3.7, the value `*C` is reused; the compiler will most likely allocate it to a specific register through the entire computation. Therefore it will be re-used and a fetch from the D1-cache between

Listing 3.7: Holistic sort aggregation

```
/* Input has already been sorted on the grouping attributes */

int *prev_A, *curr_A; // control variables for T.A
int *prev_B, *curr_B; // control variables for T.B
double sum; // sum accumulator
double avg; // avg accumulator
int group_count; // tuple cardinality for current group

void *temp, *temp_tuple;
for (page_offset = start_page; page_offset <= end_page; page_offset++) {
 page_struct *page = read_page(page_offset, table);
 for (tuple_offset = 0; tuple_offset < page->tuple_counter; tuple_offset++) {
  temp = page->data + tuple_offset * tuple_size;
  curr_A = temp + offset_A; // T.A
  curr_B = temp + offset_B; // T.B
  int *C = temp + offset_C; // T.C
  int *D = temp + offset_D; // T.D
  if (*prev_A != *curr_A || *prev_B != *curr_B) {
   avg /= group_count;
   add_to_result(temp_tuple, sum, avg);
   sum = *C ;
   avg = (1 - *C) * (*D);
   group_count = 1;
  }
  else {
   sum += *C ;
   avg += (1 - *C) * (*D);
   group_count++;
  }
  prev_A = curr_A;
  prev_B = curr_B;
  temp_tuple = temp;
 }
}
avg /= group_count;
add_to_result(temp_tuple, sum, avg);   /* inlined */
```

consecutive computations will be avoided. This cannot be applied in the vectorised algorithms (*e.g.*, [6, 60]) that drop register reuse for the sake of array computations. In the case of the given query, vectorised aggregation would perform the *sum* and *avg* calculations on separate column-wise operations, that cannot share register content.

3.5.3.2 Holistic map aggregation

The first partition-based algorithm we present builds on fine-grained partitioning. If it is applicable, *i.e.*, if the total size of the mapping tables for all grouping attributes is small enough to fit in the cache hierarchy, aggregation can be computed in a single linear scan of the input without the need for any prior staging. If the amount of necessary memory exceeds the capacity of caches, this technique is not a good option as it will lead to cache thrashing.

With fine-grained partitioning aggregation can be computed through a generated block of code as the one shown in Listing 3.8. We assume that the grouping attributes `T.A` and `T.B` take twenty and ten distinct values respectively. A value-partition map (`map_i`) is built for each grouping attribute. Multiple arrays are allocated, one for each aggregate function and one for counting the elements in each group[2], as shown in Lines 3 to 5. The length of each extra array is equal to the number of expected partitions, *i.e.*, the number of groups we can expect. Assuming a grouping across n attributes with $|M_i|$ being the size of the mapping table for attribute i, each aggregate array needs to hold $(\prod_{i=1}^{n} |M_i|)$ values – see also Figure 3.2(c). Aggregate computation then proceeds in one linear scan of the input. For each tuple, the grouping attribute maps are used to identify the group (`group_id`) the tuple belongs to (Lines 10 to 16). Then, the accumulator variables for this group are updated with the current values of the aggregate functions (Lines 19 and 20).

[2]The latter is necessary only in specific operations, like *avg* in this case.

Listing 3.8: Holistic map aggregation

```
int map_A[20]; // map for T.A
int map_B[10]; // map for T.B
double sum[200]; // sum accumulator per group
double avg[200]; // avg accumulator per group
int group_count[200]; // tuple cardinality per group
for (int page_offset = start_page; page_offset <= end_page; page_offset++) {
 page_struct *page = read_page(page_offset, table);
 for (int tuple_offset = 0; tuple_offset < page->tuple_counter; tuple_offset++) {
  void *tuple = page->data + tuple_offset * tuple_size;
  int group_id = 0;
  int *A = tuple + offset_A; // T.A
  // lookup in map_A for key *A resulting in index k (inlined)
  group_id += k * 10;
  int *B = tuple + offset_B; // T.B
  // lookup in map_B for key *B resulting in index k (inlined)
  group_id += k;
  int *C = tuple + offset_C; // T.C
  int *D = tuple + offset_D; // T.D
  sum[group_id] += *C ;
  avg[group_id] += (1 - *C) * (*D);
  group_count[group_id]++;
 }
}
for (k = 0; k < 200; k++) {
 if (group_count[k] == 0) continue;
 avg[k] /= group_count[k];
 add_to_result(map_A[k / 10], map_B[k % 10], sum[k], avg[k]);   /* inlined */
}
```

3.5.3.3 Holistic hybrid hash-sort aggregation

Map aggregation based on fine-grained partitioning can compute the result in a single linear scan of its input, but its performance degrades as the number of partitions grows (*i.e.*, as the possible combinations of distinct values across the aggregation attributes increase). We therefore propose a hybrid approach that combines partitioning and sorting, which we term *holistic hybrid hash-sort* aggregation. The key idea is to use progressively wider partitioning to stage the input. We start from fine-grained partitioning on the first grouping attribute and progressively add grouping attributes until the size of the value-partition mapping tables exceeds cache capacity. We now have disjoint partitions across a subset of the aggregation attributes; we then sort each partition on the remaining grouping attributes and proceed as for sort-based partitioning. If the value-partition mapping table for the first grouping attribute does not fit the caches, then we revert to coarse-grained partitioning on this attribute and proceed as in sort-based aggregation, *i.e.*, by sorting each partition on all grouping attributes and scanning each input partition to produce the result, as presented in Listing 3.7.

Hybrid aggregation, though similar to simple sort aggregation, has the advantage of avoiding merging the sorted partitions, while it can reduce the cost of sorting each partition. The grouping attributes are split between hash-partitioning and sorting. For instance, in case of five grouping attributes, the first three might be used for fine-grained partitioning, with the generated partitions being then sorted only on the remaining two attributes. In addition, the algorithm does not suffer from the memory limitations of fine-grained partitioning, making it more widely applicable. The generated code allows for register reuse at compile-time and for hardware prefetching at run-time.

3.6 Implementation of the code generator

The main challenges in engineering a code generator for query evaluation were (*a*) the identification of common code templates across different algorithms, (*b*) the interconnection of different operators, since no common interface is present any more, and (*c*) the verification of correctness of the generated code for all supported operations.

The holistic evaluation model eases those problems. The main advantage is that its algorithms exploit generic code templates for all operations. Data staging employs the template of Listing 3.2; sorting and partitioning operations can be interleaved inside the code. For join evaluation, the nested-loops template of Listing 3.6 is used in each case, with differences between algorithms either being taken care of through staging, or through extra steps inside the loops, as described in Section 3.5.2.5. Aggregation extends the template of Listing 3.2 by injecting code for tracking different groups and computing the aggregate functions. This is evident by the similarity of the code templates of Listings 3.7 and 3.8 for sort and map aggregation respectively. Furthermore, operators are connected by materialising intermediate results as temporary tables inside the buffer pool and streaming them to subsequent operators.

The experience of developing HIQUE has verified these claims. The introduction of new algorithms or even new operators required more effort to extend the parser and the optimiser than to extend the generator. As a general methodology of introducing algorithms, we would first create a model implementation of the new algorithm and compare it to the existing templates. In most cases, the new algorithm resulted in a few different lines of code when compared to the existing evaluation algorithms. We would then extend the templates and the code generator to support the new algorithm. This process was further aided by the output of the code generator being a C source code file: the compiler helped the developer to easily identify errors in the generated code and reduce the number of *develop and test* iterations required until the new algorithm was fully supported.

3.7 Query optimisation

The key premises of the holistic model is to generate query-specific and hardware-friendly code and then take advantage of the efficiency of modern compilers and hardware setups to improve performance. This is not, however, the only way to boost query performance. There is always room for applying traditional optimisation primitives like algorithm selection and join ordering. In this section we shall describe our system's heuristics-based optimiser and how we have adapted query optimisation for CPU- and memory-bound evaluation.

While one can coarsely differentiate between I/O and memory accesses when optimising for I/O, the multiple cache levels of modern processors complicate the analysis of main memory execution. The situation is aggravated by data (re)placement on the various levels of the cache hierarchy being controlled by the CPU; this is in contrast to the buffer manager's absolute control of which pages will be transferred from and to the hard disks.

Execution plans have been traditionally organised as binary trees of operators, with the optimiser deciding the order of operators and the evaluation algorithms. The search space grows large quite soon, and the optimiser needs accurate cost models and statistics to make good choices. The holistic model helps in reducing the complexity of search space exploration by the use and inherent efficiency of join teams. The multi-input algorithms of Section 3.5.2 (*a*) execute faster on modern CPUs, and (*b*) remove the need for intermediate result materialisation and staging. We use join teams to reduce the size of the search space by identifying them and always preferring them over binary joins. Our approach is a combination of divide-and-conquer and rule-based optimisation: join teams are ordered and scheduled according to their estimated output size to form the entire execution plan. Each join team is then separately optimised by picking the appropriate algorithm. A sketch of the optimiseQuery algorithm is shown in Algorithm 2.

Join team ordering. The objective is to minimise intermediate result materialisation

Algorithm 2: The optimiseQuery algorithm

```
Input: Query $Q$.
Output: List $S$ of holistic templates $P$ for join teams $T$,
        Holistic template $P_\Gamma$ for aggregation.
1  $S = \emptyset$ ;
2  $T = \{t \mid t$ is a (generalised) join team in$Q\}$ ;
3  $\Gamma = \{\gamma \mid \gamma$ is a grouping attribute in $Q\}$ ;
4  $O = \{o \mid o$ is a sort attribute in $Q\}$ ;
5  while $(T \neq \emptyset)$ do
6      $t_i \leftarrow$ min(T): $\forall(t_j$ in $T) \rightarrow |t_i| \leq |t_j)|$ ;
7      $S = S \sqcup$ optimiseTeam $(t_i, O)$ ;
8      if $(\exists t_j \wedge (j > i) \wedge (t_j$ shares tableX with $t_i))$ then
9          substitute $X$ for $t_i$'s output in $t_j$ ;
10     $T \leftarrow T - t_i$ ;
11 $R$ = output of $t_{\text{length}(S)}$;
12 $P_\Gamma =$ optimiseAggregate $(R, \Gamma, O)$ ;
13 return $S$, $P_\Gamma$;
```

and, therefore, the penalty for staging the output of one join team when it becomes an input of a following one. For this purpose, the system maintains value statistics (see also Section 3.3) and uses them to estimate the output cardinality of each join team. Join teams are ordered in ascending result cardinality by iteratively picking the join team with the smallest estimated cardinality (Line 6 of Algorithm optimiseQuery, where $|t_i|$ denotes the estimated output cardinality of join team t_i). This is the order in which they will be executed in the final query plan (*i.e.*, the join team with the highest estimated output cardinality will be executed last). Each team is individually optimised through a call to Algorithm optimiseTeam (Line 7 of Algorithm optimiseQuery). The output template is appended to the output list S (Line 7, with $\sqcup$ denoting list append-

Algorithm 3: The optimiseTeam algorithm

```
Input: Join team $T$, sort attribute O.
Output: Holistic template $P_T$ for $T$.
1  $D_T = \{d \mid d$ is an input (either primary or temporary) table in $T\}$ ;
2  $F_T = \{f \mid f$ is a join attribute in $D_T$ $\}$ ;
3  sort $D_T$: $\forall(d_i, d_j) \wedge i \leq j \rightarrow |d_i| \leq |d_j|$ ;
4  if $(\sum_i |M(f_i \in F_T)| < \textit{map-threshold})$ then
5      $P_T =$ fine-grained partitioned join template ;
6  else
7      if $((\max_{d \in D_T}(|d|) < \textit{sort-threshold}) \vee$
8          $(\forall f_i \rightarrow \exists Index(f_i)) \vee ((\exists a) \wedge a \in F_T \wedge a \in O \wedge T = t_{length(S)}))$ then
9          $P_T =$ merge join template ;
10     else
11         $P_T =$ hybrid hash-sort-merge join template ;
12 return $P_T$;
```

ing), along with all the necessary parameters for instantiating the chosen holistic join template. This list will be used by the generator to build the code corresponding to each join team.

After individually optimising a join team, we update the inputs of the remaining join teams. If two join teams share a common table, its occurrence in the succeeding team (according to the global join team order) is replaced by the temporary table corresponding to the preceding team's output (Lines 8 and 9 of Algorithm optimiseQuery). That way, the succeeding join team will have the correct information about the temporary input table's schema and cardinality, so we can more accurately optimise it. The last plan in S, $t_{\text{length}(S)}$, will be the final join (team) operation, and the one on which all aggregations will be applied. We therefore pass it to Algorithm optimiseAggregates for generation of the holistic template for aggregation before we return (Lines 11, 12 and 13).

Single join team optimisation. We use the optimiseTeam algorithm of Algorithm 3 to internally optimise each join team. The key decisions are the order of the nested loops and the choice of holistic algorithm for the entire team. To tackle the first problem we conducted a preliminary set of experiments over various datasets. These indicated that table ordering has minimal impact on response time, which can be justified by the enhanced D1-cache locality of the holistic nested-loops template. An impact is evident only when one of the tables can entirely fit in some cache level. To address table ordering within a join team we decided to use the simple heuristic of ordering the tables of the team (be they primary or temporary) in ascending cardinality order, and ordering loops accordingly (*i.e.*, the largest input in cardinality is the one processed by the innermost loop – Line 3 of Algorithm optimiseTeam; larger tables are more likely to have more matching tuples for each predicate value, thus increasing the efficiency of nested loops.

Regarding the choice of join team evaluation algorithm we can either use analytical cost estimations or a calibration approach. The thresholds that specify the switching points between algorithms can be extracted by using detailed cost functions, in a fashion similar to the description of the generic cost model for hierarchical memory of [54]. We defer presenting our analytical cost model, along with its extension for multithreaded query execution, to Chapter 4 for the sake of exposition. Calibration [54] is an alternative method that leads to acceptable results. This technique works by maintaining hardware-specific statistics for each evaluation algorithm using various combinations of join team input counts, tuple sizes and input cardinalities. The crossing points between the performance plots of different algorithms correspond to switching points for the choice of algorithm. Though crossing points are only indicative of trends, they are useful thresholds for deciding on the most efficient algorithm for each operator. They are used in Lines 4-11 of Algorithm optimiseTeam, to specify the thresholds for switching from fine-grained partitioning to sort-based staging (*map-threshold*) and

Algorithm 4: The optimiseAggregate algorithm

```
Input: Input table $R$, group attributes $\Gamma$, sort attributes $O$.
Output: Holistic template $P_\Gamma$ for aggregation.
1 if ($\sum_i |M(\gamma_i \in \Gamma)| <$ map-threshold) then
2     $P_\Gamma =$ map aggregation template ;
3 else
4     if ($O = \Gamma$) then
5         $P_\Gamma =$ sort aggregation template ;
6     else
7         $P_\Gamma =$ hybrid hash-sort aggregation template ;
8 return $P_\Gamma$;
```

for switching from sort-merge to hybrid staging (*sort-threshold*).

If the calibration approach is used, the optimiser first examines if the cardinality of distinct values for the join attribute is small enough to allow the efficient processing of the value-partition map (or a sequence of maps, for generalised hash teams – Line 4). If this criterion is satisfied, the chosen algorithm is fine-grained partitioned join. Otherwise, the optimiser tests if merge join is appropriate. This applies if: (*a*) the biggest input table is small enough to allow all input tables to be sorted using a single quicksort call (Line 7), or (*b*) there exist indexes for all input tables on their join attribute (Line 8) and there is enough memory to use cooperative staging, or (*c*) the optimised join team is the last one and the output needs to be grouped or sorted on the join attribute(s) (Line 8 – in which case output staging is avoid). In all other cases, the chosen algorithm is hybrid hash-sort-merge join (Line 11).

Aggregation. The optimiseAggregate algorithm of Algorithm 4 generates the holistic plan for arbitrary aggregation. The first case to examine is whether the product of distinct values for all aggregation attributes is small enough to render fine-grained map aggregation applicable (Lines 1 and 2). This is preferable since it does not require any staging and can be computed in one scan of the input. If this criterion is not satisfied, the optimiser tests if the aggregation result needs to be sorted and if the sort attributes are the same and follow the same order as the grouping ones; in this case, holistic sort aggregation is the chosen algorithm (Lines 4 and 5). In all other cases the system employs holistic hybrid hash-sort aggregation (Line 7).

3.8 Experimental Study

To test the viability of code generation as a general solution to query evaluation we experimented with different aspects of the system. Our aim was to measure (*a*) the superiority of the holistic model over the traditional iterator-based approach, (*b*) the effect of compiler optimisations on the code generated by HIQUE, (*c*) the competitiveness of the system in comparison to other approaches, both research and commercial ones, on established benchmark queries, and (*d*) the penalty for generating, compiling, and linking query-specific code at runtime.

To measure the efficiency of the holistic model against iterators we have implemented iterator-based versions of the algorithms presented in Section 3.5 and compared them to the code generated by HIQUE. We have also benchmarked HIQUE against three database systems: (*a*) PostgreSQL (version 8.2.7), a widely-used and high-performance open-source DBMS over NSM that uses iterators, (*b*) a commercial system, which we refer to as *System X* for anonymity, also using NSM and iterators but employing software prefetching instructions to reduce cache miss stalls, and (*c*) MonetDB (version 5.8.2), an architecture-conscious DBMS using a DSM-based storage layer and column-wise evaluation algorithms. This choice allowed the comparison of different storage systems and query engine designs, with PostgreSQL representing the traditional I/O-optimised design, System X bridging the gap between I/O- and CPU-bound execution with software prefetching, and MonetDB being a design optimised for main-memory execution.

Number of cores	2
Frequency	1.86GHz
Cache line size	64B
I1-cache	32KB $\times$2
D1-cache	32KB $\times$2
L2-cache	2MB (shared)
L1-cache miss latency (sequential)	9 cycles
L1-cache miss latency (random)	14 cycles
L2-cache miss latency (sequential)	28 cycles
L2-cache miss latency (random)	77 cycles
Branch misprediction penalty	15 cycles
RAM type	2x1GB DDR2 667MHz

Table 3.1: Intel Core 2 Duo 6300 specifications

3.8.1 Testbed

We report results on the widely used x86-64 processor architecture. Our system had an Intel Core 2 Duo 6300 dual core processor, clocking at 1.86GHz, and a physical memory of 2GB. The operating system was Ubuntu 8.10 (64 bit version, kernel 2.6.27); HIQUE's generated code was compiled using the GNU `gcc` compiler (version 4.3.2) and with the `-O2` compilation flag. For completeness we executed the same experiments on a box with an AMD Athlon 4200 dual core processor at 2.2GHz, with the same amount of memory and the same operating system; both setups exhibited the same trends, so we shall focus on the Intel results. More detailed information about the testing platform can be found in Table 3.1. The cache latencies were measured using RightMark Memory Analyser [65], while the branch misprediction penalty of 15 cycles corresponds to the 14-stage-deep pipeline of the Core 2 architecture.

3.8.2 Metrics and methodology

We built indexes in all systems, set their memory parameters to allow in-memory execution, gathered statistics in the highest level of detail and disabled concurrency control where possible. All queries were run in isolation and were repeated ten times each. We report average response times for each query, with the deviation being less than 3% in all cases. Each query ran in its own thread, using a single processor core. We did not materialise the output in any case, as the penalty of materialisation is similar for all systems and configurations. We also used hardware performance events as metrics. We obtained the latter with the OProfile [59] tool, which collects sampling data from the CPU's performance event counters. The events captured included unhalted core cycles, retired instructions, branch instructions and mispredictions, resource stalls, instruction and data cache reads, misses and prefetches, and software prefetching instructions. More information about these events can be found in [43].

We have used these measurements, along with the sampling frequencies and the stall penalties of Table 3.1, to compute the percentage of execution time that was wasted on each stall type. We broke down execution time to instruction execution, D1-cache miss stalls, L2-cache miss stalls and other pipeline resource stalls.[3] To account for hardware prefetching, we assumed sequential access latencies for prefetched cache lines and random access latencies for all other cache misses. This allows for approximate calculation of the cost of cache misses, as the non-blocking design of cache memory allows the CPU to continue executing instructions while fetching data. Still, this methodology provides a good approximation of actual cache miss stall times. In addition to the execution time breakdown, we also calculate the *Cycles Per Instruction* (CPI) ratio, the minimum value being 0.25 for Intel Core 2 Duo processors (*i.e.*, four instructions executed in parallel per CPU cycle). We also measured samples for retired instructions, function calls and D1-cache accesses, normalised to the highest value

[3]Other pipeline resource stalls are defined as resource stalls that are not due to D1- or L2-cache misses – see also [43].

	Branch misprediction stall time (%)		**Branch prediction efficiency (%)**		**I1-cache miss stall time (%)**		**I1-cache miss rate (%)**	
	Min	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**
PostgreSQL	0.72	3.79	93.04	98.13	0.36	2.18	0.20	1.01
System X	0.19	1.73	95.95	99.28	0.12	5.28	0.08	2.14
MonetDB	0.05	2.48	91.04	98.23	0.07	4.58	0.03	2.11
HIQUE	0.00	2.74	90.04	100.00	0.03	0.44	0.01	0.17

Table 3.2: Speculation and instruction miss statistics

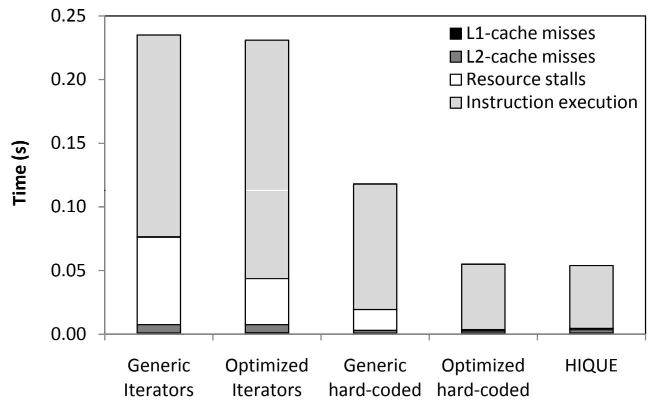
among the compared configurations for each query. Finally, we report the prefetching efficiency ratio, defined as the number of prefetched cache lines over the total number of missed cache lines. A value closer to one denotes better performance.

In Table 3.2 we show the impact of branch misprediction across all benchmark queries. We define the branch prediction *efficiency* as one minus the ratio of the number of branch mispredictions over the total number of branches. Thus, a value closer to one denotes optimal performance. For all systems, the percentage of execution time that corresponded to branch misprediction stalls was less than 3%, while the efficiency of the prediction unit was consistently greater that 90%. The table also indicates that instruction misses have a limited impact on response time. The instruction prefetching and caching mechanisms that all modern processors incorporate restrict the frequency of such stall events. Observe that HIQUE's I1-cache miss rate is less than 0.2% even for complex queries, indicating that the I1-cache can completely hold the instruction-set for each operation. Because of these results, we no further focus on the impact of branch mispredictions and instruction cache misses on the reported performance, but include them in the "Other resource stalls" measurements.

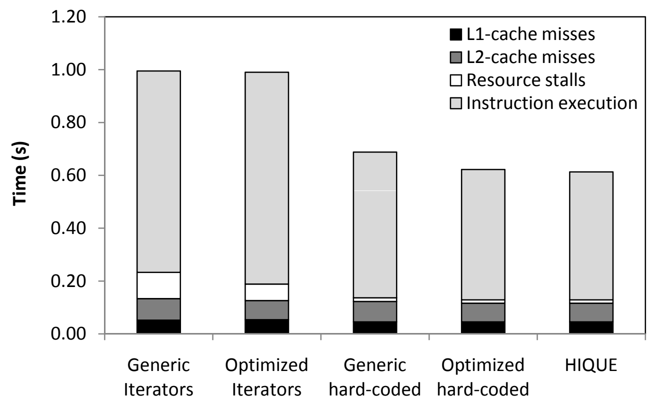
3.8.3 Iterators versus holistic code

To quantify the iterator model's deficiency compared to the proposed holistic model, we compared the following implementations: (*a*) an iterator-based one using generic functions for predicate evaluation, (*b*) a type-specific version of iterators with inlined predicate evaluation, (*c*) a hard-coded implementation using generic functions for predicate evaluation and tuple accesses, (*d*) an improved hard-coded version with direct tuple accesses using pointer arithmetic, and (*e*) the code generated by HIQUE, that further inlines predicate evaluation. We favoured the generic implementations by separately compiling the code for each query (including all parameters for instantiating the statically pipelined iterators), thus allowing their extended optimisation by the compiler. For join evaluation, we experimented with (*a*) two tables of 10,000 tuples of 72 bytes each using merge join, with each outer tuple matching with 1,000 inner tuples, and (*b*) two tables of 1,000,000 tuples of 72 bytes each using hybrid join, with each outer tuple matching with 10 inner tuples. For aggregation, we used a table of 1,000,000 tuples of 72 bytes each, two *sum* functions, and we selected as the grouping attribute one field with either (*a*) 100,000 distinct values, or (*b*) 10 distinct values. We employed hybrid aggregation in the first case and map aggregation in the second. All join and grouping attributes were integers. We used both response times and hardware performance events as metrics. We present the results for join evaluation in Figure 3.4 and for aggregation in Figure 3.5.

The first join query is inflationary, as it produces 10,000,000 tuples when joining two tables of 10,000 tuples each. In this case, the nested-loops-based template for join evaluation proves very efficient, as HIQUE is almost five times faster than the iterator implementations. The time breakdown in Figure 3.4(a) shows that all versions exhibit minimal memory stalls, so the difference in execution time is exclusively due to the lack of function calls, the reduction in retired instructions, and the elimination of resource stalls. Note that the generated code requires 26.22% of the instructions, 36.67% of the data accesses and 1.08% of the function calls when compared to the



(a) Execution time breakdown for Join Query #1



(b) Execution time breakdown for Join Query #2

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.613	100.00	100.00	100.00	8.33	43.28
Optimised iterators	0.628	91.81	66.99	94.20	10.64	68.35
Generic hard-coded	0.569	53.47	33.87	51.85	27.78	86.84
Optimised hard-coded	0.498	27.63	1.29	39.31	25.00	89.47
HIQUE	0.475	26.22	1.08	36.67	25.00	92.11

(c) Hardware performance metrics for Join Query #1

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.697	100.00	100.00	100.00	30.67	87.27
Optimised iterators	0.729	95.65	86.86	97.49	30.31	92.20
Generic hard-coded	0.720	67.32	49.56	61.95	60.55	86.38
Optimised hard-coded	0.750	56.80	32.75	56.13	60.95	89.93
HIQUE	0.769	56.62	32.37	54.03	61.07	89.97

(d) Hardware performance metrics for Join Query #2

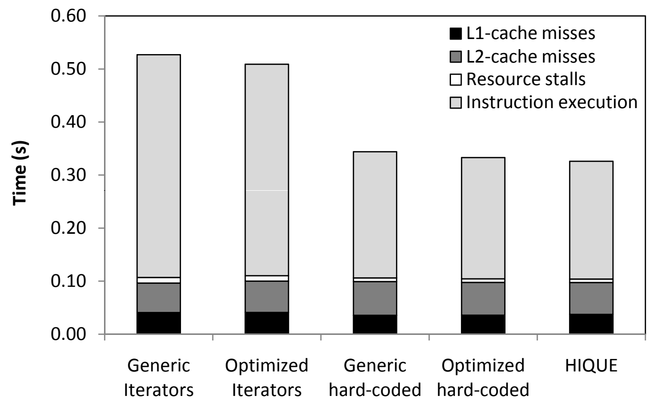
Figure 3.4: Join profiling

generic iterator version, as shown in Figure 3.4(c). Besides, the CPI ratio improves by 22.5% and closes in to the ideal value of 0.25. One can also observe that the efficiency of hardware prefetching more than doubles as the code becomes more query-specific, both for the D1- and the L2-cache.

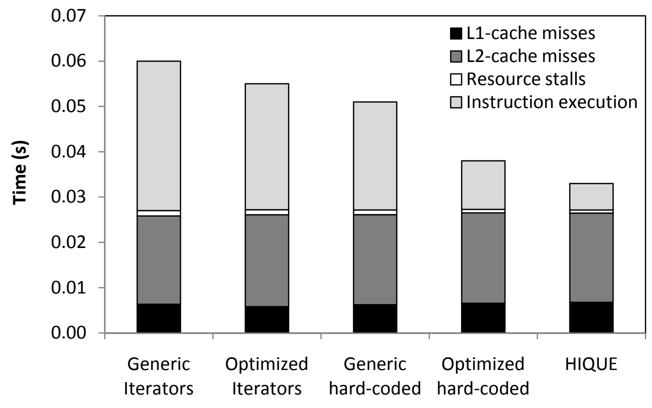
The second join query uses two larger tables as inputs and has much lower join selectivity. In this case, the majority of the execution time is spent on staging the input, *i.e.*, hash-partitioning it and sorting the partitions. Since all versions implement the same algorithm, use the same type-specific implementation of quicksort, and display similar access patterns, the differences in execution times are narrowed. As shown in Figure 3.4(b) HIQUE is almost twice faster than the iterator-based versions. The penalty for memory stalls is similar in all cases, as expected. The reduction in retired instructions, data accesses and function calls is still significant, according to Figure 3.4(d), but does not reach the levels of the previous query. Note that the CPI ratio *increases* for hard-coded versions. This is due to the retirement of fewer instructions in total, so the contribution of costly memory operations to the CPI is more substantial. Prefetching efficiency doubles for the D1-cache and is approximately 90% for the L2-cache in all cases.

In terms of aggregation, the first benchmark query was evaluated using the hybrid hash-sort algorithm. In this case staging dominates execution time, as aggregation is evaluated in a single scan of the sorted partitions. Still, as shown in Figure 3.5(a), HIQUE maintains an advantage of a factor of 1.61 over iterators. The use of the same partitioning and sorting implementations leads to similar memory stall costs for all code versions. The difference in execution times mainly stems from the reduction in instructions, data accesses and function calls, according to Figure 3.5(c). Observe that the efficiency of the D1-cache prefetcher increases three times, while that of the L2-cache reaches almost 90% for all implementations.

In the case of the proposed map-based algorithm, aggregation is evaluated in a single pass of the input without any need for intermediate staging. This allows the code



(a) Execution time breakdown for Aggregation Query #1



(b) Execution time breakdown for Aggregation Query #2

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.796	100.00	100.00	100.00	19.16	94.76
Optimised iterators	0.798	95.35	92.48	99.88	21.73	91.95
Generic hard-coded	0.872	59.85	86.83	91.19	56.79	85.59
Optimised hard-coded	0.875	54.99	77.74	89.32	56.82	86.12
HIQUE	0.919	53.86	68.65	81.63	56.90	88.95

(c) Hardware performance metrics for Aggregation Query #1

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.791	100.00	100.00	100.00	75.71	95.05
Optimised iterators	0.881	81.85	94.06	74.79	93.18	93.17
Generic hard-coded	0.936	67.62	65.35	60.21	78.93	93.44
Optimised hard-coded	0.904	53.13	32.67	52.72	78.37	95.57
HIQUE	0.899	41.89	4.95	46.13	70.39	95.86

(d) Hardware performance metrics for Aggregation Query #2

Figure 3.5: Aggregation profiling

	Join Query #1		**Join Query #2**		**Aggregation Query #1**		**Aggregation Query #2**	
	`-O0`	`-O2`	`-O0`	`-O2`	`-O0`	`-O2`	`-O0`	`-O2`
Generic iterators	0.802	0.235	1.953	0.995	1.225	0.527	0.136	0.060
Optimised iterators	0.618	0.231	1.850	0.990	1.199	0.509	0.113	0.055
Generic hard-coded	0.430	0.118	1.421	0.688	0.586	0.344	0.095	0.051
Optimised hard-coded	0.267	0.055	1.225	0.622	0.554	0.333	0.080	0.038
HIQUE	0.178	0.054	1.138	0.613	0.543	0.326	0.070	0.033

Table 3.3: Effect of compiler optimisation (response times in seconds)

generator to inline all group tracking and aggregate calculations in a single code segment. As shown in Figure 3.5(b), the code generated by HIQUE outperforms generic iterators by almost a factor of two. Memory stalls dominate execution time for the HIQUE version (though their effect might be alleviated from the operation of non-blocking caches), as the aggregate calculations require only a few instructions per tuple. Also shown in Figure 3.5(b), the reduction in function calls is gradual as the code becomes more query-specific and reaches 4.95% for the most optimised hard-coded version. Furthermore, the linear scan of the input helps the hardware prefetchers achieve high levels of efficiency, over 70% for the D1-cache and near 95% for the L2-cache in all cases.

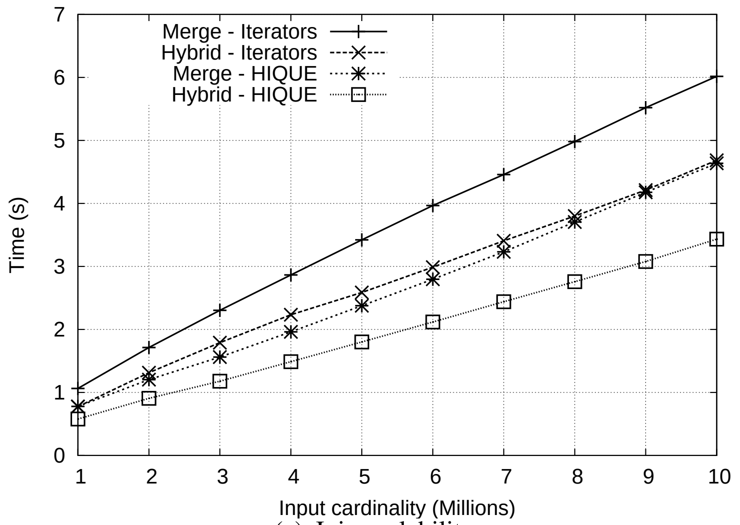
We next examined the efficiency of compiler optimisations on the iterator-based and the hard-coded implementations. We compiled the various implementations with compiler optimisations disabled (by setting the optimisation flag to `-O0` for the GNU compiler) and ran the same join and aggregation queries. The results are presented in Table 3.3. Naturally, the differences between the various code versions are more tangible when there are no compiler optimisations, since the compiler can apply some of the optimisations that are included in the code generation process. For example, the compiler may inline the functions for predicate evaluation, so the differences between the last two implementations are narrowed in all queries, but become apparent when the `-O0` optimisation flag is used.

The results show that compiler optimisations are most efficient in the first join query, resulting in speedups between 2.67 and 4.85, as the loop-oriented code transformations can improve performance on iterative tuple processing. For the rest of the queries the speedup is almost a factor of two. Since we compile the code for each query and for all implementations, the speedup is significant even for the iterator-based ones. Moreover, the compiler is less efficient on the hard-coded implementations: the source code is already minimalistic and contains various optimisations (*e.g.*, loop blocking, function inlining). Still, the simplicity of the code and the lack of function calls allows the compiler to further improve the hard-coded versions resulting in significant speedups.

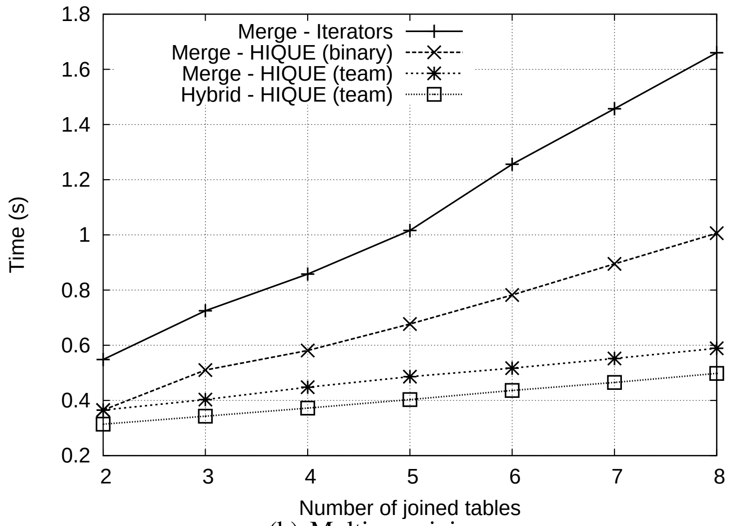
3.8.4 Performance of holistic algorithms

We now move on to examine the performance of the proposed algorithms while varying the characteristics of the input and the predicates to be applied. We compared the optimised iterator-based versions of the proposed algorithms with the code HIQUE generates for each query. In Figure 3.6(a) we examine scalability in join evaluation. We used two tables with a tuple size of 72 bytes. Each outer tuple matched with ten inner tuples on integer join attributes. The cardinality of the outer table was set to 1,000,000, while the inner one's varied between 1,000,000 and 10,000,000. The results show that all algorithms scale linearly, with iterator-based hash-sort-merge join having similar performance to HIQUE's merge join. As expected, the generated version of the hash-sort-merge join outperforms all other versions by a substantial margin, proving its efficiency in a wide range of input cardinalities.

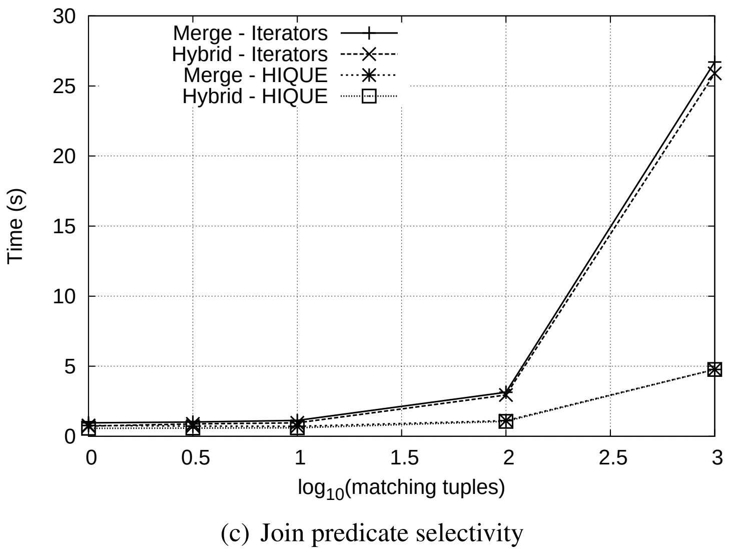
In multi-way queries, the evaluation of multiple joins using a single segment of deeply-nested loops improves performance as the generated code does not require ma-



(a) Join scalability



(b) Multi-way joins



(c) Join predicate selectivity

Figure 3.6: Join performance

terialisation of intermediate results. To verify this, we joined one table of 1,000,000 tuples with a varying number of tables of 100,000 tuples each, on a single join attribute. All tables had 72-bytes-sized tuples, while the output cardinality was 1,000,000 in all cases. We compared the binary iterator-based merge join, its equivalent when generated by HIQUE, and the code versions when join teams where enabled in HIQUE, using either merge or hybrid join. The results of Figure 3.6(b) show that although iterator-based merge join takes advantage of sorted orders, it is widely outperformed by its holistic equivalent. Furthermore, the adoption of join teams radically reduces execution time, with the difference between HIQUE and iterators reaching a factor of 3.32 when joining eight tables. The extension of the nested-loops join template to support join teams therefore pays off in the case of multi-way join queries.

Highly-selective join predicates are expected to increase the difference in performance between the iterator and the holistic model. This is due to the number of iterator calls growing larger and the join evaluation cost surpassing that of input staging; the latter cost is similar for all implementations. This is shown in Figure 3.6(c) for joining two tables of 1,000,000 tuples each. Each input tuple was 72 bytes wide, while the number of inner matching tuples per outer tuple varied between 1 and 1,000. The results show that the gap between the iterator-based and the holistic implementations widens quickly as join selectivity increases and reaches a factor of five for 1,000 matches per outer tuple.

The salient factor in aggregation performance is the domain of the grouping attribute(s). If this domain allows the value directories and the aggregate arrays (see also Section 3.5.3) to simultaneously fit in the lowest cache level, map aggregation is expected to outperform the algorithms that require input staging. We show the effect of the grouping attribute's range in Figure 3.7. The input table had 1,000,000 tuples of 72 bytes each. We used two *sum* functions and one grouping attribute as we varied the number of distinct values between 10 and 100,000. The results verify that, for small numbers of groups, map aggregation is highly efficient, both in its iterator-based and

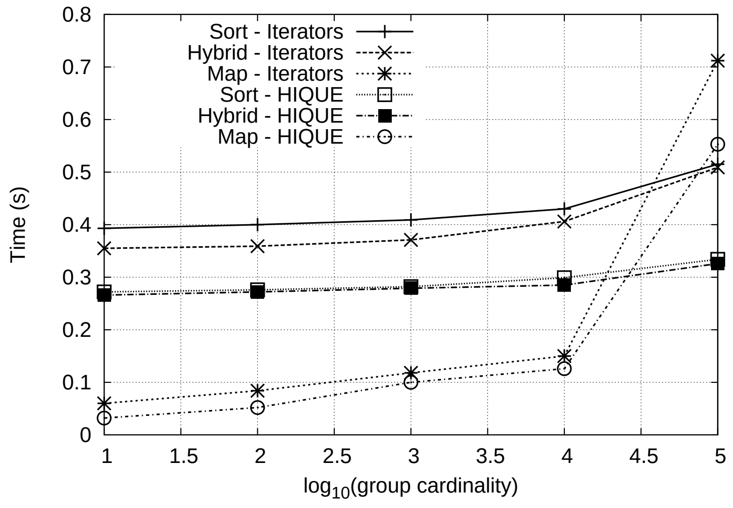


Figure 3.7: Aggregation performance

`unique1`	:	serial (primary key)
`unique2`	:	integer
`times4`	:	integer (each value appears 4 times)
`times10`	:	integer (each value appears 10 times)
`ten`	:	0...9 (ten distinct values)
`stringu`	:	52-character string

Table 3.4: Table schema for the Wisconsin benchmark

its holistic form. However, sort and hybrid aggregation are only moderately affected by the number of groups. They perform better than map aggregation when the auxiliary data structures of the latter (*i.e.*, the value directory for the grouping attribute and the aggregate arrays) span the L2-cache, the difference approaching a factor of two for 100,000 groups.

3.8.5 Wisconsin benchmark

In the next set of experiments we compared HIQUE with the other DBMSs over a variant of the Wisconsin benchmark [22]. All tables we experimented with conformed to the schema shown in Table 3.4. The `unique1` field is the primary key and takes serial values starting from 0. The `unique2` field has the same values as those of `unique1`, but in a random sequence. For fields `times4` and `times10`, each value appears exactly four or ten times respectively (*e.g.*, if the table has 1000 tuples, the used values are $[0, 250)$ for `times4`) and follows the distribution of `unique2`, *i.e.*, $\texttt{times4} = \texttt{unique2} \mod 250$. The `ten` field ranges between 0 and 9 randomly. Finally, the `stringu` field is a string of 52 characters; it was not used in any predicates, but was employed for padding the tuple and expanding the size of projection lists. The tuple size was 72 bytes. The chosen schema allowed us to easily compute the cardinality of each query result.

Based on the schema of Table 3.4, we imported four tables of $10,000$ tuples each (named `k10_1`, `k10_2`, `k10_3` and `k10_4`), three tables of $100,000$ tuples each (named `k100_1`, `k100_2` and `k100_3`) and three tables of $1,000,000$ tuples each (named `m1_1`, `m1_2` and `m1_3`). We experimented with a collection of join and aggregation queries shown in Figure 3.8. In all cases, HIQUE's optimiser chose the optimal holistic plan. For completeness we also report the performance of all applicable holistic algorithms. Note that whenever we refer to "HIQUE-index", we mean holistic merge join, combined with cooperative staging. For hardware performance metrics we only present measurements for the plan selected by the optimiser.

3.8.5.1 Join evaluation

For join queries we experimented with three variants: the first without result generation (**select** *count*(*)), the second with two fields projected from each participating table, and the third with all fields projected (**select** *). That way, we were able to check how both the input tuple sizes and the resulting tuple size affect execution times.

We experimented with joins over pairs of tables (Query 1, Figure 3.9) or as one join team (Query 2, Figure 3.10). In the first case all holistic algorithms exhibited similar performance, with the cooperatively staged merge join being slightly faster; this denotes the efficiency of combining index scanning with join evaluation in the same code

```
select    projection list
from      m1_1, m1_2, k100_1, k100_2
where     m1_1.unique2 = k100_1.times4
and       m1_2.unique2 = k100_2.times4
and       k100_1.times10 = k100_1.times10
```
(a) Query 1 (1,000,000)

```
select    projection list
from      m1_1, k10_1, k10_2, k10_3, k10_4
where     m1_1.unique2 = k10_1.times4
and       m1_1.unique2 = k10_2.times4
and       m1_1.unique2 = k10_3.times4
and       m1_1.unique2 = k10_4.times4
```
(b) Query 2 (640,000)

```
select    projection list
from      k10_1, 10k_2
where     k10_1.ten = k10_2.ten
```
(c) Query 3 (10,000,000)

```
select    projection list
from      m1_1, m1_2
where     m1_1.unique2 = m1_.unique2
order by  m1_1.unique2
```
(d) Query 4 (1,000,000)

```
select    projection list
from      t_1, t_2, t_3
where     t_1.unique2 = t_2.times10
and       t_1.unique2 = t_3.times4
          t ε {k10, k100, m1}
```
(e) Query 5 (4x|t|)

```
select    ten,
          avg(unique2),
          max(unique1)
from      m1_1
group by  ten
```
(f) Query 6 (10)

```
select    times10,
          avg(unique2),
          max(unique1)
from      m1_1
group by  times10
```
(g) Query 7 (100,000)

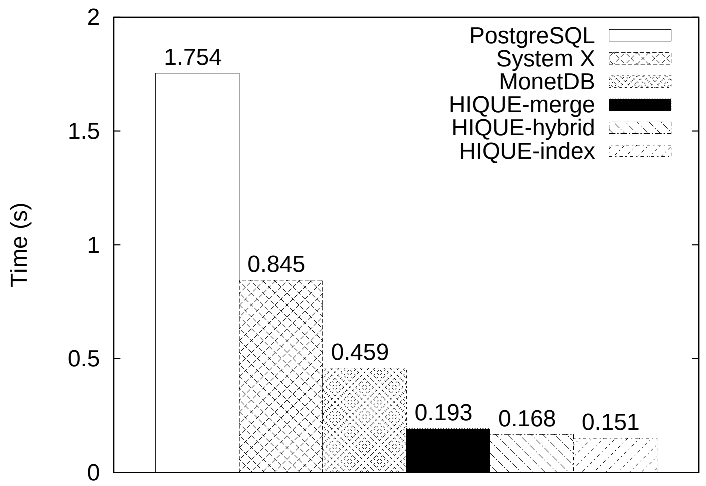
```
select    ten, times10,
          avg(unique2),
          max(unique1)
from      m1_1
group by  ten, times10
```
(h) Query 8 (100,000)

```
select    ten, times10, times4,
          avg(unique2),
          max(unique1)
from      m1_1
group by  ten, times10, times4
order by  ten, times10, times4
```
(i) Query 9 (100,000)

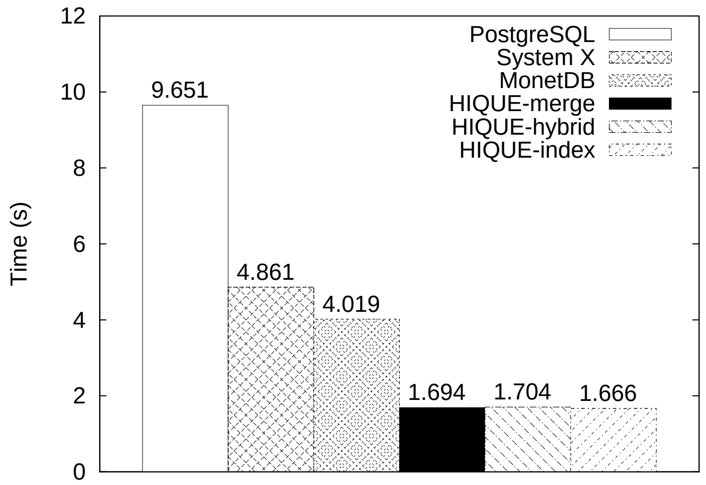
Figure 3.8: Queries for the Winsconsin benchmark (output cardinality in parentheses)

segment. In comparison to the other systems, HIQUE has an advantage ranging from a 2.5-fold performance increase over MonetDB to a 7.1-fold performance increase over PostgreSQL. The difference is greater when *count*(*) is used because of the increased cache locality of the data staging step, as well as the complete lack of function calls inside the nested loops over tuples (see also Listing 3.4). As the projection list grows, execution times suffer the penalty of output generation. Despite that, HIQUE maintains its advantage; it can handle large tuple sizes better by performing join evaluation inside D1-cache, while it integrates field types and offsets in the functions used for building the result.

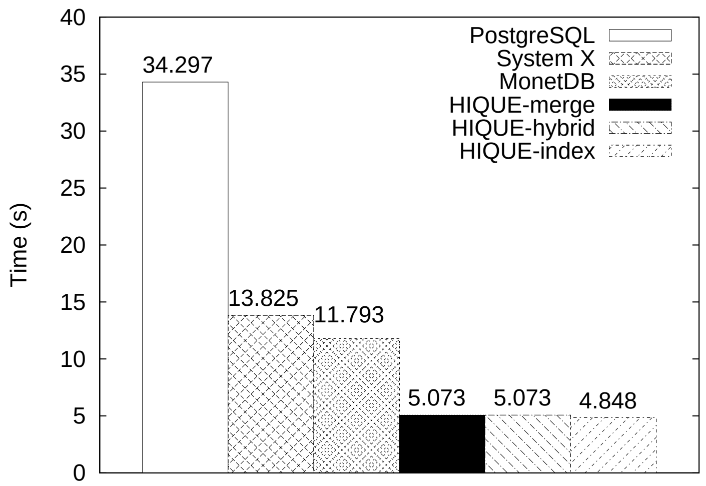
If holistic join teams are applicable, as in Query 2, the performance advantage of



(a) count(*)

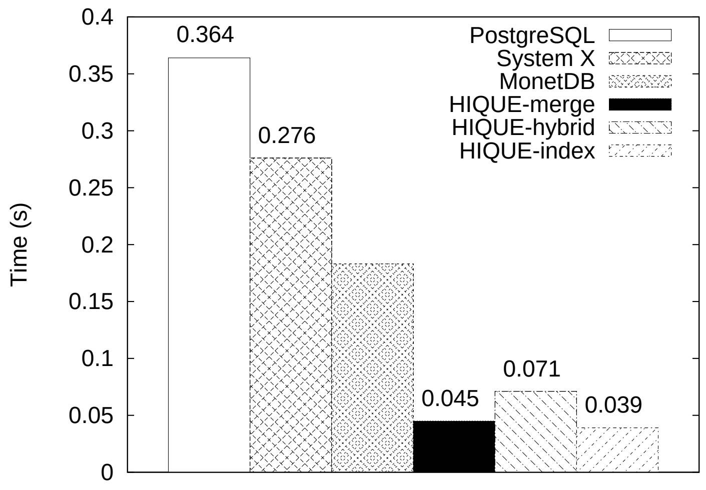


(b) two field projection

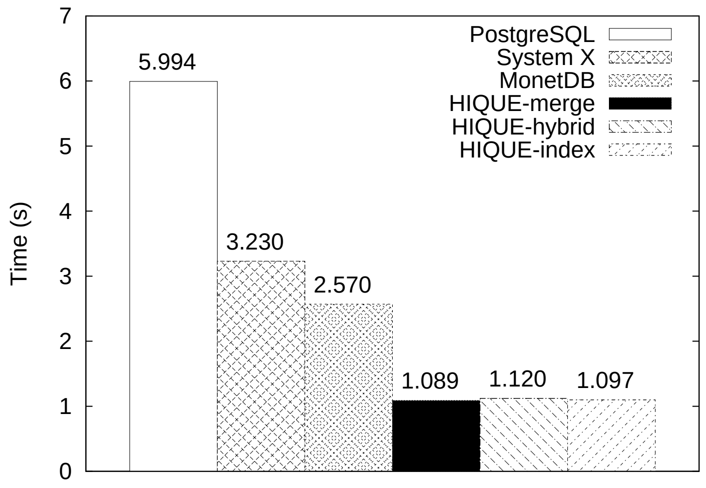


(c) all field projection

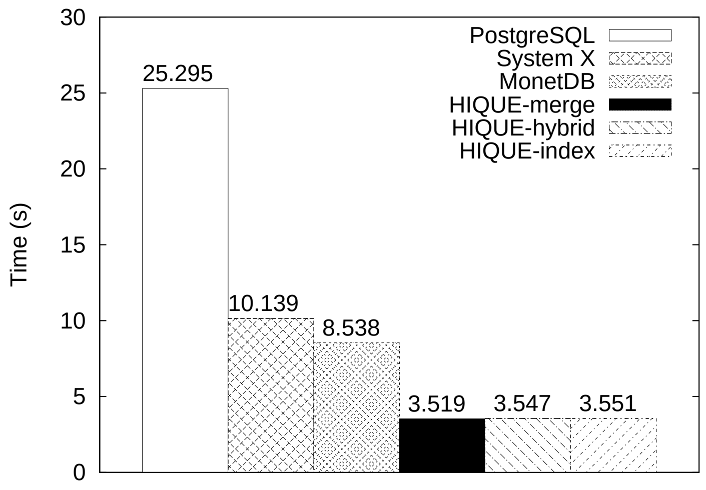
Figure 3.9: Wisconsin Query 1 - response times



(a) count(*)



(b) two field projection



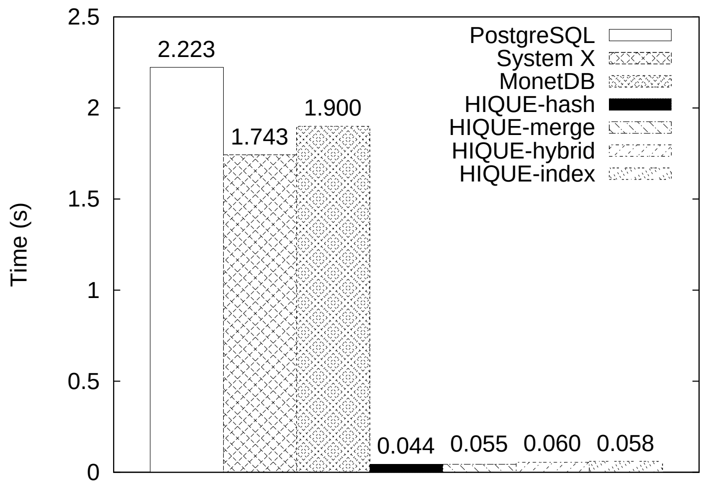
(c) all field projection

Figure 3.10: Wisconsin Query 2 - response times

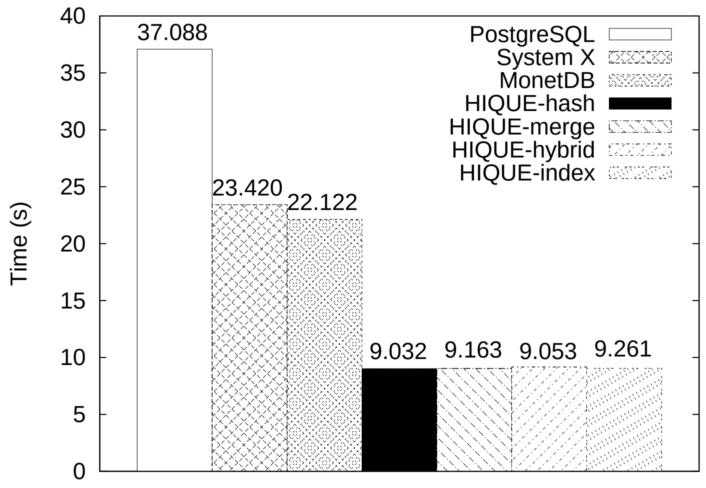
HIQUE widens to a factor of 4.1 over MonetDB, a factor of 6.2 over System X, and a factor of 8.1 over PostgreSQL (for *count*(*) queries). Note that since the `k10_i` tables can fit in the L2-cache, applying hybrid hash-sort staging is slightly slower than simply sorting them with a single quicksort call, as the (redundant for this case) hash-partitioning step is omitted. The performance of join teams justifies our decision of preferring them during query optimisation whenever they are applicable.

The common nested-loops layout of all holistic join algorithms proves its efficiency in inflationary joins, as is the case for Query 3 (Figure 3.11) where the input tables are joined on the `ten` field. Especially when there is no result tuple production, HIQUE outperforms other systems by a factor ranging from 39 to 51. This gap narrows as the projection list grows. This, however, does not prevent our system from maintaining a 2.5-times advantage even when all fields of the joined tables are projected. Note that since there are only 10 distinct values for the join attribute it is possible to use fine-grained partitioning in holistic hash join; its advantage is apparent when the size of the output tuples is minimal (*i.e.*, when *count*(*) is used) but is masked by the output generation cost for wider projection lists.

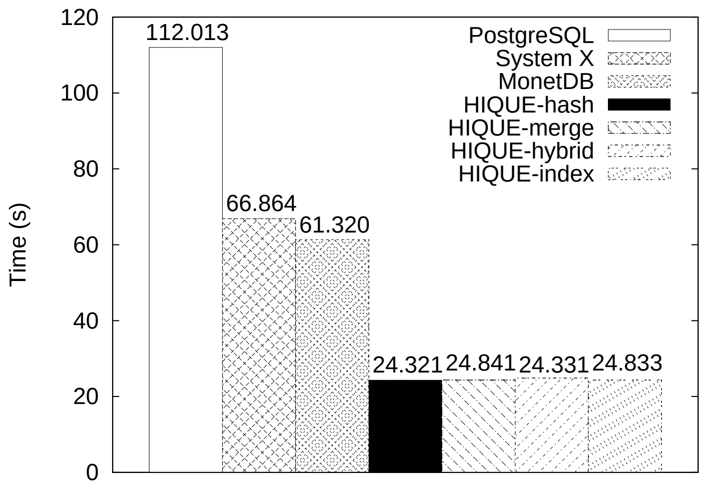
If the result has to be sorted on the join predicate's attribute, the sort-based algorithms exhibit the best performance, as the results of Query 4 (Figure 3.12) verify. Employing hybrid hash-sort-merge join and sorting the result (*i.e.*, two hash-sort staging operations and a sort-merge one) is slightly faster than using holistic merge join to compute the result (*i.e.*, two sort-merge staging operations). If there exist indexes on the join predicate, their efficient implementation and their integration with the merge join code suggest that index-scan staging should be preferred, proving the fastest algorithm overall. The performance advantage of HIQUE over the other systems is narrowed in this case. This is justified by the fact that this query includes a key-to-key join on the `unique2` field, so there effectively are no loops performed in the body of holistic nested loops, as suggested in Listing 3.5: the input tables are linearly scanned, a process calling for consecutive updates of the page and tuple loop bounds for each ex-



(a) count(*)

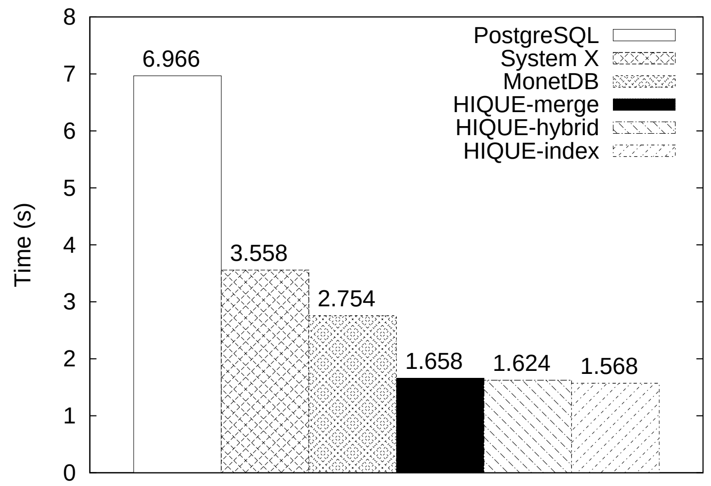


(b) two field projection

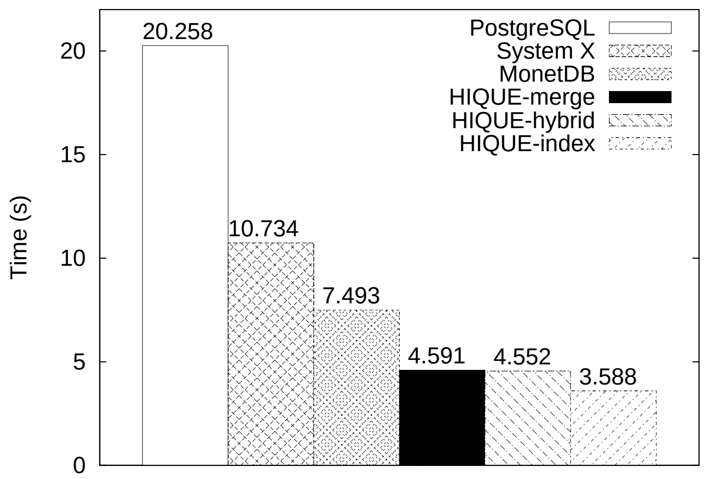


(c) all field projection

Figure 3.11: Wisconsin Query 3 - response times



(a) two field projection

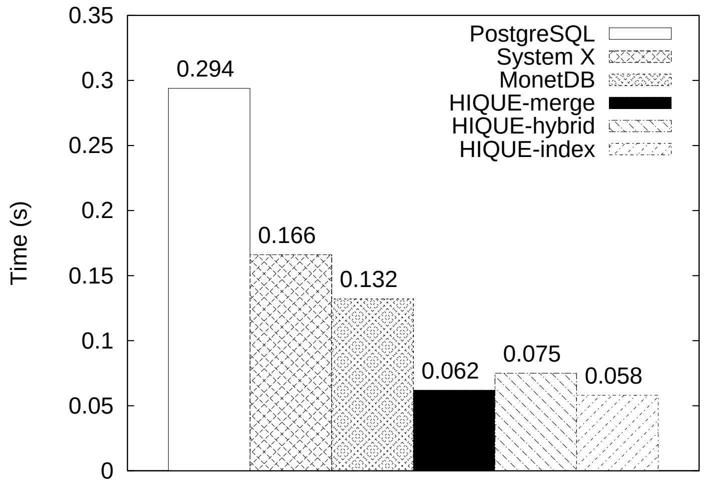


(b) all field projection

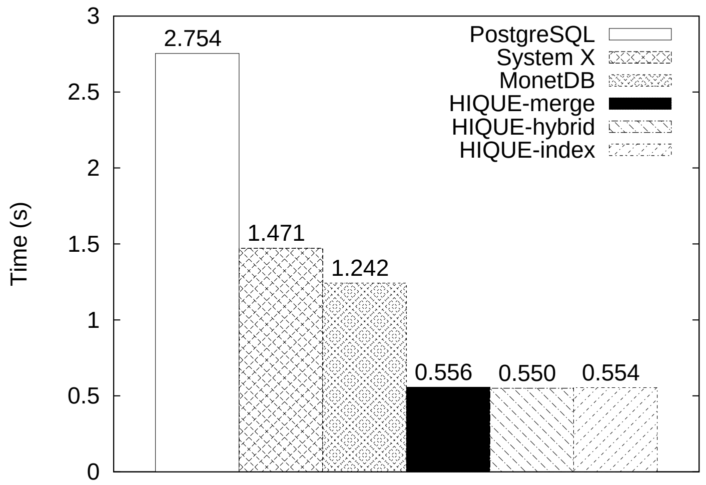
Figure 3.12: Wisconsin Query 4 - response times

amined tuple. This is the worst-case scenario for our holistic join algorithms; it does not, however, prevent HIQUE from outperforming its competition, even for key-to-key joins.

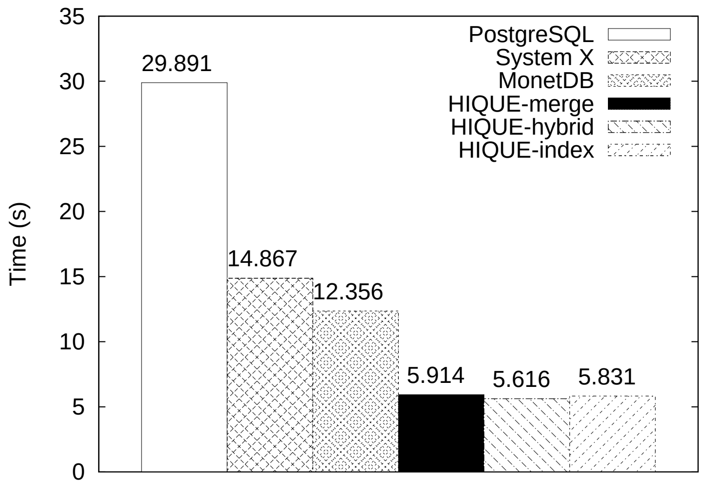
We now move on to scalability experiments. In Figure 3.13 we present response times for Query 5 as we scaled the input cardinality between $10,000$, $100,000$, and $1,000,000$ tuples. The performance of all systems scaled gracefully, with HIQUE always outperforming the competition. HIQUE's performance further corroborates the holistic model's ability to process large datasets by aligning computation to the cache



(a) input: 10k tuples

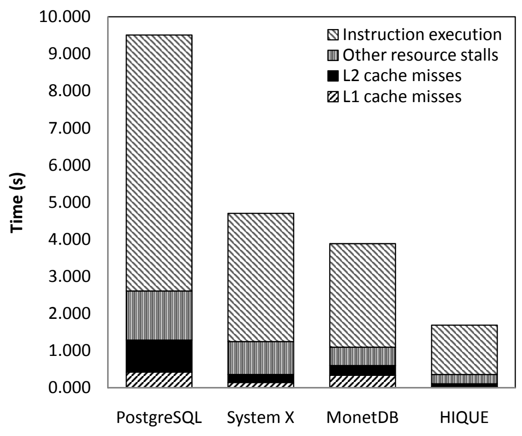


(b) input: 100k tuples

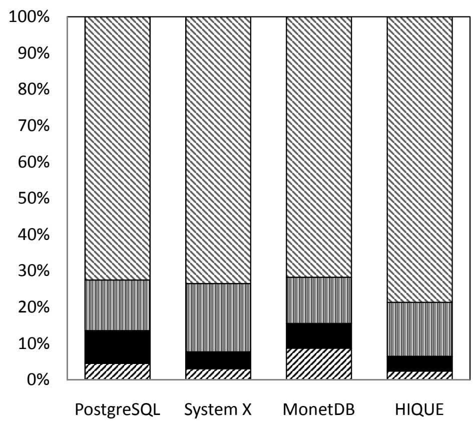


(c) input: 1m tuples

Figure 3.13: Wisconsin Query 5 - response times



(a) Execution time breakdown



(b) Normalised time breakdown

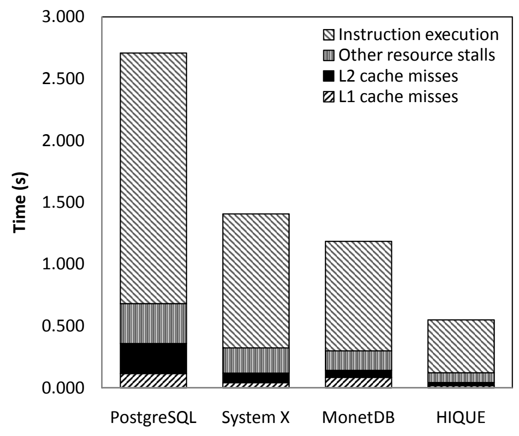
	CPI	**Retired instructions**	**D1-cache accesses**	**D1-cache prefetch efficiency**	**L2-cache prefetch efficiency**
PostgreSQL	0.638	100.00%	100.00%	21.50%	70.74%
System X	0.614	88.65%	82.43%	26.52%	72.99%
MonetDB	0.764	60.30%	56.71%	14.41%	84.74%
HIQUE	0.676	32.11%	30.49%	13.64%	66.15%

(c) Hardware performance metrics

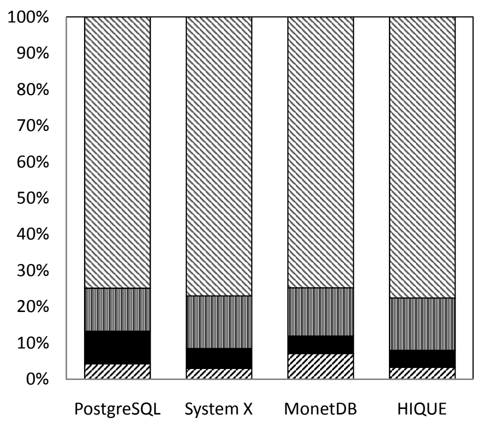
Figure 3.14: Wisconsin Query 1 - OProfile results

hierarchy, even when the size of the input is orders of magnitude greater than the cache capacity.

We have so far used response time as the metric; we shall now switch metrics and focus on the various measurements that we extracted by profiling query execution through OProfile. The profiling results for Query 1 with two fields projected from each table are presented in Figure 3.14. HIQUE bears minimal cache stalls as opposed to competing systems, while all systems have comparable CPI ratios. The small retired instruction ratio of HIQUE indicates that the per-query generated code of the holistic model only uses a minimal set of instructions and requires almost half the instructions compared to other architecture-conscious systems like MonetDB, which translates to reduced computational load. Although HIQUE does not widely exploit hardware prefetching, this is more than counterbalanced by substantially fewer cache misses as well as reduced total data requests. These results show that even though



(a) Execution time breakdown



(b) Normalised time breakdown

	CPI	Retired instructions	D1-cache accesses	D1-cache prefetch efficiency	L2-cache prefetch efficiency
PostgreSQL	0.647	100.00%	100.00%	16.47%	69.72%
System X	0.603	77.65%	64.41%	19.12%	75.12%
MonetDB	0.682	60.01%	60.52%	13.35%	87.16%
HIQUE	0.665	33.45%	28.03%	3.89%	56.80%

(c) Hardware performance metrics

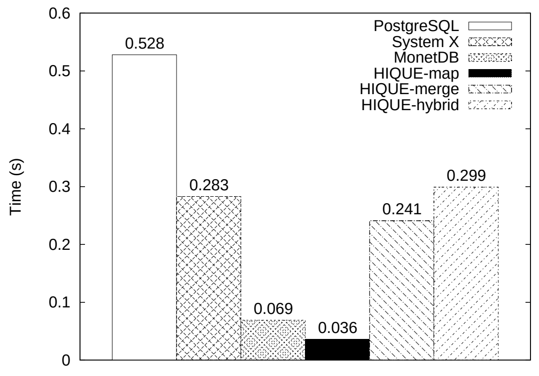
Figure 3.15: Wisconsin Query 5 - OProfile results

HIQUE employs a cache-unfriendly storage representation, unlike other systems (*i.e.*, it uses NSM over vertical partitioning), the generated code still exploits cache locality and reduces the cost of memory accesses.

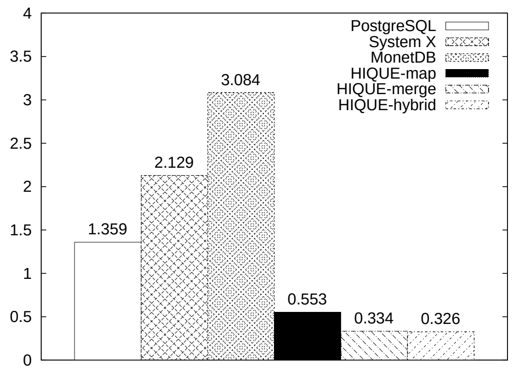
Profiling results for join team evaluation, as is the case for Query 5 with input cardinalities of 100,000 tuples, are shown in Figure 3.15. We observe the same trends for all systems as in single join evaluation. Though HIQUE's overall prefetching efficiency is not improved, note that the number of cache misses is further reduced, as no intermediate results are materialised.

3.8.5.2 Aggregation

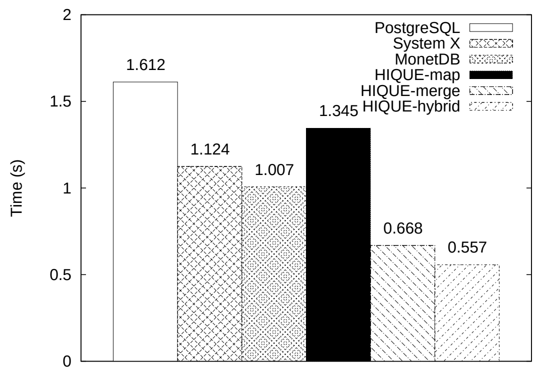
We experimented with varying group cardinalities in Queries 6–9, while input cardinality was set to 1,000,000 tuples. In Query 6, grouping on the *ten* implies only ten groups, so map aggregation is expected to perform best, as there is no need to stage



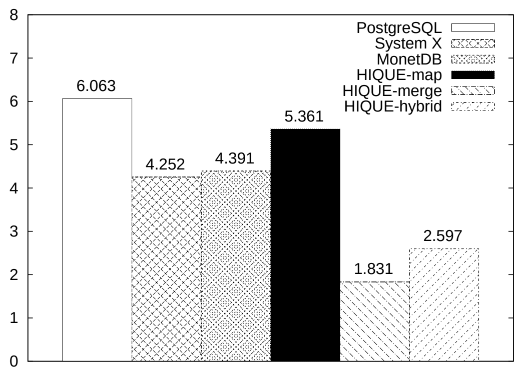
(a) Query 6



(b) Query 7



(c) Query 8



(d) Query 9

Figure 3.16: Wisconsin Aggregation Queries - response times

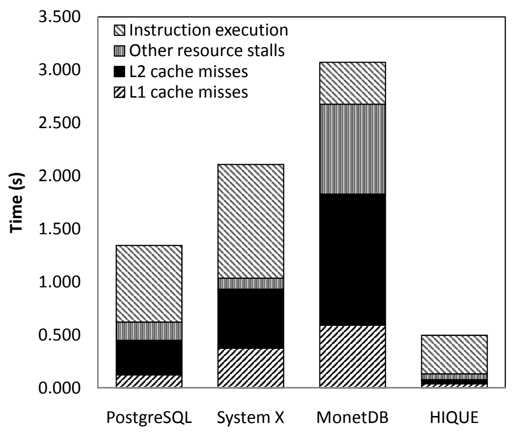
the input. The results, presented in Figure 3.16(a), show that HIQUE is 1.9 times faster than MonetDB, 7.9 times faster than System X and 14.7 times faster than PostgreSQL. Compared to the other holistic aggregation alternatives, map aggregation is 6.7 to 8.3 times faster, thus verifying the efficiency of fine-grained partitioning for a small number of partitions.

As the number of distinct values grows, however, the performance of fine-grained partitioning in map aggregation deteriorates. Recall that fine-grained partitioning requires building and maintaining mapping tables for grouping attribute. This table has as many entries as the number of distinct values for the grouping attribute, which means that for a large number of entries it may span the L1- or even the L2-cache. As a consequence, the system may suffer multiple cache misses for each map lookup. This is apparent in Query 7 (Figure 3.16(b)), where there are 100,000 distinct values of the grouping attribute: holistic sort aggregation and hybrid aggregation outperform map aggregation, due to the efficient implementation of the staging step. We can also spot the advantage of the hybrid algorithm over the sort-based one, stemming from its not having to merge partitions. MonetDB's performance deteriorates as well, while the NSM-based systems adapt better and outperform MonetDB, but they are still 2.7 to 4.3 times slower than HIQUE.

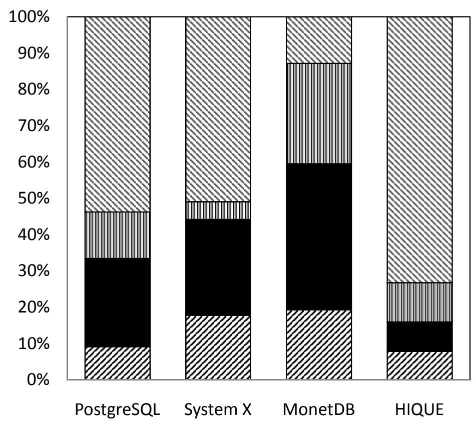
Similar conclusions can be drawn from Query 8 (Figure 3.16(c)), which uses two grouping attributes and results in an output cardinality of $100,000$. In that case, System X and MonetDB perform better and are now about two times slower than HIQUE, while PostgreSQL's performance deteriorates slightly. Again, the holistic aggregation algorithms that employ input staging prove the most efficient approach overall.

In Query 9 the grouping and sorting attributes are identical; the results are shown in Figure 3.16(d). The most appropriate algorithm is holistic sort aggregation, since it does not require output sorting and can produce the result in a single scan. For comparison, hybrid aggregation may avoid the merging of sorted partitions, but introduces another sort-merge operation over the output, so it is 30% slower than the sort-based approach. HIQUE outperforms System X by a factor of 2.2, PostgreSQL by a factor of 3.2 and MonetDB by a factor of 2.4, proving its superiority in this case as well.

We now present profiling results for two indicative aggregation queries. In Query 7 (Figure 3.17) we observe that HIQUE's staging of the input (through hybrid hash-sort staging) and inlining of all aggregation calculations in the generated source code significantly reduces cache miss stalls and results in a lower CPI ratio. In addition, the hardware prefetchers aid in reducing access latencies. Observe that the other systems exhibit suboptimal cache behaviour even though aggregation is, in all likelihood, computed in a single pass of the input table. This performance indicates that iterator-based or column-wise aggregation implementations introduce a significant overhead, compared to the holistic approach. This is mostly evident in the case of MonetDB, which has comparable to HIQUE retired instruction counts, but requires almost as many as



(a) Execution time breakdown



(b) Normalised time breakdown

	CPI	Retired instructions	D1-cache accesses	D1-cache prefetch efficiency	L2-cache prefetch efficiency
PostgreSQL	0.838	76.68%	57.87%	24.21%	69.32%
System X	1.079	100.00%	100.00%	4.85%	51.73%
MonetDB	4.105	45.17%	42.94%	1.61%	51.61%
HIQUE	0.751	39.04%	25.65%	48.76%	84.91%

(c) Hardware performance metrics

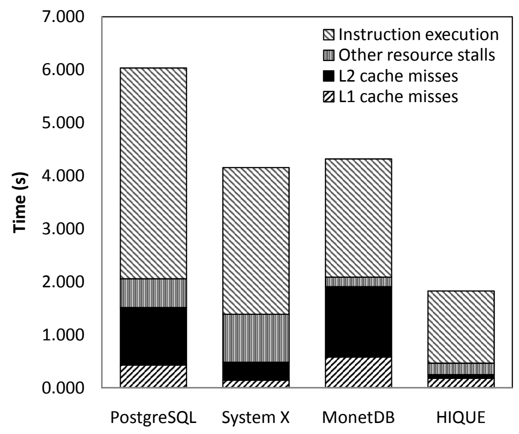
Figure 3.17: Wisconsin Query 7 - OProfile results

double accesses to the D1-cache. These accesses result in further cache misses and performance penalty.

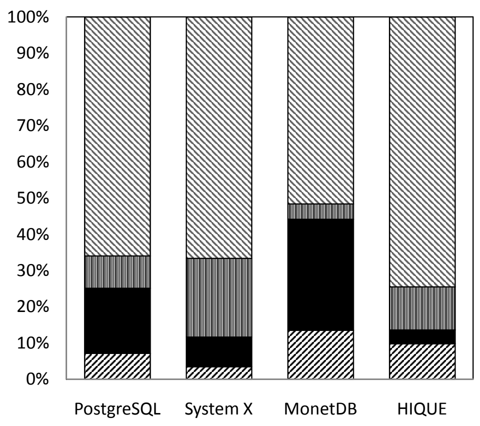
The results for Query 9 are shown in Figure 3.18. MonetDB exhibits increased cache miss stalls and makes worse use of the hardware prefetchers. As the input table is divided into L2-cache fitting partitions during sorting, HIQUE suffers more from D1- than from L2-cache miss stalls. Holistic aggregation, however, exhibits fewer instructions and data reads and writes, while the use of sequential access patterns results in a better utilisation of the hardware prefetchers, therefore shrinking the effect of cache misses.

3.8.6 TPC-H benchmark

We now move on to experimenting with more realistic datasets and queries, namely those of the TPC-H benchmark [73]. In Appendix A we describe the benchmark's



(a) Execution time breakdown



(b) Normalised time breakdown

	CPI	Retired instructions	D1-cache accesses	D1-cache prefetch efficiency	L2-cache prefetch efficiency
PostgreSQL	0.753	100.00%	74.03%	26.52%	70.42%
System X	0.706	90.16%	100.00%	24.81%	70.69%
MonetDB	1.332	56.72%	63.95%	11.68%	58.37%
HIQUE	0.887	38.17%	28.67%	19.95%	81.09%

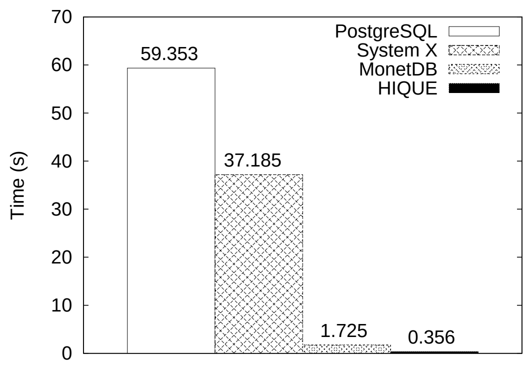
(c) Hardware performance metrics

Figure 3.18: Wisconsin Query 9 - OProfile results

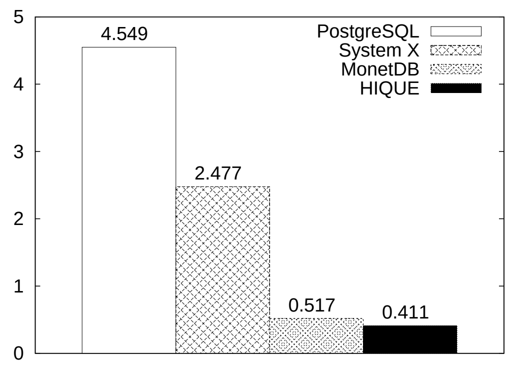
database schema in detail. We used a scaling factor of one. The dataset was generated using the benchmark's generator and we did not alter its tables in any way (*e.g.*, sort or cluster them) before importing them to the systems. The "raw" dataset size was approximately 1.3GB, without including indexes, thus fitting in our system's main memory.

We chose TPC-H Queries 1, 3, 5 and 10. These include highly selective join predicates that cannot be evaluated as join teams, as well as aggregation operations of a varying number of grouping attributes and aggregate functions. Moreover, TPC-H tables have wide tuples spanning multiple cache lines, with only a few fields actually needed by any query. The expectation, therefore, is for MonetDB to benefit from its DSM-based design and outperform all NSM-based systems.

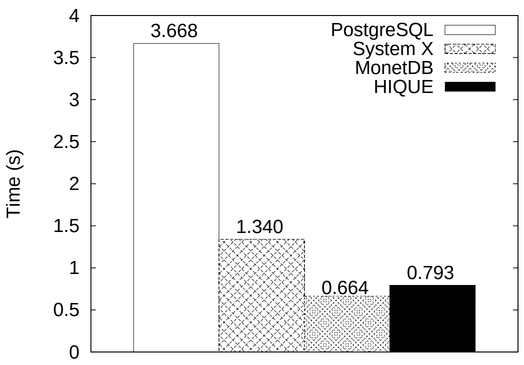
TPC-H Query 1 is an aggregation over almost the entire *lineitem* table (or about 5,900,000 tuples), that produces only four output groups. As the two aggregation at-



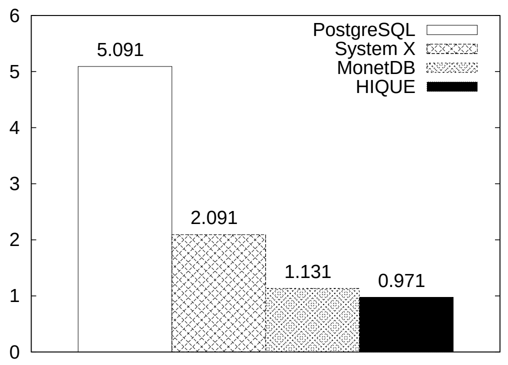
(a) Query 1



(b) Query 3



(c) Query 5

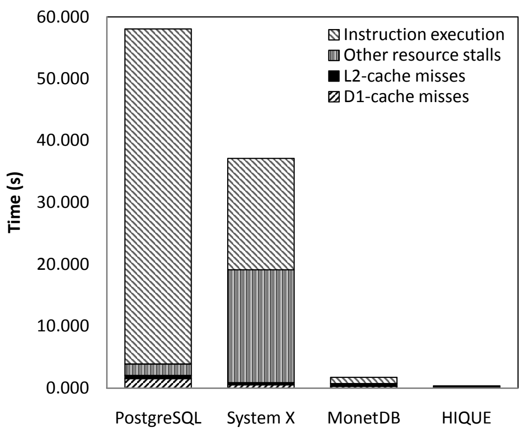


(d) Query 10

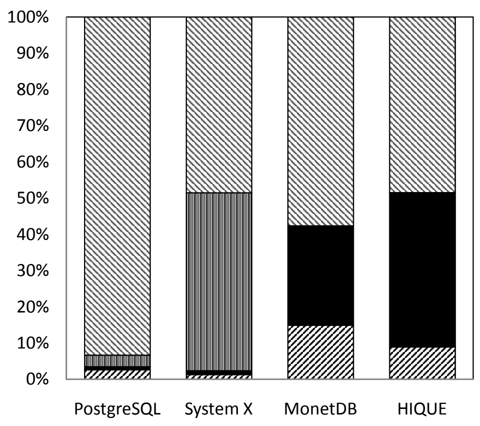
Figure 3.19: TPC-H Queries - response times

tributes have a product of distinct value cardinalities equal to six, the most appropriate holistic aggregation algorithm is hash aggregation with fine-grained partitioning. The results in Figure 3.20(a) show HIQUE outperforming MonetDB by a factor of five and the other NSM-based systems by two orders of magnitude, reaching a 167-fold advantage against PostgreSQL. This is due to the holistically generated code: it includes all selection, grouping and aggregation operations in a single succinct code block that lacks function calls and is tailored towards efficient register utilisation. The generated code resembles Listing 3.8. In such a case, the following accumulated values will be continuously updated in each iteration:

- sum(`l_extendedprice`);
- sum(`l_extendedprice` * (1 - `l_discount`));
- sum(`l_extendedprice` * (1 - `l_discount`)*(1 + `l_tax`)); and



(a) Execution time breakdown



(b) Normalised time breakdown

	CPI	Retired instructions	D1-cache accesses	D1-cache prefetch efficiency	L2-cache prefetch efficiency
PostgreSQL	0.628	100.00%	100.00%	21.28%	61.68%
System X	0.941	37.48%	36.64%	12.57%	61.39%
MonetDB	0.843	2.15%	2.66%	61.35%	96.24%
HIQUE	1.606	0.24%	0.64%	87.17%	57.73%

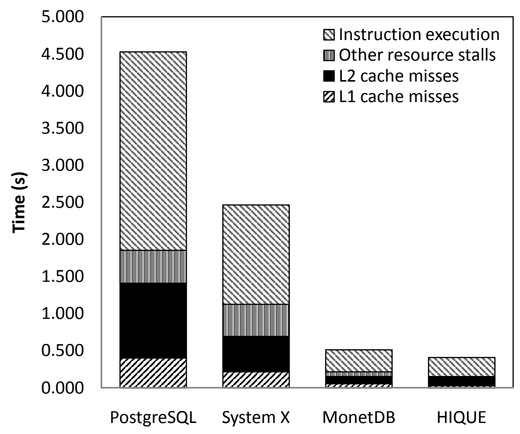
(c) Hardware performance metrics

Figure 3.20: TPC-H Query 1 - OProfile results

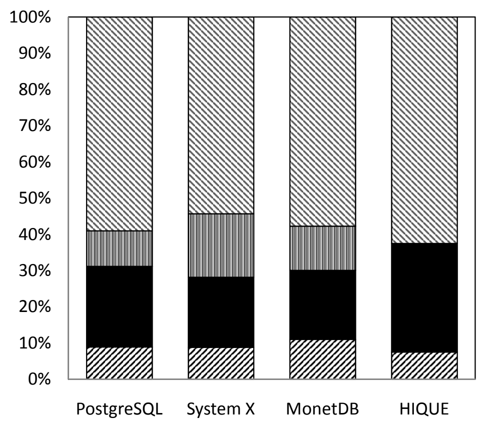
- avg(l_extendedprice).

The holistic model avoids calculation repetition and takes advantage of pipelined execution of all aggregate operations by exploiting register reuse and the processor's superscalar design. The measured performance translates to 662.16 millions of CPU cycles, which is comparable to that of MonetDB/X100's DSM-based approach and 30% faster than MonetDB/X100's NSM-based approach [81]. Hence, we posit that HIQUE generates code that is identical to a hard-coded implementation, thus achieving maximum efficiency in aggregation (at least for NSM-based systems).

In terms of hardware performance metrics, as presented in Figure 3.20, HIQUE achieves the worst CPI ratio. This is expected as it uses a minimal instruction-set



(a) Execution time breakdown



(b) Normalised time breakdown

	CPI	Retired instructions	D1-cache accesses	D1-cache prefetch efficiency	L2-cache prefetch efficiency
PostgreSQL	0.911	100.00%	100.00%	27.49%	58.31%
System X	0.833	64.21%	69.12%	20.72%	63.95%
MonetDB	1.003	10.75%	13.64%	30.82%	88.60%
HIQUE	1.842	4.60%	9.39%	38.63%	49.52 %

(c) Hardware performance metrics

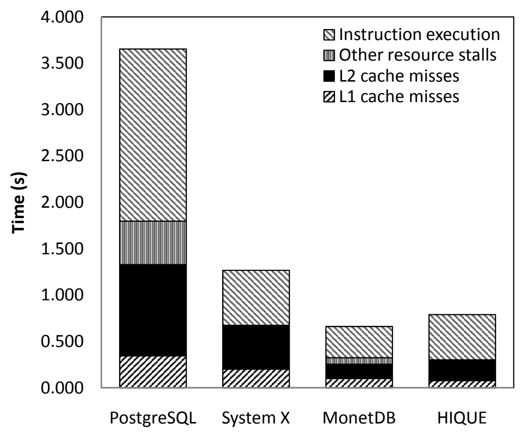
Figure 3.21: TPC-H Query 3 - OProfile results

(ten times smaller than MonetDB's), that mainly consists of expensive memory operations. HIQUE performs fifty times fewer D1-cache accesses than System X, while the input size is comparable for both systems. Its efficiency in exploiting D1-cache hardware prefetching reaches the maximum measured value of 87.17%, whereas MonetDB achieves an even better L2-cache prefetching efficiency ratio of 96.24% by operating over arrays of primitive data types. Furthermore, the number of retired instructions and data references are small for HIQUE, asserting that the holistic model can mask NSM's deficiency, when compared to the DSM.

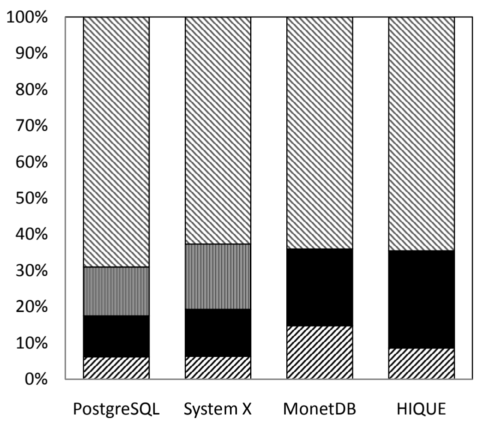
The remaining benchmark queries stress join, aggregation, and sorting performance. In such cases, the holistic optimiser stages all input tables before further operations, which proves an expensive operation over the benchmark tables. For instance, in Query 5 the `lineitem` table participates with only four fields for a total tuple length of 24 bytes, out of the original length of 152 bytes. In addition, only 10% of the `orders` table is used in the key-foreign-key join with the `lineitem` table, making it a highly selective predicate. The combination of the above makes this query a perfect match for DSM-based systems, like MonetDB; vertical partitioning allows fetching only the required fields for each operator's execution. One can make similar observations for Queries 3 and 10 with respect to the participating tables and the selectivity of the join predicates. As a result, HIQUE is 20% and 14% faster in Queries 3 and 10 respectively, and 16% slower in Query 5 when compared to MonetDB. As far as the latter query is concerned, the query plan selected by HIQUE's optimiser computes the join between `lineitem` and `orders` first. The staging cost for these tables and for the join's result for the remaining operations is quite high and impacts execution time. Performance might improve if we adopted a more sophisticated query optimiser.

Compared to the NSM-based systems, HIQUE outperforms PostgreSQL and System X by a substantial margin, with System X narrowing the gap to 40% only for Query 5. The fact that conventional query optimisers are not well-suited for optimising memory-bound queries became evident during early experiments with System X. It wrongly chose to use indexes as access method, leading to a penalty of a factor of ten in execution times. The reported times were obtained only after we manually disabled index scanning.

The OProfile results for Queries 3, 5 and 10 in Figures 3.21 to 3.23 demonstrate that HIQUE aids the D1-cache hardware prefetcher by (*a*) using cache-conscious staging algorithms, and (*b*) ensuring join evaluation stays inside this cache level. On the contrary, MonetDB consistently exploits prefetching to reduce L2-cache misses, due to its column-wise operator implementations. All systems, and especially HIQUE, achieve high CPI ratios, indicating that there are many pipeline stalls, hence superscalar execution of multiple instructions is blocked. Note that, while HIQUE exhibits the worst



(a) Execution time breakdown



(b) Normalised time breakdown

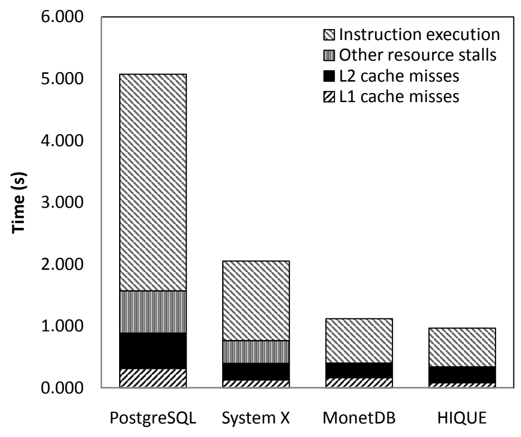
	CPI	Retired instructions	D1-cache accesses	D1-cache prefetch efficiency	L2-cache prefetch efficiency
PostgreSQL	1.237	100.00%	100.00%	16.59%	58.01%
System X	1.253	41.17%	41.26%	20.76%	65.14%
MonetDB	0.906	23.14%	20.19%	26.60%	90.52%
HIQUE	1.027	26.86%	24.33%	33.11%	53.76%

(c) Hardware performance metrics

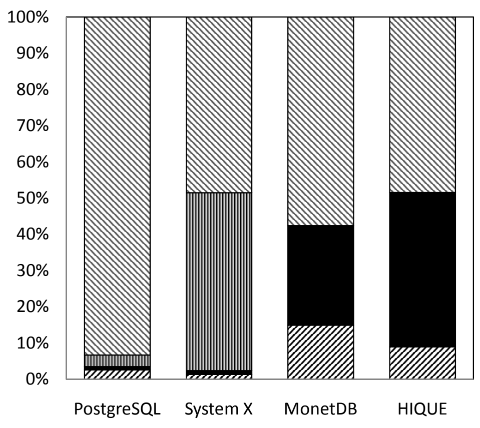
Figure 3.22: TPC-H Query 5 - OProfile results

CPI ratio in TPC-H queries, the actual percentage of time spent on instruction execution is comparable to the other systems. This is most likely because the latter can achieve a lower CPI ratio when the processor is not stalled due to memory operations, by providing the CPU with abundant instructions. Conversely, HIQUE displays substantially fewer retired instruction and D1-cache access counts than the other NSM-based systems and is surpassed by MonetDB only in Query 5.

The breakdown of execution times reveals that, despite the improvements the holistic model introduces, it cannot overcome the restrictions that modern processors impose on the performance of data-intensive applications. No matter how sophisticated the operation of caches and memory becomes, it still cannot provide data to the archi-



(a) Execution time breakdown



(b) Normalised time breakdown

	CPI	**Retired instructions**	**D1-cache accesses**	**D1-cache prefetch efficiency**	**L2-cache prefetch efficiency**
PostgreSQL	1.061	100.00%	100.00%	15.58%	65.73%
System X	1.057	55.93%	67.36%	20.06%	63.29%
MonetDB	1.033	25.14%	26.89%	18.70%	76.23%
HIQUE	1.115	20.20%	26.71%	32.69%	56.36%

(c) Hardware performance metrics

Figure 3.23: TPC-H Query 10 - OProfile results

tectural registers in rates comparable to the CPU pipeline's processing throughput: the CPU will continue to stall over memory accesses. Still, the holistic model provides code simplicity and enhances cache locality during execution, therefore reducing the number of instructions and data accesses required to evaluate queries. That way, both the processor and the memory subsystem are stressed to a lower extent, leading to a significant speedup of query evaluation. This allowed our NSM-based system to achieve performance that was so far conceivable only for systems employing vertical partitioning.

TPC-H	**SQL processing (ms)**			**Compilation (ms)**		**C file sizes (bytes)**	
Query	**Parsing**	**Optimisation**	**Generation**	**with `-O0`**	**with `-O2`**	**Source**	**Shared library**
1	21	1	1	121	274	17733	16858
3	11	1	2	160	403	33795	24941
5	11	1	2	201	578	43424	33088
10	15	1	4	213	619	50718	33510

Table 3.5: Query preparation cost

3.8.7 Query preparation penalty

The main disadvantage of per-query code generation is the cost for emitting and compiling the query-specific source code. To quantify this cost we report the preparation times for the TPC-H queries in Table 3.5. We separately show the query parsing, optimisation, code generation and compilation times, as well as the sizes of the generated source and shared-library files. The time for parsing, optimising and generating code is trivial (less than 20ms). On the contrary, compiling the generated code takes longer and compilation time depends on the optimisation level. Code compilation takes 121–213ms when no compiler optimisation is applied (`-O0` compiler flag), but needs 274–619ms when the optimisation level is increased (`-O2` compiler flag). The generated source and shared-library file sizes are limited to a few tens of kilobytes.

The query preparation time is not negligible and can cover a significant percentage of execution time even for simple queries. In such cases it is preferable to avoid applying complex compiler optimisations that increase compilation time, as the difference in execution time will be intangible. It is quite common, however, for systems to store pre-compiled and pre-optimised versions of frequently or recently executed queries. This is certainly applicable in HIQUE, especially if we take into account the small size of the generated shared-library files. Moreover, as we have seen in previous sections, the performance advantage gained through code generation is so wide, especially for non-trivial queries, that justifies the generation cost even for ad-hoc queries.

3.9 Further reading

The extension of interpreters to generate native code on-the-fly is a widely adopted technique. The Java HotSpot Compiler [50] identifies frequently executed methods inside a program during its first run and then translates them in native code. This reduces the overhead for calling these functions and further optimises their implementation. Code generation has also been used to computationally intensive workloads like the DSP transformation [61], allowing for the application of extensive algorithmic and code optimisations.

Regarding query evaluation, although a primitive form of code generation was used even in System-R [9], the adoption of the iterator model [33] has dominated query engine design. Code generation was revisited in the Daytona data management system [37], which was capable of on-the-fly generation of query-specific code. It relied, however, on the operating system to perform most of the functionality a DBMS would traditionally provide (*e.g.*, buffering, concurrency control). Similarly, the authors of [63] presented a Java prototype system employing dynamic query compilation. The bytecode for each query was then invoked by the virtual machine. Still, this system employed iterators for operator communication and execution, using the code generator only to remove virtual functions from the body of iterators. Moreover, it did not present any novel query processing options, *e.g.*, joins were exclusively evaluated through preconstructed join indexes, while execution was confined in main memory since all tables were transferred to RAM during startup. As the tables were stored in the form of contiguous arrays of tuples, this complicated concurrency control and recovery. In contrast, HIQUE not only eliminates iterator and function calls, but is designed from the ground-up with hardware efficiency in mind, without any restrictions on the storage layout.

Chapter 4

Multithreaded query execution on multicore processors

4.1 Introduction

We have so far focused on improving the single-threaded performance of main-memory query execution through code generation. In this chapter we present a detailed analysis of multithreaded query execution on multicore processors. Extending the elementary query evaluation operators for multithreaded processing is far from straightforward. Multithreading introduces resource contention that penalises scalability; cores share resources both at the hardware (caches and physical memory) and at the software (lock-based synchronisation) levels, thereby restricting the degree of parallelism. To counter that we posit that multiple threads should independently process cache-resident data to the highest possible extent, thereby minimising contention and enhancing parallelism. To that end we: (*a*) give a uniform framework to generalise the query processing algorithms of the holistic model for multithreaded execution, and (*b*) present an analytical model to estimate the multithreaded performance of the proposed algorithms. The model statically estimates the speedup of multithreaded execution.

Multicore means shared memory. Modern CPUs integrate multiple cores and provide hardware support for parallel processing. Their architecture resembles shared-memory systems: the cores share main memory and, possibly, the lowest level of the cache hierarchy. Query evaluation on this type of parallel systems has been tackled before (*e.g.*, [34]); previous work, however, has not taken into consideration the cache hierarchy and its impact on multithreaded execution. As shown in [3, 55], database workloads suffer from excessive stalls due to the high latency of memory operations. This is aggravated in multicore processors as the memory subsystem serves requests from multiple cores [57].

Busier is faster. Multicore processors have more "raw" processing power, but it is not harvested when executing data-intensive workloads. To alleviate this, we propose to exploit cache locality by maximising the amount of processing whenever a data block is in the CPU caches. For example, "pushing" more query-relevant processing into partitioning an input may result in an extra per-thread processing cost for the operation of 13%; however, this means that the cores are now busier with processing the input instead of waiting for memory operations. The busier a core is with processing cache-resident data, the less it contends with the other cores for accessing the memory. The extra per-thread cost in the previous example results in an almost three-fold improvement in the Cycles Per Instruction (CPI) ratio when the technique is applied to a quad-core Intel Xeon E5420 CPU. In turn, this results in a higher speedup of the execution of the entire query.

We apply this approach to the prominent query evaluation algorithms and provide a uniform framework for multithreaded processing. Our goals are to: (*a*) minimise data transfers from main memory, and (*b*) evenly distribute both work and data across multiple threads. To minimise synchronisation overhead, we assign different input and output streams to each thread; locking (if any) is performed on a coarse granularity, thus aiding parallel execution.

Modelling scalability. To assess multithreaded execution, we analytically model the effect of input cardinality, tuple size, selectivity, and projectivity to performance, according to the characteristics of the host hardware. We introduce the *multithreaded utility ratio*: the ratio of the time spent for fetching each input unit to its total processing time. High values of this ratio denote fetch-dominated operations; in this case memory accesses incur an inflated *effective* cost, thus restricting scalability. Conversely, low values of the utility ratio show that there is sufficient computational load to overlap with data fetching, so thread contention for memory accesses is limited and scalability is enhanced. Using this ratio, we analytically estimate the query processing cost and the expected speedup of multithreaded execution.

Contributions. The main contributions of this work can be outlined as follows:

- We give a uniform framework to extend existing query processing algorithms for multithreaded execution on multicore CPUs.
- We present partitioning and buffering techniques that determine which part of the input each thread processes and where in the memory hierarchy it is buffered.
- We introduce an analytical model to accurately estimate the speedup of multithreaded query execution.

The rest of this chapter is organised as follows: in Section 4.2 we give a general framework for multithreaded execution and algorithms for the main query processing operations. We analytically model the proposed algorithms in Section 4.3, while in Section 4.4 we conduct an experimental study of our proposals. We finally present work for further reading in parallel query optimisation in Section 4.5.

4.2 Multithreaded processing

We now provide a framework for parallelising the most frequently used query processing algorithms [33]: sorting, partitioning, join evaluation and aggregation. Our premises are:

- We use the N-ary Storage Model (NSM) with tuples stored consecutively within pages of 4kB. Each table resides in its own file on disk, and a storage manager is responsible for caching file pages in the buffer pool. We don't use vertical partitioning as we wanted to keep the same baseline with most commercial and research database systems. We also wanted to explicitly account for the interaction between the query engine and the storage manager in our analysis.

- Our techniques only depend on the number of threads that can be efficiently supported by hardware. Naturally, the techniques need to be "fitted" to a specific CPU but the approach is uniform and remains largely the same across CPUs. For instance, the Intel Xeon 5400 series of quad-core processors of Figure 2.3 (the one also used in our experiments) has per-core pipelines supporting out-of-order execution. However, there is no in-core support for TLP so only four concurrent threads are supported by hardware. We will be pointing out any such subtleties that require fitting the data flow to each CPU.

Our approach stems from the observation that CMPs are in essence shared-memory systems. Parallel query evaluation has been tackled before [21, 34]; the rule of thumb is to split the input in disjoint partitions and then process them in parallel. However, the naïve extension of this technique for multicores would not take advantage of the cache hierarchy's buffering effect. For example, synchronising accesses to a shared hash table would severely penalise performance, in case the table does not fit inside caches [17]. To that end, we fine-tune the implementation of partitioning and parallel processing to the characteristics of multicore processors. We focus on reducing concurrent memory requests by interleaving memory accesses and cached data processing to the highest possible extent. This technique keeps the cores busy and reduces memory stalls. We also avoid using fine-grained thread synchronisation. Threads are initialised once for each operation and use *restricted affinity* (*i.e.*, they are assigned to a specific core); that way they can run with the minimum synchronisation overhead. Finally, we pay special attention to avoid false sharing: we align shared data (such as mutexes) with the size of the cache line and replicate writeable variables and buffers for each thread.

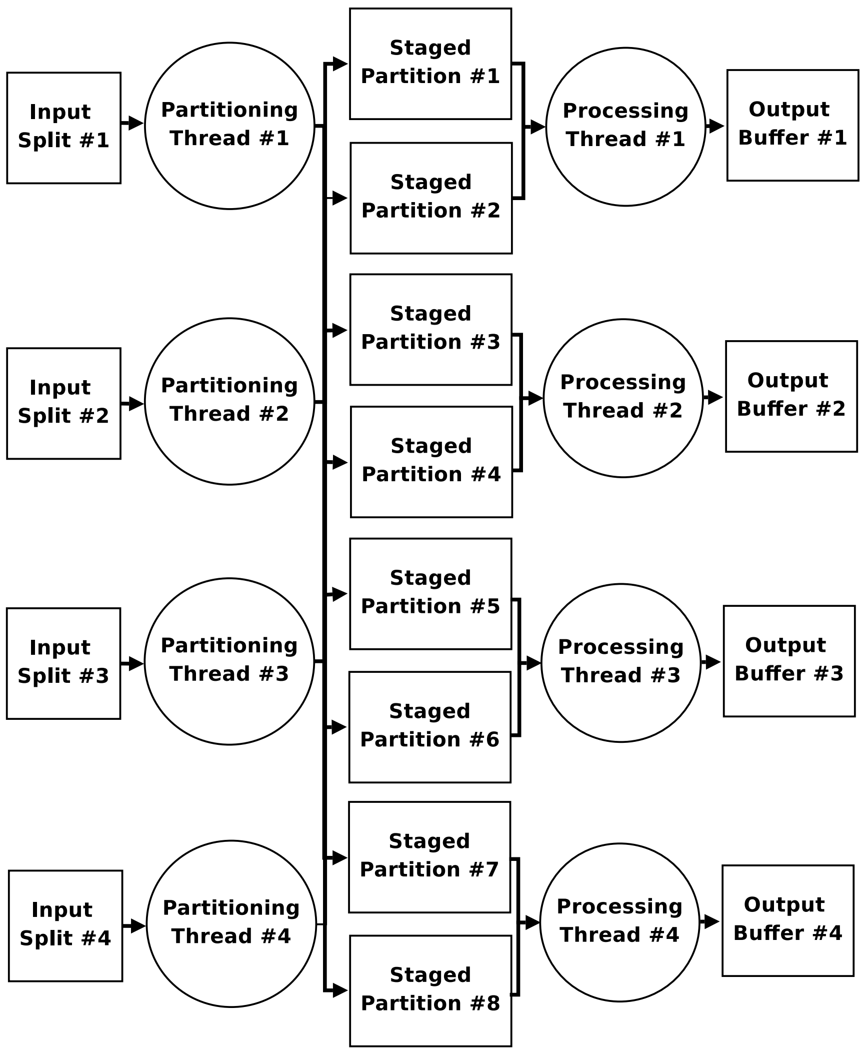


Figure 4.1: Multithreaded operator implementation

An example of the uniform framework for the implementation of each operator is shown in Figure 4.1. The input is first split in as many "splits" as there are threads of execution that can be efficiently supported by hardware (*e.g.*, four splits for the Intel Xeon 5400, eight splits for the Intel Core i7). For each primary table we divide the total page count by the number of threads; each split is assigned to one thread. Next, we partition the input in disjoint partitions using the specified number of threads. Each thread scans its split and writes tuples to appropriate output partitions. We do not use tuple references, but copy to the partitions the fields required for further processing. That way we increase cache locality and avoid uncontrollable and costly random access patterns outside the cache hierarchy. After partitioning all inputs, we invoke a new team of threads to process the partitions. A set of disjoint partitions is assigned to each thread and processed with no synchronisation overhead. Threads store output tuples to individually assigned output buffers. The set of all output buffers is the final operator output that will either be used by subsequent operators, or be forwarded to the client as a final result.

4.2.1 Data staging

During data staging selections and projections are applied and the input is appropriately “formatted”. For example, for merge join, inputs are sorted, while for hash join the input is hash-partitioned. Our measurements have shown that data staging can take up to 90% of the total execution time of an operator. It is therefore important to adapt all common staging algorithms for multithreaded execution.

Our algorithms use partitioning for multithreaded processing with minimal overhead. The main partitioning algorithms are: (a) range partitioning, (b) hash partitioning, and (c) value mapping. Range partitioning generates partitions containing tuples within a specific range of values of the partitioning attribute. Value distribution statistics, *e.g.*, histograms, can be used to extract the bounds of each partition to balance the distribution of tuples to partitions. Hash partitioning uses hash and modulo computations to map tuples to partitions with no assumption on value distributions. This leads to similarly sized partitions. Finally, the values of the partitioning attribute can be directly mapped to partitions, a technique applicable in case the partitioning attribute has only a few distinct values. We elaborate on each staging algorithm.

Sorting. We build on the AlphaSort algorithm [58], where input partitions fitting the cache hierarchy are sorted with *quicksort* and then merged through multi-way merging. We use N hardware-supported threads to sort partitions and assign $\left(\frac{1}{N}\right)^{\text{th}}$ of the total number of input pages to each thread. Each thread applies quicksort to partitions that fit inside its share of the lowest cache level. For example, in the Intel Xeon processor

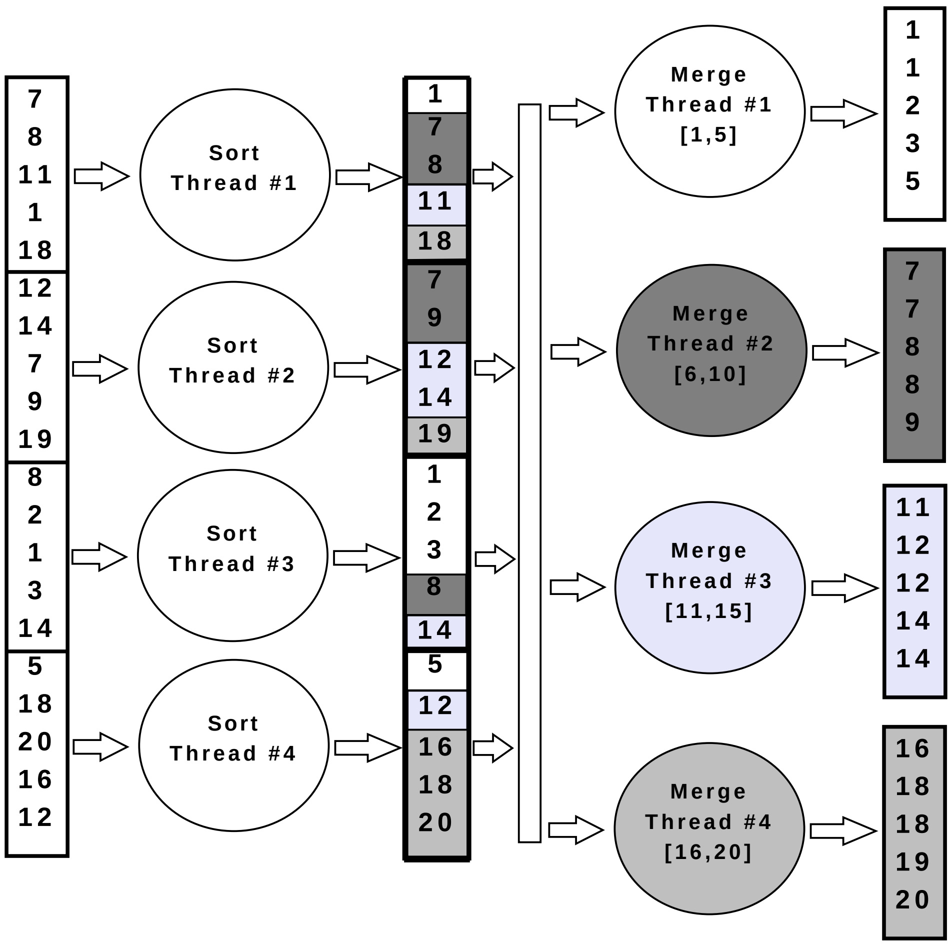


Figure 4.2: Multithreaded sorting

of Figure 2.3 the partition size is less than half the size of the L2-cache; for the AMD Phenom processor, where each core has its own L2-cache and shares the on-chip L3-cache, the partition size should be less than a quarter of the L3-cache capacity.

After sorting each partition we invoke *N* new threads to merge the partitions. We use range partitioning to separate work. We assign a specific range of values to each thread, as shown in Figure 4.2 (value ranges are individually colored). Each thread processes only the part of each partition that contains values in its assigned range. The sorting threads specify the tuple range for each merging thread in each partition during the previous step. Through value statistics, it is possible to assign ranges to threads so that each thread will output approximately the same number of tuples. That way all threads will have comparable processing rates. Each merging thread maintains a heap of the currently examined tuples from each partition to identify the tuple with the minimum value. Note that no synchronisation is needed during sorting since threads process disjoint datasets.

We tackle data skew using static and dynamic techniques. To assign value ranges to threads, the system exploits histograms and the cardinality of each table to compute ranges that are estimated to create partitions of similar size. We further adopt a dynamic approach similar to the one presented in [47]. Threads are initially assigned a specific value range, assuming that each thread will approximately process $\frac{|K|}{N}$ tuples, where $|K|$ is the input cardinality. When a thread has processed $(\frac{|K|}{N} + Thres)$ tuples, where threshold $Thres$ denotes the expected overflow factor, the input is skewed, so other threads have already processed the tuples within their assigned value range. At that point, all threads join and the remaining input is redistributed to them. This process is repeated until the input is entirely sorted.

Partitioning. Hash and range partitioning use the same multithreaded process, the difference being the function used to forward tuples to partitions. As shown in Figure 4.1, each thread scans its split of input pages and forwards tuples to partitions by applying a partitioning function. We use buffering on a page granularity, as each thread uses one page from each partition to store tuples. When a page fills up, the thread replaces it with a new one through a call to the storage manager.

This simple approach has two drawbacks. Firstly, storage manager interaction needs to be an atomic operation; thus, requests to the storage manager need to be serialised. Secondly, and more importantly, the only per-tuple processing is the evaluation of the partitioning function. This requires at most a few tens of CPU cycles, while fetching data from main memory costs an order of magnitude more. Since memory is a shared resource across all cores, if multiple cores issue memory requests concurrently, memory operations will be queued [57] and their *effective* latency will increase; this restricts the scalability of multithreaded partitioning. We have verified this hypothesis

Algorithm	**Threads**	**Time**	**CPI**	**L2-cache misses**	**Pending requests**
Partition	1	0.085s	1.68	335	4672
	4	0.072s	3.86	699	11086
Partition and Sort	1	0.148s	1.21	342	7556
	4	0.083s	1.41	661	9008

Table 4.1: Profiling results for partitioning

for the Intel Xeon 5400 processor, which uses a single memory bus, but it is likely to hold for processors with multiple embedded memory controllers.

The solution we propose is to maximise reuse by processing the input to a greater extent once it is cache-resident. One way of doing so is sorting each full partition page before replacing it with a new page. That way, the partition page is prepared to be further processed at a negligible cost. If the number of partitions is moderate we can expect the page to be inside the L2-cache (or even the L1-cache) before being sorted, thus sorting is performed efficiently. Since the partitions end up containing sorted pages, one merging phase per partition is needed to sort it. This step can be integrated with query evaluation, as we shall see in Sections 4.2.2 and 4.2.3. This technique resembles the *Map-Reduce-Merge* algorithm [75]; we combine partitioning with page sorting to better adapt execution to the characteristics of CMPs.

To quantify the difference between partitioning alone, as adopted in previous work on parallel DBMSs, and the proposed integration of page sorting to partitioning, we compare the results of hardware profiling for these two techniques on the reference CPU of Figure 2.3, as given in Table 4.1.[1] The input table has 1,000,000 tuples of 72 bytes each. The overhead of partitioning the input while sorting each partition page in single-threaded execution is 74% over partitioning the input alone, but is reduced to 13% when four threads are used. Furthermore, though in both cases the L2-cache

[1]We show sample counts for L2-cache misses and pending memory requests, as extracted with the OProfile tool [59].

Listing 4.1: Accessing the mapping directory

```
int offset = lookup(directory, value);
if (offset < 0) {
  lock(directory.lock);
  offset = lookup(directory, value);
  if (offset < 0) offset = insert(directory, value);
  unlock(directory.lock);
}
```

misses increase (due to the interaction with the storage manager and thread synchronisation), simple multithreaded partitioning increases the CPI ratio by a factor of 2.3 and the number of pending memory requests by a factor of 2.4; combined partitioning and sorting results in a slight increase of a factor of 1.2 for the CPI ratio and the pending requests. The above show that, though the same dataset is accessed in both cases, the cores need to wait longer for memory operations in hash partitioning alone because they all attempt to access main memory at the same time. When combining partitioning with sorting, while one core is busy sorting a page, the remaining cores face less contention for memory operations. Synchronisation overhead is also reduced, as the time to obtain a reference to a new page from the storage manager is only a small portion of the time to fetch a page and sort it.

Value mapping. If the partitioning attribute has a small number of distinct values, one can map each value to a specific partition, using a directory to maintain this mapping. We use a sorted array of attribute values and perform binary search for lookups. Hash-based solutions are also possible; we preferred binary search to avoid the effect of data skew in a data structure that is heavily used. Note that there is a limit beyond which this approach becomes inefficient: if the partitioning attribute has a high distinct cardinality the mapping directory will span outside the L1-cache and accesses will trigger cache misses.

Each thread scans its assigned input split and copies its tuples to the corresponding partitions. Since tuple processing requires a directory lookup (and may trigger an insertion), there is sufficient computational load to overlap with memory operations, resulting in considerable speedups. The more entries the directory has, the closer to linear the speedup will be: the time spent on lookups dominates the cost of fetching data. Note that since the number of distinct values is small, all cores share the same directory. In Listing 4.1 we show the code to synchronise directory insertions and lookups. The synchronisation penalty is paid until the directory contains all entries. From then on threads replicate the directory inside each core's L1-cache and perform lookups without locking it.

4.2.2 Join evaluation

Merge Join. The input tables are staged by sorting them on the join attributes. After sorting the input tables, we initialise a new set of threads to evaluate the join predicate. Each thread processes a specific value range of the join attribute and evaluates the join for corresponding partitions; there is also a separate output buffer per thread. Data skew is treated using the techniques we described for merging sorted partitions in Section 4.2.1. As partitions are disjoint there is no synchronisation overhead. The only performance restriction is the ability of the memory subsystem to provide the cores with data in the rates the threads consume them.

Hash Join. Recall that during hash partitioning each page of each partition is also sorted. Thus, there is no need to build per-partition hash tables during the join phase. Each input is partitioned using a fanout wide enough for the largest corresponding partitions of each table to fit in the lowest cache level. For example, if we join table A of size 100MB with table B of size 250MB using four threads on a quad-core processor with a shared 8MB L2-cache (and no L3-cache), the partitions of both tables should be smaller than 1MB: during the join phase the threads sharing the L2-cache will be joining two partitions each. Thus, we use a fanout of at least 250 for both tables (*i.e.*, the size of the largest table over the target size of each partition). In practice, it is better to use higher fanouts (even double). Doing so will amortise the variance in partition sizes, and procure for space to hold instructions and data belonging to the operating system and the storage manager, as well as the merging buffers that will be shortly introduced.

After partitioning the inputs and individually sorting the partition pages, we start new threads to join the corresponding partitions. Each thread processes a disjoint set of partitions, so all threads work independently. To address data skew, partitions are allocated to threads so as their accummulated size to be approximately the same for each thread. The first step is to merge the pages of each partition and generate a fully sorted partition. As this is repeated for all partitions, we dedicate a single output buffer per thread and we (re)use it to store the tuples of each partition in sorted order. Since the partition size is small, one can expect the merging buffers for all threads to remain inside the lower cache level during the join process, thus avoiding accesses to main memory. After merging we join corresponding partitions just as in merge join. Note that the partitions have already been brought in the lower cache level so this step is efficient. Our hybrid join technique interleaves computation with memory operations and efficiently exploits the cache hierarchy; at the same time it incurs negligible synchronisation overhead.

Map Join. If the join attributes have a small number of distinct values we stage the inputs using value mapping. We then join the partitions for the same attribute value with nested loops join. Map join applies only if both inputs have a small distinct value cardinality. Its performance degrades as more entries appear in the mapping directory: as the directory grows it will not fit the L1-cache, so lookups will trigger cache misses.

4.2.3 Aggregation algorithms

Sort aggregation. We first sort blocks of the input on the grouping attributes. In line with performing as much computation as we can during data staging, we modify the merging phase of Section 4.2.1 to incorporate the on-the-fly evaluation of the aggregate functions. That way, we avoid flushing the sorted output to memory and refetching it to the caches to compute the aggregate values of each group. Doing so reduces main memory accesses and enhances parallelism.

Partition-based aggregation. We first hash- or range-partition the input and individually sort the pages of each partition (see also Section 4.2.1). The partitioning fanout can be smaller than the one used in join evaluation, as there is only one input. Next, we invoke new threads, each processing disjoint sets of partitions. For each partition, the thread merges the sorted pages; instead of saving the output to a merge buffer (as with join evaluation) it directly evaluates the aggregate values per group and outputs them, which significantly reduces the number of memory operations.

Map aggregation. If all grouping attributes have small distinct value cardinalities, we can aggregate in a single pass. The input is first split to the number of threads used. We keep a mapping directory for each grouping attribute, with directories shared across threads. We generate an array of aggregate values, one per aggregate function per thread. A thread looks up each tuple in each directory and finds the row to update in its private array of aggregate values, as described in Section 3.5.3. Since the distinct value cardinality for the grouping attributes is small, the mapping directories quickly fill up and hold all input values; thus, aggregation bears minimal synchronisation overhead. After processing all tuples, the individual aggregate value arrays are "merged" depending on the aggregate function (*e.g.*, for `sum`, corresponding group values are added).

The scalability of multithreaded aggregation grows with the size of the mapping directories, as lookups become more expensive and overlap to a greater extent with input tuple fetching. Directories, however, should not grow too large: as the directories and aggregation arrays grow (the size of each aggregation array being the product of distinct values of each grouping attribute), they start "spilling" outside the L1-cache, or even the L2-cache, so lookups and aggregate value updates are likely to trigger cache misses. This is aggravated by multiple threads sharing the lowest cache level, so the cache capacity available per thread is reduced.

4.3 Performance modelling

In CMPs, multiple threads can work independently provided there is no synchronisation overhead and their datasets are cache-resident; this would provide linear speedups. This is not always feasible, though, as threads will contend to access memory-resident data. Consider N threads processing a single relation: they will have to share the physical memory. If all need to fetch data at the same time, requests will be serialised in the memory system [57], diminishing the performance gains of multithreaded execution.

Consider a memory block (*e.g.*, a hash partition). Each thread's operation on it can be divided in three stages: (*a*) the fetching stage, where the block is requested from main memory, (*b*) the processing stage, and (*c*) the locking stage, where the thread interacts with the storage manager to request a new block. Ideally, with N threads, one thread will be fetching and $N-1$ threads will be processing cache-resident blocks. We define the *multithreaded utility ratio* R of Equation 4.1 as the time gained by overlapping operations through having multiple threads operate on different parts of the input. The numerator, C_f, is the cost of fetching a block; the denominator is the sum of the costs of fetching, processing (C_p), and locking (C_l).

$$R = \frac{C_f}{C_f + C_p + C_l} \tag{4.1}$$

Let M be the cost of a memory access. In single-threaded execution main memory is accessed by one thread. For N threads the memory bus is shared; in the worst case an equivalent $(\frac{1}{N})^{\text{th}}$ of the maximum memory throughput is available to each core and, hence, the cost of a memory access reaches MN. Through overlapping operations, captured by the utility ratio R, the effective memory throughput will be greater. We define M', the *effective memory access cost*, as shown in Equation 6.2. If R is less than $\frac{1}{N}$, block operations will overlap so each threads will face negligible contention for accessing memory. Else, the cost will increase depending on the multithreaded utility ratio and will approach MN as $R \rightarrow 1$, *i.e.*, when there is no processing overlap among

P	page size (bytes)
CL	cache line size (bytes)
K	input tuple cardinality
K'	staged tuple cardinality, $0 \leq K' \leq K$
D	distinct value cardinality
T	input tuple size (bytes)
T'	staged tuple size (bytes), $1 \leq T' \leq T$
$L1$	cost for L1-cache access (CPU cycles)
$L2$	cost for L2-cache access (CPU cycles)
M	cost for main memory access (CPU cycles)
OUT	cost for building an output tuple (CPU cycles)
N	number of threads
LK	cost per locking operation (CPU cycles), 0 for $N = 1$
TO	overhead per thread (scheduling, joining *etc*)

Table 4.2: Model parameters

threads.

$$M' = \begin{cases} M & R \leq \frac{1}{N} \\ MNR & R > \frac{1}{N} \end{cases} \tag{4.2}$$

We use this framework to estimate the speedup of multithreaded execution and give formulas for the cost of each algorithm based on a per-memory-access model. We then extract memory utility ratios for each algorithm of Section 4.2 and "plug in" these ratios to the cost formulas. Our goal is not to have an accurate description of execution on a CPU-cycle granularity (which is most likely impossible due to the complexity of modern hardware), but a coarse characterisation of the differences between single- and multithreaded execution. We therefore track the accesses of each algorithm to each level of the memory hierarchy. We do not account for calculations running over registers, as their execution costs are negligible compared to memory operations. We also omit the impact of hardware prefetchers, cache associativity, and non-blocking caches: their effect depends on the design of each CPU and the runtime environment. The parameters of our model are shown in Table 4.2; we assume a two-level deep cache hierarchy.

4.3.1 Sorting

The first step of sorting is to split the input into partitions of B bytes each and sort them using quicksort; the partitions are merged to produce the final sorted output. To generate a single partition to be sorted, the core needs to fetch both the input data and the partition's cache lines. For primary tables we have to account for projections and for filtering the input on (any) selection predicates, as explained in Section 4.2. The size of the input that is used to fill one partition is estimated to $\frac{KT}{K'T'}B$. For each partition, $\left(1+\frac{KT}{K'T'}\right)B$ bytes will be fetched from main memory, costing M cycles for each cache line of CL bytes. The cost of fetching a single input partition is given by Equation 4.3. A generated partition of $\frac{B}{T'}$ tuples is (at least) L2-cache-resident. To apply quicksort, tuples need to be L1-cache-resident. Each tuple needs to be fetched twice from the L2-cache, for reading and writing it. In our implementation, each tuple examination and exchange required roughly four L1-cache accesses, for a total of $\frac{B}{T'}\log\left(\frac{B}{T'}\right)$ operations. The total cost of sorting a partition is shown in Equation 4.4.

$$C_f^{\text{sort}}(B) = \left(1+\frac{KT}{K'T'}\right)\frac{B}{CL}M \tag{4.3}$$

$$C_p^{\text{sort}}(B) = 2\frac{B}{CL}L2+4\frac{B}{T'}\log\left(\frac{B}{T'}\right)L1 \tag{4.4}$$

The utility ratio of the sorting step, $R^{\text{sort}}(B)$, is given by Equation 4.5. We use that to derive the cost of multithreaded execution. The entire relation will produce $\frac{K'T'}{B}$ partitions, so fetching the input and the partitions requires $\frac{KT+K'T'}{CL}$ memory accesses.

This will be divided across N execution threads, with each thread having an effective memory access cost equal to M', as defined by Equation 6.2 when R is substituted for $R^{\text{sort}}(B)$. Since sorting runs inside the cache hierarchy (mainly in the L1-cache), the use of N threads will most likely result in a linear speed-up, so the cost for sorting the input is reduced by a factor of N. Given all these observations, the cost of the sorting step is given by Equation 4.6.

$$R^{\text{sort}}(B) = \frac{C_f^{\text{sort}}(B)}{C_f^{\text{sort}}(B) + C_p^{\text{sort}}(B)} \tag{4.5}$$

$$C^{\text{sort}}(B) = \left(KT + K'T'\right)\frac{M'}{N \cdot CL} + \frac{C_p^{\text{sort}}(B)}{N} \tag{4.6}$$

The second step in sorting a relation is to merge the individually sorted partitions. We maintain a heap of processed tuples across merged partitions, as explained in Section 4.2. The input contains $\frac{K'T'}{B}$ partitions of $\frac{B}{CL}$ cache lines each, so the cost of fetching the sorted partitions during the merging phase is given by Equation 4.7. Each tuple will be fetched twice, since we need to insert its value in the heap, and then output it to the appropriate position in the merged output. However, some algorithms (*e.g.*, merge aggregation) do not require materialising the sorted output, so we include a factor S, set to 2 if we materialise the output, or 1 otherwise. The processing cost is given by Equation 4.8, stemming from heap processing: for each output tuple, the input tuple with the smallest value is retrieved and the heap is re-organised.

$$C_f^{\text{merge}}(B,S) = S\frac{K'T'}{B}M\frac{B}{CL} = SK'T'\frac{M}{CL} \tag{4.7}$$

$$C_p^{\text{merge}}(B) = 2K'\log\left(\frac{K'T'}{B}\right)L1 \tag{4.8}$$

As with partition sorting, the utility ratio of the merging step $R^{\text{merge}}(B,S)$ is given by Equation 4.9. For the total cost of the merging step we generalise the last two equations for N threads, as shown in Equation 4.10. We cater for multiple threads by substituting $R^{\text{merge}}(B,S)$ in Equation 6.2 and dividing Equation 4.7 by the number of threads N; we do the same for the heap processing cost of a partition. The cost of the entire algorithm is then the sum of Equations 4.6 and 4.10.

$$R^{\text{merge}}(B,S) = \frac{C_f^{\text{merge}}(B,S)}{C_f^{\text{merge}}(B,S)+C_p^{\text{merge}}(B)} \quad (4.9)$$

$$C^{\text{merge}}(B,S) = SK'T'\frac{M'}{N\cdot CL}+\frac{C_p^{\text{merge}}(B)}{N} \quad (4.10)$$

4.3.2 Partitioning

Recall from Section 4.2.1 that the general partitioning algorithm is similar to sorting, with two differences: (*a*) quicksort is applied on a per-page granularity, and (*b*) there is a locking overhead when directing tuples to partitions, as multiple threads will be adding pages to them. The cost $C_f^{\text{part}}(P)$ of fetching a page for partitioning is given by Equation 4.11, *i.e.*, similar to Equation 4.3 with B substituted for P, as each partition page is individually sorted. Most likely pages are buffered in the L2-cache, so they need to be fetched to the L1-cache before being sorted, and written back to the L1-cache. The cost of processing a partition page is given by Equation 4.12, *i.e.*, similar to Equation 4.4, but assuming that the page is L1-cache-resident on its second access.

$$C_f^{\text{part}}(P) = \left(1+\frac{KT}{K'T'}\right)\frac{P}{CL}M \quad (4.11)$$

$$C_p^{\text{part}}(P) = \frac{P}{CL}(L2+L1)+4\frac{P}{T'}\log\left(\frac{P}{T'}\right)L1 \quad (4.12)$$

The utility ratio of partitioning, $R^{\text{part}}(P)$, is defined as shown in Equation 4.13, where the denominator includes the locking overhead (since the new page needs to be added to the partition). The total cost of multithreaded partitioning using N threads is given by Equation 4.14, where we use the effective memory access cost (obtained by Equation 6.2 with $R = R^{\text{part}}(P)$). The formula is similar to Equation 4.6 with the difference being the addition of the cost for locking each page of each partition (a total of $\frac{K'T'}{P}$ pages).

$$\begin{aligned} R^{\text{part}}(P) &= \frac{C_f^{\text{part}}(P)}{C_f^{\text{part}}(P) + C_p^{\text{part}}(P) + LK} \\ C^{\text{part}}(P) &= \left(KT + K'T'\right)\frac{M'}{N \cdot CL} + \frac{C_p^{\text{part}}(P)}{N} + \frac{K'T'}{P}LK \end{aligned} \tag{4.13}$$

Locking is used to synchronise the interaction with the storage manager. Assuming the partitioning fanout is F, each thread will contend with the other $N-1$ threads; the probability of any thread requesting access to a partition is $\frac{1}{F}$. The probability of contention then depends on the factor $\frac{N!}{F^N}$ (*i.e.*, all permutations of threads into the probability of all threads accessing the same partition); that is very small. It also depends on the ratio of the duration of the lock to the duration of page processing, which also includes data fetching and sorting ($\frac{C_l}{C_f+C_p+C_l}$). We therefore expect that threads rarely need to wait for a lock to be released.

The partition pages are individually sorted, so we need to merge them in a separate step, similarly to general sorting. The difference lies in the use of the merge buffer that replaces memory accesses with accesses to the L2-cache. The fetching and processing costs are therefore modified as shown in Equations 4.14 and 4.15. Recall that if the size of the L2-cache is $|L2|$, the partition size will roughly be $\frac{|L2|}{2N}$.

$$C_f^{\text{merge}}(P,S,M) = K'T'\frac{M}{CL} + SK'T'\frac{L2}{CL} \tag{4.14}$$

$$C_p^{\text{merge}}(P) = 2K'\log\left(\frac{|L2|}{2NP}\right)L1 \tag{4.15}$$

In Equation 4.14, S is 0 when the output is processed on-the-fly (*e.g.*, in aggregation), or 2 when the output is saved to the merge buffer. The modified utility ratio and the merge cost are shown in Equations 4.16 and 4.17. The total cost for partitioning is the sum of Equations 4.14 and 4.17; M' is given by Equation 6.2 after setting $R = R^{\text{merge}}(P,S)$.

$$R^{\text{merge}}(P,S) = \frac{C_f^{\text{merge}}(P,S,M)}{C_f^{\text{merge}}(P,S,M) + C_p^{\text{merge}}(P)} \tag{4.16}$$

$$C^{\text{merge}}(P,S) = \frac{C_f^{\text{merge}}(P,S,M') + C_p^{\text{merge}}(P)}{N} \tag{4.17}$$

4.3.3 Join evaluation

All join algorithms run exclusively inside the L1-cache and build on the staging primitives. When joining there is no need to synchronise threads, as they operate over disjoint inputs (see also Section 4.2.2). The difference between the algorithms lies in where they “read” their data from. For sort-merge join each partition is read from main memory, while for hash join the input is buffered in the L2-cache. Thus, we only need to assess the cost of fetching the input and generating the output. Assuming two inputs A and B, and N threads, the cost of processing the entire input will be given by Equation 4.18, where $\sigma_{\bowtie}$ is the selectivity factor of the join predicate. For sort-merge join the input tables are fetched from main memory, so the cost will be given by Equation 4.19. For hash join the equivalent cost of fetching from the L2-cache is given by Equation 4.20. To those costs we need to add the thread scheduling overhead, equal to $N \cdot TO$ in all cases.

$$C_p^{\text{join}} = \frac{K'_A K'_B \sigma_{\bowtie}}{N} OUT \tag{4.18}$$

$$C_f^{\text{merge-join}} = \left(K'_A T'_A + K'_B T'_B\right) \frac{M}{N \cdot CL} \tag{4.19}$$

$$C_f^{\text{partition-join}} = \left(K'_A T'_A + K'_B T'_B\right) \frac{L2}{N \cdot CL} \tag{4.20}$$

The total cost of sort-merge join will be equal to the cost of sorting both inputs (Equations 4.6 and 4.10 with S set to 2), plus fetching the blocks of both inputs from main memory (Equation 4.19), plus the cost of generating the output (Equation 4.18), plus the cost of thread scheduling ($N \cdot TO$). Similarly, one can extract the cost of hash join evaluation: it is equal to the cost of partitioning the input (Equation 4.14 and

Equation 4.17 with S set to 3 to include each input's contribution to Equation 4.20 as well), plus the output generation cost (Equation 4.18), plus the thread overhead cost.

4.3.4 Aggregation

Recall from Section 4.2.3 that aggregation evaluates aggregates on-the-fly, without restructuring the input table. For merge and hash aggregation this means that we do not materialise the output of the merging phase; rather, we use it directly to update the aggregate values. The aggregation cost is given by the data staging cost equations: we set S to 1 for merge aggregation and to 0 for hash aggregation. We also include the scheduling cost $N \cdot TO$ for multithreaded execution.

Map aggregation makes a single pass over the input with no intermediate staging. Memory accesses overlap with lookups on the mapping directories, as the latter are cache-resident. Assuming G grouping attributes and A aggregation functions, as well as binary search for mapping directory lookups, input fetching and processing are given by Equations 4.21 and 4.22 respectively; D_i is the distinct value cardinality of group i.

$$C_f^{\text{map}} = \frac{KT}{CL} M \tag{4.21}$$

$$C_p^{\text{map}} = \left(\sum_{i=0}^{G} \left(\log\left(D_i\right) L1\right) + A \cdot L2 \right) K' \tag{4.22}$$

The first term in Equation 4.22 is the cost of binary search in each directory; the second term is the cost of updating the aggregation arrays. The assumption is that the mapping directories fit in the L1-cache, while the (possibly) larger aggregation arrays are evicted to the L2-cache. We can now extract the map aggregation cost as shown in Equation 4.24, where M' is given by using the utility ratio of Equation 4.23.

$$R^{\text{map}} = \frac{C_f^{\text{map}}}{C_f^{\text{map}} + C_p^{\text{map}}} \tag{4.23}$$

$$C^{\text{map}} = \frac{KT}{N \cdot CL} M' + \frac{C_p^{\text{map}}}{N} \tag{4.24}$$

4.4 Experimental study

To verify the efficiency of our proposals and the correctness of the analytical model, we implemented our algorithms in C and conducted an extensive experimental study. The hardware platform we used was a Dell Precision T5400 workstation, with an Intel Xeon E5420 quad-core processor, clocked at 2.5GHz with 4GB of physical memory running GNU/Linux (64-bit version, kernel 2.6.26). The C code was compiled with the GNU `gcc` compiler (version 4.3.2) using the `-O2` compilation flag. We used the `pthread` thread library. Details about the testbed are shown in Table 4.3. The cache latencies were measured with the RightMark Memory Analyser [65].

We used tables of various schemata and cardinalities and stored them using NSM. Primary tables were cached in the buffer pool of a typical storage manager. All intermediate results (*e.g.*, partitions) were saved as temporary tables, also controlled by the storage manager. To simplify the analysis of an already complex system we used uniform attribute distributions. Our framework uses techniques that have been shown to efficiently tackle data skew, as presented in Section 4.2.

We integrated the proposed framework for multithreading to HIQUE. The code generator produced multithreaded C code for each benchmark query, so the instruction-level overhead was reduced. This was beneficial to single-threaded performance, as multithreading can exploit the instruction caching and issuing mechanisms of multiple cores. We expect iterator-based implementations of our algorithms (*e.g.*, based on the *exchange* operator of [34]) to result in higher speedups but slower response times. We ran each query ten times in isolation and report the average response times; the devi-

System	Dell Precision T5400
Processor	Intel Xeon E5420
Number of cores	4
Frequency	2.5GHz
Cache line size	64B
I1-cache	32KB $\times$4
D1-cache	32KB $\times$4
L2-cache	6MB $\times$2
L1-cache access latency	3 cycles
L1-cache miss latency (sequential)	9 cycles
L1-cache miss latency (random)	14 cycles
L2-cache miss latency (sequential)	48 cycles
L2-cache miss latency (random)	85-250 cycles
RAM type	4x1GB Fully Buffered DIMM DDR2 667MHz

Table 4.3: Testbed specifications

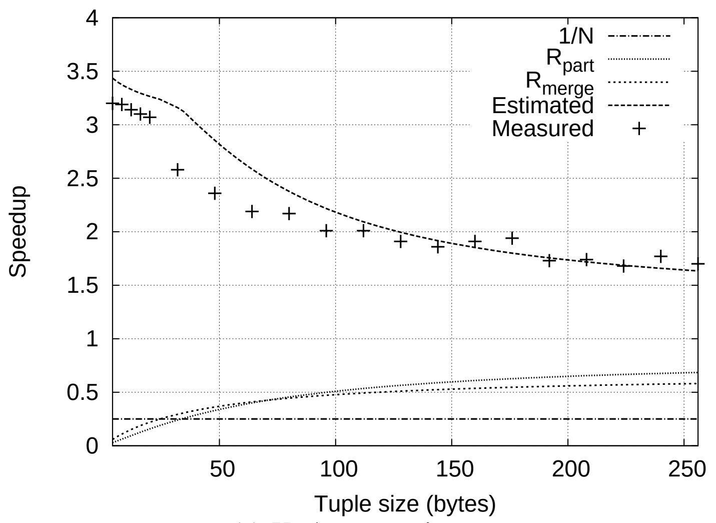
ation was less than 3% in all cases. We also report the speedup when moving from single-threaded to multithreaded execution.

Measured speedups were compared with the ones estimated by the analytical model. To apply the model, we set N to 4, as our reference CPU supports one thread per core, $L1$ to 3, $L2$ to 14 and M to 100, as accesses are both sequential and random. We calibrated the locking cost LK to $5M$ and TO to 2.5% of total execution time. We set OUT to zero and did not generate results (unless explicitly stated), to isolate the multithreaded performance of the algorithms; result generation runs inside the L1-cache for each thread and thus inflates scalability.

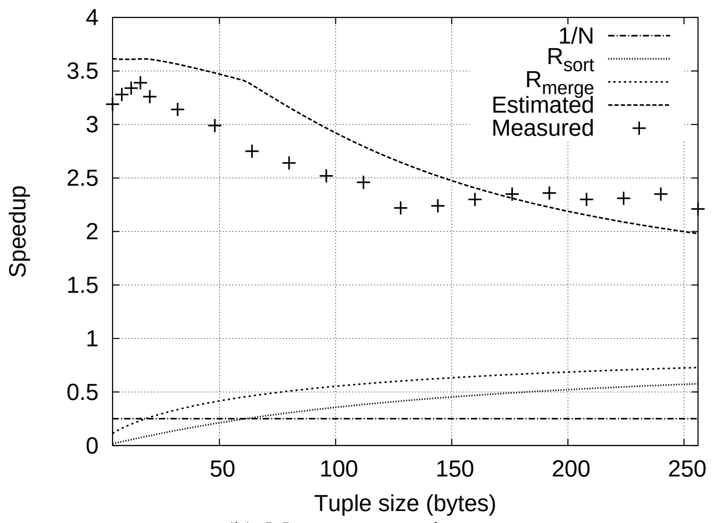
4.4.1 Aggregation

We measured the impact of input tuple size by using a table of 1,000,000 tuples ($K = K'$) and varying the tuple size between 4 and 256 bytes ($T = T' \in [4, 256]$), using one grouping attribute with 1,000 distinct values (D). The estimated and measured costs for merge, hash, and map aggregation, as well as their comparative performance when using four threads, are shown in Figure 4.3. When R becomes greater than $\frac{1}{N}$ we expect the effective memory access cost M' to start increasing. This is verified experimentally, as the slope significantly grows when R passes this threshold. The estimate for hash aggregation is more accurate than that for merge aggregation. The fluctuation in the latter is due to cache line alignment effects, which are not included in our model. In terms of algorithm performance, the measured speedup is over 3 for small tuple sizes. It degrades for wider tuples, as the cores will spend more time fetching data from memory. This is more intensive in hash than merge aggregation, as the computational load for sorting and merging larger blocks keeps the cores busy to a higher extent. For map aggregation, the mapping directory has enough entries to make the lookup cost comparable to the cost of fetching small tuples. As the tuple size grows the fetching cost scales and dominates, resulting in poorer performance. The deviation in Figure 4.3(c) for small tuple sizes is due to overestimating the cost of updating the aggregation arrays: it varied between $L1$ and $L2$, but is set to $L2$ in Equation 4.24. As shown in Figure 4.4, merge and hash aggregation have comparable performance, as they incur a similar number of accesses to main memory. Map aggregation needs no input staging and is thus faster and less sensitive to changes of the tuple size, for the given (small) number of values of the grouping attribute.

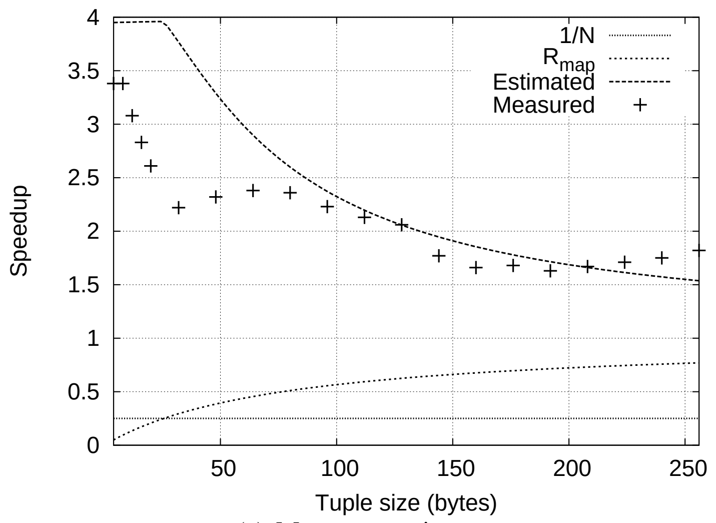
We then measured the impact of input cardinality after applying selections and projections. We used a table of 10,000,000 (K) tuples of 72 bytes (T) each and varied the selectivity between 0.1 and 1; each tuple after staging was 20 bytes (T'); D was set to 1,000 again. The results are shown in Figure 4.5. The performance is accurately modelled, with estimated and measured curves for all aggregation algorithms being



(a) Hash aggregation



(b) Merge aggregation



(c) Map aggregation

Figure 4.3: Impact of tuple size on aggregation - scalability

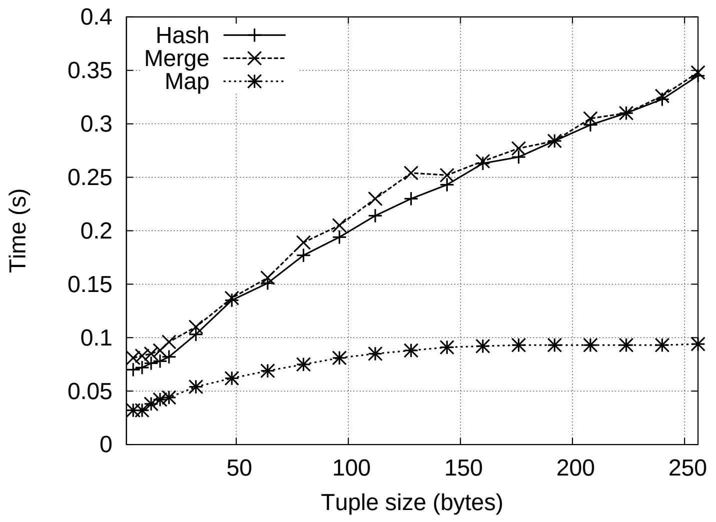
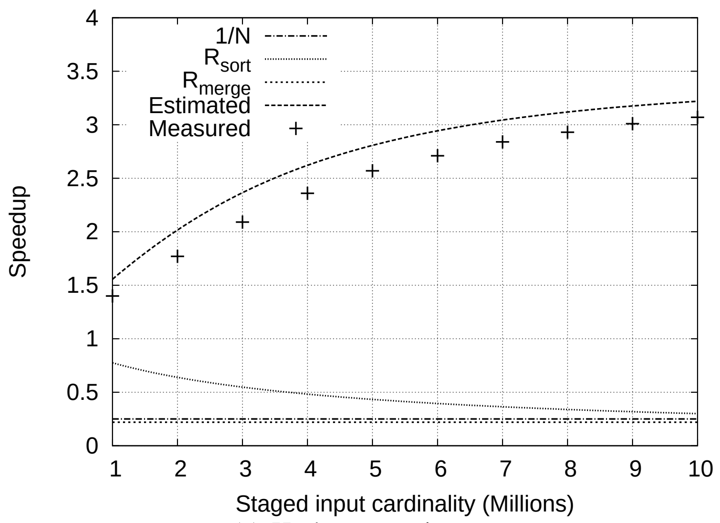


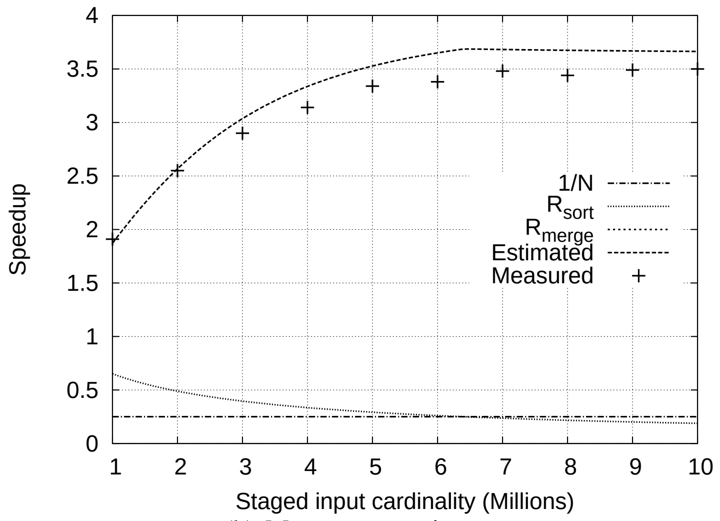
Figure 4.4: Impact of tuple size on aggregation - multithreaded performance

close and following the same trends. For a small selectivity, the cost of fetching the primary table is higher than sorting the filtered data. As selectivity grows the speedup increases and converges to a maximum value, reached when R is less than $\frac{1}{N}$. Note that the merge-based implementation gives higher speedups, as it better exploits the computational power of multiple cores. As for comparative multithreaded performance (Figure 4.6), hash aggregation outperforms merge aggregation by a factor increasing with growing selectivity. Map aggregation widely outperforms the other algorithms and is less sensitive to selectivity as it does not build intermediate partitions.

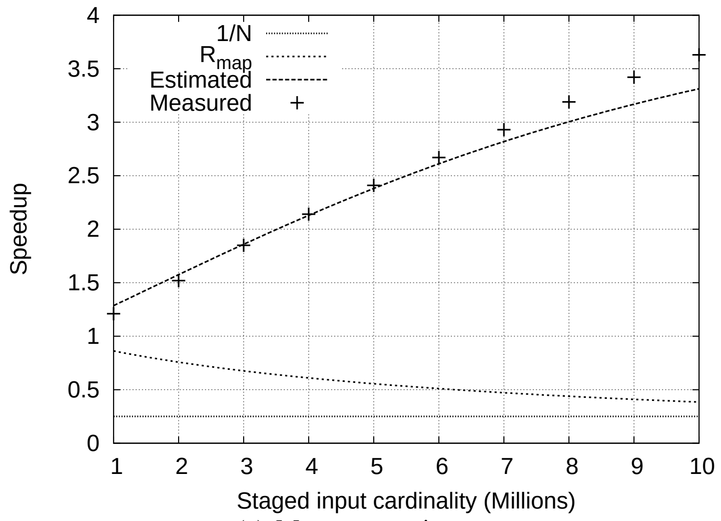
The number of distinct values of the grouping attribute(s) has a detrimental effect on the performance of map aggregation, as it affects the size of the directories and the aggregation arrays. As the grouping cardinality increases, the auxiliary data structures are evicted to lower cache levels. This penalises performance, as there is a significant increase in cache misses, and scalability, as all threads compete for accessing memory to a greater extent. This is shown in Figure 4.7 for an aggregation query on 10,000,000 tuples of 72 bytes each, using one grouping attribute of varying cardinality D and four `sum` functions. In the first two cases there is no result generation; in the third case we show the impact of result generation on scalability. Merge and hash aggregation are



(a) Hash aggregation



(b) Merge aggregation



(c) Map aggregation

Figure 4.5: Impact of selectivity on aggregation - scalability

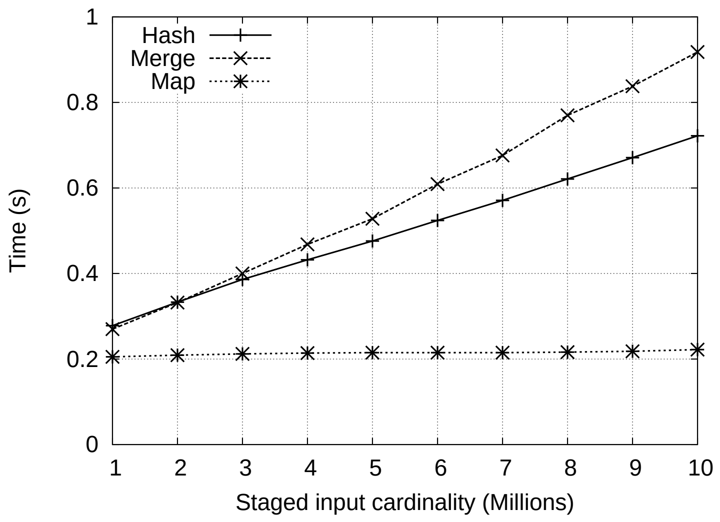
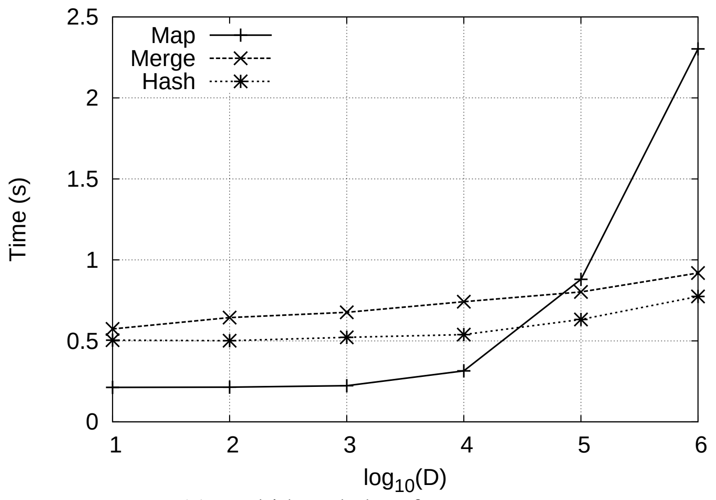


Figure 4.6: Impact of selectivity on aggregation - multithreaded performance

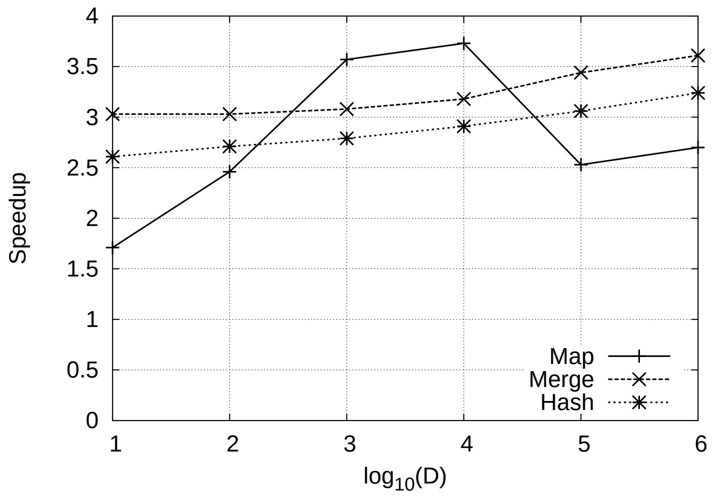
moderately affected by the cardinality of the grouping attribute, their difference being the number of iterations during quicksort runs. Map aggregation is 2.5 times faster for small cardinalities but its performance degrades fast, indicating the inflated cost for accesses to the L2-cache and the main memory. In terms of scalability (Figure 4.7(b)), hash and memory aggregation exhibit high speedups, increasing with cardinality. Map aggregation has a low speedup for small cardinalities, as the directory lookup cost is too small to hide memory latencies. Then, speedups increase with cardinality and start dropping again, as the auxiliary data structures are evicted to the L2-cache or outside it. Output generation provides sufficient computational load to mask memory accesses (Figure 4.7(c)), with all algorithms exhibiting speedups over 3 for considerable result sizes.

4.4.2 Join evaluation

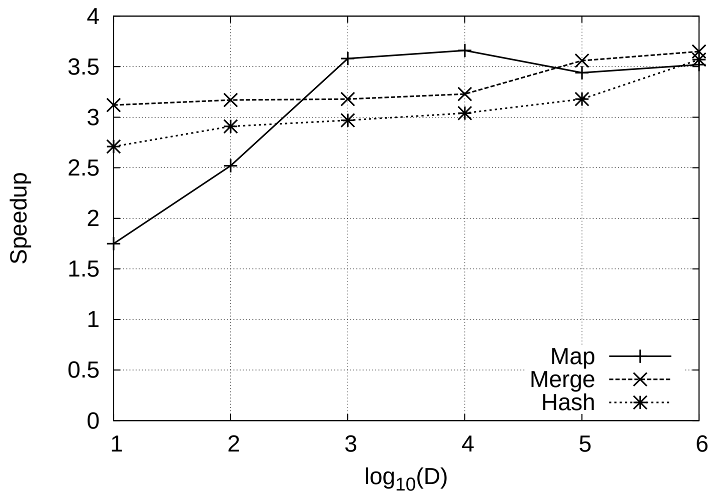
We next studied multithreaded join evaluation for varying input tuple size, input cardinality, and join selectivity. We joined two tables of 1,000,000 tuples each. The outer table's tuples were 72 bytes long; the tuple size after staging was 20 bytes. The inner



(a) Multithreaded performance

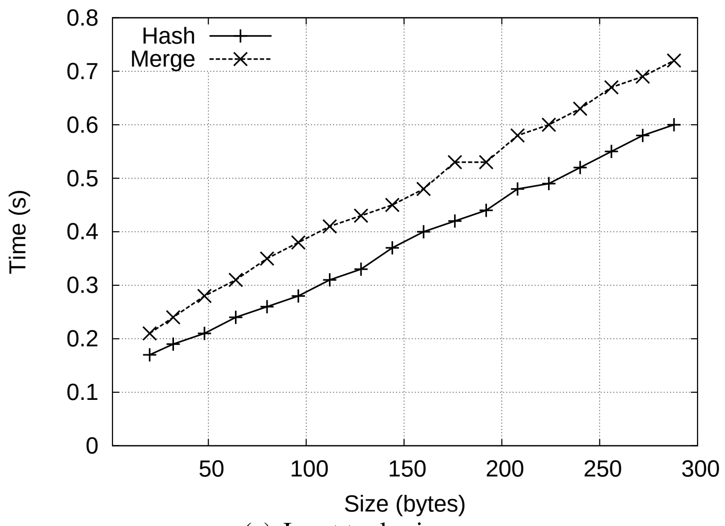


(b) Measured speedup

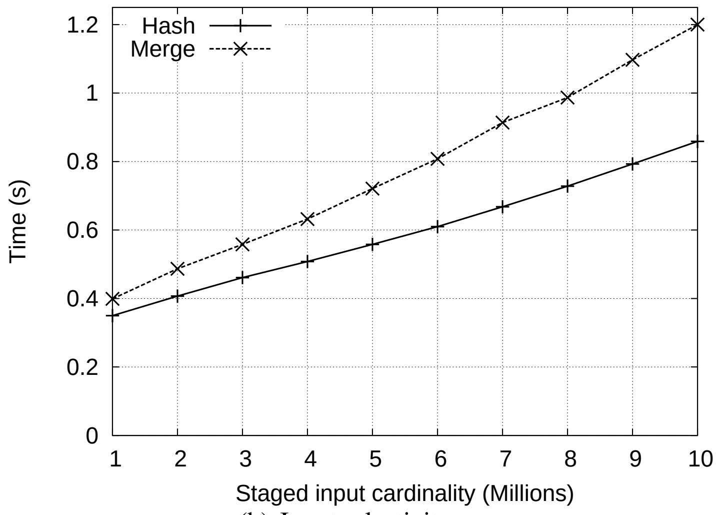


(c) Speedup with output generation

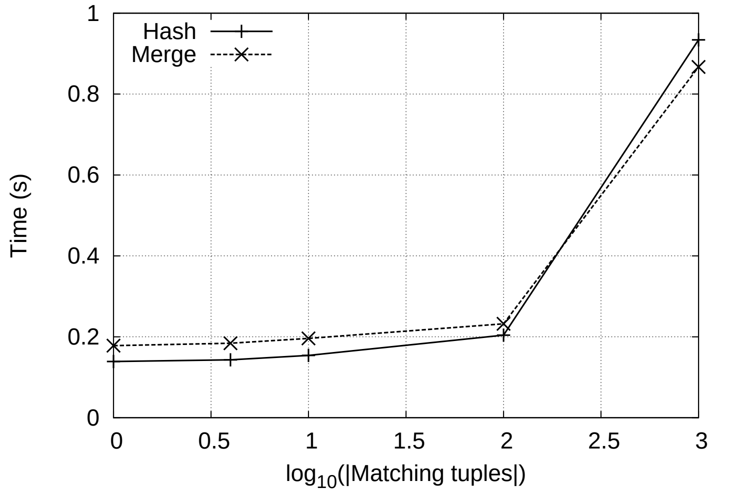
Figure 4.7: Impact of group cardinality



(a) Input tuple size

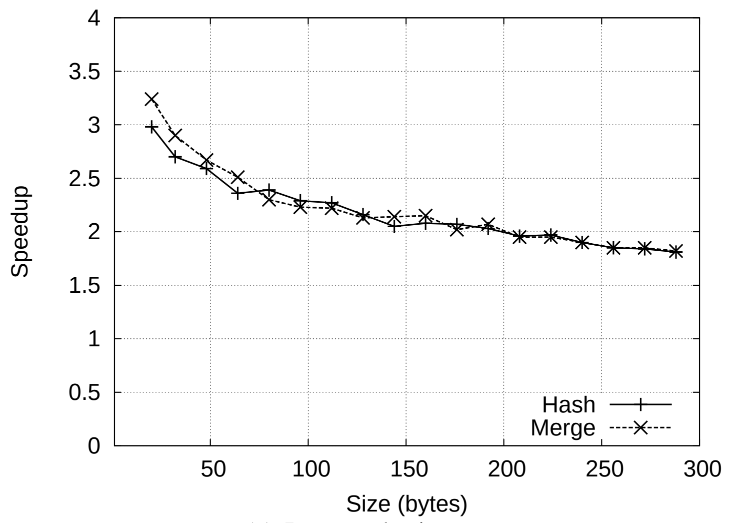


(b) Input selectivity

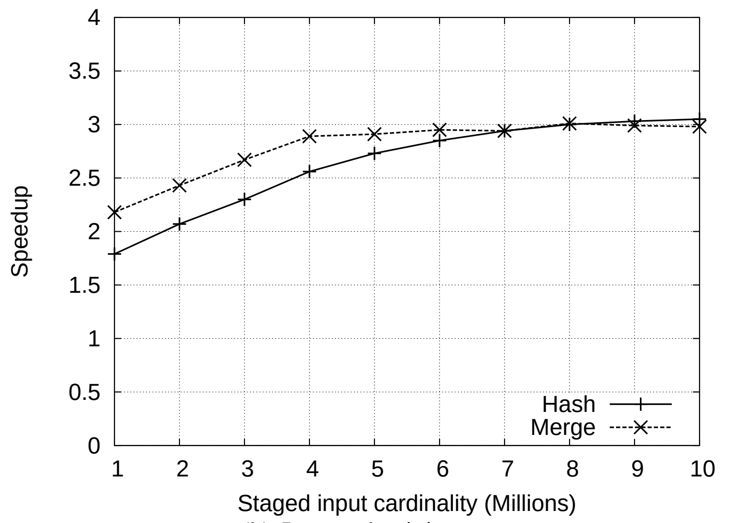


(c) Join predicate selectivity

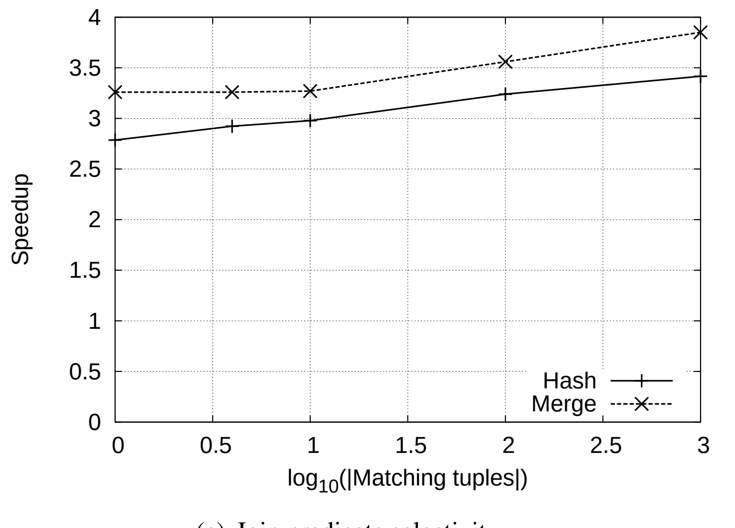
Figure 4.8: Multithreaded performance of join evaluation



(a) Input tuple size



(b) Input selectivity



(c) Join predicate selectivity

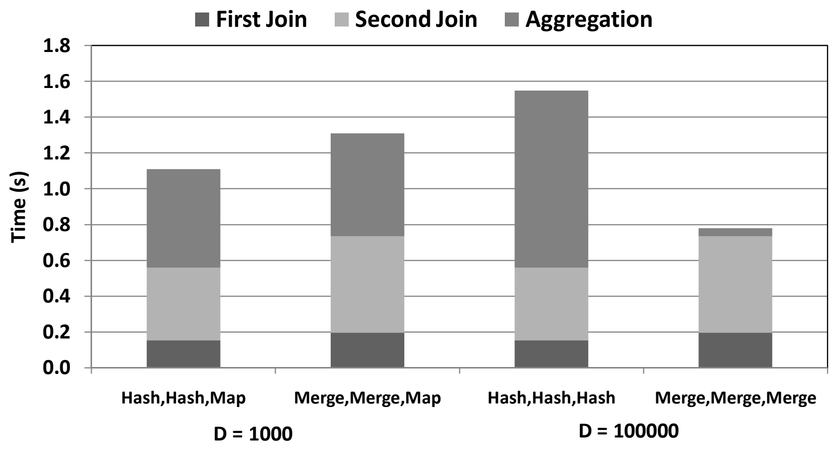
Figure 4.9: Measured speedup for join evaluation

table's tuple size varied between 20 and 300 bytes. Each outer tuple matched with 10 inner tuples. The results shown in Figures 4.8(a) and 4.9(a) exhibit trends similar to the ones of aggregation (Figure 4.3): input staging accounted for 90% of execution time (omitting result generation) and is the same process for both aggregation and join evaluation. Hash join performs better; the use of merge buffers increases cache locality and reduces the cost of memory operations. Still, merge join results in higher speedups by exploiting the higher computational cost of sorting larger blocks.

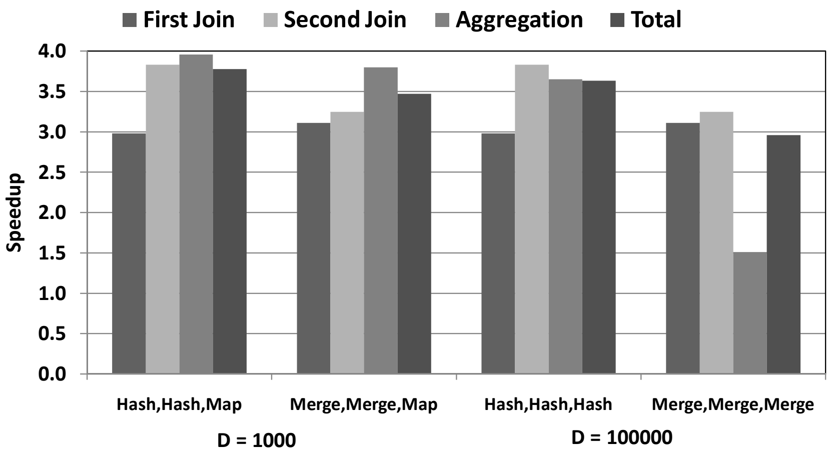
For cardinality experiments we used two tables with tuple sizes of 72 bytes, reduced to 20 bytes after staging; each outer tuple matched with 10 inner ones. The outer table's cardinality was 1,000,000 and the inner's was 10,000,000, but we filtered the inner table with a predicate of selectivity ranging between 0.1 and 1. The results of Figures 4.8(b) and 4.9(b) are similar to those of Figure 4.5, with speedups increasing and converging to a maximum value. In terms of join predicate selectivity, we joined two tables of 1,000,000 tuples, 72 bytes each, but staged to 20 bytes. We varied the number of matching inner tuples per outer tuple to 1, 4, 10, 100, and 1,000. As join selectivity grows, the speedup is close to linear for both algorithms, as shown in Figures 4.8(c) and 4.9(c). This is due to join predicate evaluation effectively "backtracking" between multiple matches. Processing runs inside the L1-cache, reducing the frequency of memory accesses and resulting in high speedups.

4.4.3 Pipelined operators

We now move on to a query combining two joins and an aggregation. We used three tables with 1,000,000 tuples of 72 bytes each. In the first join, each outer tuple matched with 4 inner ones; in the second join the number of matching inner tuples was 10. The two joins produce 4,000,000 and 40,000,000 tuples respectively. We used both merge and hash join. The result was `sum`-aggregated over one grouping attribute with either 1,000 or 100,000 distinct values. In the first case we used map aggregation. In the second case, the grouping attribute was the same as the join attribute of the second



(a) Multithreaded performance



(b) Measured speedup

Figure 4.10: Multiple operators

join, to measure the impact of sorted runs. The results are shown in Figure 4.10; the labels indicate the algorithms used for each operator.

Hash join is faster than merge join, verifying once again that the use of an L2-cache buffer for merging pays back. For aggregation, when the number of values for the grouping attribute is 1,000, the use of map aggregation is very efficient: it needs 0.55s for 40M tuples, resulting in a throughput of 72,600,000 tuples/s. In terms of scalability, the reduction in tuple size allows all operators apart from the first to work on small tuples and, hence, they do not fetch data not needed for processing. The observed speedups are over 3 and, for hash join and map aggregation, close to linear.

When the number of groups increases to 100,000, hash and merge aggregation become more efficient as map aggregation exhibits excessive cache misses. We use either all-hash or all-merge algorithms. The cost of hash aggregation is twice that of map aggregation in the previous case (*i.e.*, when $D = 1{,}000$). However, since the output of the second join is already sorted on the grouping attribute, merge aggregation does not need intermediate partitions, but is evaluated in a single pass of the join result. A direct comparison of map and merge aggregation shows that the latter needs only a small portion of the time needed by the former, as there are no directory lookups and updates of aggregate arrays. However, the speedup of merge aggregation is limited as there is no computational load to effectively mask the cost of memory accesses.

4.5 Further reading

Simultaneous multithreading (SMT), a form of TLP, was explored in [77]: a helper thread was used to aggressively prefetch data to be used by the main thread. This technique is not applicable in multicores with no in-core support for TLP, as the helper thread will fetch data to a different L1-cache than the one used by the main thread. The authors of [19] examined inter-operator communication and proposed using chunks of the output as buffers for each thread. We use a separate output buffer per thread to avoid synchronisation and a similar approach for partitioning (see also [18]), since each thread has exclusive access to one partition page. As we sort pages during partitioning, the processing time per page increases and thread contention for locking is minimised.

In [17], the authors tested and modelled the use of private and/or shared hash tables for aggregation on CMPs. Their approach is tailored to processors supporting multiple (four for the employed CPU) threads inside each core; it is not clear how it can efficiently be ported to architectures with no in-core support for TLP. Aggregation performance in [17] reached 150Mt(uples)/s against 72.6Mt/s for us (see Section 4.4.3), using arrays of two-integer records as input. A per-core reduction gives 18.75Mt/s for [17] over 18.15Mt/s for us; a per-thread one gives 18.15Mt/s for our approach over 4.69Mt/s for [17]. Still, the testbed used in [17] is entirely different than ours, so comparisons cannot be straightforward. The combination of SIMD instructions with multithreading on multicores was studied in the context of mergesort [16] and join evaluation [47]. This approach proves highly efficient when processing vertically partitioned data, but it cannot be directly applied to query engines processing NSM-based pages. Our framework is independent of the storage layout.

In [54], the authors gave an analytical model for single-threaded main-memory query execution. The model captured the cost of stalls, *e.g.*, cache and TLB misses, according to the access pattern. In our model, we do not distinguish between sequential and random access patterns but we account for accesses to the L1-cache, as CPUs do not have enough memory ports to serve successive read and write operations. Finally, [44] tackled work sharing in CMPs and modelled the performance of concurrently processed, staged queries, [62] investigated scheduling of multiple queries for scan sharing, and [51] suggested the use of page colouring to prevent cache thrashing when concurrently executing multiple queries. These are complementary to our work; we focus on intra-operator parallelism and model the contention for shared hardware resources.

Chapter 5

Scheduling threads for intra-query parallelism

5.1 Introduction

Multicore processors have now become the *de facto* standard in CPU design. The cores share main memory and (possibly) the lowest levels of the cache hierarchy, effectively turning the machine into a shared-memory parallel system. Moreover, the multicore chip supports thread parallelism at the hardware level, either by using a number of threads equal to the number of cores, or by providing hardware support for the execution of multiple threads inside each core. At any rate, there is a budget of available threads to be used for parallel processing. In Chapter 4 we have described how a query engine can be efficiently extended for intra-query parallelism (*i.e.*, many threads are used to evaluate the same query). This setup presents an interesting scheduling problem: what is the best possible allocation of threads to query plan operators, and what parts of the plan should be executed in parallel? It is not surprising that this problem is intractable and any meaningful solutions should leverage heuristics. In this chapter we present our approach to thread scheduling for this processing model. Based on our proposed analytical model for estimating scalability, as presented in Chapter 4, we develop a family of algorithms that allow the optimiser to allocate threads to operators and schedule these threads in an efficient way. Our results show that the proposed algorithms generate parallel processing schedules of high quality and significantly improve upon the existing scheduling alternatives. At the same time, the optimisation overhead is kept at a minimum.

Multicore systems can be viewed as a shared-memory execution paradigm, with multiple cores accessing the same main memory. The main difference is that, apart from main memory, cores often share the lowest level of the cache hierarchy. This design allows threads to efficiently share data and synchronise their execution, but may introduce contention for the caches and the memory bus (see also [17, 57] and Section 4.3), leading to the appearance of resource dependencies. These dependencies restrict the degree of parallelism exploited by multithreaded query execution and lead to sub-optimal scalability as the number of threads increases.

Scheduling threads for multithreaded query execution should account for the unique characteristics of the processing platform, as naïve approaches will fail to exploit the processing power of the multiple cores. Existing work on parallel systems argues that a scheduling option is to allocate a number of threads comparable to the number of processing tasks, irrespective of the hardware restrictions on how many threads can be efficiently supported. This would result in sub-optimal use of the processing resources, due to the overhead of synchronising work across different threads (using a producer-consumer approach) and scheduling their execution [34]. Another alternative is to sequentially execute query operators using all available threads, thus avoiding synchronisation across operators. However, this is adequate only in the ideal case where performance scales linearly with the number of threads: using N threads would increase performance N times. In practice, the more threads are used, the lower is the gain from multithreading, so it may be more efficient to distribute threads to a few operators and execute them concurrently than run these operators sequentially using all available threads. It is therefore important to identify the number of threads that should be allocated to each query operator; it is equally important to schedule the operator execution in a way that reduces synchronisation overhead and maximises the performance improvement from multithreading.

The number of possibilities the optimiser needs to consider when generating an execution schedule scales quite fast as the number of operators and the budget of available threads grow. Consider, for instance, a simple execution tree of three operators. Given a number of threads to be used, the optimiser needs to allocate threads to the three operators. It can either execute the operators sequentially and thus allocate all threads successively to each operator; or it can parallelise execution by allocating a different number of threads to each operator without exceeding the thread budget; or it can allocate threads to a pair of operators to be executed in parallel, and, once the pair completes its computation, allocate all threads to the third operator. The number of combinations grows exponentially with the number of threads. This is aggravated as the number of operators in the execution tree grows and as the tree becomes bushier: the portions of the execution plan that can be executed in parallel increase, as the likelihood of data dependencies decreases. Soon enough, the problem becomes intractable. Moreover, optimisation approaches like dynamic programming prove sub-optimal in parallel query processing [28]. It is therefore imperative to use heuristics to efficiently explore the search space of possible schedules.

In what follows, we present our approach to parallel query operator scheduling on multicore processors. We use estimations for the expected execution cost of each query operation, according to the number of allocated threads, to give our heuristics-based algorithms that generate schedules of high quality. The key intuition behind the proposed algorithms is to specify groups of tasks that, when scheduled together and following a specific thread distribution, (*a*) minimise delays due to data and resource dependencies and (*b*) extract the highest degree of parallelism for the given group of tasks and their scalability potential.

The contributions of our work are:

- We model the problem of intra-query parallelism (using both inter- and intra-operator parallelisation) when scheduling the operators of a plan in the context of multithreaded query evaluation.
- As the scheduling problem is intractable, we present a family of heuristics-based scheduling algorithms that operate on an already generated execution tree and allocate threads to operators to minimise the total execution time.
- We implement the proposed algorithms and experiment with their performance. The results show that our algorithms produce schedules of high quality and closely approximate optimum schedules when such a comparison is possible. In all other cases, the algorithms improve naïve schedules by a factor of 50% on average for substantially complex query plans, without penalising the running time of the optimiser.
- We identify cases where each algorithm is likely to produce schedules of higher quality and give a set of guidelines to anticipate which algorithm better suits a specific type of query.

Note that, although our modelling methodology targets multicore architectures specifically, the proposed heuristics can be applied in any parallel system where one can have reliable estimations about the execution cost and speedup. This demonstrates the wide applicability and the generality of our proposals.

The rest of this chapter is organised as follows. In Section 5.2 we present our modelling methodology for thread scheduling in the context of multithreaded query evaluation. Based on our modelling, we present a family of heuristics-based scheduling algorithms in Section 5.3. We implement our proposals and experiment with the quality of plans they produce in Section 5.4, and we discuss our most important findings in Section 5.5. Finally, we present work for further reading in parallel query execution in Section 5.6.

5.2 Formulation of scheduling

A shared-memory parallel system. We consider a CPU with many cores. The cores share main memory and possibly the lowest level of the cache hierarchy. The design is assumed *symmetric*, *i.e.*, all cores have equivalent resources and can access any level of the memory hierarchy at the same cost, in the sense that a L1-cache miss will have the same cost for all cores. The cores have private caches for the highest levels of the cache hierarchy (the L1- and possibly the L2-cache, depending on the design of each processor) and may share the lowest cache level (the L2- or the L3-cache), so data accesses are faster for the cores that can fully buffer their dataset in their private caches.

To keep our approach generic and avoid tying it to specific architectural setups, we abstract the processor design and assume the CPU provides hardware support for multithreading. This may be in the form of [41] (*a*) in-core support for multiple threads (*e.g.*, Hyperthreading in Intel Pentium 4), or (*b*) multiple cores with each one running one thread (*e.g.*, Intel Core 2 Duo, AMD Phenom), or (*c*) multiple cores with each one running many threads (*e.g.*, Intel Core i7, Sun UltraSPARC, IBM Power 6). The execution characteristics of the query engine with regard to the processor architecture are captured by the cost model; we will return to this issue at the end of this section.

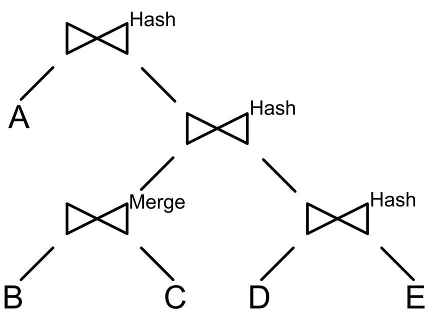
To aid the presentation of our modelling approach, we assume a *two-phase* optimisation process, which is the norm for parallel systems [40]. The first phase is *Join Ordering and Query Rewriting* (JOQR), similar to conventional query optimisation, and produces an annotated bushy execution plan. We do not address how the execution plan is generated; it is considered as given. We focus on the second phase that addresses the scheduling problem, *i.e.*, on which core(s) and when each query operator will execute. We will use the sample execution plan of Figure 5.1(a) to describe the scheduling process. This bushy plan comprises one merge join and three hash join operators.

Forms of parallelism. We assume that the query engine supports both intra- and inter-operator parallelism for each query. Intra-operator parallelism can be based on (*a*) *partitioning*, when the input of an operator is split in disjoint partitions that are scheduled to different cores and processed in parallel, or (*b*) *pipelining*, when multiple operators are pipelined and executed in unison. The latter is applicable with right-deep trees of hash join operators [66], where the build phases for all joins precede the pipelined evaluation of all the probing phases. Conversely, inter-operator, or *independent*, parallelism means that different operators are scheduled to execute concurrently.

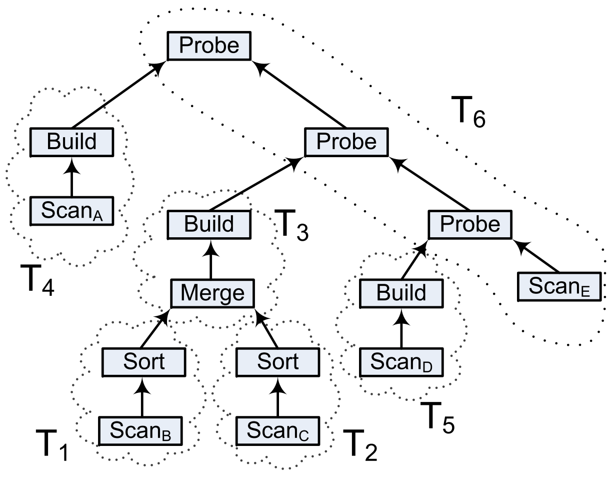
Operator tree generation. The first step in the scheduling process is to *macro-expand* [28] the execution plan. Depending on the evaluation algorithm, each operator node of the execution plan is replaced with the tree of sub-operators that constitute it. In Figure 5.1(b) we show the operator tree for the execution plan of Figure 5.1(a). Merge join is expanded to two sorting operations followed by a join, while hash join [48] is expanded to a build operation for the outer input followed by a probe operation for the inner one.

Query tasks. The next step is to specify query tasks [31], *i.e.*, groups of operators that can be scheduled to run in parallel. Results across groups can be pipelined; operators such as sorting need to block until their input is available. The goal is to reduce the number of tasks to schedule without inducing imbalances to the total computational load per thread.

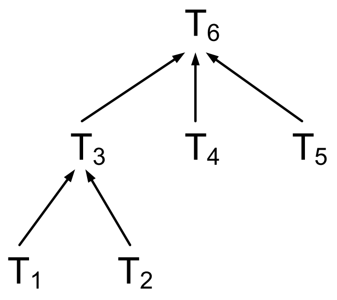
We focus on join evaluation, as aggregation and sorting of the *final* result can always use all available threads. For merge join, each input can be sorted independently, so we assign one task to each sort operator (*e.g.*, tasks T_1 and T_2 in Figure 5.1(b)). Sorting can use range partitioning to build disjoint partitions on the join attribute, that will then be joined independently. Note that the join cannot be evaluated unless both inputs have been sorted. If the output of merge join is used as input to a following join operator, there is a stream of (already partitioned) data between the two join operators. We assume that each thread also prepares (*e.g.*, sorts or builds a hash table for) the join output for the following operator. That way, the merge and sort/build tasks are scheduled together (*e.g.*, task T_3 in Figure 5.1(b)) and executed without any delays



(a) Execution plan



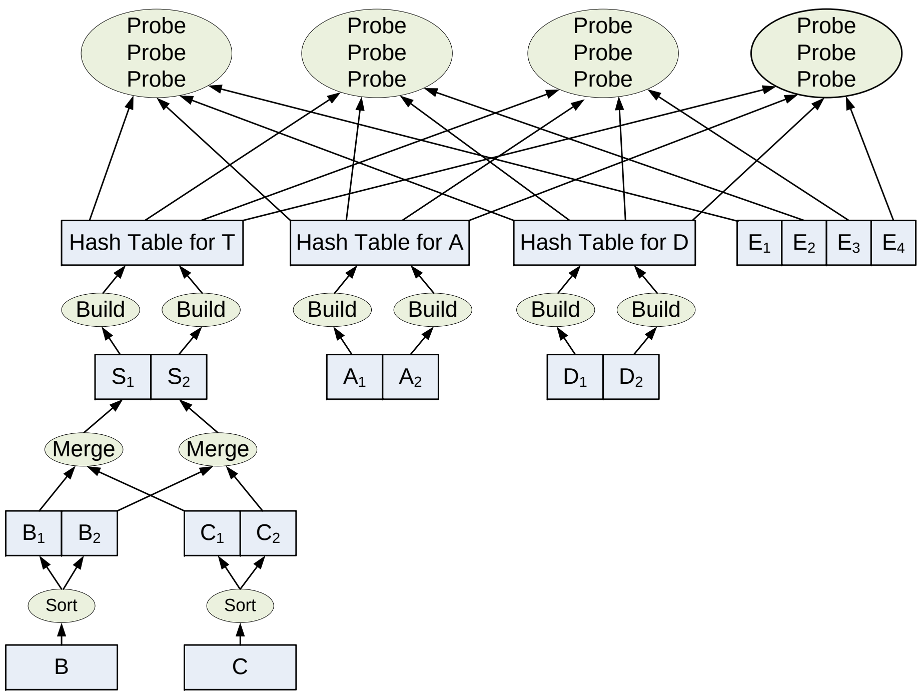
(b) Operator tree



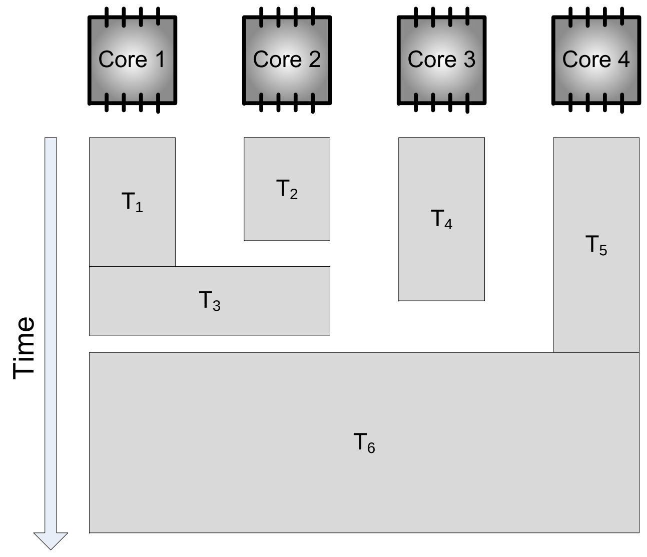
(c) Query task tree

Figure 5.1: Stages of query scheduling (a)

or synchronisation overhead. We show how partitioning works in Figure 5.2(a). The sorted output of tables B and C comprises two partitions; each pair (B_i, C_i) is then joined independently by a separate thread. Each output S_i is used to build a hash table



(a) Partitioned dataflow



(b) Scheduled plan

Figure 5.2: Stages of query scheduling (b)

for the following hash join, so each thread will create the entries of the hash table for its own output. Depending on the query engine, this can be done through pipelining, *i.e.*, each joined tuple is directly inserted to the hash table; or through blocking, by first generating all the join output tuples and then building the hash tables.

For hash join, the native support for pipelining enables multiple operators to be scheduled as a single query task. Right-deep plans of pipelined hash join operators enhance parallelism [66]: (*a*) the left inputs of all joins are first processed (possibly in parallel) to build hash tables, (*b*) the right input of the deepest operator is streamed to probe the corresponding hash table, and (*c*) the output is pipelined and used to probe the hash tables of the following join operators. The build operations are thus assigned to separate query tasks, while the probe operations for all pipelined hash joins form a single task, provided that all hash tables can simultaneously fit in memory.[1] Probing can only be scheduled after all the build tasks have completed. We show this in Figure 5.1(b): the building of the hash tables for tables A and D, and the output of the merge join operator are assigned to tasks T_4, T_5 and T_3 respectively; task T_6 includes the probing operations for the three pipelined joins. The join task may include the output preparation for any following operator, as described earlier for merge join.

We have chosen not to account for *segmented bushy trees* [52]. These are a coarse form of parallelism, more appropriate for shared-nothing architectures. Instead, we combine partitioning and pipelining, as shown in Figure 5.2(a): the deepest right table (E) is split in equally sized partitions E_i. The tuples of each partition are then used to probe the hash tables of the three join operations independently, so each partition can be processed by a separate thread. This was also adopted in [8] and results in: (*a*) different threads sharing common hash tables, which may be buffered in the lowest level of cache memory in the CPU, and (*b*) pipelined tuples being processed by the same thread, thus exploiting temporal cache locality and reducing the number of cache misses that would occur if the tuples were "snooped" to other cores.

Scheduled plan. We consider query engines supporting *fine-grained parallelism*, which means that all forms of parallelism can be combined in the same query. We

[1] In case both join inputs are partitioned, as in Grace hash join [48], the modelling of hash join is the same as that of merge join.

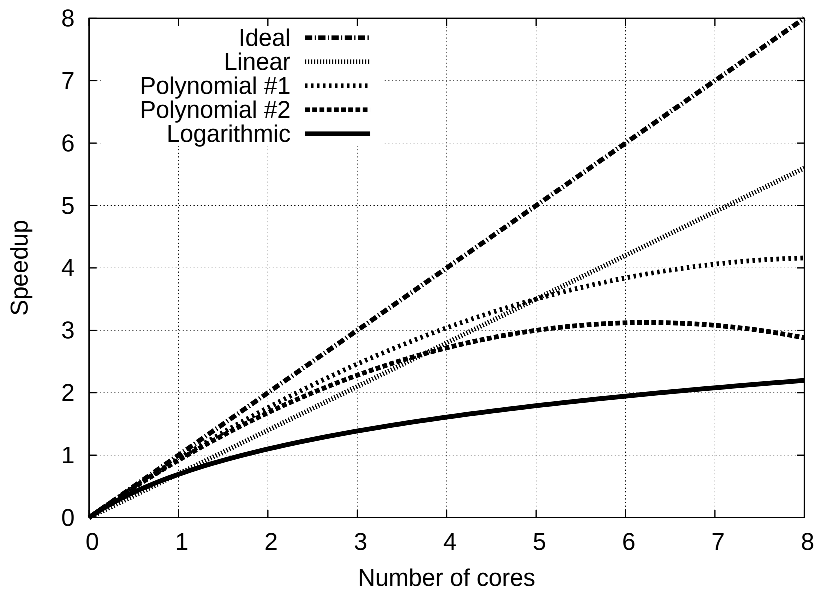


Figure 5.3: Scalability curves

illustrate this in Figures 5.2(a) and 5.2(b) where we assume a quad-core CPU using a thread-per-core execution model. We can schedule three join operations to run in parallel: a merge join on the first two cores, with each thread sorting one of input tables *B* and *C*, and two build operations for tables *A* and *D*. When both sorting operations finish, the combined merge join/build task starts and uses the first two cores; the third and fourth cores still execute the build operator. After all hash tables have been built the final task starts, where each thread executes the three pipelined hash joins over its individual partition of table *E*.

An alternative is to use all available threads for each operator and execute operators sequentially. This, however, is sub-optimal. The reason lies in the form of the scalability curve, as depicted in Figure 5.3 and analytically described in Section 4.3. If using *N* threads would speed up execution *N* times, as shown by the "ideal" curve, the naïve approach above would result in the ideal speedup and be sufficient. In practice, scalability is penalised by resource contention (for the memory bus and the caches) and synchronisation overhead. Its curve may be (approximately) linear but with a smaller slope than the ideal (the "Linear" curve of Figure 5.3); or it may follow a decreasing trend that either approaches a maximum value (the “Polynomial A” and “Logarithmic” curves); or drops after a certain point (the “Polynomial B” curve). It may therefore be preferable to distribute threads to different tasks and execute them in parallel instead of running each task using all threads.

The side-effect of inter-operator parallelism is the introduction of stalls due to differences in execution times across the tasks that start concurrently. Execution stalls stem from [28]: (*a*) *data dependencies*, caused by blocking operators that require their input be fully generated and processed before commencing, and (*b*) *resource dependencies*, caused by the unavailability of required resources, such as cores and main memory. We show how data dependencies create stalls in Figure 5.2(b). Task T_3 must wait for task T_1 to complete, though the output of task T_2 is available earlier. The same applies to task T_6, which needs to wait for tasks T_3, T_4 and T_5. Note that task T_3 does not need to wait for tasks T_4 and T_5, despite T_1, T_2, T_4 and T_5 starting at the same time. An example of resource dependency is the following: if T_4 was scheduled to immediately execute after T_3 on the same core(s), it would be stalled until T_3 completes even though their datasets are different. Worse, if the size of the combined dataset of T_1, T_2, T_4 and T_5 exceeds the memory budget, they cannot run in parallel; the execution of at least one task will be postponed until enough memory is available.

Model principles. The parameters of our model are given in Table 5.1. The query engine allocates N threads from the budget of hardware-supported threads to the execution of a query. Particularly bad thread allocations to cores, *e.g.*, allocations that may increase synchronisation overhead, or result in cache thrashing, can be captured through the scalability curve. For each query task T_i, we estimate the execution time (in CPU cycles or seconds) using analytical expressions for single-threaded execution, as in [54]. The architectural parameters should be set with respect to multithreaded execution. *e.g.*, the query is allocated a share of the caches and the memory throughput according to the number of threads it uses. Furthermore, we are given the scalability curves of each task based on the resource and the synchronisation restrictions imposed.

Parameter	Description
N	number of threads used
K	number of query tasks
T_i	query task i
S_i^n	speedup of task T_i when executed by n threads
C_i^n	cost of task T_i when executed by n threads (CPU cycles or seconds)
M	allocated memory budget
M_i	required memory for task T_i

Table 5.1: Model parameters

The curves can either be statically approximated for each operator (using the calibration method of [54]), or dynamically computed using analytical expressions, according to the methodology of Section 4.3. Note that execution pipeline stalls, *e.g.*, due to cache misses or contention for the memory bus [57], depend on the hardware platform and the query engine design; their impact is captured by the cost functions, which are considered as given.

Next, we specify the expected cost of each task when scheduled on n threads: $C_i^n = \frac{C_i^1}{S_i^n}$. Then, we extract the cost and the memory required for a given schedule by recursively applying the following formulas (shown here for two tasks):

- $C_{AB}^n = C_A^n + C_B^n$

  $M_{AB} = \max(M_A, M_B) \leq M$

  if there exists a dependency between T_A and T_B so they need to be serialised and run on n threads each, or else

- $C_{AB}^n = \max(C_A^i, C_B^j), i + j = n$

  $M_{AB} = M_A + M_B \leq M$

  if T_A and T_B can run in parallel using i and $n - i$ threads respectively and their

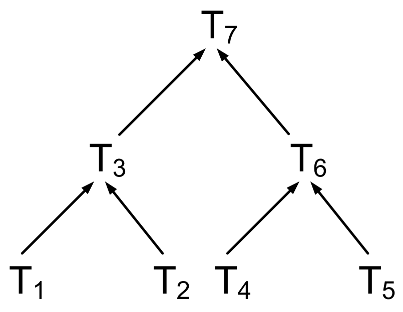


Figure 5.4: Query task tree

combined dataset fits in memory.

For example, based on the generalisation of these formulas, the cost of the schedule of Figure 5.2(b) is equal to:

$$C^4 = \max\left(\left(\max(C_1^1, C_2^1) + C_3^2\right), C_4^1, C_5^1\right) + C_6^4$$

with the memory constraint that:

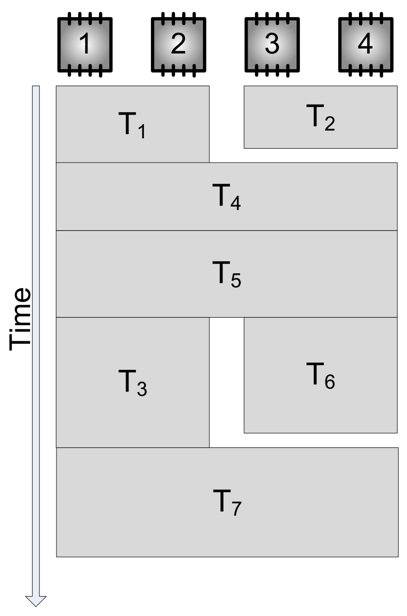
$$\max(M_1 + M_2, M_3) + M_4 + M_5 \leq M$$

We use the following notation to denote an execution schedule: $\langle T, n \rangle$ denotes the execution of task T using n threads. Execution of multiple tasks can either be parallel, denoted by the operator $||$, or sequential, denoted by the operator $+\!\!+$; an execution plan is any combination of such operations, where parentheses are used to denote groupings for readability. For instance, the schedule of Figure 5.2(b) is represented as:

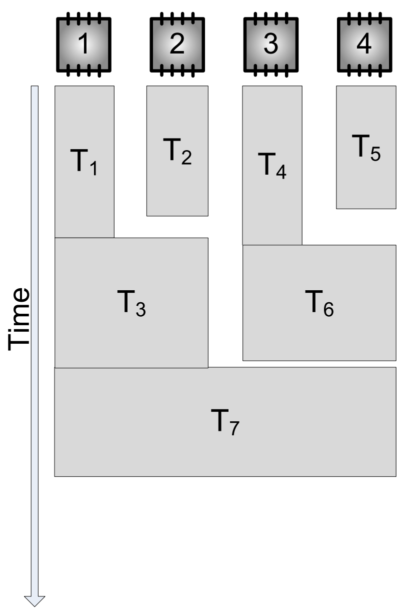
$$((((\langle T_1, 1\rangle \mid\mid \langle T_2, 1\rangle) +\!\!+ \langle T_3, 2\rangle) \mid\mid \langle T_4, 1\rangle \mid\mid \langle T_5, 1\rangle) +\!\!+ \langle T_6, 4\rangle$$

5.3 Algorithms

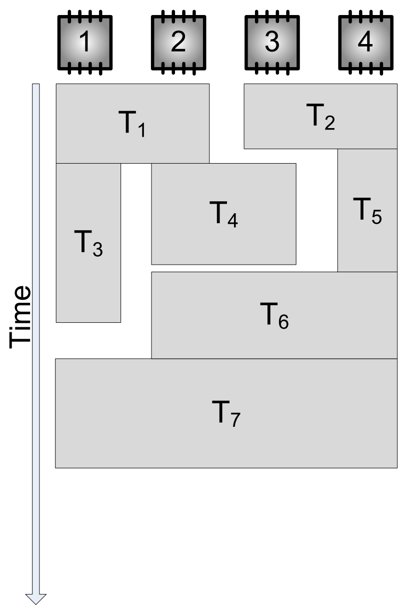
The scheduling problem we study can be reduced to parallel task scheduling, which has been proven to be NP-hard [27]. Given our parallelism model and a query of K tasks to be executed using N threads, there are $2^N - 1$ different ways to schedule a task over N threads (using some or all of them). The K tasks can be permuted in $K!$ ways,



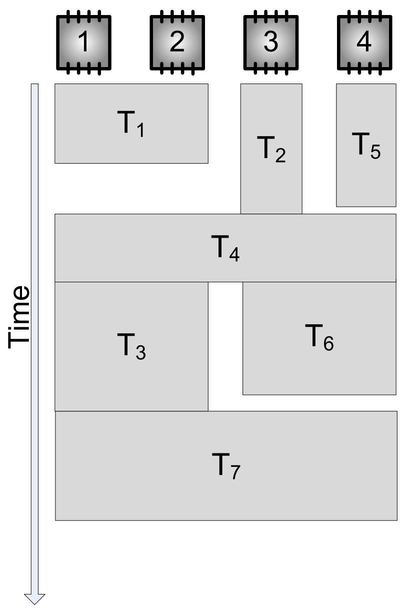
(a) expandChildren



(b) expandDescendants



(c) expandSibling



(d) greedyLevel

Figure 5.5: Scheduling algorithms

giving a total of $K! \cdot (2^N - 1)$ plans.[2] Though this is an overestimation, as data and resource dependencies will reduce the number of interesting or even possible plans, the number renders exhaustive techniques inapplicable even for small values of N and

[2]We use the terms *plan* and *schedule* interchangeably hereafter.

Algorithm 5: The scheduleTree algorithm

```
Input: query task tree $Q$, allocated memory budget $M$
Output: optimal execution plan $P_o$ and cost $C_o$
1  $\ell = \max \text{level} \in Q$;
2  if depth-first expansion then
3      for $i = \ell - 1$ to $1$ do
4          $L$ = list of nodes in $Q$ at level $i$;
5          foreach $t \in L$ do  $t.P = \mathsf{expandNode}(t, N, M)$;
6      $P_o = T_{\text{root}}.P \mathbin{+\!\!+} \langle T_{\text{root}}, N \rangle$;
7  else                                        /* breadth-first expansion */
8      $P_o = \perp$;
9      for $i = \ell$ to $1$ do
10         $L$ = list of nodes in $Q$ at level $i$;
11         $P_o = P_o \mathbin{+\!\!+} \mathsf{expandLevel}(L, N, M)$;
12 return $P_o$;
```

K (see also Section 5.4). The running time of the scheduling algorithm is also crucial; in some cases, multithreaded query execution will run quite fast and we would not want the scheduling time to be comparable to the execution time. We therefore propose and examine heuristics-based techniques that trade the scheduling time with the quality of the generated schedule. For simplicity and without loss of generality we assume that the query plan is a binary tree. Our approach, however, can be straightforwardly extended to task trees of higher arity. We use the bushy query task tree of Figure 5.4 to describe example schedules of each technique, under a thread-per-core execution model with four threads.

The scheduling algorithms take as input the query task tree and traverse it level by level, bottom-up, examining all the nodes of each level before moving to the upper level. The general algorithm is given in Algorithm 5. There are two different possibilities to expand the search space: (*a*) depth-first expansion, abstracted by function expandNode, where the subtree of each node is processed in isolation to compute an efficient schedule, or (*b*) breadth-first expansion, abstracted by the expandLevel function, where all the nodes of the level are processed to find an efficient schedule to execute them. In the former case, the data dependencies are taken into account, by assigning nodes of different subtrees to different sets of threads. Conversely, processing nodes that belong to the same level removes all data dependencies, as the inputs of all involved operators will have already been generated in a previous step.

Note the semantics and the plan generation principles of each different expansion mode. In depth-first expansion, each call to expandNode will generate a plan for the *subtree* of a node at the current level; this is also evident by the traversal starting from level $\ell - 1$. The generated plan is stored locally at that node. To denote so, we slightly "abuse" the node notation in Line 5 to annotate node t with the plan for its *descendants*. All tasks initially store the empty plan, denoted as $\perp$. Each call of the expandNode algorithm will use the stored plans of the subtree's nodes to specify the optimal schedule for this subtree. The full plan is constructed by sequentially running the plan of the root's children, followed by the task of the root node using all available threads (Line 6). Breadth-first expansion works through calls to expandLevel; each call will generate a plan for all the nodes of the current level. The plans are then run sequentially (Line 11).

5.3.1 Node expansion variants

5.3.1.1 The expandChildren algorithm

This is the simplest and quickest scheduling algorithm. The description is given in Algorithm 6. The possible ways of executing the children tasks is either to run them sequentially using all available threads, or distribute the threads across the children tasks and run them in parallel. In both cases, we ensure the schedule's memory re-

Algorithm 6: The expandChildren algorithm

```
Input: node $t$, number of threads $n$, allocated memory budget $M$
Output: optimal execution plan $P_o$ for the subtree of $t$
1  $l$ = left child of $t$; $r$ = right child of $t$;
2  $C_o = C_l^n + C_r^n$;
3  if $\max(M_l, M_r) > M$ then return $\perp$;
4  $n_t = n$;
5  if $M_l + M_r \leq M$ then
6      foreach $i \in [1, n-1]$ do
7          $C_t = \max(C_l^i, C_r^{n-i})$;
8          if $C_o > C_t$ then $n_t = i$; $C_o = C_t$;
9  if $n_t = n$ then $P_o = \langle l, n \rangle +\!\!+ \langle r, n \rangle$;
10 else $P_o = \langle l, n_t \rangle \parallel \langle r, n - n_t \rangle$;
11 return $P_o$;
```

quirements are met. An example schedule of this algorithm for Figure 5.4 is shown in Figure 5.5(a). This algorithm considers each node of the query task tree only once. The drawback is that only a small subset of the search space is explored so plan quality may be low.

5.3.1.2 The expandDescendants algorithm

This is an extension of the expandChildren algorithm to deeper levels. The complete algorithm is given in Algorithm 7. The intuition is that up to N tasks having a common ancestor and belonging to the same level may run in parallel; this means that the algorithm can look from the current level down to some level D. The parent tasks will start as soon as both their children have completed. The algorithm uses dynamic programming and stores the optimal sub-plan for the computation of each internal node. This is achieved by comparing the costs of (a) the best plan for computing the input of the chil-

Algorithm 7: The expandDescendants algorithm

```
Input: node $t$, number of threads $n$, search depth $D$, allocated memory budget $M$
Output: optimal execution plan $P_o$ for the subtree of $t$
1  $l$ = left child of $t$; $r$ = right child of $t$;
2  $P_o = l.P +\!\!+ r.P +\!\!+ \text{expandChildren}(t,n,M)$;
3  if $D = 1$ then return $P_o$;
4  foreach $i \in [2^{D-1}, n-2^{D-1}]$ do
5      $P_l = \text{expandDescendants}(l,i,D-1,M-M_l)$;
6      $P_r = \text{expandDescendants}(r,n-i,D-1,M-M_r)$;
7      $P_b = \text{bottom}(P_l,D-1) +\!\!+ \text{bottom}(P_r,D-1)$;
8      $P_{ll} = \text{top}(P_l,D-1) +\!\!+ \langle l,i \rangle$;
9      $P_{rr} = \text{top}(P_r,D-1) +\!\!+ \langle r,n-i \rangle$;
10     $P_t = P_b +\!\!+ (P_{ll} \parallel P_{rr})$;
11     if $cost(P_o) > cost(P_t)$ then $P_o = P_t$;
12 return $P_o$;
```

dren, plus the optimal way of scheduling the children (as returned by expandChildren) with (b) the best cost of recursively applying the algorithm to the node's children using various distributions of threads to each child. When the depth is 1, the algorithm returns the best plan for the children according to the expandChildren algorithm.

The plan returned by each recursive call of the algorithm is partitioned as follows: (a) nodes deeper than D levels from the current node generate the input of the nodes that will be scheduled in parallel, so they need to be scheduled first (Line 7), and (b) nodes at a depth less than or equal to D will be executed in parallel (Line 10). We use the bottom and top functions to extract the corresponding sub-plans of the plan returned by each recursive call. Both functions accept as input a plan and a depth. From that plan they isolate a sub-plan that corresponds to specific nodes of the task tree either down to the given depth (for top), or from the given depth and below (for bottom). The output of top is a parallel plan for all tasks down to the given depth; bottom returns a sequential plan of the schedules stored in tasks of the given depth. Note that these subplans are complementary: $\mathsf{bottom}(P,D) +\!\!+\ \mathsf{top}(P,D) \equiv P$.

To better understand the algorithm, consider the plan of Figure 5.4 and what happens when processing node T_7; Figure 5.5(b) depicts a possible output schedule. Recall that given the bottom-up traversal of the task tree, nodes T_3 and T_6 are already annotated with the optimal execution schedules of their children. When scheduling the execution of T_3 and T_6 themselves, there are two choices. The first is to sequentially run the schedules of the children of T_3 and T_6 (stored in $T_3.P$ and $T_6.P$) and then schedule T_3 and T_6 using N threads by calling expandChildren. This is the generated plan of Line 2 of Algorithm 7. The second alternative is to first execute sequentially the stored schedules of T_7's descendants at depth $D = 2$ and then redistribute threads to nodes of the top D levels. In this example, the bottom and top calls return:

- $\mathsf{bottom}(P_l, 1) = T_1.P +\!\!+ T_2.P = \perp$
- $\mathsf{bottom}(P_r, 1) = T_4.P +\!\!+ T_5.P = \perp$
- $\mathsf{top}(P_l, 1) = \langle T_1, 1\rangle \parallel \langle T_2, 1\rangle$
- $\mathsf{top}(P_r, 1) = \langle T_4, 1\rangle \parallel \langle T_5, 1\rangle$

to compose the schedule:

$$\begin{aligned} P_t &= (((\langle T_1,1\rangle \parallel \langle T_2,1\rangle) +\!\!+ \langle T_3,2\rangle) \parallel \\ &\quad ((\langle T_4,1\rangle \parallel \langle T_5,1\rangle) +\!\!+ \langle T_6,2\rangle)) \end{aligned}$$

The plans stored in the nodes of level 3 are empty as this is the last level of the task tree. Note that the schedule $(\langle T_1,1\rangle \parallel \langle T_2,1\rangle)$ above is generated by the recursive call of Line 5; the same nodes were processed when the algorithm was called by scheduleTree for node T_3. In the latter call the nodes were scheduled using all available threads, while recursive calls always use a smaller number of threads. The second schedule is

Algorithm 8: The expandSibling algorithm

```
Input: node $t$, number of threads $n$, search depth $D$, allocated memory budget $M$
Output: optimal execution plan $P_o$ for the subtree of $t$
1  $l =$ left child of $t$; $r =$ right child of $t$;
2  $P_o = l.P \mathbin{+\!\!+} r.P \mathbin{+\!\!+} \mathsf{expandChildren}(t,n,M)$;
3  if $D = 1$ then return $P_o$;
4  foreach $i \in [1, n - 2^{D-1}]$ do
5      $P_r = \mathsf{expandDescendants}(r, n-i, D-1, M-M_l)$;
6      $P_t = l.P \mathbin{+\!\!+} \mathsf{bottom}(P_r, D-1)$;
7      $P_t = P_t \mathbin{+\!\!+} (\mathsf{top}(P_r, D-1) \parallel \langle l, i\rangle)$;
8      if $cost(P_o) > cost(P_t)$ then $P_o = P_t$;
9      $P_l = \mathsf{expandDescendants}(l, n-i, D-1, M-M_r)$;
10     $P_t = r.P \mathbin{+\!\!+} \mathsf{bottom}(P_l, D-1)$;
11     $P_t = P_t \mathbin{+\!\!+} (\mathsf{top}(P_l, D-1) \parallel \langle r, i\rangle)$;
12     if $cost(P_o) > cost(P_t)$ then $P_o = P_t$;
13 return $P_o$;
```

generated in Line 10 and the cost-based decision between the two alternatives is taken in Line 11.

The expandDescendants algorithm is instantiated with a depth of $D = \log_2(N)$ and a number of threads $n = N$. When called, the algorithm effectively uses an N-wide window to schedule all tasks. Therefore, the maximum depth of the search space expanded for each internal node is at most $D = \log_2(N)$, processing $(2^N - 2)$ nodes. Each separate instance (*i.e.*, recursive call) of the algorithm considers $(n + (n - 2^D)) = 2n - 2^D$ plans.

5.3.1.3 The expandSibling algorithm

The expandSibling algorithm addresses the case where one node can be executed in parallel with the descendants of its sibling, therefore ensuring there are no data dependencies. As soon as each pair of descendants completes, the parent task starts. Such plans are advantageous when the cost of a task is approximately equal to the cost of its sibling and the sibling's descendants (limited to a certain depth). We present expandSibling in Algorithm 8; it is instantiated again with $D = \log_2(N)$ and N threads. Its data flow is similar to that of expandDescendants. The difference is that when scheduling the children of a node, the algorithm tests if it is more efficient to execute one child task in parallel with the top tasks of its sibling subtree (Lines 5 to 12). This algorithm also uses an N-wide scheduling window, searching the subtree of each node up to a depth of $\log_2(N)$ levels; it thus expands $(2^N - 2)$ nodes. Each instance of expandDescendants expands $(n + 2(n - 2^D)) = 3n - 2^{D+1}$ plans. The expandSibling algorithm, hence, examines more plans than expandDescendants. Its running time is greater but it has a higher probability of identifying an optimal schedule.

An example schedule generated by Algorithm 8 for the tree of Figure 5.4 is shown in Figure 5.5(c). Task T_3 is assigned to the first thread; T_6 and its children are assigned to the other three threads. Task T_3 starts at the same time as T_4 and T_5. When the last two tasks complete, T_6 starts executing with three threads, while T_3 is still executed by the first thread. When both T_3 and T_6 complete, T_7 is executed by all available threads.

5.3.1.4 The expandHybrid algorithm

The expandDescendants and expandSibling algorithms can be combined in a single hybrid algorithm; this algorithm is termed expandHybrid. Each call to expandHybrid will compare the cost of the subplans created by the expandDescendants and expandSibling algorithms. That way we can enumerate more possible plans for each internal node's subtree and explore a larger portion of the search space. The algorithm effectively tries to distribute threads across both the children of a node, as well as its sibling. The total

Algorithm 9: The findParallel algorithm

```
Input: list $L$ of remaining nodes to be scheduled, number of threads used so far
       $n_u$, approximation cost $C_p$, allocated memory budget $M$, memory used so
       far $M_u$
Output: execution plan $P_o$ for a subset of $L$, updated list $L$ of remaining nodes
        to be scheduled
1  $C_{\text{dist}} = \infty$;
2  foreach $T_i \in L$ do
3      if $M_i + M_u > M$ then continue;
4      foreach $j \in [1, N - n_u]$ do
5          if $|C_p - C_i^j| < C_{dist}$ then
6              $C_{\text{dist}} = |C_p - C_i^j|$; $T_o = T_i; n_o = j; M_o = M_i$;
7  if $C_{dist} = \infty$ then return $(\perp, L)$;
8  $L = L - T_o$;
9  if $n_o + n_u < N$ then
10     $P_t, L \leftarrow$ findParallel$(L, n_o + n_u, C_p, M, M_u + M_o)$;
11     return $(\langle T_o, n_o \rangle \parallel P_t, L)$;
12 return $(\langle T_o, n_o \rangle, L)$;
```

number of plans considered by the hybrid algorithm is $5n - 3 \cdot 2^D$, *i.e.*, the sum of the numbers of plans considered by expandDescendants and expandSibling.

5.3.2 Level expansion

Recall from the scheduleTree algorithm (Algorithm 5), that either each node can be processed by itself, or an optimal schedule can be computed for all the nodes of a level. The latter task is undertaken by the greedyLevel algorithm that we will introduce shortly. It works by identifying operators at the same level of the execution plan that bear a similar execution cost, thereby minimising execution stalls. This is the goal of the findParallel algorithm of Algorithm 9. Its input is a list of unscheduled tasks, the number of already allocated threads and the approximation cost. The algorithm identifies which task of the non-scheduled ones bears a cost closer to the target cost when scheduled in parallel (*i.e.*, when using some or all of the remaining threads). If the closest cost corresponds to a schedule that does not use all unallocated threads, the function is recursively applied to identify the task(s) that can be scheduled to be executed by the remaining thread(s).

Based on the findParallel algorithm, we give the greedyLevel algorithm in Algorithm 10. The algorithm groups the tasks of each level in sets that can be efficiently scheduled in parallel. The key issue is to identify the combination of task and number of threads, the cost of which will be used as the approximation cost by the findParallel algorithm. There are two cases: if the number of remaining tasks to be scheduled is below a specific threshold $\mathcal{R}$, we test all possible combinations of tasks and number of threads. Otherwise, we use the findMinDistance function to identify the input task. This function works as follows: for each task T_i and possible thread allocation for its execution n_i, we compute a list $L(T_i,n_i)$ of all other tasks that can be executed in parallel with it, *i.e.*, the list contains pairs (T_j,n_j) of tasks $T_j, j \neq i$ executed with n_j threads such that $n_i + \sum_j n_j \leq N$ and $M_i + \sum_j M_j \leq M$. We pick the task T_m and thread allocation n_m that minimises $\sum_{(T_j,n_j)\in L(T_m,n_m)} |C_m^{n_m} - C_j^{n_j}|$. The intuition is that we pick a task that introduces the smallest possible stall if run in parallel with other tasks of the same level. Note that this is only one possible way to identify a task. The experimental results of Section 5.4 show that this heuristic works well in practice.

The number of plans considered by this algorithm depends on the threshold used to specify if one or multiple approximation costs are passed to the findParallel algorithm. This is because findMinDistance greedily decides on the task and the target cost to be used. The $\mathcal{R}$ threshold can be specified through calibration, *i.e.*, by gradually increasing the number of tasks explored until the scheduling cost becomes significant

```
Algorithm 10: The greedyLevel algorithm
Input: list $L$ of remaining tasks to be scheduled, allocated memory budget $M$
Output: optimal plan $P_o$ for $L$, cost of plan $C_o$
1  $C_o = \infty$;
2  if $length(L) < \mathcal{R}$ then
3      foreach $T_i \in L$ do
4          $L' = L - T_i$;
5          foreach $j \in [1, N]$ do
6              $(P', L') = \mathsf{findParallel}(L', j, C_i^j, M, M_i)$;
7              $P_t = \langle T_i, j \rangle \parallel P'$;
8              if $length(L') > 0$ then
9                  $P_t = P_t \mathbin{+\!\!+} \mathsf{greedyLevel}(L', M)$;
10             if $cost(P_t) < C_o$ then
11                 $P_o = P_t$; $C_o = cost(P_t)$;
12 else
13     $(T_m, n_m) = \mathsf{findMinDistance}(L, N, M)$;
14     $L' = L - T_m$;
15     $(P', L') = \mathsf{findParallel}(L', n_m, C_{T_m}^{n_m}, M, M_{T_m})$;
16     $P_t = \langle T_m, n_m \rangle \parallel P'$;
17     if $length(L') > 0$ then
18         $P_o = P_t \mathbin{+\!\!+} \mathsf{greedyLevel}(L', M)$;
19 return $P_o$;
```

for the given implementation and the host hardware platform.

5.4 Experimental study

Our aim is to examine the behaviour of the proposed algorithms in various hardware platforms and for a wide range of query engine designs, the performance of which is captured by the scalability curves. The only practically applicable method to achieve this is to use simulation. This method is widely used in the architecture community and has been adopted in previous work as a simple and reliable technique to experiment with query engines and architectural setups that are not available or do not currently exist as commercial or research products. The restrictions of the simulation method are the reliability of the scalability estimations and the accuracy of the optimiser's cost model; both are assumed to be given. However, these restrictions are present in every aspect of query optimisation – the optimiser's choices are only good as its cost model and search strategy.

We implemented all the algorithms we have proposed, along with an exhaustive algorithm that uses *branch and bound*. All algorithms were implemented in C++ and compiled using the GNU g++ compiler (version 4.3.3, `-O2` optimisation level). We ran our experiments on a Dell Precision T5400 workstation, with an Intel Xeon E5420 quad-core processor and 4GB of physical memory, clocked at 2.5GHz, and running GNU/Linux (64-bit version, kernel 2.6.26). The scheduling code ran in a single thread; its running time will improve in a multithreaded implementation, in line with [38]. We did not use multiple threads to run the scheduling algorithms as we merely wanted to compare the performance of the scheduling algorithms and the quality of the schedules they produce. Through calibration we set the threshold $\mathcal{R}$ of the greedyLevel algorithm to seven tasks.

The parameters of our experiments are shown in Table 5.2. For each run, we specified: (*a*) the number of threads to be used (N), (*b*) the number of query tasks in the task tree (K), and (*c*) the form of the scalability curve (SP). We would then create eight distinct bushy binary query task trees; left- or right-deep trees reduce the number of scheduling alternatives and therefore narrow the differences in optimality across the

Parameter	**Values**
N	$[4,32]$ threads
K	$[3,63]$ tasks
C_i^1	$[100,100000]$ milliseconds
SP	*Linear*, *Polynomial #1*, *Polynomial #2*, *Logarithmic*

Table 5.2: Simulation parameters

K	**Time for** $N = 4$ **(s)**	**Time for** $N = 8$ **(s)**
3	0.001	0.001
5	0.001	2263.635
7	5.421	-
9	9553.883	-

Table 5.3: Execution time for exhaustive search

proposed algorithms.

The cost of single-threaded execution for each operator was set to randomly vary between 100ms and 100s. We consider the following scalability curves: (*a*) a linear function, with the gradient varying between 0.4 and 0.9 (*Linear*), (*b*) a polynomial non-decreasing function of random curvature, similar to the "Polynomial A" curve of Figure 5.3 (*Polynomial #1*), (*c*) a polynomial function of random curvature that may be decreasing, resembling either the "Polynomial A" or the "Polynomial B" curves of Figure 5.3 (*Polynomial #2*), and (*d*) a logarithmic function of random curvature (*Logarithmic*).

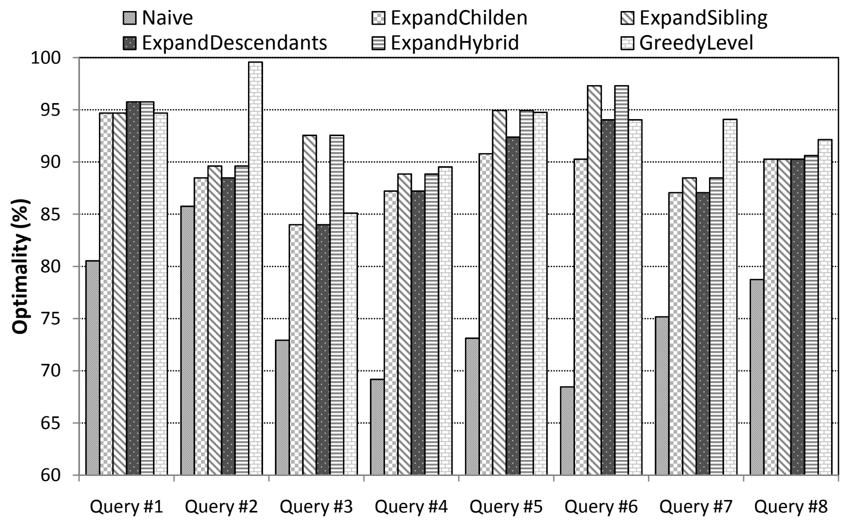
We report the average execution time of ten runs of each algorithm, the deviation being less than 1% in all cases, as well as the optimality of the output schedule. To measure optimality, we compared the output schedule with two reference schedules: (*a*) the optimal schedule, identified through exhaustive enumeration, and (*b*) the naïve schedule of assigning all available threads to each task. The first approach is the best metric of optimality; however, it is practical only for a small number of tasks and threads, as we show in Table 5.3. Exhaustive enumeration needs 5.4s for seven tasks and four threads; it needs more than 2.5 hours for nine tasks; scaling to eight threads renders exhaustive enumeration practically inapplicable even for a simple query of five tasks. Hence, for more complex queries and setups, we used the second approach, which shows the improvement in execution time each algorithm results in when compared to the naïve schedule.

5.4.1 Schedule optimality

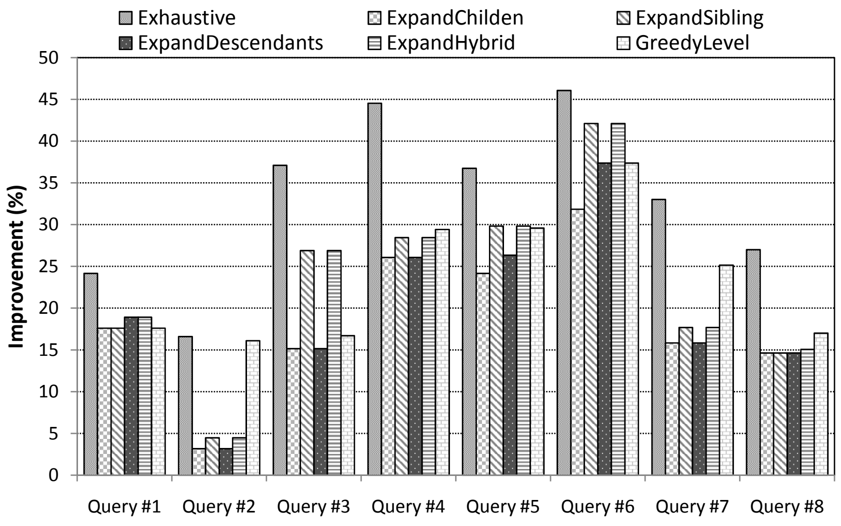
We first measured the optimality of the generated schedules. We started off with four threads and trees of seven tasks, to allow exhaustive enumeration to complete. We present the results only for the *Polynomial #2* scalability curve type as we treat the impact of all scalability curves in the next section. We optimised eight randomly generated queries of the specified number of tasks. The results are presented in Figure 5.6. On the left we show the comparison between the generated schedules and the optimum one; on the right we show the improvement over the naïve schedule of each query. In all cases either extendHybrid or greedyLevel achieved an optimality of 90%; greedyLevel also identified the optimum plan for the second query. Compared to the naïve approach, the proposed heuristics improve the quality of the schedule by at least 16%, on average by 26%, to a maximum of 42% for the sixth query of the workload.

5.4.2 Impact of the scalability curve

The scalability curve significantly affects the optimality of the generated schedules. As the curve approximates the ideal one (see also Figure 5.3), the room for performance



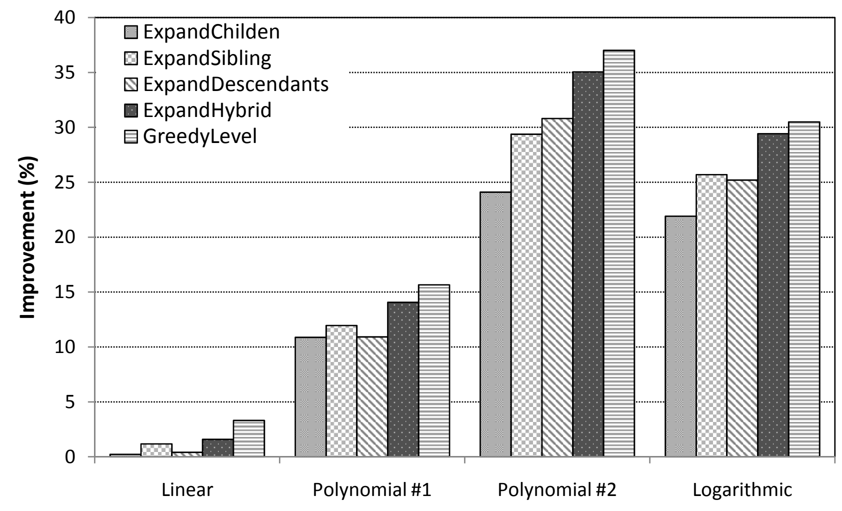
(a) Optimality



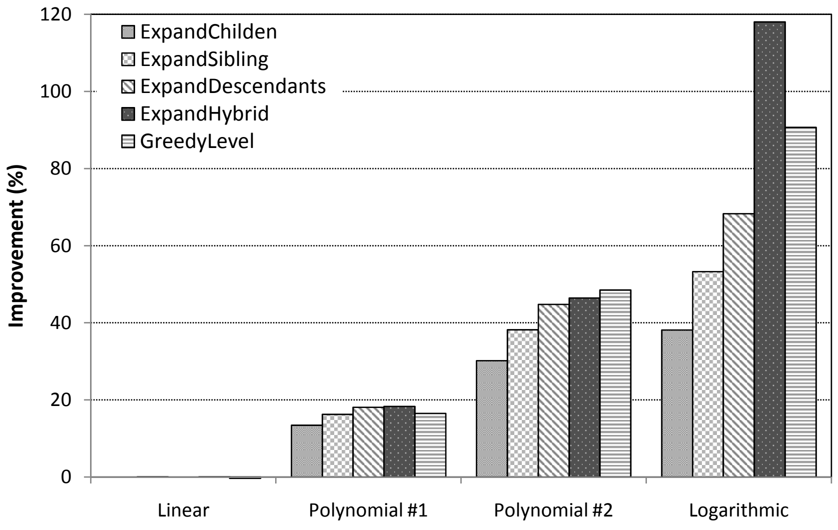
(b) Improvement

Figure 5.6: Algorithm Optimality ($N = 4$, $K = 7$)

improvement from elaborate task scheduling shrinks. This is shown in Figure 5.7, where we give the improvement of each scheduling algorithm over the naïve approach for various scalability curves. We used queries of fifteen tasks and either eight or thirty-two execution threads. If the speedup is linear, the improvement over the naïve approach is marginal (less than 3%), so the use of sophisticated scheduling algorithms is redundant. However, this kind of speedup is rather rare [11]; contention due to re-



(a) Improvement ($N = 8$)



(b) Improvement ($N = 32$)

Figure 5.7: Impact of scalability curve ($K = 15$)

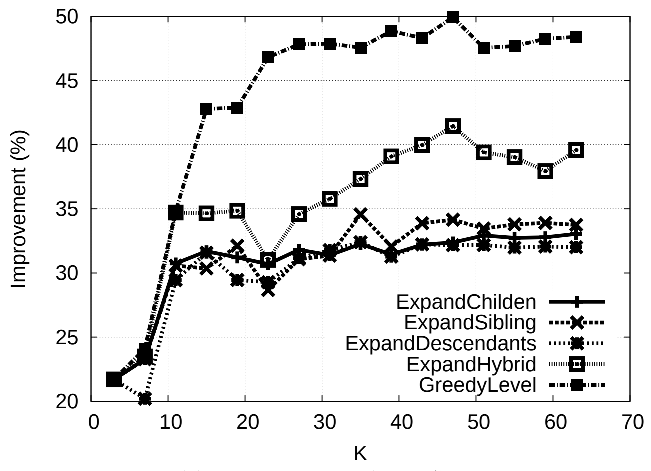
source sharing and synchronisation gradually degrades scalability as the number of threads increases. In such cases our scheduling algorithms exhibit significant improvements over the naïve plan. These improvements increase as scalability deteriorates. For eight threads, the greedyLevel and expandHybrid algorithms reach similar levels of optimality, outperforming the naïve approach by 30% or more. With thirty-two threads available, the expandHybrid algorithm outperforms its breadth-first counter-part and achieves a 120% improvement over the naïve schedule, when scalability is low (as is the case for the *Logarithmic* curve).

5.4.3 Impact of the number of tasks and threads

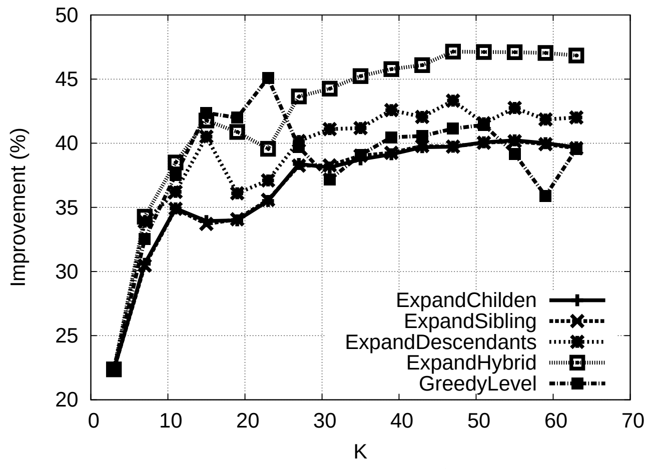
The optimality of the schedules produced by our algorithms depends on both the number of tasks in the query and the number of threads. The depth-first algorithms consider alternative plans in a depth up to $\log_2(N)$. Thus, the more threads there are, the more likely the algorithms will identify an optimal plan. On the contrary, the greedyLevel algorithm processes each level separately; its efficiency increases as the levels grow wider. This behaviour is verified in Figures 5.8 and 5.9, where we present the optimality and the execution time of our algorithms for various combinations of query task and thread counts. In all cases, we use the *Polynomial 2* scalability curve type.

In Figure 5.8 we vary the number of tasks between three and sixty three; we set the number of threads to either four or sixteen. In the first case (Figure 5.8(a)), the optimality of greedyLevel improves quickly and stabilises to about 50% for twenty-three tasks or more. Conversely, the depth-first algorithms process up to a depth of only two, or up to six children of each node in the query task tree. Since they explore a smaller part of the search space the quality of their results is poorer, maximising at an improvement of 40% for the expandHybrid algorithm. The fluctuation of all curves is due to the randomness of the queries used and the scalability curve of each query task. As the number of threads grows to sixteen, the greedyLevel algorithm stops being the best choice as shown in Figure 5.8(b). In this case, the depth-first algorithms explore up to four additional levels, or thirty children nodes of each node in the query task tree (at most), covering a larger subset of the search space than the greedyLevel algorithm. The latter converges to a 40% improvement over the naïve approach, while the expandHybrid algorithm reaches a 48% improvement.

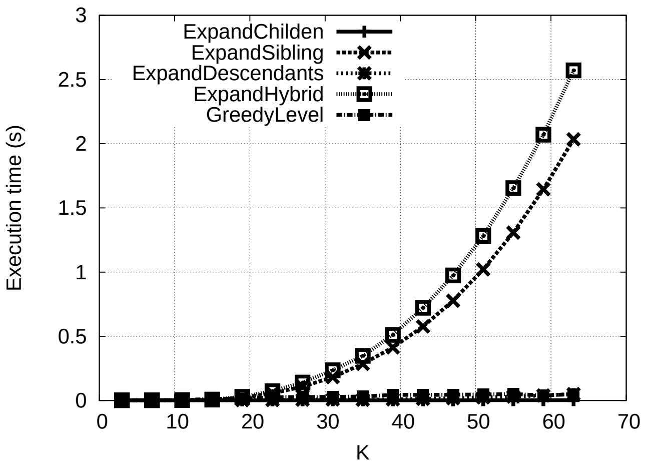
There is, however, a penalty in execution time for this advantage in quality. The depth-first algorithms compare an increasingly larger number of possible sub-plans as



(a) Improvement ($N = 4$)



(b) Improvement ($N = 16$)



(c) Execution time ($N = 16$)

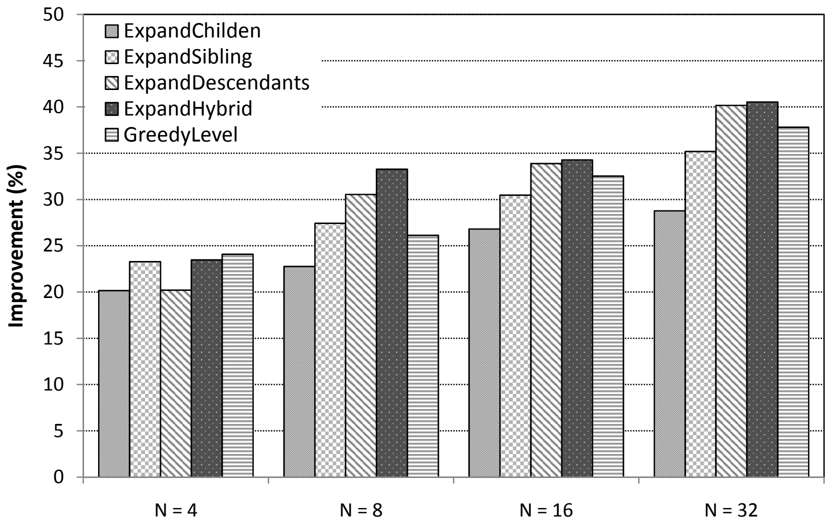
Figure 5.8: Impact of the number of tasks

the number of tasks increases; this is not the case for the greedyLevel algorithm, as the number of compared plans is set by the $\mathcal{R}$ threshold, irrespective of the number of tasks. This is verified in Figure 5.8(c), where the execution time of the depth-first algorithms grows exponentially with the number of tasks. On the other hand, the execution time of the breadth-first one is almost constant. Still, this is only observable for large task counts (more than twenty); it is negligible in the case of four threads, where all algorithms execute in less than 1ms. Note that the implementation of the expandDescendants algorithm is very efficient: it identifies schedules of good quality, while examining fewer plans than the expandSibling and expandHybrid algorithms. That way, it runs in time comparable to the greedyLevel algorithm.

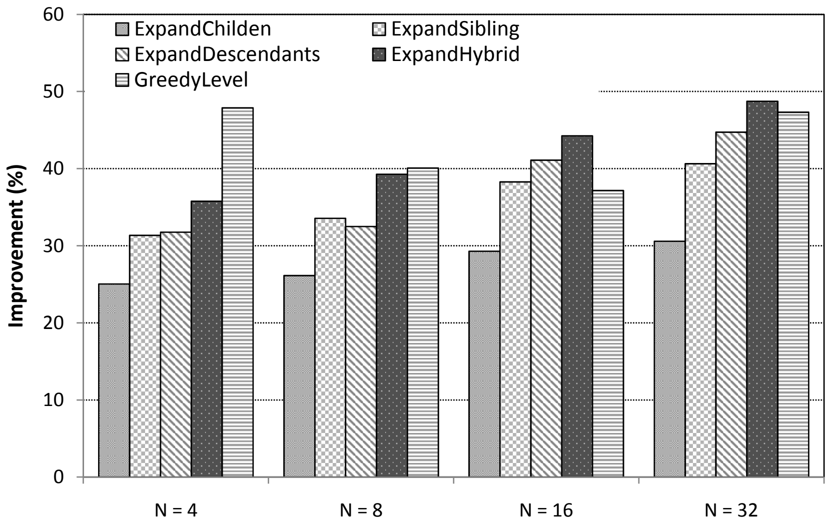
Finally, we show in Figure 5.9 how the number of threads affects the optimality of the generated schedules. We varied the number of threads between four and thirty-two, while the number of tasks was either seven (Figure 5.9(a)) or thirty-one (Figure 5.9(b)). Again, we observe that the greedyLevel algorithm performs best for small thread counts; as the number of threads grows, the expandHybrid algorithm produces plans of better quality than the competition. In terms of execution time (Figure 5.9(c)), the difference between the expandHybrid and expandSibling algorithms and the rest becomes significant for sixteen threads or more and follows an exponential trend.

5.5 Discussion

Since our algorithms explore a small subset of the search space, there is no guarantee that one will yield a better plan than another. Still, the number of schedules each algorithm considers, along with the experimental results, can provide some intuition on the best-fitting scheduling algorithm for the query at hand. There are four basic parameters that affect the choice of algorithm: (*a*) the estimated running time of the query, *i.e.*, if it is a long-running or a short-running query; (*b*) the scalability curve, *i.e.*, the utility of multithreaded execution; (*c*) the complexity of the query, *i.e.*, the number



(a) Improvement ($K = 7$)



(b) Improvement ($K = 31$)

N	expandChildren	expandSibling	expandDescendants	expandHybrid	greedyLevel
4	0.002	0.003	0.003	0.003	0.004
8	0.002	0.008	0.003	0.008	0.007
16	0.002	0.183	0.008	0.236	0.029
32	0.002	2.699	0.219	6.429	0.108

(c) Execution time in seconds ($K = 31$)

Figure 5.9: Impact of the number of threads

of tasks in the query task tree; and (*d*) the available processor resources that can be devoted to the query, *i.e.*, the number of threads the execution engine can allocate to process the query. The interaction of those parameters is intricate and complex to model. At best, we can only come up with some empirical rules that, given each parameter, can help determine the best-fitting scheduling approach.

Query running time. The key observation is that the longer a query is estimated to run for, the longer the optimiser can afford to take for scheduling its tasks. Situations where this arises are when the query will process a large dataset, or when the query is inflationary, *e.g.*, it produces large intermediate result sets. In such cases it is more beneficial to use one of the algorithms that explore a larger portion of the search space, *i.e.*, the expandHybrid, or greedyLevel algorithms. One might even consider a combined approach: both algorithms can be used and the plan with the highest estimated performance improvement will be chosen in the end.

Utility of multithreaded execution. The chosen algorithms during the first phase of optimisation (join ordering and query rewriting) will perform differently depending of how well they scale in a multithreaded implementation. If this speedup is estimated to be close to linear – especially with a high slope – it is not sensible to use elaborate scheduling algorithms: scalability is high, regardless of thread allocation. As such, it pays off to use an algorithm like expandChildren, which will quickly identify the parallelisation potential for every task in the query tree.

Query complexity. The more query tasks are present, the larger is the search space of potential schedules (recall that the size of the search space grows to the factorial of the number of tasks). As such, it is more sensible to use a strategy that explores a large enough portion of the search space and has a high probability of identifying a plan with optimal performance. Given the evidence of Figure 5.8, for small query task counts it is preferable to use the expandHybrid algorithm. However, as the number of query tasks increases, the greedyLevel algorithm seems the best compromise: though its optimality may be lower than the expandHybrid algorithm's, it achieves comparable performance improvements, but only at a fraction of the execution time.

Available resources. The size of the search space is exponential to the number of threads used. Therefore, it is a case similar to the previous one. Algorithms that explore a larger portion of the search space are beneficial, especially if they run fast. GreedyLevel proves a better choice for restricted thread budgets as the number of tasks examined on each call of expandHybrid is low; the choice is complementary for higher thread counts (see also Figure 5.9). If there are only a few tasks in the query tree, the tree's levels are most likely narrow so an algorithm like expandDescendants may be a viable option: it generates high-quality schedules in minimal optimisation time.

Finally, note that the differences between the schedules produced by the different algorithms are usually small. Even if not the most appropriate algorithm is chosen, it is likely that the optimality of its produced plan will be close to that of the plan chosen by the most appropriate algorithm. Furthermore, our heuristics give plans of high optimality in all cases, at a low running cost. We believe these to be the key aspects of a good optimisation algorithm, thus offering conducive evidence of the viability of our proposals.

5.6 Further reading

Parallel query execution and optimisation have been extensively studied in the past, in the context of shared-nothing and shared-memory systems (*e.g.*, [24, 25, 34, 53, 70]). A theoretical perspective on parallel query optimisation was given by Ganguly *et al.* [28], who analyzed the impact of data and resource dependencies on scalability and proved that dynamic programming does not satisfy the principle of optimality for parallel query optimisation, thus enforcing the use of heuristics. In [74], the authors studied the impact of pipelining and partitioning on PrismaDB. They concluded that combining both forms of parallelism is the most efficient approach, especially for bushy operator trees. However, they elaborate neither on how the system decides the number of processors to be allocated to each task task, nor on the order of task execution; we address both issues.

Scheduling for parallel execution was first studied by Chen *et al.* [11], where it was suggested that processors be allocated to tasks according to the scalability curve. Our proposed scheduling techniques compare multiple schedules and core allocation alternatives and therefore exploit parallelism to a higher extent. In [31, 32] the authors studied how near-optimal heuristics can be used to allocate multi-dimensional resources to tasks. Their techniques, however, are restricted to separately optimising each level of the task tree and statically allocating processing cores to tasks according to the scalability curve. If applied to the thread scheduling problem we address, they are equivalent to the naïve approach. In [56], it was proposed that resources should be allocated to pipelined operators with the objective of matching their processing rates and maintaining a constant tuple flow between them. In our model, by combining pipelining and partitioning we avoid queueing issues between pipelined operators. Our techniques are based on allocating tasks to threads so the tasks that are scheduled together will complete in approximately equal times.

Dynamic load balancing for parallel query execution was studied in [7, 8]. The proposal was to decompose operator execution in elementary self-contained units that can be scheduled independently. If a thread finishes with executing its assigned set of units, it can execute units that were originally assigned to other threads. This technique is especially useful in the presence of skew. We consider this approach complementary: we can statically assign tasks to threads according to value distribution statistics and dynamically re-distribute work to them. However, there is a significant synchronisation overhead when units are shared across threads, while the dataset of an execution unit might need to migrate from the private caches of a delaying core to the idling one's caches. Therefore, synchronisation and data transfers may diminish any performance gains from load balancing.

Query optimisation for multicores was addressed in [1]. The authors proposed to assign at least one thread to each query operator and initialise all threads at the same time. The operating system is responsible for scheduling the execution of threads to the available cores. It has been known, however, that creating more processes than available processors in a shared-memory system penalises performance due to scheduling overhead [34]: the processes compete for the CPUs and their execution will most likely be interrupted multiple times before completion. This is aggravated when scheduling threads to cores because: (*a*) the scheduling overhead induced by the OS is comparable to the execution time for short-running threads, and (*b*) thread switching evicts the dataset of each thread from the cache hierarchy, leading to cache thrashing. Thus, we argue that thread budgets should be restricted by the ability of the processor to support their parallel execution and run with minimal synchronisation overhead.

Finally, the authors of [44] studied work sharing in multicores and modelled the performance of concurrently processed, staged queries, while [62] investigated the scheduling of multiple queries with the goal of work sharing (for scans). Moreover, the authors of [51] employ page coloring to prevent cache thrashing when concurrently executing multiple queries. These approaches are complementary to ours; we focus on intra-query parallelism and opportunities for improved execution schedules for the tasks of each query in isolation, given the number of allocated threads and the available memory.

Chapter 6

Conclusions and future directions

6.1 Conclusions

In this thesis we have presented the holistic query evaluation model. Our objective is to optimise query execution when the query dataset fits in main memory; in this case, performance is CPU-bound rather than I/O-bound. The starting point is to replace the static, iterator-based implementations of query operators with code templates. The query engine instantiates and combines these templates at run-time to create query-specific source code, which is then compiled, linked and executed to produce the query results. This technique radically reduces the required number of instructions and data accesses for each query, while the executed code is customised for the host hardware; the latter task is performed both by the query engine and the compiler. To leverage the processing power of multiple cores that modern CMPs incorporate, we present a uniform framework for parallelising query operators. We identify thread contention for memory accesses and for locking as the main scalability restrictions in this execution environment. To quantify the effect of thread contention to performance, we introduce the multithreaded utility ratio and provide formulas to analytically estimate the speedup for each query operation, based on hardware performance metrics. These estimations can be used by the scheduler to generate query schedules that exploit the multicore design for enhanced intra-query parallelism. Towards this direction, we introduce heuristics-based scheduling algorithms that produce schedules of high quality while incurring negligible optimisation cost.

The first part of the holistic evaluation model focuses on template-based code generation. This technique applies just-in-time code generation to query evaluation. To achieve this, the query engine maintains code templates for the supported query operations, which are instantiated according to the parameters of each query, as specified at run-time. The code layout inside the templates is chosen with the following objectives: (*a*) minimisation of function calls, (*b*) reduction of instructions and memory accesses, and (*c*) enhanced cache locality. Our template-based framework for code generation is flexible enough to accommodate sophisticated query evaluation algorithms, such as combined hash partitioning and sorting, and join teams.

To study the viability of code-generation as an alternative query engine design, we implemented a prototype system named HIQUE — the Holistic Integrated Query Engine. Extensive experiments with a variety of datasets and query workloads proved HIQUE's performance advantage when compared with established and currently-emerging database technology. This verifies the efficiency of a query engine that uses per-query code generation. At the same time HIQUE operates over the conventional NSM-based storage layer, so it does not affect any orthogonal aspects of a DBMS like concurrency control and recovery.

We next studied multithreaded query processing on multicore processors. By identifying main memory accesses and thread synchronisation as the main performance bottlenecks, we introduce a uniform framework for implementing query processing algorithms that: (*a*) reduces contention for hardware resources, and (*b*) bears minimal synchronisation overhead. Using this framework, we provide multithreaded versions of sorting, partitioning, join evaluation and aggregation. We extend traditional query evaluation algorithms to adapt their data-flow to the characteristics of CMPs. For instance, combining partitioning with partition page sorting, as opposed to partitioning alone, reduces contention for memory accesses and enhances scalability.

To analytically model the performance and scalability of each algorithm, we introduce the multithreaded utility ratio R for each input unit:

$$R = \frac{C_f}{C_f + C_p + C_l} \tag{6.1}$$

where C_f, C_p and C_l are the fetching, processing and locking costs for this unit accordingly. Using this ratio, the effective memory access cost M' can be calculated as follows:

$$M' = \begin{cases} M & R \leq \frac{1}{N} \\ MNR & R > \frac{1}{N} \end{cases} \tag{6.2}$$

where N is the number of threads used and M is the memory access cost for single-threaded execution. By “pluging in” the equations above to the analytical cost expressions for each query operation we can estimate the expected speedup when utilising multiple threads for its execution.

To verify the efficiency of the proposed multithreaded algorithmic extensions and the correctness of our analytical model for scalability, we extended HIQUE to implement the proposed framework for multithreading and experimented with various queries and schemata. The results verify that the proposed model adequately captures the effect of various parameters to multithreaded performance and provides accurate estimations of the expected speedup. At the same time, our framework for multithreading proves highly efficient and reduces thread contention thus achieving high speedups that, in some cases, are almost linear.

The existence of analytical expressions for scalability in multithreaded query execution can be exploited by the query optimiser to generate scheduled plans of high quality. This stems from the fact that thread contention penalises scalability, so using all available threads for each query operator is suboptimal; on the other hand, assigning pipelined query operators to concurrently executing threads (using a producer–consumer approach) incurs high synchronisation cost. To that end, we model intra-query parallelism for multicore processors and describe how a wide range of algorithms and implementations can be efficiently extended for multithreading.

Regarding the optimisation problem of scheduling query operators to threads, the size of the search space grows quickly, so it is crucial to have an optimisation strategy that converges equally fast to a schedule of high quality. To that end, we present a family of heuristic-based scheduling algorithms that study a subset of the search space using either a depth-first or a breadth-first strategy, and offer a trade-off between running time and schedule quality. We have implemented and experimentally evaluated our techniques. The results show that our algorithms significantly improve schedule quality. Based on these results, we identify cases where an algorithm is to be preferred over its alternatives.

To summarise, the major contributions of this thesis are the following:

- We propose a template-based framework for per-query code generation. This framework maintains the compositional aspects of the iterator model and eases the process of engineering a query engine leveraging code generation. This fact, combined with the increased efficiency of the generated code when compared with static operator implementations, renders holistic query evaluation a viable query engine design alternative.
- We introduce a uniform framework for extending query evaluation algorithms for multithreading, as supported by hardware inside modern CPUs. We then model contention for main memory and provide analytical tools to estimate the speedup from multithreaded execution for any query evaluation algorithm.
- We model the scheduling problem of assigning threads to query operators, in the context of intra-query parallelism on multicore processors. We then propose a family of heuristic-based scheduling algorithms that produce plans of high quality while incurring negligible optimisation cost.

6.2 Future directions

Certain future research directions can be identified. In Section 3.8.7 we saw that code generation is expensive when compared to the direct execution of operator-based query engines. This may penalise the response times of simple ad-hoc queries if there is no provision for caching the generated code. It is therefore important to modify code preparation in ways that will minimise the generation and compilation costs. To that end, it is possible to use the compiler to directly produce the shared library file from the SQL query. This will avoid the process of producing the C source file and compiling it, and is expected to cut down the query preparation cost. Moreover, due to the importance of compilation in the holistic query engine, compiler optimisation techniques should be adapted to data-intensive applications. This will contribute to improving both cache locality and pipelined execution, thus achieving a higher level of processor utilisation during query evaluation.

Our framework for multithreading focuses on intra-operator parallelism. The next step is to extend our approach to multi-query execution. A possible research direction is to combine inter- and intra-query parallelism to schedule operations and maximise processing throughput. The task is to exploit work and data sharing across queries under the constraints imposed by hardware resources. Also, it is interesting to see how our modelling methodology can be extended to estimate scalability when multiple queries compete for the cache capacity and the memory bandwidth.

We model the scheduling problem of intra-query parallelism based on the premise that the optimiser can accurately estimate the utility of multithreaded execution. It is interesting to extend our framework to account for misestimations in scalability estimations. The optimiser can then follow a more conservative approach by staying away from decisions that may prove seriously sub-optimal if the speedup has been erroneously estimated; it can choose a schedule with comparable performance but lower sensitivity to misestimations.

Appendix A

The TPC-H benchmark

The TPC-H benchmark is an established benchmark for DSS applications. We will briefly describe the employed schema and the statistics involved in the standard table population. The interested reader can refer to [73] for more details.

A.1 Schema

The database schema is depicted in Figure A.1 and consists of the following tables:

- PART
- SUPPLIER
- PARTSUPP
- REGION
- NATION
- CUSTOMER
- ORDERS
- LINEORDERS

The following list defines the structure (list of columns) of each table.

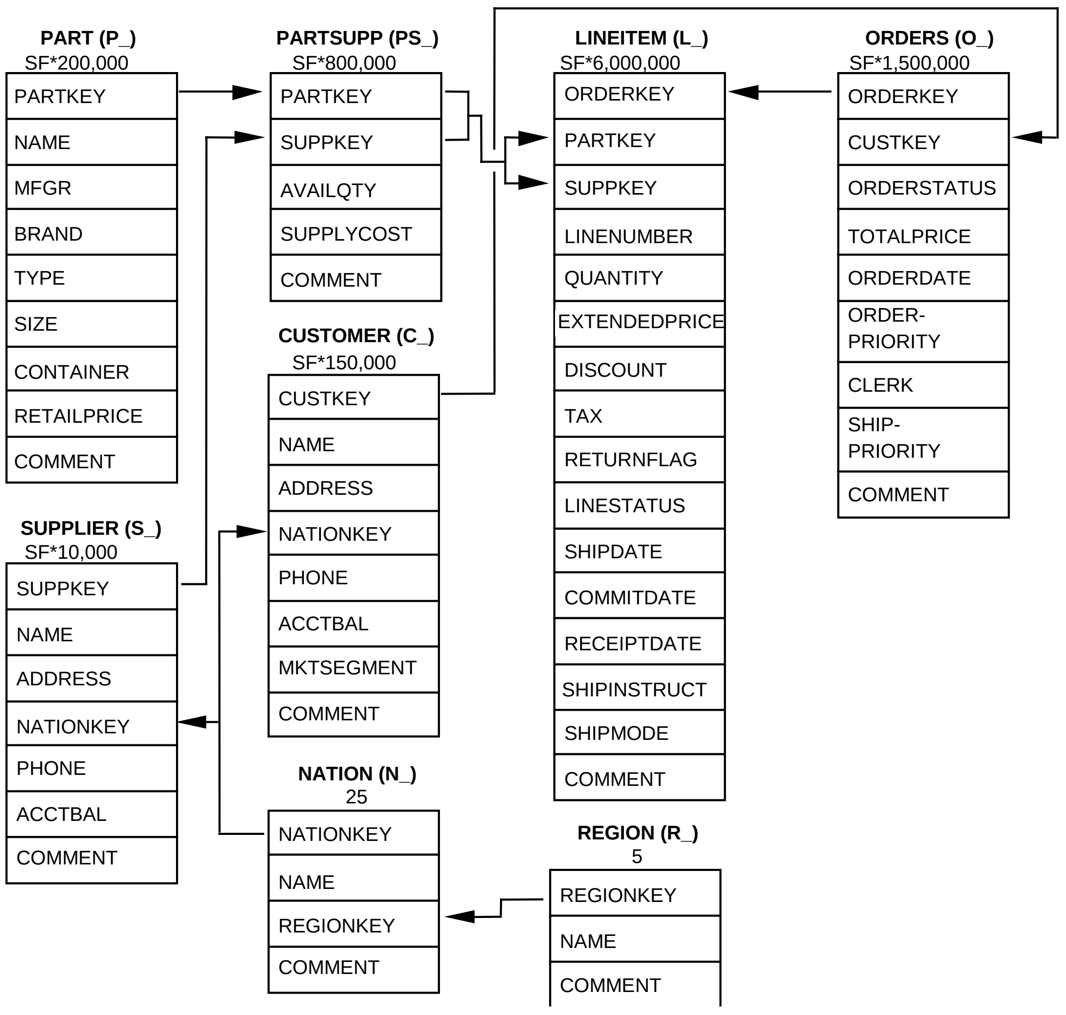


Figure A.1: The TPC-H schema

PART Table Layout

Column Name	*Datatype Requirements*	*Comment*
P_PARTKEY	identifier	SF*200,000 are populated
P_NAME	variable text, size 55	
P_MFGR	fixed text, size 25	
P_BRAND	fixed text, size 10	
P_TYPE	variable text, size 25	
P_SIZE	integer	
P_CONTAINER	fixed text, size 10	
P_RETAILPRICE	decimal	
P_COMMENT	variable text, size 23	

Primary Key: P_PARTKEY

SUPPLIER Table Layout

Column Name	*Datatype Requirements*	*Comment*
S_SUPPKEY	identifier	SF*10,000 are populated
S_NAME	fixed text, size 25	
S_ADDRESS	variable text, size 40	
S_NATIONKEY	Identifier	Foreign key reference to N_NATIONKEY
S_PHONE	fixed text, size 15	
S_ACCTBAL	decimal	
S_COMMENT	variable text, size 101	

Primary Key: S_SUPPKEY

PARTSUPP Table Layout

Column Name	*Datatype Requirements*	*Comment*
PS_PARTKEY	Identifier	Foreign key reference to P_PARTKEY
PS_SUPPKEY	Identifier	Foreign key reference to S_SUPPKEY
PS_AVAILQTY	integer	
PS_SUPPLYCOST	Decimal	
PS_COMMENT	variable text, size 199	

Compound Primary Key: PS_PARTKEY, PS_SUPPKEY

NATION Table Layout

Column Name	*Datatype Requirements*	*Comment*
N_NATIONKEY	identifier	25 nations are populated
N_NAME	fixed text, size 25	
N_REGIONKEY	identifier	Foreign key reference to R_REGIONKEY
N_COMMENT	variable text, size 152	

Primary Key: N_NATIONKEY

REGION Table Layout

Column Name	*Datatype Requirements*	*Comment*
R_REGIONKEY	identifier	5 regions are populated
R_NAME	fixed text, size 25	
R_COMMENT	variable text, size 152	

Primary Key: R_REGIONKEY

CUSTOMER Table Layout

Column Name	*Datatype Requirements*	*Comment*
C_CUSTKEY	Identifier	SF*150,000 are populated
C_NAME	variable text, size 25	
C_ADDRESS	variable text, size 40	
C_NATIONKEY	Identifier	Foreign key reference to N_NATIONKEY
C_PHONE	fixed text, size 15	
C_ACCTBAL	Decimal	
C_MKTSEGMENT	fixed text, size 10	
C_COMMENT	variable text, size 117	

Primary Key: C_CUSTKEY

ORDERS Table Layout

Column Name	*Datatype Requirements*	*Comment*
O_ORDERKEY	Identifier	SF*1,500,000 are sparsely populated
O_CUSTKEY	Identifier	Foreign key reference to C_CUSTKEY
O_ORDERSTATUS	fixed text, size 1	
O_TOTALPRICE	Decimal	
O_ORDERDATE	Date	
O_ORDERPRIORITY	fixed text, size 15	
O_CLERK	fixed text, size 15	
O_SHIPPRIORITY	Integer	
O_COMMENT	variable text, size 79	

Primary Key: O_ORDERKEY

LINEITEM Table Layout

Column Name	*Datatype Requirements*	*Comment*
L_ORDERKEY	identifier	Foreign key reference to O_ORDERKEY
L_PARTKEY	identifier	Foreign key reference to P_PARTKEY, Compound Foreign Key Reference to (PS_PARTKEY, PS_SUPPKEY) with L_SUPPKEY
L_SUPPKEY	Identifier	Foreign key reference to S_SUPPKEY, Compound Foreign key reference to (PS_PARTKEY, PS_SUPPKEY) with L_PARTKEY
L_LINENUMBER	integer	
L_QUANTITY	decimal	
L_EXTENDEDPRICE	decimal	
L_DISCOUNT	decimal	
L_RETURNFLAG	fixed text, size 1	
L_LINESTATUS	fixed text, size 1	
L_SHIPDATE	date	
L_COMMITDATE	date	
L_RECEIPTDATE	date	
L_SHIPINSTRUCT	fixed text, size 25	
L_SHIPMODE	fixed text, size 10	
L_COMMENT	variable text size 44	

Compound Primary Key: L_ORDERKEY, L_LINENUMBER

A.2 Statistics

The data generated by the standard benchmard dataset generator DBGEN is used to populate the database as follows (where SF is the scale factor):

SF * 10,000 rows in the **SUPPLIER** table with:

- S_SUPPKEY unique within [SF * 10,000].
- S_NAME text appended with digit [”Supplier”, S_SUPPKEY].
- S_ADDRESS random v-string[25].
- S_NATIONKEY random value [0 .. 24].
- S_PHONE int.
- S_ACCTBAL random value [-999.99 .. 9,999.99]
- S_COMMENT text string [63].
- SF * 5 rows are randomly selected to hold at a random position a string matching ”Customer%Complaints”. Another SF * 5 rows are randomly selected to hold at a random position a string matching ”Customer%Recommends”, where % is a wildcard that denotes zero or more characters.

SF * 200,000 rows in the **PART** table with:

- P_PARTKEY unique within [SF * 200,000].
- P_NAME generated by concatenating five unique randomly selected strings from the following list, separated by a single space: ”almond”, ”antique”, ”aquamarine”, ”azure”, ”beige”, ”bisque”, ”black”, ”blanched”, ”blue”, ”blush”, ”brown”, ”burlywood”, ”burnished”, ”chartreuse”, ”chiffon”, ”chocolate”, ”coral”, ”cornflower”, ”cornsilk”, ”cream”, ”cyan”, ”dark”, ”deep”, ”dim”, ”dodger”, ”drab”, ”firebrick”, ”floral”, ”forest”, ”frosted”, ”gainsboro”, ”ghost”, ”goldenrod”, ”green”, ”grey”, ”honeydew”, ”hot”, ”indian”, ”ivory”, ”khaki”, ”lace”, ”lavender”, ”lawn”, ”lemon”, ”light”, ”lime”, ”linen”, ”magenta”, ”maroon”, ”medium”, ”metallic”, ”midnight”, ”mint”, ”misty”, ”moccasin”, ”navajo”, ”navy”, ”olive”, ”orange”, ”orchid”, ”pale”, ”papaya”, ”peach”, ”peru”, ”pink”, ”plum”, ”powder”,

"puff", "purple", "red", "rose", "rosy", "royal", "saddle", "salmon", "sandy", "seashell", "sienna", "sky", "slate", "smoke", "snow", "spring", "steel", "tan", "thistle", "tomato", "turquoise", "violet", "wheat", "white", "yellow".

- P_MFGR text appended with digit ["Manufacturer",M], where M = random value [1,5].
- P_BRAND text appended with digit ["Brand",MN], where N = random value [1,5] and M is defined while generating P_MFGR.
- P_TYPE random string [Types].
- P_SIZE random value [1 .. 50].
- P_CONTAINER random string [Containers].
- P_RETAILPRICE = (90000 + ((P_PARTKEY/10) modulo 20001) + 100 * (P_PARTKEY modulo 1000))/100 P_COMMENT text string [14].

For each row in the **PART** table, four rows in **PARTSUPP** table with:

- PS_PARTKEY = P_PARTKEY.
- PS_SUPPKEY = (PS_PARTKEY + (i * ((S/4) + (int)(PS_PARTKEY-1)/S)))) modulo S + 1 where i is the ith supplier within [0 .. 3] and S = SF * 10,000.
- PS_AVAILQTY random value [1 .. 9,999].
- PS_SUPPLYCOST random value [1.00 .. 1,000.00].
- PS_COMMENT text string [124].

SF * 150,000 rows in **CUSTOMER** table with:

- C_CUSTKEY unique within [SF * 150,000].
- C_NAME text appended with digit ["Customer", C_CUSTKEY].
- C_ADDRESS random v-string [25].
- C_NATIONKEY random value [0 .. 24].
- C_PHONE int.

- C_ACCTBAL random value [-999.99 .. 9,999.99].
- C_MKTSEGMENT random string [Segments].
- C_COMMENT text string [73].

For each row in the **CUSTOMER** table, ten rows in the **ORDERS** table with:

- O_ORDERKEY unique within [SF * 1,500,000 * 4].
- O_CUSTKEY = random value [1 .. (SF * 150,000)]. The generation of this random value must be such that O_CUSTKEY modulo 3 is not zero.
- O_ORDERSTATUS set to the following value:
  ”F” if all lineitems of this order have L_LINESTATUS set to ”F”.
  ”O” if all lineitems of this order have L_LINESTATUS set to ”O”.
  ”P” otherwise.
- O_TOTALPRICE computed as:
  sum (L_EXTENDEDPRICE * (1+L_TAX) * (1-L_DISCOUNT)) for all LineItem of this order. O_ORDERDATE uniformly distributed between STARTDATE and (ENDDATE - 151 days).
- O_ORDERPRIORITY random string [Priorities].
- O_CLERK text appended with digit [”Clerk”, C] where C = random value [000000001 .. (SF * 1000)].
- O_SHIPPRIORITY set to 0.
- O_COMMENT text string [49].

For each row in the **ORDERS** table, a random number of rows within [1 .. 7] in the **LINEITEM** table with:

- L_ORDERKEY = O_ORDERKEY.
- L_PARTKEY random value [1 .. (SF * 200,000)].
- L_SUPPKEY = (L_PARTKEY + (i * ((S/4) + (int)(L_partkey-1)/S)))) modulo S + 1 where i is the corresponding supplier within [0 .. 3] and S = SF * 10,000.

- L_LINENUMBER unique within [7].
- L_QUANTITY random value [1 .. 50].
- L_EXTENDEDPRICE = L_QUANTITY * P_RETAILPRICE

  Where P_RETAILPRICE is from the part with P_PARTKEY = L_PARTKEY.
- L_DISCOUNT random value [0.00 .. 0.10].
- L_TAX random value [0.00 .. 0.08].
- L_RETURNFLAG set to a value selected as follows:

  If L_RECEIPTDATE ¡= CURRENTDATE

  then either ”R” or ”A” is selected at random

  else ”N” is selected.
- L_LINESTATUS set the following value:

  ”O” if L_SHIPDATE ¿ CURRENTDATE

  ”F” otherwise.
- L_SHIPDATE = O_ORDERDATE + random value [1 .. 121].
- L_COMMITDATE = O_ORDERDATE + random value [30 .. 90].
- L_RECEIPTDATE = O_ORDERDATE + random value [1 .. 30].
- L_SHIPINSTRUCT random string [Instructions].
- L_SHIPMODE random string [Modes].
- L_COMMENT text string [27].

25 rows in the **NATION** table with:

- N_NATIONKEY unique value between 0 and 24.
- N_NAME string from the following series of

  (N_NATIONKEY, N_NAME, N_REGIONKEY).

  (0, ALGERIA, 0);(1, ARGENTINA, 1);(2, BRAZIL, 1); (3, CANADA, 1);(4, EGYPT, 4);(5, ETHIOPIA, 0); (6, FRANCE, 3);(7, GERMANY, 3);(8, INDIA, 2); (9, INDONESIA, 2);(10, IRAN, 4);(11, IRAQ, 4); (12, JAPAN, 2);(13, JORDAN, 4);(14, KENYA, 0); (15, MOROCCO, 0);(16, MOZAMBIQUE, 0);(17, PERU, 1); (18, CHINA, 2);(19, ROMANIA, 3);(20, SAUDI ARABIA, 4); (21, VIETNAM, 2);(22, RUSSIA, 3);(23, UNITED KINGDOM, 3); (24, UNITED STATES, 1)

- N_REGIONKEY is taken from the series above.
- N_COMMENT text string [95].

5 rows in the **REGION** table with:

- R_REGIONKEY unique value between 0 and 4.
- R_NAME string from the following series of (R_REGIONKEY, R_NAME). (0, AFRICA);(1, AMERICA); (2, ASIA);(3, EUROPE);(4, MIDDLE EAST) R_COMMENT text string [95].

Bibliography

[1] R. Acker, C. Roth, and R. Bayer. Parallel Query Processing in Databases on Multicore Architectures. In *ICA3PP*, pages 2–13, 2008.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[4] AMD corporation. Software Optimization Guide for AMD64 Processors, 2005.

[5] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, 2002.

[6] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR '05*, 2005.

[7] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic Load Balancing in Hierarchical Parallel Database Systems. In *VLDB*, pages 436–447, 1996.

[8] L. Bouganim, D. Florescu, and P. Valduriez. Load Balancing for Parallel Query Execution on NUMA Multiprocessors. *Distributed and Parallel Databases*, 7(1):99–121, 1999.

[9] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[11] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 58–67, Washington, DC, USA, 1992. IEEE Computer Society.

[12] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance through Prefetching. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 116, Washington, DC, USA, 2004. IEEE Computer Society.

[13] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 817–828. VLDB Endowment, 2005.

[14] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 235–246, New York, NY, USA, 2001. ACM.

[15] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-Trees: optimizing both cache and disk performance. In *SIGMOD '02: Pro-*

ceedings of the 2002 ACM SIGMOD international conference on Management of data, pages 157–168, New York, NY, USA, 2002. ACM.

[16] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, 2008.

[17] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 339–350. VLDB Endowment, 2007.

[18] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, pages 25–34, 2008.

[19] J. Cieslewicz, K. A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

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

[21] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[22] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future, 1993.

[23] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB '1985: Proceedings of the 11th international conference on Very Large Data Bases*, pages 151–164. VLDB Endowment, 1985.

[24] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[25] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, 1992.

[26] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access, 2005. White paper.

[27] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discret. Math.*, 2(4):473–487, 1989.

[28] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 9–18, New York, NY, USA, 1992. ACM.

[29] P. Garcia and H. F. Korth. Database hash-join algorithms on multithreaded computer architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 241–252, New York, NY, USA, 2006. ACM.

[30] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.

[31] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional Resource Scheduling for Parallel Queries. In *SIGMOD Conference*, pages 365–376, 1996.

[32] M. N. Garofalakis and Y. E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *VLDB*, pages 296–305, 1997.

[33] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2), 1993.

[34] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[35] G. Graefe, R. Bunker, and S. Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 86–97, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[36] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.

[37] R. Greer. Daytona And The Fourth-Generation Language Cymbal. In *SIGMOD*, 1999.

[38] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1(1):188–200, 2008.

[39] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79–87, 2007.

[40] W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Rec.*, 25(3):28–33, 1996.

[41] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaumann Publishers Inc., 4 edition, 2006.

[42] Intel. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem), 2008. White paper.

[43] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2007.

[44] R. Johnson, N. Hardavellas, I. Pandis, N. Mancheril, S. Harizopoulos, K. Sabirli, A. Ailamaki, and B. Falsafi. To Share or Not To Share? In *VLDB*, 2007.

[45] A. Kemper, D. Kossmann, and C. Wiesner. Generalised Hash Teams for Join and Group-by. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 30–41, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[46] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., 2002.

[47] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[48] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and Its Architecture. *New Generation Comput.*, 1(1), 1983.

[49] D. E. Knuth. *The Art Of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[50] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot$^{\text{TM}}$client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[51] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *PVLDB*, 2(1):373–384, 2009.

[52] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 829–840. VLDB Endowment, 2005.

[53] M.-L. Lo, M.-S. S. Chen, C. V. Ravishankar, and P. S. Yu. On optimal processor allocation to support pipelined hash joins. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 69–78, New York, NY, USA, 1993. ACM.

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

[55] S. Manegold, P. A. Boncz, and M. L. Kersten. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 339–350, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[56] M. Mehta and D. J. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 382–394, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[57] T. Moscibroda and O. Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–18, Berkeley, CA, USA, 2007. USENIX Association.

[58] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: a cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–628, 1995.

[59] OProfile. A System Profiler for Linux, 2008. `http://oprofile.sourceforge.net/`.

[60] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proceedings of the 17th International Conference on Data Engineering*, pages 567–574, Washington, DC, USA, 2001. IEEE Computer Society.

[61] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo.

SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[62] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core CPUs. *Proc. VLDB Endow.*, 2008.

[63] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled Query Execution Engine using JVM. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 23, Washington, DC, USA, 2006. IEEE Computer Society.

[64] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *The VLDB Journal*, pages 78–89, 1999.

[65] RightMark. RightMark Memory Analyser, 2008. `http://cpu.rightmark.org/products/rmma.shtml`.

[66] D. A. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *VLDB*, 1990.

[67] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM.

[68] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[69] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[70] E. J. Shekita, H. C. Young, and K.-L. Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *VLDB*, pages 479–492, 1993.

[71] M. Stonebraker and U. Cetintemel. ”One size fits all”: an idea whose time has come and gone. In *ICDE ’05. Proceedings of the 21th International Conference on Data Engineering*, pages 2–11. IEEE Computer Society, 2005.

[72] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it’s time for a complete rewrite). In *VLDB ’07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[73] Transaction Processing Performance Council. The TPC-H benchmark, 2009. `http://www.tpc.org/tpch/`.

[74] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel evaluation of multi-join queries. *SIGMOD Rec.*, 24(2):115–126, 1995.

[75] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[76] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD Conference*, pages 425–436, 2001.

[77] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving Database Performance on Simultaneous Multithreading Processors. In *VLDB*, 2005.

[78] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD*, 2004.

[79] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *DaMoN ’06: Proceedings of the 2nd international workshop on Data management on new hardware*, page 6, New York, NY, USA, 2006. ACM.

[80] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 59, Washington, DC, USA, 2006. IEEE Computer Society.

[81] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware*, pages 47–54, New York, NY, USA, 2008. ACM.