In-Memory Parallel Join Processing on Multi-Core Processors

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

ÇAĞRI BALKESEN

Master of Science ETH in Computer Science, ETH Zurich

born on 20.12.1984

citizen of

Turkey

accepted on the recommendation of

Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Donald Kossmann, co-examiner
Prof. Dr. M. Tamer Özsu, co-examiner
Dr. Stefan Manegold, co-examiner

2014
Typeset with \LaTeX
© 2014 by Çağrı Balkesen
Abstract

During the last few years, there have been radical changes in the hardware landscape. Intel’s co-founder Gordon E. Moore’s famous conjecture that the number of transistors on integrated circuits doubles roughly every two years still holds today. However, multi-core processors caused a paradigm shift in hardware where increasing number of transistors are invested in parallelism and advanced features on chip designs. The game-changing technological trends constitute the driving force behind how new data processing systems and algorithms should be designed and tailored today.

The architectural changes introduced with multi-core processors have triggered a redesign of main-memory join algorithms for relational database systems. Join processing in database systems is a complex and a demanding operation. Traditionally, there have been sorting and hashing based approaches for implementing joins. However, in the last few years several diverging views have appeared regarding how to design and implement main-memory joins on multi-core processors.

On the one hand, there are two camps on how to implement main-memory hash joins on multi-core. Hardware-oblivious hash join variants do not depend on hardware-specific parameters. Rather, they consider qualitative characteristics of modern hardware and are expected to achieve good performance on any technologically similar platform. The assumption that these algorithms make is that hardware has now become good enough at hiding its own limitations—through automatic hardware prefetching, out-of-order execution or simultaneous multithreading (SMT)—to make hardware-oblivious algorithms competitive without the overhead of carefully tuning to the underlying hardware. Hardware-conscious implementations, such as (parallel) radix join, aim to maximally exploit a given architecture by tuning the algorithm parameters (e.g., hash table sizes) to the particular features of that architecture. The assumption here is that explicit parameter tuning yields enough performance advantages to warrant the effort required.

The thesis compares the two approaches for hash joins under a wide range of workloads (relative table sizes, tuple sizes, effects of sorted data, etc.) and configuration parameters (VM page sizes, number of threads, number of cores, SMT,
SIMD, prefetching, etc.). The results show that *hardware-conscious* algorithms generally outperform *hardware-obliviou*s ones. However, on specific workloads and special architectures with aggressive simultaneous multi-threading, *hardware-obliviou*s algorithms become competitive. As an answer to the controversy on how to implement hash joins on existing multi-core architectures, the thesis shows that it is still important to carefully tailor algorithms to the underlying hardware to get the necessary performance.

On the other hand, with the advent of modern multi-core architectures, it has been argued that sort-merge join is now a better choice than radix-hash join. This claim is justified based on the width of SIMD instructions (sort-merge outperforms radix-hash join once SIMD is sufficiently wide), and NUMA awareness (sort-merge is superior to hash join in NUMA architectures). Through extensive experiments on the original and optimized versions of these algorithms, we show that, contrary to these claims, radix-hash join is still clearly superior, and sort-merge approaches to performance of radix only when very large amounts of data are involved. Thus, the thesis resolves another controversy in the literature regarding the relative performance of sort and hash based join algorithms. The thesis provides the fastest implementations of these algorithms, and covers many aspects of modern hardware architectures relevant not only for joins but for any parallel data processing operator.
In den letzten Jahren ist es zu drastischen Veränderungen betreffend Hardware in
Computersystemen gekommen. Die berühmte Feststellung von Gordon E. Moore
(Mitglied der Firma Intel), dass sich die Anzahl Transistoren in elektronischen
Schaltungen etwa alle zwei Jahre verdoppelt, gilt auch heute noch. Mehrkern-
Prozessoren haben aber zu einem Paradigmenwechsel bei der Hardware geführt, da
immer mehr Transistoren für Parallelität und fortgeschrittene Funktionen einge-
setzt werden. Diese wegweisenden technologischen Trends begründen die treibende
Kraft für neue Datenverarbeitungssysteme und entsprechende Algorithmen.

Veränderungen in der Hardware-Architektur, die mit Mehrkern-Prozessoren
eingeführt wurden, haben zu neuen Ansätzen für Hauptspeicher-basierte Join-
Algorithmen in relationalen Datenbanken geführt. Die Berechnung eines Joins
ist eine komplexe und aufwändige Operation. Es gibt zwei klassische Vorge-
hensweisen, um eine Join Operation effizient zu implementieren: Die erste basiert
auf der Sortierung von Relationen, die andere auf Hashwert-Berechnung. Welche
Methode sich besser für Joins im Hauptspeicher eignet wurde in den letzten Jahren
oft diskutiert, ebenso wie geeignete Implementationsstrategien.

Bezüglich Hash-Join gibt es grundsätzlich zwei unterschiedliche Meinungen, wie
man im Hauptspeicher-Joins am besten implementiert. Ein Variante von Hash-
Joints kümmert sich nicht um Hardware-spezifische Parameter. Vielmehr werden
quantitative Eigenschaften moderner Hardware berücksichtigt, in der Hoffnung,
dass gute Performance auf vielen ähnlichen Plattformen erzielt werden kann. Die
zugrundeliegende Annahme ist, dass Hardware heutzutage ausgereift genug ist
und allfällige Einschränkungen meist gut umgangen werden können, zum Beispiel
durch automatisches “Prefetching”, “Out-of-order Execution”, oder “Simultaneous
Multi-threading (SMT)“, so dass es nicht nötig ist die Algorithmen sorgfältig für
die Plattform zu optimieren, und trotzdem gute Leistung erzielt werden kann.
Die andere Variante von Hash-Joins (zum Beispiel der sogenannte “Radix Join”),
versucht maximale Performance auf einer gegebenen Plattform zu erzielen, indem
gewisse Parameter (zum Beispiel die Größe der Hash-Tabelle) auf die jeweilige
Hardware-Eigenschaften abgestimmt werden. Hier ist die Annahme, dass sich
das gezielte Optimieren des Algorithmus auf eine Hardware-Plattform ausreichend lohnt, damit der damit verbundene Aufwand gerechtfertigt ist.

Diese Dissertation untersucht die zwei Ansätze für Hash-Joins für viele verschiedene Szenarien (relative Größe der Relationen, Tupel Größe, Effekte verursacht durch sortierte Daten, etc.) und Konfigurationsparameter (VM Page Größe, Anzahl Threads, Anzahl Prozessorkerne, SMT, SIMD, prefetching, etc.). Die Experimente zeigen, dass Algorithmen, welche gezielt Eigenschaften der Hardware-Plattform ausnutzen bessere Leistung aufweisen als andere Algorithmen. Dennoch, erzielen manchmal –unter gewissen Voraussetzungen– auch die Algorithmen, die die Hardware Eigenschaften nicht berücksichtigen, vergleichbare Leistung. Als Antwort auf die Frage, wie Hash-Joins am besten auf moderner Hardware implementiert werden sollen, zeigt diese Dissertation auf, dass es immer noch wichtig ist Algorithmen für die entsprechende Hardware zu optimieren.

I would like to express profound gratitude to my advisor Prof. Dr. Gustavo Alonso for his helpful and invaluable support, encouragement and supervision throughout my Ph.D. studies. I learned a lot from him and his continuous guidance helped me to complete my Ph.D. studies successfully. I would like to express my sincere gratitude to Dr. Nesime Tatbul for encouraging me to pursue a Ph.D. degree. I would like to thank Prof. Dr. Jens Teubner for his guidance and for being a great role model researcher to me.

I am thankful to Prof. Dr. M. Tamer Özsu for his great guidance, insightful comments and wisdom during the time we had the chance to work together. It was a great pleasure to learn from him. I also would like to thank Prof. Dr. Donald Kossmann and Dr. Stefan Manegold for being part of my Ph.D. exam committee.

Last but not least, I am as ever, especially indebted to my family for their love and support throughout my life. They supported me to follow my dreams with an endless source of morale and encouragement for me.
## Contents

<table>
<thead>
<tr>
<th>Abstract</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kurzfassung</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vii</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Motivation .............................................. 13
1.2 The Focus of the Thesis ................................. 15
1.3 The Challenges and Design Principles .................. 16
1.4 Thesis Contributions .................................. 19
1.5 Outline ................................................. 20

### 2 Modern Processor Architectures

2.1 Introduction .............................................. 23
2.2 Cache and the Memory Hierarchy .......................... 24
  2.2.1 Cache Internals ..................................... 25
  2.2.2 Cache and Memory Level Parallelism ................. 28
  2.2.3 Virtual Memory and Address Translation ............ 28
2.3 Instruction Execution and Instruction Level Parallelism .. 30
2.4 Vectorization and Data Level Parallelism ............... 31
2.5 Multi-core and Thread Level Parallelism ............... 33
  2.5.1 Locality and Memory Bandwidth ..................... 34
  2.5.2 Multi-core and Caches .............................. 35
  2.5.3 Synchronization ..................................... 35
  2.5.4 Simultaneous Multi-Threading (SMT) ................. 36
2.6 Implications on Data Processing ........................ 37

### 3 Hash Joins

3.1 Introduction .............................................. 39
3.2 In-Memory Hash Join Algorithms ........................ 41
  3.2.1 Canonical Hash Join Algorithm ....................... 42
# Contents

3.2.2 No Partitioning Join .............................................. 42
3.2.3 Radix Join .......................................................... 43
3.2.4 Parallel Radix Join ................................................ 45
3.3 Experimental Setup .................................................. 46
3.3.1 Workload ............................................................ 46
3.3.2 Hardware Platforms ............................................... 47
3.4 Hardware-Oblivious Hash Joins ................................. 48
3.4.1 Build Cost ......................................................... 48
3.4.2 Cache Efficiency .................................................. 50
3.4.3 The Role of SMT Threads ........................................ 51
3.5 Hardware-Conscious Hash Joins ............................... 52
3.5.1 Configuration Parameters ...................................... 52
3.5.2 Hash Tables and Cache Efficiency .......................... 53
3.5.3 Overall Execution Time ......................................... 55
3.5.4 Speedup from SMT Threads ..................................... 57
3.6 Hardware-Conscious or Not? ..................................... 57
3.6.1 Effect of Workloads .............................................. 58
3.6.2 Scalability .......................................................... 58
3.6.3 Sun UltraSPARC T2 “Niagara” ............................... 61
3.6.4 Barrier Synchronization and Load Balancing .............. 61
3.6.5 Skewed Data ........................................................ 64
3.6.6 Effect of Relation Size Ratio .................................. 64
3.6.7 TLB and Virtual Memory Page Sizes ........................ 66
3.6.8 Further Experiments on Large Virtual Memory Pages .... 69
3.6.9 Evaluation on Recent Multi-Core Servers .................. 71
3.6.10 When Does Hardware-Conscious Not Work Well? ........ 75
3.7 Further Hardware-Conscious Optimizations ................. 77
3.7.1 Making No Partitioning Hardware-Conscious .............. 77
3.7.2 Improving Radix with Software-Managed Buffers ........ 80
3.8 Estimating Ideal Parameters of Radix Join ................... 83
3.8.1 An Empirical Model for Configuring Radix Join .......... 83
3.8.2 Evaluation of the Empirical Model .......................... 86
3.8.3 Extending the Empirical Model for Optimizations ....... 89
3.9 Factorial Analysis of Performance ............................. 92
3.9.1 Analysis of Machines, Workloads and Algorithms ....... 92
3.9.2 Analysis of Optimizations on No Partitioning Join ....... 95
3.10 Related Work .......................................................... 99
3.11 Conclusion ............................................................. 100
## 4 Sort-Merge Joins

4.1 Introduction ............................................. 103
4.2 Background and Related Work ................................. 104
   4.2.1 Canonical Sort-Merge Join Algorithm ................ 104
   4.2.2 Hardware-Assisted Sorting ............................ 105
   4.2.3 Sort-Merge Joins .................................... 106
   4.2.4 The Role of NUMA ................................ 107
4.3 Parallelizing Sort with SIMD ............................... 107
   4.3.1 Run Generation .................................. 107
   4.3.2 Merging Sorted Runs ................................ 109
   4.3.3 Implementation Details ............................... 111
4.4 Cache Conscious Sort Joins ................................ 114
   4.4.1 Sorting and the Memory Hierarchy ..................... 114
   4.4.2 Balancing Computation and Bandwidth ................. 115
   4.4.3 Implementation of Multi-way Merge .................... 117
4.5 Efficient Data Partitioning ................................ 119
   4.5.1 Radix Partitioning ................................ 119
   4.5.2 Software-Managed Buffers ............................ 119
4.6 Sort-Merge Join Algorithms ................................. 120
   4.6.1 Sort-Merge Join Algorithm – \textit{m-way} ............ 120
   4.6.2 Sort-Merge Join Algorithm – \textit{m-pass} ............. 122
   4.6.3 Massively Parallel Sort-Merge Join – \textit{mpsm} ....... 122
4.7 Experimental Setup ....................................... 123
   4.7.1 Workloads ........................................ 123
   4.7.2 System ............................................. 123
4.8 Analysis of the Sort Phase ................................ 124
   4.8.1 Raw Sorting Performance ............................. 124
   4.8.2 Comparison with other Algorithms .................... 125
4.9 Analysis of the Merge Phase ................................ 126
   4.9.1 Modeling the Merge Phase ............................ 126
   4.9.2 Performance of the Merge Phase ....................... 126
4.10 Optimizing the Merge Phase ............................... 128
   4.10.1 Performance of the Partitioning Phase ............... 128
   4.10.2 Using Partitioning with Sort ........................ 130
   4.10.3 Alternative Implementations for Merge .............. 131
4.11 Evaluation of Sort-Merge Joins ............................ 132
   4.11.1 Comparison with Different Table Sizes .............. 132
   4.11.2 Execution Break Down and Throughput ............... 133
   4.11.3 Dissecting the Speedup of \textit{m-way} ............... 135
   4.11.4 Scalability of Sort-based Join Algorithms .......... 135
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.11.5</td>
<td>Scalar Sorting-based Join Algorithms</td>
<td>137</td>
</tr>
<tr>
<td>4.11.6</td>
<td>NUMA Characteristics and Efficiency</td>
<td>138</td>
</tr>
<tr>
<td>4.12</td>
<td>Summary</td>
<td>141</td>
</tr>
<tr>
<td>5</td>
<td>Sort vs. Hash Revisited</td>
<td>143</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>143</td>
</tr>
<tr>
<td>5.2</td>
<td>Related Work</td>
<td>145</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Sort vs. Hash—Early Work</td>
<td>145</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Sort vs. Hash—Multi-core Era</td>
<td>145</td>
</tr>
<tr>
<td>5.3</td>
<td>Hash Join Algorithms</td>
<td>146</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Radix Hash Join—radix</td>
<td>146</td>
</tr>
<tr>
<td>5.3.2</td>
<td>No-Partitioning Hash Join—n-part</td>
<td>147</td>
</tr>
<tr>
<td>5.4</td>
<td>Experimental Setup</td>
<td>147</td>
</tr>
<tr>
<td>5.5</td>
<td>Sort or Hash?</td>
<td>147</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Comparison with Different Workloads</td>
<td>147</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Effect of Input Size</td>
<td>149</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Sort vs. Hash, 20 Years Later</td>
<td>149</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Effect of Skew</td>
<td>150</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Scalability Comparison</td>
<td>152</td>
</tr>
<tr>
<td>5.5.6</td>
<td>Sort vs. Hash with All Algorithms</td>
<td>154</td>
</tr>
<tr>
<td>5.5.7</td>
<td>Effect of Relative Table Sizes</td>
<td>154</td>
</tr>
<tr>
<td>5.5.8</td>
<td>Sort vs. Hash for other Database Algorithms</td>
<td>156</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusions</td>
<td>158</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>159</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
<td>159</td>
</tr>
<tr>
<td>6.2</td>
<td>Directions for Future Work</td>
<td>161</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>165</td>
</tr>
</tbody>
</table>
Introduction

1.1 Motivation

Nowadays the world relies more and more on computer-based techniques to analyze, identify and extract information from data. Nearly all of the major business activities of enterprises such as reporting, online analytical processing, complex event processing or information storage and retrieval are carried out through complex information systems. At the same time, the data volume pouring into the information systems of enterprises is increasing at an unprecedented rate. As the world becomes more dependent on processing and extracting meaningful information from massive data volumes, it is apparent that new techniques are needed to enable processing of future data volumes. Essentially, data processing systems (e.g., databases) and algorithms (e.g., relational joins) needs an architectural rethinking and redesign for data volumes that have not been considered so far.

The requirement for redesigning and rethinking the architecture of data processing systems is not only driven by the data volume growth. During the last few years, there has been a hardware revolution going on at the same time. Although it has been a long while since Intel’s co-founder Gordon E. Moore stated his famous conjecture that the number of transistors on integrated circuits doubles roughly every two years (Moore’s law) [Moo65], the trend is likely to continue for many years to come—maybe only with a little slowdown. Increasing number of transistors on integrated circuits are driving the hardware revolution and hardware is becoming more advanced and complex. In this respect, there are number of game-
changing technological trends that constitute the driving force behind how new
data processing systems and algorithms should be designed and tailored:

**Proliferation of Large Main-Memory**

The economics of the memory technology made it possible to increase memory
capacity over the past years at a reduced cost. Looking at the historical price
per bit trends for memory, one can observe a yearly 35% reduction in memory
costs. While the memory was still expensive a few years ago, nowadays it is not
expensive any more.

Latest servers from vendors started to come with multiple terabytes of main-
memory. For instance, Oracle Exadata Database Machine X3-8 [Ora14] provides
up to 4 terabytes of memory, whereas a further extreme example system by IBM
and SAP was demonstrated to run on top of 100 terabytes of memory [IBM14]. The
trend is likely to make even commodity servers equipped with multiple terabytes
of memory in the next few years.

On the other hand, existing database systems were designed by assuming that
the data would be stored on disk. The algorithms and data structures were ex-
plicitly targeted for disk storage and the related problems associated with data
transfer between the disk and the CPU. Today, it is even possible to keep entire
databases in main-memory. The assumptions behind the design of the existing
database systems need to be revised and new systems should be developed as-
suming a completely in-memory computing scenario. The proliferation of large
main-memory is likely to be a game-changer for data processing technologies.

**Advent of Parallelism with Multi-core Processors**

It has been a while since the sequential processor designs hit the diminishing re-
turns in performance gains. Therefore, the focus of today’s technology is providing
parallelism at the hardware level. Thanks to Moore’s law, more and more transis-
tors are used aggressively for increasing parallelism in chip designs. The paradigm
shift in hardware started the multi-core era for processors. New processors pro-
vide multiple number of physically replicated CPU cores in a single chip. Once
perceived as an esoteric art, the parallel computing has now become mainstream.

As current database engine architectures are not designed for the modern hard-
ware, these dramatic advances in hardware require rethinking the architecture of
the engines. Hence, the multi-core era requires exploring parallelism and operator
execution within modern processor architectures. In order to cope with the data
explosion, data management systems must benefit from parallelism by dividing
problems into smaller parts and execute those parts on different processor cores
provided by the multi-core era.
1.2 The Focus of the Thesis

Advanced Hardware Features

Increasing number of transistors are not only invested in the multi-core parallelism. Along the leap towards multi-core, modern processors are also becoming seemingly complex. Sophisticated execution and caching mechanisms are being developed. Nowadays, most of the real estate in chips are invested in fast on chip caches that amplify the performance of the memory subsystem. At the same time, execution units are becoming extremely advanced. CPUs can dynamically reorder execution of instructions and extract parallelism from the sequential instruction streams. Even control and memory access patterns can be predicted and speculated to a certain degree to improve performance.

As a result, modern processors provide parallelism at more levels than just multi-core; e.g., instruction parallelism via super scalar execution, data-level parallelism by extended support for single instruction over multiple data (i.e. SIMD, 128-bits; AVX, 256-bits) and thread-level parallelism through simultaneous multithreading (SMT). All these advanced architectural features are waiting to be explored by new algorithms and systems to extract performance from every bit of the hardware.

This thesis focuses on how data processing algorithms – namely relational database joins – should be designed and implemented in order to benefit from game-changing recent technological trends of modern processor architectures. Relational database join is one of the most important database operators and appears frequently in database query workloads. Moreover, joins are commonly accepted to be both data and computation intensive and usually dominate the execution costs in query workloads. Therefore, by focusing on the important relational database join operator on modern processor architectures, this thesis takes an important step towards rethinking and redesigning the architecture of database engines.

The thesis visits several promising approaches to in-memory join processing on modern multi-core processors. First, different algorithmic approaches for join processing with their pros and cons are described in detail. While doing so, our focus is to provide important insights for the design of data processing algorithms on modern processor architectures for achieving performance expectations. The evaluation of algorithms on plenty of different multi-core processors provide an understanding of the effects of multi-core on algorithm design and constitutes a reference for implementing other database operators as well as rethinking the architecture of the database engines.
Since joins are complex and demanding operations, there have been several important related works on joins in the literature. However, the landscape in join processing is inconclusive. The related work have opposing views on how to implement main-memory joins on multi-core most efficiently. One of the main discussions is on whether hash joins should be fine tuned to the underlying hardware or not. One line of thought claims that hardware became good enough so that the tuning is not required. Another camp claims that the best performance can be only achieved by being hardware conscious. On another aspect, there have been controversies whether the sort or hash based join is a better option given the architectural advances. By discussing all of the existing algorithms and claims in depth on recent multi-core processors, the thesis provides clear answers to the controversies existing in the literature.

1.3 The Challenges and Design Principles

Through the study of in-memory relational joins on modern processors, the thesis identifies number of challenges and design principles for achieving the best performance. These challenges and principles essentially convey the message and the findings of this thesis and are also relevant for rethinking the design and architecture of database engines and other algorithms on multi-core processors.

End of Free Lunch

Until the recent past few years, orders of magnitude performance increase in program execution was achieved by only means of either improvements in technology (frequency scaling, more logic gates per processors, etc.) or improvements in hardware architectures (RISC-based machines, instruction-level parallelism, large caches, etc.). During this era, programmers enjoyed a luxury and a comfort of obtaining performance by the push of a button (i.e., by simply running their program on faster computers).

Moreover, all of these advances enabled programmers to have more freedom in how they programmed their software. The productivity side of programming has gained popularity by programming languages such as Java, Phyton, C#, Scala etc. instead of languages such as Assembly, C, C++. For instance, the cost of overheads in object-oriented programming technology was deemed acceptable thanks to the advances in processor speeds.

However, the trend seems to fade away. The performance of single-processors no longer maintain the acceleration that has been achieved for a long time. Therefore, programmers accustomed to getting increased performance with each generation of the processors will not be able to get performance for free anymore.
1.3. The Challenges and Design Principles

There is a pressing need for a better understanding of the underlying processor architectures and utilizing the microarchitectural features for achieving better performance. More effort needs to be invested in tailoring the software design and programming to the underlying hardware.

The thesis demonstrates this phenomenon through the study of in-memory joins. The join algorithms that are hardware-conscious perform better than hardware-oblivious ones that rely on hardware alone to become better. In most of the cases, tuning main-memory join algorithms to the underlying hardware yields performance advantages that warrant the effort required during tuning.

Paradigm Shift in Achieving Performance

The performance gains in processor speeds usually have been through achieving better instruction level parallelism (ILP) starting with the RISC-style machines and continuing with the superscalar multi-issue execution. However, this is also over. The limits of sequential program improvement have been reached.

In 2004, Intel canceled its high-performance single-processor project and changed its direction to a whole new paradigm of attaining performance. It has been observed that heat and power dissipation from single-processors and the lack of ILP would not allow processors to become faster. Therefore, a paradigm shift has occurred with multi-core. The shift towards multi-core was not merely based on a breakthrough, but was viewed as a workaround for challenge in building power-efficient, high-clock rate, single-core chips [ABD+09]. Major chipmakers opted to pursue developing multiple processors per chip, where effort and real-estate on chips are invested in increasing thread level parallelism (TLP) and data level parallelism (DLP).

In the past there have been approaches to introduce parallel computers. However, all failed. The reason was with every generation of the processors, faster performance was achieved automatically without changing any piece of code. Therefore, programmers did not see a need to explore parallelism. Moreover, parallel programming was always perceived as complicated and hard to understand. Due to these perceptions and challenges, parallel programming was never widely adapted.

As we show in this thesis, parallelism is an important facet in achieving high performance in modern multi-core processors. In case of main-memory joins, almost a linear scalability in the number of cores can be achieved by careful implementation of parallel variants of the algorithms. Therefore, the parallelism at hardware-level shines to be the major way of attaining performance in multi-core processors. New database algorithms need to be designed to utilize this parallelism and parallel programming should go mainstream for data processing.
Chapter 1. Introduction

Every Hardware is a Different Story

In the past, hardware architecture of conventional processors used to bear many similarities. The amount of knowledge that the programmer needed to know about the underlying hardware was minimal. Accordingly, programmers were accustomed to the principle of “program once, run everywhere” with nominal performance difference and anomalies on different hardware. Today, this principle does not work anymore, especially for performance critical data processing tasks.

Multi-core era has not introduced a commonly accepted form of processor design yet. Processors interpreting the multi-core in different forms of style are abundant. While some processors contain symmetric, one-to-one replicated CPU cores; others opt for asymmetric designs where small and large CPU cores exist in an heterogeneous multi-core architecture [ARM14]. The diversity is also prevalent in architectural features. Various forms of memory/cache hierarchy, instruction execution, vectorization and non-uniform memory access (NUMA) architectures exist in different processors.

The diversity in the underlying hardware and its architectural features make it more challenging than ever to understand, model and estimate the performance of algorithms and systems on different processors. An algorithm running on two different processors is very likely to show different performance behavior due to the differences in the underlying architecture. Essentially, each hardware becomes a different story on its own.

The thesis highlights this fact through an extensive set of experiments ran on a diverse set of hardware platforms. A variant of main-memory hash join algorithm not performing well on a certain x86 architecture, performs surprisingly well on a radically different Sparc architecture. The architectural features such as cache sizes, fine-grained multi-threaded execution etc. play an important role to change the picture among those different processors. Moreover, each hardware requires fine tuning of algorithms to its underlying architectural features and parameters. By tuning, algorithms perform significantly better on each hardware platform.

Furthermore, hardware features play an important role and introduce significant constants which are usually ignored in theoretical complexity estimations. The thesis also provides insights from the complexity perspective. Different algorithmic approaches for in-memory joins –namely hash-based and sort-based– have different theoretical complexities. By just looking at complexities, one might tend to rule out the algorithm with the higher complexity (i.e., sort-based joins). However, as shown in the thesis, hardware details make this comparison inadequate. Under certain conditions, sort-based join algorithms can reach the performance of hash-based counterparts by utilizing hardware features such as large SIMD vectorization and NUMA-awareness. Therefore, algorithmic complexity needs to be enhanced with an understanding of the hardware complexity.
1.4 Thesis Contributions

In this dissertation, main-memory join algorithms on modern processor architectures are studied in detail. The thesis assumes a completely in-memory processing environment which is envisioned as the future of database engine architectures. In summary, the thesis make the following contributions:

**Algorithmic Contributions.** A set of contributions of the thesis can be categorized as algorithmic. First, main-memory hash join algorithms proposed in the literature are analyzed. We then propose several new techniques and important optimizations leading to new algorithms that are more efficient and robust to parameter changes. Second, main-memory sort-merge join algorithms proposed in the literature are analyzed. We propose efficient sorting techniques utilizing large SIMD vectorization units on modern hardware. We then introduce novel sort-based join algorithms that are tailored to the underlying hardware in terms of memory bandwidth usage and hardware peculiarities such as NUMA. The new algorithms are more efficient and aware of the underlying architecture and achieves better performance than the existing ones as shown in the thesis.

**Effects of Multi-Core on Algorithm Design.** While analyzing existing algorithms and proposing new ones, the thesis provides important insights on the effects of multi-core hardware and architectural features on algorithm design. There are many hardware parameters that affect the behavior of joins on modern processor architectures: cache sizes, cache sharing, TLB sizes, page sizes, SIMD size, NUMA topology, use of SMT, synchronization primitives, etc. The thesis explores these parameters in a systematic way and demonstrates the impact of these parameters on the design of algorithms for multi-core. The results shed light on what parameters play a role in multi-core systems, thereby establishing the basis for the choices a query optimizer for multi-core will need to make.

**Efficient Implementation of Main-Memory Join Algorithms.** Through optimizations and tuning of the proposed algorithms to modern processor architectures, the thesis provides some of the fastest in-memory join algorithm implementations. The implementations of our algorithms are significantly faster than the existing ones in the literature for both hash and sort-merge joins, demonstrating how to use modern processors to improve the performance of data operators. We also provide the source code of the algorithms publicly in the hope of being a reference point for design and implementation of other database algorithms and operators on modern hardware.

**Hardware-Conscious or Not?** The study of the main-memory hash join algorithm in this thesis answers an important question that appeared in the literature.
The hardware-conscious approach advocates careful tailoring of the algorithm to the architectural parameters (cache sizes, TLB and memory bandwidth). The hardware-oblivious approach argues that modern hardware is good enough at hiding cache and TLB miss latencies and, consequently, the careful tailoring can be omitted without sacrificing performance. The question of whether tuning to the underlying hardware by hardware-conscious algorithms plays a role in performance is answered in this thesis. The thesis demonstrates through an extensive experimental analysis of algorithms and architectures that hardware still matters. Hash join algorithms that are hardware-conscious perform generally better than hardware-oblivious approaches.

**Sort or Hash?** The relative performance of different algorithmic approaches – namely sort-based and hash-based joins – for implementing relational joins have been a topic of discussion for a long time. The thesis experimentally studies the performance of main-memory, parallel, multi-core joins with both sort-merge and (radix-)hash join variants. With the advent of modern multi-core architectures, it has been argued that sort-merge join is now a better choice than radix-hash join. This claim is justified based on the width of SIMD instructions (sort-merge outperforms radix-hash join once SIMD is sufficiently wide), and NUMA awareness (sort-merge is superior to hash join in NUMA architectures). In the thesis, we conduct extensive experiments on the original and optimized versions of these algorithms. The thesis shows that, contrary to existing claims in the literature, radix-hash join is still clearly superior, and sort-merge approaches to performance of radix only when very large amounts of data are involved.

### 1.5 Outline

The dissertation is structured as follows:

**Chapter 2** provides an overview of technological trends in current and emerging processor architectures and highlights the implications that these trends have on data processing systems. We discuss important hardware features such as CPU architectures, cache hierarchies, memory characteristics, advanced instruction execution and various forms of parallelism abundant in modern processors. The overall goal of the chapter is to give a background and convey an understanding of modern processor architectures.

**Chapter 3** is devoted to the main-memory hash join algorithms. The chapter begins by first introducing main-memory hash join algorithm variants. Hardware-conscious and hardware-oblivious hash joins are then discussed in detail with accompanying experiments. Through the illumination of the experiments that com-
1.5. Outline

Compare existing and new ones, the impact of optimizations and several hardware issues are discussed. Finally, the chapter evaluates the hardware-conscious and hardware-oblivious algorithms side-by-side in an extensive set of experiments that cover many aspects of algorithms and hardware on a diverse set of hardware platforms.

Chapter 4 explores sort-based algorithmic approaches for main-memory joins. The chapter describes parallel execution of sort-merge joins on modern multicore machines in a bottom up manner starting from register level SIMD sorting. The design choices for different phases, namely parallel sorting and merging, are described with accompanying experiments. The choices made among the options are justified with the behavior of algorithms on modern multi-core processors. After discussing parallel sorting and merging as the building blocks, the chapter goes on to discuss overall sort-merge join algorithms that are both tailored for the memory architectures and SIMD on multi-core CPUs.

Chapter 5 resolves the classic sort vs. hash controversy for in-memory joins on modern multi-core CPUs. First, the behavior of sort-merge join algorithms introduced in the thesis are evaluated. Afterwards, highly optimized (radix-)hash join and sort-merge join algorithms introduced in the thesis are evaluated side-by-side under a variety of experimental parameters. The chapter concludes that radix-hash join is still superior to sort-merge join in general.

Chapter 6 The chapter first takes a look at the contributions and results obtained in the thesis. The findings are summarized and conclusions are listed. Lastly, the thesis discusses possible future work directions and open problems.

The following published papers constitute the preliminary and condensed form of the material presented in this thesis:


Chapter 1. Introduction
2

Modern Processor Architectures

2.1 Introduction

“Memory is the new disk, disk is the new tape.” Attributed to Jim Gray, this phrase stresses the importance of main-memory in modern computing architectures as well as in database workloads where most of the work is typically data-intensive.

Advanced modern processor features, such as vast parallelism through multiple cores, multi-level caches, enhanced instruction sets (e.g., SIMD), virtual memory support, or multi-socket system architectures improve performance and functionality of systems executing on top and mitigate some of the limitations that come with memory technology. But they also cause new problems that are often even harder to model than, say, a memory hierarchy alone.

As a consequence of Moores Law, and through a series of advanced hardware features (high clock speeds, multi-issue execution units, extended instruction sets, hardware parallelism, . . . ), today’s processor designs have reached spectacular nominal performance rates. But despite all efforts, these advances have only widened, not closed, the performance gap between CPU and memory. The only promising way to escape the resulting bottlenecks is to adapt software algorithms to better match the characteristics of modern computing hardware.

The complexity of real-world systems, unfortunately, makes such an adaptation everything but trivial. A series of features and hardware details interacts and tries to work around one another, making it harder and harder to understand, model, and exploit the overall system behavior.
Chapter 2. Modern Processor Architectures

In this chapter, we try to convey such an understanding. We provide an overview of technological trends in current and emerging processor architectures and highlight the implications that these trends have on data processing systems. By visiting the modern architectural features of processors, we emphasize how data processing algorithms should be designed, so hardware evolution will become their friend, not their foe. We discuss important hardware features such as CPU architectures, cache hierarchies, memory characteristics, advanced instruction execution and various forms of parallelism abundant in modern processors waiting to be exploited by data processing algorithms.

Cache and Memory Hierarchies. The first part of the chapter covers the memory hierarchy in modern computing systems. Multiple levels of caches try to hide the inherent access latency of DRAM, which forms the bottom of the hierarchy. As a consequence, the access pattern to main memory has dramatic impact, up to orders of magnitude, on the observed memory performance. Cache-aware algorithms thus should avoid random access to memory wherever possible, or at least keep them constrained to a region that does not exceed the sizes of CPU caches.

Instruction Execution and Advanced Instruction Sets. First, we revise the inner workings of the CPU and various forms of parallelism and advanced techniques for executing instructions. Then, we cover vectorized execution through data parallelism. Extensions in CPU instruction sets, such as vectorized processing (SIMD) or hints to the CPU caching system, allow tailor-made software to better leverage the capabilities of modern hardware.

Multi-core Parallelism. Lastly, we cover thread level parallelism provided by the multi-core revolution. In addition to revising issues that affect performance on multi-core, we also highlight the multi-threading technology.

Much of the material presented in this chapter is based on the contents of the lecture “Data Processing on Modern Hardware” that was taught by Jens Teubner and Cagri Balkesen at ETH Zurich. In addition, more detailed material on the topics discussed in this chapter can be found in “Computer Architecture: A Quantitative Approach” by Hennessy and Patterson [HP11].

2.2 Cache and the Memory Hierarchy

In the early years of the computers, the CPU and the memory was in the same order of speed. However, over the years technological trends allowed CPUs to attain higher speeds while memory speed only nominally increased. The focus of the memory technology has been providing large capacities at a certain cost efficiency. Therefore, there has been an increasing gap between CPU and memory speeds. This gap has been commonly named as the “memory wall”. In terms of
the software running on these processors, this had impact on performance where
CPU instructions are exposed to long stalls for memory accesses and CPUs spent
most of the time waiting for the memory.

On the memory side, while it is possible to create faster but small memory based
on static RAM (SRAM) technology, it has not been economically just feasible
to create larger SRAM based memory. On the other hand, with the relatively
cheaper dynamic RAM (DRAM) technology, it was possible to create larger and
larger memory capacities at same or reduced costs over the years. The increasing
speed gap between memory and CPU speeds in addition to the economical and
technological trends in memory technology required computer architects to design
a memory hierarchy for modern processors. The fast memory that is small in
capacity but expensive in cost took place in the top of the hierarchy in the form of
CPU registers where access latencies stay below a nanosecond. In the next layer
of the hierarchy, first level (L1) caches implemented with SRAM technology took
place. The size of L1 caches are commonly in the order of kilobytes and access
latency is in the order of a nanosecond. In the subsequent layers, modern processors
commonly have L2 and L3 caches where capacities range from a megabyte to a
few ten megabytes and access latencies range from five to twenty nanoseconds.
Moreover, upcoming generations of modern processor architectures are likely to
provide another level of cache (L4) to close the ever widening gap to memory
[Int13, Ana13].

Due to the principle of locality, the cache or the memory hierarchy described
above usually works well on modern processors. In temporal locality programs
usually access data items which are temporally local, meaning that the reuse of a
data item over the time is quite common. In spatial locality, other data items in
spatial closeness to the recently accessed items are likely to be accessed. In most
of the programs, usually relatively small subset of the data (10\%) is accessed at
most of the time (90\%). Therefore, a hot subset of the data often fits into caches
thereby increasing the effectiveness of the cache hierarchy.

2.2.1 Cache Internals

We first revise the inner workings of CPU caches. Caches are organized as blocks
of data transfer, also called as “cache lines”. Lines should ideally be large enough
to compensate the overhead of the cache operation and be small enough to elimi-
nate unnecessary data movement. Nowadays, typical cache line sizes are 64-bytes.
Block-based organization supports the principle of locality.

On a typical memory access, the CPU first checks the nearest cache level for
the requested address. If the data exists in the cache, the data is provided from
the cache to the CPU and it is called a “cache hit”. However, if the requested data
does not exist, it becomes a “cache miss”. In this case, data is requested from the
Chapter 2. Modern Processor Architectures

next level of the memory hierarchy. Unfortunately, the CPU needs to stall until the data becomes available. Modern processors try to eliminate this problem by allowing multiple number of cache misses in-flight through out-of-order execution. Once the data is available, it is also inserted into the upper level cache by evicting some existing line.

**Block Placement and Associativity**

The retrieved block from the lower level can be placed into different locations depending on the cache organization. Caches can be organized in three different ways as shown in Figure 2.1.
2.2. Cache and the Memory Hierarchy

Fully Associative Cache. Figure 2.1(a) shows an example of a fully associative cache organization. In this scheme, a retrieved block can be inserted into any of the available cache lines. This scheme offers a flexibility in cache line evictions and utilizes the cache space optimally. However, it comes with the cost of expensive searches. During the search, a given address needs to be checked with all the cache lines in the cache and this introduces an inherent latency problem at the logic gate level. Due to this problem, fully associative caches are only practical for small caches and does not scale for larger caches.

Direct Mapped Cache. Figure 2.1(b) shows an example of a direct mapped cache organization. In this scheme, a retrieved block can only be inserted into a specific single cache line. This schemes turns out to be very simple to implement and can achieve very low latency of access. However, the downside of this approach is the increasing risk of conflicts and below optimal usage of the cache space.

Set Associative Cache. Figure 2.1(c) shows an example of a set associative cache organization with two-ways or four sets. In this scheme, a retrieved block can only be inserted into a specific cache set. However, it can be inserted into any of the cache lines within the specific set. The associativity is determined by the number of cache lines within each set. This scheme balances the trade-offs of the two approaches described above. Moreover, most of the processor caches today are implemented as a set associative cache with varying associativities.

Cache Block Addressing and Replacement

Caches need to store additional meta-data to identify blocks in the cache during cache probes. Figure 2.2(a) depicts typical cache contents. The valid bit identifies whether the given cache line is a valid data item retrieved from the memory. The dirty bit determines whether the data in cache has been modified since it was loaded into the cache. In associative caches, different data items can map to a given cache line. Therefore, during probes each cache line needs to be checked against the probed item to see if it is indeed the searched item. This check is carried out by comparing the tag field in the cache with the tag value extracted
from the given memory address. Finally, the data field stores the actual data item (i.e., cache lines) in the cache.

The addressing scheme for the cache works as follows. First, since the memory is byte addressable, each byte in the cache line needs to be addressed. In a given memory address, $\log_2(\text{cache line size})$ least significant bits are designated to address each byte in the cache line (shown as “offset” in Figure 2.2(b)). To identify the cache set of a block, next $\log_2(\text{number of sets})$ are reserved as set index bits. Finally, rest of the memory address bits are used as tag to identify individual cache lines.

When bringing in a new cache line into the cache, usually a cache line needs to be evicted from the cache to open up space for the new one. The strategy to select which line to evict is usually in the form of a least recently used (LRU). However, modern processors usually choose to implement a pseudo-LRU strategy in hardware as a fast approximation of LRU.

### 2.2.2 Cache and Memory Level Parallelism

Modern processor architectures have recently started exploiting memory level parallelism to achieve a better memory-subsystem performance [QLMP06]. Essentially, the parallelism for the cache/memory hierarchy is supported first by designing non-blocking caches which allows multiple number of cache-misses in-flight. This mechanism enables partly to hide miss latencies by overlapping multiple number of misses. Modern processors are quite successful at hiding L1 miss penalty by exploiting these mechanisms. To illustrate, Intel Sandy Bridge microarchitecture can manage up to 10 requests of outstanding cache misses simultaneously using the line fill buffers. In addition, by the support of out-of-order speculative processing, useful instructions can be overlapped with the memory instructions.

On the other side, prefetching mechanisms also increase the memory level parallelism. Recent processors come with hardware-prefetchers that automatically detect access patterns and prefetch necessary data in advance. Moreover, if the access patterns are predictable, software-prefetching instructions can be used to improve the memory latency by hiding access latency behind useful computation. Lastly, multi-banked caches improve the performance by simultaneous access to different cache banks at the same time. In summary, all of these trends show that techniques to increase cache and memory level parallelism will become more and more prevalent in upcoming processor architectures.

### 2.2.3 Virtual Memory and Address Translation

Operating systems usually provide a separate address space to each of the running processes in the system as if the process owns the entire physical memory. This
mechanism is called \textit{virtual memory} and it forms the basis for sharing the entire physical memory among multiple number of processes. Apart from sharing, virtual memory comes with other benefits such as security–address space of each process is invisible to the outside– and easy management of the memory hierarchy. For more details on virtual memory we refer the reader to [Tan07, Section 3.3]. Despite the benefits, however, virtual memory comes with its cost trade-off. Since all the memory accesses are done through virtual addresses, addresses have to be translated to the physical memory address before each access. Therefore, each address translation can lead to additional memory accesses since address mapping tables usually reside in main memory.

In order to exploit the principle of locality, modern processors provide mechanisms to accelerate address translation. Address mappings for recently accessed pages are stored in a small, fast cache called \textit{translation lookaside buffer} (TLB). During address translation, first mappings are checked from the TLB instead of going to the translation tables in memory. This mechanism speeds up address translation significantly given the high chance of address reuse due to locality of accesses. The TLB and the address translation is depicted in Figure 2.3. This example assumes a fully associative TLB. Each TLB tag from the given virtual address is compared against all the valid entries in the TLB. The corresponding physical page number (or frame number) returned from the TLB and the page offset bits from the virtual address constitute the final physical address.

TLB must be fast enough to be in the critical path of the CPU. Therefore, its size needs to be small and it can also be fully associative. Due to this reason, TLB sizes have not increased much over the years. Typical TLB sizes on modern processors vary between 64 and 1024. Moreover, recent processors implement two level TLB hierarchy. For instance, Intel Sandy Bridge microarchitecture comes with a first level TLB of size 64 and a second level TLB of size 512. The mechanism works like the L1/L2 cache hierarchy, where the second level TLB is larger but
usually slower.

On the other hand, modern processors provide a configurability for TLBs. Typically a small virtual memory page size of 4 KiB is a default and larger page sizes such as 2 MiB, 1 GiB can also be chosen. The impact on TLB however varies from processor to processor. For instance, Intel Sandy Bridge only supports 32 first level TLB entries when using 2 MiB pages and no second level TLB. Although the coverage of the entire TLB ($32 \times 2 \text{MiB}$ vs. $512 \times 4 \text{KiB}$) increases with larger pages, number of TLB entries available become fewer.

Finally, we note that TLB misses constitute an important cost factor in the memory access costs. Especially, in random access patterns where number of concurrently accessed memory pages exceed the hardware TLB sizes, each access to the memory results in a TLB miss in addition to the cache misses. This in turn results in significant slowdown in application performance. Therefore, an application can improve the performance by paying attention to the architectural TLB specifications. Access patterns should be turned into TLB friendly ones, where number of open memory pages should be restricted by the TLB size.

2.3 Instruction Execution and Instruction Level Parallelism

One of the prevalent techniques in modern processors for improving performance is to exploit instruction level parallelism during the execution. This is commonly achieved by using pipelining where multiple number of instructions are overlapped at execution. The technique essentially divides CPU instructions into multiple stages and allows different multiple stages to be active at any time. A $k$-stage pipeline is therefore likely to improve performance by a factor of $k$ (assuming pipeline stages are of equal length). Figure 2.4(b) illustrates the pipelining idea with a simple example of five execution stages. Moreover, the length of the slowest pipeline stage determines the clock frequency.

In the past, there have been processor architectures with very deep pipelining for increasing the clock speed of the CPU. For instance Intel Pentium 4 supported a pipeline with more than 31 stages and provided clock rates up to 3.8 GHz. However, the performance of deep pipelines suffers from different forms of hazards (structural, data and control). The hazards often cause pipeline stalls or pipeline flushes and significantly deteriorate the performance depending on the application. For instance a stall in the example in Figure 2.4(b) causes bubbles (i.e., empty stages) instead of executed stages. Therefore, the trend towards very deep pipelines was abandoned due to inefficiency.

There are several other approaches to exploit instruction level parallelism. First
2.4 Vectorization and Data Level Parallelism

Instruction level parallelism turns out to be less effective for programs with mostly integer, branch and control instructions. In case of programs with loop level parallelism and floating point instructions, the benefit can be higher. However, the
maximum level of parallelism that can be achieved with ILP still remains limited and requires significant complexity at hardware level with high consumption of chip resources (e.g., transistors). An alternative for exploiting parallelism and making use of increasing number of transistors in processor designs is data level parallelism. The idea takes its roots from vector style processing or vector processors [Rus78]. A number of data elements are loaded into vector registers and a single instruction is applied to multiple data elements at once. Hence, the style of processing is classified as single instruction, multiple data or SIMD for short [Fly72, FR96].

Today, modern processors provide special vector or SIMD units. These units have special registers which are typically 128/256 bits. Multiple number of data types with different number of bits (e.g., 8-bit char, 16-bit short, 32-bit int or 64-bit long) can be loaded into these registers. Various types of ALU instructions can then be applied on these multiple data items at the same time. Figure 2.4 illustrates SIMD addition instruction on two SIMD registers that hold $n$ data items.

SIMD instructions provide independent operation on multiple elements over the same register. Therefore, it eliminates the risk for data hazards. In practice, the explicit independence increases the amount of parallelism that can be achieved. Although ideal linear speedup with vector length is quite achievable with many instructions, there are also instructions where ideal parallelism cannot be achieved due to complexity at the hardware logic level (e.g., instructions that operate across lanes).

The programming model of SIMD requires explicit programmer control in most of the cases. Although compilers provide automatic vectorization opportunities for simple code that exhibit parallelism explicitly, the extent of automatic vectorization remains rather limited for complex code. Compiler effectiveness for exploiting vectorization can be improved with explicit compiler annotations. For instance, compiler can use those annotations to infer vector sizes and vectorization oppor-
tunities. This mechanism turns out to be dependent on the compiler and proves to be not too effective either. The most effective option for vectorization turns out to be careful and explicit programming with corresponding SIMD instructions. However, rather than using assembly instructions this process can be facilitated easier by SIMD intrinsics. *SIMD intrinsics* are function like calls to corresponding SIMD instructions implemented as compiler macros. Among the benefits are easier programming, leaving register allocation to the compiler and more flexible control. The down side of the intrinsics approach is being prone to portability issues between different architectures and compilers.

Recent generation processors provide many advancements on the vectorization technology. Most of the effort goes into increasing the vector width (*i.e.*, register size). To illustrate, Intel Sandy Bridge and AMD Bulldozer microarchitectures provide 256-bit SIMD registers and a new instruction set named as advanced vector extensions (AVX). More recent Intel Many Integrated Cores (MIC) architecture [Int14] is announced to provide 512-bit SIMD instructions. In the future, newer generations of processors are likely to provide 512-bit or wider SIMD instructions. On the other hand, execution performance of SIMD instructions are as important as the SIMD size. For instance, Sandy Bridge executes 256-bit multiply, add and shuffle in single cycle and can dispatch more than one SIMD instruction per cycle. However, for some instructions such as shuffle, min/max performance remains to be further improved. Overall, technological trends provide us with wider and faster SIMD execution units but higher performance will be achieved by carefully tuned and tailored programs that exploit these features.

### 2.5 Multi-core and Thread Level Parallelism

As we have discussed earlier, the diminishing performance gains in other approaches (frequency scaling, ILP, vectorization, etc.) shifted the focus to multi-processor/multi-core architectures in processor designs. In modern processor architectures, multi-core brings vast opportunities for parallel processing. However, to fully exploit the potential of multi-core processors a paradigm shift in the programming model is required. Programmers need to think explicitly about concurrency in their applications. A common wisdom is to think in terms of task or thread level parallelism where independent stream of instructions are executed on different processing cores. The processing model for multi-processor/multi-core hardware is often classified as multiple instruction streams, multiple data streams (*MIMD*) [Fly72, FR96] and illustrated in Figure 2.6.
Modern multi-core processor architectures implement a *shared-memory* design. All the cores in the processor access the same physical memory address space through a cache hierarchy. The cache hierarchy often consists of three levels where the lower levels L1 and L2 are exclusive to a core and L3 is shared among a number of cores that reside in the same processor socket. The processor might also consist of multiple CPU sockets each attached to its own memory. In such designs, *thread locality* becomes a more subtle issue. First, programmer needs to pay attention to where the memory is located respective to the cores where the threads of a program are running. Second, the last level cache is shared among the cores in the same CPU socket and therefore placement of threads might have significant impact on performance depending of whether threads are contending or cooperating with each other.

The overall memory bandwidth to a CPU socket is shared among all the processing cores inside the socket. The number of cores per CPU socket is likely to increase with the Moore’s law for years to come. However, the memory bandwidth is not likely to increase at the same pace. Therefore, bandwidth per core will decrease and bandwidth problem will become more pronounced in future multi-core processors. As such, programmers need to pay more attention to underlying architecture to avoid potential problems. One such problem is thread placement. Two threads accessing same memory region (*e.g.*, same data array) but placed on different CPU sockets need to fetch the same array from memory twice. However, if threads were placed in the same CPU socket, data cached in the last level cache would be reused by threads resulting in less memory fetches. Unfortunately, current operating systems does not mitigate this problem automatically, hence careful placement of threads to CPU cores are necessary through interfaces and libraries that expose thread affinity methods.
2.5.2 Multi-core and Caches

Shared caches such as last level cache can be useful in cases where threads cooperate in their behavior as described above. However, threads often also contend for using the cache exclusively. A thread with strong locality usually needs to reserve large amount of the cache exclusively for high efficiency. In contrast, a thread with weak locality does not require keeping its entire working set in the cache. Threads with strong locality tend to be sensitive to the sharing of the cache. If a thread with strong locality and another thread share the same cache, then the impact on the performance of the strong locality thread becomes detrimental [LDC09]. In fact, the cached data of the strong locality thread becomes evicted from the cache by the other thread (i.e., cache pollution). One solution to overcome this problem is partitioning the shared cache among threads with a page coloring mechanism at the OS level [LDC09]. Another mechanism is to place threads that contend for the same cache on different CPU sockets by explicit thread affinity control or smarter OS scheduling mechanisms.

The consistency model of shared-memory often requires that the changes to any memory address by any of the threads should be visible to all the threads. Therefore, processors need to implement a cache coherency protocol to invalidate and update older values of any given memory address in all the processor caches. Sharing of data and variables among threads are quite common—regardless of whether they are in the same CPU socket or not. For instance two threads might want to check a flag variable and set it under some condition. Despite this flag being a single byte variable, a change of this flag implies ping-pong of the relevant cache line among all the processor cores due to the cache coherency protocol. Moreover, other variables residing on the same cache line suffer from this performance penalty. The problem is called false sharing and results in performance degradations such as unnecessary memory stalls, wasted bandwidth and interconnect usage. The problem—once recognized—can be solved easily by putting frequently modified variables into a single cache line of their own. Due to this and similar problems, the underlying processor often makes unnecessary use of resources. In this respect, programmer knowledge against such underlying issues in multi-core architectures becomes more than a necessity.

2.5.3 Synchronization

As a rule of thumb, synchronization and shared data structures should be minimized as much as possible for ideal benefits from multi-core parallelism. Shared-memory architecture enables fine-granular synchronization mechanisms to reduce shared accesses. Nevertheless, it might not be possible to eliminate synchronization completely.
Among the synchronization mechanisms, locks or short-duration locks (i.e., latches in databases) are quite commonly used to protect shared access to data structures. Locks can be implemented in various ways. One is to implement them as blocking where waiting thread is de-scheduled and put into sleep with a context switch. This mechanism is a viable option if the blocking duration is long enough so that it compensates the context switch overhead. Another approach is the use of spinning or spin locks where the waiting thread busy waits on the lock until it becomes available. This approach is preferable whenever the expected lock duration is rather short. Otherwise busy waiting wastes valuable CPU cycles and increases energy consumption. Moreover, if number of threads contending for the same lock is high (which suggests the lock is not fine-granular enough), then false sharing issues and increased latencies for acquiring locks becomes visible. As both approaches have meaningful use cases, the correct one for the problem at hand must be carefully selected and implemented by the programmer.

Another common synchronization pattern in parallel programs is barriers. Often threads need to synchronize at an execution point after finishing certain task. At the implementation level barriers might require usage of short-duration locks. However, usually the problem with barriers is not the underlying implementation but the usage pattern of barriers. The execution tasks assigned to threads might be imbalanced in terms of the execution time required. Therefore, some threads can finish their tasks earlier and must wait idle for the other threads to arrive to the barrier. This results in under utilization of CPU cores. This problem requires thinking by the programmer at the software level to balance the execution of tasks.

\subsection*{Simultaneous Multi-Threading (SMT)}

Modern processors also support multi-threading on a single processor core to a certain degree. In this scheme, instead of relying on fully replicated physical processor cores, instruction streams from multiple threads are executed on the same physical CPU core. Essentially, in this form, the parallelism achieved is mainly due to ILP where threads actually only exist at the logical level. The technique is called \textit{simultaneous multi-threading} (SMT).

The idea of SMT is illustrated in Figure 2.7. In addition to instructions from the same thread being dynamically scheduled to different execution units with ILP, SMT enables instructions from different threads to run on idle execution units. SMT requires certain complexity at the hardware implementation level. However, most of the existing setup for ILP can be utilized (virtual registers, register renaming, functional unit dispatching, etc.). Furthermore, resource sharing is even more pronounced with SMT. SMT threads share almost all of the hardware such as caches, TLB, execution units, SIMD units, branch prediction units,
etc. Therefore, performance gains achievable through SMT is rather limited and usually not linear with the number of SMT threads.

In recent processors, there are various flavors of the SMT technology. For instance Intel calls its multi-threading technology “hyper-threading” and supports two threads per physical core. Sparc architecture from Oracle supports multi-threading with eight threads per physical core where threads are switched at a cycle granularity. This technology is called “fine-grained multi-threading” [SGG+12]. Although the implementations differ in their complexity and sophistication, the overall goal of multi-threading remains the same: Filling execution units of a single physical processor core with instructions from different logical threads and hiding latency of stalls behind useful computation.

2.6 Implications on Data Processing

The previous sections provided an overview of features of modern processor architectures. In this section, we briefly mention implications of such features for data processing algorithms.

Main-memory is becoming abundant in modern processors. This has implications for moving and keeping much of the data in-memory. Database sizes commonly just suitable for storage in disk are nowadays can be fully resident in main-memory. Accordingly, data processing algorithms must be tailored for the main-memory storage and access.

There is an ever increasing amount of parallelism available in modern processors. Multi-core revolution enables a parallel/distributed computing architecture within a single machine. Essentially, years of experience and knowledge from parallel and distributed databases can be utilized in multi-core machines with the same objective of achieving better performance, however with different architectural constraints this time. As well as multi-threading, hardware acceleration and features such as advanced instructions, wider SIMD must be exploited for further levels of parallelism.
Data structures and layouts for data processing needs to be made cache and memory hierarchy conscious. Caches amplify the performance of the main-memory, therefore making data structures as much cache resident as possible results in significant performance gains. Data access patterns must be cache and TLB friendly to utilize all the benefits of the underlying hardware.

Query execution algorithms must take into account the parallelism, architecture and advanced hardware features into consideration. Existing algorithms must be redesigned and tuned to the hardware to extract the performance premises of new hardware.
3

Hash Joins

3.1 Introduction

The radical changes and advances in processor architecture caused by multi-core have triggered a revision of the algorithms for implementing main-memory hash joins. The results available in the literature so far present a very confusing and contradictory landscape.

On the one hand, some authors claim that the best performance is to be achieved by fine tuning to the underlying architecture. For instance, Kim et al. [KSC+09] look at the effects of caches and TLB (translation lookaside buffer) on main-memory parallel hash joins and show how careful partitioning according to the cache and TLB sizes leads to improved performance. Along the same lines, Lang et al. [LLA+13] have shown how tuning to the non-uniform memory access (NUMA) characteristics also leads to improved performance of parallel hash joins. We will refer to the algorithms that take hardware characteristics into consideration as hardware-conscious. Some of this work further emphasizes the impact of hardware by suggesting that, in the future, as SIMD becomes wider, sort-merge join is likely to perform better than hash joins [KSC+09].

On the other hand, other authors argue that parallel hash join algorithms can be made efficient while remaining hardware-oblivious [BLP11]. That is, there is no need for tuning—particularly of the partition phase of a join where data is carefully arranged to fit into the corresponding caches—because modern hardware hides the performance loss inherent in the multi-layer memory hierarchy. In addition, so the
argument goes, the fine tuning of the algorithms to specific hardware makes them less portable and less robust to, e.g., data skew.

Further contradictory results exist around the performance of sort-merge joins. For instance, Albutiu et al. [AKN12] claim that sort-merge join is already better than hash join and can be efficiently implemented without using SIMD [AKN12]. These results contradict the claims of both Blanas et al. [BLP11] and Kim et al. [KSC+09] (the former argues that hardware consciousness is not necessary to achieve high performance; the latter concludes that sort-merge will only be competitive on wide SIMD architectures). Recent work by Balkesen et al. [BATÖ13] has looked into this question in more detail and found out that hardware-conscious (radix-)hash join algorithms still maintain an edge over sort-merge joins despite the advances on SIMD.

In this chapter we focus on main-memory, parallel hash join algorithms and address the question of whether it is important to tune the algorithms to the underlying hardware as claimed by Kim et al. [KSC+09] and by Albutiu et al. [AKN12] or whether hardware-oblivious approaches provide sufficient performance to make the overhead of being hardware-conscious unnecessary.

Answering this question is a non-trivial task because of the intricacies of modern hardware and the many possibilities available for tuning. To make matters worse, many parameters affect the behavior of join operators: relative table sizes, use of SIMD, page sizes, TLB sizes, structure of the tables and organization, hardware architecture, tuning of the implementation, etc. Existing studies share very few common points in terms of the space explored, making it difficult to compare their results.

The first contribution of this chapter of the thesis is algorithmic. We analyze the algorithms proposed in the literature and introduce several important optimizations leading to algorithms that are more efficient and robust to parameter changes. In doing so, we provide important insights on the effects of multi-core hardware on algorithm design.

The second contribution is to put existing claims into perspective, showing what choice of parameters or hardware features cause the observed behaviors. These results shed light on what parameters play a role in multi-core systems, thereby establishing the basis for the choices a query optimizer for multi-core will need to make.

The third and final contribution is to settle the issue of whether tuning to the underlying hardware plays a role. The answer is affirmative in most situations. However, for certain combinations of parameters and architectures, hardware-oblivious approaches have an advantage. As our results show, architectural features such as effective and aggressive on-chip multi-threading on Sparc RISC architecture CPUs favor the hardware-oblivious no partitioning idea. Therefore, heavy
3.2. In-Memory Hash Join Algorithms

SMT might change the picture in the future.

The material presented in this chapter extends and expands on the results by Balkesen et al. [BTAÔ13]. Through additional experiments on more recent Sparc RISC processors, we have determined that machines with aggressive SMT benefit **hardware-oblivious** algorithms. This advantage does not happen in all architectures but it is the result of a combination of features (low synchronization cost, fully associative TLBs, aggressive on-chip multi-threading, etc.). These results open up an interesting research direction for future processor architectures that could be tailored to more efficient data processing operators. All previous work, including [BTAÔ13], explored only deployments on single sockets. In this chapter, we look into deployments on multi-sockets using the latest x86 multi-core processors (Intel Sandy Bridge). Multi-socket deployment turns out to be a key factor in boosting the performance of join algorithms leading to results that are the fastests published to date. The chapter also contains an analysis of the impact of cache hierarchies and large SMT architectures on the performance of hash joins and why these hardware features do not benefit **hardware-conscious** algorithms. In addition to the existing optimizations in [BTAÔ13], the chapter includes as well a discussion on how to turn hardware-oblivious algorithms into hardware-conscious ones by using software prefetching and cache alignment. Finally, the chapter also explores new optimizations like the use of software-managed buffers.

### 3.2 In-Memory Hash Join Algorithms

Hash join algorithms can be classified into two camps. **Hardware-oblivious** hash join variants, represented here by *no partitioning join* (Section 3.2.2), do not depend on any hardware-specific parameters. Rather, they consider qualitative characteristics of modern hardware and are expected to achieve good performance on any technologically similar hardware. **Hardware-conscious** implementations, such as *(parallel)* radix join (Sections 3.2.3 and 3.2.4), aim to maximally exploit a given piece of hardware by tuning algorithm parameters (*e.g.*, hash table sizes) to its particular features.

The goal of our work is to compare two alternatives. One is to assume hardware has now become good enough at hiding its own limitations—through automatic hardware prefetching or out-of-order execution—to make hardware-oblivious algorithms competitive. The other is to assume that explicit parameter tuning\(^1\) yields enough performance advantages to warrant the effort required.

\(^1\)usually by means of automated tools, such as **Calibrator** [MBK02b]
Chapter 3. Hash Joins

3.2.1 Canonical Hash Join Algorithm

The basis behind any modern hash join implementation is the canonical hash join algorithm [ÖV11, KTMO83], which operates in two phases as shown in Figure 3.1. In the first build phase, the smaller of the two input relations, $R$, is scanned to populate a hash table with all $R$ tuples. Then the probe phase scans the second input relation, $S$, and probes the hash table for each $S$ tuple to find matching $R$ tuples.

Both input relations are scanned once and, with an assumed constant-time cost for hash table accesses, the expected complexity for the canonical hash join algorithm is $O(|R| + |S|)$.

3.2.2 No Partitioning Join

To benefit from modern parallel hardware, Blanas et al. [BLP11] proposed a variant of the canonical algorithm that they termed no partitioning join, essentially a direct parallel version of the canonical hash join. It does not depend on any hardware-specific parameters and—unlike alternatives that will be discussed shortly—does not physically partition data. The argument is that the partitioning phase requires multiple passes over the data and can be omitted by relying on modern processor features such as simultaneous multi-threading (SMT) to hide cache latencies.

Both input relations are divided into equi-sized portions that are assigned to a number of worker threads. As shown in Figure 3.2, in the build phase, all worker threads populate a shared hash table accessible to all worker threads.

After synchronization via a barrier, all worker threads enter the probe phase and concurrently find matching join partners for their assigned $S$ portions.

An important characteristic of no partitioning is that the hash table is shared among all participating threads. This means that concurrent insertions into the hash table must be synchronized. To this end, each bucket is protected via a
3.2. In-Memory Hash Join Algorithms

latch that a thread must obtain before it can insert a tuple. The potential latch contention is expected to remain low, because the number of hash buckets is typically large (in the millions). The probe phase accesses the hash table in read-only mode. Thus, no latches have to be acquired in that second phase.

On a system with \( p \) cores, the expected complexity of this parallel version of hash join is \( O\left(\frac{1}{p}(|R| + |S|)\right) \).

3.2.3 Radix Join

Hardware-conscious, main-memory hash join implementations build upon the findings of Shatdal et al. [SKN94] and Manegold et al. [MBK02b, BMK99]. While the principle of hashing—direct positional access based on a key’s hash value—is appealing, the resulting random access to memory can lead to cache misses. Thus, the main focus is on tuning main-memory access by using caches more efficiently, which has been shown to impact query performance [ADHW99]. Shatdal et al. [SKN94] identify that when the hash table is larger than the cache size, almost every access to the hash table results in a cache miss. Consequently, partitioning the hash table into cache-sized blocks reduces cache misses and improves performance. Manegold et al. [MBK02b] refined this idea by considering as well the effects of translation look-aside buffers (TLBs) during the partitioning phase. This led to multi-pass partitioning, now a standard component of the radix join algorithm.

Partitioned Hash Join. The partitioning idea is illustrated in Figure 3.3. In the first phase of the algorithm the two input relations \( R \) and \( S \) are divided into partitions \( r_i \) and \( s_j \), respectively. During the build phase, a separate hash table is created for each \( r_i \) partition (assuming \( R \) is the smaller input relation). Each of
these hash tables now fits into the CPU cache. During the final probe phase, $s_j$ partitions are scanned and the respective hash table is probed for matching tuples.

During the partitioning phase, input tuples are divided up using hash partitioning (via hash function $h_1$ in Figure 3.3) on their key values (thus, $r_i \not\bowtie s_j = \emptyset$ for $i \neq j$) and another hash function $h_2$ is used to populate the hash tables.

While avoiding cache misses during the build and probe phases, partitioning the input data may cause a different type of cache problem. The partitions will typically reside on different memory pages with a separate entry for virtual memory mapping required for each partition. This mapping is cached by TLBs in modern processors. As Manegold et al. [MBK02b] point out, the partitioning phase may cause TLB misses if the number of created partitions is too large.

Essentially, the number of available TLB entries defines an upper bound on the number of partitions that can be efficiently created or accessed at the same time.

**Radix Partitioning.** Excessive TLB misses can be avoided by partitioning the input data in multiple passes. In each pass $j$, all partitions produced by the preceding pass $j - 1$ are refined, such that the partitioning fan-out never exceeds the hardware limit given by the number of TLB entries. In practice, each pass looks at a different set of bits from the hash function $h_1$, which is why this is called radix partitioning. For typical in-memory data sizes, two or three passes are sufficient to create cache-sized partitions, without suffering from TLB capacity limitations.

**Radix Join.** The complete radix join is illustrated in Figure 3.4. Both inputs are partitioned using two-pass radix partitioning (two TLB entries would be sufficient
3.2. In-Memory Hash Join Algorithms

In radix join, multiple passes have to be done over both input relations. Since the maximum “fanout” per pass is fixed by hardware parameters, \( \log |R| \) passes are necessary, where \( R \) again is the smaller input relation. Thus, we expect a runtime complexity of \( O((|R| + |S|) \log |R|) \) for radix join.

**Hardware Parameters.** Radix join needs to be tuned to a particular architecture essentially via two parameters: (i) the maximum fanout per radix pass is primarily limited by the number of TLB entries of the hardware; (ii) the resulting partition size should roughly be the size of the system’s CPU cache. Both parameters can be obtained in a rather straightforward way, e.g., with help of benchmark tools such as Calibrator [MBK02b]. As we shall see later, radix join is not overly sensitive to a potential mis-configuration of either parameter.

### 3.2.4 Parallel Radix Join

Radix join can be parallelized by subdividing both input relations into sub-relations that are assigned to individual threads [KSC+09]. During the first pass, all threads create a *shared set of partitions*. As before, the number of partitions in this set is limited by hardware parameters and typically small (few tens of partitions). They are accessed by potentially many execution threads, creating a contention problem (the low-contention assumption of Section 3.2.2 no longer applies).
To avoid this contention, for each thread a dedicated range is reserved within each output partition. To this end, both input relations are scanned twice. The first scan computes a set of histograms over the input data, so the exact output size is known for each thread and each partition. Next, a contiguous memory space is allocated for the output and, by computing a prefix-sum over the histogram, each thread pre-computes the exclusive location where it writes its output. Finally, all threads perform their partitioning without any need to synchronize.

After the first partitioning pass, there is typically enough independent work in the system (cf. Figure 3.4) that workers can perform work on their own. Load distribution among worker threads is typically implemented via task queueing (cf. [KSC+09]).

### 3.3 Experimental Setup

In this section we describe the experimental setup used for the evaluation of the hash join algorithms.

#### 3.3.1 Workload

For the comparison, we use machine and workload configurations that mimic scenarios where in-memory join processing is most relevant. In particular, all systems where the component truly matters assume a column-oriented storage model. We thus deliberately choose very narrow \( \langle \text{key}, \text{payload} \rangle \) tuple configurations, where \text{key} and \text{payload} are four or eight bytes wide. As a side effect, narrow tuples better pronounce the effects that we are interested in since they put more pressure on the system’s caching system.\(^2\)

We adopted the particular configuration of our workloads from existing work, which also eases the comparison of our results with those published in the past.

\(^2\)The effect of tuple widths was studied, \textit{e.g.}, by Manegold et al. [MBNK04].

<table>
<thead>
<tr>
<th></th>
<th>A (from [BLP11])</th>
<th>B (from [KSC+09])</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of key / payload</td>
<td>(8/8) bytes</td>
<td>(4/4) bytes</td>
</tr>
<tr>
<td>size of (R)</td>
<td>(16 \cdot 2^{20}) tuples</td>
<td>(128 \cdot 10^6) tuples</td>
</tr>
<tr>
<td>size of (S)</td>
<td>(256 \cdot 2^{20}) tuples</td>
<td>(128 \cdot 10^6) tuples</td>
</tr>
<tr>
<td>total size (R)</td>
<td>(256) MiB</td>
<td>(977) MiB</td>
</tr>
<tr>
<td>total size (S)</td>
<td>(4096) MiB</td>
<td>(977) MiB</td>
</tr>
</tbody>
</table>
3.3. Experimental Setup

### Table 3.2: Hardware platforms used in our evaluation

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>Intel Sandy Bridge</th>
<th>AMD Bulldozer</th>
<th>Sun Niagara 2</th>
<th>Intel Sandy Bridge</th>
<th>Oracle Sparc T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Xeon</td>
<td>Xeon</td>
<td>Opteron</td>
<td>UltraSPARC</td>
<td>Xeon</td>
<td>SPARC T4</td>
</tr>
<tr>
<td>L5520</td>
<td>L5520</td>
<td>E5-2680</td>
<td>6276</td>
<td>T2</td>
<td>E5-4640</td>
<td>T4</td>
</tr>
<tr>
<td>2.26 GHz</td>
<td>2.7 GHz</td>
<td>2.3 GHz</td>
<td>1.2 GHz</td>
<td>2.4 GHz</td>
<td>3.0 GHz</td>
<td></td>
</tr>
<tr>
<td>Cores/Threads</td>
<td>4/8</td>
<td>8/16</td>
<td>16/16</td>
<td>8/64</td>
<td>32/64</td>
<td>32/256</td>
</tr>
<tr>
<td>Cache sizes</td>
<td>32 KiB</td>
<td>32 KiB</td>
<td>16 KiB</td>
<td>8 KiB</td>
<td>32 KiB</td>
<td>16 KiB</td>
</tr>
<tr>
<td>(L1/L2/L3)</td>
<td>256 KiB</td>
<td>256 KiB</td>
<td>2 MiB</td>
<td>4 MiB</td>
<td>256 KiB</td>
<td>128 KiB</td>
</tr>
<tr>
<td></td>
<td>8 MiB</td>
<td>20 MiB</td>
<td>16 MiB</td>
<td>-</td>
<td>20 MiB</td>
<td>4 MiB</td>
</tr>
<tr>
<td>TLB (L1/L2)</td>
<td>64/512</td>
<td>64/512</td>
<td>32/1024</td>
<td>128/-</td>
<td>64/512</td>
<td>128/-</td>
</tr>
<tr>
<td>Memory</td>
<td>24 GiB DDR3</td>
<td>32 GiB DDR3</td>
<td>32 GiB DDR3</td>
<td>16 GiB</td>
<td>512 GiB DDR3</td>
<td>1 TiB</td>
</tr>
<tr>
<td>1066 MHz</td>
<td>1600 MHz</td>
<td>1333 MHz</td>
<td>FB DIMM</td>
<td>1600 MHz</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VM Page size</td>
<td>4 KiB</td>
<td>4 KiB</td>
<td>4 KiB</td>
<td>8 KiB</td>
<td>2 MiB</td>
<td>4 MiB</td>
</tr>
</tbody>
</table>

As illustrated in Table 3.1, we adopted workloads from Blanas et al. [BLP11] and Kim et al. [KSC+09] and refer to them as A and B here, respectively. All attributes are integers, and the keys of R and S follow a foreign key relationship. That is, every tuple in S is guaranteed to find exactly one join partner in R. Most of our experiments (unless noted otherwise) assume a uniform distribution of key values from R in S.

#### 3.3.2 Hardware Platforms

We evaluated the algorithms on six different multi-core machines shown in Table 3.2. Being relatively older, the Sun UltraSPARC T2 dates back to 2007 and provides eight thread contexts per core where eight threads share the L1 cache with a line size of 16 bytes. All the Intel machines support SMT with two thread contexts per core. The Sun UltraSPARC T2 comes with two levels of cache, where cores share the L2 cache with line size of 64 bytes. On the Intel machines, cores use a shared L3 cache and a cache line size of 64 bytes. The AMD machine has a different architecture than the others: two cores are packaged as single module and share some resources such as instruction fetch, decode, floating point unit and L2 cache. Accordingly, the effective L2 cache available per core is reduced to half, i.e., 1 MiB.

Among these platforms, Oracle Sparc T4 [SGG+12] and Intel E5-4640 are recent high-end multi-core servers where both of them come in a four-socket configuration. In Oracle Sparc T4, each of the sockets provides eight processor cores and each core has hardware support for eight threads. Therefore the machine provides a total of 256 hardware threads. The T4 has 16 KiB L1 and 128 KiB L2 per core with 32-byte cache lines, and a shared 4 MiB L3 with 64-byte cache lines. The memory management unit has a fully associative, 128-entry DTLB and supports
8 KiB, 64 KiB, 4 MiB, 256 MiB and 2 GiB page sizes with a hardware table-walk engine. However, the DTLB and hardware table-walk engine is shared by eight threads dynamically. In comparison to the previous generations of Sparc (such as T2), T4 provides many improvements such as an aggressive clock frequency of 3.0 GHz, advanced branch prediction, out-of-order execution and shared L3. The system runs Solaris 11.1 (SunOS 5.11). The Intel E5-4640 has a total memory of 512 GiB and a clock frequency of 2.4 GHz. In addition, the system runs Debian Linux 7.0, kernel version 3.4.4-U5 and it uses 2 MiB VM pages for memory allocations transparently. The compiler and flags used have a significant impact on T4 and discussed in Section 3.6.9 in detail.

The Intel L5520, E5-2680 and AMD systems run Ubuntu Linux (kernel version 2.6.32) and Sun UltraSPARC T2 runs a Debian (kernel version 3.2.0-3-sparc64-smp). For the results we report here, we used gcc 4.4.3 on Ubuntu and gcc 4.6.3 on Debian and the -O3 and -mtune=niagara2 -mcpu=ultrasparc command line options to compile our code. Additional experiments using Intel’s icc compiler did not show any notable differences, qualitatively or quantitatively. For the performance counter profiles that we report, we instrumented our code with the Intel Performance Counter Monitor [Int12c].

### 3.4 Hardware-Oblivious Hash Joins

In this section we first study and optimize the no partitioning strategy. To make our results comparable, we use similar hardware to that employed in earlier work, namely a Nehalem L5520 system (cf. Table 3.2).

#### 3.4.1 Build Cost

The overall cost of hardware-oblivious no partitioning join is given by

\[
\text{cost} = c_{\text{put}} \cdot |R| + c_{\text{get}} \cdot |S|,
\]

where \(c_{\text{put}}\) and \(c_{\text{get}}\) denote the (constant) cost of adding or reading an entry to/from the hash table (respectively). Writing to the hash table is generally more expensive, since it involves the acquisition of a bucket latch, hence, \(c_{\text{put}} \gtrsim c_{\text{get}}\).

No partitioning was proposed and evaluated by Blanas et al. in [BLP11]. Surprisingly, in their experiments—based on what we call Workload A in our work—the build phase accounts for only 2% of the overall execution time. In this workload, \(|R| = \frac{1}{16} \cdot |S|\), so we would expect the build phase to take at least \(\approx 6\%\) of the overall cost.
3.4. Hardware-Oblivious Hash Joins

Table 3.3: Effect of sorted input on the build phase (Code by [BLP11] vs. our own code; Performance counters in millions; Workload A)

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
<th>L3 miss</th>
<th>Instr.</th>
<th>TLB load miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code of [BLP11], sorted input</td>
<td>322</td>
<td>2</td>
<td>2215</td>
<td>1</td>
</tr>
<tr>
<td>Code of [BLP11], unsorted input</td>
<td>1415</td>
<td>45.3</td>
<td>2263</td>
<td>52.7</td>
</tr>
<tr>
<td>Our code, unsorted input</td>
<td>966</td>
<td>25</td>
<td>572</td>
<td>56</td>
</tr>
</tbody>
</table>

![Graph showing cycles per output tuple for hardware-oblivious no partitioning strategy](image)

Figure 3.5: Cycles per output tuple for hardware-oblivious no partitioning strategy (Workload A; Intel Xeon L5520, 2.26 GHz).

The code used to obtain these results is publicly available [BP12]. Analysis of this code reveals that their results are based on experiments where $R$ is pre-sorted. As a result, as data items are hashed using a modulo hash function, they map to consecutive hash buckets, leading to strictly sequential memory accesses. The sorted input also removes any contention for the bucket latch.

Re-running the experiments with randomly permuted input (i.e., the general case) results in build costs of about 6%, consistent with our assumption stated above. To confirm that our assessment is correct, we collected cache profile data. Table 3.3 illustrates how sorted input essentially eliminates all TLB and L3 cache misses. Otherwise, we could basically reproduce other performance results (cf. Figure 3.5, dark bars).
Chapter 3. Hash Joins

3.4.2 Cache Efficiency

When running the no partitioning code of [BLP11], the cache profile information in Table 3.3 indicates the hash table build-up incurs a very high number of cache and TLB misses. Processing 16 million tuples results in 45.3/52.7 million L3/TLB misses, or about three misses per input tuple.

The reason for this inefficiency becomes clear as we look at the code of [BLP11]. The hash table in this code is implemented as illustrated in Figure 3.6.

The hash table itself is an array of head pointers, each of which points to the head of a linked bucket chain. Each bucket is implemented as a 48-byte record. A free pointer points to the next available tuple space inside the current bucket. A next pointer leads to the next overflow bucket, and each bucket can hold two 16-byte input tuples.

Since the hash table is shared among worker threads, latches are necessary for synchronization. As illustrated above, they are implemented as a separate latch array which is position-aligned with the head pointer array.

In this table, a new entry can be inserted in three steps (ignoring overflow situations due to hash collisions): (1) the latch must be obtained in the latch array; (2) the head pointer must be read from the hash table; (3) the head pointer must be dereferenced to find the hash bucket where the tuple can be inserted. In practice, each of these three steps is likely to result in a cache miss.
3.4. Hardware-Oblivious Hash Joins

Optimized Hash Table Implementation. To improve the cache efficiency of no partitioning, in our re-implementation we directly combined locks and hash buckets to neighboring memory locations. More specifically, in our code we implemented the main hash table as a contiguous array of buckets, as shown in Figure 3.7. The hash function directly indexes into this array representation. For overflow buckets, we allocate additional bucket space outside the main hash table. Most importantly, the 1-byte synchronization latch is part of the 8-byte header that also contains a counter indicating the number of tuples currently in the bucket. In line with the original study [BLP11], for Workload A, we configured our hash table to two 16-byte tuples per bucket. An 8-byte next pointer is used to chain hash buckets in the case of overflows.

The effect of this modified hash table representation is significant. As listed in Table 3.3, it cuts by half the number of cache misses in the build phase (and also in the probe phase, though not shown in Table 3.3) and speeds up join processing by a fair margin.

In terms of absolute join performance, our re-written code is roughly three times faster than the code of Blanas et al. [BLP11], as shown in Figure 3.5. Yet, our code remains strictly hardware-oblivious: no hardware-specific parameters are needed to tune the code.

3.4.3 The Role of SMT Threads

Blanas et al. [BLP11] argue that no partitioning draws its true benefit from its good interplay with simultaneous multi-threading (SMT) hardware. Simply speaking, SMT provides the illusion of an extra CPU by running two threads on the same CPU and cleverly switching between them at the hardware level. This gives the hardware the flexibility to perform useful work even when one of the threads is stalled, e.g., because of a cache miss.

To study the interaction between no partitioning and SMT, we repeated the original SMT experiment [BLP11] on comparable hardware. Our Nehalem system contains four cores with two hardware contexts each. As in the original study, we start by assigning threads to different physical cores. Once the physical cores are exhausted, we assign threads to the available hardware context in a round-robin fashion.

Figure 3.8 illustrates the performance of no partitioning relative to the performance of a single-threaded execution of the same algorithm (“speedup”). Our experiment indeed confirms the scalability with SMT threads on the un-optimized code of [BLP11]. However, once we run the same experiment with our optimized code (with significantly better absolute performance, cf. Figure 3.5), SMT does not help the no partitioning strategy at all or only brings negligible improvement.
Chapter 3. Hash Joins

3.5 Hardware-Conscious Hash Joins

We perform a similar analysis for the parallel radix join. Blanas et al. [BLP11] also provide an implementation for this hardware-conscious join execution strategy.

3.5.1 Configuration Parameters

The key configuration parameter of radix join is the number of radix bits for the partitioning phase ($2^\text{# radix bits}$ partitions are created during that phase). Figure 3.9 illustrates the effect that this parameter has on the runtime of radix join.

The figure confirms the expected behavior that partitioning cost increases with the partition count, whereas the join phase becomes faster as partitions become smaller. Configurations with 14 and 11 radix bits are the best trade-offs between these opposing effects for the Nehalem and AMD architectures, respectively. But even more interestingly, the figure shows that radix join is fairly robust against parameter mis-configuration: within a relatively broad range of configurations, the performance of radix join degrades only marginally.

Figure 3.8: Speedup of no partitioning algorithm on SMT hardware. First four threads are “native” threads; threads 5–8 are “hyper threads.”

As the result shows, SMT can only remedy cache miss latencies if the respective code contains enough cache misses and enough additional work for the second thread while the first one is waiting. For code with less redundancy, SMT brings only negligible benefit. These results raise questions about a key hypothesis behind the hardware-oblivious no partitioning strategy.
3.5. Hardware-Conscious Hash Joins

3.5.2 Hash Tables and Cache Efficiency

Following the partitioning of the input tables, hash tables are very small and always fully cache resident. Thus, our assessment about cache misses for hash table accesses in the previous section no longer holds for the hardware-conscious join execution strategy.

Various implementations have been proposed for radix join. Manegold et al. [MBK02b] use a rather classical bucket chaining mechanism where individual tuples are chained to form a bucket. Following good design principles for efficient in-memory algorithms, all pointers are implemented as array position indexes (as opposed to actual memory pointers).

Kim et al. [KSC+09] build their hash table analogously to the parallel partitioning stage. The input relation is first scanned to obtain a histogram over hash values. Then, a prefix sum is used to help re-order relation $R$ (to obtain $R'$), such that tuples with the same hash value appear contiguously in $R'$. The prefix sum table and the re-ordered relation now together serve as a hash table as shown in Figure 3.10.

The advantage of this strategy is that contiguous tuples can now be compared using SIMD instructions. In addition, software prefetching mechanisms can be applied to bring potential matches to the L1 cache before comparisons by storing the probe items in a small buffer.

**Evaluation.** We evaluated the impact of different hash table implementation strategies on the join phase of radix join. Figure 3.11 shows the join phase cost in cycles per output tuple for three different strategies.

![Figure 3.9: Cost vs. radix bits (Workload B; Nehalem: 2-passes; AMD: 1-pass).](image-url)
Figure 3.10: Relation re-ordering and histogram-based hash table design.

Figure 3.11: Cost of join phase in radix join for three different hash table implementation techniques (Workload B; Intel Xeon L5520, 2.26 GHz; Using 8 threads and 2 pass partitioning).
3.5. Hardware-Conscious Hash Joins

<table>
<thead>
<tr>
<th></th>
<th>code from [BLP11]</th>
<th>our code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Part.</td>
<td>Build</td>
</tr>
<tr>
<td>Cycles</td>
<td>9398</td>
<td>499</td>
</tr>
<tr>
<td>Instructions</td>
<td>33520</td>
<td>2000</td>
</tr>
<tr>
<td>L2 misses</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>L3 misses</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>TLB load misses</td>
<td>9</td>
<td>0.3</td>
</tr>
<tr>
<td>TLB store misses</td>
<td>325</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4: CPU performance counter profiles for different radix join implementations (in millions); Workload A

As can be seen, the Manegold et al. implementation [MBK02b] still has an edge over the more recent one by Kim et al. [KSC+09], in spite of the potential for SIMD optimization in the latter implementation. The graph also confirms that the join cost generally decreases as the input data is partitioned in a more fine-granular way. In practice, there is a sweet spot, because the partitioning cost (which has to be invested before joining) increases with the number of partitions (cf. Figure 3.9).

Since the Manegold et al. approach comes out best in this comparison, we will use it for all following experiments. We note that the choice we are making here does not depend on hardware parameters (this is a hardware-oblivious optimization). As we shall see in a moment, the impact of our choice is limited, however, since the cost of partitioning adds to either of those implementation techniques.

3.5.3 Overall Execution Time

The overall cost of join execution consists of the cost for data partitioning and the cost of computing the individual joins over partitions. To evaluate the overall cost of join execution (and to prepare for a comparison with the hardware-oblivious no partitioning algorithm), we measured our own, carefully tuned implementation, as well as those reported in earlier work.

We had two implementations of radix join available. For the code of Blanas et al. [BP12], we found one pass and 2,048 partitions to be the optimal parameter configuration (matching the configuration in their experiments [BLP11]). Partitioning in that code turns out to be rather expensive. We attribute this to a coding style that leads to many function calls and pointer dereferences in critical code paths. Partitioning is much more efficient in our own code. This leads to a situation where two-pass partitioning with 16,384 partitions becomes the most efficient configuration. Table 3.4 illustrates how the different implementations lead
to significant differences in the executed instruction count. Our code performs two partitioning passes with 40% fewer instructions than the Blanas et al.’s code \cite{BLP11} needs to perform one pass.

The resulting overall execution times are reported (as cycles per output tuples) in Figure 3.12. This chart confirms that partitioning is rather expensive in the code of Blanas et al. Ultimately, this results in a situation where the resulting partition count is sub-optimal for the subsequent join phase, causing their join code to be also expensive. With optimized code, partitioning becomes the dominant cost, which is consistent with the findings of Kim et al. \cite{KSC09} that showed comparable cost at similar parameter settings. Overall, our code is about three times faster than the code of Blanas et al. for all tested configurations.

**Performance Counters.** We have also instrumented the available \textit{radix join} implementations to monitor CPU performance counters. Table 3.4 lists cache and TLB miss counts for the three tasks in \textit{radix join}. The table shows a significant difference in the number of cache and TLB misses between the implementation of Blanas et al. and ours. The idea behind \textit{radix join} is that all partitions should be sufficiently small to fully fit into caches. So one should expect a very low number of misses. This is true for our implementation but not for the one of Blanas et al.

The reason for the difference is an unfortunate execution order of hash building and probing in the latter code. Their code performs \textit{radix join} strictly in three
3.6. Hardware-Conscious or Not?

In this section we compare the algorithms above under a wide range of parameters and hardware platforms.

Figure 3.13: Speedup of radix algorithm on SMT hardware. First four threads are “native” threads; threads 5–8 are “hyper threads” (Workload A; Xeon L5520).

phases. After partitioning (first phase), hash tables are created for all partitions (second phase). Only then, in the third algorithm phase, are those hash tables probed to find join partners. Effectively, the created hash tables will long be evicted from CPU caches before their content is actually needed for probing. Our code avoids these unnecessary memory round-trips by running the build and probe phases for each partition together.

3.5.4 Speedup from SMT Threads

Figure 3.13 shows that neither of the two radix join implementations that we evaluated can significantly benefit from SMT threads. Up to the number of physical cores, both implementations scale linearly, and in the SMT threads region both suffer from the sharing of hardware resources (i.e., caches, TLBs) between threads. These results are also in line with the results of Blanas et al. [BLP11]. As pointed out before, cache-efficient algorithms cannot benefit from SMT threads to the same extent since there are not many cache misses to be hidden by the hardware. The results are also useful in validating our code against that of Kim et al. [KSC+09]. With our optimized implementation, we achieve a speedup of 4.6, very close to the 4.4 factor reported by Kim et al. on a similar Intel Nehalem processor (at comparable absolute performance).
3.6.1 Effect of Workloads

The results of extensive experiments over all workloads and hardware platforms are summarized in Figure 3.14. Figure 3.14(a) shows the performance of our own implementation using Workload A on several hardware platforms (this workload is the one used by Blanas et al. [BLP11]).

While Blanas et al. [BLP11] report only a marginal performance difference between no partitioning and radix join on x86 architectures, in our results the hardware-conscious radix join is appreciably faster when both implementations are equally optimized. Only on the Sun Niagara the situation looks different. We will look into this architecture in the next sub-section.

The results in Figure 3.14(a) may still be seen as a good argument for the hardware-oblivious approach. An approximate 25% performance advantage, e.g., on the two Intel platforms might not justify the effort needed for parameter tuning in radix join.

However, running the same experiments with our second workload, Workload B (Figure 3.14(b)), radically changes the picture. Radix join is approximately 3.5 times faster than no partitioning on Intel machines and 2.5 times faster on AMD and Sun machines. That is, no partitioning has comparable performance to radix join when the relative relation sizes are very different. This is because in such a situation, the cost of the build phase is minimized. As soon as table sizes grow and become similar, the overhead of not being hardware-conscious becomes clearly visible (see the differences in the build phases for no partitioning).

3.6.2 Scalability

To study the scalability of the two join variants, we re-ran the experiments with a varying number of threads, up to the maximum number of hardware contexts available on each of the architectures. Figure 3.15 illustrates the results.

Besides the SMT issues that we already discussed in Sections 3.4.3 and 3.5.4, all platforms and both join implementations show good scalability. Thanks to this scalability, the optimized radix join implementation reaches a throughput of 196 million tuples per second. As far as we are aware, this is the highest throughput reached for in-memory hash joins on a single CPU socket so far.

On the AMD machine, no partitioning shows a clear bump around 8–10 threads. This is an artifact of the particular AMD architecture. Though the Opteron is marketed as a 16-core processor, the chip internally consists of two interconnected CPU dies [CKD+10]. It is likely that such an architecture requires a tailored design for the algorithms to perform well, removing an argument in favor of hardware-conscious algorithms as, even if it is parameter-free, some multi-core architectures
Figure 3.14: Cycles per output tuple for hardware-oblivious no partitioning and hardware-conscious radix join algorithm, for different hardware architectures and workloads. Experiments based on our own, optimized code. Using 8 threads on Nehalem, 16 threads on Sandy Bridge and AMD, and 64 threads on Niagara.
Figure 3.15: Throughput comparison of algorithms on different machines using Workload B. Computed as input-size/execution-time where input-size $= |R| = |S|$.
3.6. Hardware-Conscious or Not?

Table 3.5: Latch cost per build tuple in different machines.

<table>
<thead>
<tr>
<th>Used instruction</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Bulldozer</th>
<th>Niagara 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchgb</td>
<td>~20 cycles</td>
<td>~25 cycles</td>
<td>~50 cycles</td>
<td>3 cycles</td>
</tr>
<tr>
<td>ldstub</td>
<td>7-9 cycles</td>
<td>6-9 cycles</td>
<td>30-34 cycles</td>
<td>1-1.5 cycles</td>
</tr>
</tbody>
</table>

Table 3.5: Latch cost per build tuple in different machines.

may require specialized designs anyway. Note that NUMA would create significant problems for the shared hash table used in no partitioning (let alone in future designs where memory may not be coherent across the machine).

3.6.3 Sun UltraSPARC T2 “Niagara”

On the Sun UltraSPARC T2, a totally different architecture than the x86 platforms, we see a similar result with Workload B. Hardware-conscious radix join achieves a throughput of 50 million tuples per second (cf. Figure 3.15(d)), whereas no partitioning achieves only 22 million tuples per second.

However, when looking to Workload A, no partitioning becomes faster than radix join on the Niagara 2 (shown in Figure 3.14(a)). One could attribute this effect to the highly effective on chip multi-threading functionality of the Niagara 2. However, there is more than that. First, the virtual memory page size on UltraSPARC T2 is 8KiB and the TLB is fully associative, which are significant differences to other architectures.

Second, the Niagara 2 architecture turns out to have extremely efficient thread synchronization mechanisms. To illustrate that, we deliberately disabled the latch code in the no partitioning join. We found out that the ldstub instruction which is used to implement the latch on UltraSPARC T2 is very efficient compared to other architectures as shown in Table 3.5. These special characteristics of Sun UltraSPARC T2 also show the importance of architecture-sensitive decisions in algorithm implementations.

3.6.4 Barrier Synchronization and Load Balancing

In the two join variants that we consider in this work, parallel execution threads synchronize in two ways: (i) write accesses to the shared hash table in no partitioning are protected by latches; (ii) both join variants operate in multiple phases which are separated by barriers. While barrier synchronization allows threads to perform a lot of work independently, there is a risk for wasted idle time when work
is distributed unevenly over threads, so threads have to wait for each other.

Barrier synchronization is not a problem for the no partitioning join execution strategy since, by construction, tuples are distributed evenly across threads and per-tuple cost is basically independent of the tuple values.

Radix join, by contrast, is more vulnerable to penalties due to barrier synchronization whenever tasks are not scheduled properly over available worker threads (we discussed a related issue, the cache locality problem of the radix join implementation of [BLP11], already in Section 3.5.3). In total, radix join consists of five processing stages (assuming a two-pass partitioning scenario): ① compute local histogram for $R$; ② compute local histogram for $S$; ③ partitioning pass 1; ④ partitioning pass 2; ⑤ join phase (partition-wise build and probe). And while threads are guaranteed to receive an equal share of input data in the first three stages, partition sizes produced by stage ④ depend on the distribution of values in $R$ and $S$.

To study the potential load imbalance, we modified our data generator to produce a heavily skewed input data set. Foreign keys in $S$ no longer reference keys in $R$ with a uniform likeliness, but according to a Zipf distribution law with Zipf factor $z = 1.5$. Figure 3.16(a) illustrates, for each of the eight threads in our system ($x$ axis), the type of work it is doing as time progresses (along the $y$ axis).

As can be seen in the figure, all threads perform useful work near the beginning of each execution stage (indicated through different gray shades). But some finish their stage earlier than others, meaning that they have to wait until their last peer finishes the stage (threads 7 and 4 in the figure). The resulting idle times, indicated as $\circ$, waste CPU resources without any real thread progress.

Fine-Granular Task Decomposition. The barrier synchronization problem in Figure 3.16(a) is a result of the task queueing mechanism that we adopted from [BLP11] to distribute load. This mechanism is insufficient to adapt to skewed input data.

To combat the problem, we modified our radix join implementation to perform task decomposition, similar to the strategy proposed by Kim et al. [KSC+09]. In a nutshell, whenever a partition after stage ③ significantly exceeds its expected size (as it would result from a uniform distribution), we break up the partition into smaller chunks that are handled by all threads in concert. This avoids such partitions that can “hog” one of the execution threads and affect overall throughput.

Figure 3.16(b) illustrates the effect on the execution progress. The modification successfully avoids load imbalances and speeds up join execution by about 25%. Though the improved scheduling mechanism applies mainly to the radix join algorithm, we note that its realization is actually parameter-free (and not in itself a hardware-conscious optimization).
3.6. Hardware-Conscious or Not?

Figure 3.16: Barrier synchronization cost in *radix join* (Workload A; foreign key distribution in S skewed with Zipf parameter $z = 1.5$; tasks that make progress are indicated using shades of gray; wait time for barrier synchronization indicated as \(\text{●} \)). Simple task queueing leaves many tasks under-utilized (leading to significant wait times \(\text{●} \) in (a)). Fine-granular task decomposition in (b) (similar to [KSC*09]) improves load distribution and increases join throughput by 25%.
3.6.5 Skewed Data

In this section, we study the effects of skew following the same methodology of Blanas et al. [BLP11]. More specifically, we populate the foreign key column (table $S$) of our data sets such that the probability of referencing individual key values (of $R$) follows a Zipf distribution law (we varied the Zipf factor between $z = 0$ and $z = 1.75$).

Figure 3.17 illustrates how no partitioning and radix join react to skew. The graphs confirm that skew helps the performance of the hardware-oblivious no partitioning join, which was observed already by Blanas et al. [BLP11] and claimed “a big advancement over state-of-the-art” methods. Ultimately, no partitioning surpasses radix join in join throughput when using Workload A.

The observation does not come as a surprise, however, and only happens for data that is heavily skewed. For instance, in the “low skew” case of [BLP11] ($z = 1.05$), the most frequent value in $S$ occurs with a probability of 8.4%; the chance to hit one of the 600 most frequent join keys (out of 16 million) already exceeds 50%. For the “high skew” case of [BLP11] ($z = 1.25$), more than 22% of all $S$ tuples carry the same value and the chance to hit one of the top-600 values is more than 83%. Effectively, even a small L1 cache is sufficient to hold the small hot set of $R$ that is relevant during the probe phase.

Our results indicate that the benchmark configuration of [BLP11] (very high skew, suitable relation sizes) hits a sweet spot of the no partitioning algorithm. This can be seen also in Figure 3.17(b), where the same experiment with Workload B does not help no partitioning as much as the previous configuration.

Performance improvement with increasing skew can be seen as an advantage of no partitioning. The effect also means, however, that the runtime characteristics of the algorithm becomes dependent on the input data distribution and thus difficult to predict (e.g., by a cost-based query optimizer). Radix join, by contrast, offers predictable performance over a wide range of skew, a characteristic that is desirable in the context of robust query processing, an important and active criterion especially for productive query processors [Gra11].

3.6.6 Effect of Relation Size Ratio

The experiments above show that the relative sizes of the tables to join play a big role in the behavior of the algorithms. In the following set of experiments, we explore the effect of varying relation cardinalities on join performance. For these experiments, we use the Intel Xeon L5520 and fixed the number of threads at 8. We varied the size of the primary key build relation $R$ in the non-equal data set from $1 \cdot 2^{20}$ to $256 \cdot 2^{20}$ tuples. The size of the foreign key relation $S$ is fixed at $256 \cdot 2^{20}$. However, as we changed the size of $R$, we have also adjusted the
3.6. Hardware-Conscious or Not?

Figure 3.17: Join performance when foreign key references follow a Zipfian data distribution (Intel Xeon L5520, 2.26 GHz).
distribution of values in $S$ accordingly (i.e., when $|R| = 1$ M, each tuple in $R$ will match 256 tuples in $S$).

Figure 3.18 shows the cycles per output tuple for each phase as well as the entire run for different sizes of $R$ in a log-log plot.

The results confirm the observations made so far and provide a clearer answer to the controversy between hardware-conscious and hardware-oblivious algorithms. No partitioning does very well when the build relation is very small compared to the large relation. Performance goes down as the size of $R$ increases because of the cost of the build phase (Figure 3.18(a)). Radix join is much more robust to different table sizes and offers almost constant performance across all sizes of $R$. More importantly, the contribution of the partitioning phase is the same across the entire range, indicating that the partitioning phase does its job regardless of table sizes.

In other words, no partitioning join is better that radix join only under skew and when the sizes of the tables being joined significantly differs. In all other cases, radix join is better (and significantly better in fact) in addition to also being more robust to different parameters like skew or relative table sizes.

### 3.6.7 TLB and Virtual Memory Page Sizes

In-memory hash joins are known to be sensitive to the virtual memory subsystem of the underlying system, in particular to the caching of address translations via translation look-aside buffers (TLBs). The virtual memory setup of modern systems is, to a small extent, configurable. By changing a system’s page size, every address mapping (potentially cached in the TLB) covers a different amount of main memory, and with a large page size, fewer TLB entries might be needed for the operations on a given memory region.

Intel Nehalem hardware can essentially be operated in either of two modes with the support of the OS [Int12a]: (i) with a page size of 4 KiB (the default), the level 1 data TLB can hold up to 64 memory mappings; (ii) alternatively, when the page size is set to 2 MiB, only 32 mappings can be cached in TLB1. Here we study the effect of these two options on join performance.

**No Partitioning Joins.** During the hash table build and probe phases, the hardware-oblivious no partitioning join algorithm randomly accesses an element in the hash table that is created for the smaller join relation $R$. For our workload configuration $A$, this hash table is 384 MiB in size (tuples plus latches and bucket structure). Consequently, the chance to hit a memory page that is cached in TLB1 is $64/98304$ ($= 1/1536$) or $32/192$ ($= 1/6$), depending on whether the system is configured for a 4 KiB or 2 MiB page size (respectively). The latter configuration might significantly reduce the number of TLB misses and thus improve execution performance.
3.6. Hardware-Conscious or Not?

Figure 3.18: Cycles per output tuple with varying build relation cardinalities in Workload A (Intel Xeon L5520, 2.26 GHz, Radix join was run with the best configuration in each experiment where radix bits varied from 13 to 15).
Table 3.6: Performance of no partitioning join when using large pages.

<table>
<thead>
<tr>
<th>No Partitioning Join (Workload A)</th>
<th>4 KiB pages</th>
<th>2 MiB huge pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build cycles per build tuple</td>
<td>57.92</td>
<td>49.74</td>
</tr>
<tr>
<td>Probe cycles per output tuple</td>
<td>26.10</td>
<td>22.88</td>
</tr>
<tr>
<td>Overall cycles per output tuple</td>
<td>29.72</td>
<td>25.99</td>
</tr>
</tbody>
</table>

Table 3.7: Performance of radix join when using large pages.

<table>
<thead>
<tr>
<th>Radix Join (Workload A)</th>
<th>4 KiB pages (2 pass / 14 bits)</th>
<th>2 MiB huge pages (2 pass / 14 bits)</th>
<th>2 MiB huge pages (1 pass / 12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning cycles per input tuple</td>
<td>19.73</td>
<td>21.71</td>
<td>15.54</td>
</tr>
<tr>
<td>Join cycles per output tuple</td>
<td>2.77</td>
<td>2.75</td>
<td>3.64</td>
</tr>
<tr>
<td>Overall cycles per output tuple</td>
<td>23.74</td>
<td>25.81</td>
<td>20.15</td>
</tr>
</tbody>
</table>

Additionally, modern processors contain paging-structure caches, which become far more effective with a smaller number of total pages.

As listed in Table 3.6, we could indeed observe a performance improvement for no partitioning with larger pages. The dominating cost of no partitioning are actual data cache misses, however, which are unaffected by the page size configuration. This is why the performance improvement remains limited to about 15% in our configuration.

Radix Join. Our hardware-conscious algorithm, radix join, is more sensitive to TLB behavior. In fact, the TLB size is often considered the limiting factor that determines the maximum number of partitions that can be created per partitioning pass. Since the 64-entry TLB1 of our system is assisted by a 512-entry shared TLB2, our Nehalem system actually achieved best join performance with two 128-way passes (cf. Section 3.5.1).

Changing the system page size now may have opposing effects. On the one hand side, a 2 MiB page size reduces the number of available TLB entries (only 32 TLB1 entries). But on the other hand side, the in-memory page table structure of the system’s virtual memory setup becomes smaller; fewer page tables have to be traversed for every TLB miss. In effect, the cost of a single TLB miss gets reduced.

Table 3.7 illustrates what these opposing effects mean to the join performance.
of our Nehalem system. For the workload we used, changing the system page size to 2 MiB shifted the optimal \textit{radix join} configuration to a single-pass 12-bit partitioning phase (with a throughput improvement of $\approx 15\%$).

\textit{Large Pages or Not?} The above measurements indicate a performance advantage of systems that use a large page size configuration. But we note that this is a two-edged sword. Large pages generally increase the memory footprint of processes in the system, which in productive systems might be more problematic than in our micro-benchmarks.

In our benchmarks, both join strategies equally benefit from large pages, leaving the \textit{“hardware-conscious or not?”} question unchanged. As already mentioned above, the small benefit from large pages might also disappear as input data sizes are scaled up in future systems.

Large pages may have a more significant effect on the performance of \textit{no partitioning}—but only if their use is combined with hardware-conscious optimizations such as explicit data prefetching. We detail this combination in Section 3.7.1 where we discuss hardware-conscious optimizations for \textit{no partitioning} hash join.

### 3.6.8 Further Experiments on Large Virtual Memory Pages

In this section, we perform a number of in-depth experiments in order to further evaluate the impact of large virtual memory pages to in-memory join processing in a broader aspect. In Section 3.6.7, we discussed the impact of virtual memory page sizes on the performance of in-memory hash joins. Our results showed that when running Workload A on our Intel Nehalem machine, the use of larger pages slightly improved the performance of both implementation strategies. It turns out, this effect is quite sensitive to the used hardware architecture (\textit{e.g.}, Intel or AMD) and to the input data characteristics.

#### Input Data Sizes

In Section 3.6.6, we studied the effect that the relative sizes of $R$ and $S$ have on the runtime of our two join alternatives. In Figure 3.19, we repeat the same experiment, but add configurations where we used large virtual memory pages. All measurements in this figure were performed on an Intel Nehalem system.

The runtime characteristics that we see in Figure 3.19 is a consequence of two somewhat opposing effects when the page size is changed from 4 KiB to 2 MiB: \textit{(a)} TLB1 in our Intel Nehalem system can hold only 32 entries when configured for 2 MiB pages, but 64 entries for 4 KiB pages; \textit{(b)} the page table tree becomes less deep in the 2 MiB case, hence, a TLB miss incurs a lower cost.

In case of the \textit{no partitioning} join, a larger build relation size leads to an increased number of TLB misses (for both page size configurations). As can be
Figure 3.19: Cycles per output tuple with varying build relation cardinalities in Workload A (Using 8 threads on Intel Xeon L5520, 2.26 GHz, Radix join was run with the best configuration in each experiment).
seen in Figure 3.19(a), this emphasizes the latter of the above two effects. The relative advantage of the 2 MiB configuration over the 4 KiB configuration improves from 15% for a small build relation to 30% for larger build relations.

The page size configuration might affect \textit{radix join} in two ways:

\textit{(a)} the reduced number of TLB entries (32 vs. 64) may reduce the \textit{fanout} that can efficiently performed in each partitioning phase;

\textit{(b)} in the final radix pass, a single TLB entry might cover multiple partitions (each of which should later fit into L1 caches) in a 2 MiB page size configuration.

Effectively, this might shift the sweet spot configuration that minimizes the overall cost (partitioning cost plus probe cost). In our benchmark setting (cf. Figure 3.19(b)) this favors 4 KiB pages when the build relation is very small and large pages when the build relation size increases.

\textbf{Machine Architectures}

The Intel Nehalem system discussed above features 64 TLB1 entries (plus 512 shared TLB2 entries) when the page size is set to 4 KiB, but only 32 entries for a 2 MiB page size configuration. In the AMD Bulldozer architecture, the second test platform that we used, the number of TLB entries does not depend on the configured page sizes. Our AMD Bulldozer features a fully associative 32-entry TLB1 and an 8-way associative 1024-entry TLB2 [AMD12].

As a consequence, page size configurations have less of an effect on our AMD machine, as can be seen in Figure 3.20. \textit{No partitioning} (Figure 3.20(a)) can benefit from large pages only when the size of the build relation grows very large. Conversely, there is a slight improvement for small build relation sizes in case of \textit{radix join} (Figure 3.20(b)).

The exact behavior of both algorithms on this machine is rather hard to model. We constrained our benchmark and pinned all sixteen join threads to a single CPU package. This CPU package, however, is internally divided into two NUMA regions. The whole machine consists of four sockets (eight NUMA regions) in total, so we see an interplay of NUMA effects and the hybrid broadcast/directory-based coherency model of the AMD architecture [CKD+10].

Overall, the effect of large pages on performance stays within about 25%, similar to what we observed also on the Intel Nehalem machine.

\subsection{Evaluation on Recent Multi-Core Servers}

In this section, we extend our evaluation of algorithms to two more recent high-end multi-core servers. The first one is Oracle Sparc T4 [SGG+12] which comes in a
Figure 3.20: Cycles per output tuple with varying build relation cardinalities in Workload A (Using 16 threads on AMD Bulldozer Opteron 6276, 2.3 GHz, Radix join was run with the best configuration in each experiment).
3.6. Hardware-Conscious or Not?

Figure 3.21: Performance on recent multi-core servers, Sparc T4 and Sandy Bridge (SB) using Workload B. Throughput is in output tuples per second, i.e. \( \frac{|S|}{\text{execution time}} \).

four-socket configuration and the specifications of this server is shown in Table 3.2. In comparison to the previous generations of Sparc (such as T2), T4 provides many improvements such as aggressive clock frequency of 3.0 GHz, advanced branch prediction, out-of-order execution and shared L3. The system runs Solaris 11.1 (SunOS 5.11).

The second server is an Intel Sandy Bridge E5-4640 with a four-socket configuration also shown in Table 3.2. Each CPU socket has similar specifications to that of E5-2680 shown in Table 3.2 except the clock frequency of 2.4 GHz and a total memory of 512 GiB. In addition, The system runs Debian Linux 7.0, kernel version 3.4.4-U5 and it uses 2 MiB VM pages for memory allocations transparently.

Figure 3.21 shows the performance of different algorithms on Sparc T4 and Sandy Bridge. First, the \textit{no partitioning} algorithm draws its true benefit from effective on-chip multi-threading on T4. When eight threads are executed on a single physical core, the underlying system takes care of problems such as cache and TLB misses. The performance comparison of running a single thread per core (cf. \( \rightarrow \)) vs. eight threads per core (cf. \( \rightarrow \)) clearly demonstrates this phenomena with a speedup factor of more than four. On the other hand, \textit{radix} join also scales linearly while using all the physical cores. However, it does not benefit from
Chapter 3. Hash Joins

Is on-chip multi-threading or SMT always beneficial?

Simultaneous multi-threading (SMT) hardware provides the illusion of an extra CPU by running two threads on the same CPU and cleverly switching between them at the hardware level. This gives the hardware the flexibility to perform useful work even when one of the threads is stalled, e.g., because of a cache miss.

The results on T4 in Figure 3.21 highly suggest that hardware-oblivious algorithms benefit from SMT hardware. However, the question is whether this is a particular feature of T4 (such as aggressive SMT) or this can be generalized to other hardware. In order to answer this question, we conducted experiments on the Nehalem system which contains four cores with two hardware contexts each (experiments with other Intel SMT hardware produced similar results). In the experiment, we start by assigning threads to different physical cores. Once the physical cores are exhausted, we assign threads to the available hardware context in a round-robin fashion.

Figure 3.22(a) illustrates the performance of no partitioning relative to the performance of a single-threaded execution of the same algorithm (“speedup”). The experiment with our optimized implementation of the no partitioning join indicate that SMT does not help the hardware-oblivious no partitioning strategy at all.

As the result shows, in this particular Intel SMT hardware, SMT can only remedy cache miss latencies if the respective code contains enough cache misses...
and enough additional work for the second thread while the first one is waiting. For code with less redundancy, SMT brings only a negligible benefit.

Similarly, Figure 3.22(b) shows that our \textit{radix join} implementation also cannot significantly benefit from SMT threads. Up to the number of physical cores, the performance scale linearly, and in the SMT threads region it suffers from the sharing of hardware resources (i.e., caches, TLBs) between threads. Cache-efficient algorithms cannot benefit from SMT threads to the same extent since there are not many cache misses to be hidden by the hardware.

Overall, results in Figure 3.21, 3.22(a) and 3.22(b) indicate that aggressive SMT can make hardware-oblivious algorithms competitive but this only occurs in particular hardware and cannot be generalized. On Intel architectures, tuning to the underlying hardware clearly still matters.

\textbf{Impact of architecture aware compilation}

Another interesting fact on the T4 is the compiler optimizations. Using Sun Studio compiler 12.3 with compilation options \texttt{"-xO4 -xipo -fast -xtarget=T4 -Bdynamic"} resulted in $\approx 30\%$ performance improvement over \texttt{gcc 4.7.1.} with options \texttt{"-mtune=niagara4 -mcpu=niagara4 -O3"}. This fact highlights the benefit of architecture aware compilation optimizations.

\textbf{Multi-socket deployment and scalability}

Looking at the performance of \textit{radix} join on the Intel Sandy Bridge machine (cf. in Figure 3.21), we observe that the x86 retains a significant performance advantage over the Sparc architecture despite the radical architectural changes and improved single thread performance on T4. Meanwhile, the performance of \textit{radix} is still superior to that of \textit{no partitioning} on Intel almost by a factor of five, showing the importance of hardware-consciousness on this architecture (not shown in the graph). Finally, this experiment also demonstrates the scalability of hash join algorithms beyond a single CPU socket and exhibits the fastest performance of a radix hash join implementation to date ($\approx 400$-650 M/s).

\subsection*{3.6.10 When Does Hardware-Conscious Not Work Well?}

Some of the architectural features have an important impact on the performance of the algorithms. In this section, we shed a light on such features and highlight what makes a hardware-conscious algorithm such as \textit{radix} join to misbehave.
Chapter 3. Hash Joins

![Figure 3.23: Impact of number of threads on Sparc T4 on overall performance for different algorithms. Using Workload B (977 MiB ⋊ ⋉ 977 MiB); Oracle Sparc T4.](image)

Less is more for hardware-conscious algorithms

In dynamically multi-threaded architectures, most of the physical core resources are shared among all active threads. The Sparc T4 architecture takes this to an extreme where resources such as caches, load/store buffer entries, DTLB, hardware table-walk engine, and branch prediction structures are dynamically shared between all the eight threads running on a core. This usually works well for applications with a mix of high and low IPC threads running concurrently and provides a better utilization of system resources. However, in case of hardware-conscious algorithms such as radix join, this introduces detrimental effects to performance.

The performance of radix join with different number of threads is shown on the left side of Figure 3.23. In the case with 32 threads, each thread runs on an individual physical core and it does not contend with other threads for the core resources. In case of 256 threads, each physical core runs eight threads concurrently where all the core resources are shared. Surprisingly, using more threads ruins the hardware-consciousness of radix join making it slower than the 32-thread case. The performance problem can be clearly seen in the actual join phase of the algorithm, which is the cache-sensitive, high IPC phase of the algorithm. The eight threads contend for the same L1/L2 space and due to excessive cache misses this phase slows down. Essentially, the cache-conscious nature of radix join becomes destroyed.
Incongruent cache architecture

The cache architecture plays an important role for hardware-conscious algorithms. In the Sparc T4, the 16 KiB L1 cache is relatively small in comparison to x86 architectures. First, the small cache size requires a higher partitioning fan-out which makes the entire partitioning costlier in this architecture. Second, the maximum partitioning fan-out is also limited by the L1 size. Due to the use of a fully-associative 128-way DTLB and large pages, TLB misses are no longer the main bottleneck in partitioning. However, the L1 cache can hold 512 cache lines and puts a hard limit at around 512 for efficient partitioning fan-out per pass. Moreover, the 4-way associativity of the L1 cache does not match the number of eight hardware threads. Therefore, accesses from different threads can potentially cause unnecessary conflict misses. In addition, the 32 bytes L1/L2 cache line size reduces the potential benefit from sequential writes during partitioning compared to a cache line size of 64 bytes in x86 architecture. Last but not least, the relatively small shared L3 brings almost no benefit to a hardware-conscious algorithm such as radix join as the effective L3 share per thread ($\approx 64$ KiB) is less than the size of the L2 cache. Overall, all these characteristics of the cache architecture in Sparc T4 makes the radix join ill-behaved in this architecture and perform less than its potential.

3.7 Further Hardware-Conscious Optimizations

First, hardware-oblivious algorithms can be turned into hardware-conscious ones by using software prefetching and cache alignment optimizations, both of which become more effective when using large virtual memory pages. Second, by using new optimization techniques like software-managed buffers, performance of hardware-conscious algorithms can also be further improved. This section visits these optimizations and first outlines how to turn no partitioning into a hardware-conscious algorithm and then describes software-managed buffers technique for radix join.

3.7.1 Making No Partitioning Hardware-Conscious

In this section, we propose hardware-conscious optimizations such as a combination of cache alignment, software-prefetching, and use of large pages to improve the performance of no partitioning join. These techniques abandon the strictly hardware-oblivious nature of the original algorithm but require little parameter tuning.
Chapter 3. Hash Joins

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Build</td>
<td>Probe</td>
</tr>
<tr>
<td>L2 misses</td>
<td>2.97</td>
<td>2.94</td>
</tr>
<tr>
<td>L3 misses</td>
<td>2.72</td>
<td>2.65</td>
</tr>
</tbody>
</table>

Table 3.8: No partitioning join; cache misses per tuple (original code of Blanas et al. [BLP11] vs. our own implementation).

Cache Alignment

As discussed in Section 3.4.2, the no partitioning implementation of Blanas et al. [BLP11] uses a hash table design where up to three accesses to different memory locations are needed to access a single hash bucket (latch array, pointer array, and actual data buckets; cf. Figure 3.6). To avoid this potential memory access bottleneck, in our own code we wrapped the necessary latches into the bucket data structure and removed the indirection caused by the pointer array of Blanas et al. In effect, only a single record needs to be accessed per data tuple. Only true hash collisions will require extra bucket fetches.

Using our profiling framework, we measured the number of cache misses required per build/probe tuple in both of the implementations (cf. Table 3.8). Somewhat counter-intuitively, the number of misses per tuple is considerably higher, however. This is most noticeable during the build phase of our own implementation, where we see more than 1.5 misses/tuple even though only a single hash bucket must be accessed per tuple.

The reasons for this is the lack of cache alignment of both hash table implementations. As illustrated in Figures 3.6 and 3.7, both hash table implementations use a bucket size of 48 bytes. If such buckets are packed one after another, a single bucket access may span two cache lines and thus cause more than a single cache miss on access. Specifically, four 48-byte buckets will occupy three successive cache lines. On average, each bucket intersects with 1.5 cache lines, which coincides with the cache miss numbers shown in Table 3.8.

Hash buckets can be forced to stay within a cache line by aligning them to 64-byte boundaries. As the last two columns in Table 3.8 show, changing no partitioning in this way reduces the cache miss rate to the expected one miss per tuple.
3.7. Further Hardware-Conscious Optimizations

Software Prefetching

Another way to avoid cache misses is the use of prefetching. Chen et al. [CAGM07, CAGM04], for instance, described how hash table accesses like ours can be accelerated by issuing software prefetch instructions. If those instructions are issued early enough, the CPU can overlap memory accesses with instruction execution and thus hide memory access latencies.

We applied the prefetching mechanisms of Chen et al. to our no partitioning implementation. The proper prefetch distance is a hardware-specific parameter, which we manually tuned to the behavior of our machine. The effect of this optimization is illustrated in Figure 3.24 for Workloads A and B (bars labeled “w/ prefetch”).

Figure 3.24 also illustrates the effect of cache alignment (bars labeled “aligned”). Interestingly, cache alignment alone does not significantly improve the performance of no partitioning. Both optimizations together, however, can improve the throughput of no partitioning by more than 40%. To achieve this improvement, however, we had to give up the strictly hardware-oblivious nature of no partitioning and introduce tuning parameters such as prefetch distance and cache line size.

Figure 3.25, in fact, illustrates how sensitive our code changes are to the underlying hardware platform. When running the same experiment on the AMD Opteron machine, aligning hash buckets to the cache line size has a significant impact on overall throughput. Software prefetching can improve only little over that. Together, both optimizations again yield a performance gain of ≈ 40% over...
Chapter 3. Hash Joins

3.7.2 Improving Radix with Software-Managed Buffers

We observed in the earlier experiments (e.g., in Section 3.6.1) that the partitioning time almost dominates the overall execution time in radix join. In this section, we describe the software-managed buffers technique to improve the performance of radix partitioning.

Conceptually, each partitioning phase of radix join takes all input tuples one-by-one and writes them to their corresponding destination partition (pos[] keeps track of the current write location within each partition):

```plaintext
1 foreach input tuple t do
2     k ← hash(t);
3     p[k][pos[k]] = t; // copy t to target partition k
4     pos[k]++;
```

Generally, partitions are far apart and on separate VM pages. Thus, if the fanout of a partitioning stage is larger than the number of TLB entries in the system, copying each input tuple will cause another TLB miss. Typically, the
Figure 3.26: Partitioning performance comparison when using 4 KiB and 2 MiB pages (Using a single core on Intel Xeon L5520, 2.26 GHz).
number of TLB entries is considered an upper bound on the partitioning fanout that can be realized efficiently.

This TLB miss count can be reduced, however, when writes are first buffered inside the cache. The idea is to allocate a set of buffers, one for each output partition and each with room for up to $N$ input tuples. Buffers are copied to their final destination only when they are full:

```
1 foreach input tuple $t$ do
  2   $k \leftarrow \text{hash}(t)$;
  3   $\text{buf}[k][\text{pos}[k] \mod N] = t$; // copy $t$ to buffer
  4   $\text{pos}[k]++$;
  5   if $\text{pos}[k] \mod N = 0$ then
      6     copy $\text{buf}[k]$ to $p[k]$; // copy buffer to partition $k$
```

Obviously, buffering leads to additional copy overhead. However, for sufficiently small $N$, all buffers will fit into a single memory page. Thus, a single TLB entry will suffice unless a buffer becomes full and the code enters the copying routine in line 6. Beyond the TLB entry for the buffer page, an address translation is required only for every $N$th input tuple, significantly reducing the pressure on the TLB system. As soon as TLB misses become infrequent, it is likely the CPU can hide their latency with its usual out-of-order execution mechanisms.

The software-managed buffering strategy mentioned above follows the idea of Satish et al. [SKC+10a], which employed the same technique to reduce the TLB pressure of radix sort.

We added an implementation of such software-managed buffers to our radix join code and configured $N$ such that one buffer will exactly fill one cache line (64 bytes); i.e., $N = 4$ for Workload A and $N = 8$ for Workload B. Configuring the buffer size in this manner allows for another low-level optimization. Since we are now always writing a full cache line at once to global memory, the CPU can take advantage of its write combining facilities, thus avoiding to read the cache line before writing it back.

Figure 3.26 illustrates the effect of software-managed buffers on the performance of partitioning. In both figures, we partition a 128 million-tuple data set with 8 bytes per tuple (Workload B) and measure the achievable throughput for single-pass radix partitioning with and without software-managed buffers.

As can be seen in the figure, software-managed buffers indeed cause some copying overhead. But the investment clearly pays off once the available TLB entries are exhausted. At about 8 radix bits (Figure 3.26(a)) the performance of the naïve strategy begins to suffer from the growing TLB miss cost,\(^3\) whereas the

\(^3\)Note that the 64-entry TLB1 is assisted by a 512-entry TLB2.
3.8. Estimating Ideal Parameters of Radix Join

Manegold et al. [MBK02a, MBK02b] develop generic analytical models for estimating the cost of database operations, especially main-memory radix join. Their models take hardware parameters into account and accurately estimate the optimal configuration of radix join based on calibrated hardware specifications such as cache and TLB latencies. Although not impossible, the current landscape in hardware technology makes it more challenging to accurately estimate the costs: The hardware is more complex with multiple cores, SMT threads running on the same chip and more advanced in predicting memory access patterns. For instance, it is very hard to estimate cache misses just based on algorithm behavior as processor also causes cache misses due to automatic prefetching with various–usually not disclosed– strategies. On the other hand, our experimental study of parallel radix hash joins investigated the performance on a broad range of hardware with a comprehensive set of experiments. Therefore, in this section we develop an empirical model for estimating the ideal configuration parameters of radix hash join and evaluate it based on our extensive results.

3.8.1 An Empirical Model for Configuring Radix Join

The essential idea of radix join is to partition the data in a way that the resulting partitions fit into processor caches. In addition, the partitioning should also be cache and TLB aware such that its cost can be compensated by the efficiency of the join. As a result we can use the following as a rule of thumb for determining the number of required partitions:
Rule 1) “The partition sizes resulting from the radix partitioning must fit into either L1 or L2 cache.”

For the TLB efficiency of the radix partitioning the following rule can guide us in addition to the previous rule:

Rule 2) “Number of partitions from the radix partitioning at each pass must be in the order of the TLB entries.”

We translate these rules into an empirical model for estimating the ideal configuration parameters of the radix join. The symbols and their definitions used in this model is shown in Table 3.9. We first compute the ideal number of partitions based L1 and L2 cache sizes. Here we assume that a hash table (e.g., bucket-chained) usually requires the tuple (∥t∥) and a hash table entry (∥h∥) to be resident in the corresponding cache.

\[
N_{L1} = \frac{|R| \cdot (∥t∥ + ∥h∥)}{∥L1∥} \tag{3.1}
\]

\[
N_{L2} = \frac{|R| \cdot (∥t∥ + ∥h∥)}{∥L2∥} \tag{3.2}
\]
We then compute the ideal number of radix bits for partitioning based on the found ideal number of partitions required in Equations 3.1 and 3.2.

\[
B_{L1} = \lceil \log_2(N_{L1}) \rceil \\
B_{L2} = \lceil \log_2(N_{L2}) \rceil 
\]  

(3.3)

**Algorithm 1:** An empirical model for determining ideal number of radix bits and partitioning passes for radix join.

1. \( n = \log_2(|TLB2|/|TLB1|) \);
2. for \( i = 0 \) to \( n \) do  // compute a range of partitioning pass values
   3. \( |TLB| \leftarrow 2^i \cdot |TLB1| \);
   4. \( P_{L1}^i = \lceil B_{L1}/\log_2(|TLB|) \rceil \);
   5. \( P_{L2}^i = \lceil B_{L2}/\log_2(|TLB|) \rceil \);
   6. \( P_{L1}^{min} \leftarrow \infty \);
   7. \( P_{L2}^{min} \leftarrow \infty \);
8. for \( i = 0 \) to \( n \) do  // compute the min partitioning pass
   9. if \( P_{L1}^i < P_{L1}^{min} \) then
      10. \( P_{L1}^{min} = P_{L1}^i \);
   11. if \( P_{L2}^i < P_{L2}^{min} \) then
      12. \( P_{L2}^{min} = P_{L2}^i \);
13. if \( P_{L1}^{min} \leq P_{L2}^{min} \) then  // partitioning is based on L1 cache
   14. number of partitioning passes \( \leftarrow P_{L1}^{min} \);
   15. radixbits \( \leftarrow B_{L1} \);
else  // partitioning is based on L2 cache
   16. number of partitioning passes \( \leftarrow P_{L2}^{min} \);
   17. radixbits \( \leftarrow B_{L2} \);

Finally, the number of required passes for radix partitioning is computed based on radix bits found in Equation 3.3 and TLB hardware sizes. Since modern processors provide a two level TLB hierarchy, the ideal number of partitioning passes should be determined from a range of values based on TLB1 and TLB2 sizes. The procedure for determining ideal number of partitioning passes is described in Algorithm 1. The algorithm first finds a set of pass values by assuming that number of TLB entries vary from the size of first level TLB to second level TLB. At each assumed TLB value, a set of ideal number of partitioning passes is created for both L1 and L2 cache based partitioning (cf. lines 3-5). Afterwards, a minimum number of partitioning passes \( (P_{L1}^{min} \text{ and } P_{L2}^{min}) \) are determined among those set of values (cf. lines 8-12). Finally, the minimum number of partitioning passes and
its partitioning target (either L1 or L2 cache) are identified and the configuration parameters of the radix join are found (cf. lines 13-18). The algorithm essentially enforces creating L1 cache sized partitions as long as the TLB efficiency can be sustained at that level. Otherwise, it resorts back to creating L2 cache sized partitions. For a system with a single level TLB, the procedure remains the same by replacing $|TLB_2|$ also with $|TLB_1|$ in the algorithm.

### 3.8.2 Evaluation of the Empirical Model

In this section we evaluate the empirical model for identifying ideal parameters of radix join. The evaluation is carried out on a number of different machines with different workloads discussed throughout this chapter. We first run a batch of experiments by varying the configurations manually and record achieved performance numbers. The goal of this step is to identify the ideal configurations with exhaustive set of experiments. Later we identify the ideal parameters with the model described in Section 3.8.1. The goal is to assess the capability and accuracy of the proposed empirical model.

**Intel Nehalem L5520**

*Workload A.* The results of the configuration experiment are presented in Figure 3.27. The experiment evaluates the radix join performance with a wide range of configuration options. The shape of the graph validates the behavior of radix join mentioned earlier: Join cost gets cheaper with increasing number of partitions.
3.8. Estimating Ideal Parameters of Radix Join

Table 3.10: Estimation of the radix join configuration parameters using the empirical model for Intel Nehalem L5520 and Workload A.

| |R| ||t|| ||h|| ||L1|| ||L2|| ||TLB|| |N_{L1} | B_{L1} | P_{L1}^0 | N_{L2} | B_{L2} | P_{L2}^0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|2^24| 16| 8| 32768| 262144| 64| 12288| 14| 3| 1536| 11| 2 |
|2^24| 16| 8| 32768| 262144| 128| 12288| 14| 2| 1536| 11| 2 |
|2^24| 16| 8| 32768| 262144| 256| 12288| 14| 2| 1536| 11| 2 |
|2^24| 16| 8| 32768| 262144| 512| 12288| 14| 2| 1536| 11| 2 |

Figure 3.28: Configuration experiment on Nehalem L5520 with Workload B.

whereas partitioning cost gets more expensive. The results indicate that the best performance is achieved with 13 radix bits and 2 partitioning passes. However, performance with 14 radix bits is also very close to the best case and the difference is in the range of measurement noise.

We turn to the empirical model to estimate the ideal parameters. The numbers of the model are summarized in Table 3.10. We observe that number of radix bits are 11 and 14 for L2 and L1 sized partitions, respectively. Moreover, we also see that 2 pass partitioning with L1 sized partitions is feasible. Therefore, the algorithm picks 14 radix bits and 2 partitioning passes as the ideal configuration. Comparing with the best configuration in the previous experiment, we see that the performance is only 1.22% slower.

Workload B. The results of the configuration experiment are presented in Figure 3.28. The results indicate that the best performance is achieved with 14 radix bits and 2 partitioning passes.
Chapter 3. Hash Joins

Table 3.11: Estimation of the radix join configuration parameters using the empirical model for Intel Nehalem L5520 and Workload B.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Workload A</th>
<th>Workload B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Config.</td>
<td>Slow down %</td>
<td>Best Config.</td>
</tr>
<tr>
<td>Model Config.</td>
<td></td>
<td>Model Config.</td>
</tr>
<tr>
<td>Intel Nehalem E5520</td>
<td>1.22 %</td>
<td>1.97 %</td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>12.80 %</td>
<td>14.53 %</td>
</tr>
<tr>
<td>Sun Niagara T2</td>
<td>0.00 %</td>
<td>13.01 %</td>
</tr>
<tr>
<td>Intel E5 4640</td>
<td>0.42 %</td>
<td>12.53 %</td>
</tr>
<tr>
<td>Oracle Sparc T4</td>
<td>2.42 %</td>
<td>21.90 %</td>
</tr>
</tbody>
</table>

Table 3.12: Evaluation of the empirical model.

The numbers obtained from the empirical model to estimate the ideal parameters are summarized in Table 3.11. We observe that number of radix bits are 13 and 16 for L2 and L1 sized partitions, respectively. Moreover, we also see that 2 pass partitioning with L1 sized partitions is feasible. Therefore, the algorithm picks 16 radix bits and 2 partitioning passes as the ideal configuration. Comparing with the best configuration in the previous experiment, we see that the performance is only 3.97 % slower.

Other Machines

We have evaluated the empirical model on other machines following the steps as shown for the Intel Nehalem L5520 above. We present the summary of the evaluations for all the machines in Table 3.12. We noticed that the empirical model requires a special attention about the cache sharing issues, particularly on Sparc architectures. For the shared L2 cache, one should use 1/64 fraction of the cache (i.e., share per hardware thread) in the model to get these accurate numbers. Other than that, essentially, the results indicate that the empirical model estimates the ideal radix join configurations quite successfully. Almost in all the estimations, the slow down compared to the best configuration remains below 20 %. Estimations are usually off by one or two bits and sometimes the model even finds the best configuration.

Lastly, we present an experiment which shows that the success of the empirical model is not specific to the workloads considered in this chapter. The experiment varies the build relation size from $2^{20}$ ($\approx 16.8$ M) tuples to $2^{28}$ ($\approx 268.4$ M) tuples while keeping the probe relation size at $2^{28}$ tuples. The tuple size in this experiment
3.8. Estimating Ideal Parameters of Radix Join

is 16-bytes. At each experiment point, we first determine the best configuration of radix join experimentally by trying various configurations (shown as best config.). We then estimate the radix join configurations with the empirical model (shown as model config.). Figure 3.29 shows the overall execution time with corresponding configurations at each experiment point. The results clearly indicate that the empirical model estimates the configurations quite accurately regardless of the build relation size. Moreover, Figure 3.30 shows the slow down of performance with the model configurations in comparison to the best configurations. The slow down with respect to the best performance remains below 10% showing the effectiveness of the model.

3.8.3 Extending the Empirical Model for Optimizations

The empirical model is successful for determining radix join configurations as shown in the previous experiments. However, the empirical model needs to be extended to cover the optimizations such as use of large pages and partitioning with software-managed buffers. In this section we show its applicability for the new optimizations in radix join and evaluate its performance.

In Section 3.6.8 and 3.6.7, we showed that use of 2 MiB large pages might have opposing effects on the performance of radix join. On Intel architectures, the number of available TLB entries for 2 MiB pages is only 32 and there is no second level TLB support. However, we also showed in Section 3.7.2 that the
costs arising from TLB misses can be significantly avoided by the use of software-
managed buffers technique. The software-managed buffers technique essentially
moves the partitioning fanout limit imposed by TLB size to the number of cache
lines available in L1 or L2 caches. If the partitioning fanout is less than the number
of L1 cache lines, then the partitioning is carried out with L1 cache resident buffers.
Otherwise, for a fanout up to the number of cache lines available in the L2 cache,
the partitioning is carried out with L2 cache resident buffers. Essentially, the
\(|TLB1|\) and \(|TLB2|\) numbers in the empirical model shown in Section 3.8.1 needs
to be replaced by the number of cache lines in L1 (\(|L1|\)) and L2 (\(|L2|\)) caches,
respectively. Moreover, since the optimizations enable larger partitioning fanouts,
creating L2 or L3 sized partitions in less number of passes becomes much more
efficient than creating L1 or L2 sized partitions. The ideal partition sizes shift by
one level in the cache hierarchy. The empirical model can be easily modified to
incorporate this finding. Instead of the previously assumed \(|L1|\) and \(|L2|\) sizes
used in Equations 3.1 and 3.2, one must use \(|L2|\) and \(|L3|\), respectively. As one
more subtle point, the cache sharing issues must also be considered. For instance
in the Intel machines, \(|L2|\) must be assumed as 1/2 of the original L2 cache due
to SMT thread sharing and \(|L3|\) must be assumed as 1/16 of the original L3 cache
due to sharing within the socket.

**Evaluation.** We evaluate the applicability and accuracy of the extended empir-
ical model for the optimized radix join implementation on the Intel Sandy Bridge
E5-4640 machine. Figure 3.31 shows the configuration experiment of radix join
with new optimizations (i.e., use of large pages and partitioning with software-
managed buffers). The results indicate that partitioning fanout up to \(|L2|\) (2^{12})
is quite efficient. Moreover, in contrast to the previous results, the entire join can be
done with a single pass partitioning due to the optimizations. However, one can

![Figure 3.30: Slow down of model configuration from the best configurations at
each instance of the experiment.](image-url)
Figure 3.31: Configuration experiment of radix join with new optimizations on Sandy Bridge E5-4640 with Workload B.

also observe that once the partitioning fanout goes beyond $|L_2|$, the partitioning cost significantly increases. For a partitioning fanout beyond $2^{12}$, the partitioning buffers do not fit into the L2 cache and each tuple in the partitioning causes a cache miss. Therefore, number of L2 cache lines puts a more stricter limit on the partitioning fanout and a fanout larger than $|L_2|$ must be avoided at any case.

After identifying the best configurations, we now turn to the extended empirical model for determining the radix join configuration. The application of the extended empirical model is summarized in Table 3.13. In the model, numbers for $|TLB_1|$, $|TLB_2|$, $|L_1|$ and $|L_2|$ are replaced with $|L_1|$, $|L_2|$, $1/2 \times 256$ KiB and $1/16 \times 20$ MiB, respectively, as described above. Once run with the new numbers, the procedure shown in Algorithm 1 determines 12 radix bits and 1 partitioning passes as the ideal configuration for radix join. The configuration is indeed the best configuration of radix join as shown in Figure 3.31. We have also repeated similar evaluations with different workloads and machines. We found out that the extended model finds radix join configurations close to the best configurations and works well in practice. We omit the details of those experiments which produced similar results.
### Table 3.13: Determining the configuration parameters of radix join with new optimizations using the extended empirical model (Intel E5-4640 and Workload B).

| $|R|$ | $|t|$ | $|h|$ | $|L2|$ ($\frac{1}{2} \times 256$ KiB) | $|L3|$ ($\frac{1}{16} \times 20$ MiB) | $TLB$ ($|L1| \rightarrow |L2|$) | $N_{L2}$ | $B_{L2}$ | $P_{L2}^{0.3}$ | $N_{L3}$ | $B_{L3}$ | $P_{L3}^{0.3}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 M | 8 | 8 | 131072 | 655360 | 512 | 15625 | 14 | 2 | 3125 | 12 | 2 |
| 128 M | 8 | 8 | 131072 | 655360 | 1024 | 15625 | 14 | 2 | 3125 | 12 | 2 |
| 128 M | 8 | 8 | 131072 | 655360 | 2048 | 15625 | 14 | 2 | 3125 | 12 | 2 |
| 128 M | 8 | 8 | 131072 | 655360 | 4096 | 15625 | 14 | 2 | 3125 | 12 | 2 |

#### 3.9 Factorial Analysis of Performance

The purpose of this section is to provide a detailed factorial analysis of some of the important experiments that have been discussed in this chapter. By means of the detailed analysis over the experimental results, the impact of various factors that affect the performance are separated out. Moreover, significant factors are identified analytically through this study. Finally, the findings in the section might guide the query optimizer in choosing the algorithms based on a given set of parameters.

The methodology followed in this section is the following. First, the goal is not add or extend any of the experiments discussed in this chapter. Therefore, some of the already presented experiments will be revisited. For a given experiment, the experimental design, important experimental factors and the question to be answered will be described. Afterwards, the experimental design (e.g., a $2^k$ factorial design) and the analytical setup will be described. Lastly, through the computation of each of the effects, the impact of each experimental factor and its significance will be pointed out.

For the background and terminology on experimental design and factorial analysis, we refer the reader to [Jai91, Part IV].

#### 3.9.1 Analysis of Machines, Workloads and Algorithms

The purpose of this section is to shed a light on the individual impact of several choices on hash join performance. We mainly focus on three main parameters that affect the hash join performance: machine architecture, workload type and algorithm. We provide a detailed factorial analysis considering these three parameters with a $2^k$ factorial design. For the machine architectures we choose Intel Nehalem and Sun Niagara T2 as the representatives of x86 and Sparc architectures, respectively. The other parameters are already considered with two different options throughout the study in this chapter. For workloads we consider Workload A and B as described in Section 3.3. For the algorithms we consider radix join and no partitioning join.
3.9. Factorial Analysis of Performance

Table 3.14: Effect of machine, workload and algorithm on hash join performance in terms of cycles per output tuple metric.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Workload A</th>
<th>Workload B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Radix join</td>
<td>No partitioning join</td>
</tr>
<tr>
<td></td>
<td>Radix join</td>
<td>No partitioning join</td>
</tr>
<tr>
<td>Nehalem</td>
<td>23.68</td>
<td>29.79</td>
</tr>
<tr>
<td></td>
<td>25.91</td>
<td>91.25</td>
</tr>
<tr>
<td>Niagara T2</td>
<td>19.23</td>
<td>10.77</td>
</tr>
<tr>
<td></td>
<td>24.33</td>
<td>52.32</td>
</tr>
</tbody>
</table>

Table 3.14 shows the setup of the experimental design as a $2^3$ factorial design with the performance numbers taken from the experiments shown in Figure 3.14. There are three factors each with two levels. Let us define three variables $x_A$, $x_B$ and $x_C$ as follows:

$$
x_A = \begin{cases} 
-1 & \text{if machine is Nehalem} \\
1 & \text{if machine is Niagara T2} 
\end{cases}$$

$$
x_B = \begin{cases} 
-1 & \text{if workload is Workload A} \\
1 & \text{if workload is Workload B} 
\end{cases}$$

$$
x_C = \begin{cases} 
-1 & \text{if algorithm is radix join} \\
1 & \text{if algorithm is no partitioning join} 
\end{cases}$$

The performance in execution cycles can be regressed on $x_A$, $x_B$ and $x_C$ using a nonlinear regression model as the following:

$$y = q_0 + q_A x_A + q_B x_B + q_C x_C + q_{AB} x_A x_B + q_{AC} x_A x_C + q_{BC} x_B x_C + q_{ABC} x_A x_B x_C$$

The response variable $y$ is the cycles per output tuple metric for the join execution performance and values of $y$ for different experiments are given in Table 3.14. The expression above can be substituted by using different values of $x_A$, $x_B$, $x_C$ and $y$ and eight equations can be solved analytically. However, we opt for the $8 \times 8$ sign table matrix method as shown in Table 3.15.

The matrix method works as follows. The first column of the matrix labeled $I$ consists of all 1’s. The next three columns represent the three main parameters $x_A$, $x_B$ and $x_C$ and labeled with their corresponding letters. The values for these columns contain all possible combinations of 1 and $-1$. The other columns are combination of these main parameters and essentially represent the impact of interaction of individual parameters in overall performance. Their values are computed by column-wise multiplying their constituents. Lastly, the total values for
Table 3.15: Sign table calculation of the effects in hash join performance.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>AB</th>
<th>AC</th>
<th>BC</th>
<th>ABC</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>23.68</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>19.23</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>25.91</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>24.33</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>29.79</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>10.77</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>91.25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>52.32</td>
</tr>
</tbody>
</table>

| 277.28 | -63.98 | 110.34 | 90.98 | -17.04 | 95.68 | -22.78 | Total |
| 34.66  | -7.9975 | 13.7925 | 11.3725 | -2.13 | -6.49 | 11.96 | -2.8475 | Total/8 |

Each column is computed by a vector dot product of entries in the corresponding column and the values of response under y. The mean value of each column is then computed by dividing the total values by eight. The computed mean values for each of the factors and their interaction represent the mean impact of each factor in the overall performance.

**Allocation of Variation.** Now, we proceed with the importance of each factor. The importance of a factor is determined by the proportion of the variation that a factor causes in the total variation. Therefore, we first compute the total variation in the output cycles per tuple metric, in other words **Sum of Squares Total (SST):**

\[
\text{Total variation of } y = SST = \sum_{i=1}^{2^3} (y_i - \overline{y})^2 = 2^3q_A^2 + 2^3q_B^2 + 2^3q_C^2 + 2^3q_{AB}^2 + 2^3q_{AC}^2 + 2^3q_{BC}^2 + 2^3q_{ABC}^2
\]

Moreover, SST equals to the sum of the variation explained by the effects of A (SSA), B (SSB), C (SSC) and combination of interactions AB (SSAB), AC (SSAC), BC (SSBC) and ABC (SSABC):

\[
SST = SSA + SSB + SSC + SSAB + SSAC + SSBC + SSABC
\]

Finally, the fraction of variation explained by each factor can be found as follows:

\[
\text{Fraction of variation explained by factor } F = \frac{SSF}{SST}
\]
3.9. Factorial Analysis of Performance

<table>
<thead>
<tr>
<th>SSA</th>
<th>SSB</th>
<th>SSC</th>
<th>SSAB</th>
<th>SSAC</th>
<th>SSBC</th>
<th>SSABC</th>
<th>SST</th>
</tr>
</thead>
<tbody>
<tr>
<td>511.68005</td>
<td>1521.86445</td>
<td>1034.67005</td>
<td>36.2952</td>
<td>336.9608</td>
<td>1144.3328</td>
<td>64.86605</td>
<td>4650.6694</td>
</tr>
</tbody>
</table>

% explained: 11.00 % 32.72 % 22.25 % 0.78 % 7.25 % 24.61 % 1.39 % 100 %

Table 3.16: Allocation of variation and percentage explained by each factor.

For our analysis, sum of squares total and variation explained by each factor is computed and presented in Table 3.16. The variation explained by the workload turns out to be the most significant (32.72 %). Therefore, it points out that the workload under consideration is the most important factor when comparing two algorithms for performance. The variation explained by the interaction of machine and algorithm is found to be the second most significant factor (24.61 %). The third most significant factor is found to be the algorithm (22.25 %). Other than these parameters, the others seem to be relatively less important. For instance, it is hard to decide which algorithm would perform best merely on the machine type without knowing about the workload. Overall, the analysis provides an important insight for the query optimizers: A query optimizer should use the workload at hand as the first determinant for which algorithm to choose. The type of the machine can then be used as the second determinant.

3.9.2 Analysis of Optimizations on No Partitioning Join

In Section 3.7.1 we have discussed several hardware-conscious optimizations to hardware-oblivious no partitioning join. The accompanying experiments demonstrated the empirical benefits of each of the optimizations. The goal of the present discussion is to identify the quantitative impact of each of these optimizations on no partitioning join performance. We focus on four parameters that affect the performance: machine type, virtual memory page size, software prefetching and hash table bucket alignment to cache lines. The machines we consider are Intel Nehalem and AMD Bulldozer as described in Section 3.3. The workload is fixed at Workload A for all of the experiments. Virtual memory page size can be either 4 KiB or 2 MiB. Lastly, software prefetching and cache alignment is either enabled or disabled. We separate the analysis into two where we analyze the experimental factors first on the Intel Nehalem machine then on the AMD Bulldozer machine.

Analysis of Optimizations on Intel Nehalem

Table 3.17 summarizes the results of experiments where we varied all of the factors discussed in this section. The numbers are taken directly from the experiments shown in Figure 3.24 and 3.25. For the first analysis we setup a $2^3$ experimental
Table 3.17: Impact of machine and optimizations no partitioning join performance in terms of cycles per output tuple metric.

Design with three factors each with two levels. Let us define three variables $x_A$, $x_B$ and $x_C$ as follows:

$$x_A = \begin{cases} 
-1 & \text{if VM Page size is 4 KiB} \\
1 & \text{if VM Page size is 2 MiB}
\end{cases}$$

$$x_B = \begin{cases} 
-1 & \text{if prefetching is enabled} \\
1 & \text{if prefetching is disabled}
\end{cases}$$

$$x_C = \begin{cases} 
-1 & \text{if bucket cache line alignment is enabled} \\
1 & \text{if bucket cache line alignment is disabled}
\end{cases}$$

The performance in execution cycles can be regressed on $x_A$, $x_B$ and $x_C$ using a nonlinear regression model as described previously in Section 3.9.1. However, we again opt for the sign table matrix method. The response variable $y$ is the cycles per output tuple metric for the join execution performance and values of $y$ for different experiments are given in Table 3.17. The $8 \times 8$ sign table matrix is shown in Table 3.18.

The computation of the matrix method is described earlier in Section 3.9.1. The computed mean values for each of the factors and their interaction represent the mean impact of each factor in the overall performance in Table 3.18.

**Allocation of Variation.** Now, we proceed with the importance of each factor. The importance of a factor is determined by the proportion of the variation that a factor causes in the total variation. Therefore, we first compute the total variation in the output cycles per tuple metric, in other words **Sum of Squares Total (SST)**:

$$\text{Total variation of } y = SST = \sum_{i=1}^{2^3} (y_i - \bar{y})^2 = 2^3 q_A^2 + 2^3 q_B^2 + 2^3 q_C^2 + 2^3 q_{AB}^2 + 2^3 q_{AC}^2 + 2^3 q_{BC}^2 + 2^3 q_{ABC}^2$$
3.9. Factorial Analysis of Performance

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>AB</th>
<th>AC</th>
<th>BC</th>
<th>ABC</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>17.24</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>15.85</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>29.68</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>24.92</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>29.76</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>21.37</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>26.00</td>
</tr>
<tr>
<td></td>
<td>185.76</td>
<td>-9.48</td>
<td>34.96</td>
<td>10.38</td>
<td>-7.56</td>
<td>2.82</td>
<td>-8.06</td>
<td>-0.82</td>
</tr>
<tr>
<td></td>
<td>23.22</td>
<td>-1.185</td>
<td>4.37</td>
<td>1.2975</td>
<td>-0.945</td>
<td>0.3525</td>
<td>-1.0075</td>
<td>-0.1025</td>
</tr>
</tbody>
</table>

Table 3.18: Sign table calculation of the effects of optimizations in no partitioning join performance on Intel Nehalem.

<table>
<thead>
<tr>
<th></th>
<th>SSA</th>
<th>SSB</th>
<th>SSC</th>
<th>SSAB</th>
<th>SSAC</th>
<th>SSBC</th>
<th>SSABC</th>
<th>SST</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.2338</td>
<td>152.7752</td>
<td>13.46805</td>
<td>7.1442</td>
<td>0.99405</td>
<td>8.12045</td>
<td>0.08405</td>
<td>193.8198</td>
<td></td>
</tr>
<tr>
<td>% explained</td>
<td>5.8%</td>
<td>78.8%</td>
<td>7.0%</td>
<td>3.7%</td>
<td>0.5%</td>
<td>4.2%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 3.19: Allocation of variation and percentage explained by each factor in no partitioning join performance on Intel Nehalem.

Finally, the fraction of variation explained by each factor can be found as follows:

\[
\text{Fraction of variation explained by factor } F = \frac{SS_F}{SST}
\]

For our analysis, sum of squares total and variation explained by each factor is computed and presented in Table 3.19. The variation explained by the software prefetching optimization turns out to be the most significant factor on the Intel Nehalem machine (78.8%). It mainly stems from the fact that software prefetching in this architecture is quite successful in hiding cache misses. The variation explained by the cache line alignment is found to be the second most significant factor, though not as much significant as software prefetching (7.0%). This is no surprise as Intel Nehalem microarchitecture is known to handle split cache line accesses as much efficient as aligned cache line accesses. The third most significant factor is found to be the virtual memory page size (5.8%). Overall, software prefetching optimization is the single most dominant factor in determining no partitioning join performance on the Intel Nehalem.
Chapter 3. Hash Joins

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>AB</th>
<th>AC</th>
<th>BC</th>
<th>ABC</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>20.46</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>19.74</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>19.85</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>19.41</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>28.29</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>28.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>35.16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>34.68</td>
<td></td>
</tr>
</tbody>
</table>

205.92 -1.6 12.28 47 -0.24 0.72 14.16 -0.8 Total
25.74 -0.2 1.535 5.875 -0.03 0.09 1.77 -0.1 Total/8

Table 3.20: Sign table calculation of the effects of optimizations in no partitioning join performance on AMD Bulldozer.

<table>
<thead>
<tr>
<th>SSA</th>
<th>SSB</th>
<th>SSC</th>
<th>SSAB</th>
<th>SSAC</th>
<th>SSBC</th>
<th>SSABC</th>
<th>SST</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.32</td>
<td>18.8498</td>
<td>276.125</td>
<td>0.0072</td>
<td>0.0648</td>
<td>25.0632</td>
<td>0.08</td>
<td>320.51</td>
</tr>
</tbody>
</table>

% explained 0.1% 5.9% 86.2% 0.0% 0.0% 7.8% 0.0% 100%

Table 3.21: Allocation of variation and percentage explained by each factor in no partitioning join performance on AMD Bulldozer.

Analysis of Optimizations on AMD Bulldozer

Similar to the analysis in the Intel Nehalem, for the factorial analysis of optimizations on no partitioning performance in the AMD Bulldozer machine, we again setup a $2^4$ experimental design with the same factors. The $8 \times 8$ sign table matrix for the performance numbers in AMD Bulldozer is shown in Table 3.20. The computed mean values for each of the factors and their interaction represent the mean impact of each in the overall performance in Table 3.20.

Allocation of Variation. For the factorial analysis in the AMD Bulldozer, sum of squares total and variation explained by each factor is computed and presented in Table 3.21. The table also summarizes the importance of each factor based on the proportion of the variation caused.

In the AMD Bulldozer, the variation explained by the cache line alignment optimization of hash table buckets turns out to be the most significant factor (86.2%). The reason is AMD Bulldozer’s inability to handle unaligned accesses to cache lines. Unfortunately, this type of accesses on this architecture needs to be split into multiple requests to the cache and performance degrades significantly.
The variation explained by the combination of software prefetching and cache line alignment is found to be the second most significant factor (7.8%). As accesses are cache aligned, this architecture can benefit from software prefetching in a better way. For unaligned accesses, even prefetching needs to fetch multiple number of cache lines and do not improve performance by itself alone. The third most significant factor is found to be the software prefetching optimization (5.9%). It is also surprising to see that the impact of virtual memory page size has less of an impact on this architecture. Overall, the analysis shows that the particular property of the Bulldozer architecture strictly requires aligned accesses to memory locations for achieving better performance. Otherwise performance significantly deteriorates.

3.10 Related Work

Following the insights on the importance of memory and caching effects on modern computing hardware by Manegold et al. [BMK99] and Ailamaki et al. [ADHW99], new algorithm variants have emerged to run classical database operators efficiently on modern hardware.

One of the design techniques to achieve this goal is the use of partitioning, which we discuss extensively also in this work. Besides a use for in-memory joins, partitioning is also relevant for aggregation, as recently investigated by Ye et al. [YRV11]. While the aggregation problem differs from join computation in many ways, the observations made by Ye et al. about different hardware architectures are consistent with ours.

While we mainly looked at local caching and memory latency effects, Teubner et al. [TM11] earlier demonstrated how the topology of modern NUMA systems may add additional complexity to the join problem. Handshake join is an evaluation strategy on top of existing join algorithms to make those algorithms topology-aware.

With a similar motivation, Albutiu et al. [AKN12] proposed to use sort-merge algorithms to compute joins, leading to a hardware-friendly sequential memory access pattern. It remains unclear, however, whether the switch to a parallel merge-join is enough to adequately account for the topology of modern NUMA systems.

Similar in spirit to the no partitioning join is the recent GPU-based join implementation proposed by Kaldewey et al. [KLMV12]. Like in no partitioning, the idea is to leverage hardware SMT mechanisms to hide memory access latencies. In GPUs, this idea is pushed to an extreme, with many threads/warps sharing one physical GPU core.

Cache-conscious algorithms have been in the focus of database research in the
Chapter 3. Hash Joins

past years [SKN94, RR99, MBNK04, RR00, ZBNH05, BZN05, CMR09]. In their seminal work, Shatdal et al. [SKN94] show that traditional query processing algorithms can significantly benefit from a cache-conscious redesign by fully utilizing the processor caches. Following on the same trend, Rao et al. [RR99, RR00] apply cache-conscious design techniques to database indexing and propose new indexing data structures that perform better than existing ones by paying attention to reference locality and cache behavior. Similarly, Manegold et al. [MBNK04] propose cache-conscious techniques for the materialization operator which is usually applied after joins on column-store databases. Finally, cache-conscious design also requires a shift from the one tuple at a time iterator model of query processing to take advantage of processor caches in a great extent [ZBNH05, BZN05, CMR09].

He and Luo [HL08] addressed the “hardware-conscious or not?” question on a slightly higher system level. They evaluated a full query engine (EaseDB), which was built on cache-oblivious design principles [FLPR99]. Their conclusion was that cache-oblivious strategies can achieve comparable performance, but only when those algorithms are very carefully and sophisticatedly designed.

As the hardware trends provide more and more transistors, utilizing those on special function units implemented in hardware turns out to be a natural next step. Accordingly, hardware acceleration for many of the database tasks have been an interesting topic recently. Lisa et al. [WBKR13] explored data partitioning in the hardware with a special focus on energy efficiency. As we have also shown in this work, data partitioning is an important and frequent operation in data processing. Therefore, a hardware acceleration support for partitioning with energy efficiency promises is likely to be of great benefit for databases. With a similar goal in mind, Kocberber et al. [KGP+13] introduces an on-chip accelerator for database hash index lookups that achieves higher performance and power efficiency in operators such as hash joins.

3.11 Conclusion

In this chapter of the dissertation we revisit the existing results in the literature regarding the design of main-memory, hash join algorithms on multi-core architectures. Through an extensive experimental evaluation, we trace back the observed differences in performance of existing algorithms to a wide range of causes: workload, caching effects, and multi-threading. In doing so, we provide optimizations to all existing algorithms that significantly increase their performance. The results in the latest hardware available to us indicate that hardware-conscious algorithms maintain an edge over hardware-oblivious in most systems and configurations. Moreover, with the novel ideas introduced in the dissertation, hardware-conscious algorithms can be made significantly faster than what has been published so far.
and more robust to a wider set of parameters. These algorithms can also be easily tuned to the underlying hardware, as shown in the dissertation, significantly reducing the argument that they are more difficult to port than their hardware-oblivious counterparts. However, there are specific cases where features of modern hardware such aggressive on-chip multi-threading make hardware-oblivious algorithms competitive, thereby establishing an interesting architectural option for future systems.
4 Sort-Merge Joins

4.1 Introduction

Modern processor architectures introduce many possibilities as well as challenges for the implementation of parallel data operators. Advanced instruction sets, vector operations, multi-core architectures, and NUMA constraints create a very rich design space where the effects of given design decisions on the performance of data operators are not always easy to determine. There is a need for developing a better understanding of the performance of parallel data operators on new hardware.

In this chapter, we explore the design space of main-memory sort-merge joins on modern multi-core processors. We propose efficient sort-merge join algorithms utilizing modern hardware features such as large SIMD vectorization units (e.g., 256-bit AVX). Furthermore, we tailor sort-based join algorithms to the underlying hardware in terms of memory bandwidth awareness and hardware peculiarities such as NUMA. The chapter describes parallel execution of sort-merge joins on modern multi-core machines in a bottom up manner starting from register level, data parallel SIMD sorting. The design choices for different phases of a sort-merge join algorithm, namely parallel sorting and merging, are described with accompanying experiments. Through an experimental evaluation of each design choice, we choose the best options for implementing sort-merge joins on modern multi-core processors. After discussing parallel sorting and merging as the building blocks, the chapter goes on to discuss overall sort-merge join algorithm variants that are both tailored for the memory architectures and SIMD on multi-core CPUs.
Through the experimental evaluation of proposed sort-merge join algorithms, this chapter demonstrates the performance of multi-core sort-merge joins and their scalability on a recent, large multi-core server. Furthermore, we provide several insights on the implementation of data operators on modern processors. The results present the fastest algorithms available to date for sort-merge join algorithms—2-3 times faster than available results in the literature.

Furthermore, the chapter brings the factors and number of issues that affect the implementation and performance of sort-merge join algorithms to light. These include:

Degree of Parallelism. Existing work has studied algorithms using a small degree of parallelism. As the number of available hardware contexts increases, contention in large merge trees for sorting also increases. This contention is not visible in the four-core configurations used in earlier studies but it becomes a dominant factor in larger systems.

Cache Contention. Cache-conscious approaches make a significant difference in choosing the sort-merge join algorithm variants. For efficient merging, Kim et al. [KSC+09] assume that entire merge trees can fit into a single last-level cache. As the degree of parallelism, data sizes, and merge fan-ins increase, this assumption may no longer hold, which calls for a closer look at the implementation of multi-way merge trees.

SIMD Performance. There are many ways to exploit SIMD in all algorithms. Our results show, however, that hardware peculiarities play a big role, and it turns out that the width of the SIMD registers is not the only relevant factor. The complex hardware logic and signal propagation delays inherent to more complex SIMD designs may result in latencies larger than one cycle, limiting the advantages of SIMD.

4.2 Background and Related Work

4.2.1 Canonical Sort-Merge Join Algorithm

The classical sort-merge join algorithm is depicted in Figure 4.1. The algorithm mainly consists of two separate steps. The first step is sorting of both input relations according to the designated join key. The second step is the merge-join of the sorted relations. If it is known that input relations are already sorted, then the first step can be skipped; otherwise input relations need to be sorted using the “sort” operator that commonly exists in database engines. Afterwards, the merge-join operation proceeds as follows: Both relations are scanned sequentially and a head pointer keeps track of the current position in each relation. The join condition is evaluated on the head elements of both relations and, if successful,
an output tuple is generated with the appropriate columns. The cursor on the relation with the smaller value is advanced and evaluation of the join condition is repeated, as described above.

The algorithmic complexity of the sort-merge join mainly emanates from the sorting operator. The complexity of sorting is $O(N \log N)$ in the size of the input relations. The merge-join operation has a linear complexity in the order of the input sizes and hence can be ignored with regards to sorting complexity.

### 4.2.2 Hardware-Assisted Sorting

Recent work on sorting has explored the use of SIMD data parallelism [CNL+08, GBY07, I+07, SKC+10a]. Inoue et al. [I+07] introduced AA-Sort which utilized both SIMD and thread-level parallelism. AA-Sort eliminates unaligned loads for maximum utilization of SIMD where unaligned accesses cause performance bottlenecks in architectures such as PowerPC and Cell processors. Gedik et al. [GBY07] also investigated parallel sorting for the Cell processor and implemented an efficient sorting algorithm with bitonic sorting and merging using SIMD parallelism. Chhugani et al. [CNL+08] provided a multi-core SIMD sorting implementation over commodity x86 processors. Their algorithm, based on merge sort, extensively used bitonic sort and merge networks for SIMD parallelism, following the ideas introduced by Inoue et al. [I+07] for in-register sorting. However, the machine they used had only four cores and further scalability was based on projections. Satish et al. [SKC+10a] have analyzed comparison and non-comparison-based sorting algorithms on modern CPU/GPU architectures. Their study provided some of the fastest sorting implementations and found that non-comparison-based scalar sort-
ing such as radix sort is faster with smaller keys. Moreover, they showed that SIMD-based merge sort is more competitive with larger keys and will be even more favorable with the future hardware trends such as larger SIMD width. A more recent technical report by Satish et al. [SKC+10b] extend the comparison of the earlier work to the Intel Many Integrated Core (MIC) architecture and report a performance improvement of 2.2X and 1.7X over the earlier best sort performance on the Intel Core i7 CPU and Nvidia GTX 280 GPU respectively. They also report further optimizations for radix sort with an improvement of 1.6X over the previous results. Wassenberg et al. [WS11] also demonstrate optimizations to the radix sort algorithm by taking advantage of virtual memory and write-combining features in modern processor microarchitectures. They report that their algorithm is faster than [SKC+10a] by a factor of 1.64. However, the optimizations look similar to the ones mentioned in the technical report by Satish et al. [SKC+10b] and it is likely that both of the recent approaches would perform similarly. More recently, Kim et al. [KPS+12] implemented distributed in-memory sorting over a cluster of multi-core machines. While their main solution focuses on overlapping computation and inter-node communication, their approach also makes extensive use of parallel SIMD sorting on each of the machines.

4.2.3 Sort-Merge Joins

Sort-merge join has long been recognized as the most prominent approach for evaluating relational join queries [BE77, ME92]. Sort-merge joins are traditionally known to be applicable to more join types (e.g., band join, spatial join, temporal join and similarity join) than equi-joins. While the classic sort-merge join is a blocking operator, there has been proposals in the literature to make it produce results as early as possible [DSTW02]. In terms of parallel execution, Schneider et al. [SD89] evaluate a parallel sort-merge join algorithm along with different hash-based algorithms in a shared-nothing multiprocessor environment.

In one of the earlier recent works, Kim et al. [KSC+09] provided a sorting-based join exploiting both SIMD and multiple processor cores. The hardware they used provides 128-bit SIMD vectorization and the speedup they gained using SIMD remained below a factor of two due to narrower SIMD register width. However, through a simulation and analytical estimation they concluded that wider SIMD registers will soon make sort-merge scale much better.

Albutiu et al. [AKN12] presented a “massively parallel sort-merge join” (MPSM) tailored for modern multi-core and multi-socket NUMA processors. Despite not using SIMD when sorting the data, MPSM demonstrated good performance and linear scalability while performing better than some of the commercial database systems. NUMA-aware access patterns was observed to be the key element of the
4.3 Parallelizing Sort with SIMD

MPSM algorithm. In addition, MPSM avoided full sorting of the outer relation and saved significant costs when the outer relation is of large size.

Skew might pose performance degradation risks for sort-merge joins. Li et al. [LGS02] address the negative impact of skew in sort-merge join and proposes several solutions to deal with data skew. However they do not consider a parallel execution environment. Accordingly some of their algorithms are not directly applicable for mitigating risks of data skew for parallel sort-merge joins. In our work, we develop fine-grained skew handling mechanisms for parallel sort-merge joins following the ideas initially proposed by Albutiu et al. [AKN12].

4.2.4 The Role of NUMA

For better performance in NUMA systems, algorithms must be hardware conscious by taking the increasingly more complex NUMA topologies into consideration [AKN12, LPM+13]. Li et al. [LPM+13], for instance, showed that a hardware-conscious “ring-based” data shuffling approach across NUMA regions achieves a much better interconnect bandwidth and improves the performance of sort-merge join algorithm of Albutiu et al. [AKN12]. Therefore, we followed a similar approach and made our algorithms NUMA aware.

4.3 Parallelizing Sort with SIMD

The dominant cost factor in sort-merge joins is sorting the input relations. We thus now discuss strategies to implement sorting in a hardware-conscious manner. Typically, sort-merge joins use merge sort—a tribute to the latency/bandwidth gap in modern system architectures. Both building blocks of merge sort, (a) initial run generation and (b) the merging of pre-sorted runs, benefit from SIMD.

4.3.1 Run Generation

For initial run generation, many chunks with a small number of tuples need to be sorted. This favors sorting algorithms that can process multiple chunks in parallel over ones that have a good asymptotic complexity with respect to the tuple count. Sorting networks provide these characteristics and fit well with the SIMD execution model of modern CPUs [CNL+08, GBY07, MTA12].

Sorting Networks

Figure 4.2 on the left illustrates, in the notation of Knuth [Knu98, Section 5.3.4], a sorting network for four input items. A set of four items \( \langle 9, 5, 3, 6 \rangle \) enters the
network on the left and travels toward the right through a series of comparators. Every comparator emits the smaller of its two input values at the top, the larger on the bottom. After traversing the five comparators, the data set is sorted \((3, 5, 6, 9)\).

The beauty of sorting networks is that comparators can be implemented with help of min/max operators only. Specifically, the five comparators in Figure 4.2 compile into a sequence of ten min/max operations as illustrated here on the right (input variables \(a, \ldots, d\) and output variables \(w, \ldots, z\)). Limited data dependencies and the absence of branching instructions make such code run very efficiently on modern hardware.

Sorting networks are also appealing because they can be accelerated through SIMD instructions. When all variables in the code on the right are instantiated with SIMD vectors of \(\kappa\) items and all min/max calls are replaced by SIMD calls, \(\kappa\) sets of items can be sorted in approximately the same time that a single set would require in scalar mode (suggesting a \(\kappa\)-fold speedup through SIMD).

**Speedup Through SIMD**

However, the strategy illustrated above will sort input items across SIMD registers. That is, for each vector position \(i\), the sequence \(w_i, x_i, y_i, z_i\) will be sorted, but not the sequence of items within one vector (\(i.e., w_i, \ldots, w_\kappa\) is in undefined order). Only full SIMD vectors can be read or written to memory consecutively. Before writing back initial runs to main-memory, SIMD register contents must thus be transposed, so items within each vector become sorted (\(i.e., w_2\) must be swapped with \(x_1, w_3\) with \(y_1\), etc.).

Transposition can be achieved through SIMD shuffle instructions that can be used to move individual values within and across SIMD registers. A common configuration in the context of join processing is to generate runs of four items
with $\kappa = 4$. Eight shuffle instructions are then needed to transpose registers. That is, generating four runs of four items each requires 10 \text{min}/\text{max} instructions, 8 shuffles, 4 loads, and 4 stores. Shuffle operations significantly reduce the effective SIMD speedup for run generation from optimal $\kappa = 4$ to about 2.7.

### 4.3.2 Merging Sorted Runs

#### Scalar Merge

Two sorted runs can be combined into a larger sorted run by scanning both input runs concurrently and comparing the head items. The branching in a straightforward implementation has a very high misprediction probability, and causes expensive pipeline flushes in modern processors. Software predication [BZN05]—or hardware predication as, e.g., in Intel's Itanium architecture—can remedy the problem. Even more effective is the use of conditional move instructions \texttt{CMOVNC}, as suggested by Kim et al. [KSC+09].

#### Bitonic Merge Networks

Although sequential in nature, merging also benefits from SIMD acceleration. The basic idea comes from Inoue et al. [I+07] and has been used for sorting [CNL+08] and joins [KSC+09].

*Merger Networks. Looking back to the idea of sorting networks, larger networks can be built with help of merging networks that combine two pre-sorted inputs into an overall sorted output. Figure 4.3 shows a network that combines two input lists of size four.*

*SIMD Execution. The network in Figure 4.3 is a sequence of three stages, each consisting of four comparator elements $\ddagger$. Each stage can thus be implemented
Algorithm 2: Merging larger lists with help of bitonic merge kernel

\[ \text{bitonic\_merge}_k() \ (k = 4). \]

\begin{verbatim}
1 a ← fetch4 (in_1); b ← fetch4 (in_2);
2 repeat
3   ⟨a, b⟩ ← bitonic\_merge_4(a, b);
4   emit a to output;
5   if head (in_1) < head (in_2) then
6     a ← fetch4 (in_1);
7   else
8     a ← fetch4 (in_2);
9 until eof (in_1) or eof (in_2):
10  ⟨a, b⟩ ← bitonic\_merge_4(a, b);
11  emit4 (a); emit4 (b);
12  if eof (in_1) then
13     emit rest of in_2 to output;
14  else
15     emit rest of in_1 to output;
\end{verbatim}

using one \texttt{max} and one \texttt{min} SIMD instruction (assuming \( \kappa = 4 \)). Shuffle instructions in-between stages bring vector elements into their proper positions (for instance, if \( a \) and \( b \) are provided as one SIMD register each, \( b \) must be reversed using shuffles to prepare for the first \texttt{min}/\texttt{max} instruction pair).

On current Intel hardware, for \( \kappa = 4 \), implementing a bitonic merge network for \( 2 \times 4 \) input items requires 6 SIMD \texttt{min}/\texttt{max} instructions and 7–10 shuffles. The exact number of shuffles depends on the bit width of the input items and the instruction set offered by the hardware (SSE, AVX, AVX2).

Merging Larger Lists using Bitonic Merge

For larger input sizes, merge networks scale poorly [MTA12]: sorting networks for \( N \) input items require \( O(N \log^2 N) \) comparators—clearly inferior to alternative algorithms. But small merge networks can be used as a \textit{kernel} within a merging algorithm for larger lists [I+07]. The resulting merging algorithm (Algorithm 2) uses a working set of \( 2 \times k \) data items (variables \( a \) and \( b \), both implemented as SIMD registers). In each iteration of the algorithm’s loop body, that working set is sorted (using the merge kernel \texttt{bitonic\_merge}_k() and knowing that \( a \) and \( b \) themselves are sorted already) and the smaller \( k \) items are emitted to the merge result.
The emitted SIMD vector is then replaced by fresh data from the input. As in the classical scalar merge algorithm, the two head elements of the input runs are used to decide which new data to load (line 5 in Algorithm 2). Unlike in the classical algorithm, however, the decision is used to load an entire vector into the working set. The rationale is that the resulting working set still contains at least $k$ items that are smaller than the larger of the two head items, and only $k$ items will be emitted in the next loop iteration.

In terms of performance, the separation between control flow and merge kernel operations in Algorithm 2 fits well with the execution model of modern CPUs. In particular, no values have to be moved between the scalar and vector execution units of the CPU (a costly operation in many architectures). Branch mispredictions will still occur, but their effect will now be amortized over $k$ input elements. Also, optimizations such as predication or conditional moves can be applied in the same way as in scalar merge implementation.

The vector size used for merging is a configuration parameter, typically $k = 2^p \times \kappa$ (where $\kappa$ is the hardware SIMD width). For the hardware that we used in our experimental study, we found $k = 8$ to be a sweet spot that amortizes branching cost well, while not suffering too much from the complexity of large merge networks. Each loop iteration requires 36 assembly instructions to produce 8 output items.

### 4.3.3 Implementation Details

#### In-Register Sorting using AVX

The implementation of the small sorted run generation as described in Section 4.3.1 is presented in Figure 4.4. The code implements a series of comparisons followed by transposition of the SIMD registers using shuffle instructions.

#### Bitonic Merge Networks and AVX

An interesting property of merging is that it has a highly data parallel evaluation characteristic. By comparing and merging multiple elements at once, it can profit from SIMD instructions. We implemented each single step of the merging of multiple elements using a bitonic merge network as described in Section 4.3.2. The input to the example bitonic merge network in Figure 4.3 is two registers with 4 items each. However, as an initial step, one of the registers has to be reversed. Afterwards, a pair of min/max instructions are applied to the registers and their results are propagated to the next level by using shuffle instructions. Assuming the availability of a generic any-to-any single instruction shuffle operation, $k$-wide bitonic merge network requires $2 \cdot \log_2 k$ min/max instructions and $1 + 2 \cdot \log_2 k$
// 1st level of comparisons
__m256d H1 = __mm256_min_pd(A, B);
__m256d L1 = __mm256_max_pd(A, B);
__m256d H2 = __mm256_min_pd(C, D);
__m256d L2 = __mm256_max_pd(C, D);

// 2nd level of comparisons
__m256d A' = __mm256_max_pd(H1, H2);
__m256d M1 = __mm256_min_pd(H1, H2);
__m256d M2 = __mm256_max_pd(L1, L2);
__m256d D' = __mm256_max_pd(L1, L2);

// 3rd level of comparisons
__m256d B' = __mm256_max_pd(M1, M2);
__m256d C' = __mm256_min_pd(M1, M2);

// shuffle A'[2] and B'[1] - shuffle A'[4] and B'[3]
__m256d X = __mm256_unpacklo_pd(A', B');
__m256d Y = __mm256_unpackhi_pd(A', B');

// shuffle C'[2] and D'[1] - shuffle C'[4] and C'[3]
__m256d Z = __mm256_unpacklo_pd(C', D');
__m256d W = __mm256_unpackhi_pd(C', D');

// shuffle (A'[3],B'[3]) and (C'[1],D'[1]) pairs
__m256d A'' = __mm256_permute2f128_pd(X, Z, 0x20);
__m256d C'' = __mm256_permute2f128_pd(X, Z, 0x31);

// shuffle (A'[4],B'[4]) and (C'[2],D'[2]) pairs
__m256d B'' = __mm256_permute2f128_pd(Y, W, 0x20);
__m256d D'' = __mm256_permute2f128_pd(Y, W, 0x31);

// final results are in A'', B'', C'', Y''

Figure 4.4: In-Register sorting with four AVX registers.

shuffle instructions and a total of log₂k levels.

Ideally, we want to merge 8-byte items which are 4-byte key and 4-byte payload pairs. Thus 4 items can fit into a 256-bit AVX register. Therefore, applying the formulas to the shown bitonic merge network shown in Figure 4.3, we should normally need 6 min/max and 7 shuffle instructions to merge 2 AVX registers. However, in case of AVX instruction set, this is not possible due certain limitations:

First, any-to-any generic shuffles are not available in the AVX instruction set. Each 256-bit AVX register conceptually consists of two 128-bit lanes and it is either possible to shuffle two 128-bit lanes or any-to-any shuffles within each 128-bit lane. Due to this limitation, in order to reverse the order of four 64-bit values in a 256-bit AVX register, two instructions are needed as shown in Figure 4.5. Additionally, as AVX does not allow both shuffle inputs to come from the same 128-bit lane, the final shuffle after level 3 also cannot be implemented using a single instruction. In
4.3. Parallelizing Sort with SIMD

```c
1 // reverse values within each 128-bit lane
2 REG = _mm256_permute_pd(REG, 0x5);
3 // now shuffle 128-bit lanes
4 REG = _mm256_permute2f128_pd(REG, REG, 0x1);
5 // With single _mm256_permute4x64_pd() in AVX2
```

Figure 4.5: Reversing the order of 64-bit values in an 256-bit AVX register. In the new AVX2 instruction set, it will be possible with a single instruction.

this level both of the outputs must be shuffled to get the final ordered result and this introduces two more additional instructions (cf. lines 17-21 in Figure 4.6).

Second, another property of AVX is that operations usually have 3 operands. Hence, instructions are not destructive and preserve the initial value of input registers. As a result, the load on register use decreases, immediate operations can reuse the operands and additional register-to-register moves are not required. Overall, the bitonic merge network of size four requires a total of 16 AVX instructions (6 min/max + 10 shuffle variants) excluding the initial load and final store instructions. The implementation using AVX intrinsics is shown in Figure 4.6 for reference.

**Static Code Analysis of the Kernels**

We have analyzed our kernel codes for different size bitonic merge networks using Intel Architecture Code Analyzer [Int12b]. The tool statically analyzes a given code snippet (or kernel), and gives a first order estimate of the performance on different Intel micro-architectures. The latency analysis measures the latency of execution from the first instruction to the end of the last instruction, as if the kernel is executed once. The throughput analysis measures the throughput as if the kernel is executed in an infinite loop. Additionally, the tool provides hints on performance bottlenecks and resource conflicts. The analysis results of our kernels on Sandy Bridge micro-architecture are shown in Table 4.1. The analysis reveals that the 8-wide bitonic merge network shows a good trade-off in terms of number of instructions and latency per item. We have also validated experimentally that using 8-wide bitonic merge network achieves the best performance in our experiment machine.
REVERSE(B); // Reverse B with 2 instructions

// Level-1 comparisons
__m256d l1 = _mm256_min_pd(A, B);
__m256d h1 = _mm256_max_pd(A, B);

// Level-1 shuffles
__m256d l1p = _mm256_permute2f128_pd(l1, h1, 0x31);
__m256d h1p = _mm256_permute2f128_pd(l1, h1, 0x20);

// Level-2 comparisons
__m256d l2 = _mm256_min_pd(l1p, h1p);
__m256d h2 = _mm256_max_pd(l1p, h1p);

// Level-2 shuffles
__m256d l2p = _mm256_shuffle_pd(l2, h2, 0x0);
__m256d h2p = _mm256_shuffle_pd(l2, h2, 0xF);

// Level-3 comparisons
__m256d l3 = _mm256_min_pd(l2p, h2p);
__m256d h3 = _mm256_max_pd(l2p, h2p);

// Level-3 shuffles: vunpcklqdq/vunpckhpd + vperm2f128
__m256d l4 = _mm256_unpacklo_pd(l3, h3);
__m256d h4 = _mm256_unpackhi_pd(l3, h3);
O1 = _mm256_permute2f128_pd(l4, h4, 0x20);
O2 = _mm256_permute2f128_pd(l4, h4, 0x31);

Figure 4.6: Implementation of `bitonic_merge4()` \((k = 4)\) using AVX instruction intrinsics.

4.4 Cache Conscious Sort Joins

4.4.1 Sorting and the Memory Hierarchy

The cache hierarchies in modern hardware require separating the overall sorting into several phases to optimize cache access: (i) in-register sorting, with runs that fit into (SIMD) CPU registers; (ii) in-cache sorting, where runs can still be held in a CPU-local cache; and (iii) out-of-cache sorting, once runs exceed cache sizes.

In-Register Sorting. Phase (i) corresponds to run-generation as discussed in Section 4.3.1.

In-Cache Sorting. In Phase (ii), runs are then merged until runs can no longer be contained within CPU caches. In-cache sorting corresponds to Algorithm 2 (bitonic merging) in Section 4.3.2. It is backed up by a bitonic merge kernel such as `bitonic_merge4()`. Using bitonic merging, runs are repeatedly combined until runs have reached \(\frac{1}{2}\) cache size (since in- and output runs must fit into cache).

Out-of-Cache Sorting. Phase (iii) continues merging until the data is fully sorted. Once runs have exceeded the size of the cache, however, all memory refer-
### Algorithm 3: Overall merge sort algorithm.

```plaintext
1 \( C_1 \ldots C_m \leftarrow I; \) // Divide input into \( m \) cache-sized chunks
2 \textbf{foreach} cache-sized chunk \( C_i \) \textbf{do}
3 \( C'_i \leftarrow \text{In-Cache-Sort}(C_i); \) // Incorporates In-Register Sorting in the first phase
4 \( npairs \leftarrow m/2; \)
5 \textbf{for} \( j \leftarrow 1 \) to \( \lceil \log_2(m) \rceil \) \textbf{do}
6 \textbf{for} \( k \leftarrow 1 \) to \( npairs \) \textbf{do}
7 \( C'_k \leftarrow \text{Merge}(C'_{2k}, C'_{2k+1}); \) // Out-of-Cache Sorting: Merge 2-pairs iteratively (using bitonic merging)
8 \( npairs \leftarrow npairs/2; \)
```

The overall sorting algorithm tailored to the memory hierarchy and consisting of the steps described above is shown in Algorithm 3.

### 4.4.2 Balancing Computation and Bandwidth

Accesses to off-chip memory make out-of-cache sorting sensitive to the characteristics of the memory interface. As mentioned in Section 4.3.2, an 8-wide bitonic merge implementation requires 36 assembly instructions per eight tuples being merged—or per 64 bytes of input data.

We analyzed the 36-instructions loop of our code using the Intel Architecture Code Analyzer [Int12b]. The tool considers the super-scalar instruction pipeline of modern Intel processors and infers the expected execution time for a given instruction sequence. For our code, the tool reported a total of 29 CPU cycles per 64 bytes of input data. The tool does not consider potential pipeline stalls due to memory accesses (bandwidth and/or latency).

With a clock frequency of 2.4 GHz, this corresponds to a memory bandwidth of \( 2 \times 5.3 \text{GB/s} \) (read+write) or \( 10.6 \text{GB/s} \) for a single merging thread. This is more

---


table

<table>
<thead>
<tr>
<th></th>
<th># Instr.</th>
<th>Latency</th>
<th>Lat./item</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-wide bitonic merge</td>
<td>16</td>
<td>29 cyc.</td>
<td>7.25 cyc.</td>
<td>10 cyc.</td>
</tr>
<tr>
<td>8-wide bitonic merge</td>
<td>36</td>
<td>36 cyc.</td>
<td>4.5 cyc.</td>
<td>29 cyc.</td>
</tr>
<tr>
<td>16-wide bitonic merge</td>
<td>80</td>
<td>59 cyc.</td>
<td>3.6875 cyc.</td>
<td>51.15 cyc.</td>
</tr>
<tr>
<td>4×4 In-register sort</td>
<td>18 + 4 loads</td>
<td>27 cyc.</td>
<td>6.75 cyc.</td>
<td>10.05 cyc.</td>
</tr>
</tbody>
</table>

Table 4.1: Analysis of our kernels using Intel Architecture Code Analyzer on Sandy Bridge architecture.
than existing interfaces support, considering also that typical CPUs contain eight or more cores per chip. Out-of-cache merging is thus severely bound by the memory bandwidth.

Memory bandwidth demand can be reduced by merging more than two runs at once. *Multi-way merging* saves round-trips to memory and thus precious memory bandwidth.

To still benefit from CPU-efficient, SIMD-optimized bitonic merging, we implement multi-way merging using multiple two-way merge units, as illustrated in Figure 4.7. Two-way merge units are connected using FIFO queues; FIFO queues are sized such that all queues together still fit into the CPU cache. Thus, external memory bandwidth is needed only at the front of the multi-way merging tree.

As indicated in Figure 4.7, a complete multi-way merging is realized by a single operating system thread. This thread will treat two-way merges as individual tasks and switch between tasks whenever a task is blocked by a FIFO over- or underrun.

**Compute vs. Bandwidth**

Task switching will cause new CPU overhead that is not necessary in an implementation that merges two runs from memory until completion. This overhead will increase when the FIFO queues between merging tasks are small, because more task switches are needed then to perform a full multi-way merge. Since we size FIFO buffers such that all buffers in one thread fill a CPU cache, CPU overhead increases together with the merge fan-in.

This offers us a handle to balance bandwidth and compute. Merging only two runs is bound by memory bandwidth, with plenty of stalled CPU cycles that could be spent on additional CPU instructions. As we increase merge fan-in, memory pressure becomes reduced until the system becomes CPU-bound. At that point, larger fan-in will degrade performance again because of increasing CPU load.
4.4. Cache Conscious Sort Joins

Impact of NUMA

In practice, at least some merging passes will inevitably cross NUMA boundaries (if not, NUMA crossing has to be performed in the later join phase, which in this regard behaves like a two-way merge). As pointed out by Li et al. [LPM+13]—and confirmed by our measurements—multi-socket systems show an increasing asymmetry, where the NUMA interconnect bandwidth stays further and further behind the aggregate memory bandwidth that the individual memory controllers could provide. With multi-way merging, we can combat this problem in an effective and scalable way. In the experimental part of this work we study how merging can be tuned to avoid memory bottlenecks even across NUMA boundaries.

4.4.3 Implementation of Multi-way Merge

Merge tree and nodes

Each node in the merge tree is implemented as a circular buffer. Each circular buffer must be sized accordingly to ensure that overall working set always remains in the lowest level (i.e., L3) cache. At the leaf level of the merge tree, a pair of runs are merged until their output can fill the circular buffer, thereby overlapping the data read from sorted runs and their first merging. The internal buffers are in turn processed similarly by consuming from their children nodes, and filling in their own buffers. The processing propagates until the root node in a similar fashion. In the root node, data is read from two children and merged directly to an output buffer.

Efficient Access to the Circular Buffers

Circular buffers are ideal for reusing the buffer space of consumed input and for keeping the buffers always in-cache resident. However, each access to an element requires a modulo (%) calculation for its index in the buffer. Long latency instructions such as modulo (i.e., in fact a division) introduces stalls in the execution pipeline of CPUs. For instance in our un-optimized code, modulo operation generates a division instruction (div) and according to Fog et al. [Fog12], it has $\approx 20-28$ cycles latency on Sandy Bridge micro-architecture. In order to remove the modulo operation-based access, we sized our circular buffers as a power of 2 and implemented the modulo operator with a bitwise-and operation (i.e., i & (size-1)).

Decomposing and Parallelizing the Merge

So far we have discussed the scalar merging of circular buffers. However, the techniques for data parallel merging with bitonic merge networks (cf. Section 4.3.2)
Figure 4.8: Efficient, parallel circular buffer merging by decomposition.

are still applicable to merging circular buffers in the multi-way merge. In order to
reuse our efficient data parallel merging routines, we decompose the merge of two
circular buffers as follows. Assume that the two circular buffers we are about to
merge are A and B as shown in Figure 4.8. The next item to read from a circular
buffer is given by the head position and the next position to append an item is
given by the tail position. In certain times, head can go ahead of tail which is the
case for two circular buffers in the figure.

The idea of decomposed merging is basically finding the largest extent that
two circular buffers can be merged linearly without requiring a modulo operator.
For instance in Figure 4.8, $A_1$ and $B_1 + B_2$ can be merged without requiring a
modulo calculation for indices and until one of them reaches to the end. The first
merge ends after one of the head positions reach the end of the buffer, thus the
new head for that buffer becomes the 0th position (new head points to $A_2$ for A in
the example). For the next merge, again the maximum possible extent of linear
merge is determined and the decomposed merge continues until end of both of the
circular buffers are reached. In the example, decomposed merge completes with 3
sequence of calls to our bitonic merge kernel shown in Algorithm 2. Overall, with
this optimization our multi-way merging technique becomes quite efficient, data
parallel and does not require additional three modulo operations per each iteration
of the merge.

In general, multi-way merging solves the memory bandwidth issues of the multi-
pass merging technique. However, there are still certain limitations which has not
been considered in the related work. We discuss the performance characteristics
of multi-way merging in detail in Section 4.9.
4.5 Efficient Data Partitioning

Data merging and partitioning are duals of each other. This duality leads to an alternative where sort-merge join algorithms can also be implemented using data partitioning. This type of sort-merge join algorithms usually require partitioning to be done based on different range values of the input. The range partitioning requirement of such algorithms bear an important commonality with hash join algorithms. Essentially, hash-based radix partitioning commonly used for hash joins can also be used for implementing efficient range partitioning for sort-merge joins.

In Chapter 3, we extensively discussed hash-based joins and radix partitioning algorithms. For the sake of completeness and coherency of the discussion in this chapter, we briefly recap hash-based radix partitioning and the new optimizations for implementing it on modern processors. We discuss the use of efficient data partitioning in sort-merge joins later in the chapter along with accompanying experiments.

4.5.1 Radix Partitioning

Manegold et al. [MBK02b] refined the partitioning idea by considering as well the effects of translation look-aside buffers (TLBs) during the partitioning phase, leading to the multi-pass radix partitioning join. Conceptually, radix partitioning takes all input tuples one-by-one and writes them to their corresponding destination partition (pos[·] keeps the write location in each partition):

1. \texttt{foreach input tuple }t\texttt{ do}
2. \( k \leftarrow \text{hash}(t) \);
3. \( p[k][\text{pos}[k]] = t; \quad // \text{copy } t \text{ to target partition } k \)
4. \( \text{pos}[k]++; \)

Generally, partitions are far apart and on separate VM pages. If the fan-out of a partitioning stage is larger than the number of TLB entries in the system, copying each input tuple will cause another TLB miss. The number of TLB entries is thus treated as an upper bound to the partitioning fan-out.

4.5.2 Software-Managed Buffers

The TLB miss limitations on maximum fan-out can be reduced, when writes are buffered inside the cache first. The idea is to allocate a set of buffers, one for each output partition and each with room for up to \( N \) input tuples. Buffers are copied to final destinations only when full:
foreach input tuple t do
  k ← hash(t);
  buf[k][pos[k] mod N] = t; // copy t to buffer
  pos[k]++;
  if pos[k] mod N = 0 then
    copy buf[k] to p[k]; // copy buffer to part. k

Buffering leads to additional copy overhead. However, for sufficiently small $N$, all buffers will fit into a single memory page and into L1 cache. Thus, a single TLB entry will suffice unless a buffer becomes full and the code enters the copying routine in line 6. Beyond the TLB entry for the buffer page, an address translation is required only for every $N$th input tuple, significantly reducing the pressure on the TLB system. And as soon as TLB misses become infrequent, it is likely that the CPU can hide their latency through out-of-order execution mechanisms. This optimization follows the idea of Satish et al. [SKC+10a] who used it to reduce the TLB pressure of radix sort.

For implementing an efficient range partitioning of data we utilize such software-managed buffers and configure $N$ such that one buffer will exactly fill one cache line (64 bytes). This in turn allows for another low-level optimization. Since we are now always writing a full cache line at once to global memory, the CPU can take advantage of its write combining facilities together with non-temporal writes, thus avoiding to read the cache line before writing it back.

In practice, the advantage of software-managed buffers is two-fold: (i) for many situations, software-managed buffers offer better absolute performance, since fewer passes can usually achieve the same overall fan-out; (ii) it is possible to partition even larger data sizes in a single pass, which has not been considered previously.

4.6 Sort-Merge Join Algorithms

In this section, we describe all the sort-merge join algorithms designed, analyzed and evaluated in this chapter.

4.6.1 Sort-Merge Join Algorithm – $m$-way

The $m$-way algorithm is a highly parallel sort-merge join that relies on both data and thread parallelism and is carefully optimized toward NUMA. The general idea and the individual phases of the algorithm are presented in Figure 4.9 assuming a hypothetical machine with four NUMA regions and one thread per region.

Initially, input relations $R$ and $S$ are stored such that they are equally distributed across NUMA regions. In the first phase, each thread is assigned its
NUMA-local chunk and all the threads range-partition their local chunks in parallel using the software-managed buffers technique (Section 4.5.2). The main intuition behind partitioning in the first place is allowing threads in the subsequent phases to work independently without any synchronization. In this phase, the partitioning fan-out is usually on the order of the number of threads (64–128) and can be done efficiently using a single pass at the speed of total memory bandwidth of the machine. Then, each local partition is sorted using the AVX sorting algorithm. In this phase, different threads can sort different partitions independently, again just reading from and writing to the NUMA-local memory. Partitioning and local sorting are indicated in Figure 4.9 as \( \text{Phase 1} \).

Phase (2) in Figure 4.9 is the only phase that requires shuffling data between different NUMA regions. Therefore, it is likely to be limited by the memory/interconnect bandwidth. Hence, we employ multi-way merging here as described in Section 4.4.2. Multi-way merging successfully overlaps the data transfer and merging and brings computation and bandwidth into a balance. Outcome of this phase is a globally sorted copy of \( R \), indicated as \( R' \) in Figure 4.9.

The same steps are also applied to relation \( S \) (indicated as Phases 3/4 in

---

**Figure 4.9:** *m-way*: NUMA-aware sort-merge join with multi-way merge and SIMD.
Table 4.2: Sort-merge join algorithms studied.

<table>
<thead>
<tr>
<th>short notation</th>
<th>algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-way</td>
<td>Sort-merge join with multi-way merging</td>
</tr>
<tr>
<td>m-pass</td>
<td>Sort-merge join with multi-pass naïve merging</td>
</tr>
<tr>
<td>mpsm</td>
<td>Our impl. of massively parallel sort-merge [AKN12]</td>
</tr>
</tbody>
</table>

Figure 4.9). \( R' \) and \( S' \) are stored in the NUMA-local memory of each thread. Finally, each thread concurrently evaluates the join between NUMA-local sorted runs using a single-pass merge join (Phase 5). This join amounts to an extremely fast linear scan of both sorted runs where matching pairs constitute the join result.

### 4.6.2 Sort-Merge Join Algorithm – m-pass

The second variant for sort-merge join is \( m\text{-pass} \). The algorithm differs from \( m\text{-way} \) only in Phase 2 in Figure 4.9. Instead of applying a multi-way merge for merging NUMA-remote runs, \( m\text{-pass} \) applies successive two-way bitonic merging. The first iteration of merging of sorted runs is done as the data is transferred to the local memory. As a result of the first iteration, the number of runs reduces to \( 1/2 \) of the initial total number of runs. The rest of the merging continues in local memory, using the multi-pass merging technique (cf. Section 4.3.2) in an iterative manner.

### 4.6.3 Massively Parallel Sort-Merge Join – mpsm

The \( mpsm \) algorithm first globally range-partitions relation \( R \). This step ensures that different ranges of \( R \) are assigned to different NUMA-regions/threads. Next, each thread independently sorts its partition, resulting in a globally-sorted \( R' \). In contrast, \( S \) is sorted only partially. Each thread sorts its NUMA-local chunk of \( S \) without a prior partitioning. Therefore, during the last phase, a run of \( R \) must be merge-joined with all the NUMA-remote runs of relation \( S \). For cases where relation \( S \) is substantially larger than \( R \), avoiding the global partitioning/sorting may pay off and the overall join may become more efficient. For further details, we refer to [AKN12].

Table 4.2 summarizes the algorithms considered in the experiments and the shorthand notation used in the graphs.
4.7 Experimental Setup

4.7.1 Workloads

To facilitate comparisons with existing results, we use similar workloads to those of Kim et al. [KSC+09], Albutiu et al. [AKN12] and Balkesen et al. [BTAÖ13]. They all assume a column-oriented storage model and joins are assumed to be evaluated over narrow (8- or 16-byte) \( \langle \text{key}, \text{payload} \rangle \) tuple configurations, where \( \text{key} \) and \( \text{payload} \) are four or eight bytes. To understand the value of data parallelism using vectorized instructions, we assume \( \text{key} \) and \( \text{payload} \) are four bytes each, unless we are running scalar algorithms without using SIMD. The workloads used are listed in Table 4.3. All attributes are integers, but AVX currently only supports floating point; therefore we treat integer keys as floats when operating with AVX.\(^1\) There is a foreign key relationship from \( S \) to \( R \). That is, every tuple in \( S \) is guaranteed to find exactly one join partner in \( R \). Most experiments (unless noted otherwise) assume a uniform distribution of key values from \( R \) in \( S \).

4.7.2 System

Experiments are run on an Intel Sandy Bridge machine with a 256-bit AVX instruction set (cf. Table 4.4). It has a four-socket configuration, with each CPU socket containing 8 physical cores and 16 thread contexts by the help of the hyper-threading. Cache sizes are 32 KiB for L1, 256 KiB for L2, and 20 MiB L3 (the latter shared by the 16 threads within the socket). The cache line size of the system is 64 bytes. TLB1 contains 64/32 entries when using 4 KiB/2 MiB pages (respectively) and 512 TLB2 entries (page size 4 KiB). Total memory available is 512 GiB (DDR3 at 1600 MHz). The system runs Debian Linux 7.0, kernel version 3.4.4-U5 compiled with transparent huge page support and it uses 2 MiB VM pages for memory allocations transparently. This has been shown to improve join performance up to \( \approx 15\% \) under certain circumstances [BTAÖ12, BTAÖ13]. Therefore, we assume

\(^{1}\)AVX2 will support vectorized integer operations, thus there will be no longer semantical differences for our code.

<table>
<thead>
<tr>
<th></th>
<th>A (adapted from [AKN12])</th>
<th>B (from [KSC+09, BTAÖ13])</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of key / payload</td>
<td>4 / 4 bytes</td>
<td>4 / 4 bytes</td>
</tr>
<tr>
<td>size of ( R )</td>
<td>1600 ( \cdot ) ( 10^6 ) tuples</td>
<td>128 ( \cdot ) ( 10^6 ) tuples</td>
</tr>
<tr>
<td>size of ( S )</td>
<td>( m ) ( \cdot ) 1600 ( \cdot ) ( 10^6 ) tuples, ( m = 1, \ldots, 8 )</td>
<td>128 ( \cdot ) ( 10^6 ) tuples</td>
</tr>
<tr>
<td>total size ( R )</td>
<td>11.92 GiB</td>
<td>977 MiB</td>
</tr>
<tr>
<td>total size ( S )</td>
<td>( m ) ( \cdot ) 11.92 GiB</td>
<td>977 MiB</td>
</tr>
</tbody>
</table>

Table 4.3: Workload characteristics.
Chapter 4. Sort-Merge Joins

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Sandy Bridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets/Cores/Threads</td>
<td>Xeon E5-4640 2.40 GHz</td>
</tr>
<tr>
<td>Cache sizes (L1/L2/L3)</td>
<td>4/32/64</td>
</tr>
<tr>
<td>TLB (L1/L2)</td>
<td>32 KiB / 256 KiB / 20 MiB</td>
</tr>
<tr>
<td>Memory</td>
<td>64/512</td>
</tr>
<tr>
<td>VM Page size</td>
<td>512 GiB DDR3 1600 MHz</td>
</tr>
<tr>
<td></td>
<td>2 MiB</td>
</tr>
</tbody>
</table>

Table 4.4: Hardware platform used in our evaluation.

the availability of large page support in the system. The code is compiled with gcc 4.7.2 using -O3. Experiments using Intel’s icc compiler did not show any notable differences, qualitatively or quantitatively. For the performance counter profiles that we report, we instrumented our code with the Intel Performance Counter Monitor [Int12c].

4.8 Analysis of the Sort Phase

In a first set of experiments, we make sure that our single-thread sorting performs well compared to alternatives. In doing so, we illustrate a problem associated with sorting large amounts of data that is relevant to the discussion on joins.

4.8.1 Raw Sorting Performance

Figure 4.10 shows the performance of our SIMD sorting algorithm implemented using AVX instructions. As a baseline, we include the C++ STL sort algorithm. Overall, AVX sort runs between 2.5 and 3 times faster than the C++ sort. One expected result is also visible: whenever the size of the input increases, both algorithms suffer due to the increasing number of trips to the main-memory. AVX sort mainly suffers from the multi-pass, pair-wise merging of cache-sized sorted runs. As the input size increases, the number of runs double at each data point. Accordingly, the number of trips to the memory also increases logarithmically in the input runs. Whenever used alone, this performance degradation of single-threaded AVX sort might be acceptable since it still runs 2.5 times faster than a serial of-the-shelf algorithm.

On the other hand, the excessive memory bandwidth consumption indicates that there will be a problem when multiple threads are active and contend for the same bandwidth. One approach to deal with this problem is bandwidth efficient *multi-way merging* as also suggested by Chhugani et al. [CNL+08]. Another
4.8. Analysis of the Sort Phase

![Graph showing sort throughput vs. number of tuples in R (in $2^{20}$) for AVX sort and C++ STL sort.]

Figure 4.10: Single-threaded sorting performance where input table size varies from 8 MiB to 2 GiB.

<table>
<thead>
<tr>
<th>No. of Elem.</th>
<th>Elem. Size</th>
<th>512K</th>
<th>1M</th>
<th>4M</th>
<th>16M</th>
<th>64M</th>
<th>256M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chhugani et al. [CNL+08]</td>
<td>4-Byte</td>
<td>2.90</td>
<td>2.90</td>
<td>2.90</td>
<td>3.00</td>
<td>3.10</td>
<td>3.30</td>
</tr>
<tr>
<td>AVX sort (our code)</td>
<td>8-Byte</td>
<td>3.60</td>
<td>3.57</td>
<td>3.96</td>
<td>4.07</td>
<td>4.14</td>
<td>4.20</td>
</tr>
</tbody>
</table>

Table 4.5: Single-thread SIMD sorting performance in cycles per element per iteration metric (cepi).

approach is applying partitioning first, and therefore sorting smaller chunks afterwards. We look into this problem when discussing the merge phase.

4.8.2 Comparison with other Algorithms

As a further reference, we compare our sorting algorithm with those of Chhugani et al. [CNL+08] and Kim et al. [KSC+09]. As suggested by Chhugani et al. [CNL+08], the execution of the SIMD merge sort is extremely regular and it permits reporting performance at a finer granularity – cycles per element per iteration (cepi). The overall time of sorting for $N$ elements is therefore given by $\text{cepi} \times N \times \log_2 N$.

Table 4.5 shows the comparison of our 256-bit AVX sorting implementation vs. numbers reported by Chhugani et al. [CNL+08] over their 128-bit SSE sorting implementation. The size of tuples (i.e., elements) in our benchmark is 8-bytes, therefore an approximate 20-25% performance drawback of our algorithm is reasonable. Kim et al. [KSC+09] also report results for 8-byte tuples, however they...
can only fit 2-tuple per 128-bit SIMD register. They report 4.5 cepi for 8-byte \( \langle key, rid \rangle \) tuples for sorting of 128M tuples. Our result for the same setup is 4.17 cepi. The cepi metric is machine independent, and, thus, we are confident that our implementation behaves well in comparison to others and it is not a factor affecting the comparisons with other join algorithms.

4.9 Analysis of the Merge Phase

Increasing input sizes require multiple passes over the entire data if sub-lists are pair-wise merged. Multi-way merging remedies this problem by doing the merge in single pass. For efficiency, Kim et al. [KSC+09] assume that entire merge trees fit into a single last-level cache. However, as the degree of parallelism, data sizes, and merge fan-ins increase, this assumption no longer holds.

4.9.1 Modeling the Merge Phase

As described in Section 4.4.1, overall sorting starts with in-cache sort which generates \( \frac{1}{2} \)-L2-cache-sized sorted runs. Therefore, the fan-in \( (F) \) of the multi-way merge operation equals to the number of \( \frac{1}{2} \)-L2-sized sorted runs: Given an input size of \( N \) tuples, \( F = \frac{N \times \text{tuple-size}}{\frac{1}{2} \times \text{L2-size}} \). Secondly, efficient multi-way merging requires the merge tree to reside in the last level (L3) cache. A multi-way merge tree is essentially a binary tree and the number of total nodes \( (M) \) in a merge tree with fan-in of \( F \) equals to \( M = 2^\lceil \log_2 F \rceil + 1 \). Moreover, the shared L3 must be divided by the number of threads \( (T; \ i.e., \ T = 16 \ for \ our \ system) \) in a CPU socket. Accordingly, in order to be L3 resident, each node in the merge tree must have at most \( B \) tuples as follows \( B = \frac{\text{L3-size}}{(M \times \text{tuple-size}) \times T} \). Finally, and most importantly, \( B \) must be on the order of hundreds of tuples to ensure that multi-way merge will not degrade to single or a few item-at-a-time (scalar) merging.

4.9.2 Performance of the Merge Phase

The model we have outlined essentially suggests that increasing fan-in, and hence input sizes, will deteriorate the merge performance. To verify this finding, we performed the following experiment: We increased the number of fixed-size sorted runs from 4 to 2,048 and ran multi-way merging with this number as fan-in. To isolate the effect of concurrent threads, we ran our AVX multi-way merge on a single core using \( \frac{1}{16} \) of our 20 MiB L3 cache exclusively assigned to the merging buffer. The results are shown in Figure 4.11 denoted with \( \rightarrow \). The results confirm the model: merge performance decreases steeply with increasing fan-in. The main reason is the decreasing size of the L3 buffer per merge node in the merge tree. For
4.9. Analysis of the Merge Phase

Figure 4.11: Impact of fan-in/fan-out on multi-way merging/partitioning (1-pass and single-thread).

<table>
<thead>
<tr>
<th>Merge Fan-in</th>
<th>IPC /Core</th>
<th>Instr. /Tup.</th>
<th>Tot.-Bytes /Cyc.</th>
<th>Read /Tup.</th>
<th>Write /Tup.</th>
<th>L3-Miss /Tup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1.04</td>
<td>19.70</td>
<td>1.38</td>
<td>16.53</td>
<td>9.50</td>
<td>0.12</td>
</tr>
<tr>
<td>16</td>
<td>1.13</td>
<td>26.74</td>
<td>1.08</td>
<td>16.47</td>
<td>8.96</td>
<td>0.12</td>
</tr>
<tr>
<td>32</td>
<td>1.17</td>
<td>34.00</td>
<td>0.87</td>
<td>16.45</td>
<td>8.73</td>
<td>0.12</td>
</tr>
<tr>
<td>64</td>
<td>1.23</td>
<td>43.68</td>
<td>0.71</td>
<td>16.47</td>
<td>8.73</td>
<td>0.12</td>
</tr>
<tr>
<td>128</td>
<td>1.26</td>
<td>56.81</td>
<td>0.57</td>
<td>16.52</td>
<td>8.92</td>
<td>0.12</td>
</tr>
<tr>
<td>256</td>
<td>1.35</td>
<td>79.55</td>
<td>0.44</td>
<td>16.63</td>
<td>9.41</td>
<td>0.13</td>
</tr>
<tr>
<td>512</td>
<td>1.46</td>
<td>126.87</td>
<td>0.31</td>
<td>16.83</td>
<td>10.32</td>
<td>0.13</td>
</tr>
<tr>
<td>1024</td>
<td>1.64</td>
<td>245.51</td>
<td>0.20</td>
<td>17.32</td>
<td>12.16</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Table 4.6: Performance counter monitoring results for multi-way merge with increasing merge fan-in.
instance, with fan-in of 256, the L3-resident FIFO buffer of each merge node holds just 321 tuples and decreases to 160 tuples with fan-in of 512. At higher fan-in values, multi-way merge becomes less effective as the number of tuples processed at once drops significantly.

Another reason for performance degradation can be observed through performance counters: The increasing depth of the merge tree means a logarithmically increasing number of executed instructions per tuple. At that point, multi-way merging becomes CPU bound as shown in Table 4.6. The instruction count per tuple increases significantly with increasing fan-in along with the instructions-per-cycle (IPC). On the Intel Sandy Bridge, a maximum of 2 IPC for AVX instructions is barely possible. Consequently, the thread is far from being memory bound (cf. column “Total Bytes/Cycle”). In our microbenchmarks, the system was able to provide up to 3.71 total bytes/cycle or roughly 9 GB/sec single thread memory bandwidth measured with the STREAM [McC95] benchmark. The numbers measured through the microbenchmarks are clearly higher than the values shown in column “Total Bytes/Cycle” in Table 4.6. Other profiling metrics are also in line with the expectations: each tuple is read and written once where the write also causes another read for a total of 2 reads and 1 write of the 8-byte tuple (compare with “Read/Tup.” and “Write/Tup.”). In terms of the cache efficiency, theoretically the first tuple in the cache line should cause a miss and the rest of the 7 tuples will hit the L3 cache. Therefore, the expected \( \frac{1}{8} (0.125) \) L3 misses per tuple is also in line with the measured L3 misses.

The cache contention problem in multi-way merging raises a question against the assumption that multi-way merge can be done efficiently. This is possible only for certain data sizes/fan-ins, however increasing data sizes will require multiple passes over the data, affecting the overall performance of the join operator.

### 4.10 Optimizing the Merge Phase

The previous results show the importance of the input size on the performance of sort-merge operations. The same problem occurs for the partitioning phase of hash-based joins. Interestingly, the solution for that problem can also be used for sort-merge joins as we show in this section.

#### 4.10.1 Performance of the Partitioning Phase

Figure 4.11 shows the performance of partitioning with the same amount of total data as in the analysis for merging. This randomly generated data is partitioned with given fan-out. The results denoted with ← show that radix partitioning is sensitive to the TLB size, which is 32 in our system for 2 MiB pages. Therefore,
4.10. Optimizing the Merge Phase

Partitioning throughput significantly decreases after a fan-out of 32 due to TLB misses. However, the software-managed buffers idea described in Section 4.5.2 is much better (cf. ). The robust performance of the software-managed buffers strategy for high fan-outs is clearly a big improvement over the normal radix partitioning previous authors have considered so far.

Partitioning and merging are duals of each other. However, the cost of these two operations differ as shown in Figure 4.11. When compared with the software-managed buffering technique (i.e., optimized radix partitioning), multi-way merging is significantly slower than partitioning. This raises the question of why not using partitioning with the software-managed buffers technique for sorting. If we also use this technique for hash-based joins, previous assumptions about the number of passes over the data for the different join approaches no longer hold. With the optimized technique, a significant amount of data can be partitioned in a single pass as opposed to the two or more passes previously assumed by Kim et al. [KSC+09]. This in turn calls for a more careful comparison of join algorithms under a wider range of data sizes.

Figure 4.12: Impact of input size on different multi-threaded sorting approaches (using 64 threads).
4.10.2 Using Partitioning with Sort

The overall NUMA-aware sorting-based join algorithm, $m$-way is described in Section 4.6.1. The implementation of the first phase, local sorting, can choose either partitioning or merging:

**Partition-then-Sort** approach first range-partitions the input with efficient software-managed radix partitioning (using most-significant radix bits). Each of the partitions are individually sorted using the AVX sort routine, and concatenation of these naturally creates a sorted output.

**Sort-then-Merge** approach, instead, first creates cache-sized sorted runs using the AVX sort routine. These sorted runs are then merged to form a complete sorted output using the AVX multi-way merge routine.

In Figure 4.12, we compare the performance of the two approaches. The workload consists of 64M to 1024M 8-byte tuples. First, the Partition-then-Merge approach achieves a throughput of up to 680 million tuples per second. More importantly, it shows a stable performance with increasing input sizes, sorting 8 GB in less than 2 seconds. On the other hand, the performance of Sort-then-Merge approach drops significantly beyond table sizes of 256 M; mainly due to increasing fan-in of the multi-way merge. For instance with table size of 512 M, multi-way merge runs with a fan-in of 512. The main performance difference stems from
4.10. Optimizing the Merge Phase

Partitioning vs. merging performance. Figure 4.13 shows the throughput achieved for merging and partitioning phases in isolation. Partitioning achieves up to 4 billion tuples per second by effectively utilizing all the available bandwidth in our machine \((4 \text{ billion} \times 8 \text{ bytes} \times 4 \approx 128 \text{ GB/s}; \text{i.e., 3 reads/1 write for radix partitioning})\), whereas merge performance drop significantly from 1.5 to 0.5 billion tuples per second. The performance drop is also in line with what we observed in single-threaded merge results in Figure 4.11.

4.10.3 Alternative Implementations for Merge

The new way of doing the merge seems promising but needs to be evaluated against alternative proposals from the literature. Chhugani et al. \([\text{CNL}^+08]\) claim that multi-way merging accommodates parallelism in a natural way and works well for multi-core processors. In this approach, each node in the merge tree that has enough data in its children nodes can be picked up by one of the idle threads and processed further. However, as the available degree of hardware parallelism increases, contention in large merge trees also increases. This contention is possibly not visible in four-core configurations considered by Chhugani et al. \([\text{CNL}^+08]\). But it critically affects performance in modern systems with a few ten hardware threads. It turns out that the new approach we propose to merging, also addresses this problem. We have confirmed this experimentally by implementing the two different approaches to multi-way merging:

The cooperative \(m\)-way approach follows the original idea by Chhugani et al. \([\text{CNL}^+08]\) where there is a single multi-way merge tree and multiple threads cooperatively merge the data. Once the children of a node have enough data, the node becomes available for merging by any of the threads. This approach has a potential advantage: It increases the buffer space per merge node as there is a single merge tree resident in last-level cache (in Section 4.9 we showed that buffer space is important for merge performance).

The independent sorting approach follows the Partition-then-Sort idea discussed in Section 4.10.2. Each thread locally partitions its data and then sorts smaller individual partitions. In this case, the thread can independently merge the sorted runs without further interaction with other threads.

Figure 4.14 shows the throughput of the two approaches when sorting an input relation of 256 M tuples. We intentionally restrict to a single socket to ensure that all threads use the same last-level cache. As the figure shows, cooperative multi-way merging does not scale for a variety of reasons: contention for upper-level nodes in the merge tree, idle threads due to lack of enough work, and synchronization overhead. The independent sorting approach, in contrast, scales linearly up to the physical number of cores as shown in Figure 4.14. However, the scalability in the hyper threads region remains limited. This is the common case for
hardware-conscious algorithms where most of the physical hardware resources are shared among hyper-threads. As a conclusion, even though all the threads have a fraction of the last level cache for their multi-way merge, the synchronization-free nature of this approach shows its benefit and independent sorting proves to be better than the originally proposed cooperative multi-way merging.

### 4.11 Evaluation of Sort-Merge Joins

After identifying the factors affecting the performance of the components of a sort-merge join algorithm and choosing the best-possible implementations for the different phases, we now compare the performance of the resulting sort-merge join operator (\textit{m-way}) with that of \textit{mpsm} and \textit{m-pass}. All the algorithms presented in this section are specially tailored to use 256-bit SIMD (AVX) instructions.

#### 4.11.1 Comparison with Different Table Sizes

We run first an experiment for different sizes of table \(S\) using Workload A shown in Figure 4.15. The results show that \textit{m-way} runs significantly faster than the other options and is robust across different relation size ratios while producing
4.11. Evaluation of Sort-Merge Joins

Figure 4.15: Execution time comparison of sort-merge join algorithms. Workload A, 64 threads.

fully sorted output. Algorithm mpsm draws its true benefit from only sorting the smaller of the input tables completely whereas the larger one is only partially sorted. Yet, mpsm can match the performance of m-pass only when the $S$ relation is significantly large ($\gg 12.8$ billion tuples (100 GiB)). Nonetheless, mpsm is a scalar algorithm applicable to wider keys as well. For reasons of space, we omit the results of scalar sorting-based joins that are also applicable to wider keys. In general, the scalar $m$-way has a good performance despite not using SIMD and performs better than mpsm even with 8-byte tuples and large $S$ relations.

In order to understand the behavior of sorting-based algorithms with larger equal sized relations $R$ and $S$, we ran an experiment where relation sizes are increased up to 2 billion tuples. The results of this experiment are shown in Figure 4.17. As expected, $m$-way runs significantly faster than all others and preserves the performance advantage with larger input relations. Interestingly, even the scalar version of this algorithm, which is not using SIMD acceleration, achieves a competitive performance in comparison to all other sort-merge join variants.

4.11.2 Execution Break Down and Throughput

Figure 4.16 shows execution time breakdown and throughput for the equal-sized table case in Workload A using 64 threads. First, the merge phase in $m$-way is 3 times faster than m-pass with bandwidth-aware multi-way merging. Second, in contrast to mpsm, the “mjoin” phase is a linear merge-join operation on NUMA-
Chapter 4. Sort-Merge Joins

Figure 4.16: Performance breakdown for sort-merge join algorithms. Workload A. Throughput metric is output tuples per second, i.e. \( \frac{|S|}{\text{execution time}} \).

Figure 4.17: Execution time of different sorting-based join algorithms for increasing table sizes where \( R \) and \( S \) has equal number of 8-byte tuples.
4.11. Evaluation of Sort-Merge Joins

Local sorted runs in the other algorithms and overhead of that phase becomes negligible.

Overall, due to these optimizations, \textit{m-way} reaches a throughput of up to 315 million output tuples per second while using all the 64 threads.

4.11.3 Dissecting the Speedup of \textit{m-way}

In order to understand the efficiency of \textit{m-way}, we calculated the speedup ratios of \textit{m-way} over the other algorithms (Figure 4.18). The bars denoted with “speedup from merge” shows the speedup of \textit{m-way} attained over \textit{m-pass}. This metric reflects the actual benefit of multi-way merging alone. As seen in the figure, up to 16 threads the speedup from multi-way merge is \( \approx 1.5X \) in which case there is enough aggregate memory bandwidth for that number of threads. However, once the number of threads go beyond 16, memory bandwidth per thread becomes limited and multi-way merge benefit goes up to a factor of 2. The bars denoted with “speedup from AVX” show the speedup of \textit{m-way} attained over the same algorithm’s scalar variant. Generally, speedup from AVX is between 2X and 2.3X. Lastly, the overall speedup of \textit{m-way} over \textit{mpsm} is \( \approx 3X \).

4.11.4 Scalability of Sort-based Join Algorithms

Figure 4.19 shows the scalability of sorting-based join algorithms with increasing number of threads where both axes are in logarithmic scale. All the algorithms exhibit linear scaling behavior up to 16 physical CPU cores. However, as all
Figure 4.19: Scalability of sorting-based joins with equal sized $R$ and $S$. Throughput metric is output tuples per second, \( \frac{|S|}{\text{execution time}} \).
of these algorithms are cache- and CPU resource-sensitive, the scaling with hyper threads is rather limited due to the contention on CPU resources by logical threads. Additionally, bandwidth unconscious \textit{m-pass} and \textit{mpsm} do not scale as linearly as \textit{m-way} for number of threads beyond 16.

### 4.11.5 Scalar Sorting-based Join Algorithms

Our focus in this thesis is implementing joins using SIMD instructions. However, as pointed out by Albutiu et al. [AKN12], this may not always be possible when operating on larger join keys. Consequently, in this section we particularly consider the same join workload setup where keys are 8-bytes, tuples are 16-bytes and all of the algorithms use scalar sorting without SIMD.

#### Comparison with Different Table Sizes

We run first an experiment for different sizes of table $S$ using Workload A (Figure 4.20). Due to lack of enough memory, we could run algorithms only up to 6.4 billion tuples in $S$. In addition to our own implementations, we also show the execution time numbers of \textit{mpsm} from Albutiu et al. [AKN12] which should be only taken as a rough reference point as the machines used are also slightly different. By looking at our \textit{mpsm} numbers, we can see that our code is a fair representation of the \textit{mpsm} idea. As pointed out by the authors, the \textit{mpsm} draws its true benefit from only sorting the smaller of the input tables completely whereas the larger one is only partially sorted. Consequently, per input tuple cost of this algorithm decreases as the outer table becomes larger. As a result, when $S$ grows, this algorithm can be competitive with the naive, complete sorting-based join algorithm \textit{m-pass}.

Figure 4.20 also shows that our optimized multi-way merging algorithm over NUMA regions has good performance even without using SIMD and \textit{m-way} performs better than \textit{mpsm} even when $S$ is relatively large.

#### Execution Time Break Down

Figure 4.21 shows the execution time break down of phases in scalar sort-merge join algorithms. In this setup, \textit{m-way} achieves a speedup of $\approx 2.3X$ over \textit{mpsm}. The figure also shows that \textit{mpsm} spends more than half of the time in merge-join of local and remote sorted chunks where local sorted chunk is repetitively scanned for each remote chunk. Although \textit{mpsm} is NUMA-efficient for reading remote chunks due to the sequential reads, it overloads the local memory bandwidth because of repeated local chunk scans. A similar pressure on local memory bandwidth is observable in \textit{m-pass} during the merge phase: First, remote NUMA-chunks are
read and merged pair-wise to the local memory. But then the rest of the merge iterates by pair-wise merging until fully-sorted output is created. However, in case of \textit{m-way}, this problem is resolved by merging all remote NUMA chunks in one pass using the multi-way merging. As shown in Figure 4.21, this technique shows its benefit where merge cost reduces almost by a factor of 3.

Overall, results presented in this section (Figure 4.20 and Figure 4.21) confirm that memory bandwidth efficient, complete sorting-based join algorithms, even without the support of SIMD, have an edge over recently proposed \textit{mpsm} that relies on sequential scans over NUMA regions.

### 4.11.6 NUMA Characteristics and Efficiency

In this section, we present the NUMA characteristics of our machine and NUMA efficiency of the merge phase.

**NUMA Characteristics of the Test Machine**

The NUMA topology of the test machine is shown in Figure 4.22. The machine has four sockets and each socket is connected to two other sockets with bi-directional QPI links. The third socket is only reachable through one of the other sockets and hence the topology is not fully connected.

A recent study by Li et al. [LPM+13] concluded that data shuffling must be done hardware-consciously for best utilizing the bandwidth of NUMA machines. We followed a similar methodology and our results are summarized in Figure 4.23.
4.11. Evaluation of Sort-Merge Joins

Figure 4.21: Comparison of scalar sort-merge join algorithms using cycles per output tuple metric. Workload A (23.84 GiB ⋉ 23.84 GiB), 64 threads.

In best approach from Li et al. [LPM+13], “ring-based shuffling”, four sets of four threads in each socket read data from remote and local NUMA regions in a topology-aware manner. The aggregate bandwidth achieved with this approach is a combination of local (1/4 of threads read local memory) and remote memory bandwidth. It performs around 64 GB/sec on our machine as shown in Figure 4.23.

Secondly, different from Li et al. [LPM+13], we measured the aggregate NUMA interconnect bandwidth by ensuring that none of the threads read from the local memory in the previous approach. The maximum aggregate bandwidth that can be achieved reaches up to 48 GB/sec, which is below 4 × 16 GB/sec = 64 GB/sec theoretical bandwidth of the QPI interconnect specified by Intel. While not clearly described by Intel, the discrepancy likely stems from the extra messages exchanged due to system’s cache coherency protocol. Lastly, we measured the aggregate memory bandwidth as 120 GB/sec when all the accesses are local.
Chapter 4. Sort-Merge Joins

Figure 4.23: NUMA characteristics of our test machine with different memory access policies.

Table 4.7: Profiling of NUMA efficiency.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instr./Tup.</th>
<th>Cyc./Tup.</th>
<th>IPC/core</th>
<th>L3-Miss/Tup.</th>
<th>BW.-GB/s.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mway</td>
<td>55.47</td>
<td>1.19</td>
<td>1.45</td>
<td>0.17</td>
<td>66.21</td>
</tr>
<tr>
<td>mpass-remote</td>
<td>6.34</td>
<td>0.81</td>
<td>0.25</td>
<td>0.15</td>
<td>70.35</td>
</tr>
<tr>
<td>mpass-local</td>
<td>36.96</td>
<td>2.85</td>
<td>0.41</td>
<td>0.37</td>
<td>135.78</td>
</tr>
<tr>
<td>scalar mway</td>
<td>187.78</td>
<td>1.95</td>
<td>3.01</td>
<td>0.14</td>
<td>39.65</td>
</tr>
<tr>
<td>memcpy w/ ring shuf.</td>
<td>1.39</td>
<td>0.89</td>
<td>0.05</td>
<td>0.14</td>
<td>66.96</td>
</tr>
</tbody>
</table>

NUMA Efficiency of Merging

As a baseline for merging, we conducted a memory copy microbenchmark using the ring-based shuffling approach. The data read by threads from remote NUMA-regions are copied to NUMA-local memory without merging. Therefore, this approach is a bare baseline for merging with a bandwidth of $\approx 67 \text{ GB/sec.}$ Table 4.7 presents the results of profiling NUMA efficiency of different approaches in comparison to the basic memory copy approach.

Multi-pass merging consists of two steps. In first step, “mpass-remote”, all the remote data are brought to the local node and the first pass of the merging is done on-the-fly. As a result, 128 remote sorted runs are merged into 64 local sorted runs. Table 4.7 strongly suggests that this step is memory bandwidth bound: “IPC/core” is very low and bandwidth achieved (“BW.-GB/s”) is slightly above our memcpy baseline (improvement is due to use of SIMD). In the second step, “mpass-local”, 64 sorted runs are iteratively pair-wise merged and after six-passes, a single sorted run is created in local memory. In this case, SIMD loads/stores bring an improvement on total aggregate bandwidth achieved. However, performance counters strongly suggest that this step is severely memory bandwidth bound.

On the other hand, the “mway” numbers indicate that the memory bandwidth-bound characteristic of merging shifts into a balance with CPU utilization. Multi-
way merging not only overlaps the merging and data transfers between different
NUMA-regions, but also turns the six-pass merging into a single-pass merging.
The results in Table 4.7 address the efficiency of “mway” over “mpass”. An IPC
of 1.45 for AVX is close to being CPU-bound in the Intel Sandy Bridge micro-
architecture. On the other hand, the bandwidth achieved suggests that “mway”
utilizes memory bandwidth efficiently while not being memory-bound (cf. compare
with “mpass-remote” and “memcpy”). In terms of cache efficiency, 0.17 miss per
tuple is very close to the baseline and “mway” successfully keeps the working set
in the L3 cache. Finally, the improvement of “mway” over its scalar version is also
clearly visible in the table, where for instance number of instructions are cut by
3.4 times.

4.12 Summary

This chapter of the thesis visited the design choices for implementing sort-merge
joins on modern multi-core CPUs. The combination and preference of the avail-
able choices (e.g., use of SIMD, use of partitioning, and use of multi-way merge)
resulted in several sort-merge join algorithm variants. Through the experimental
evaluation of the available choices, we identified the best sort-merge join algo-
rinthm among the options as the m-way algorithm. Our results show that this is
the fastest in-memory sort-merge join algorithm on modern multi-core CPUs. It is
2–3 times faster than available results known to us. The results also demonstrate
the importance of awareness of architectural hardware features and tuning to them
in algorithm design.
5

Sort vs. Hash Revisited

5.1 Introduction

“Sort or Hash?”—For in-memory joins, this question has always been answered in favor of hash-based strategies. Long disk seek times, an enemy for the latency-sensitive random-access patterns of hashing, do not arise when data is kept entirely in memory. And hash joins can be parallelized almost trivially, a benefit that promises scalability and performance also for large data sets.

But evolving hardware characteristics give rise to re-open the classical question. The gap between DRAM latency and bandwidth keeps widening; and multiple flavors of parallelism (e.g., multi-core parallelism, vector instruction sets) may favor different join processing strategies.

In this chapter, we explore the relative performance of radix-hash vs. sort-merge join algorithms in main-memory, multi-core settings. Our main goal is to analyze the hypothesis raised by recent work claiming that sort-merge joins over new hardware are now a better option than radix-hash joins, the algorithm traditionally considered to be the fastest [AKN12, KSC+09]. Kim et al. [KSC+09] have suggested that, once hardware provides support for vector instructions that are sufficiently wide (SIMD with 256-bit AVX and wider), sort-merge joins would easily outperform radix-hash joins. This claim was reinforced by recent results by Albutiu et al. [AKN12] who report that their NUMA-aware implementation of sort-merge join is already superior to hash joins even without using SIMD instructions. Furthermore, there are a number of new optimizations for parallel
radix join [BTAÖ12, BTAÖ13, BLP11] that have not been considered in these studies, but that should be part of any analysis of the relative performance of the two options.

In this chapter, we approach the question experimentally. We bring carefully-tuned implementations of all relevant, state-of-the-art join strategies (including \textit{radix join} [BTAÖ13, KSC⁺09, MBK02b], \textit{no-partitioning join} [BLP11], \textit{sort-merge join} [KSC⁺09], and \textit{massively parallel sort-merge (MPSM) join} [AKN12]) to a common and up-to-date hardware platform. We then compare the relative performance of all these algorithms under a wide range of experimental factors and parameters: algorithm design, data sizes, relative table sizes, degree of parallelism, use of SIMD instructions, effect of NUMA, data skew, and different workloads. Many of these parameters and combinations thereof were not foreseeable in earlier studies, and our experiments show that they play a crucial role in determining the overall performance of join algorithms.

Through an extensive experimental analysis, this chapter makes several contributions: (1) we show that radix-hash join is still superior to sort-merge join in most cases; (2) we provide several insights on the implementation of data operators on modern processors; and (3) we present the fastest algorithms available to date for both sort-merge—2-3 times faster than available results—and radix-hash join, demonstrating how to use modern processors to improve the performance of data operators.

In addition, the chapter sheds light on a number of relevant issues involving the processing of “big data” and the factors that affect the choice of the algorithm. These include:

\textbf{Input Sizes.} Our results show that the relative and absolute input table sizes have a big effect on performance. Moreover, as the data size grows, the duality between hashing and sorting becomes more pronounced, changing the assumption that only hashing involves several passes over the data when it is sufficiently large. We show that sort-merge joins also have to do multiple passes over the data, with their performance suffering accordingly.

\textbf{Cache Contention.} Cache-conscious approaches make a significant difference in both hash and sort-merge joins. Cache consciousness becomes the decisive factor in choosing the best hash join algorithm, favoring the radix join approach of [BTAÖ13, KSC⁺09] over the no-partitioning approach of [BLP11]. As shown in Chapter 4, sort-merge joins when implemented in a cache-conscious manner performs much better than the alternative sort-merge join algorithm variants. However, increasing number of cores and threads with parallelism poses challenges on the shared cache usage and the assumptions that cache-conscious sort-merge joins work well for multi-cores no longer hold for large.

The results in this chapter show that hash joins still have an edge over sort-
merge alternatives unless the amount of data involved is very large. Some advances in hardware will eventually favor sort-merge joins, e.g., wider SIMD registers and higher memory bandwidth, but our results show that exploiting such advances will also benefit radix join algorithms. Furthermore, new processor features such as memory gather support in Intel’s upcoming Haswell series may play a bigger role in improving hash joins than the factors considered so far.

5.2 Related Work

5.2.1 Sort vs. Hash—Early Work

“Sort or hash” has long been a point of discussion in databases. Initially, sort-merge join was the preferred option [Mer83]. Later, the invention of hashing-based techniques [Bra84, KTMO83] changed the balance. Schneider et al. [SD89] compared hash-based with sort-merge joins and concluded that hash-based joins were superior unless memory was limited. Hash join was also the main choice in most of the early parallel database systems [DGG86, F+86, SD89].

Changes in hardware, memory, and data volumes prompted researchers to revisit the “sort or hash” question regularly over the years. Graefe et al. [GLS94] provided an extensive study of sort- and hash-based query processing algorithms. They outlined the dualities and similarities among the approaches and concluded that performance only differs by percentages rather than factors if both algorithms are implemented in a careful and equally-optimized manner. Moreover, the study pointed out that there are cases when one of the algorithms would be preferable over the other and therefore both algorithms would be needed in a database system. Subsequently, Graefe et al. [Gra94] used histogram-based partitioning to improve hash-based joins and finally concluded that sort-merge joins only win in a number of corner cases.

5.2.2 Sort vs. Hash—Multi-core Era

Multi-core has changed things once again. Kim et al. [KSC+09] compared parallel radix-hash join with a sorting-based join exploiting both SIMD data parallelism and multiple threads. They concluded that wider SIMD registers will soon make sort-merge a better option.

Albutiu et al. [AKN12] presented a “massively parallel sort-merge join” (MPSM) tailored for modern multi-core and multi-socket NUMA processors. MPSM is claimed to observe NUMA-friendly access patterns and avoids full sorting of the outer relation. In their experiments on a 32-core, four socket machine, they report that sort-merge join is faster than the “no-partitioning” hash join implementa-
tion of Blanas et al. [BLP11] (see below). Unlike the projections of Kim et al. [KSC+09], the claim is that sort-merge join is faster than hash join even without using SIMD parallelism.

On the hash join side, cache-aware, partitioning-based algorithms such as “radix join” provide the best performance [MBK02b, SKN94]. More recently, Blanas et al. [BLP11] introduced the “no-partitioning” idea and advocated its simplicity, hardware obliviousness, and efficiency. Recent work has studied these hash join algorithms and showed that hardware-conscious, parallel radix join has the best overall performance [BTAÖ12, BTAÖ13]. As the code for the no-partitioning and radix join algorithms is available, we use these algorithms in this study. We also refer to the literature for detailed descriptions of these algorithms and focus here mainly on sort-merge joins.

In this chapter, we revisit the “sort or hash” question in the light of the recent results and projections. The hardware available today (such as one with 256-bit SIMD registers) partly enables us to see the validity of previous projections and allows us to conduct more extensive experiments comparing existing algorithms and claims.

5.3 Hash Join Algorithms

In Chapter 3, we discussed in-memory hash joins in detail. For the coherency of the discussion in this chapter, we briefly recap hash-based join algorithms.

The principle of hashing is direct positional access based on a key’s hash value in $O(1)$ complexity and it is appealing in many query processing algorithms such as joins and aggregation. Traditional hash-based algorithms usually consist of two phases in which a hash table is first built and then probed. While efficient, hashing results in random access to memory, which can lead to cache misses. Shatdal et al. [SKN94] identified that when the hash table is larger than the cache size, almost every access to the hash table results in a cache miss. As a result, a partitioning phase to the hash joins is introduced to reduce cache misses. The performance of the resulting join is largely dictated by this partitioning phase.

5.3.1 Radix Hash Join – radix

For parallel radix-hash join, we partition both input relations as discussed in Section 4.5.2. The goal is to break at least the smaller input into pieces that fit into caches. Then, we run a cache-local hash join on individual partition pairs. For a detailed discussion, refer to Chapter 3 of the thesis and [BTAÖ13, BLP11, KSC+09, MBK02b].
5.3.2 No-Partitioning Hash Join – $n$-part

The no-partitioning join is a direct parallel version of the canonical hash join. Both input relations are divided into equi-sized portions that are assigned to a number of worker threads. In the build phase, all worker threads populate a shared hash table with all tuples of $R$. After synchronization via a barrier, all worker threads enter the probe phase and concurrently find matching join partners for their assigned $S$ portions. For further details, we refer to Chapter 3 of the thesis and [BTAÖ13, BLP11].

Table 5.1 summarizes the algorithms considered in the experiments and the shorthand notation used in the graphs.

5.4 Experimental Setup

The experimental setup in this chapter is the same as the one used in Chapter 4. For the description and the details of the experimental setup and the hardware platform used we refer to Section 4.7 of Chapter 4. The workloads used are named as Workload A and B here too.

5.5 Sort or Hash?

In this section, we present the best sort and hash join algorithms side-by-side under a wide range of parameters.

5.5.1 Comparison with Different Workloads

The results of best sorting-based join algorithm $m$-way and best hash join algorithm radix in our study are summarized in Figure 5.1 over various workloads. The first two workloads replicate the setups used by the related work [KSC*09, AKN12, BTAÖ13]. The other workloads extend the setup by using a
larger foreign key relation $S$. We kept the ratio of two relations as 4 in this experiment to reflect real-world workload settings (we also have experiments with larger and varying ratios).

In Figure 5.1 we observe that hash-based join algorithms still maintain an edge over sort-based counterparts. When input table sizes are in the hundred millions, \textit{radix hash join} is more than 2 times faster than \textit{m-way sort-merge join}. The speed difference is maintained even when the outer table size becomes significantly larger than the primary key table. Despite all the optimizations discussed in this thesis and the performance boost through 256-bit vector instructions, sort-merge joins cannot match the performance of hash joins on existing recent hardware and for these workloads.

For significantly larger workloads (as suggested by Albutiu et al. [AKN12]) the picture becomes more favorable for sort-merge joins. Input table sizes in the range of billions of tuples make \textit{m-way} more competitive with \textit{radix}. Since \textit{m-way} produces fully sorted output, it could be a good choice for large data volumes that must be further processed.

The main reason behind sort-merge being competitive with hash join lies in the growing data sizes of “big data” age. As data sizes grow, join algorithms are likely to require multiple passes over the data. This appears as no surprise in radix join where the number of partitioning passes goes to two for billion tuples range and
the contribution of partitioning becomes more pronounced. On the other hand, architectural advances such as AVX boost sort-merge join but does not make it faster than hash joins yet, in contrast to the projection by Kim et al. [KSC+09].

5.5.2 Effect of Input Size

The effect of the input relation size can be better captured by the following experiment: we vary the size of each equi-sized $R$ and $S$ tables from 128M tuples ($\approx 1$ GB) to 1,920M tuples ($\approx 15$ GB) and run $m$-way and radix at each data point using 64 threads. For radix join, we ran the algorithm through all the possible radix configurations (number of passes between 1 and 2 and total radix bits between 6 and 20) and determined the best radix configuration for this experiment. The results are shown in Figure 5.2.

The conclusion is clear: sort-merge join approaches the performance of radix only when the input tables become significantly large. Radix join performance degrades with the increasing input size due to the increasing number of passes for the partitioning phase. To illustrate, radix configurations vary from 1 pass/12 bits up to 1 billion tuples and from that point on resorts back to 2 pass/18 bits optimal configurations. The optimized radix partitioning with software-managed buffers is very efficient up to 9-bits/512-way partitioning since the entire partitioning buffer can reside in L1 cache ($32 \text{KiB} = 512 \times 64\text{-bytes-cache-lines}$). Even higher radix-bits/fan-outs such as a maximum of 12/4,096 can be tolerated when backed up by the L2 cache ($256 \text{KiB} = 4,096 \times 64\text{-bytes}$). Consequently, the partitioning performance degrades gracefully up to L2 cache size. Once the partitioning requirement goes above 16,384, 2-pass partitioning, each pass with 9-bits, and fully L1-resident buffers become the best option. Further, sort-merge join demonstrates robust performance due to the optimizations previously discussed in this thesis. Finally, sorted output in query plans is an attractive argument to make sort-merge joins even more favorable in this range of the spectrum of input sizes.

We have also analyzed the potential impact of relative table sizes (partly shown in Figure 5.1). The experiments show that both algorithms are insensitive to the variance in relation size ratios, being affected only by the size of the output as the amount of data grows.

5.5.3 Sort vs. Hash, 20 Years Later

For the historical perspective, we present a one-to-one replica of an experiment from 20 years ago by Graefe et al. [GLS94]. The results are presented in Figure 5.3. It is interesting to see that the number of input tuples went from 10K – 500K in the original experiment to 32M – 2000M in the present experiment, a factor of $\approx 4000$ and a total size factor of $\approx 150$ considering tuple widths. Although a
direct comparison is not too meaningful, execution time changed from 50 sec. – 5000 sec. to 50 msec. – 7.5 sec., a factor of $\approx 1000$. More importantly, one thing has a slight variation: In the present experiment, sort vs. hash performance is almost a factor of 2 up to significant amount of data and only afterwards differ by percentages, whereas in the original experiment the difference was mentioned as percentages in most of the cases.

### 5.5.4 Effect of Skew

Skew in key distributions generally has negative impact if it is not properly handled. In this section, we take a look at the effect when the foreign key column in table $S$ follows a Zipf distribution when referencing the key column of $R$.

For handling skew in parallel radix hash join, we previously proposed a fine-granular task decomposition method. The key idea is to parallelize the processing of larger partitions through a refined partitioning. We refer to [BTAÖ13, KSC+09] for details.

We briefly describe our skew handling mechanism for the $m$-way sort-merge algorithm. The first and second phases in $m$-way are not vulnerable to skew since all the threads have an equal share of the data and each thread locally partitions
5.5. Sort or Hash?

Figure 5.3: Running the same experiment of Graefe et al. after 20 years: Sort vs. hash join performance for equal input sizes with 8-byte tuples.

and sorts their chunks. Phase 2 in Figure 4.9, multi-way merging over different NUMA-regions, is prone to skew. Each thread merges partitions from all of the other threads. When some of the partitions have more data than the others, the data to be merged by some threads will exceed data of other threads and cause a load-imbalance.

We handle the skew in two steps: 1) When creating a multi-way merge task for the thread, if the total size of the merge exceeds an expected average size, we create multiple merge tasks by finding boundaries within each of the sorted runs with binary search. These tasks are then inserted into a NUMA-local task queue shared by the threads in the same NUMA region. 2) For extremely large tasks, we identify heavy hitters by computing an equi-depth histogram over sorted runs (which is a fast operation) in a similar approach to Albutiu et al. [AKN12]. Then heavy hitters are explicitly handled and directly copied to their output targets without a need for merging.

Even creating multiple multi-way tasks and using a task queueing mechanism may not handle the skew when a task with a large total size resulting from heavy hitter keys occur. In this case, we determine the heavy hitter keys utilizing sorted runs and efficiently compute an equi-depth histogram. After identifying heavy hitters through the equi-depth histogram and their boundaries with binary search, we create copy tasks instead of merge-tasks for those heavy hitters. Therefore, heavy hitters are directly copied to their target locations instead of merging.

Figure 5.4 illustrates the behavior of \textit{m-way} and \textit{radix} with increasing Zipf
skew factor. The enhancements to m-way, explicit skew and heavy-hitter handling mechanisms, result in a robust performance against skewed distribution of keys while showing less than 10% overhead for reasonably high skew. On the other hand, radix join is also robust against skew with the fine-granular task decomposition technique. Overall, the results in Figure 5.4 show that both algorithms can be made skew-resilient and the comparison of sort vs. hash joins does not significantly change due to skew.

5.5.5 Scalability Comparison

We compare the algorithms in terms of scalability by varying the number of threads up to the available 64 threads in our system. Threads are assigned to NUMA regions in a round-robin fashion. Between 32 and 64 threads, algorithms use SMT (hyper-)threads. Results for two different workloads are shown in Figure 5.5.

First, both algorithms show almost linear scalability up to the physical number of cores. Consequently, radix hash join achieves approximately 650 million whereas m-way sort-merge join achieves approximately 315 million output tuples per second. As mentioned earlier, performance gap between algorithms closes only with significant input sizes as in Figure 5.5(b), where the number of passes for radix partitioning increases to two. On the other hand, only within the SMT region, algorithms scale poorly. This is an inevitable, well-known situation for hardware-conscious algorithms: SMT provides the illusion of running 2 threads on a single core where almost all of the resources are shared. Therefore, hardware-conscious algorithms either benefit minimally or do not benefit from SMT.
Figure 5.5: Scalability of sort vs. hash join. Throughput is in output tuples per second, i.e. \( \text{[output tuples/sec]} \).
Finally, SMT scalability for radix join are different in the two workloads mainly because of the optimal partitioning configuration: In Figure 5.5(a), radix join runs with single-pass partitioning with fan-out of 4,096 that fully stresses the L2 cache. Whenever the other thread starts to contend for the L2 cache, SMT shows no benefit apart from hiding the self-caused misses. However, in Figure 5.5(b), the partitioning buffer is L1 resident with a fan-out of 512 in each of the two passes. Therefore, a limited benefit can be observed where certain amount of L1 misses are hidden by SMT.

5.5.6 Sort vs. Hash with All Algorithms

In this section, we bring all of the state-of-the-art join strategies together for a side-by-side comparison using common workloads. The results are shown in Figure 5.6. Radix hash join comes out as the fastest algorithm in this comparison. Albutiu et al. [AKN12] previously compared their massively-parallel sort-merge join (mpsm) algorithm to no-partitioning join (n-part) implementation of Blanas et al. [BLP11] and found out that sort-merge joins are already faster than hash joins. However, in our recent work [BTAÖ13], we have optimized the no-partitioning join idea further (nevertheless it is still not fully NUMA-aware as the hash table is spread over NUMA regions). Therefore, we extend the comparison of Albutiu et al. [AKN12] with our optimized implementations of the algorithms. The results in Figure 5.6 indicate that mpsm and n-part algorithms in fact achieve similar performance. Optimized n-part is only faster by ≈10-15% while lacking the partially sorted output benefit of mpsm. Nevertheless, all the results clearly indicate that hash joins are still faster than sort-merge join counterparts.

5.5.7 Effect of Relative Table Sizes

As input size plays an important role in sort vs. hash discussion, it is legitimate to question whether the relative table sizes also play a role. In order to answer this question, we set-up the following experiment: While fixing the size of the inner table at 128M tuples, we vary the size of the outer table from 128 million to 1024 million tuples. We show the comparison of radix hash join and mway sort-merge join algorithms in Figure 5.7 using 64-threads. For radix join, we configured to run with the optimal configuration.

Figure 5.7 shows the results of this experiment. As clearly visible in the figure, both algorithms are insensitive to the variance in relation size ratios. Cycles-per-output tuple count decreases in both algorithms as a direct result of increasing outer table size. More importantly, for this setup, the speed ratio of the algorithms do not change significantly and radix hash join maintains its performance advantage.
5.5. Sort or Hash?

Figure 5.6: Sort vs. hash join comparison with extended set of algorithms. All using 64 threads.

Figure 5.7: Effect of relative table sizes in join performance, using 64 threads.
5.5.8 Sort vs. Hash for other Database Algorithms

In this section, we briefly take a look at other database algorithms in terms of the sort vs. hash discussion. In addition to relational joins, many other relational operators such as group by, projection and duplicate elimination operators can also be implemented with sorting-based algorithm variants. Therefore, these algorithms can also utilize and reuse most of the techniques discussed in this chapter and Chapter 4.

As a small case study, we describe how to apply the design principles of the best sort-merge join algorithm \textit{m-way} to implement the group by aggregation operator. As an alternative of sorting-based aggregation, we also describe a state-of-the-art hashing-based aggregation. Finally, we briefly compare the performance of sorting and hashing based aggregation implementations without going into further details.

\textbf{Sorting-based Aggregation}

The group by operator takes only a single relation as input and most of the discussion from Chapter 4 can be extended for its implementation. The algorithm shown in Figure 4.9 applied on \( R \) produces the sorted version \( R' \) by applying phases 1 and 2, partitioning/sorting and multi-way merging respectively. As a difference to the join algorithm shown in Figure 4.9, the last phase (phase 5) must go over the NUMA-local runs and aggregate over the sorted keys. This results in a very efficient, single pass aggregation algorithm over the sorted relation.

As a further optimization, before merging NUMA-remote runs in phase 2, it is possible to do a partial aggregation on local sorted runs before doing the merge. This optimization results in a significant benefit if the cardinality of the \textit{group by key} is smaller than the local run size. As a result, less data is shipped between NUMA regions and merged in phase 2. This is a classic optimization for processing aggregates in databases [Eps79, GLS94]. Furthermore, as a special instance of aggregation, duplicate elimination operator can also be carried out with a similar algorithm [BD83].

\textbf{Hashing-based Aggregation}

In a similar vein to hash join, group by aggregation is also sensitive to cache and TLB parameters of the underlying system. One of the state-of-the-art aggregation algorithms proposed in literature is called Partition with a Local Aggregation Table (PLAT) [YRV11]. PLAT includes a data partitioning phase when the number of distinct group by values go beyond system cache and TLB sizes. The main idea of PLAT is extending the partition and then aggregate technique by using a local private table during the partitioning. As a result, partitioning is used in this algorithm when the tuples miss the local table. We use an implementation of
5.5. Sort or Hash?

PLAT in our experiments. For a detailed description of the algorithm we refer to [YRV11].

**Experimental Evaluation**

*Experimental Setup.* In the experimental setup input relation size is fixed at 2 billion 8-byte tuples consisting of 4-byte keys and 4-byte values. The group by cardinality is varied from 128 unique groups to 536 million. The aggregation query is computing the number of occurrences of each distinct key in the permuted input relation (*i.e.*, SELECT key, COUNT(*) GROUP BY key). The machine used in the experiments is the Intel E5-4640 that is described in Section 4.7.

Figure 5.8 shows the comparison of sort and hash based aggregation algorithms described in this section. First, looking at the numbers of the hash-based aggregation PLAT, one can observe a pattern of the cache hierarchy of the Intel E5-4640. Up to group by cardinalities where the entire hash table can be kept in the local L1 cache of each core, PLAT performs significantly faster than the sorting-based aggregation algorithms. In this region, the choice of algorithm is clearly the hashing-based algorithm. In the next region, up to group by cardinalities fitting in the L2 cache, hash-based aggregation performs similarly to the sort-based aggregation. However, once the partial aggregation optimization described earlier in this section is applied, the sort-based aggregation becomes a better option than the hash-based
Chapter 5. Sort vs. Hash Revisited

aggregation. In the last region, up to group by cardinalities fitting in the L3 cache size, the performance further degrades for the hash-based aggregation algorithm. Sort-based aggregation maintains the performance advantage in this region over the hash-based algorithm.

An important fact to note is that the advantage of the sort-based aggregation algorithms comes mainly from the acceleration with SIMD. While this is appreciable for narrow input tuples, for larger tuples the performance advantage can only be preserved with wider SIMD registers in future processor architectures. Lastly, Figure 5.8 also demonstrates an important feature of the sort-based algorithms: Sort-based algorithms are robust regardless of the input cardinality. Therefore, sort-based aggregation algorithms constitute an important preference when the input group by cardinality cannot be estimated well in advance.

This section briefly touched the discussion of sort vs. hash for other database algorithms. As the results show, there is still a case for including both type of algorithms in a database engine. Moreover, the preliminary results in this section open up an important area for investigation. Sort-based algorithms are fairly more robust to input characteristics and can be competitive alternative against hash-based algorithms for some of the database operators. Further research is needed to explore and investigate these alternatives on a broader spectrum with extensive experimental analysis.

5.6 Conclusions

As hardware changes, the “Sort vs. Hash” question needs to be revisited regularly over time. In this thesis, we look at it in the context of in-memory data processing for recent multi-core machines. Our results provide the fastest in-memory join processing algorithms using sorting (2–3 times faster than available results) and hashing. Moreover, we show that hash-based join algorithms still have an edge over sort-merge joins despite the advances on the hardware side such as 256-bit SIMD. Finally, sort-merge join turns out to be more comparable in performance to radix-hash join with very large input sizes.
6 Conclusions

6.1 Summary

The data volume pouring into the information systems of enterprises is increasing rapidly. The data processing systems such as databases are challenged to process these ever increasing volumes of data. On the technology side, the processor architectures are becoming faster and more advanced than ever before. However, to utilize the full potential of modern computing hardware, there is a pressing need for tailoring systems and algorithms to the underlying hardware. Essentially, the data processing systems must follow a hardware-conscious design principle. Databases need to become more hardware-aware and data processing algorithms must be tuned to the underlying hardware. Data processing systems can address the data volume challenge by benefiting from the abundant levels of parallelism and sophisticated hardware features. Moreover, new data processing algorithms and database architectures should be designed to take full advantage of the hardware evolution.

The thesis took a step towards the goal of achieving hardware and architecture conscious design of database management systems. The main-memory relational joins—a complex and a demanding operation in databases— are studied in-depth on modern processor architectures. The thesis discussed a broad range of algorithms for implementing main-memory joins on multi-core processors. In addition to analyzing the algorithms proposed in the literature, the thesis also proposed optimizations for hash and sort-merge joins leading to new algorithms that are more
efficient than the existing ones. Through optimizations and tuning of the proposed algorithms to modern processor architectures, the thesis provided the fastest in-memory join algorithm implementations to date. The performance achieved for in-memory joins in this thesis is significantly faster than the existing ones in the literature for both hash and sort-merge joins, demonstrating how to use modern processors to improve the performance of data operators. The source code of the algorithms are publicly released to contribute towards hardware-conscious design of data operators on modern hardware.

As joins are an important problem for database systems, there have been a number of significant claims that have appeared in the literature. These claims in the related work created a rather inconclusive set of views regarding how to design and implement main-memory joins on multi-core processors. This thesis brought all different approaches for implementing joins under a common setup for an extensive study. By doing so, the thesis addressed the existing controversy in the literature.

First, it has been shown in the thesis that hash joins that are hardware-conscious perform better than hardware-oblivious approaches under a broad range of setup. The analysis and the comparisons in the thesis showed that many of the claims regarding the behavior of hash join algorithms are due to selection effects (relative table sizes, tuple sizes, the underlying architecture, using sorted data, etc.) and are not supported by experiments run under different parameter settings. Moreover, hardware-conscious hash join algorithms provide the fastest implementation for joins on multi-core and are tunable to different hardware platforms in addition to being robust to different parameter settings.

Second, there has been a debate over the relative performance of sort vs. hash join algorithms in the literature. While this is a classical debate in database research community for many years, it has recently become important and more relevant given the architectural trends in processors. SIMD vectorization units in modern hardware enable data-level parallelism which can significantly speed up sort-merge joins. It has been projected that as the SIMD units become wider, sort-merge joins will become faster than hash joins that are traditionally known to perform better. After providing the best implementations of hash and sort-merge joins, the thesis experimentally studied the performance of main-memory, parallel, multi-core joins with both sort-merge and (radix-)hash joins. The results of extensive experiments on the original and optimized versions of these algorithms show that, contrary to existing claims in the literature, radix-hash join is still clearly superior, and sort-merge approaches to performance of radix-hash join only when very large amounts of data are involved. On the other hand, while SIMD makes sort-merge a good alternative algorithm for implementing joins, it is not yet there to tip the decision to sort-merge joins completely. Nevertheless, sort-merge
joins are still promising as future generations of processors are likely to invest chip area for increasing the width of SIMD units. Therefore, not 256-bit AVX but wider SIMD has potential to make sort-merge joins faster than hash joins. In current processors, hash join still has an edge over sort-merge join.

6.2 Directions for Future Work

Result Materialization on Multi-Core. The common setup considered for in-memory joins in related work is a column-oriented storage model (i.e., column-store database). In this setup, joins are evaluated on narrow tuples consisting of a key and a record identifier. Moreover, in column-store databases, the final result materialization is usually deferred until the last processing stage of the query where results are presented to the user [AMDM07]. Therefore, similar to the related work in this field, we focused on the actual join computation by ignoring the final result materialization. Although we did not study the materialization problem, we believe that it leads to an interesting research direction given the advances with multi-core processors. To address the problem of materialization, Manegold et al. [MBNK04] propose cache-conscious and efficient techniques. However, at the time they designed and evaluated those techniques, multi-core machines did not exist. Therefore, the materialization operation deserves an in-depth revisit and investigation on multi-core processors. It is likely that materialization can also benefit from some of the optimization techniques such as software-managed buffers discussed in this thesis. Moreover, given the architectural changes, materialization algorithms might need a redesign.

An extension to the research direction mentioned above is investigating the applicability and performance of the algorithms on joins with larger tuples or traditional row-store databases. The research in this area might help legacy database management systems to benefit from the algorithmic advances of newer systems.

Cost Models for Joins on Multi-Core. We have shown in this thesis that every hardware is a different story. Each different platform requires a certain amount of tuning to get the best performance out of hardware-conscious radix hash join. Therefore, there is a need for a cost-modeling to understand the performance of joins on different machines. Manegold et al. [MBK02a] address this problem by providing generic cost models to estimate the performance of radix join. Their cost models depend on memory access patterns and calibrated hardware parameters such as cache, TLB latencies. They show that their models accurately predict both the performance of radix join and its ideal configuration parameters. While we provided an empirical model in this thesis for identifying the configuration parameters of radix join, there is still a need for cost modeling on multi-core.
The hardware advances with multi-core make it a more challenging problem than what Manegold et al. assumed on single-core processors back then. In modern processors, counting just cache misses due to the algorithmic behavior might not be sufficient to model the costs. Sophisticated automatic hardware prefetchers, multiple threads sharing processor cores, SMT and cache sharing in multi-core designs strongly require revisiting the cost modeling of joins on multi-core.

**Combined Operators on Multi-Core.** The thesis studied mainly the relational join operator on multi-core. We only briefly touched upon the discussion of the group by operator regarding the sort vs. hash question. The state-of-the-art implementation of a group by operator essentially operates in a similar way to a join. On one hand, the radix join first partitions the input into cache sized chunks and evaluate joins on smaller partitions. On the other hand, the PLAT [YRV11] group by algorithm also partitions the input table first and then evaluates aggregates over smaller partitions. Due to the similarities in the main stages of these operators, the repeated work in these algorithms can be eliminated whenever these operators appear one after another. The new combined operator applies the partitioning once and evaluates the join and aggregation together to achieve a better performance on multi-core. In addition to eliminating redundant processing, a better cache locality can be achieved by immediately processing the data once it is in the cache. While we have initial promising results in this direction, more research is needed to show under what circumstances these two operators should be combined.

**Multiple Joins.** In this thesis, we focused on executing a single join operator in a best way on multi-core. However, there are two problems that needs to be addressed. First, in a real database management system there can be multiple number of concurrently arriving join queries. Second, multi-core machines are becoming larger and larger by providing tens of cores. It might simply not be feasible to reserve all the processing cores for a single operator. These problems need an investigation on how to schedule multiple concurrent parallel joins in a given large multi-core machine. As these algorithms are cache-conscious, the interference of one join to another can be detrimental. Therefore, the scheduling mechanisms should be aware of the underlying hardware in terms of architectural features such as cache and execution unit sharing. The approach of partitioning of machine resources among different joins can be viewed as a spatial scheduling approach. The opposite of this approach is temporal scheduling. In this approach, one join query uses all the processor cores until completion without any interference. Once this join finishes, another join operator can be picked from the queue to be executed. As we have shown in this thesis, even join queries with billions of tuples complete in several seconds when all of the machine resources are used without contention.
Therefore, by applying a smart scheduling at the queue level, the temporal scheduling approach might outperform the spatial scheduling approach. Further research is required to investigate these approaches for execution of multiple number of join operators.

**Distributed Joins.** Large deployments of database management systems are typically organized in a distributed setup. The data is spread over multiple number of machines for both scalability and high availability. While the amount of memory per machine has a trend of increase, distributed setup is also a good strategy to increase the amount of total available memory. Therefore, in addition to considering single large machine joins, investigation of join processing over a cluster of machines is an interesting research direction. The algorithms discussed in this thesis can be used as building blocks of join processing within each machine. The data exchange part among the machines can utilize advanced networking technologies such as remote direct memory access (RDMA). We believe that the networking part is likely to introduce additional research problems that have not been covered in this thesis.


