DATA PROCESSING TECHNIQUES ON MODERN HARDWARE ARCHITECTURES

by

Dimitrios Tsirogiannis

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2011 by Dimitrios Tsirogiannis
Abstract

Data Processing Techniques on Modern Hardware Architectures

Dimitrios Tsirogiannis
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2011

The last decade has been characterized by radical changes in the computing landscape. We have witnessed the advent of multi-core processors, flash-based storage systems and the proliferation of scale out architectures, such as map-reduce-based systems and massively parallel databases. Although data management systems have embraced modern hardware technologies to some extent, they have not realized their full potential.

The goal of this thesis is two-fold. Primarily, it demonstrates the staggering potential for performance improvement offered by modern hardware architectures and, then, proposes how data management systems must alter in order to realize this potential. Additionally, this thesis demonstrates that fully utilizing modern hardware architectures is important both for performance and energy-efficiency. Towards this goal, we propose query processing and indexing techniques for chip multiprocessors and we analyze the trade-offs of executing complex database queries on modern processor technologies. Subsequently, we propose query processing methods tailored to flash-based storage systems. Finally, we analyze the power consumption of database systems and we reveal opportunities for improving their energy efficiency.
Acknowledgements

This thesis would not have been possible without the support of a number of people.

First of all, I would like to thank my thesis advisor Prof. Nick Koudas for his endless support during the past five years. Prof. Koudas is one of the smartest, hardworking and determined persons I have ever met and has been both a mentor and a friend. He taught me how to identify and solve research problems and supported me financially all these years. But more importantly, he set an example of how to set goals, how to work hard to achieve these goals and how to always remain focused despite any difficulties that may appear on the way. I consider these to be the most valuable lessons that I have learned as a PhD student at the University of Toronto and I owe my sincerest gratitude to my advisor for teaching me these lessons.

I would like to thank my wife Amalia for her endless love, patience, and support all these years. Without her smile, love and understanding I could not have completed this thesis.

I owe my deepest gratitude to Dr. Stavros Harizopoulos. He has been a mentor, a collaborator and a good friend for many years. My collaboration with Stavros has always been one of the most rewarding, useful and enjoyable experiences I have ever had and I honestly thank him for that.

I would also like to thank Prof. Alex Delis and Prof. Yannis Ioannidis. They both believed in me and helped me start this journey. Without their help, none of this would have happened.

I would like to thank my office mate and good friend Nikos Sarkas. Nikos has always been a true friend and one of the few people I admire and respect for his character.

There not enough words to express my gratitude to my parents and my brother for their love and support. I thank them all very much.

Last but not least, I would like to thank all the people I met and worked with during the past five years, especially Dr. Mehul Shah and Prof. Sudipto Guha.
# Contents

1 Introduction .................................................. 1
   1.1 Thesis Outline ........................................... 2

2 Architecture-concious Data Management ................. 5
   2.1 Introduction ............................................. 5
   2.2 Improving Cache Performance ........................... 5
      2.2.1 Data Placement ..................................... 6
      2.2.2 Access Methods ..................................... 6
      2.2.3 Query Processing .................................... 8
   2.3 Leveraging On-Chip Parallelism .......................... 10
      2.3.1 Query Processing on SMT and CMP Architectures ... 10

3 List Intersection Algorithms on Chip Multiprocessors .. 13
   3.1 Introduction ............................................. 13
   3.2 Overview of List Intersection Algorithms ............... 15
   3.3 Load Balancing on a CMP ............................... 16
      3.3.1 List Partitioning on a Single Processor ........... 17
      3.3.2 List Partitioning on CMPs .......................... 19
   3.4 List Intersection Algorithms ........................... 21
      3.4.1 The Case of Sorted Lists ............................ 21
      3.4.2 The Case of Unsorted Lists ......................... 28
   3.5 Experimental Evaluation ............................... 32
3.5.1 Methodology ......................................................... 32
3.5.2 Experimental Results ............................................ 33
3.6 Conclusions ............................................................. 44

4 Query Processing on Chip Multiprocessors: Analyzing the Trade-offs 46
4.1 Introduction ............................................................ 46
4.2 Chip Multiprocessor Architectures .................................. 47
4.3 CMPDB: A Database Query Engine for CMPs ..................... 48
  4.3.1 General Architecture ........................................... 48
  4.3.2 Query Execution Engine ........................................ 51
4.4 Analysis ................................................................. 58
  4.4.1 Experimental Setup ............................................. 59
  4.4.2 Results ........................................................... 60
4.5 Conclusions ............................................................. 75

5 Suffix Tree Construction Algorithms on Chip Multiprocessors 76
5.1 Introduction ............................................................ 76
5.2 Survey of Suffix Tree Construction Algorithms .................... 77
  5.2.1 In-memory Algorithms ........................................... 77
  5.2.2 Disk-based Algorithms .......................................... 79
  5.2.3 Parallel Algorithms .............................................. 80
5.3 Background ............................................................ 80
  5.3.1 Suffix Trees ..................................................... 80
  5.3.2 Ukkonen’s Algorithm ........................................... 81
5.4 Cache-Partitioning .................................................... 82
  5.4.1 PreCache Partitioning Algorithm ................................ 83
5.5 Suffix Tree Construction Algorithms for CMP Architectures ......... 87
  5.5.1 Exploiting on-chip Parallelism .................................. 87
  5.5.2 CMPUTree Algorithm ........................................... 88
  5.5.3 MAPST Algorithm ............................................... 89
5.6 Experimental Evaluation ................................................. 94
  5.6.1 Experimental Setting ................................................. 94
  5.6.2 Improving Cache Performance ...................................... 95
  5.6.3 Exploiting on-chip Parallelism ..................................... 101
  5.7 Conclusions ............................................................... 102

6 Query Processing Techniques for Solid State Drives .................. 106
  6.1 Introduction .............................................................. 106
  6.2 SSD Trends and Related Work ......................................... 107
    6.2.1 SSD Characteristics, Costs, and Trends .......................... 107
    6.2.2 Databases and SSDs ............................................... 108
    6.2.3 Related Query Processing Techniques ............................ 110
  6.3 Scan Selections & Projections ........................................ 112
    6.3.1 FlashScan Operator ................................................ 112
    6.3.2 Scan Experiments .................................................. 115
    6.3.3 FlashScan and Column Stores ..................................... 118
  6.4 FlashJoin ................................................................. 119
    6.4.1 FlashJoin Overview ................................................ 119
    6.4.2 Join Kernel .......................................................... 121
    6.4.3 Materialization Strategy .......................................... 121
    6.4.4 Fetch Kernel ........................................................ 122
  6.5 Experimental Evaluation ............................................... 125
    6.5.1 Experimental Setting .............................................. 125
    6.5.2 Two-way Join Results ............................................. 126
    6.5.3 Multi-way Joins .................................................... 130
  6.6 Conclusions .............................................................. 136

7 Analyzing the Energy Efficiency of a Database Server ............... 137
  7.1 Introduction ............................................................ 137
    7.1.1 Where does power go? ............................................ 138
7.1.2 Power use and resource utilization ........................................ 139
7.1.3 Affecting energy efficiency: what “knobs”? ............................ 139
7.1.4 Energy efficiency vs. performance ...................................... 141
7.1.5 Analysis of results and implications .................................. 142
7.2 Background and motivation .................................................. 143
7.3 Where Does Power Go? ...................................................... 145
  7.3.1 Disks ................................................................. 147
  7.3.2 CPUs ............................................................... 148
7.4 Energy vs Performance ...................................................... 155
  7.4.1 Algorithm/Plan Selection ............................................ 155
  7.4.2 Intra-operator Parallelism ............................................ 159
  7.4.3 Inter-Query Parallelism .............................................. 161
7.5 Analysis and implications .................................................. 163
  7.5.1 The balance of power and performance ............................ 164
  7.5.2 Implications for database computing .............................. 166
7.6 Conclusions ................................................................. 168

8 Conclusions ................................................................. 170
  8.1 Extensions ................................................................. 171
    8.1.1 Designing a Database Query Engine for Chip Multiprocessors 171
    8.1.2 Database Systems on Flash-based Storage Systems ............. 171

Bibliography ................................................................. 173
List of Tables

3.1 Range of parameter values in our experimental design. .......................... 33
4.1 Characteristics of the dual socket server used in our study. ...................... 59
4.2 Relations ................................................................. 60
4.3 Speedup of two-way join .................................................. 64
4.4 Hardware performance counters ................................................. 65
5.1 Characteristics of Text Collections ................................................. 95
5.2 Memory Usage of CMPUTree .................................................. 96
5.3 Memory Usage of MAPST .................................................... 99
6.1 Disk and Flash characteristics from manufacturer specs or as measured where possible. Prices from online retailers as of Nov 25, 2008. SATA-disk: Seagate Barracuda; SATA-Flash: Mtron; FC-Flash: STech’s ZeusIOps 3.5” FibreChannel; ioD: FusionIO ioDrive. 109
6.2 Relations participating in N-way STAR joins. ...................................... 134
7.1 Configuration of our database server. .............................................. 144
List of Figures

3.1 Example of Dynamic Probes algorithm. ........................................... 24
3.2 Different probing policies in two segments with non-uniform distributions of document identifiers. ......................................................... 27
3.3 Example of Hash-based algorithm for two unsorted lists on a CMP with two cores. . . 30
3.4 Performance of intersection algorithms for four sorted lists with synthetic data using four cores. .......................................................... 36
3.5 Performance of intersection algorithms for eight sorted lists with synthetic data using four cores. ......................................................... 36
3.6 Performance of intersection algorithms for sixteen sorted lists with synthetic data using four cores. ..................................................... 37
3.7 Performance of intersection algorithms for sixteen sorted lists with synthetic data using eight cores. ..................................................... 37
3.8 Performance of intersection algorithms for four sorted lists with synthetic data using four cores when each list contains 100 million elements. . . . 38
3.9 Performance of intersection algorithms for sixteen sorted lists with anti-correlated data using eight cores. ......................................... 39
3.10 Performance of intersection algorithms for different number of sorted lists with real data using four cores. ........................................ 40
3.11 Performance of the DP algorithm as we increase the number of cores. ................ 41
3.12 Performance of the QB algorithm as we increase the number of cores. ................ 41
3.13 Impact of threshold $T_{QB}$ on the performance of QB algorithm on real data. ....... 42
4.19 Throughput of CMPDB as we vary the number of clients in the system. ....... 74
4.20 Throughput of CMPDB engine as we execute multiple three-way join queries in parallel using different intra-query and inter-query configurations. ............... 75

5.1 Execution time breakdown of two suffix tree construction algorithms. .......... 78
5.2 Suffix tree of string BANANA$. ................................................. 81
5.3 Example of variable-length prefix-based partitioning. .......................... 85
5.4 LZW-trie ................................................................. 93
5.5 Evaluation of CMPUTree for biological text data. ............................ 97
5.6 Evaluation of CMPUTree for English text data. ................................ 97
5.7 Execution time breakdown of CMPUTree_E and CMPUTree_PC algorithms when the input string size is 80MBs. ................................................ 98
5.8 Evaluation of MAPST for biological text data. ............................... 99
5.9 Evaluation of MAPST for English text data. .................................... 100
5.10 Effect of prefix size on execution time of MAPST for biological data. ...... 100
5.11 Execution time breakdown of suffix tree construction algorithms. .......... 101
5.12 Speedup of CMPUTree as we increase the number of cores utilized. ....... 102
5.13 Speedup of MAPST as we increase the number of cores utilized. .......... 103

6.1 Disk pages in NSM (left) and PAX (right) storage layout of a relation with four attributes. .......................................................... 113
6.2 FlashScan avoids reading attributes that are not projected; FlashScanOpt additionally avoids reads of minipages without selected tuples. When run over sorted predicate attributes, FlashScanOpt(S) is able to skip many minipages of projected attributes. ... 115
6.3 Scans on SSDs: FlashScan is much faster than a traditional scan when either few attributes (small projectivity) or few rows (small selectivity – SEL) are selected. Data analysis queries contain many scans on large relations that project few attributes and select few rows. .................. 116
6.4 When more attributes are projected, FlashScan must read nearly as much data as NSM-Scan. The contrast between FlashScan and FlashScanOpt is therefore much greater (see Figure 6.2) when not all pages contain selected tuples. .......................... 117
6.5 Execution strategy for a three-way join when FlashJoin is employed. ................. 120
6.6 Performance comparison of join algorithms for two-way joins when the join is computed in one pass. ......................................................... 127
6.7 Performance comparison of join algorithms for two-pass two-way joins. ............ 128
6.8 Performance comparison of two-way join algorithms as a function of memory size. 129
6.9 Performance comparison of join algorithms during three-way joins as a function of join result cardinality, when projectivity is 25%. ................................. 131
6.10 Performance comparison of join algorithms during three-way joins as a function of join result cardinality, when projectivity is 75%. ................................. 132
6.11 Performance comparison of join algorithms during three-way joins, as a function of memory size allocated per join. ............................................. 133
6.12 Multi-way joins on SSDs: FlashJoin is at least 2x faster than hybrid hash join over either traditional row-based (HNSM) or PAX layouts. ......................... 134
6.13 Performance comparison of join algorithms during the execution of TPC-H queries. 135
7.1 Power breakdown of test machine. .................................................... 138
7.2 CPU power vs. utilization for database operators. .................................... 140
7.3 Energy efficiency vs. performance in System-X. .................................... 141
7.4 Scan results (including compression). .................................................. 143
7.5 Power consumption of SSDs and HDDs as a function of device utilization. .......... 148
7.6 Different process scheduling policies. .................................................. 150
7.7 Memory bus utilization of different operators. ....................................... 152
7.8 Hashjoin operators. ................................................................. 153
7.9 Sort operators. ................................................................. 153
7.10 Scan Operators. ................................................................. 154
7.11 Performance and energy-efficiency of access methods in PostgreSQL. ........... 156
7.12 Performance and energy-efficiency of join algorithms in PostgreSQL ........................................ 157
7.13 Performance and energy efficiency of different query plans for executing TPC-H Q5 on System-X ............................................................................................................................................. 158
7.14 Energy efficiency vs. performance for parallel in-memory hash join, as the number of CPU cores used varies from 1 to 8 ..................................................................................................................................... 159
7.15 Energy efficiency of parallel in-memory sorting ............................................................................. 160
7.16 Parallel scan queries in PostgreSQL ............................................................................................... 162
7.17 Performance and energy-efficiency of parallel TPC-H Q12 queries in System-X ......................... 163
7.18 Recalculating energy efficiencies in two hypothetical scenarios .................................................... 164
7.19 Performance of different configurations of Q5 when power target is set below 200W .......... 169
List of Algorithms

1. Partition Unsorted Algorithm ......................................................... 22
2. Dynamic Probes Algorithm ........................................................... 26
3. Quantile-based Algorithm .............................................................. 28
4. Hash-based Algorithm ................................................................. 31
5. Probe Phase ................................................................................... 58
6. Ukkonen’s Algorithm ..................................................................... 82
7. PreCache - Partition Set Computation ............................................ 104
8. MAPST Algorithm ......................................................................... 105
9. Algorithm for computing the lcp .................................................... 105
10. Fetch Kernel to produce the join result with multiple passes. .............. 123
Chapter 1

Introduction

The increasing need for performance improvement as well as physical limitations have revolutionized computer hardware architectures. All the major components of systems, namely processor, memory and disks, have undergone radical changes: a) physical limitations have shifted the design of processors towards chip-multiprocessor (multi-core) architectures, b) the advent of 64-bit architectures has removed the 4GB memory-size barrier, c) the proliferation of flash technology has resulted in the progressive replacement of magnetic disks with solid-state drives (SSDs) that promise to deliver significant performance improvements in enterprise systems.

These technologies exhibit fundamentally different performance and power characteristics compared to their predecessors, and for that reason, they have not been adopted by data management systems to large extent. Along these lines, the goal of this thesis is two-fold. First, it demonstrates that for a large class of data processing tasks, hardware-aware techniques leverage the full potential for performance improvement offered by modern processor and storage technologies. Second, it demonstrates that hardware-conscious techniques that fully utilize hardware resources, do not only accelerate data processing tasks but also improve the energy efficiency of data management systems.

Towards these goals, this thesis studies the interaction between data management and modern processor technologies, such as chip-multiprocessors (CMP), and proposes novel data processing techniques optimized for the characteristics of CMP architectures. Under the assumption that all data fit in main memory, the proposed techniques are designed to leverage on-chip parallelism and improve cache-performance in order to reduce memory access cost. In this context, this thesis initially proposes
CMP-aware algorithms for important data processing tasks such as text and relational query processing. Furthermore, it considers the design of indexing techniques, such as suffix tree construction algorithms, optimized for modern processor technologies.

Although increasing memory sizes have allowed for more operations to be performed entirely in memory, the performance of large scale database systems and data warehouses often depends on the performance of the underlying storage system. As a result, data processing techniques in database systems have always been optimized for the performance characteristics of magnetic disks, which has been the prevalent storage technology for more than thirty years. This thesis proposes new query processing techniques that are tailored to the performance characteristics of solid-state drives. In particular, it studies novel algorithms that leverage the fast random I/O offered in flash technology to accelerate core query processing operations such as scans, projections and joins.

Finally, this thesis investigates the interplay of data management software and energy efficiency. In particular, it analyzes the role of database software in affecting and, ultimately, improving the energy efficiency of database systems. In particular, this thesis demonstrates that maximizing the utilization of hardware resources is key to improving the energy efficiency of data management systems, such as databases.

1.1 Thesis Outline

This thesis is organized as follows:

Chapter 2 surveys architecture-conscious data processing techniques. Initially, approaches that improve temporal and spatial locality to reduce memory access cost are presented. Subsequently, techniques that exploit on-chip parallelism are considered.

Chapter 3 presents in-memory list intersection algorithms tailored to chip-multiprocessors. Initially, Dynamic Probes is presented, a list intersection algorithm that leverages multi-level cache hierarchies to reduce the overhead of random memory accesses. The problem of load balancing during the parallelization of list intersection algorithms is studied and an effective and efficient load balancing algorithm based on approximate quantiles is proposed. Quantile-based is presented next, a list intersection algorithm that exploits the free inspection of lists, when a load balancing algorithm is employed, to reduce
the cost of intersection. For the case of unsorted lists, *Hash-based* is introduced, a list intersection algorithm for chip multiprocessors that avoids the overhead of sorting. Through a detailed experimental evaluation using synthetic and real data, it is demonstrated that all algorithms achieve linear speedup. For sorted lists, the proposed techniques exhibit the best performance compared to existing list intersection algorithms. When the lists are unsorted, the Hash-based algorithm performs better than the combination of sorting and any list intersection algorithm for sorted lists.

Chapter 4 analyzes the performance trade-offs of executing complex database queries on modern processor technologies. Initially, *CMPDB* is presented, a database query processing engine for main memory relational data that leverages inter-query and intra-query parallelism to improve the performance of complex database queries on chip multi-processors. A detailed performance analysis is presented that investigates: a) the potential for performance improvement offered by CMPs, b) limiting performance bottlenecks, and c) the performance impact of different hardware configurations.

Chapter 5 presents in-memory suffix-tree construction algorithms tailored to chip-multiprocessors. Initially, the problem of cache-partitioning is studied in the context of suffix trees and a low overhead partitioning algorithm, termed *PreCache*, is proposed. Next, two suffix tree construction algorithms are introduced, termed *CMPUTree* and *MAPST* (MAterialized Prefixes Suffix Tree), that exhibit different time-space trade-offs. CMPUTree is a linear time algorithm with large memory requirements. In contrast, MAPST exhibits a smaller memory footprint at a theoretical worse time complexity. Both algorithms employ PreCache to leverage on-chip parallelism and improve cache performance. Through an extensive experimental evaluation using real text data it is demonstrated that the proposed techniques significantly reduce the memory access cost and achieve satisfactory speedup on CMP architectures.

Chapter 6 presents query processing techniques for solid-state drives. Initially, *FlashSCan* is introduced, a scan operator that leverages a columnar layout to accelerate database scan projections and selections on solid-state drives. Subsequently, *FlashJoin* is presented, a general pipelined join algorithm that minimizes memory requirements and I/Os needed by each join in a query plan. FlashJoin combines in a novel way three well-known ideas that in the past were only evaluated for hard drives: using a column-based layout when possible, creating a temporary join index, and using late materialization to retrieve the non-join attributes. The evaluation of the proposed techniques is presented through a prototype implementation inside a real database system (PostgreSQL). Our study demonstrates that
the performance of enterprise systems can be considerably improved by leveraging the performance characteristics of solid-state drives.

Chapter 7 presents a detailed analysis of the energy-efficiency of database systems. Initially, the power-performance profiles of database operators under different configuration parameters are characterized to increase our understanding of how database operators affect power consumption and to reveal opportunities for improving energy efficiency. Then, a detailed experimentation with several classes of database systems and storage managers is presented that investigates how different software and hardware configuration parameters affect the energy efficiency of database systems. Our study reveals that for a single node database system, the most energy efficient configuration is typically the highest performing one. Furthermore, it discusses opportunities for improving the energy-efficiency of database systems in multi-node configurations.

Conclusions and extensions of our work are presented in Chapter 8.
Chapter 2

Architecture-conscsious Data Management

2.1 Introduction

This chapter presents architecture-conscious data management techniques, primarily in the context of database systems. All the techniques described herein leverage the characteristics of the underlying hardware architecture to improve the performance of data processing tasks. Two important categories are considered: a) cache-conscious techniques that improve temporal and spatial locality to reduce memory access cost, and b) techniques that exploit on-chip parallelism.

2.2 Improving Cache Performance

Memory access latency has not improved at the same rate as processor speed. Hence, in terms of CPU cycles, memory access cost has increased dramatically over the years. To alleviate this cost, fast and small caches have been introduced that rely on applications’ spatial and temporal locality to store the most frequently accessed data. However, data processing techniques either have large working sets that exceed the size of CPU caches or they exhibit irregular memory access patterns. Hence, multi-level cache hierarchies are severely under-utilized [4].

A large collection of techniques has been proposed to overcome this limitation. Without loss of generality, these techniques can be classified into three distinct categories: a) data placement, b) access methods, and c) query processing.
2.2.1 Data Placement

Data placement determines how data are stored and processed. Traditionally, database systems have utilized the N-ary Storage Model (NSM) in which records are stored contiguously in pages. NSM has been shown to be ineffective for low projectivity (number of projected attributes) and low selectivity queries, as disk and memory bandwidth is wasted and memory hierarchy is “polluted” with unnecessary data (attributes). In an effort to improve spatial locality and improve bandwidth utilization, an alternative storage model, termed DSM or column-store, was proposed [26]. DSM uses vertically partitioned relations, i.e. every attribute is stored in a different relation. Low projectivity queries only transfer the attributes needed, thereby increasing disk and memory bandwidth utilization. However, when a query accesses many attributes, materialization cost is pronounced and DSM performs worse than NSM [50].

Ailamaki et al. proposed a new page layout strategy, termed Partition Attributes Across (PAX), to combine the advantages of NSM and DSM [3]. In PAX, every disk page stores the same records as in NSM. Nevertheless, within a page the attributes are organized as in DSM. PAX partitions the records vertically within each page, storing the values of each attribute in contiguous locations, termed minipages. PAX does not affect the I/O cost, as the same number of pages is required to store a relation as in NSM. Furthermore, materialization cost is negligible because all the attributes of a record are stored in the same page. PAX performs better than NSM and DSM in sequential file-scan operations. However, for other common operations such as index scans, PAX exhibits poor cache performance compared to NSM as frequently accessed attributes reside in different cache lines.

Hankins and Patel proposed a generalization of PAX, called Data Morphing (DM) [45]. DM groups attributes based on their frequency of co-occurrence in a query workload, thereby achieving good cache performance for different types of queries.

2.2.2 Access Methods

Indexing data structures are utilized to accelerate a variety of operations in data management systems. Existing indexing techniques, such as B+-Trees and hash tables, were initially optimized for I/O performance. Hence, they exhibit poor temporal and spatial locality when applied to memory resident data. The performance of various indexing techniques on modern computer architectures has been studied.
extensively in the past [21, 22, 87, 88, 107].

Rao and Ross compared the lookup time and space requirements of various main memory indexing techniques [87]. Hash indexes, binary search trees, T-Trees, B+-Trees, interpolation search, and binary search were considered in their study. Their results suggest that hash indexes are the most efficient in terms of lookup time but with the largest space overhead. Cache-optimized B+-Trees and T-Trees are the best tree-based indexing techniques. Interpolation search and binary search are the most space-efficient indexing techniques. However, they exhibit poor temporal locality, as the number of cache misses during lookups is in the order of comparisons performed. A new indexing technique, termed CSS-Trees, was proposed for read-mostly data. CSS-Trees achieve the best tradeoff between space and lookup time as data entries are stored in contiguous memory locations and tree traversal is performed through offset computations.

_Cache-sensitive B+-Trees (CSB+-Trees)_ was proposed by Rao and Ross to improve the performance of CSS-Trees during incremental updates [88]. Each node maintains only one pointer to its leftmost child node to improve spatial locality during lookups and space utilization. Tree traversal is performed through offset computations using the address of the first child node. CSB+-Trees always store the sibling nodes in contiguous memory locations during updates, thus increasing memory allocation and copy overhead during update-intensive workloads.

All the tree-based indexing techniques described above set every tree node to be equal to a transfer unit from memory (cache line). However, smaller nodes increase the height of the tree, and consequently the number of cache misses during tree traversals. Chen et al. proposed the use of software prefetching to allow for wider nodes in a B+-Tree, and improving its cache performance during lookups [21]. A jump array with leaf nodes addresses was utilized to improve the cache performance of range scans in B+-Trees. Using a simulation based experimental evaluation, it was demonstrated that the combination of prefetching and wider tree nodes achieves a speedup of $1.27X$ to $1.55X$ over B+-Trees for index search. For index-scans, the utilization of a jump pointer array achieves a speedup of $6.5X$ to $8.7X$ over B+-Trees.

Chen et al. proposed a variation of B+-Trees, termed _fractal prefetching B+-Trees (fpB+-Trees)_ that exhibit good cache and I/O performance at the same time [22]. Two techniques for implementing fpB+-Trees are proposed: (i) _disk-first_ and (ii) _cache-first_. Disk-first starts with an I/O-optimized B+-
Tree, organizing the keys and the pointers of each page as a cache-conscious tree. On the contrary, cache-first approach begins with a cache-optimized B+-Tree and then groups tree nodes in pages so that a node and most of its child nodes reside on the same page. When there is sufficient memory to hold most of the index pages, the cache-first approach results in the best performance. However, when this is not the case, the disk-first approach is recommended for its minimal impact on I/O performance.

Zhou and Ross proposed the use of buffering to improve spatial and temporal cache locality between consecutive lookups and increase the rate at which lookups are processed in a tree-like indexing structure [107]. Buffers are created at non-root nodes in the tree to store lookup entries. Periodically, these buffers are emptied and the cost of accessing a node is amortized among several lookups. Their experimental evaluation demonstrated that buffering can improve index lookup throughput by a factor of two over cache-conscious approaches such as CSB+-Trees and at the same time retain a reasonable response time when updates occur along with lookups.

### 2.2.3 Query Processing

Traditionally, query processing techniques were designed to operate on disk-resident data and they were not optimized for cache performance. In particular, it has been demonstrated experimentally that for memory resident data, L2 data cache misses, L1 instruction cache misses, and DTLB misses are the main performance bottlenecks. Hence, there has been a continuous effort to alleviate the overhead of these stalls and improve the performance of query processing algorithms in modern architectures [14, 19, 49, 72, 95, 108].

Shatdal et al. studied the cache performance of several query processing operators [95]. A number of optimization techniques were proposed to improve cache locality: a) blocking, b) partitioning, c) extracting relevant data, and d) loop fusion. Blocking alters the access patterns of an algorithm to improve temporal locality. Partitioning divides a task into sub-tasks such that the working set of each sub-task fits in cache. Extracting relevant data technique reduces the amount of data accessed during query processing operations, thereby improving cache utilization and reducing the demand for memory bandwidth. It was demonstrated that, utilizing these techniques, algorithms such as hash joins, sorting and hash-based aggregation run 8% – 200% faster than traditional (unoptimized) versions.

Boncz et al. investigated the effect of memory access cost on common database operations such
as selections, aggregations and equi-joins [14]. Vertically fragmented data structures and a radix-based cache-partitioning algorithm were employed to improve the cache performance of equi-joins. An analytical model was also proposed to quantify the cost of relational queries in terms of CPU cycles, translation look-aside buffer (TLB) misses, and cache misses.

Manegold et al. studied the performance of pre-projection and post-projection strategies for both NSM and DSM storage schemes, in conjunction with a cache-conscious hash join algorithm [71]. The post-projection strategy executes the join first and deals with the projection columns afterwards. Their experimental evaluation suggests that post-projection strategy is best suited for DSM storage scheme, while for NSM the pre-projection technique exhibits the best performance.

Chen et al. proposed the use of software prefetching to improve the performance of hash joins [19]. Two software prefetching techniques are employed: a) group prefetching and b) software-pipelined prefetching. Both techniques leverage inter-tuple parallelism to overlap cache misses with useful computation. Through a simulation-based experimental evaluation, it was demonstrated that the proposed techniques achieve up to 1.6X speedup over GRACE hash join algorithm and simple prefetching approaches.

Harizopoulos and Ailamaki proposed a methodology to improve instruction cache performance of database servers executing transactional workloads [49]. The methodology, termed STEPS, operates in two levels to increase temporal locality of instructions. At a higher level, synchronization barriers are utilized to form groups of threads executing the same system component. Within each group, fast context-switches are employed to enable the reuse of cache-resident instruction across team members. Furthermore, an automatic profiling tool, termed autoSTEPS, was designed to detect points in the code where context-switches should occur. Using the TPC-C benchmark, it was demonstrated that STEPS eliminates two thirds of instruction-cache misses and achieves up to 1.4X overall speedup.

Zhou and Ross utilized buffering to improve instruction cache performance of database operations [108]. A buffer operator is placed in a query plan between a parent and a child operator to store pointers to intermediate results. Child and parent operators are executed in a non-interleaved fashion to increase utilization of instruction cache. A plan refinement algorithm was introduced to decide the placement and size of buffers in a query execution plan. Query performance of decision support workloads was improved by up to 15% over a memory resident database.
Chen et al. proposed the Inspector join algorithm for shared memory (SMP) architectures [20]. Bloom filters are utilized in the I/O partition phase to generate cache-sized sub-partitions of the build relation. Cache-resident hash tables are created from these sub-partitions to reduce memory access cost during build and probe phases. Inspector join is highly efficient for primary-foreign key joins and when the size of each build partition is relatively small compared to the size of the cache; however, its performance degrades significantly when this is not the case.

2.3 Leveraging On-Chip Parallelism

Hardware architectures that support multiple threads of control have emerged to address physical limitations and limited instruction level parallelism (ILP). Although, parallelism offers a large potential for performance improvement, it requires a complete redesign of software. A set of query processing techniques tailored to multi-processor architectures are presented next. In particular, we focus on techniques designed for simultaneous multithreading (SMT) and chip-multiprocessor (CMP) architectures.

2.3.1 Query Processing on SMT and CMP Architectures

Zhou et al. studied how simultaneous multithreading architectures can be leveraged to improve the performance of database operations on memory-resident data [106]. Intra-operator and inter-operator parallelism were studied and the notion of a helper thread was introduced to improve cache performance of database operations. Helper thread performs aggressive data preloading in cache so that the main execution thread does not experience memory access latencies. A 30% to 70% performance improvement was demonstrated over single-threaded implementations. However, no clear advantage was demonstrated of the helper thread over multi-threaded implementations.

Garcia and Korth examined the utilization of software prefetching on simultaneous multithreading architectures in the context of hash joins [34]. Their investigation revealed that although the benefits attained from software prefetching have been overestimated in previous simulation-based studies ([19]), the combination of simultaneous multithreading and software prefetching can significantly improve the performance of hash joins.

The performance of multi-way join queries on multithreaded architectures was studied by Garcia and
Korth [35]. The benefits of Inter-operator and intra-operator parallelism were investigated and a buffer management strategy for inter-operator communication was introduced. However, this study considered the execution of "small" pipelines that consisted of only two join operators. They demonstrated a performance improvement of $1.5X$ over single threaded architectures using a duo-core processor on uniform data.

Gedik et al. studied the problem of executing stream joins on the Cell processor [36]. A lightweight dynamic window partitioning technique was proposed to distribute the load evenly among cores in the presence of variable input stream rates as well as load variations. The join windows are organized in memory in a column-wise fashion for improved spatial locality. Double-buffering techniques were utilized at a window level to mask transfer delays between cores. It was demonstrated experimentally that windowed stream joins achieve high scalability and surpass the performance attained by conventional high-end processors.

Cieslewicz and Ross proposed an adaptive hash-based aggregation strategy for chip multiprocessors [24]. The trade-offs with respect to the use of a shared versus multiple independent hash tables were explored and a new hybrid algorithm that combines both techniques was proposed. An adaptive framework was utilized to analyze the input data and dynamically decide the best strategy to apply. Furthermore, the performance of several concurrency control schemes was examined. Their experimental evaluation indicates that the adaptive hash-based aggregation operator achieves good performance irrespective of the input distribution.

Cieslewicz et al. proposed a parallel buffer to alleviate the communication overhead among operators on CMP architectures. It is utilized to avoid the contention between threads and the need for excessive synchronization which results in severe performance degradation [25].

The performance characterization of various database workloads on different CMP architectures was presented in [46]. It was demonstrated that stalls due to data cache misses are the limiting performance bottleneck when data processing operations are executed on CMPs.

Johnson et al. analyzed the tradeoffs between work sharing and parallelism when database systems run complex decision-support queries [59]. It was demonstrated that work sharing is not always beneficial when applied to chip multiprocessor architectures, because the execution of queries is serialized at the point of sharing. An analytical model was proposed to capture the trade-offs between parallelism
and work sharing and to predict when work sharing is helpful. The proposed model was integrated in a prototype execution engine, demonstrating that selective work sharing outperforms never-share static schemes by 20% on average and always-share ones by 2.5X.

Qiao et al. proposed a scheduling algorithm that implements shared scans on a main memory DBMS to improve memory bandwidth utilization on multi-core CPUs [84]. They demonstrated that naive application of scan sharing can lead to thrashing as the working set size of queries exceeds cache. A technique was proposed that estimates the working set size of each query and effectively groups queries in a way that avoids thrashing. The performance of BI queries was improved by a factor of 2.5X.
Chapter 3

List Intersection Algorithms on Chip Multiprocessors

3.1 Introduction

Set (list) intersection is a central operation in query processing and text analytics. However, existing algorithms for computing the intersection of sorted and unsorted lists are not optimized for modern hardware architectures. They exhibit poor temporal and spatial locality and they do not leverage the computational power of chip multiprocessors.

We present in-memory list intersection algorithms for sorted and unsorted lists, tailored to the characteristics of chip multiprocessors. Our algorithms achieve the following: a) they reduce the overhead of random memory accesses by leveraging the large on-chip cache hierarchy, and b) they leverage on-chip parallelism of CMP architectures.

To alleviate the memory access cost, we propose a list intersection algorithm for sorted lists, termed Dynamic Probes, that utilizes previous probes (searching for a particular element on a list) to construct a cache-resident micro-index of the lists. The micro-index is utilized to reduce the search range of each probe and consequently, the number of random memory accesses performed. The micro-index is continuously updated during the computation of the intersection, allowing the algorithm to adapt to the characteristics of each list by dynamically changing its probing order (order the lists are probed).

An important consideration when designing parallel algorithms is load balancing. Primarily, load
has to be distributed evenly among cores. If this is not the case, load disparity underutilizes the computational power available, thus reducing the speedup achieved. Furthermore, the overhead of load balancing should be kept to a minimum, otherwise it may offset the benefits of parallelization.

Exact algorithms for partitioning a number of lists ensure perfect load balancing. However, when a large number of lists is intersected, the overhead of such algorithms is high [57]. Approximate algorithms with explicit error guarantees offer attractive alternatives as they trade accuracy for speed. Load is not perfectly balanced but the overhead is lower than the overhead of exact algorithms. Even when approximate algorithms are utilized, the overhead of load balancing is non-negligible as these algorithms are not optimized for cache performance. Furthermore, for the case of unsorted lists, they deploy an expensive sorting phase.

We propose a CMP adaptation of an approximate quantile identification algorithm demonstrating how it can be utilized to efficiently partition an arbitrary number of sorted (Partition Sorted algorithm) and unsorted lists (Partition Unsorted algorithm) into any number of independent partitions. The quantile identification algorithm used has explicit error guarantees, allowing us to control the load disparity across cores.

Applying a quantile identification algorithm for load balancing purposes provides a unique opportunity to inspect list elements and identify lists with highly non-uniform distribution of elements. As Demaine et al. pointed out, this non-uniformity may result in severe performance degradation of list intersection algorithms [28]; they proposed adaptive intersection algorithms to address this issue. However, when a large number of lists is intersected, their approach introduces notable overhead because it tends to be “over-adaptive” [29, 86]. Instead, we propose a list intersection algorithm for sorted lists, termed Quantile-based, that takes advantage of quantiles, computed by the load balancing algorithm, to detect lists with highly non-uniform distribution of elements and determine a good probing order at low cost, thus eliminating the overhead of adaptivity.

For the case of unsorted lists, all known approaches deploy a sorting algorithm to sort the lists. Then, a list intersection algorithm for sorted lists is applied. Instead, we propose Hash-based, a list intersection algorithm that is tailored to CMPs. It utilizes the quantile identification algorithm to avoid sorting the lists and computes their intersection in a cache-conscious manner using hashing.

We present a detailed experimental evaluation using synthetic and real data on a chip multiprocessor
with eight cores. We demonstrate that all algorithms proposed herein scale almost linearly as the number of cores increases. For sorted lists, the proposed techniques exhibit the best performance compared to existing list intersection algorithms. When the lists are unsorted, the Hash-based algorithm performs better than the combination of sorting and any list intersection algorithm for sorted lists.

### 3.2 Overview of List Intersection Algorithms

The majority of algorithms proposed in the literature follow a similar pattern in the way they compute intersection. The elements of one list are used as *eliminators* and *probes* are conducted in the other lists using a search method (e.g. binary or interpolation search) to locate these elements. If an eliminator belongs to the intersection, the probes will locate it in all lists. If this is not the case, the eliminator is simply ignored. The algorithms proposed in the literature vary in how they select the eliminators and in the order in which they probe the lists (*probing order*).

The problem of intersecting two sorted lists has been studied in the past by Hwang and Lin [55, 54]. Demaine et al. proposed the *Adaptive* algorithm for computing the intersection of a collection of sorted sets [28]. Their algorithm computes the intersection by repeatedly cycling through the sets in a round-robin fashion. On a follow-up study they compared the *Adaptive* algorithm with existing algorithms using real data and demonstrated that in practice their algorithm does not always exhibit the best performance due to the overhead of adaptivity (repeatedly cycling through the sets) [29]. Based on these findings, a number of improvements were proposed, the most significant of which are: a) galloping search [13], and b) a limited form of adaptivity, compared to the initial version. The resulting algorithm, termed *Small Adaptive*, intersects the two smallest sets and it starts cycling through the remaining sets only when a match is found.

Barbay et al. repeated the study of [29] using a larger dataset [10]. In their experiments they included a list intersection algorithm proposed in [9]. Their study verified that *Small Adaptive* has the best performance in practice and that interpolation search improves the performance of all intersection algorithms.

An adaptive row id list intersection strategy for query processing has been proposed by Raman et al. [86]. In order to avoid the risk of using wrong cardinality estimations during the computation of
an AND-tree, they use an adaptive n-ary intersection algorithm based on the _Adaptive_ algorithm [28]. Instead of using a round-robin probing policy as suggested in [28], they propose a probabilistic probing policy to determine which list to examine next based on historical data from previous probes. In their experimental evaluation they compare their approach against the _Adaptive_ algorithm and against the traditional pipelined approach that utilizes the best AND-tree. As we will show in Section 3.5, the performance of _Small Adaptive_ [29] in many cases surpasses or is equivalent to the performance of their algorithm.

Iyer et al. proposed an exact percentile identification algorithm for an arbitrary number of sorted lists [57]. The overhead of the algorithm increases by a factor of $m \log m$, where $m$ is the number of sorted lists. Hence, it is not applicable to a large number of lists. Furthermore, in order to apply the algorithm in unsorted lists, a sorting algorithm must be utilized at first, introducing significant overhead.

Several algorithms for computing approximate quantiles of large datasets have been proposed [73, 43, 27]. Manku et al. presented single pass algorithms for computing approximate quantiles of large datasets with explicit approximation guarantees [73]. Greenwald and Khanna proposed a distributed single-pass algorithm to compute an $\epsilon$-approximate quantile summary over sensor data [43]. Cormode et al. proposed deterministic algorithms for biased quantiles, ranked queries and targeted quantile queries that have guaranteed space bounds [27].

### 3.3 Load Balancing on a CMP

A major issue when designing algorithms for CMP architectures is load balancing. Since load disparity among cores and the overhead of the load balancing technique limit the achievable speedup through parallelism, we seek low overhead load balancing techniques with explicit error guarantees. Techniques that achieve perfect load balancing usually introduce considerable overhead and in many cases the benefit derived from the elimination of the load disparity does not compensate for the overhead introduced [31]. We verify that effect in Section 3.5.

A quantile identification algorithm computes any order statistic over a set of elements\(^1\) $S$. Such an algorithm is used as a building block in our framework to divide (sorted and unsorted) lists into a

\(^1\)Element and document identifier (id) are used interchangeably in the text.
number of independent partitions of approximately equal size. Each partition is assigned to a core and in this way the computation of list intersection is parallelized in a load balanced way. For the case of sorted lists, list elements are accessed once while computing the intersection and the only requirement is load to be evenly partitioned among cores. Hence, the number of partitions is set to be equal to the number of cores utilized. For the case of unsorted lists, we require each partition to be cache-resident. Consequently, for a system utilizing \( C \) cores, we choose the number of partitions such that each partition fits in \( 1/C \)-th part of L2 cache.

Let us first discuss how a quantile identification algorithm can be utilized to partition \( k \geq 2 \) sorted lists into \( P \) independent partitions of approximately equal size on a uniprocessor architecture (Section 3.3.1). Next, we will demonstrate how these quantiles can be computed efficiently on CMPs (Section 3.3.2).

### 3.3.1 List Partitioning on a Single Processor

The independence property states that for any pair of partitions \( P_i, P_j (i < j, i, j = 1, \ldots, P) \), all the elements of \( P_i \) must be strictly smaller or larger than all the elements of \( P_j \). Consider for example two sorted lists \( A_1 = \{1, 2, 3, 5, 9, 10, 12, 15, 18, 20, 40\} \) and \( A_2 = \{4, 8, 11, 13, 14, 16, 17, 39, 41, 42, 50\} \). A possible partitioning of these two lists in two partitions that satisfies the independence property could be: \( P_1 = \{\{1, 2, 3, 5\}, \{4\}\} \) and \( P_2 = \{\{9, 10, 12, 15, 18, 20, 40\}, \{8, 11, 13, 14, 16, 17, 39, 41, 42, 50\}\}. In this partitioning, \( P_1 \) contains four elements of \( A_1 \) and one element of \( A_2 \); \( P_2 \) contains seven elements of \( A_1 \) and ten elements of \( A_2 \). Notice, that all the elements of \( P_1 \) are strictly smaller than the elements of \( P_2 \).

In order to identify the elements of each partition \( P_i \), it is sufficient to determine its boundary values \( lo(P_i), hi(P_i) \). The following invariant is true for boundary values: if element \( id \) belongs to \( P_i \) then \( lo(P_i) \leq id \leq hi(P_i) \). In the example described, the boundary values are: \( lo(P_1) = 1, hi(P_1) = 5 \), \( lo(P_2) = 8, hi(P_2) = 50 \). In general, a partition is a “set” of sublists - one from each of the original lists - where each of the sublists shares the same boundary values. The size of a partition is the sum of sizes of the sublists.

Note that although this is one possible partitioning of the two lists, it is not necessarily the best as \( P_1, P_2 \) are not of equal size. In order to divide an arbitrary number of sorted lists into \( P \) partitions...
of equal size, it is sufficient to compute \( P - 1 \) order statistics of the elements in the union of the lists; specifically, the elements with rank \( \frac{i}{P}, i = 1, \ldots, P - 1 \). The \( P - 1 \) order statistics along with the extreme values (min, max) of the lists determine the boundary values of the partitions. In our example, we need to compute the median in the union of \( A_1 \) and \( A_2 \) (value 13). Hence, the boundary values of the partitions in the optimal partitioning of \( A_1, A_2 \) are: \( \text{lo}(P_1) = 1, \text{hi}(P_1) = 13, \text{lo}(P_2) = 14, \text{hi}(P_2) = 50 \), i.e. \( P_1 = \{1, 2, 3, 5, 9, 10, 12\}, P_2 = \{15, 18, 20, 40\}, \{14, 16, 17, 39, 41, 42, 50\} \). The resulting partitions, \( P_1, P_2 \) have the same size (11 elements each).

Existing exact quantile identification algorithms can be used to compute order statistics and consequently to partition the lists in a load balanced way. However, these algorithms introduce notable overhead. Several approximate quantile algorithms with explicit error guarantees have been proposed in the literature [73, 43, 27]; any of these algorithms can be utilized in our setting. In this study we choose to elaborate utilizing the algorithm proposed by Greenwald and Khanna [43] since it has better space requirements compared to the algorithm in [73]. It is easy to see that if the size of the dataset \( N \) is known, the space requirements of the algorithm in [43] is \( O(\frac{1}{\epsilon} \log N) \) compared to the algorithm in [73] which requires \( O(\frac{1}{\epsilon} \log^2 N) \) memory, where \( \epsilon \) is the error with which any order statistic is computed. Furthermore, the algorithm in [43] is utilized because we want the simplest algorithm for deterministic unbiased quantiles; the algorithm in [27] is for biased quantiles.

An \( \epsilon \)-approximate quantile summary \( Q \) for a set of elements \( S \) is an ordered subset of \( S \) that can be utilized to answer any order statistic query over \( S \) with error \( \epsilon \). The resulting summary \( Q \) contains \( (1/2\epsilon + 1) \) elements. If we query \( Q \) for an element with rank \( r \) with error \( \epsilon \), then the output element is guaranteed to have rank within \( r \pm \epsilon |S| \). Next, we describe the computation of an \( \epsilon \)-approximate quantile summary for a set of elements residing in two sorted lists in a uniprocessor architecture using the algorithm in [43]. For brevity, we refer to an \( \epsilon \)-approximate quantile summary as \( \epsilon \)-summary.

We will describe the basic operations of the algorithm using a simple example. Assume that we wish to compute an \( \epsilon \)-summary \( Q \) for the elements residing in two sorted lists \( A_1, A_2 \) (same as in the previous example) with error \( \epsilon = 0.1 \). Initially, the algorithm computes two \( \epsilon \)-summaries \( Q_1, Q_2 \), for the elements in \( A_1 \) and \( A_2 \), respectively. An \( \epsilon \)-summary \( Q_i \) for the elements of \( A_i \) is computed by choosing at most \( \frac{1}{2\epsilon} + 1 \) elements. These are the elements with ranks \( 1, 2\epsilon |A_i|, 4\epsilon |A_i|, \ldots, |A_i| \), where \( |A_i| \) is the number of elements in \( A_i \). The \( \epsilon \)-summaries \( Q_1, Q_2 \) contain six elements each, \( Q_1 = \{1, \),
3, 9, 12, 18, 40} and \(Q_2 = \{4, 11, 14, 17, 41, 50\}\). In order to compute the final \(\epsilon\)-summary \(Q\), we apply the \(\text{combine}(Q_1, \ldots, Q_m)\) operation [43]. The \(\text{combine}\) operation produces a new summary \(Q\) by merging the smaller summaries \(Q_1, \ldots, Q_m\) provided as input. The error of \(Q\) is the maximum error of the input summaries in \(\text{combine}\). The size of \(Q\) is equal to the union size of summaries provided as input in \(\text{combine}\). In our example, we apply the \(\text{combine}\) operation using as input the \(\epsilon\)-summaries \(Q_1, Q_2\) obtaining \(Q = \{1, 3, 4, 9, 11, 12, 14, 17, 18, 41, 40, 50\}\) which is an \(\epsilon\)-summary that can be used to answer any order statistic over all the elements in \(A_1\) and \(A_2\) with error 10%.

3.3.2 List Partitioning on CMPs

Given the description of the approximate quantile algorithm for uniprocessors, we now propose an adaptation tailored to CMPs, resulting in \(\text{Partition Sorted}\) algorithm for sorted lists (presented in Section 3.3.2) and \(\text{Partition Unsorted}\) algorithm for unsorted lists (presented in Section 3.3.2). The adaptation aims to: a) enhance the algorithm so that it exploits all the available processing elements, and b) improve its cache locality, thus reducing the number of expensive memory accesses performed.

We consider a CMP architecture consisting of \(C\) cores and an on-chip cache hierarchy with a private L1 cache per core and a shared L2 cache which can store \(|L2|\) elements. Our algorithms are general enough to function on any type (organization) of the on-chip cache hierarchy. We refer to L2 as the lowest level of the on-chip cache hierarchy which is shared among cores.

\textbf{Partition Sorted Algorithm}

To parallelize the computation of an \(\epsilon\)-summary from \(k\) sorted lists, we assign the lists to cores (approx. \(\frac{k}{C}\) lists per core). Let \(k_i\) be the number of lists assigned to core \(i\). Note, that there is no physical transfer of lists because they reside in memory which is shared among cores. Core \(i\) computes \(k_i\) \(\epsilon\)-summaries \(Q_1, \ldots, Q_{k_i}\), one from each assigned list. Then, it merges the \(k_i\) \(\epsilon\)-summaries using the \(\text{combine}\) operation to produce a single \(\epsilon\)-summary \(Q^i\). A single core is responsible for merging the resulting \(\epsilon\)-summaries \(Q^1, \ldots, Q^C\) using the \(\text{combine}\) operation and produces the final \(\epsilon\)-summary \(Q\). The \(\epsilon\)-summaries are small enough to reside in cache and since cache is shared among cores, the intermediate \(\epsilon\)-summaries \(Q^1, \ldots, Q^C\) as well as the final \(\epsilon\)-summary \(Q\) are produced without the need for expensive off-chip memory accesses.
If \( k < C \), we “greedily” partition the lists in order of decreasing length. The partitioning requires \( C \) steps and at every step \( \min\{\# \text{ of remaining unassigned elements of current list}, \frac{N}{C} \} \) contiguous elements from the currently examined list are assigned to a core; \( N \) is the total number of elements in the lists. Each core produces an \( \epsilon \)-summary and the \( \epsilon \)-summaries are merged by a single core to produce the final \( \epsilon \)-summary \( Q \).

**Partition Unsorted Algorithm**

In contrast to the sorted case, when the lists are unsorted, it is not possible to create an \( \epsilon \)-summary by simply choosing a set of elements with specific rank. The naive solution is to sort the lists and then compute an \( \epsilon \)-summary utilizing the *Partition Sorted* algorithm (Section 3.3.2). We demonstrate that it is possible to avoid a possibly expensive sorting phase.

In Algorithm 1, we present pseudo-code for the *Partition Unsorted* algorithm. The main idea is to partially sort each list into a number of sorted *runs* and create an \( \epsilon \)-summary from each sorted run (lines 1-7). Once the \( \epsilon \)-summaries have been produced, they are merged into the final \( \epsilon \)-summary \( Q \) (lines 9-21).

Each run contains \( \frac{|L_2|}{C} \) elements from a list and it is sorted using Quicksort which is an in-place sorting algorithm. The size of each run is chosen so that \( C \) runs can fit simultaneously in \( L_2 \). Therefore, the generation of sorted runs does not cause any \( L_2 \) capacity cache misses. Only compulsory cache misses are incurred when the elements are fetched for the first time into the cache. An \( \epsilon \)-summary is produced from a sorted run immediately after its creation. The overhead of producing the \( \epsilon \)-summary is negligible as the run already resides in cache.

Once the initial \( \epsilon \)-summaries of all the runs have been produced, they are merged in parallel by the cores to produce the final \( \epsilon \)-summary \( Q \). Let \( M \) be the number of initial \( \epsilon \)-summaries. The \( M \) \( \epsilon \)-summaries are assigned to cores (approximately \( \frac{M}{C} \) \( \epsilon \)-summaries per core) and each core merges its assigned \( \epsilon \)-summaries using the *combine* operation; an \( \epsilon \)-summary \( Q^i \) is produced by core \( i \). The \( C \) resulting \( \epsilon \)-summaries \( Q^1, \ldots, Q^C \) are merged by a single core to produce \( Q \).

If all \( \epsilon \)-summaries can fit in \( L_2 \), the cores do not experience any \( L_2 \) cache misses while the \( \epsilon \)-summary \( Q \) is produced (lines 10-13). If this is not the case, we resort to a multi-phase merging (lines 15-21). In each phase the cores read a sufficient number of summaries fitting in \( L_2 \). When in cache, the
summaries are merged by the cores and a single summary is produced. Then, the size of the resulting summary is reduced using the \textit{prune} operation [43].

The \textit{prune} operation reduces the size of a quantile summary. It takes as input an \( \epsilon \)-summary \( Q \) for a set of elements \( S \) and a parameter \( B \) and produces a new \( \epsilon' \)-summary \( Q' \) of size at most \( B + 1 \) elements with error \( \epsilon' \) by querying \( Q \) for the elements of rank \( 1, |S|/B, 2|S|/B, \ldots, |S| \). The error of \( Q' \) is \( \epsilon' = \epsilon + \frac{1}{2B} \).

In the next phase, a sufficient number of pruned summaries fitting in L2 is read by the cores. The summaries are merged and a new summary is created which is again pruned. In the last phase all the summaries fit together in L2 and the final summary \( Q \) is computed (line 21).

Since the number of initial summaries is \( M \), the \textit{prune} operation will be applied at most \( \log M \) times, each time increasing the error of the resulting quantile summary by \( \frac{1}{2B} \). Hence, in order for the final quantile summary to have error \( \epsilon \), we must create the initial quantile summaries with error \( \frac{\epsilon}{2} \) (\( \frac{\epsilon}{2} \)-summaries) and set the value \( B = \frac{1}{\epsilon} \log M \). Consequently, if we apply the \textit{prune} operation \( \log M \) times, we will generate a quantile summary with error \( \epsilon \).

3.4 List Intersection Algorithms

In this section, we present several algorithms to compute the intersection of sorted and unsorted lists. Section 3.4.1 presents the \textit{Dynamic Probes} and \textit{Quantile-based} algorithms for sorted lists. Section 3.4.2 presents the \textit{Hash-based} algorithm for unsorted lists.

Assume we wish to compute the intersection of \( k \) lists \( A_1, \ldots, A_k \), where \( k \geq 2 \). Each list \( A_i \) contains \( |A_i| \) unique elements (ids); the total number of elements in the lists is \( N = \sum_{i=1}^{k} |A_i| \). We assume that the lists reside in main memory and the result of the intersection is also written in memory.

3.4.1 The Case of Sorted Lists

Dynamic Probes Algorithm

The basic idea explored by most of the algorithms proposed for intersecting \( k \) lists is to use the elements of one list as eliminators and conduct \textit{probes} (lookups) in the remaining lists using a search method (binary search, galloping search, etc). The \textit{probing order} (order in which the lists are probed) and the
Algorithm 1 Partition Unsorted Algorithm

**Input:** \( k \) unsorted lists \( A_1, \ldots, A_k \), \( |A_i| \) elements in list \( A_i \), \( |L2| \) elements in L2 cache, \( C \) cores, error \( \epsilon \)

**Output:** an \( \epsilon \)-approximate quantile summary \( Q \)

1: for \( l = 1, \ldots, k \) do
2:   while \( \exists \) unread elements in \( A_l \) do
3:     Read the next \( \frac{|L2|}{C} \) elements from \( A_l \).
4:     Sort the elements in-place using Quicksort and generate a sorted run \( r \).
5:     Compute an \( \epsilon \)-approximate summary for \( r \) containing at most \( \frac{1}{2\epsilon} + 1 \) elements.
6:   end while
7: end for
8: barrier // Synchronization point - There are \( M = \sum_{l=1}^{k} m_l \) quantile summaries, where \( m_l = \lceil \frac{|A_l| \cdot C}{|L2|} \rceil \) is the number of runs of list \( A_l \).
9: if \( M \cdot (\frac{1}{2\epsilon} + 1) < |L2| \) then
10:   The quantile summaries are assigned to cores, i.e. \( \frac{M}{C} \) quantile summaries are assigned to each core.
11:   Each core merges its assigned quantile summaries using the combine operation.
12:   barrier // Synchronization point - \( C \) quantile summaries have been created.
13:   The \( C \) quantile summaries are combined by a single core into the final quantile summary \( Q \).
14: else
15:   while summaries cannot fit in L2 cache do
16:     Each core reads \( \frac{M'}{C} \) summaries in L2 cache. // Let \( M' \) be the number of summaries that can fit in L2 cache.
17:     Each core merges its assigned summaries using combine operation.
18:     barrier // Synchronization point - \( C \) quantile summaries have been created.
19:     The \( C \) quantile summaries are combined by a single core into the quantile summary \( Q' \). prune is applied to \( Q' \) to reduce its size to at most \( B + 1 \) elements.
20:   end while
21: \( Q \) is computed. // Lines 10 - 13
22: end if

search interval length of each probe affects the performance of a list intersection algorithm.

For every probe, the number of steps (comparisons) required to either locate an element or exclude it from the intersection depends on the length of the search interval. Due to the random access pattern that most search methods exhibit, every step results in a random memory access that with high probability
cannot be resolved through cache. Furthermore, an effective probing order can speedup the computation of intersection. Probing the lists that are less likely to contain an element first, reduces the number of probes that is required to exclude an element from the intersection.

The Dynamic probes (DP) algorithm dynamically decides the probing order and the length of each probe using information from previous probes. This information is utilized as a cache-resident micro-index. Let us describe how the DP algorithm computes the intersection of \( k \) sorted lists \( A_1, \ldots, A_k \). The pseudo-code for the DP algorithm is presented in Algorithm 2. Initially, the lists are sorted by increasing length (\(|A_1| \leq |A_2| \leq \ldots \leq |A_k|\)) and the first element of the smallest list is set as eliminator, \( e = A_1[0] \). A binary search is performed for \( e \) in the remaining lists \( A_2, \ldots, A_k \) and two arrays \( Val, Pos \) are utilized to store the ids and the positions, respectively, of the elements that are “touched” during a binary search. For every list \( A_i \), the ids are stored in the \( i \)-th row of array \( Val \) and the positions are stored in the \( i \)-th row of array \( Pos \) (lines 3-5). \( Val \) and \( Pos \) store the ids and positions of only the elements with ids larger than \( e \). \( Val \) and \( Pos \) are two \((k - 1) \times \log n_{\text{max}}\) arrays, where \( n_{\text{max}} \) is the number of elements in the largest list. Next, we describe how the information stored in these two arrays is utilized as a micro-index to speedup following probes.

In the next step, the successor of \( e \) in the current smallest unexplored list is used as eliminator, \( e = \text{succ}(e) \) (line 6). Instead of using a round robin probing policy as the one applied in [29], we dynamically decide the probing order and the search interval length for \( e \) as follows. For every list \( A_i \), we search in \( Val \) for the ids \( \text{lo}_val^i(e), \text{hi}_val^i(e) \) that bound \( e \), i.e. \( \text{lo}_val^i(e) < e < \text{hi}_val^i(e) \) (line 9). The corresponding entries in \( Pos \) \( (\text{lo}_pos^i(e), \text{hi}_pos^i(e)) \) of \( \text{lo}_val^i(e) \) and \( \text{hi}_val^i(e) \), define a range of positions to search for \( e \) in list \( A_i \). \( e \) will either be found in range \([\text{lo}_pos^i(e), \text{hi}_pos^i(e)]\) or it does not exist in that list. Given their small size and the fact that they are frequently accessed, \( Val \) and \( Pos \) will reside with high probability in some level of the on-chip cache hierarchy. Consequently, computing the search intervals for \( e \) imposes minimal overhead. Once we determine for every list the search interval of \( e \), we sort the intervals by increasing length (line 15) and we greedily probe the lists in that order using interpolation search (line 20). The same procedure is applied repeatedly until one of the lists is exhausted.

In Figure 3.1, we illustrate a sample run of DP in three lists \( A_1, A_2, A_3 \) with 5, 100 and 150 elements (document identifiers), respectively. DP uses the first id (100) of list \( A_1 \) as eliminator and
performs a binary search in $A_2$ and $A_3$. A binary search for id 100 in $A_2$ locates id 100 at position 6 and “touches” the following position-id pairs: (50-750), (25-200), (12-150). Hence, the second row of $Val$ is populated with ids (150, 200, 750) and the second row in $Pos$ is populated with their positions (12, 25, 50). In accordance, after performing a binary search for id 100 in list $A_3$, the third row of $Val$ stores ids (350, 500, 700) and the third row of $Pos$ stores their positions (18, 37, 75).

Now, consider a search for the second id (400) of $A_1$. Any list intersection algorithm will consider as search interval the range (6, 100] of positions in $A_2$ and the range (9, 150] of positions in $A_3$. Moreover, if the probing order is determined using the remaining unexplored portion of each segment, then $A_2$ is probed first and $A_3$ second. Instead, the DP algorithm operates in a less conservative manner exploiting the entries in $Val$ and $Pos$ to refine the search interval for id 400. By comparing the id searched with the ids stored in $Val$, it sets the search interval to be the range [25, 50] of positions in $A_2$ and the range [18, 37] of positions in $A_3$. Furthermore, since the search interval in $A_3$ is smaller, $A_3$ is probed first and $A_2$ second.

The entries in arrays $Val$ and $Pos$ are updated in two cases using binary search: a) when the length of the search interval (number of elements) is larger than a threshold $T_{DP}$, and b) when for a given eliminator $e$ and a list $A_i$, it is not possible to identify a search interval in $A_i$ using the entries in $Val$ and $Pos$ (lines 10-13). The latter case occurs when the value of $e$ is larger than all the values in the $i$-th row of $Val$. In that case, the remainder unexplored portion of $A_i$ determines the search interval of $e$.

For example, consider searching for id 800 of $A_1$ in Figure 3.1. Id 800 is larger than all the ids
stored in $Val$ for both $A_2$ and $A_3$. Hence, we set $lo\_pos^i(e)$ to be the maximum between the position in $A_i$ where the search for the predecessor of $e$ halted and the last entry in the $i$-th row of $Pos$ (maximum position in $i$-th row). We set $hi\_pos^i(e)$ to be the length of list $A_i$, $|A_i|$. Thus, for id 800, the search interval is the range $[50, 100]$ of positions in $A_2$ and the range $[75, 150]$ of positions in $A_3$. When probing list $A_i$ for $e$, we perform a binary search and we update the entries in the $i$-th row of $Val$ and $Pos$ to benefit subsequent probes (lines 17-18); a binary search is performed in $A_2$ and $A_3$ for id 800.

In Section 3.5 we describe how the value of $T_{DP}$ is determined.

**Quantile-based Algorithm**

When we are dealing with real life documents such as news articles or blogs, we may often encounter situations in which the distribution of document identifiers (ids) to lists is highly non-uniform. Demaine et al. referred to this non-uniformity as “burstiness” and argued that although adaptive list intersection algorithms may be theoretically superior to static algorithms, in practice the effectiveness of adaptivity depends on the burstiness of the actual data [29]. Based on these findings, we are trying to detect as early as possible lists with non-uniform distribution of document identifiers and select in advance a good probing order to eliminate the overhead of adaptivity.

In order to determine the existence of lists with non-uniform distribution of ids, some form of data inspection is required. When a serial list intersection algorithm is employed, inspecting the lists introduces notable overhead that offsets any benefit attained. On the contrary, computing the intersection in a CMP environment provides a unique opportunity to examine the lists during the execution of a load balancing algorithm. The load balancing algorithms presented in Section 3.3 generate a statistical description of the lists in the form of approximate quantiles.

In this section, we present a list intersection algorithm, termed Quantile-based (QB), that utilizes quantiles to detect lists with non-uniform distribution of ids. QB recursively splits the partitions created by the load balancing algorithm into sub-partitions for which it determines a “good” probing order. The probing order of a partition remains fixed while that partition is being processed, thus avoiding the overhead of adaptivity. Pseudo-code for the QB algorithm is presented in Algorithm 3.

Let us illustrate the QB algorithm with a simple example. In Figure 3.2, we present parts of two sorted lists $A_1, A_2$ that belong to a partition $p$. We use the same notation ($A_i$) to refer to a list and
Algorithm 2 Dynamic Probes Algorithm

Input: $k$ sorted lists, $A_1, \ldots, A_k$, threshold $T_{DP}$

Output: The elements in list intersection

1: Sort the lists by length ($|A_1| \leq |A_2| \leq \ldots \leq |A_k|$).
2: Choose the eliminator $e$ to be the first element in $A_1$.
3: for each list $A_i$, $i = 2 \ldots k$ do
4: Perform a binary search for element $e$ and store the values and the positions of the elements “touched” by the binary search in arrays $Val$ and $Pos$, respectively.
5: end for
6: Set new eliminator $e = \text{succ}(e)$.
7: while the eliminator $e \neq \infty$ do
8: for each list $A_i$, $i = 2 \ldots k$ do
9: Using $Val$ and $Pos$, find the search interval of $e$ in list $A_i$.
10: if a search interval is not found for list $A_i$ using $Pos$ and $Val$ then
11: Set the maximum possible search interval of $A_i$.
12: Mark $A_i$ for binary search.
13: end if
14: end for
15: Sort lists by increasing search interval ($|A_2| \leq \ldots \leq |A_k|$).
16: for $i = 2 \ldots k$ do
17: if $A_i$ was marked for binary search or the search interval is larger than $T_{DP}$ then
18: Perform binary search for $e$ in $A_i$ and update $Pos$ and $Val$ entries.
19: else
20: Perform interpolation search in the search interval of $A_i$.
21: end if
22: if $e$ is found in $k$ lists then
23: Output $e$.
24: end if
25: end for
26: Set new eliminator $e = \text{succ}(e)$.
27: end while
the part of that list that belongs to a partition. The median of the ids in \( p \) is 7. Given the position of the median in each list, it is evident that the distribution of ids in \( A_1, A_2 \) is non-uniform; \( A_1 \) contains six elements with values which are smaller than 7, while \( A_2 \) contains only two such elements. Now, consider how different probing policies would work in this setting. The adaptive probing policy \([29]\) performs 10 probes to compute the intersection (shown in Figure 3.2). A probing policy with a fixed probing order requires either 9 or 10 probes depending on the order used.

Algorithm QB on the other hand, utilizes the median to detect the existence of non-uniformity in the distribution of ids (lines 3-8). The median splits every list \( A_i \) in two sub-lists \( A^1_i, A^2_i \), e.g. \( A_1 \) is divided into \( A^1_1 = \{1, 2, 3, 4, 5, 6, 7\} \) and \( A^2_1 = \{8, 11, 13\} \). For every list, QB compares the length difference of its sub-lists with a threshold \( T_{QB} \) (non-uniformity condition) to identify non-uniformity in the distribution of ids. In Section 3.5 we demonstrate how the value of \( T_{QB} \) is determined.

If at least one list \( A_i \) satisfies the non-uniformity condition, i.e. \( |A^1_i| - |A^2_i| > T_{QB} \), QB divides \( p \) in two sub-partitions \( p_1, p_2 \) (line 6). \( p_1 \) consists of sub-lists \( A^1_1, A^2_2 \) with elements which are smaller or equal than the median (shown in Figure 3.2). Accordingly, \( p_2 \) consists of sub-lists \( A^2_1, A^1_2 \) with elements which are larger than the median. For each sub-partition \( p_i \), QB greedily selects a probing order by sorting the sub-lists of \( p_i \) in increasing length \((A^2_2 \rightarrow A^1_2 \rightarrow A^1_1 \rightarrow A^2_1 \text{ for } p_1 \text{ and } A^1_1 \rightarrow A^2_2 \text{ for } p_2)\) resulting in 6 probes.

In general, the same procedure is applied recursively for every newly created sub-partition until...
Algorithm 3 Quantile-based Algorithm

**Input:** $\epsilon$-approximate quantile summary $Q$, $k$ lists of partition $P, A_1, \ldots, A_k$, threshold $T_{QB}$.

**Output:** The elements in list intersection

1: enqueue($P$); $\text{subpartitions} = 0$
2: while queue is not empty() && $\text{subpartitions} < \text{maximum number of sub-partitions allowed}$ do
3: \hspace{1em} $P' = \text{dequeue}()$
4: \hspace{1em} Compute the approximate median $m$ of $P'$ using $Q$.
5: \hspace{1em} Search for $m$ in each list $A_i$, $i = 1, \ldots, k$, of $P'$.
6: \hspace{1em} if at least one list of $P'$ is split by the median into sub-lists with length difference which is larger than $T_{QB}$ then
7: \hspace{2em} Split $P'$ in two sub-partitions $P_1, P_2$ using $m$.
8: \hspace{2em} enqueue($P_1$); enqueue($P_2$); $\text{subpartitions} += 2$
9: \hspace{1em} end if
10: end while

11: Sort the lists by increasing length to determine the probing order for that (sub-)partition.
12: Use the elements of the current smallest unexplored list as eliminators. Probe the remaining lists using interpolation search.

either a threshold of maximum number of generated partitions allowed is exceeded or until none of the lists satisfies the non-uniformity condition (lines 2-10). List intersection is computed in every (sub-)partition using the elements of the current smallest unexplored list as eliminators and probing the remaining lists in the probing order derived (lines 11-12). Each probe is performed using interpolation search.

### 3.4.2 The Case of Unsorted Lists

When the lists are unsorted, one way to compute the intersection would be to sort the lists and then apply any algorithm for sorted lists. However, sorting imposes an overhead. In this section we present Hash-based, an algorithm that computes the intersection of unsorted lists in a CMP context. It is a cache-conscious algorithm that uses hashing and avoids the overhead of sorting.
Hash-based Algorithm

The Hash-based (HB) algorithm utilizes the Partition Unsorted algorithm (Section 3.3.2) to partition the lists. Furthermore, it exploits the partial sorting of the lists, that Partition Unsorted performs, to locate the elements of each partition at low cost without examining all the elements in the lists. Then, hashing is utilized to identify the elements of intersection.

Algorithm 4 presents pseudo-code for the HB algorithm. The Partition Unsorted technique (Section 3.3.2) is utilized to compute an $\epsilon$-summary $Q$ of the elements in $k$ unsorted lists $A_1, \ldots, A_k$ (line 1). Using $Q$, the lists are divided into $P$ partitions, where $P = \left\lceil \frac{N \cdot C}{|L_2|} \right\rceil$ (line 2). The size of each partition $p$ is chosen such that $C$ partitions can fit in L2 simultaneously, i.e. $|p| = \frac{|L_2|}{C}$. Note, that since the lists are unsorted, at the present time we only know the boundary values $(lo(p), hi(p))$ of $p$. Next, the $P$ partitions are assigned to cores so that each core processes approximately the same number of partitions, i.e. $\frac{P}{C}$.

The next step, for a core processing a partition $p$, is to locate the positions of the elements that belong to $p$ in the lists. The naive approach is to read all the elements from the lists and filter out those that do not belong to $p$. However, this approach imposes an overhead, which is the cost of reading all the elements in the lists. Instead, the HB algorithm exploits a partial sorting of the lists (performed by Partition Unsorted) to locate the elements of $p$ without reading undesirable elements, i.e. elements that belong to other partitions.

Recall that the Partition Unsorted algorithm generates, along with a quantile summary $Q$, a number $m_l = \left\lceil \frac{|A_l| \cdot C}{|L_2|} \right\rceil$ of sorted runs per list $A_l (r_1, \ldots, r_{m_l})$, each containing $\frac{|L_2|}{C}$ elements. The core that processes partition $p$ utilizes these sorted runs to locate the elements of $p$ as follows (lines 4-8). In every sorted run of each list, it performs two binary searches for the boundary values of $p (lo(p), hi(p))$. For a sorted run $r$, the two binary searches determine the range of positions in $r ([lo\_pos(p), hi\_pos(p)])$ with the elements that belong to $p$.

In Figure 3.3, we present the execution of the HB algorithm for two unsorted lists $A_1, A_2$ using a CMP with two cores. Let us assume for simplicity that the number of partitions is two ($P = 2$) and the boundary values are $(1, m)$ for $P_1$ and $(m, MAX\_ID)$ for $P_2$, where $m$ is the median in the union of $A_1, A_2$ and $MAX\_ID$ the maximum document identifier in $A_1, A_2$. 
Assume that every list in Figure 3.3 consists of two sorted runs, produced by Partition Unsorted algorithm. Furthermore, assume that core 1 processes partition $P_1$ and core 2 processes partition $P_2$. Core 1 performs a binary search for the median $m$ on each sorted run of $A_1$ ($r_1^1, r_1^2$) and on each sorted runs of $A_2$ ($r_2^1, r_2^2$). The position of $m$ in a sorted run $r$ divides the elements of $r$ in two partitions, $P_1$ (denoted in black) and $P_2$ (denoted in white). The elements of $A_1$ that belong to $P_1$ are distributed in two sorted runs, i.e. the black parts of $r_1^1$ and $r_1^2$. In accordance, the elements of $A_2$ that belong to $P_1$ are distributed in the two sorted runs of $A_2$ (black parts of $r_2^1$ and $r_2^2$).

Next, we describe how a partition $p$ is processed to compute the intersection. We refer to the set of elements of list $A_i$ that belong to partition $P_j$ as segment $s_i^j$. Let $s_1, \ldots, s_k$ be the segments of $p$ (superscript is omitted for brevity). Initially, the segments are sorted by increasing length, $|s_1| \leq \ldots \leq |s_k|$ (line 10). Algorithm HB uses the elements of the smallest segment, $s_1$ to build a hash table and the elements of the remaining segments, $s_2, \ldots, s_k$ to probe that hash table and compute the intersection (lines 11-20). The entries of the hash table are $\langle id, counter \rangle$ pairs. $Id$ is used as a key and $counter$ is the payload recording the number of lists in which this $id$ exists; $counter$ is initialized to one. Every probe to an existing $id$ increments the corresponding counter. When the value of $counter$ becomes equal to the number of lists, the corresponding $id$ belongs to the intersection. In Figure 3.3, core 1 (processing partition $P_1$) utilizes segment $s_2^1$ to build a hash table and segment $s_1^1$ to probe the hash table and compute
Algorithm 4 *Hash-based Algorithm*

**Input:** $N$ elements in total in $k$ lists $A_1, \ldots, A_k$, $C$ cores, $m_l = \left\lceil \frac{|A_l| \cdot C}{L_2} \right\rceil$ sorted runs $r_1, \ldots, r_{m_l}$ per list $A_l$

**Output:** The elements in list intersection

1. Compute an $\epsilon$-approximate quantile summary $Q$ using *Partition Unsorted* algorithm.
2. Using $Q$, divide the lists into $P$ approximately equal sized partitions, where $P = \left\lceil \frac{N \cdot C}{L_2} \right\rceil$.
3. **while** $\exists$ unprocessed partitions **do**
   5. **for** $l = 1$ to $k$ **do**
   6. **for** $j = 1$ to $m_l$ **do**
   7. Search for the boundary values of $p$ in $r^j_l$. // The output is a range of positions in $r^j_l$, $[lo_{pos}(p), hi_{pos}(p)]$.
   8. **end for**
   9. **end for** // The segments $s_1, \ldots, s_k$ of $p$ have been computed.
10. Sort the segments of $p$ by increasing length ($|s_1| \leq \ldots \leq |s_k|$).
11. Build a hash table $h$ using the elements of $s_1$ (smallest segment).
12. **for** $i = 2$ to $k$ **do**
13. **if** a match $e$ is found **then**
14. Increment $counter(e)$.
15. **if** $counter(e) == k$ **then**
16. Output element $e$.
17. **end if**
18. **end if**
19. **end if**
20. **end for**
21. **end while**

the intersection (assuming $|s^2_2| < |s^1_1|$). In a similar fashion, core 2 processes partition $P_2$.

Since the size of each partition is chosen appropriately, the cores are not competing for L2 while the intersection is computed. With high probability, each hash table will reside in the L2 cache and hash probes will not cause any off-chip memory accesses. The cores are experiencing only compulsory misses while reading the elements of a partition and conflict misses due to cache associativity.
3.5 Experimental Evaluation

In this section we present a series of experiments on sorted and unsorted lists using a chip multiprocessor with eight cores. We compare the Dynamic Probes (DP) and the Quantile-based (QB) algorithms with Small Adaptive (SA) [29], Probabilistic Probes (PP) [86] and Linear Merge (LM). The latter computes the intersection of sorted lists using merging similar to the execution of a sort-merge join algorithm. Alternatively, one could view the problem of intersecting \( k \) lists as a special case of joining \( k \) relations and compare the performance of DP and QB algorithms against static AND-trees. However, as Raman et al. demonstrated, adaptive list intersection algorithms perform in most cases at least as good as the best AND-tree and are not prone to cardinality estimation errors [86]. Hence, we compare our algorithms only against other adaptive list intersection algorithms.

For unsorted lists, we evaluate the performance of the Hash-based (HB) algorithm. Due to the lack of a multi-way hash join algorithm for CMPs, we compare HB against an adaptation of a parallel and cache-conscious sorting algorithm [31] for CMPs (PS for brevity). PS is a variant of AlphaSort [80] that utilizes an exact partitioning technique [57]. The sorting algorithm can be used in conjunction with any algorithm for sorted lists. We experiment with synthetically generated datasets as well as with real data. Section 3.5.1 discusses the datasets, the hardware configurations, and implementation details. The results are presented in Section 3.5.2.

3.5.1 Methodology

We experiment with real and synthetic datasets. Our synthetic data capture uniform and correlated distributions of document identifiers. Uniform synthetic data were generated utilizing the guidelines on [42]. For real data we use a sample corpus from a web search engine\(^2\) [8]. The size of the corpus is 3.1GB of raw text and the query log contains 1000 multi-keyword queries.

All the experiments were conducted on a Dell PowerEdge 2950 server with dual quad-core Intel E5355 CPUs (8 cores) running at 2.6GHz. It has a 4MB of shared L2 cache per quad cores (8MB in total) and each core has a private 64KB L1 cache. It runs the Linux operating system with kernel 2.6.15 and has 32GB of main memory. All the algorithms were implemented in C using Posix threads (Pthreads).

\(^2\)wwwblogscope.net
Table 3.1: Range of parameter values in our experimental design.

<table>
<thead>
<tr>
<th>List size</th>
<th>1 - 100 million ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>1 - 8</td>
</tr>
<tr>
<td>Lists</td>
<td>2 - 16</td>
</tr>
</tbody>
</table>

We used the Intel icc-10 compiler with optimization level O3. In all the experiments conducted, the `sched_setaffinity` function of the Linux OS is employed to schedule the threads to specific cores and assure that they are not re-scheduled during the execution of the algorithms.

### 3.5.2 Experimental Results

In all the experiments we measured the wall-clock time required to read the lists from main memory, conduct partitioning using the quantile algorithms (for sorted and unsorted lists), compute the intersection and write the intersection result in memory. For the case of unsorted lists, where we evaluate intersection time utilizing sorting, the time includes sorting the lists, computing the partitioning using the quantile algorithm, computing the intersection using the best algorithm for sorted lists and writing the intersection result in memory. All the numbers reported are averages over 100 runs. The standard deviation for all the algorithms was less than 2%.

The maximum possible size of the intersection of \( k \) lists is the length of the smallest list. We consider the selectivity of a list intersection to be the ratio of the size of the intersection to the maximum possible size of the intersection. Unless stated otherwise, all the lists in the synthetic datasets have the same size, containing one million elements. Each element is a four byte integer. In Table 3.1, we present the range of parameter values examined in our experiments.

All the algorithms examined utilize the list partitioning techniques presented in Section 3.3 to achieve load balancing. In all the experiments that follow, we set the error threshold at \( \epsilon = 0.01 \). The maximum load disparity across cores observed in all the experiments conducted (using eight cores) never exceeded 10%.
Sorted Lists

In the first set of experiments, we assess the performance of list intersection algorithms at different selectivities and different number of sorted lists using uniform synthetic data. Figures 3.4, 3.5, 3.6 present the results for 4, 8 and 16 lists, respectively when four cores are utilized. We present the normalized time of each algorithm with respect to LM. The latter exhibits the worst performance with running times reaching 0.3 seconds (for 16 lists and 4 cores).

For four sorted lists (Figure 3.4) and when the selectivity is small (1%), SA and QB have approximately the same performance with a small advantage for the latter (10%). SA cycles repeatedly through the lists only when a match is found in the two smallest lists currently being intersected. When the selectivity is small, the frequency at which matching elements are found is proportionally small too. Hence, SA changes its probing order less frequently and does not pay the overhead of cycling through the lists. For larger selectivities though, where matching elements are discovered more frequently, SA acts more adaptively, continuously updating its probing order. That has a negative impact on its performance compared to algorithms that update their probing policy more effectively, such as DP and QB. As the number of lists increases (Figures 3.5, 3.6), the overhead of repeatedly cycling through a large number of lists results in SA performing worse than DP and QB even for small selectivities.

For small selectivities and small number of lists SA performs better than DP. However, as selectivity increases (more than 10% in our experiment), DP performs better than SA and the performance difference between these two algorithms is more pronounced. For every eliminator, DP consults the micro-index in order to refine the search range for every list and to determine the best probing order. Even though the micro-index is cache-resident, there is still some overhead involved in this procedure. On the other hand, when selectivity is small, SA discards most of the eliminators by performing a single probe on the second smallest list. Hence, it is more efficient than DP. However, as selectivity increases and the lists are probed more frequently, the use of the micro-index starts paying off and DP performs better than SA.

For small selectivities, DP and QB have approximately the same performance. However, for large selectivities the DP algorithm performs worse than QB. The latter determines a good probing order at low cost which remains fixed during its execution. The benefit of the simple probing policy that
QB applies is more profound for larger selectivities and larger number of lists where more adaptive algorithms such as SA, DP and PP experience large overheads.

Algorithm LM has the worst performance overall for the range of selectivities and the number of lists examined. Compared to other intersection algorithms, it reads the lists sequentially resulting in better utilization of cache lines. Furthermore, the sequential memory access pattern of LM is detected by the hardware prefetcher of the Intel architecture. For small number of lists the sequential memory access pattern of LM compensates for the larger number of comparisons it performs (compared to other intersection algorithms); for two sorted lists the performance of LM is equivalent to the performance of other list intersection algorithms. As the number of sorted lists increases, the execution time of LM is dominated by the excessive number of comparisons and the performance difference between LM and the rest of the algorithms increases proportionally.

The PP algorithm, although it performs significantly better than LM, is $1.6X$ slower than QB and $1.44X$ slower than SA for four lists and small selectivities (1%). The PP algorithm repeatedly cycles through the lists with some probability $p$ and chooses the next “promising” list to probe with probability $1−p$. Unless there is a probing order that is highly beneficial, PP will behave like the Adaptive algorithm [28]; the latter has been shown to be inferior to SA. The results in Figures 3.4, 3.5, 3.6 demonstrate that the PP algorithm is consistently worse than SA regardless of selectivity and number of lists.

We conducted the same set of experiments utilizing eight cores. Figure 3.7 presents the results for sixteen lists (normalized time with respect to LM). The results for smaller number of lists are consistent with those presented in Figures 3.4 and 3.5. The most important observation is that, when eight cores are utilized, the relative performance of the intersection algorithms does not change, indicating the effectiveness of the load balancing technique applied.

Next, we wanted to verify that our results are consistent when larger lists are intersected. Figure 3.8 presents the performance of list intersection algorithms for four lists when every list contains 100 million elements; four cores are used in this experiment. The results presented in Figure 3.8 are similar to those in Figure 3.4, verifying that our results are consistent irrespective of list size.

Anti-correlated Data In all the experiments presented thus far, the document identifiers that belong to the intersection were positioned randomly in each list. In the next experiment, we study the performance
of intersection algorithms for sorted lists in the case where the positions of the document identifiers that belong to the intersection are anti-correlated. This means that these document identifiers are clustered (placed close to each other in each list) and the clusters are in different positions in each of the lists. The
position of each cluster is selected randomly.

Figure 3.9 presents the results for 16 sorted lists with anti-correlated data when eight cores are utilized. LM is not included in these results as its performance does not depend on the distribution of
Figure 3.8: Performance of intersection algorithms for four sorted lists with synthetic data using four cores when each list contains 100 million elements.

the document identifiers in the lists. We present the normalized time of each algorithm with respect to PP; its running times reach 0.03 seconds. DP and QB perform consistently better than SA and PP. As selectivity increases, the performance difference of the proposed algorithms with SA and PP is more pronounced. QB performs consistently better than DP and SA as it is able to detect the non-uniformity in the lists and select from the beginning a good probing strategy. For large selectivities and large number of lists (50% in our experiments) QB is 2.5X faster than DP and 6X faster than SA and PP. The anti-correlated document identifiers degrade the performance of SA due to its conservative round robin probing policy. PP is slightly worse than SA. These results are consistent for different number of lists as well as for correlated data. In all cases examined, QB is the algorithm of choice.

Real Data The performance comparison of intersection algorithms as a function of the number of sorted lists using real data is presented in Figure 3.10; four cores were utilized. For each algorithm we measured the average time to execute a set of queries with the same number of keywords. We present the normalized average time of each algorithm with respect to PP; its running times reach 0.05 seconds. One characteristic exhibited by real data distributions on inverted lists is that the resulting selectivities of intersections is low. Hence, in practice the LM algorithm is not considered to be a competitive
algorithm. SA performs better than DP for small number of lists but as we increase the number of lists the conservative probing policy of SA reduces its performance. QB has the best performance and the performance difference with the other algorithms is increasing for larger number of lists. Notice that for 6 lists QB is $1.6X$ faster than SA and PP; the latter has approximately the same performance as SA.

**Speedup** In the next set of experiments we measure the speedup of the proposed algorithms for sorted lists as well as the performance of the parallel partitioning algorithm. In Figures 3.11 and 3.12 we demonstrate the performance of DP and QB algorithms, respectively, for computing the intersection of eight lists as we increase the number of cores; selectivity is 10%. We present the normalized running time with respect to the case when a single core is utilized (serial algorithm). We also present the fraction of time spent on the partitioning technique.

Both algorithms achieve adequate speedup as we increase the number of cores from one to four, verifying the effectiveness of the proposed partitioning technique. However, we observe that as we increase the number of cores to eight, the reduction in total running time of the DP algorithm is merely 27% compared to the case where four cores are utilized. Interestingly, the intersection time is reduced approximately 50%, indicating the ability of the partitioning technique to balance the load evenly. The
problem however, as we increase the number of cores, is the increase in the partitioning cost. Recall that our partitioning technique is tailored to CMP architectures and relies on the fact that the cores share the L2 cache. However, the hardware that we utilized contains two quad-cores which communicate through main memory. This has the following implication. In the last step of the partitioning technique a single core is producing the final $\epsilon$-summary by merging a number of summaries. Since each quad-core has its own shared L2 cache, half the accesses to the quantile summaries will have to be resolved through main memory, thus increasing the partitioning cost.

In CMP architectures in which all the cores share the L2 (L3) cache, all the quantile summaries will reside on the same cache hierarchy –as it happens when four cores are utilized in our experiment– and the partitioning cost will be significantly lower. Analogous is the case for the QB algorithm (presented in Figure 3.12).

**Micro-benchmarks** We run several micro-benchmarks to identify the best values for the thresholds used in algorithms QB and DP. Recall that algorithm QB utilizes threshold $T_{QB}$ in a *non-uniformity* condition, to decide whether a partition should be divided into smaller sub-partitions. The length of the search interval currently examined is compared to threshold $T_{DP}$ in DP, to decide whether binary search
Figure 3.11: Performance of the DP algorithm as we increase the number of cores.

Figure 3.12: Performance of the QB algorithm as we increase the number of cores.

should be applied to update the cache resident index.

We measured the running time of the QB algorithm using real data for increasing number of lists as a function of the value of $T_{QB}$. We report the average of 100 runs in Figure 3.13 for two, three and four lists. As observed, a single global optimal value for $T_{QB}$ does not exist. The value of $T_{QB}$ that results in the best performance depends on the number of lists intersected. A second observation is that the optimal value of $T_{QB}$ is shifted towards smaller values as the number of lists increases. As the number
of lists increases, a wrong decision in the probing policy applied has an even larger (negative) impact on performance. For small values of $T_{QB}$, QB tends to be less conservative, changing its probing order more often to handle fluctuations in the distribution of document identifiers.

The results for larger number of lists are consistent with those presented here. We choose to set the value of $T_{QB}$ in our experiments according to these micro-benchmarks. Namely, for each experiment involving a number of lists $k$, we set $T_{QB}$ to the best value observed in the micro-benchmarks for $k$ lists.

(a) 2 lists  
(b) 3 lists  
(c) 4 lists

Figure 3.13: Impact of threshold $T_{QB}$ on the performance of QB algorithm on real data.

Similar trends are observed in the micro-benchmarks conducted for the DP algorithm. As the number of lists increases, the optimal value of $T_{DP}$ that results in the best performance for DP is shifted towards smaller values. For small values of $T_{DP}$, DP updates its cache-resident index more often and probes the lists more effectively. Similarly, in our experiments we set $T_{DP}$ values according to the values yielding the best performance in these micro-benchmarks.

Unsorted Lists

In this section we evaluate the performance of intersection algorithms for unsorted lists using real and synthetic data. We compare the performance of Hash-based (HB) with that of sorting (PS) in conjunction with the best intersection algorithm for sorted lists; the QB algorithm is used in all the experiments. The selectivity of the intersection does not affect the performance of HB and PS algorithms. Their cost depends solely on the number of elements in the lists. Hence, in all the experiments we fix the selectivity at 10%.
In Figure 3.14 we present a performance comparison of HB and PS in conjunction with the QB algorithm as we increase the number of unsorted lists from 2 to 16, when eight cores are utilized. All the lists have the same length (1 million unique ids).

As it is evident in Figure 3.14, HB performs consistently better than the combination of PS and QB for different number of unsorted lists. HB utilizes the Partition Unsorted algorithm to leave the data partially sorted and computes the intersection without sorting the lists completely. Each core reads only the elements of the partition currently processed. Hence, the elements of the lists are read only once. However, HB pays the overhead of building and probing hash tables. We reduce this overhead by building hash tables using the elements of the smallest segments. Moreover, since the partitions are cache resident, probing the hash tables results in smaller overhead compared to the cost of sorting the lists (performed by PS). In addition to reading the elements once for performing the final merge phase, PS pays the overhead of storing the final sorted run in memory. Finally, we notice that the cost of computing the intersection using QB is relatively small compared to the cost of sorting the lists.

Figure 3.14: Performance of intersection algorithms for different number of unsorted lists with synthetic data using eight cores.

A performance comparison of the algorithms for unsorted lists using real data, when eight cores are utilized is presented in Fig 3.15. HB performs consistently better than the combination of sorting and QB (best algorithm for sorted lists with real data), verifying the efficacy of HB in practice.
In the next experiment we measure the speedup of the HB algorithm as well as the cost of the partitioning technique as we increase the number of cores (Figure 3.16). The first observation is the good speedup achieved by the HB algorithm, verifying the effectiveness of the partitioning technique. The second observation is that the total running time of HB is dominated by the partitioning cost and that the cost is significantly reduced as we increase the number of cores, demonstrating the necessity for a parallel partitioning technique. Note that, in contrast to the case of sorted lists, the partitioning technique is also employed when a single core is utilized, in order to improve cache locality.

In contrast to the speedup experiments of DP and QB algorithms (Figures 3.11 and 3.12), when eight cores are utilized we achieve significant reduction in the partitioning cost even though the cores do not operate on the same shared cache. The reason is that for unsorted lists, that cost is dominated by the sorting phase which is performed independently by each core.

### 3.6 Conclusions

We have presented algorithms to compute the intersection of an arbitrary number of sorted and unsorted lists tailored to commodity chip-multiprocessors.
Figure 3.16: Performance of the HB algorithm as we increase the number of cores.

The problem of load balancing is initially studied and cache-conscious algorithms for partitioning sorted and unsorted lists in a CMP context have been presented. Two new intersection algorithms have been proposed for sorted lists: Dynamic Probes, and Quantile-based. Dynamic Probes exploits information from previous probes as a cache-resident index. Quantile-based utilizes quantiles to detect lists with non-uniform distributions of document identifiers and select in advance and at low cost a good probing policy. For unsorted lists we have proposed a cache-conscious intersection algorithm, termed Hash-based, that computes the intersection of unsorted lists using hashing.

In a detailed experimental evaluation using real and synthetic data on a CMP with eight cores, we demonstrate that our intersection algorithms for sorted and unsorted lists have superior performance and achieve very good speedup due to the effectiveness of the load balancing strategy.
Chapter 4

Query Processing on Chip

Multiprocessors: Analyzing the Trade-offs

4.1 Introduction

Database query processing engines have been designed for legacy hardware systems consisting of uniprocessor architectures and small memory sizes. The advent of chip multiprocessors corroborated by the increasing memory sizes and the emergence of numerous techniques, such as columnar storage layout, materialized views and compression that reduce the I/O bandwidth requirements of data processing tasks, constitute existing design principles obsolete. Although several query processing algorithms have been optimized for modern hardware architectures, the interaction between database query execution engines and modern processor technologies has not been investigated adequately.

Existing database engines either rely on inter-query parallelism (multiple queries are executed in parallel) or combine data partitioning techniques with exchange operators to accelerate individual queries on multiprocessor systems. Unless memory bandwidth can keep up with the increase in the number of cores, query execution will become bandwidth-bound when inter-query parallelism is applied. On the other hand, for complex queries that process many relations, exchange operators can introduce significant overhead.

To overcome the shortcomings of existing database query engines, we investigate alternative designs for parallel query execution tailored to chip-multiprocessors. Our goal is to study: a) how database en-
gines can leverage on-chip parallelism to accelerate the execution of complex database queries, and b) what are the limiting performance bottlenecks in modern hardware architectures. Towards that goal, we introduce CMPDB, a main memory database query engine tailored to chip multiprocessors. CMPDB is capable of executing complex business intelligence queries, such as multi-way joins, on relational data and is primarily designed for parallel query execution. It leverages intra-operator and inter-operator parallelism to improve the performance of database queries without using exchange operators. Furthermore, it employs inter-query parallelism to allow for concurrent query execution.

The main contributions of this work are the following:

- We present CMPDB, a new design for parallel query execution that is tailored to chip multiprocessors.
- We performed an extensive experimental evaluation using real hardware to investigate: a) the performance improvement of database queries on chip multiprocessors, b) the limiting performance bottlenecks, and c) the performance impact of simultaneous multi-threading and multi-socket configurations on complex database queries.
- By executing database queries with different characteristics (e.g. number of join operators, selection conditions) on uniform and skewed data we have demonstrated that, on a system with 8 cores, the performance of multi-way join queries can be improved by a factor of 8 and throughput can be improved by a factor of 10. Furthermore, we have shown that non-uniform data distributions and selection conditions can significantly affect the speedup of database queries and the optimal allocation of hardware resources to query operators. Finally, we have shown that increased contention for shared resources can have a larger impact on the performance of database queries than inter-socket communication overhead.

4.2 Chip Multiprocessor Architectures

Chip multiprocessors (CMPs) or multi-core CPUs is the outcome of a continuous effort to extract performance from processors while addressing physical limitations. By supporting the concurrent execution of multiple threads of control using multiple simple processing units (cores), CMPs leverage thread-
level parallelism to deliver more performance per die area without increasing the power consumption of the CPU.

Although many vendors have adopted this design paradigm [5, 56, 62], major differences exist between different multi-core designs. Ailamaki et al. categorized chip multiprocessors in two groups according to their design principles [46]: a) lean camp, and b) fat camp. Lean camp contains multi-core CPUs with many “simple cores”, i.e. cores with short pipelines, typically operating at lower frequencies. An example of such a design is Sun Niagara T2 processor which has eight physical cores and can execute 64 threads in parallel. In contrast, multi-core CPUs in the fat camp have more complicated designs. They support both thread-level and instruction-level parallelism to deliver more performance through data parallelism without compromising the performance of sequential applications.

Despite these differences, many commonalities exist between the two camps. Commonly, a multi-level cache hierarchy resides on chip, the lowest level of which is shared among cores, enabling fast inter-core communication. Another common characteristic is that memory bandwidth is shared among cores. Figure 4.1 presents a simplified design of a quad-core CPU. Every core supports two logical threads and has private L1 and L2 caches. The L3 cache and memory bandwidth are shared among cores.

CMP architectures exhibit different computation and communication trade-offs compared to other multi-processor architectures. Communication among cores is considerably faster as it is typically performed through shared cache. At the same time, the existence of many shared resources among processing elements may introduce performance bottlenecks. These characteristics demand a shift in the design and optimization of data processing techniques. In the context of CMP architectures it is imperative to consider cache-conscious and bandwidth-conscious techniques that manage shared resources more judiciously to reduce contention among cores.

4.3 CMPDB: A Database Query Engine for CMPs

4.3.1 General Architecture

CMPDB is a main-memory database engine tailored to chip multiprocessors. It leverages intra-query parallelism to improve the performance of database queries and inter-query parallelism to allow for
Figure 4.1: An example of a quad-core CPU with two threads per core and shared L3 cache.
concurrent execution of queries. A high level description of the CMPDB architecture is presented in Figure 4.2. CMPDB consists of three main components: a) the catalog, b) the hardware coordinator, and c) the query execution engine.

![Diagram of CMPDB Architecture](image)

Figure 4.2: General architecture of the CMPDB database engine.

The catalog stores the following information for each relation: a) the schema (number of attributes, attributes length/type), b) the location and size on persistent storage (disk), and c) statistics such as cardinality (number of records). Relations are organized using the N-ary storage model (NSM) and records are stored contiguously in 32KB pages. Relations are preloaded in memory using memory mapped I/O upon the initialization of CMPDB. For simplicity, only fixed-length attributes are supported in CMPDB. Although PAX or DSM layouts may seem more suitable for main memory database systems, we consider the use of NSM as our starting point for this analysis due to its popularity in commercial
databases. PAX can be easily integrated in CMPDB since it is also a page-based layout. DSM would require significant changes in the current code-base.

The hardware coordinator is the module responsible for allocating processing elements (cores/threads) to queries and individual query operators. Currently, CMPDB assigns processing elements to queries/operators statically, i.e. that assignment does not change during query execution. Though not optimal, the static allocation is sufficient for the needs of our study. In future implementations of CMPDB we plan to integrate a dynamic hardware coordinator that in collaboration with a monitoring module (denoted in dotted rectangle in Figure 4.2), will dynamically allocate hardware resources depending on the needs of queries and operators.

The main module of CMPDB is the query execution engine, described in detail in Section 4.3.2. The engine can execute any database query using intra-query parallelism. Multiple instances of the query engine may exist at the same time to allow for concurrent execution of queries. Clients connect to the CMPDB server and issue queries. Due to the lack of an SQL-based interface, clients submit queries in the form of query execution plans (QEP) described in an XML-like format. Furthermore, due to the absence of a query optimizer, a QEP must contain all the information needed for the engine to execute the query. For instance, the QEP for a three-way join query must describe the relations, the join attributes, the projected attributes, any selection conditions and the join order. Optionally, the QEP describes the allocation of processing elements to query operators.

4.3.2 Query Execution Engine

Primarily, CMPDB focuses on data warehousing workloads or business intelligence (BI) queries such as those found in the TPC-H benchmark\(^1\). Typically, these are long running queries that access multiple relations and contain many operators such as join, sort and aggregation. For simplicity, we consider multi-way join queries executed as a pipeline of binary joins. An example of a four-way join among relations \(R_1, R_2, R_3,\) and \(R_4\), that is executed as a pipeline of three binary join operators is presented in Figure 4.3 along with its corresponding SQL statement. Any combination of join algorithms can be employed to execute this query. In our study, we only consider the use of hash joins.

The query execution engine of CMPDB employs a combination of partition (intra-operator) and

\(^1\)http://www.tpc.org/tpch/
pipeline (inter-operator) parallelism to accelerate multi-way join queries like the one presented in Figure 4.3 on a CMP architecture. Partition parallelism denotes the execution of a single operator using multiple cores. Pipeline parallelism denotes the concurrent execution of operators of the same query. The following sections describe these two concepts and how they are incorporated in the design of CMPDB.
Partition (Intra-Operator) Parallelism

Partition or intra-operator parallelism is applied to accelerate an individual operator (e.g. join, sort). We discuss how partition parallelism improves the performance of the join operator when the hash join algorithm is employed; we consider the classical two-phase hash join algorithm. In the first phase, termed build phase, a hash table is built from the records of the smallest (in size) relation. In the second phase, termed probe phase, the hash table is probed using the records of the bigger relation to identify pairs of matching records. A new record is formed from every pair of matching records that is either sent to the output or the parent operator, if any.

When the hash table of the build relation does not fit in cache, memory access is a major performance bottleneck during the execution of the hash join algorithm as every probe results in a cache and, possibly, a TLB miss. Several approaches have been proposed to alleviate memory access cost and improve the performance of hash joins on modern hardware architectures. Typically, a three-phase algorithm is applied to improve the cache performance of the hash join algorithm. Initially, cache partitioning is applied to divide the relations into a number of cache-sized partitions. Build and probe phases are applied for every pair of partitions and the number of partitions is chosen so that the hash table of each build partition fits in cache, thereby reducing the number of data cache misses during the probe phase.

There are two issues with cache partitioning techniques. First, they assume exclusive use of the cache hierarchy by a single operator. This is not a realistic assumption to be made in database systems where queries with different working set sizes arrive continuously and are executed in parallel. The problem is exacerbated by the fact that caches are managed by hardware and not by the database engine.

The second and most important issue is that by introducing a partitioning step, joins are transformed into blocking operators, thereby negating the benefits of pipeline parallelism in the context of complex queries. At the same time, storing the intermediate results of each operator increases the memory requirements of database queries. To avoid these limitations, we do not consider the use of cache-partitioning techniques in CMPDB. Instead, we rely on software prefetching techniques to hide memory access stalls with useful computation. Next, we describe how intra-operator parallelism is applied to speedup the hash join algorithm.
**Build Phase**  In the build phase, a hash table is constructed from the records of the smaller relation. The hash table is an array of buckets that stores *key-pointer* pairs. Join attribute values are used as keys. A pointer stores the memory address of a record. For every record of the build relation, a key-pointer pair is inserted in a hash bucket, the address of which is determined by hashing the key value. Collisions are resolved through chaining, i.e. a linked list of buckets is formed. Hash buckets are 64 bytes (cache line size) and are aligned to cache line boundaries to avoid false sharing and improve memory bandwidth utilization.

The build phase is executed in parallel by $N$ cores as follows. Every core processes a disjoint set of $B/N$ pages, where $B$ is the number of pages of the build relation. All cores update the same hash table in parallel and a fine-grain synchronization scheme is applied to ensure correctness. In particular, every hash bucket is associated with a mutex. A core trying to update a hash bucket must gain exclusive access by locking the corresponding mutex. Upon completion the mutex is released, allowing other cores to update the same bucket. Software prefetching is utilized to bring hash buckets in cache before they are updated.

**Probe Phase**  In the probe phase, the cores probe the hash table using records from the larger relation to identify pairs of matching records. Every core processes a disjoint set of $P/N$ pages, where $P$ is the number of pages of the probe relation. The same hash function is applied as in the build phase. A new record is formed from each pair of matching records by merging the corresponding projected attributes. No synchronization is required during the probe phase as there are no updates. Software prefetching is also applied to reduce memory access cost while probing the hash table.

**Pipeline (Inter-Operator) Parallelism**

Partition parallelism is employed to improve the performance of a single operator. Next, we present the concurrent execution of operators of the same query using pipeline (inter-operator) parallelism.

Existing database systems are designed for single processor architectures and execute queries using the *iterator* model. The iterator model provides a simple and intuitive interface that facilitates the integration of new operators in the query engine. Every operator must implement a *getNext* function. A *getNext* call executes a single operator until a new record is produced and send to the parent operator.
in the query execution plan. Query operators are executed in an interleaved fashion through successive 
\textit{getNext} calls. On the contrary, CMPDB applies pipeline parallelism by executing different operators in 
parallel. A number of cores is assigned to each operator and that assignment remains fixed during query 
execution. Consequently, query operators are executed in a producer-consumer mode.

In this setting, the iterator model with its tuple-at-a-time ”communication” mechanism is inappro-
priate. It serializes the execution of neighboring operators and increases synchronization overhead. 
Furthermore, it is susceptible to increased load disparity across cores due to variable record produc-
tion/consumption rates. To overcome these limitations, CMPDB introduces a buffer mechanism and a 
modified execution model to facilitate inter-operator communication and allow for concurrent operator 
execution in the context of pipeline parallelism.

The buffer mechanism is implemented as a new database operator, termed \textit{BufferManager}, always 
residing between two operators that are executed in parallel. BufferManager is a placeholder for \textit{buffers}, 
which are fixed-size arrays of records. Internally, a BufferManager operator consists of two queues: 
\textit{EmptyQueue}, and \textit{FullQueue}. EmptyQueue stores empty buffers. They are used as placeholders for 
result records generated from the BufferManager’s child operator. FullQueue stores full buffers used as 
input to the BufferManager’s parent operator.

A new execution model is applied in CMPDB through the following interface that is implemented 
in BufferManager: a) \textit{getEmpty}, b) \textit{getFull}, c) \textit{putEmpty}, and d) \textit{putFull}. Function \textit{getEmpty} returns the 
next available buffer from EmptyQueue. If there are no buffers left, the calling thread blocks. Function 
\textit{getFull} returns the next available buffer from FullQueue. If there are no buffers left in that queue, 
the calling thread blocks. In a similar fashion, \textit{putEmpty} (\textit{putFull}) inserts an empty (full) buffer in 
EmptyQueue (FullQueue). BufferManager has a fixed capacity of \textit{buffers} that does not change during 
the execution of a query. A mutex is utilized to synchronize access to a BufferManager and ensure 
correctness when the queues are updated by multiple threads.

The buffer mechanism facilitates inter-operator communication and alleviates synchronization over-
head as threads do not have to block or wait for every record. Furthermore, the buffer mechanism makes 
CMPDB more resistive to varying record production/consumption rates.
Query Execution Engine in Practice

We demonstrate the execution of multi-way join queries on CMPDB using a running example. We use a three-way join query with an execution plan shown in Figure 4.4.

Upon entering the CMPDB engine, each query is prepared for execution by allocating hardware resources and inserting BufferManager operators, if needed. Figure 4.5 illustrates the transformed execution plan of the example query once the preprocessing step has been applied. We assume that each join operator is executed by two cores; cores 1, 2 are assigned to Join 1 and cores 3, 4 are assigned to Join 2. Furthermore, a BufferManager operator with 4 buffers has been inserted between the two join operators. All the buffers in the BufferManager are initially empty and reside in EmptyQueue.

Since every join operator uses the two phase hash join algorithm, the query is executed in two steps. The build phase of every join operator is executed in the first step. The probe phase of every join is executed in the second step. In the example three-way join query, cores 1, 2 build a hash table using the records of relation $R_2$, while cores 3, 4 build a hash table using the records of relation $R_3$. The two build phases proceed in parallel. The second step begins when both the hash tables have been constructed. Subsequently, cores 1, 2 execute the probe phase of Join 1 and cores 3, 4 execute the probe phase of
Join 2.

Algorithm 5 presents the pseudocode of the probe phase executed by each core and illustrates the execution model applied in CMPDB. There is an initialization phase (line 1) in which every core initializes an output buffer. If the core is executing the last join operator in the pipeline no output buffer is required and the results are sent directly to the client. Otherwise, an empty output buffer is retrieved by calling the getEmpty function of its parent operator (BufferManager).

The main execution of the probe phase is presented in lines 2 – 12. Every join operator repeatedly calls the the getNext function of the left child operator until the left input is exhausted. If the left child is a BufferManager operator, getNext is translated into a getFull call, returning the next available buffer from FullQueue. If the left child is a scan operator, getNext returns the next input page from the base relation. In both cases, records from the left input are used to probe the corresponding hash table. When a match is found, a result record is inserted in the output buffer (line 6). If the output buffer is full, it is sent to the parent operator through the putFull call (line 8). At the same time a new empty buffer is requested (line 9). The full buffer is placed in FullQueue until it is requested from the second join operator (Join 2). The execution of the second step is demonstrated in Figure 4.6.
Algorithm 5 Probe Phase

**Input:** OP: current operator, Parent: parent operator of OP in QEP, LeftChild: left operator, HT: hash table

**Output:** Join result

1: outputBuf = Parent.getEmpty()
2: while (inputBuf = LeftChild.getNext()) ≠ EMPTY do
3:   Probe HT using records of inputBuf.
4:   if a match is found then
5:     Produce result record r.
6:     Insert r in outputBuf.
7:   if outputBuf == FULL then
8:     Parent.putFull(outputBuf)
9:     outputBuf = Parent.getEmpty()
10:   end if
11: end if
12: end while

Figure 4.6: An instance of the probe phase execution for a three-way join query.

4.4 Analysis

We analyze the performance trade-offs of executing database queries on CMP architectures using the CMPDB engine. We discuss the experimental setup in Section 4.4.1. The results of our analysis are
presented in Section 4.4.2.

### 4.4.1 Experimental Setup

The CMPDB engine was implemented in C using Posix threads and compiled using the gcc compiler with -O2 optimization flag. All the experiments were conducted on a dual socket server the characteristics of which are presented in Table 4.1. It has two quad-core Intel Xeon E5540 (Nehalem) processors and 48GB of memory. It runs the 64-bit Linux operating system (kernel version 2.6.31).

<table>
<thead>
<tr>
<th>Microprocessor Architecture</th>
<th>Nehalem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets (CPUs)</td>
<td>2</td>
</tr>
<tr>
<td>Processor</td>
<td>Xeon E5540</td>
</tr>
<tr>
<td>Cores</td>
<td>8</td>
</tr>
<tr>
<td>Threads</td>
<td>16</td>
</tr>
<tr>
<td>L1 Cache (per core)</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 Cache (per core)</td>
<td>256KB</td>
</tr>
<tr>
<td>L3 Cache (shared)</td>
<td>8MB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>32GB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>48GB</td>
</tr>
</tbody>
</table>

Xeon E5540 processors support simultaneous multithreading, allowing each core to run two threads in parallel. Each quad-core has a unified L3 cache that is shared among cores. Furthermore, the Nehalem micro-architecture has an integrated memory controller that supports three channels of DDR3 memory and a fast point-to-point processor interconnect (QPI link) that enables direct inter-socket communication. The maximum theoretical memory bandwidth is 32GB/sec when 1333MHz DDR3 is used. The maximum transfer rate of the QPI link is 12GB/sec.

We experimented with synthetically generated data and queries from the TPC-H benchmark\(^2\). We

\(^2\)[http://www.tpc.org/tpch/]
run experiments with uniform and skewed data distributions. Skewed data were generated using a modified TPC-H data generator. Relations were generated with scale factor 10. Table 4.2 presents the characteristics of the relations used in our analysis. The default page size was 32KB.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Size</th>
<th>Tuple Length (bytes)</th>
<th>Attributes</th>
<th>Cardinality</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lineitem</td>
<td>7.2GB</td>
<td>120</td>
<td>16</td>
<td>60,000,000</td>
<td>229831</td>
</tr>
<tr>
<td>Orders</td>
<td>1.8GB</td>
<td>120</td>
<td>16</td>
<td>15,000,000</td>
<td>57471</td>
</tr>
<tr>
<td>Customer</td>
<td>260MB</td>
<td>168</td>
<td>22</td>
<td>1,500,000</td>
<td>7936</td>
</tr>
<tr>
<td>Nation</td>
<td>32KB</td>
<td>88</td>
<td>12</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>Region</td>
<td>32KB</td>
<td>76</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

In the majority of experiments, we measured the wall-clock time to run database queries on CM-PDB. Since relations are pre-loaded in memory, the running time includes the time to execute the query and write the results to memory, i.e. there is no I/O activity. We also measured system’s throughput (queries/sec) while executing multiple queries in parallel.

4.4.2 Results

Speedup

The first step in our analysis is to examine the speedup of database queries on chip-multiprocessor architectures. We are interested in investigating the effect of intra-query parallelism on performance as well as the impact of different hardware configurations, such as simultaneous multi-threading and multi-socket systems.

Initially, we study the performance of a two-way join query between relations Lineitem and Orders as we increase the number of threads from 1 to 8. It is a primary-foreign key join in which every tuple from relation Lineitem matches exactly one tuple from relation Orders. We consider two configurations: a) all threads are executed on the same CPU (SINGLE SOCKET), b) threads are evenly distributed among CPUs (DUAL SOCKET). For the single socket configuration, simultaneous multi-threading is enabled when the number of threads exceeds the number of physical cores. Unless stated otherwise,
there are no selection conditions and all the attributes of the participating relations are send to the output, i.e. size of result tuples is 240 bytes. The results are presented in Figure 4.7.

![2-way Join](image)

Figure 4.7: Performance of a two-way join query on a single and dual-socket configuration as we increase the number of threads from 1 to 8.

When the number of threads is less than the number of physical cores per CPU (4 in our architecture), the single socket configuration performs better. The dual-socket configuration exhibits higher communication overhead during the execution of the build phase due to the coherency protocol. When the number of threads exceeds the number of physical cores, the dual-socket configuration is 14% faster compared to the use of simultaneous multi-threading (single socket). As the number of threads increases, contention for shared resources escalates. As the results from this experiment suggest, increased contention for shared resources does not compensate for the reduced inter-processor communication overhead and, consequently, the dual-socket configuration performs better.

The previous experiment considered a database query with a single join operator. Next, we analyze the speedup of database queries that involve multiple join operators. We measured the performance of
a three-way join query among TPC-H relations Lineitem, Orders, and Customer. The query is executed using a combination of partition and pipeline parallelism, i.e. the two join operators run concurrently and every join is executed by the same number of threads. Figure 4.8 presents query execution time for different number of threads and different allocation of threads to physical cores (single vs dual socket configuration).

![3-way Join](image)

Figure 4.8: Performance of a three-way join query on different hardware configurations.

The dual-socket configuration performs 20% better irrespective of the number of threads that execute the three-way join query. In the single socket configuration all the join operators are executed on the same CPU. Although this alleviates inter-operator communication overhead, it increases the number of cache misses because operators compete for the shared cache. When the dual socket configuration is used, each operator is executed on a different CPU at the expense of a higher communication overhead. The same experiments were conducted using multi-way join queries with 3 and 4 join operators. In all the experiments, the dual-socket configuration was always 24% to 35% faster.

The results from the last two experiments demonstrate that the performance of database queries
is largely affected by the allocation of logical threads to physical resources (cores and sockets). The hardware configuration that results in the best performance depends on: a) the number of threads utilized to execute a query, and b) the complexity (denoted by the number of join operators) of a database query.

Next, we analyze the speedup of complex join queries and investigate the limiting performance bottlenecks in our system. We measure the performance of increasingly complex join queries as we vary the number of threads. The hardware configuration (allocation of threads to cores) that results in the best performance is always selected.

![2-way Join](image)

Figure 4.9: Speedup of a two-way join.

In Figure 4.9, we present the execution time of a two-way join among relations `Lineitem` and `Orders` as we increase the number of threads from 1 to 16. We also present the running time of each join phase. The two-way join query achieves near linear speedup as we increase the number of threads from 1 to 4. As we increase the number of threads even further, the running time is reduced at a slower rate; speedup is 6.05 and 7.91 when 8 and 16 threads are used, respectively.

In Table 4.3, we present the speedup of each join phase. The probe phase achieves almost linear
Table 4.3: Speedup of two-way join

<table>
<thead>
<tr>
<th># Threads</th>
<th>Total</th>
<th>Build Phase</th>
<th>Probe Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.95</td>
<td>1.71</td>
<td>2.00</td>
</tr>
<tr>
<td>4</td>
<td>3.80</td>
<td>2.98</td>
<td>4.00</td>
</tr>
<tr>
<td>8</td>
<td>6.05</td>
<td>3.93</td>
<td>7.57</td>
</tr>
<tr>
<td>16</td>
<td>7.91</td>
<td>4.23</td>
<td>11.71</td>
</tr>
</tbody>
</table>

speedup for up to 8 threads. When more than 8 threads are used in our experiment, simultaneous multi-threading is enabled and speedup improves significantly, though at a smaller rate. As Table 4.3 demonstrates, the performance of the build phase does not improve at the same rate, thereby limiting the performance improvement of the join query on CMP architectures.

Hardware performance counters were utilized to analyze the performance of the join operator and explain the poor scalability of the hash join algorithm. The most interesting results are presented in Table 4.4. The second column shows how CPI (cycles per instructions) varies as we increase the number of threads that execute the join operator on a single socket configuration. CPI increases significantly (60% in our experiment) as we increase the number of threads from 1 to 8, indicating the existence of a larger number of stalls.

The third column, termed Coherency Overhead, is a data latency analysis ratio that counts the overhead of accessing modified cachelines from another core’s cache. We utilize this ratio as a proxy of the cache-coherency protocol overhead. Increasing the number of threads increases the number of cachelines read that exist in some other core’s private cache in modified state. This is attributed to the write-intensive nature of the build phase in conjunction with the fact that the hash table constructed in this phase is shared among cores.

Another interesting observation is that more time (measured in CPU cycles) is spent on system calls as we increase the number of threads; the results are presented in the last column of Table 4.4. The number of cycles that were spent on the Linux kernel increased by 40% as we varied the number of threads from 1 to 8. We identified that most of this time was spent on system calls related to memory allocation during the execution of the build phase.
Our findings suggest that the small speedup of the build phase is attributed to: a) the write-intensive nature of the build phase that increases the overhead of the cache-coherency protocol, and b) the poor scalability of the operating system’s memory allocator.

Table 4.4: Hardware performance counters

<table>
<thead>
<tr>
<th># Cores</th>
<th>CPI</th>
<th>Coherency Overhead</th>
<th>Kernel (% of total cycles) - CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.05</td>
<td>0%</td>
<td>19.12% - 1.68</td>
</tr>
<tr>
<td>2</td>
<td>1.1</td>
<td>3%</td>
<td>20% - 2.14</td>
</tr>
<tr>
<td>4</td>
<td>1.18</td>
<td>7%</td>
<td>21% - 2.32</td>
</tr>
<tr>
<td>8</td>
<td>1.68</td>
<td>10%</td>
<td>26.5% - 3.29</td>
</tr>
</tbody>
</table>

In Figure 4.10, we present the execution time of multi-way join queries as we vary the number of join operators for different number of threads. For each query, we set the number of threads to be a multiple of the number of join operators and we allocate the same hardware resources to each join operator.

![Figure 4.10: Speedup of multi-way join queries as we increase the number of threads.](image)

As the results in Figure 4.10 illustrate, the performance of multi-way join queries is significantly improved on CMP architectures, indicating the effectiveness of the buffer mechanism and the ability of our query processing engine to leverage on-chip parallelism of CMP architectures. The performance of each multi-join query is improved by at least a factor of 7 when 8 cores are fully utilized. Again, the
limiting performance bottleneck is the poor speedup of the build phase and the increased conflict on shared resources when simultaneous multi-threading is enabled.

**Effect of non-uniform data distributions**

We examine the effect of non-uniform data distributions on the performance of multi-way join queries. We consider a three-way join query among relations *Lineitem*, *Orders*, and *Customers*. The join attribute values of the probe relation (*Lineitem*) are uniformly distributed and the join attribute values of the build relations (*Orders* and *Customers*) are generated using a Zipf data distribution; the frequency of value $k$ is proportional to $(\frac{1}{k})^Z$, where $0 \leq Z \leq 1$ is the skew. When $Z = 0$, join attribute values are uniformly distributed. In Figure 4.11, we present the speedup of the three-way join query for different skew values as we increase the number of threads allocated.

![Performance of 3-way join on skewed data](image)

Figure 4.11: Speedup of a three-way join query on non-uniform data.

For a given number of threads, the performance of the join query is worsen as we increase the value of skew ($Z$). As the join attribute values of the build relations become more skewed, a smaller set of
hash buckets is continuously updated, thereby increasing synchronization, cache-coherency and memory allocation overheads. The increasing number of hash bucket collisions also results in the formation of long bucket overflow lists. Traversing these overflow lists during the probe phase diminishes the performance of the join query even further. When 16 threads execute the query, speedup is reduced from 9 to 5.4 as we increase skew from 0.1 to 0.9, indicating that skew can significantly impact the performance of database queries on CMP architectures.

Effect of buffer mechanism

The buffer mechanism (BufferManager operator) is utilized to facilitate inter-operator communication of multi-operator database queries. Every BufferManager operator has two configuration parameters: a) the number of buffers, and b) the size of each buffer. The effect of these configuration parameters on performance is examined next. Initially, we consider the execution of a three-way join query as we vary the number of buffers from 2 to 64. The size of each buffer is 32KB. The results for different number of threads are presented in Figure 4.12.

As the results in Figure 4.12 clearly demonstrate, the best performance is attained when the number of buffers of a BufferManager operator is at least equal to the number of threads that access that operator. If less buffers are used, some threads remain idle and performance is worsen; more buffers do not impact performance as they are rarely accessed.

The impact of buffer size on performance is investigated in the following experiment. We executed the same three-way join query as before but we varied the buffer size from 4KB to 128KB. In all cases, the BufferManager operator contained 16 buffers. The results are presented in Figure 4.13.

The results from this experiment indicate that buffer size can impact the performance of database queries. Small buffers exhibit small memory footprint but fill faster. Hence, threads spend more time interacting with the BufferManager. Since accesses to the BufferManager are synchronized, small buffers result in higher synchronization overheads. As buffer size increases, the synchronization overhead is reduced and the performance of the database query is improved. However, very large buffers (more than 64KB in our experiment) degrade query performance as they pollute the cache hierarchy.
Using selection conditions

In all the experiments presented so far, there were no selection conditions, i.e. all records participated in the join operators. Typically, complex business intelligence queries have selection conditions on the participating relations that affect the output cardinality of individual operators and queries. Next, we examine the performance impact of selection conditions on multi-way join queries in the context of CMP architectures.

Initially, we consider the performance of a two-way join query (single join operator) when there is a selection condition on the build relation (*Orders*). We vary selectivity from 10% to 100%. When selectivity is 100% all the tuples of the build relation participate in the build phase. The results for 1, 8 and 16 threads are presented in Figure 4.14.

As expected, the performance of the join query is worsened as selectivity increases because more
tuples participate in the build phase. We observe that selectivity has smaller impact on the performance of the join query as we increase the number of threads. For example, the performance difference between
10% and 100% selectivity is 100%, 66%, and 18% when 1, 8, and 16 threads are used, respectively.

When selectivity is low, most of the tuples that are read from memory do not satisfy the selection condition. Hence, most of the time is spent on transferring data from memory and the memory subsystem of the CPU is saturated with load requests. As selectivity increases, more tuples participate in the build phase and the average per tuple overhead increases due to more frequent hash table updates. Consequently, more instructions are executed and all the components of the CPU (cores) are utilized more effectively. Interestingly, selectivity has an even smaller impact on performance when simultaneous multithreading is employed (16 threads in our experiment). The use of simultaneous multi-threading alleviates the latency problem as it effectively overlaps memory access stalls with useful computation.

This effect is best illustrated in Figure 4.15 in which we plot the speedup of the two-way join query for 8 and 16 threads as a function of selectivity. For selectivities less than 20%, using 8 threads achieves better speedup. Using more threads increases saturation of CPU memory sub-system and worsens speedup. However, this situation is reversed for higher selectivities since there are more opportunities for hiding memory access stalls.

Similar results are observed for the case when there is a selection condition on the probe relation. The performance of the two-way join query for different number of threads and different selectivities are presented in Figure 4.16; Figure 4.17 also presents how speedup varies as a function of selectivity for different number of threads. The results from these experiments indicate that the memory sub-system of modern CMP architectures can become a limiting performance bottleneck in complex database queries.

In all the previous experiments with multi-way join queries, all join operators had the same output cardinality and produced tuples at approximately the same rate. Consequently, an even allocation of hardware resources to operators was optimal (in terms of execution time) not resulting in load disparity across threads. Next, we investigate what is the best allocation of threads to operators when the join operators produce tuples at different rates.

We consider the execution of a three-way join query (two join operators) using 8 threads. A selection condition is applied on the build relation of the second join to alter the output cardinality and consequently the tuple generation rate of that operator. For example, when selectivity is 100% both operators produce result tuples at the same rate. When selectivity is 50% the first join produces tuples at a double rate. We use a $k - m$ pair to denote an allocation of threads to two join operators. For example,
Figure 4.15: Speedup of a two-way join query when there is a selection condition on the build relation.

Figure 4.16: Selection condition on the probe relation.

pair 4 – 4 denotes that 4 threads are allocated to each join. In Figure 4.18, we present the execution time of the three-way join query for different allocation schemes (configurations) as we vary the selectivity
of the second build relation from 10% to 100%.

When selectivity is 100% the optimal allocation is the one that evenly distributes threads to cores (4 − 4). However, as selectivity decreases and the first join performs more work, more threads must be allocated to the first join to accommodate for the different tuple production rates. Otherwise, load disparity is introduced across threads and performance is worsen. When selectivity is 10%, the best performance is attained when 6 threads are allocated to the first join and 2 threads are allocated to the second (6 − 2). As the results from this experiment suggest, optimizing the execution of complex queries on CMPs requires the design of a resource allocation mechanism that takes into account the characteristics of each query operator.

**Throughput**

In all the previous experiments, we considered the isolated execution of database queries, i.e. only one query is executed in CMPDB. Next, we consider the concurrent execution of database queries and we
measure system’s throughput (number of queries executed per unit of time). Initially, we examine how throughput varies as we increase the number of queries executed concurrently. To measure system’s throughput we assume a close system in which $k$ clients issue queries continuously with no think time. For each value of $k$, 100 queries were executed in total. All clients issued the same three-way join query that was executed by a single thread (no intra-query parallelism). We considered two variants of the three-way join query. The first had no selection conditions (label \textit{NO SEL}) and the second had a selection condition with 10% selectivity on the probe relation of the first join operator (label \textit{WITH SEL (10%)}).

The results for different number of clients are presented in Figure 4.19. They demonstrate that throughput increases near linearly to the number of clients when queries do not have selection conditions. Throughput improved by a factor of 10 when both CPUs were fully utilized (16 clients), suggesting that CMP architectures can significantly improve the throughput of query processing engines.
When queries with selection conditions were executed, throughput levelled off when more than 8 clients issued queries. In our experiment, throughput was improved by a factor of 7.5 when all CPUs were fully utilized (16 clients). Queries with low selectivity selection conditions spend most of their time transferring data from memory. Hence, as the number of clients increases, memory controller is saturated with load requests and eventually throughput reaches a plateau.

![Throughput of CMPDB vs. number of clients](image)

**Figure 4.19**: Throughput of CMPDB as we vary the number of clients in the system.

Next, we examine the effect of using both inter-query and intra-query parallelism on system’s throughput. A \((k, m)\) configuration denotes the number of concurrent queries \(k\) and the number of threads \(m\) that execute each query. For example, configuration \((2, 8)\) denotes the execution of two queries in parallel while each query is executed using 8 threads. The results are presented in Figure 4.20. We considered only configurations that fully exercise both CPUs, i.e. \(k \times m = 16\).

As the results from this experiment indicate, the best throughput is achieved when each query is executed by a single thread (no intra-query parallelism). This is attributed to the sub-linear speedup of multi-way join queries.
Chapter 4. Query Processing on Chip Multiprocessors: Analyzing the Trade-offs

4.5 Conclusions

We investigated the trade-offs of executing database queries on chip multiprocessors. Initially, we presented CMPDB, a main memory database query engine tailored to CMP architectures and we demonstrated the changes required in the execution engine to leverage on-chip parallelism. We demonstrated how intra-query parallelism can be applied to improve the performance of database queries and we examined the performance impact of non-uniform data distributions and different hardware configurations.
Chapter 5

Suffix Tree Construction Algorithms on Chip Multiprocessors

5.1 Introduction

Suffix trees are indexing data structures that enhance the performance of numerous string operations. However, existing suffix tree construction algorithms do not consider the characteristics of modern hardware architectures such as multi-level cache hierarchies and on-chip parallelism.

We present in-memory suffix tree construction algorithms tailored to chip-multiprocessors. Initially, we consider the problem of cache-partitioning in the context of suffix trees. Cache-partitioning provides the means to parallelizing a task and improving its cache performance. Existing partitioning techniques introduce significant overhead that negates the benefits of improved cache performance. To address this issue, we propose a low overhead cache-partitioning algorithm, termed PreCache, that is utilized as a building block in suffix tree construction algorithms. PreCache employs sampling and is designed for CMP architectures.

Next, we propose two suffix tree construction algorithms, termed CMPUTree and MAPST (MAterialized Prefixes Suffix Tree), which are tailored to CMP architectures. CMPUTree is a cache-conscious parallel adaptation of Ukkonen’s algorithm [102]. Ukkonen proposed a linear time suffix tree construction algorithm that severely underutilizes modern hardware resources. In contrast, CMPUTree is a linear time algorithm that utilizes Ukkonen’s algorithm and the PreCache partitioning technique as building blocks.
in order to construct suffix trees in a cache-conscious and parallel way.

The second algorithm proposed, termed MAPST, is a cache-conscious suffix tree construction algorithm tailored to CMPs that is more space efficient than CMPUTree at the expense of a theoretical worse construction time \( O(n \log n) \). Inspired by the WOTD algorithm [38], MAPST constructs the suffix tree in a top-down fashion, allowing for a more space efficient representation of tree nodes. By utilizing materialized prefixes as well as different compression techniques, it effectively trades space for cache performance, significantly reducing the suffix tree construction cost. Furthermore, it employs PreCache as a building block to parallelize the suffix tree construction task and to bound the size of each core’s working set.

Finally, we present an extensive experimental evaluation of the algorithms proposed herein using real text corpora from different domains. We demonstrate that the applied techniques improve cache performance of in-memory suffix tree construction algorithms. Furthermore, all the algorithms achieve satisfactory speedup, significantly reducing the suffix tree construction cost.

### 5.2 Survey of Suffix Tree Construction Algorithms

In this section, we describe existing suffix tree construction algorithms by categorizing them as follows: a) in-memory algorithms (Section 5.2.1), b) algorithms that are optimized for disk performance (Section 5.2.2), and c) parallel algorithms (Section 5.2.3).

#### 5.2.1 In-memory Algorithms

Linear time algorithms for constructing suffix trees in main memory have been proposed by Weiner [103], McCreight [75], and Ukkonen [102]. One common characteristic of these algorithms is that, due to the use of suffix links, they all have irregular tree traversal patterns and hence, they exhibit poor cache performance. However, Ukkonen’s is the algorithm of choice due to the following reasons: a) it poses less space requirements in practice, b) it is constructed online, and c) it is easier to understand and implement. A high level description and the limitations of Ukkonen’s algorithm are presented in detail in Section 5.3.

Several algorithms have been proposed that abandoned the use of suffix links in order to reduce
the space requirements of suffix trees and improve temporal locality [23, 38]. These algorithms exhibit an $O(n^2)$ worst time complexity; $O(n \log n)$ for finite alphabets. The most space efficient suffix tree construction algorithm in this category is the write-only top-down algorithm (WOTD) proposed by Giegerich et al. [38]. The WOTD algorithm proceeds in a top-down fashion, completely evaluating each tree node (identifying child nodes and edge labels) before it proceeds to the next one. Hence, it is able to apply a more space efficient representation of tree nodes. Furthermore, the child nodes of a particular tree node are stored in contiguous memory locations, resulting in better cache performance during query time. Nevertheless, WOTD exhibits poor temporal locality during the evaluation of tree nodes and it is not designed for CMP architectures.

To demonstrate the poor cache performance of existing in-memory suffix tree construction algorithms, we analyzed the execution time breakdown of two WOTD and CMPUTree on an input string of 100MB (DNA sequence). Using Intel’s VTune performance analyzer, we measured how much time each of these algorithms spends on useful computation and how much time is wasted on stalls like data cache and TLB misses. As illustrated in Figure 5.1, both algorithms spend more than 70-80% of their time on stalls. Hence, there is a significant potential for performance improvement.

![Figure 5.1: Execution time breakdown of two suffix tree construction algorithms.](image)

Other linear time algorithms have also been proposed in [33, 60]. Karkkainen and Ukkonen intro-
duced the notion of sparse suffix trees (SST) to represent a subset of suffixes and proposed a linear time construction algorithm for sparse SSTs [60]. Farach et al. proposed a linear time in-memory suffix tree construction algorithm for integer alphabets [33]. However, these algorithms have not been optimized for cache performance.

5.2.2 Disk-based Algorithms

Existing in-memory algorithms exhibit random access patterns which renders them ill-suited for disk-based scenarios. To address these issues, many suffix tree construction algorithms have been proposed that are optimized for disk performance [12, 23, 37, 53, 82, 100].

Hunt et al. proposed a disk-based suffix tree construction algorithm for biological sequences [53]. Instead of using suffix links, their algorithm performs multiple passes over the input string in order to construct the suffix tree. In every pass, the suffixes with a specific prefix are indexed. Bedathur and Haritsa proposed a low-overhead buffering policy, called TOP-Q, to improve the on-disk behavior of suffix tree construction algorithms [12]. Cheung et al. proposed a top-down disk-based suffix tree construction algorithm, termed DynaCluster, that utilizes a fixed-length prefix-based partitioning technique [23].

A disk-based suffix tree construction algorithm, termed TDD, that utilizes WOTD as building block was proposed by Tian et al. [100]. TDD applies a different buffer replacement policy for every data structure of WOTD. A fixed-length prefix-based partitioning technique is employed to divide the suffix tree into independent sub-trees such that each sub-tree is constructed in memory. A variant of this algorithm, termed ST-merge, is proposed for the case when the input string does not fit in memory.

Phoophakdee and Zaki proposed a disk-based suffix tree construction algorithm that utilizes Ukko nen's algorithm [82]. Their algorithm, termed TRELLIS, deploys a variable-length prefix-based partitioning algorithm to partition the suffixes of the input string into groups such that the suffix tree of each group fits in main memory. The input string is initially divided into a number of consecutive and disjoint substrings and a suffix tree is constructed from each substring in main memory. The sub-trees are subsequently merged into a single suffix tree in a disk optimized way.

Ghoting and Makarychev proposed serial and parallel suffix tree construction algorithms to deal with the case where the input string does not fit in memory [37]. By carefully adapting a simple suffix tree construction algorithm, their algorithm, termed Waterfront, is able to maintain a constant working
set size, effectively reducing the amount of I/O performed. A parallel version is proposed that is tailored to massively parallel shared-nothing architectures.

5.2.3 Parallel Algorithms

Theoretical parallel suffix tree construction algorithms have been proposed in the past [47, 64]. The algorithm in [64] runs in $O(\log n)$ parallel time and uses $n$ processors. The first work-optimal parallel suffix tree construction algorithm was presented in [47]. It does $O(n)$ work and runs in $O(\log^4 n)$ time using $n$ processors. To the best of our knowledge, no practical implementations of these algorithms have been reported.

A parallel algorithm for constructing suffix trees on a computational grid was proposed by Chen and Schmidt [18] in which the suffixes are partitioned into groups using a fixed-length prefix-based partitioning technique. A greedy load balancing algorithm is applied to assign the construction of each suffix tree to a processing node and a suffix tree is constructed from each group utilizing Ukkonen’s algorithm.

Carvalho et al. studied the problem of determining a balanced partition of a lexicographic trie in the context of parallelizing the extraction of structured motifs and proposed an approximate partition algorithm with explicit load balancing guarantees [17]. However, their algorithm cannot be utilized for cache-partitioning as it does not provide any guarantees with respect to the size of the partitions produced.

5.3 Background

In this section, we present a general description of suffix trees. We also provide a high level description of Ukkonen’s algorithm as it is used as building block in CMPUTree (Section 5.5).

5.3.1 Suffix Trees

Let $\Sigma$ be an alphabet of $|\Sigma|$ symbols. We denote $S = s_1, \ldots, s_n$ to be a string over $\Sigma$ of length $n \geq 1$, where $s_i \in \Sigma$ and $\$ is a terminating symbol not occurring in $S$. For any $i \in [1, n]$, $S_i = s_i \ldots s_n$ denotes the $i$-th suffix of $S$ ($S_1 = S$). A suffix tree of $S$, $ST(S)$, is a rooted tree with the following properties.
Every node, except from the root, has at least two edges and every edge has a label representing a substring of $S$. For every node, each of its edges starts with a different symbol from $\Sigma$. For every node $u$, $p(u)$ denotes the path from the root node to $u$ and $\overline{u}$ is the concatenation of edge labels on the path from the root to $u$. Every leaf node corresponds to a suffix of $S$. Finally, a pointer from a node $u$ to another node $v$ is called suffix link if $\overline{u} = \alpha x$ and $\overline{v} = x$, where $\alpha \in \Sigma$ and $x$ is a substring of $S$. The suffix tree of string $BANANA$ is presented in Figure 5.2.

![Figure 5.2: Suffix tree of string BANANA$.](image)

### 5.3.2 Ukkonen’s Algorithm

Ukkonen’s algorithm constructs the suffix tree of a string $S$ in $n$ phases, where $n$ is the size of $S$ [102]. It processes the input string from left to right and during each phase it expands the partially constructed suffix tree by considering a larger prefix of $S$. Algorithm 6 presents a high-level description of Ukkonen’s algorithm.

The time complexity of Algorithm 6 is $O(n^3)$. Ukkonen made several observations that enabled the reduction of time complexity to $O(n)$. In this section, we discuss those that are of interest to this work. The first observation was that edge labels do not have to be explicitly stored in a suffix tree. Instead, a pair of string indices is sufficient for representing the corresponding substring. The indices specify the beginning and end position of a substring in $S$. 
Algorithm 6 Ukkonen’s Algorithm

**Input:** The input string \( S = s_1, \ldots, s_n \) of size \( n \)

**Output:** The suffix tree of \( S \)

1: Initialize an empty suffix tree.
2: for \( i \) from 1 to \( n - 1 \) do
3:   Begin phase \( i + 1 \).
4:     for \( j \) from 1 to \( i + 1 \) do
5:       Locate in the current tree the path with the longest common prefix with substring \( s_j, \ldots, s_i \).
6:       if symbol \( s_{i+1} \) is not already present at the end of that path then
7:         “extend” the path by adding symbol \( s_{i+1} \) to ensure that substring \( s_j, \ldots, s_{i+1} \) is in the tree.
8:     end if
9:   end for
10: end for

Secondly, Ukkonen observed that traversing the partially constructed suffix tree to locate the path that has the longest common prefix with the currently examined substring is a frequent and expensive operation. Ukkonen utilized suffix links to reduce the number of edges examined in this step and consequently, the algorithm’s time complexity.

Despite its linear time complexity, Ukkonen’s algorithm has a number of major limitations in practice. Firstly, it exhibits poor cache performance which is attributed to two reasons: a) the irregular tree traversal, and b) the implicit representation of edge labels using indices. Secondly, Ukkonen’s algorithm is not space efficient and exhibits poor cache performance during query time. Even with the most space efficient implementations of suffix trees [63], Ukkonen’s algorithm requires at least 20 bytes per symbol on a 32-bit architecture.

### 5.4 Cache-Partitioning

*Partitioning* is the process of dividing data into *partitions* so that a particular task can be performed as a set of sub-tasks, each applied to a different partition. Depending on the role of partitioning, different requirements are imposed. For example, if the goal is to improve disk performance, the requirement is that every sub-task is performed on memory-resident data. Similarly, when the goal is to improve
CHAPTER 5. SUFFIX TREE CONSTRUCTION ALGORITHMS ON CHIP MULTIPROCESSORS

cache performance \((\text{cache-partitioning})\), the working set of each sub-task is required to fit in cache. The working set of each sub-task must fit in the \(1/C\)-th part of the cache when a CMP with \(C\) cores is considered, otherwise the cores start competing for the shared resource.

Cache-partitioning is more challenging than partitioning for disk performance. The performance improvement from reducing disk accesses compensates by a considerable margin the overhead of the applied partitioning technique and the partitioning cost has a minor contribution to overall execution time \([82]\). This is not the case for cache-partitioning. Dividing large datasets into many small (in the order of few MBs) partitions such that all partitioning requirements are met may incur significant overhead that negates the benefits of improved cache performance \([95]\).

In the context of disk-based suffix tree construction algorithms, a prefix-based partitioning technique is typically employed to distribute the suffixes of a string into a number of memory-sized partitions such that every partition contains suffixes with the same prefix; variable-length prefixes are utilized to handle skewed data. As we demonstrate in Section 5.6, these techniques are ill-suited for cache performance as they introduce notable overhead.

5.4.1 \textit{PreCache} Partitioning Algorithm

In this section, we present \textit{PreCache}, a low overhead partitioning algorithm that divides the suffixes of a string into a number of cache-sized prefix-based partitions. \textit{PreCache} is tailored to CMPs and is highly efficient for the following reasons: a) it utilizes all processing elements (cores), b) it exploits the shared L2 cache and the fast inter-processor communication on CMPs to improve cache performance and reduce the demand for memory bandwidth, and c) it employs sampling to reduce partitioning cost.

For a partition \(p\), \(freq(p)\) denotes the number of suffixes of \(p\). Formally, we define the partitioning problem as follows. Given a string \(S\) of size \(n\), compute a set of prefix-based partitions \(P\) such that the following conditions are satisfied: a) every suffix belongs to exactly one partition (\textit{coverage condition}), b) \(\forall P_i \in P : freq(P_i) \leq th \ (\textit{space condition})\), and c) \(\forall P_i \in P\), all the suffixes of \(P_i\) have prefix \(\overline{P_i}\) in common; in this case, partition \(P_i\) corresponds to prefix \(\overline{P_i}\).

The \textit{PreCache} algorithm proceeds in two steps. The partitions of \(P\) as well as their corresponding prefixes are computed in the first step. The partitions are populated with the suffixes of the input string in the second step. In the following sections, we describe these two steps in detail. In Section 5.5, we
also describe how the space condition is set in order to improve the cache performance of suffix tree construction algorithms.

### Computing the Partition Set

Partition expansion is an essential procedure of the PreCache algorithm. Expanding a partition \( p \) that corresponds to a prefix of size \( l \), is the process of replacing \( p \) with \(|\Sigma|\) new – potentially smaller – partitions \( p_1, \ldots, p_{|\Sigma|} \) such that every partition \( p_i \) corresponds to a distinct prefix of size \( l + 1 \). For example, if we assume an alphabet \( \Sigma = \{A, C, G, T\} \), expanding a partition \( p \), where \( p = A \), would result in the replacement of \( p \) with four new partitions \( p_1 = AA \), \( p_2 = AC \), \( p_3 = AG \), \( p_4 = AT \).

Given the expansion procedure, the computation of the prefix-based partitions works as follows (presented in Algorithm 7). PreCache creates an initial set of partitions \( P' \) (line 2) that cover all suffixes of \( S \) and progressively computes the final partition set \( P \) by performing multiple passes over a sample of \( S \) (lines 4-21). In every pass, the partitions that satisfy the space condition are removed from \( P' \) and stored in \( P \) (line 17). The partitions of \( P' \) that do not satisfy the space condition are expanded into \(|\Sigma|\) new partitions which are inserted in \( P' \) (the original partition is discarded from \( P' \)) (line 15) and a new pass over the sample of \( S \) is performed. Algorithm 7 terminates when all the conditions are met, i.e. \( P' \) is an empty set.

A high level description of Algorithm 7 is presented in Figure 5.3. In this particular example, three passes are required to divide the suffixes of the input string into a number of partitions such that the space requirement is satisfied, i.e. the number of suffixes of each partition is less than or equal to two. The numbers next to the prefixes indicate the number of suffixes of each partition. Next, we describe in greater detail the most important implementation aspects of Algorithm 7.

\( P' \) is initialized with the partitions that correspond to all possible prefixes of length \( l \), i.e. the size of \( P' \) is \(|\Sigma|^l\) (line 2). \( P' \) is stored in a hash table and the value of \( l \) is set so that \( P' \) fits in cache. For every partition of \( P' \), the corresponding prefix and a counter are stored in a hash bucket. The address of the hash bucket is determined by applying a hash function on the prefix. Since sampling is utilized, computing the exact number of suffixes of each partition is impossible. Instead, a frequency estimate (\(\text{freq} \)) is computed for each partition of \( P' \) in every pass (lines 5-11). The frequency estimate of a partition is stored in the counter of the corresponding hash bucket.
In every pass, the frequency estimates are computed from a random sample of $S$ that consists of a number of sample units (substrings of $S$), each containing $b$ symbols. We set the value of $b$ to be a multiple of a cache line size (typically 64 or 128 bytes) and we align sample units to cache line boundaries in order to improve spatial locality and memory bandwidth utilization. From every sample unit, we extract all possible substrings of length $l_{cur}$ to probe the hash table that stores $P'$. If a substring is found in the hash table, we increment its counter (lines 7-9). If this is not the case, the substring is ignored. Once all the sample units have been processed, we scale the counter of each partition $p$ by $n/B$ to get a frequency estimate (line 13) and we compare this value with $th$ (threshold) to decide whether to expand $p$ or place it in $P$ (lines 14-18); $B$ is the number of substrings of size $l_{cur}$ in the sample of $S$.

Similar to hash-based aggregation on CMPs, there are multiple ways to execute Algorithm 7 that either introduce significant synchronization overhead (single shared hash table) or increase memory footprint (multiple hash tables) [24]. In this work, we apply a different approach that avoids the overhead of synchronization and does not increase memory requirements at the expense of extra CPU overhead. A single hash table is utilized and its hash buckets are partitioned to cores such that every core updates the frequencies of a disjoint set of hash buckets. The pitfall of this approach is that every core has to process the entire sample of $S$ and discard those substrings that are not hashed to the set of assigned buckets. However, there is a great advantage when this approach is applied to a CMP. Since the same input is
processed by all cores, only one core (the fastest) reads the sample units from memory, operating as a
d prefetcher. The remaining cores exploit the fast inter-processor communication of CMPs to effectively
retrieve the sample units from the cache, thus reducing memory accesses and the demand for memory
bandwidth.

Once all cores have processed the sample of $S$, the partitions that satisfy the space condition are
stored in $P$ while the remaining partitions are expanded. A single core is responsible for initializing the
hash table with the new partitions of $P'$ and for assigning the hash buckets to cores. All cores proceed
to the next phase at the same time, this being the only synchronization point.

### Populating the Partitions

Once $P$ has been computed, its partitions are populated with the suffixes of the input string. A single
pass is performed over $S$ to identify the partition of each suffix. The partitions of $P$ are stored in a hash
table using their prefixes as keys. The payload for each partition is a vector of suffix indices.

To determine the partition of a suffix $s$, the hash table that stores $P$ is probed using at most $l_{\text{max}} - l$
prefixes of $s$ of sizes $l, l + 1, \ldots, l_{\text{max}}$, where $l_{\text{max}}$ is the length of the longest prefix that corresponds to
a partition of $P$. The partition of $s$ is the one corresponding to the smallest of these prefixes for which
a match is found in the hash table; the index of $s$ is stored in the corresponding suffix indices vector.
Populating the partition set requires $\frac{n}{2} \cdot (l_{\text{max}} - l)$ hash probes in expectation.

For the case of populating the partitions on a CMP, the best approach is to range partition $S$ (divide
in consecutive substrings) among cores and deploy a single hash table in conjunction with a synchro-
nization mechanism (mutexes). The “shared input” approach applied in Section 5.4.1 is not optimal in
this case for the following reason. Since $S$ is significantly larger than the size of the shared cache, the
cores cannot effectively coordinate in order to process the same part of the input string simultaneously.
Consequently, they end up competing for the shared cache, resulting in a thrashing behavior. Using
a separate hash table per core was also found to be suboptimal due to the increasing number of TLB
misses, attributed to a considerably larger number of memory addresses accessed (the number of vectors
increase by a factor of $C$).
5.5 Suffix Tree Construction Algorithms for CMP Architectures

In this section, we present CMPUTree and MAPST, two cache-conscious suffix tree construction algorithms tailored to CMP architectures. Initially, we present a framework for exploiting on-chip parallelism that is utilized by both algorithms (Section 5.5.1). Subsequently, CMPUTree and MAPST algorithms are presented in Sections 5.5.2 and 5.5.3, respectively. The description of the aforementioned algorithms focuses on how to improve cache performance.

5.5.1 Exploiting on-chip Parallelism

In this section, we present a framework for constructing a suffix tree on a CMP with $C$ cores. The framework comprises four main phases which are executed in the order presented herein: a) partition phase, b) construction phase, c) merging phase, and d) suffix-links recovery phase. As we demonstrate in Section 5.6, the total cost of constructing a suffix tree depends primarily on the time spent on the construction phase. The last two phases are optional as they affect performance during query time.

Partition Phase

In the partition phase, the suffix tree construction task is divided into a number of sub-tasks such that each sub-task can be executed independently by a single core. In this phase, the PreCache partitioning algorithm (Section 5.4) is utilized to divide the input string into a number of prefix-based partitions such that a suffix tree can be constructed from each partition in cache. Then, a simple scheduling algorithm is employed to assign these partitions to cores. The scheduling algorithm is the best-fit heuristic algorithm for solving the bin-packing problem [105].

Construction Phase

In the construction phase, every core builds a suffix tree from each assigned partition. CMPUTree and MAPST, as well as any other in-memory suffix tree construction algorithm can be utilized as a building block in this phase.
Merging Phase

The merging phase is employed to merge into a single suffix tree the suffix trees that were produced in the previous phase. Recall that every suffix tree corresponds to a prefix-based partition. To merge the suffix trees, the corresponding prefixes are considered to be suffixes of a single string and a suffix tree, $ST'$, is constructed from these suffixes. By the definition of suffix trees, the number of leaf nodes of $ST'$ is equal to the number of partitions and every leaf node corresponds to the root node of a suffix tree. Hence, the merging phase is reduced to the construction of $ST'$.

To execute the merging phase on a CMP with $C$ cores, the prefixes are divided into $\lvert \Sigma \rvert^l$ fixed-length prefix-based partitions, where $l$ is the smallest prefix length satisfying $\lvert \Sigma \rvert^l \geq C$. The partitions are assigned to cores using the same scheduling algorithm as in Section 5.5.1 and every core constructs a part of $ST'$ independently. A single core is responsible for merging these parts and connecting the leaf nodes of $ST'$ to the root nodes of the suffix trees.

Suffix-links Recovery Phase

Since suffix links are needed in many string processing algorithms, we provide an optional suffix links recovery phase. Recovering the suffix links requires a depth-first traversal of the suffix tree. To perform this operation on a CMP, we partition the suffix tree among cores using fixed-length prefix-based partitions and every core computes the corresponding suffix links.

5.5.2 CMPUTree Algorithm

In this section, we present the CMPUTree algorithm, a cache-conscious suffix tree construction algorithm that utilizes Ukkonen’s algorithm as a building block. In order to improve cache performance during the construction of a suffix tree, we need to ensure that the working set of Ukkonen’s algorithm fits in cache. The working set of Ukkonen’s algorithm consists of the partially constructed suffix tree and the substrings referenced by its edges. CMPUTree employs the PreCache algorithm to divide the input string into prefix-based partitions such that each partition contains at most $th$ suffixes. By properly setting the value of $th$, we ensure that all memory accesses are resolved through cache during the construction of a suffix tree. In particular, we set the value of $th$ so that the following condition is
satisfied:

\[ th \cdot (\text{node size} + \text{cache line}) \leq \frac{L2}{C}, \]

where node size is the size of a tree node (it depends on implementation details), cache line is the cache line size, and L2 is the size of the L2 cache.

The product \( th \cdot \text{node size} \) denotes the space requirement of tree nodes and \( th \cdot \text{cache line} \) is the space requirement of edge labels (substrings). We set the space requirement of every referenced substring to be the cache line size for two reasons: a) the cache line is the minimum transfer unit from memory, and b) the length of the longest common prefix and consequently, the length of every edge label is \( O(\log n) \) in expectation [7]. Although, there is no explicit guarantee that all edge references will be resolved through cache, we observed that considering the cache line size as the space overhead of each edge label works well in practice (Section 5.6).

### 5.5.3 MAPST Algorithm

In this section, we present the MAPST algorithm, a cache-conscious suffix tree construction algorithm for CMPs inspired by WOTD. MAPST constructs a suffix tree in a top-down fashion, completely evaluating a tree node before proceeding to the next one. Furthermore, it utilizes the same space efficient representation of tree nodes as in WOTD. Evaluating a tree node is the process of identifying its child nodes and their corresponding edge labels. In particular, evaluating a node \( u \) is performed by processing the set of suffixes that have string \( u \) as their longest common prefix, a procedure that generates a large number of random memory accesses.

To eliminate the accesses to the input string (suffixes) and improve cache performance during the evaluation of tree nodes, MAPST materializes a fixed-length prefix from every suffix. The tree nodes are evaluated using the materialized prefixes without accessing the input string, thus significantly reducing the random memory accesses and the suffix tree construction cost. Different compression techniques are employed by MAPST, according to the type of text data, to reduce the space overhead of materialized prefixes and the CPU overhead for evaluating tree nodes. Compression also reduces the probability of accessing the input string as it enables larger prefixes to be materialized in a specific space budget.

Finally, MAPST utilizes PreCache to ensure that its working set, i.e. the materialized prefixes for constructing the suffix tree of each partition, fits in cache. This is accomplished by setting the value of
such that the following condition is satisfied:

\[ th \cdot \text{prefix.size} \leq \frac{L^2}{C}, \]

where \( \text{prefix.size} \) is the size (in bytes) of each materialized prefix; \( th \) is the number of suffixes of each partition. Next, we describe the MAPST algorithm in detail.

The pseudo-code of the MAPST algorithm is presented in Algorithm 8. We assume that the partitioning phase has already been applied and MAPST is utilized to construct the suffix tree from the suffixes of a particular partition. For the evaluation of the root node, MAPST produces the initial set of compressed prefixes that are stored in array \( \text{Suffixes} \) (line 1). Counting sort is applied to sort these prefixes based on their first symbol (line 2). After the sorting phase, every symbol of the alphabet corresponds to a (possibly empty) group of suffixes, represented as a range of positions in \( \text{Suffixes} \). The group corresponding to symbol \( \alpha \in \Sigma \) is denoted as \( \alpha \)-group.

For example, assume that \( \text{Suffixes} \) contains the following prefixes of length 3: \( [\text{AAA, GTA, CAT, TTT, TTT, AGA, GAC}] \), where \( \Sigma = \{A, C, T, G\} \). For simplicity, we do not present the prefixes in compressed form. After sorting the prefixes based on their first letter, \( \text{Suffixes} \) becomes \( [\text{AAA, AGA, CAT, GAC, GTA, TTT, TTT}] \). Symbol \( A \) corresponds to \( A \)-group (range \([1, 2]\) of prefixes), symbol \( C \) corresponds to \( C \)-group (range \([3, 3]\)), and so on. The \( \alpha \)-groups that are produced during the evaluation of a tree node are stored in a stack (lines 3, 13). A new tree node is created for every \( \alpha \)-group that is extracted from the stack and the algorithm terminates when the stack is empty (lines 4-15).

If an \( \alpha \)-group contains a single prefix, a leaf node is produced (line 7). For instance, when the \( C \)-group (represented by range \([3, 3]\)) is extracted, a leaf node is created to represent the suffix that corresponds to prefix \( \text{CAT} \). An internal node of the suffix tree is produced for every extracted \( \alpha \)-group (\( \alpha \)-group for simplicity) that contains more than one prefixes (lines 9-13). The edge label of this node is the longest common prefix (lcp) of the corresponding suffixes.

The lcp is efficiently computed using the materialized prefixes, without accessing the input string (line 9). For example, consider processing the group represented by the range \([1, 2]\) of prefixes. In this case, the lcp is computed directly from the cache-resident prefixes, i.e. \( \text{lcp(\text{AAA, AGA}) = A} \). However, this may not always be the case. Consider for example the range \([6, 7]\) of prefixes. In this case, we are only certain that the lcp contains at least substring \( \text{TTT} \). To find the exact lcp, the corresponding suffixes
must be accessed, resulting in at least two random memory accesses. However, by using compression, MAPST materializes larger prefixes in a specific space budget (cache) and reduces the probability of accessing the input string during the evaluation of tree nodes.

Once the \( lcp \) has been computed from a set of prefixes, these prefixes are updated by removing the \( lcp \) (line 11). For example, prefixes \( AAA, AGA \) of range \([1, 2]\) become \( AA, GA \) by removing symbol \( A \). If during that procedure a prefix is exhausted (no more symbols left), a random memory access is performed to the input string to update the materialized prefix. Again, the merit of using compression is that it reduces the probability of exhausting the materialized prefixes. Counting sort is employed next to sort the resulting prefixes and distribute them to new groups which are inserted in stack (lines 12-13).

Although compression significantly reduces the tree node evaluation cost, it also introduces significant CPU overhead during decompression. Next, we describe two variants of MAPST that leverage the “internals” of different compression techniques to effectively reduce the tree node evaluation cost for different types of text data. The following compression techniques are considered: a) run-length encoding (RLE), and b) Lempel-Ziv-Welch (LZW). RLE is appropriate for strings with many sequences of repeated symbols such as biological data. However, for other types of text data that do not have this property, such as English texts, a directory based compression technique such as LZW is more suitable. Both compression techniques are lossless and require a single pass over the input string.

**Using RLE**

An RLE-character is a \((\alpha, n)\) pair, denoting that symbol \( \alpha \in \Sigma \) appears \( n \) consecutive times. For every suffix \( s \) of an \( \alpha \)-group we materialize the following fields: a) a compressed prefix of \( s \) that consists of \( k \) RLE-characters, where \( k \) is a user defined parameter, and b) a pointer to \( s \) in the input string. For example, for the first suffix \((S_1)\) of string \( AAACCCCCAGG \) we would store the pair \((A3C5, 1)\), assuming \( k = 2 \). Larger values of \( k \) enable the evaluation of more tree nodes without accessing the input string. However, the space requirements for storing an \( \alpha \)-group increase with \( k \). We demonstrate these tradeoffs in Section 5.6.

Next, we describe how the RLE encoding is leveraged to reduce the cost of computing the \( lcp \) of an \( \alpha \)-group. For every \( \alpha \)-group \( g \), we explicitly store the maximum \((max_n)\) and minimum \((min_n)\) \( n \) of the first RLE-character of every suffix of \( g \). We consider three cases with respect to the values of these two
variables:

- Case 1: $\text{max}_n > \text{min}_n$. This is the best case, as we can automatically infer at a cost of a single arithmetic comparison that the lcp of $g$ is the single symbol $\alpha$.

- Case 2: $\text{max}_n = \text{min}_n > 1$. We know that $\text{lpc} \geq \alpha\alpha\ldots\alpha$ (symbol $\alpha$ appears at least $\text{max}_n$ times). In this case, we need to examine the materialized prefixes of $g$ to determine the exact lcp. However, we do not need to consider the first $\text{max}_f$ symbols of each prefix since we already know that they are part of the lcp.

- Case 3: $\text{max}_n = \text{min}_n = 1$. This is the worst case, as we don’t have any information regarding the lcp and we must examine the prefixes of $g$ exhaustively.

In all but the first case, we have to examine the compressed prefixes in order to compute the lcp of $g$. However, this procedure is highly efficient for the following reasons: a) the compressed prefixes reside in cache, b) the prefixes are stored in contiguous memory locations and are accessed sequentially, and c) the comparisons are performed on RLE-characters, thereby reducing the computational cost. The above are verified in Section 5.6.

Once the lcp of an $\alpha$-group has been computed, the prefixes are updated by removing the lcp and then sorted using counting sort. Both operations can be performed directly on compressed prefixes.

Using LZW

LZW is an efficient dictionary-based compression technique. Commonly, the dictionary, the size of which is user defined, is stored in an LZW-trie, where every trie node corresponds to a dictionary entry. We refer to a dictionary entry as LZW-code. When LZW is employed in MAPST, for every suffix of an $\alpha$-group we store the following: a) a compressed prefix that consists of $k$ LZW-codes, and b) a pointer to the position of the suffix in the input string.

Computing the lcp and sorting the suffixes of an $\alpha$-group is performed at low cost by accessing the LZW-trie. In MAPST, we set the size of the dictionary to be significantly smaller than the cache (in the order of few KBs). The small size of the LZW-trie and the fact that it is frequently accessed, increase the probability of it residing in cache during the construction of the suffix tree.
Algorithm 9 presents the pseudo-code for computing the $lcp$ of an $\alpha$-group. The $lcp$ is initialized with the first prefix (line 1) and is progressively refined as more prefixes are processed (line 3). If the $lcp$ is reduced to a single symbol during this refinement, Algorithm 9 terminates (line 5).

We illustrate Algorithm 9 with the following example. Assume that the $lcp$ is initialized with compressed prefix $P_1 = C_1 C_2 C_3$, where $C_1, C_2, C_3$ are dictionary entries and consider the process of refining the $lcp$ by computing the longest common prefix between the current value of $lcp (P_1)$ and the second prefix $P_2 = C'_1 C'_2 C'_3$. To compute the $lcp$ between any pair of compressed prefixes, we initially inspect their code values to efficiently identify a common prefix. Subsequently, we compute the exact longest common prefix by accessing the LZW-trie.

![Figure 5.4: LZW-trie](image)

If we assume in our example that $C_1 = C'_1$ and $C_2 \neq C'_2$, the $lcp$ between $P_1$ and $P_2$ contains at least $C_1$ and thus, we only need to compute the $lcp$ between $C_2$ and $C'_2$. Notice, that we do not need to consider codes $C_3$ and $C'_3$. Since every code corresponds to a trie node, computing the longest common prefix is equivalent to computing the lowest common ancestor ($lca$) of the corresponding trie nodes. Hence, we employ a constant-time $lca$ retrieval algorithm, proposed in [92].

In our example, the $lcp$ between $C_2$ and $C'_2$ is the trie node that corresponds to code $C'$ (Figure 5.4) and the new value of $lcp$ is $C_1 C'$. The same refinement procedure is then applied to the remaining prefixes as illustrated in Algorithm 9 (line 2) until either the $lcp$ contains a single symbol or all the prefixes have been processed. If at some point a prefix is exhausted, the corresponding suffix is accessed...
and a new compressed prefix is produced. Once the \( lcp \) has been computed, the prefixes are updated by removing the \( lcp \) and they are sorted using counting sort. All these operations are performed at low cost by accessing the LZW-trie which with high probability resides in cache.

5.6 Experimental Evaluation

In this section, we present the experimental evaluation of the proposed suffix tree construction algorithms using real text corpora. In Section 5.6.1, we describe implementation details of our algorithms, the datasets used to assess their performance, and the hardware configuration of the CMP on which we run our experiments. The results are presented in Sections 5.6.2 and 5.6.3.

5.6.1 Experimental Setting

We compare the two suffix tree construction algorithms proposed (\textit{MAPST}, \textit{CMPUTree}) against \textit{WOTD} and Ukkonen’s algorithm, respectively. Implementations of the latter were retrieved from publicly available sources\(^1\). Initially, the implementation of Ukkonen’s algorithm utilized sibling lists (linked lists) to store the child nodes of each tree node. We altered its implementation using a combination of hash map and sibling lists to reduce the number of random memory accesses during the evaluation of tree nodes.

Two factors affect the cost of constructing a suffix tree besides the size of the input string: a) the alphabet size, and b) the average longest-common-prefix. To demonstrate the effectiveness of the proposed suffix tree construction algorithms on datasets with different characteristics, we selected two text collections from Ferragina and Navarro’s \textit{Pizza\&Chilli corpus}\(^3\). The first text collection has small alphabet size (16 symbols) and contains gene DNA sequences from the Human Genome Project. The second collection has a relatively larger alphabet size (239 symbols) and is a concatenation of English text files from the Gutenberg Project.\(^4\) The characteristics of the selected collections are summarized in Table 5.1.

\(^1\)http://www.cs.ucdavis.edu/gusfield/strmat.html
\(^2\)http://bibiserv.techfak.uni-bielefeld.de/wotd
\(^3\)http://pizzachili.dcc.uchile.cl/index.html
\(^4\)http://www.gutenberg.org/wiki/Main_Page
All the experiments were conducted on a Dell PowerEdge 2950 server with two quad-core Intel E5355 CPUs (8 cores) running at 2.6GHz. Each quad-core has a 4MB of shared L2 cache (8MB is total), and every core has a private 64KB L1 cache. The server runs the 64-bit Linux operating system with kernel 2.6.15 and has 32GB of main memory. All the algorithms proposed were implemented in C using Posix threads (Pthreads) and the gcc-4.3 compiler was used with optimization level 02. In all the experiments conducted, the `sched_setaffinity` system call of the Linux OS was employed to schedule the threads to specific cores and ensure that they were not re-scheduled during execution.

### 5.6.2 Improving Cache Performance

A set of experiments is presented below to demonstrate that the proposed cache-aware algorithms exhibit better cache performance and run faster than their cache-unaware counterparts. In all the experiments presented in this section, a single core was utilized to construct a suffix tree. The wall-clock time required to construct and store the suffix tree in memory was measured in all the experiments; the input string was preloaded in memory. Intel’s VTune performance analyzer was utilized to measure performance counters such as TLB and L2 cache misses. We also report the memory usage of the proposed algorithms.

#### Evaluation of CMPUTree

Initially, we compare the performance of CMPUTree against that of Ukkonen’s algorithm. We consider two variants of the CMPUTree algorithm, namely CMPUTree_E and CMPUTree_PC. CMPUTree_PC utilizes the PreCache partitioning algorithm (Section 5.4). In contrast, CMPUTree_E employs the variable length, prefix-based partitioning technique proposed in [82]. The latter is an exact partitioning technique that performs multiple passes over the input string. It has been utilized in several disk-based suffix tree construction algorithms to effectively partition skewed datasets such as DNA sequences.
Since the suffix links recovery phase is optional, we also consider a variant that does not apply this phase, termed \( nsl \) (no suffix links).

In Figure 5.5, we present the suffix tree construction time of these algorithms as we increase the size of a DNA sequence from 20MB to 80MB. The memory usage of \( CMPUTree \) is reported in Table 5.2.

For the entire range of string sizes examined, \( CMPUTree_{PC} \) performs \( 1.5X - 1.6X \) faster than Ukkonen’s algorithm and \( 1.2X - 1.7X \) faster than \( CMPUTree_{CE} \). For small string sizes, the overhead of the partitioning technique is small, and \( CMPUTree_{CE} \) performs better than Ukkonen’s algorithm. However, as the string size becomes larger, the improved cache performance of the suffix tree construction phase does not compensate for the overhead introduced by the partitioning algorithm and \( CMPUTree_{CE} \) performs worse than Ukkonen’s algorithm (when the input string is 80MB in our experiment). In contrast, \( CMPUTree_{PC} \) consistently outperforms both \( CMPUTree_{CE} \) and Ukkonen’s algorithm, verifying its improved cache performance and the effectiveness of our low overhead partitioning algorithm.

<table>
<thead>
<tr>
<th>String Size (MB)</th>
<th>Memory Usage (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>2.1</td>
</tr>
<tr>
<td>40</td>
<td>4.3</td>
</tr>
<tr>
<td>60</td>
<td>6.7</td>
</tr>
<tr>
<td>80</td>
<td>8.9</td>
</tr>
</tbody>
</table>

The same experiment was conducted using English text data (Figure 5.6). As in the case of DNA sequences, \( CMPUTree_{PC} \) consistently performs \( 1.3X - 1.5X \) faster than both Ukkonen’s algorithm and \( CMPUTree_{CE} \). The performance difference between \( CMPUTree \) and Ukkonen’s algorithm is smaller compared to the case of biological text data for a given input string size. Due to the larger average LCP of the English text collection, \( CMPUTree \) underestimates the space overhead of some edge labels, resulting in higher number of random memory accesses while traversing a partially constructed suffix tree.

In the next experiment, the goal is to study the tradeoffs in using a sample-based partitioning technique. Figure 5.7 presents the execution time breakdown of \( CMPUTree_{CE} \) and \( CMPUTree_{PC} \) algorithms on different phases when the input string size is 80MBs. The cost of the merging phase is not reported as it is negligible in this experiment. The sample-based partitioning technique clearly reduces
the partitioning cost. However, since it relies on estimates of the number of suffixes of each partition, it typically produces more partitions (depending on the sample size) than the exact method (applied in \texttt{CMPUTree}$_E$). Although, the larger number of partitions increases the time spend on the remaining phases (construction, suffix link recovery, merging), the reduction of the partitioning cost compensates
for the overhead introduced.

![Figure 5.7: Execution time breakdown of CMPUTree_E and CMPUTree_PC algorithms when the input string size is 80MBs.](image)

**Evaluation of MAPST**

The performance of the MAPST algorithm is assessed next. The suffix tree construction time of MAPST and WOTD is compared for biological and English text data. The results are presented in Figures 5.8 and 5.9, respectively. The memory usage of MAPST is reported in Table 5.3. For small string sizes (20MB in our experiment) MAPST and WOTD exhibit the same performance (Figure 5.8). However, as the string size increases from 20MB to 100MBs, their performance difference grows from 1% to 43%, respectively. The working set of the WOTD algorithm decreases as the construction of the suffix tree moves downwards, i.e. as lower nodes of the suffix tree are being evaluated. Hence, for the case of small strings, it takes only few node evaluations for the working set to fit in cache. As the string size increases, more tree nodes are evaluated by accessing memory resident data. Consequently, the performance difference between WOTD and MAPST is amplified, as the latter ensures that every node evaluation is performed in cache by utilizing PreCache and the compressed materialized prefixes.

For the case of English text data (Figure 5.9), the performance difference between WOTD and MAPST also escalates as we increase the string size, but at a lower pace compared to the case of bi-
\begin{table}
\centering
\caption{Memory Usage of MAPST.}
\begin{tabular}{|c|c|}
\hline
String Size (MB) & Memory Usage (GB) \\
\hline
20 & 0.8 \\
40 & 1.7 \\
60 & 2.7 \\
80 & 3.6 \\
\hline
\end{tabular}
\end{table}

Figure 5.8: Evaluation of MAPST for biological text data.

ological data. For small strings (20MB in our experiment) WOTD is 20\% faster than MAPST and when the string size is 100MBs, the latter is 27\% faster. Compared to the case of biological data, the smaller performance difference is attributed to two reasons: a) the considerably larger alphabet size of English text, and b) the relatively higher overhead of LZW compared to RLE. The larger the alphabet size, the faster the reduction of the working set size (factor of $|\Sigma|$) and consequently, fewer node evaluations are required for the working set of WOTD to fit in cache (for a given string size).

In the next experiment, we study the effect of prefix size on the performance of MAPST. Figure 5.10 presents the time required to construct a suffix tree from a DNA sequence of size 100MBs as we vary the number of RLE codes ($k$) of each materialized prefix. As we increase the prefix size to 8 codes, more nodes are evaluated from the materialized prefixes before they are exhausted, thus improving the cache performance of MAPST. However, as we increase the value of $k$, MAPST performs worse. Larger
values of \( k \) increase the space requirements of each partition, the number of partitions created and the compression cost.

In order to properly set the value of \( k \), we must have explicit knowledge of the characteristics (distribution of the lcp) of a text collection. If that knowledge is available, we can compute the value of \( k \) that minimizes the expected number of memory accesses. If this is not the case, sampling could be utilized to provide this information.
CHAPTER 5. SUFFIX TREE CONSTRUCTION ALGORITHMS ON CHIP MULTIPROCESSORS

Under the Microscope

In all the aforementioned experiments, the wall-clock time required to construct a suffix tree is measured and it is demonstrated that the proposed cache-conscious algorithms perform better than their cache-unaware counterparts. Next, we use hardware performance counters to corroborate that this performance improvement is attributed to the reduction of cycles wasted on various stalls and to the improved CPU utilization.

In Figure 5.11, we present the normalized execution time breakdown (in cycles) of all the algorithms that were examined in our experimental evaluation for processing a DNA sequence of size 100MBs. It can be seen that all cache-aware algorithms increase the amount of time spent on useful computation and reduce the number of cycles wasted on stalls. CMPUTree reduced the L2 cache misses by 63% and the TLB misses by 80%. The wasted cycles accounted for the 82% of the total execution time for the case of Ukkonen’s algorithm and for the 49% for the case of the CMPUTree algorithm. Significant reduction of the number of wasted cycles is also reported for the case of the MAPST algorithm.

![Execution Time Breakdown](image)

Figure 5.11: Execution time breakdown of suffix tree construction algorithms.

5.6.3 Exploiting on-chip Parallelism

In this section, we study the ability of the proposed algorithms to exploit the processing elements (cores) of a CMP. In Figures 5.12 and 5.13, we demonstrate the performance of CMPUTree and MAPST, re-
spectively for processing a DNA sequence of size 100MBs as we increase the number of utilized cores. We present the normalized execution time with respect to the case when a single core is utilized.

![Speedup of CMPUTree](image)

Figure 5.12: Speedup of CMPUTree as we increase the number of cores utilized.

As illustrated in Figures 5.12 and 5.13, the proposed algorithms effectively utilize the computational power of CMPs, significantly reducing the suffix tree construction cost. In both cases, the execution time is reduced by a factor of 6 when eight cores are utilized (in our experiment). The non-linear speedup is attributed to the approximate partitioning technique and the online scheduling algorithm applied to assign the partitions to cores. This experiment demonstrates the effectiveness of the PreCache partitioning algorithm in parallelizing the suffix tree construction task. This experiment was also conducted with English text data with the same results, verifying that algorithms designed for CMP architectures can significantly reduce the suffix trees construction cost.

### 5.7 Conclusions

In this chapter, we studied the problem of improving the performance of in-memory suffix tree construction algorithms. Initially, we proposed PreCache, a low-overhead cache partitioning algorithm that is utilized as a building block to improve the cache performance and parallelize the suffix tree construction task. We proposed, CMPUTree and MAPST, two novel suffix tree construction algorithms that are
Figure 5.13: Speedup of MAPST as we increase the number of cores utilized.

tailored to CMP architectures. Through a detailed experimental evaluation, we demonstrated that the algorithms proposed exhibit improved cache performance and effectively utilize the computational power of CMP architectures, thus achieving very good speedup.
Algorithm 7 PreCache - Partition Set Computation

**Input:** $S$: string, $l$: initial prefix length, $th$: threshold, $B$: number of substrings of length $l_{cur}$ in the sample of $S$

**Output:** $P$: partition set

1. Set $l_{cur} \leftarrow l$.
2. Initialize a partition set $P'$ with $|\Sigma|^l$ partitions.
3. Compute a sample of $S$. // The sample units are substrings of $S$.
4. **while** $P'$ is not empty **do**
   5. **for** each sample unit $s$ **do**
      6. **for** every substring $sb$ of $s$ of length $l_{cur}$ **do**
         7. **if** $sb \in P'$ **then**
            8. Increment the counter of the corresponding partition.
         9. **end if**
   10. **end for**
   11. **end for**
   12. **for** every partition $p$ of $P'$ **do**
       13. Set $\overline{\text{freq}}(p) = \text{count}(p) \cdot n/B$
       14. **if** $\overline{\text{freq}}(p) > th$ **then**
           15. Expand $p$ into $p_1, \ldots, p_{|\Sigma|}$ and insert them into $P'$.
       16. **else**
           17. Remove $p$ from $P'$ and insert it into $P$.
       18. **end if**
   19. **end for**
   20. Set $l_{cur} = l_{cur} + 1$. // Increment $l_{cur}$
   21. **end while**
Algorithm 8 MAPST Algorithm

**Input:** \( S \): input string, \( p = \{s_1, \ldots, s_{th}\} \): partition with \( th \) suffixes, \( \text{Suffixes} \), \( \text{Temp} \): arrays of materialized prefixes, \( Z \): compression technique (RLE or LZW), \( \Sigma \): alphabet

**Output:** \( ST \): suffix tree of the suffixes of \( p \)

1. For every suffix of \( p \), produce a compressed prefix using \( Z \) and store it in \( \text{Suffixes} \).
2. Sort the compressed prefixes using the first symbol of each prefix as key. The sorted prefixes are stored in \( \text{Suffixes} \) and every symbol of \( \Sigma \) corresponds to an \( \alpha \)-group (range of prefixes in \( \text{Suffixes} \)).
3. Store every \( \alpha \)-group in a stack.
4. while the stack is not empty do
   5. Retrieve the next \( \alpha \)-group (range \( r = [i,j] \) of compressed prefixes) from the stack.
   6. if \( r \) contains a single prefix \( (j = i) \) then
      7. Generate a new leaf node in \( ST \).
   8. else
   9. Compute the \( lcp \) of the suffixes corresponding to the compressed prefixes that are referenced by \( r \).
   10. Create a new node in \( ST \) and use the \( lcp \) as the edge label.
   11. Update the compressed prefixes of \( r \) by removing the \( lcp \).
   12. Apply counting sort on the updated prefixes of \( r \).
   13. Store the \( \alpha \)-groups that correspond to the symbols of \( \Sigma \) in stack.
   14. end if
5. end while

Algorithm 9 Algorithm for computing the \( lcp \)

**Input:** \( g \): \( \alpha \)-group, \( m \): number of suffixes represented in \( g \), \( P_1, \ldots, P_m \): compressed prefixes of \( g \)

**Output:** \( lcp \) of the suffixes of \( g \)

1. \( lcp \leftarrow P_1 \)
2. for \( i = 2, \ldots, m \) do
3. \( lcp \leftarrow \text{compute}_lcp(lcp, P_i) \)
4. if \( \text{size}(lcp) == 1 \) then
   5. Terminate
6. end if
7. end for
Chapter 6

Query Processing Techniques for Solid State Drives

6.1 Introduction

Solid state drives perform random reads more than 100x faster than traditional magnetic hard disks, while offering comparable sequential read and write bandwidth. Because of their potential to speed up applications, as well as their reduced power consumption, these new drives are expected to gradually replace hard disks as the primary permanent storage media in large data centers. However, database query processing engines are tailored to the performance characteristics of magnetic disk drives and their algorithms currently emphasize sequential accesses for disk-resident data.

We propose query processing techniques that leverage the performance characteristics of flash-based storage systems (solid state drives). We present and evaluate data structures and algorithms that leverage fast random reads to speed up selection, projection, and join operations. Along with sort (which can benefit from SSDs without algorithmic changes [67]) and aggregation (which typically applies to in-memory data or to the output of scan or join operators and therefore cannot benefit from SSDs), these operations are the most common in complex query plans.

Initially, we consider using a column-based layout within each database page, such as PAX [3]. While PAX was originally proposed to improve CPU cache performance, we demonstrate it can reduce the amount of data read from SSDs. Using a PAX-based page layout, we propose FlashScan, a scan
operator that reads from the SSD only those attributes that participate in a query. FlashScan proactively evaluates predicates before fetching additional attributes from a given row, thus further reducing the amount of data read when few tuples are selected.

Building on FlashScan’s ability to efficiently extract needed attributes, we introduce FlashJoin, a general pipelined join algorithm that minimizes accesses to relation pages by retrieving only required attributes, as late as possible. FlashJoin consists of a binary join kernel and a separate fetch kernel. Multiway joins are implemented as a series of two-way pipelined joins. The join kernel accesses only the join attributes, producing partial results in the form of a join index for each join node. Our current implementation uses a hash-based join kernel that employs the hybrid-hash join algorithm [94]; any other join algorithm may be used instead. Subsequently, FlashJoin’s fetch kernel retrieves the attributes for later nodes in the query plan as they are needed, using different fetching algorithms depending on the join selectivity and available memory. We show that FlashJoin significantly reduces the amount of memory and I/O needed for each join in the query.

Finally, we demonstrate how the proposed techniques can be incorporated inside PostgreSQL, a full-featured database system. Through a detailed experimental evaluation using an enterprise SSD, we demonstrate that our techniques are able to speed up queries, ranging from simple scans to multiway joins and full TPC-H queries, by up to 6X.

6.2 SSD Trends and Related Work

6.2.1 SSD Characteristics, Costs, and Trends

Most current commercial SSDs use NAND Flash for bulk data storage. NAND flash is a purely electronic, non-volatile store whose performance and price characteristics put it between DRAM and traditional disks. Table 6.1 summarizes the relevant characteristics of current Flash SSDs compared to traditional magnetic hard disks. All of the SSDs provide orders of magnitude (10-100x) faster random read IO/s than traditional drives and comparable sequential read and write bandwidth. Their random write performance, however, is much worse than read, especially on the consumer-grade drives. Finally, SSDs well outperform traditional drives on the price-performance metric IO/s/$ (random IO rate per dollar) and the energy-efficiency metric IO/s/W (random IO rate per Watt consumed).
The asymmetry between read and write performance is due to the underlying technology. NAND flash is organized into large 128K erase blocks while read and write IOs are to 4K pages. However, a 4K page write succeeds only if the page has been previously erased. Erasing is expensive in terms of both time and power consumed. Thus, a naive random write requires an expensive 128K read, erase, and rewrite operation. Further, most NAND flash is limited to about 100,000 erase-write cycles per block.

Fortunately, SSDs embed logic to hide these details. All SSDs include wear-leveling logic that remaps writes to evenly update all blocks. With wear leveling, writing continuously to a 32GB drive at 40 MB/s would cause the drive to wear-out after 2.5 years, which implies an acceptable lifespan of 5-10 years with average utilization. Moreover, as Table 6.1 shows, enterprise SSD vendors are improving random write performance by overprovisioning the underlying flash capacity and embedding additional logic in the drive. Enterprise SSDs have so far been much more expensive than consumer SSDs, but these cost differences are shrinking as enterprise SSD volumes increase.

Overall, flash SSDs are seeing an annual $/GB decline of 50% per year [58], which is a faster drop rate than for hard disks. We expect flash SSDs to eventually be competitive with hard disks in terms of $/GB and continue to outperform them by orders of magnitude in read and write IO/s/$ and IO/s/W. Thus, we expect that for many data analysis applications, SSDs will replace hard disks in the future.

6.2.2 Databases and SSDs

Several recent studies have measured the read and write performance of flash SSDs [16, 77, 83]. uFLIP [16] defines a benchmark for measuring sequential and random read and write performance and presents results for 11 different devices. Of particular note, they identify a “startup” phase where random writes may be cheaper on a clean SSD (since no blocks need to be erased) but quickly degrade as the disk fills. Polte et al. perform a similar study using less sophisticated benchmarks, but focus on the behavior of filesystems running on top of SSDs. They show the degraded mode is 3-7X worse than the “startup” phase but still an order of magnitude faster than any current HDD [83].

Graefe [40] reconsiders the trade-off between keeping data in RAM and retrieving it as needed from non-volative storage in the context of a three-level memory hierarchy in which flash memory is positioned between RAM and disk. The cited paper recommends disk pages of 256KB and flash pages of 4KB to maximize B-tree utility per I/O time. Interestingly, these page sizes derive retention times of
Table 6.1: Disk and Flash characteristics from manufacturer specs or as measured where possible. Prices from online retailers as of Nov 25, 2008. SATA-disk: Seagate Barracuda; SATA-Flash: Mtron; FC-Flash: STech’s ZeusIOps 3.5” FibreChannel; ioD: FusionIO ioDrive.

<table>
<thead>
<tr>
<th>GB</th>
<th>SATA Disk</th>
<th>SATA Flash</th>
<th>FC Flash</th>
<th>ioD Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>$/GB</td>
<td>$0.12</td>
<td>$15.62</td>
<td>$85</td>
<td>$30</td>
</tr>
<tr>
<td>Watts (W)</td>
<td>13</td>
<td>2</td>
<td>8.4</td>
<td>6</td>
</tr>
<tr>
<td>seq. read (MB/s)</td>
<td>60</td>
<td>80</td>
<td>92</td>
<td>700</td>
</tr>
<tr>
<td>seq. write (MB/s)</td>
<td>55</td>
<td>100</td>
<td>108</td>
<td>500</td>
</tr>
<tr>
<td>ran. read (IO/s)</td>
<td>120</td>
<td>11,200</td>
<td>54,000</td>
<td>79,000</td>
</tr>
<tr>
<td>ran. write (IO/s)</td>
<td>120</td>
<td>9,600</td>
<td>15,000</td>
<td>60,000</td>
</tr>
<tr>
<td>IO/s/$</td>
<td>2.0</td>
<td>11.2</td>
<td>4.4</td>
<td>8.3</td>
</tr>
<tr>
<td>IO/s/W</td>
<td>9.2</td>
<td>5,600</td>
<td>6,430</td>
<td>13,166</td>
</tr>
</tbody>
</table>
about 5 minutes, reinforcing the various forms of the “five-minute rule”.

Both Myers [77] and Lee et al. [67] measure the performance of unmodified database algorithms when the underlying storage is a flash SSD. Myers considers B-tree search, hash join, index-nested-loops join, and sort-merge join, while Lee et al. focus on using the flash SSD for logging, sorting, and joins. Both conclude that using flash SSDs provides better performance than using hard disks. For example, fast random reads allow a larger fan-in during the merge phase of sorting, thus increasing the maximum size relation that can be sorted in two passes.

Then, there are a set of papers that propose new database algorithms designed especially for flash characteristics: these algorithms generally emphasize random reads and avoid random writes (where traditional algorithms stress sequential reads and writes and try to avoid any random I/O). Lee et al. [66] modify database page layout to make writing and logging more efficient. Ross [91] proposes new algorithms for counting, linked lists, and B-trees that minimize writes. Shah et al. [93] present a new hash-based join algorithm that, in combination with a new page layout, uses random reads to retrieve less data than hybrid hash join. However, their algorithm focuses on binary joins and there is no implementation of the proposed join algorithm. Nath and Gibbons [78] define a new data structure, the B-file, for maintaining large samples of tables dynamically. Their algorithm writes only completely new pages to the flash and they observe that writes of pages to different blocks may be interleaved efficiently on flash SSDs. Li et al. [69] propose a new index structure: it uses a B-tree in memory to absorb writes and then several levels of sorted runs of the data underneath the tree. This structure uses only sequential writes to periodically merge the runs and random reads to traverse the levels during a search.

Finally, Koltsidas and Viglas [61] consider hybrid SSD/HDD configurations of database systems and design a buffer manager that dynamically decides whether to store each page on a flash SSD or a hard disk, based on its read and write access patterns. However, their approach assumes that all page accesses are random.

### 6.2.3 Related Query Processing Techniques

Our aim is to modify traditional query processing techniques to leverage the fast random reads of flash SSDs. Our modifications have three goals: (1) avoid reading unnecessary attributes during scan selections and projections, (2) reduce I/O requirements during join computations by minimizing passes over
participating tables, and (3) minimize the I/O needed to fetch attributes for the query result (or any intermediate node in the query plan) by performing the fetching operation as late as possible. We discuss related techniques here and revisit some of those techniques in later sections.

Traditionally, database systems use the N-ary storage model (NSM), a page-based storage layout in which tuples (or rows) are stored contiguously in pages. NSM may waste disk and memory bandwidth if only a small fraction of each row is needed. In contrast, the decomposition storage model (DSM) [26], proposed in the 80s, decomposes relations vertically, allocating one sub-relation per attribute. DSM had its own disadvantages, including storage overhead for storing tuple IDs and expensive tuple reconstruction costs. With changing market needs and more favorable technology trends, newer DSM-like (column-store) commercial products and academic prototypes have recently appeared (such as SybaseIQ, Vertica, C-store [98], and MonetDB/X100 [15]). These systems eliminate storage overhead through virtual IDs and offer fast scans of few attributes [50] at the cost of additional disk seeks to fetch non-contiguous attributes.

PAX [3] (Partition Attribute Across) is a hybrid approach, essentially a DSM-like organization within an NSM page. While the disk access pattern of PAX is indistinguishable from that of NSM, it improves on the memory bandwidth requirements. On SSDs, as we show in the next section, PAX, in combination with the much faster seek time, allows reading only those columns needed by the query, essentially enjoying the read efficiency of DSM while retaining existing NSM functionality.

The ability of column-stores to read only part of a tuple led to an examination of different tuple materialization strategies [2]. A late materialization policy is particularly effective in reducing the amount of data passed through the query operators and we discuss its use in FlashJoin in Section 6.4.3. Late materialization was originally implemented by Semijoin reducers [97] which first computed a semi-join (using an index, if one was available) and then revisited base relations to obtain remaining attributes, without needing to redo predicate evaluation. Late materialization was also the essence behind TID hash joins [74], which compute a join index first and then construct the final join result. Although TID hash joins waste disk bandwidth when reading the input relations and rely on an inefficient fetching strategy for constructing the final result, they were shown to outperform traditional hash joins for highly selective joins on large inputs. A more efficient fetching strategy, given a pre-existing join index, was described for Jive/Slam-joins [70]. However, these algorithms were limited to two-way non-pipelined joins.
Our work builds on these ideas, re-examining fundamental tradeoffs in light of the different performance characteristics of SSDs.

6.3 Scan Selections & Projections

In this section, we demonstrate that a PAX-like page organization is a natural choice for database systems over SSDs. We first describe our implementation of FlashScan, a scan operator that leverages the PAX layout to improve selections and projections on flash SSDs. We evaluate the implementation of FlashScan inside Postgres in Section 6.3.2. Then, in Section 6.3.3 we discuss the similarities and differences between PAX layout and DSM or column layout and describe how FlashScan applies to column-stores.

6.3.1 FlashScan Operator

Figure 6.1 shows the page organization of both NSM and PAX for a relation with four attributes. In general, for an \( n \)-attribute relation, PAX divides each page into \( n \) minipages, where each minipage stores the values of a column contiguously. The size of each minipage is computed based on the average sizes of the attribute values. In our implementation, we do not handle overflowing of minipages caused by variable-length attributes; Ailamaki et al. describe how PAX can support variable-length attributes in [3]. The column-based layout of PAX offers a physical separation of values from different columns, within the same page, thus allowing an operator to access only the attributes that are needed in a query. Because regular scanners access a full page from disk, PAX does not have an impact on disk bandwidth. Once a page is brought into main memory though, PAX allows the CPU to access only the minipages needed by the query, thus reducing memory bandwidth requirements [3]. Data transfer units in main memory can be as small as 32 to 128 bytes (the size of a cacheline), whereas a database page is typically 8K to 128K. With SSDs, the minimum transfer unit is 512 to 2K bytes, which allows us to selectively read only parts of a regular page.

FlashScan takes advantage of the small transfer unit of SSDs to read only the minipages of the attributes that it needs. Consider a scan without selection predicates that projects the first and third column of the relation in Figure 6.1. For each page, FlashScan initially reads the minipage of the first
attribute and then “seeks” to the start of the third minipage and reads it. Then it “seeks” again to the first minipage of the next page. This procedure continues over the entire relation, resulting in a random (albeit strided) access pattern.

In general, every “seek” results in a random read. The only exception is when the scan query requests contiguous minipages; in that case, FlashScan coalesces the reads and performs one random access for each set of contiguous minipages. Using random instead of sequential reads is beneficial only if the storage medium supports fast random access and the reduction in the amount of data read compensates for the overhead induced by the random I/O. As we demonstrate in the next section, this is indeed the case for flash SSDs.

![Figure 6.1: Disk pages in NSM (left) and PAX (right) storage layout of a relation with four attributes.](image)

Note that we cannot expect the minipages to be exact multiples of the SSD minimum transfer unit. We also did not want to align minipages to OS page boundaries since that approach could result in severe fragmentation and poor space utilization. Instead, in our implementation of PAX and FlashScan, we decouple minipage size from SSD transfer units and compute the size of each minipage solely as a function of the page size and the average sizes of attribute values. Inside Postgres, we implemented PAX by dividing every database page into tightly packed minipages. Therefore, some unneeded data may be transferred when reading a single minipage.

We modified the Postgres buffer manager and bulk loader to work with the new page layout. The
buffer manager allocates space and performs replacements at the granularity of a database page. However, a page in the buffer pool may be partially full containing only the minipages transferred by FlashScan. Although this design wastes space in the buffer pool, it simplifies our implementation and ensures that the invocation of the buffer manager does not influence our results in the experimental evaluation. It also allows us to reuse existing Postgres functionality and utilities.

We added FlashScan to Postgres as a new scan operator that produces tuples in row-format, after having read in the buffer pool all minipages containing the projected attributes. One of the features of Postgres that adds a considerable per-tuple overhead, is multi-version concurrency control (MVCC). Currently, our implementation does not support in-place updates (and therefore we have no use of MVCC), as our goal was to evaluate read-only queries. While certain database systems and several data warehousing applications typically operate under alternating periods of read-only queries and bulk-loading data, adding support for updates in PAX, if needed, could be performed without any penalty, as shown in [3]. To ensure a fair comparison with Postgres’ default page layout (NSM) that includes MVCC information, we added the same overhead to our PAX pages by reserving an extra minipage inside each page. Additionally, we removed any calls related to MVCC when NSM is used.

**Optimizations for Selection Predicates**

For selection predicates, FlashScan can improve performance even further by reading only the minipages that contribute to the final result. Consider a scan query that specifies a set of projected attributes \( P \) (in the `select` clause in an SQL query) as well as a set of attributes \( S \) that participate in selection conditions (in the `where` clause). Assume for simplicity that \( S \) and \( P \) are disjoint, hence only the attributes in \( P \) are in the final result. FlashScan reads from each page only the minipages that correspond to the attributes in \( S \) and evaluates the selection conditions. If, for a given page, there is at least one tuple that passes the selection conditions, then FlashScan reads its minipages for the attributes in \( P \). If none of the tuples of that page satisfy the selection conditions, FlashScan skips to the next page.

As we demonstrate next, this technique is beneficial for highly selective conditions (with a small number of tuples in the result) and for selection conditions that are applied to sorted or partially sorted attributes. In general, the more clustered the attribute values that satisfy the conditions, the bigger the performance improvement.
6.3.2 Scan Experiments

Figure 6.2: FlashScan avoids reading attributes that are not projected; FlashScanOpt additionally avoids reads of minipages without selected tuples. When run over sorted predicate attributes, FlashScanOpt(S) is able to skip many minipages of projected attributes.

For the experiments in this section we generated a relation with 70 million tuples occupying about 10GB. The relation consists of 11 columns (eight 4-byte and three 32-byte attributes) for a tuple length of 128 bytes. NSM layout in Postgres includes a 23-byte header for every tuple. Hence, in order to ensure that relations stored in NSM and PAX layouts have the same size, we allocated an extra minipage to each PAX page for the tuple headers. The page size for both NSM and PAX was set to 64KB. PAX pages contained 12 minipages and the minimum transfer unit from the SSD was 4KB. All experiments were performed on a system with an Intel Core 2 Duo CPU at 2.33GHz and 4GB of RAM, running Ubuntu 8.04 Linux with kernel 2.6.24-21. We used an MTron 32GB SSD, the performance characteristics of which are presented in Table 6.1, formatted with the Linux ext2 file-system.
Varying the Number of Projected Attributes

Figure 6.3: Scans on SSDs: FlashScan is much faster than a traditional scan when either few attributes (small projectivity) or few rows (small selectivity – SEL) are selected. Data analysis queries contain many scans on large relations that project few attributes and select few rows.

In the first experiment, we compare the performance of FlashScan to Postgres’ original scan operator (NSMScan) as we vary the percentage of the tuple projected (its projectivity) from 4% to 100%. The results are in Figure 6.3. NSMScan always reads the whole relation, regardless of projectivity, and exhibits constant performance in scan queries. FlashScan is up to 3X faster than NSMScan for low projectivity: it exploits fast random accesses to “seek” efficiently to the required minipages. As projectivity increases, FlashScan reads more data from every page. At the point where it reads the entire tuple (100% projectivity), FlashScan performs the same sequential read of the relation as NSMScan.

For low percentages of selectivity (0.01%, bottom line), FlashScan consistently outperforms the traditional scan by 3-4x, as it avoids reading projected attributes that belong to non-qualifying tuples. The original scan reads all pages regardless of selectivity. As we discuss in Section 6.3.3, column-store systems already enjoy similar performance benefits when scanning large relations on HDDs. Our techniques, however, are easier to integrate in existing row-based database systems than building a column-store from scratch.
Figure 6.4: When more attributes are projected, FlashScan must read nearly as much data as NSMScan. The contrast between FlashScan and FlashScanOpt is therefore much greater (see Figure 6.2) when not all pages contain selected tuples.

### Varying Selectivity

In this experiment, we consider a scan query with a single equality predicate and vary its selectivity from 0.01% to 100%. We consider two versions of FlashScan. Plain FlashScan first reads all minipages of the projected attributes, discarding non-qualifying tuples after evaluating the predicate. \textit{FlashScanOpt} implements the optimization described in Section 6.3.1: minipages of projected attributes are read only if there is at least one tuple in the page that satisfies the predicate. We experiment with an equality predicate on both sorted and unsorted attributes. When the attribute is sorted, then all matching tuples are stored contiguously. The results for 25% and 75% projectivities are in Figures 6.2 and 6.4 respectively (we use the letters U and S to distinguish between the runs of FlashScanOpt on unsorted and sorted attributes).
The execution time for NSMScan and plain FlashScan remains flat across different selectivities (the small increase for FlashScan for large percentages of selectivity is due to the higher tuple reconstruction cost of PAX). The optimized version of FlashScan, however, performs significantly better with lower percentages of selectivity. For predicates on unsorted attributes and selectivity below 1%, FlashScanOpt skips entire minipages that do not contain any qualifying tuples, thus outperforming plain FlashScan by up to 3x (for 0.01% selectivity and 75% projectivity). For more than 1% selectivity there is at least one tuple that satisfies the predicate in every page (in our relation there were 400 tuples per page). When applying the predicate on a sorted attribute, however, FlashScanOpt outperforms plain FlashScan for all selectivities below 100%: only a few pages contain the contiguous matching tuples and all other minipages can be skipped.

### 6.3.3 FlashScan and Column Stores

We discuss next the relation of FlashScan to column oriented storage on both HDDs and SSDs. For scans with many qualifying tuples, FlashScan’s behavior is in many ways similar to a column-store scan on a traditional hard drive. FlashScan needs to “seek” between minipages, and that seek, although very fast, adds a small overhead (but is preferable over a full sequential scan). Column-stores on HDDs read a large portion of a single column at a time (called *chunk* in MonetDB/X100), to amortize the real disk head seek between different columns. For SSDs and HDDs with comparable bandwidths, these two scans would perform similarly. For highly selective scans, however, FlashScan has an advantage: it can skip minipages that do not contain qualifying tuples. A column-store on HDDs can only skip entire chunks which are *two* orders of magnitude larger than flash SSD transfer units. This means that column-store scans on HDDs need to be 100 times more selective than FlashScan to witness similar benefits.

If we were to run a column-store on SSDs, however, we would find similar behavior to FlashScan regardless of selectivity. In fact, by getting rid of the notion of a chunk, and instead rely on the transfer unit of SSDs, both FlashScan and column-stores would exhibit the exact same SSD reading times (since “seeks” on SSDs are constant in time). While this assertion needs experimental validation, if true, it would be a step towards converging row-store and column-store functionality (which, for current HDD systems, has been a subject of much debate [1, 52]). Several other aspects of row- and column-
stores would need to be reexamined, however, this is a subject of future (and much promising) research. Compression, for example, might be implemented differently on a PAX-based row-store than in a pure column-store. One important aspect of our work is to demonstrate that the proposed techniques can be relatively easily integrated inside a full-blown relational DBMS, retaining much of the existing, rich functionality that otherwise would have to be re-implemented, if one was to build a column-store from scratch.

Since in the next section we describe a general join method which is based on FlashScan for reading data off disk, existing column-stores on SSDs could potentially adopt the same method, by plugging a column-scanner in the place of PAX-based FlashScan. Since our scope is limited to traditional, row-based DBMS, we do not revisit column-store applicability.

6.4 FlashJoin

In this section, we present FlashJoin, a multi-way join algorithm tailored for solid state drives. We first give an overview of our algorithm in Section 6.4.1 and then describe its main components in the following sections.

6.4.1 FlashJoin Overview

To take advantage of the fast random reads of SSDs, FlashJoin uses the same principal techniques as FlashScan. It avoids reading unneeded attributes and postpones retrieving attributes in the result until absolutely necessary.

FlashJoin is a multi-way equi-join algorithm, implemented as a pipeline of stylized binary joins. Each binary join in the pipeline is broken into two separate pieces, a join kernel and a fetch kernel, each of which is implemented as a separate operator, as shown in Figure 6.5. The join kernel computes the join and outputs a join index. Each join index tuple consists of the join attributes as well as the row-ids (RIDs) of the participating rows from base relations. The fetch kernel retrieves the needed attributes using the RIDs specified in the join index. FlashJoin uses a late materialization strategy, in which intermediate fetch kernels only retrieve attributes needed to compute the next join and the final fetch kernel retrieves the remaining attributes for the result. This approach offers some important benefits
over traditional joins, which use an *early materialization* strategy.

First, FlashJoin reduces the amount of data retrieved from the input relations to compute the join result. It accesses the join attributes and accesses other projected attributes only from rows that participate in the result. Second, since join kernels process join indices instead of all projected attributes from the input relations, the indices on the build input are smaller. Thus, the join kernel is more memory-efficient. Moreover, when multiple passes are needed, the join kernel incurs lower partitioning costs than traditional joins.

These benefits come at the cost of additional random reads for retrieving the join attributes and other projected attributes separately. Moreover, they come at the cost of additional passes over data in the fetch kernel. In the experimental section, we show that this tradeoff is worthwhile when using SSDs.
6.4.2 Join Kernel

The join kernel leverages FlashScan to fetch only the join attributes needed from base relations. The join kernel is implemented as an operator in the iterator model. In general, it can implement any existing join algorithm: block nested loops, index nested loops, sort-merge, hybrid hash, etc.

We explore the characteristics of a join kernel that uses the hybrid-hash algorithm [30]. This kernel builds an index (hash table) on the join attribute and RID from the inner relation and probes the index in the order of the outer. Depending on the available memory and the size of the input, hybrid-hash may make multiple passes over the input. Compared to a traditional hash join, this join kernel is more efficient in two important ways because it does not need to manage all of the projected attributes from the input. First, the join kernel needs less memory, thus many practical joins that would otherwise need two passes can complete in one pass. Second, when the join kernel spills to disk, materializing the runs is cheaper. Although we only explore hybrid-hash, we believe a sort-merge kernel would offer similar results because of the duality of hash and sort [39].

6.4.3 Materialization Strategy

To implement different materialization strategies, the query plan must also change at a logical level internal to FlashJoin. Typically in a query plan, associated with each node is a description of the tuple the node produces. For example, in Figure 6.5, a traditional plan would indicate that the first join (Join 1) produces tuples with attributes \(B, C, E, G, H\). By adjusting these descriptions, which indicate to the fetch kernel the attributes to retrieve, we can specify any strategy we like for retrieving projected attributes, as long as it is consistent with the plan.

For any individual binary join, there is a tension between the cost of retrieving projected attributes from base relations immediately after the join and the cost of retrieving the attributes further downstream. The former increases the cost for partitioning in subsequent join kernels, if those joins cannot complete in one pass. The latter forces the final fetch kernel to make additional read and write passes over the output, which can be expensive if the output cardinality is large. Choosing the optimal strategy requires estimating the cost of these options and minimizing the total cost across the whole multi-way join. The number of possibilities is large since at every node in the plan, we can choose to material-
ize any subset of the remaining needed attributes. To make matters more complicated, the cost for a particular strategy may be significant enough to affect the total cost and thereby affect the optimal join order.

Instead of exploring this optimization space, in this work, we use a simple heuristic: late materialization. This heuristic postpones retrieving projected attributes as far downstream as possible. Every join produces the minimum set of attributes needed by the next operator and the final join produces the output needed by the remaining plan operators. This heuristic works well as we show in Section 6.5 because typical query plans reduce the output cardinality at each level of the plan. In that case, an extra pass over the output is usually cheaper than paying the additional materialization cost through multiple levels of the plan. In order to perform the late materialization, the join kernel output carry forward the RIDs for each of the base relations needed downstream.

Figure 6.5 shows an example of this strategy. Each tuple produced by FlashScan 1 contains only attribute $A$, which is one of the join attributes of Join 1, and each tuple produced by FlashScan 2 contains only attribute $D$, the second join attribute of Join 1. Similarly, the tuples produced by Join 1 contain only attribute $G$, which is one of the join attributes of Join 2. Finally, Join 2, the last join node, produces tuples with all of the attributes needed in the result: $B, C, E, H, F$. Moreover, if there was a sort operator after Join 2 that sorted the results based on attribute $L (R_3)$, then each tuple produced by Join 2 would also contain $L$.

### 6.4.4 Fetch Kernel

The fetch kernel uses the join index, produced by the join kernel, to materialize the attributes of the join result from their base relations. For example, in Figure 6.5, the join kernel of Join 1 outputs pairs of RIDs $(id_1, id_2)$ from relations $R_1$ and $R_2$, respectively. A RID specifies the page and offset within the page for that row. The fetch kernel uses $id_2$ to locate and retrieve attribute $G$ from relation $R_2$. Similarly, Join 2 produces a join index containing $(id_1, id_2, id_3)$ pointing to rows of $R_1$, $R_2$, and $R_3$. The corresponding fetch kernel, uses $id_1$ to retrieve attributes $B, C$, $id_2$ to retrieve attributes $E, H$ and $id_3$ to retrieve attribute $F$.

A straightforward strategy for the fetch kernel is to retrieve projected attributes in a tuple-at-a-time, non-blocking fashion. For each join index tuple, the kernel locates the needed mini-pages in the buffer
pool or retrieves them from the underlying relation, and composes the result tuple. This approach is reasonable when all of the pages needed to generate the result can fit in memory because random reads are cheap. When available memory is insufficient, however, this approach may result in reading some pages multiple times because the RIDs in the join index are usually unordered. The larger the result, the worse is the overhead for re-reading pages. TID hash joins implemented this approach, and this overhead was their biggest weakness [74].

Algorithm 10 Fetch Kernel to produce the join result with multiple passes.

**Input:** \( R_1, \ldots, R_k \): base relations, \( id_1, \ldots, id_k \): corresponding RID attributes, \( A_1, \ldots, A_k \): \( A_i \) is the set of attributes to fetch from relation \( R_i \), \( |A_1| < \ldots < |A_k| \), \( JI \): join index

**Output:** \( JR \): join result

1: \( T = \text{SortedStream}(JI, id_1) \)
2: for \( i = 1 \) to \( k \) do
3: \( Z = \{\} \)
4: while \( T \) is exhausted do
5: for all memory resident tuples \( t \) of \( T \) do
6: \( \text{Add GeneratePartialResultTuple}(t, id_i, R_i, A_i) \) to \( Z \)
7: end for
8: Produce sorted run of \( Z \) on \( id_{i+1} \)
9: end while
10: \( T = \text{SortedStream}(Z, id_{i+1}) \)
11: end for
12: \( JR = T \)

To mitigate this overhead, we present a fetching strategy, inspired by Jive-Join [70], that reads each containing page from each base relation only once at the cost of additional passes over the join index. Algorithm 10 makes multiple passes over the join index fetching attributes in row order from one relation at a time. Roughly speaking, in each pass, it sorts the join index based on the RIDs of the current relation to be scanned. Then, it retrieves the needed attributes from that relation for each tuple and augments the join index with those attributes. If the join index does not fit in memory, it uses an external merge sort. Sorting ensures that once a mini-page from a relation has been accessed, it will not
need to be accessed again, thus placing minimal demands on the buffer pool. Sorting, however, does not ensure sequential access to the underlying relation because the needed pages can be far apart. Hence, this strategy of decoupling and postponing materialization is better suited to SSDs than HDDs.

We explain some important optimizations in Algorithm 10 above. First, the final merge of the sort and attribute retrieval are pipelined to avoid the unnecessary final write. SortedStream sorts the join index but leaves out the last merge step that creates a final sorted run (line 1). As tuples are fetched from $T$ (a sorted stream), it merges the underlying runs on demand. For each tuple, GeneratePartialResultTuple retrieves the needed attributes and augments the join index (line 6). We ensure that only enough tuples are read so that the result $Z$, the new join index with the attributes $A_i$ projected, can fit in memory (line 5). Finally, we sort $Z$ on the RID for the next relation (line 8). The sorted $Z$ runs are then merged into a single sorted stream $T$ (line 10) which is then pipelined with attribute retrieval from the next relation.

A second optimization is that our fetch kernel processes relations in order of the width of the attributes retrieved, from smallest to largest. This order reduces the data spilled to SSD when producing the intermediate runs. Third, if $Z$ and $JI$ fit in memory, we avoid spilling them to disk. Fourth, if a fetch kernel (esp. intermediate ones in the plan) already has its input sorted on the RIDS for one of the relations, it processes that relation first to avoid the sort. The cost for this fetch strategy depends on the cardinality of the result, the width of the attributes, and the number of relations that need to be accessed.

Although we did not implement it, an analogous fetching strategy would be to use hashing instead of sorting, as done in RARE-join [93]. In each pass, we could hash join index tuples into buckets based on the page id in the RID. This would ensure that all tuples that need the same page fall into the same bucket. The buckets could be sized such the number of distinct pages referenced fit in the available memory. We could then process each bucket fetching the the appropriate pages from base relations. This strategy requires less CPU overhead than the sort-based one, but would need more memory than the sort-based one for the same number of partitions.

In summary, if the optimizer estimates that all the pages needed to retrieve projected attributes can fit in memory, we opt for the naive, tuple-at-a-time strategy. Otherwise, we resort to our sort-based relation-at-a-time fetching strategy.
6.5 Experimental Evaluation

This section presents our experiments with FlashJoin. Our objective is to demonstrate the effectiveness and efficiency of FlashJoin during the execution of multi-way joins and complex BI queries. In Section 6.5.1, we describe the algorithms evaluated, the datasets and queries used, and implementation details. The results are presented in the remainder of the section.

6.5.1 Experimental Setting

FlashJoin was implemented inside PostgreSQL as a new join operator. Significant changes had to be performed in PostgreSQL’s planner component to support the late materialization strategy. The planner employs a cost-based optimizer to determine the most efficient way to execute a query. The output of the planner is a query execution plan that details which access methods to use and which attributes to access from each base relation. Additionally, it determines for each operator in the query plan the schema (attributes) of output tuples. We altered the planner in order to generate query execution plans which comply to the late materialization strategy. Specifically, we added a recursive function that takes as input the query execution plan and process it in a top-down fashion (recall that a query execution plan is a tree of operators). For every operator (node in the query execution plan) our function alters the schema of output tuples by removing unnecessary attributes. Overall, we added approximately 7K new lines of code in PostgreSQL, half of which was the implementation of FlashJoin and planner component and the rest divided between FlashScan, bulk-loading utilities, the buffer manager and the storage layout.

There are several limitations with respect to our implementation of FlashJoin and the execution of complex queries. Currently, we support query execution plans that contain a pipeline of hash joins followed by other operators, such as sort and aggregation. Furthermore, the optimizer is not aware of the late materialization strategy and the cost of FlashJoin and so it produces the same plan that it would using the hybrid hash operator. In the future, we plan to integrate the cost of FlashJoin into the optimizer.

In all of our experiments, we compare the performance of FlashJoin with the hybrid-hash join algorithm implemented in PostgreSQL. We consider two variations of the hybrid-hash join algorithm: one for each storage layout. We refer to the default implementation over the NSM layout as HNSM. We call our version of hybrid-hash over the PAX layout HPAX. HPAX and HNSM use the same materialization
strategy, but when HPAX is used, the scan nodes use the FlashScan operator to read only the projected attributes of each relation.

We conducted a series of experiments using synthetically generated datasets and queries. We also evaluated the performance of FlashJoin using full TPC-H queries. All of the experiments were conducted with the same page sizes and relations and on the same system that we described in Section 6.3.2. The sizes of the relations and the amount of memory allocated are described separately for each experiment. After the execution of each query, PostgreSQL was restarted and all cached pages were flushed from memory using the `drop_caches` utility of the Linux kernel. In all the experiments, the performance metric measured is wall-clock time to produce the full join result.

### 6.5.2 Two-way Join Results

In this section, we assess FlashJoin during the execution of two-way joins. We consider the equijoin of two relations $R, S$ on a single join attribute. $R$ contains 70 million tuples with a total size of approximately 10GB and $S$ contains 7 million tuples with a total size of 1GB. We consider queries of the following form: 

```
SELECT R.a1, ..., R.am, S.a1, ..., S.am FROM R, S WHERE R.ai = S.aj
```

We use `projectivity` to refer to the percentage of the tuple length projected from each join relation. When projectivity is 100%, a join result tuple contains all of the attributes of $R$ and $S$ and has a total tuple length of 256 bytes.

#### One-pass Joins

In the first experiment, we vary projectivity from 4% to 100%. The amount of memory allocated to the join is 1GB to ensure that all algorithms compute the join in one pass. Figure 6.6 shows the runtimes of HNSM, HPAX, and FlashJoin with three different result cardinalities. The percentage next to each algorithm’s label indicates the cardinality of the join result, expressed as a percentage of the cardinality of the larger join relation. In this experiment, join result cardinality had a negligible effect on the performance of HPAX and HNSM, so we only present one set of execution times for them.

The performance of HNSM is independent of projectivity. The input scans of HNSM read relations $R$ and $S$ entirely, regardless of how many attributes participate in the join result. HPAX, however, uses the FlashScan operator to read from flash disk only the attributes that are actually needed in a query.
Figure 6.6: Performance comparison of join algorithms for two-way joins when the join is computed in one pass.

For low projectivities (4% in our experiment), HPAX is 3X faster than HNSM. However, as projectivity increases, more attributes are read and the performance difference between HPAX and HNSM diminishes.

The performance of FlashJoin depends on both the projectivity and the join result cardinality. When the cardinality is 1% or greater, FlashJoin and HPAX perform the same since every disk page of $R$ and $S$ contains at least one tuple that belongs to the join result and must be read. For cardinalities less than 1%, FlashJoin reads all of the minipages of the join attributes but only a fraction of the minipages containing projected attributes. Consequently, it reads less data than HPAX. As projectivity increases, the number of minipages that FlashJoin does not read increases accordingly and causes a more pronounced performance difference of up to a factor of 3x.
Two-pass joins

Figure 6.7: Performance comparison of join algorithms for two-pass two-way joins.

In the second experiment, we compare the performance of the join algorithms when they require two passes to compute the join result. The amount of memory allocated to the join is 100MB and the same query was used as above. Figure 6.7 shows results for HNSM and HPAX with a join result cardinality of 1% and for FlashJoin with cardinalities, varying from 0.1% to 100%.

The execution times of HNSM and HPAX increase linearly with projectivity: since projections are performed early in the query execution plan, more data participate in the partitioning phase as projectivity increases. Consequently, the partitioning cost increases. HPAX is faster than HNSM for lower projectivities because it reads only the attributes needed in the query, until at 100% projectivity, both HPAX and HNSM read all attributes and perform the same.

FlashJoin, however, is much faster than HNSM and HPAX, and increasingly so as projectivity increases. FlashJoin accesses only the join attributes during the expensive partitioning phase, when two
passes are required for the computation of the join. Consequently, the partitioning cost of FlashJoin does not depend on projectivity.

The performance of FlashJoin does depend on the join result cardinality, however. When the cardinality is low, FlashJoin reads only a few minipages of projected attributes in order to construct the join result tuples and performs up to 7X faster than HPAX and HNSM. As cardinality increases, FlashJoin reads a larger fraction of minipages, thus increasing the join result construction cost. Furthermore, when the join index does not fit in memory, FlashJoin pays the cost of materializing the join index, whose size depends on the result cardinality. When the result cardinality is 100%, FlashJoin must read as many minipages as HPAX and is only marginally faster than HPAX.

**Memory impact on joins**

![Two-way join](image)

Figure 6.8: Performance comparison of two-way join algorithms as a function of memory size.

In the third experiment, we compare the performance of the join algorithms as we vary the amount of memory allocated. We set the join result cardinality at 1% of the cardinality of \( R \) (larger relation)
and the projectivity at 25%. We vary the amount of memory from 100MB to 600MB.

Figure 6.8 shows that FlashJoin is faster than both HPAX and HNSM for all memory sizes examined. HPAX and HNSM require at least 500MB to compute the join in one pass. The hash table on the join attribute of the build relation requires 270MB and the projected attributes of the build relation require 230MB. When memory is less than 500MB, HPAX and HNSM compute the join in two passes. Since they both use the hybrid hash join algorithm, they exploit any memory available to avoid writing all of the partitions. When the allocated memory is between 250MB and 500MB, one partition fits in memory and only the second partition gets written.

By reading only the join attributes, FlashJoin has a smaller memory footprint and increases the range of memory sizes at which a two-way join is executed in one pass. FlashJoin requires only 270MB of memory (for the hash table) to compute the join in one pass. Consequently, FlashJoin is 2X faster than the other algorithms when memory size is between 270MB and 500MB. When memory is larger than 500MB, FlashJoin is 1.2X faster. When memory is less than 270MB, FlashJoin computes the join in two passes; it is 3X faster than HPAX and HNSM due to the reduced partitioning cost.

6.5.3 Multi-way Joins

Next, we assess FlashJoin during the execution of multi-way joins. We consider three-way joins first, then look at star joins of up to six relations. Finally, we examine the impact of changing the join algorithm on more complex queries such as those in TPC-H.

Three-way Joins

For this experiment, we executed queries of the form: “SELECT $R.a_n, \ldots, R.a_m, S.a_n, \ldots, S.a_m, T.a_n, \ldots, T.a_m$ FROM $R, S, T$ WHERE $R.a_i = S.a_j \text{ AND } R.a_k = T.a_l$”. $R$ contains 50 million tuples, $S$ contains 20 million tuples and $T$ contains 7 million tuples; their sizes are 7.5GB, 3GB and 1GB, respectively. The join between $R$ and $S$ is a primary-foreign key join returning 50 million tuples ($R$ contains the foreign-key). We control the cardinality of the join result by varying the selectivity of the join between $R$ and $T$. The join order is determined by the optimizer and is the same for all queries executed. The amount of memory allocated to each join is 100MB. All of the joins are executed in two passes.
Figures 6.9 and 6.10 show the results for two different projectivities: 25% and 75%, respectively. For each projectivity, we vary the cardinality of the join result from 0.01% (5000 rows) to 100% (50 million rows) of the larger relation’s cardinality.

FlashJoin is up to 3X faster than HPAX when projectivity is 25% and up to 5X faster than HPAX when projectivity is 75%. For a given projectivity, increasing the cardinality of the join result increases the execution time of all algorithms: higher cardinalities causes more rows in the intermediate results and hence higher partitioning costs. However, FlashJoin suffers the least since it only partitions the join attributes. It benefits more at higher projectivity since there are more attributes that it avoids partitioning.
Figure 6.10: Performance comparison of join algorithms during three-way joins as a function of join result cardinality, when projectivity is 75%.

Memory impact on joins

In the next experiment, we compare the algorithms’ performance for three-way joins as a function of the amount of memory allocated per join. We consider the same three-way join queries as in the previous experiment. We set the join result cardinality at 1% of the cardinality of relation \( R \) and the projectivity of each relation at 25%. We vary the amount of memory allocated per join from 100MB to 1GB. Figure 6.11 presents our results.

Both HPAX and HNSM require 500MB to execute the join between \( R \) and \( T \) in one pass and 1600MB to execute the join between \( R \) and \( S \) in one pass. By accessing only the join attributes, FlashJoin only requires 270MB to compute the join between \( R \) and \( T \) in one pass and 900MB to compute the join between \( R \) and \( S \) in one pass. FlashJoin is therefore 1.3X to 2.5X faster than HNSM and 1.1X to 2.3X faster than HPAX. We also conducted experiments with larger join result cardinalities and
Figure 6.11: Performance comparison of join algorithms during three-way joins, as a function of memory size allocated per join.

got consistent results.

Star joins

Next, we assess FlashJoin during the execution of $N$-way STAR joins for different values of $N$, ranging from 3 to 6. The relations and their sizes are presented in Table 6.2. $R_0$ is the fact table and $R_1, \ldots, R_5$ are dimension tables. In this experiment, an $N$-way join involves the first $N$ relations ($R_0, \ldots, R_{N-1}$) of Table 6.2. The join between $R_0$ and $R_{N-1}$ returns 5 million tuples (10% of $R_0$’s cardinality). Every other join between $R_0$ and $R_i$, where $i = 1, \ldots, N - 2$, is a primary-foreign key join. We set the projectivity of each relation at 25%. Hence, for the 6-way join, the size of each result tuple produced is 192 bytes.

Figure 6.12 in Section 6.1 presents our results. FlashJoin outperforms both HPAX and HNSM by at least a factor of 2X for all of the values of $N$ examined. FlashJoin uses the drive more efficiently than
hybrid-hash join over both traditional row-based (NSM) and column-based (PAX) layouts, by reading the minimum set of attributes needed to compute the join, and then fetching only those attributes that participate in the join result. By accessing only the join attributes needed by each join, FlashJoin also reduces memory requirements, which is beneficial in two ways: it decreases the number of passes needed in multi-pass joins (hence speeding up the join computation), and it frees up memory space to be used by other operators (hence leading in improved overall performance and stability in the system).

Figure 6.12: Multi-way joins on SSDs: FlashJoin is at least 2x faster than hybrid hash join over either traditional row-based (HNSM) or PAX layouts.

**TPC-H Queries**

Finally, we evaluate FlashJoin in the context of more complex queries. We chose queries Q3 and Q10 from the TPC-H benchmark. Q3 retrieves unshipped orders. It involves a three-way join among tables `CUSTOMER`, `ORDERS` and `LINEITEM`. There is a `GROUP BY` clause, applied on two attributes of
ORDERS and one attribute of LINEITEM. Q3 also sorts orders by decreasing revenue. Q10 identifies customers having problems with shipped parts. In involves a four-way join among tables CUSTOMER, ORDERS, LINEITEM and NATION. A GROUP BY clause is applied on six attributes of CUSTOMER and one attribute of NATION. Customers are sorted in decreasing order of lost revenue. We generated data using a scale factor (SF) of 10. In all tables, we replaced variable length attributes with fixed length attributes. In both queries, we set the memory allocated per operator at 100MB.

Figure 6.13: Performance comparison of join algorithms during the execution of TPC-H queries.

Figure 6.13 presents results for both queries. For Q3, FlashJoin is 1.6X faster than HNSM and 1.4X faster than HPAX. For the amount of memory allocated, all join algorithms require two passes to execute one of the joins in Q3. Hence, FlashJoin performs better than HPAX and HNSM due to its reduced partitioning cost.

For Q10, FlashJoin is 1.8X faster than HNSM and 1.5X faster than HPAX. As in Q3, there is one join in Q10 that is executed in two passes by all join algorithms. However, Q10 exhibits higher projectivity than Q3, producing larger tuples. Consequently, the partitioning cost of HNSM and HPAX is even
higher for Q10 and their performance difference with FlashJoin increases.

6.6 Conclusions

SSDs constitute a significant shift in hardware characteristics, comparable to large CPU caches and many-core processors. In database systems, SSDs change not only power efficiency but also query execution efficiency. We demonstrate that SSDs can improve database performance for business intelligence and other data analysis queries. We first show that a column-based page data layout is a natural choice for speeding up selections and projections on SSDs. By combining a column-based storage layout with temporary join indexes and late materialization, we produce a new pipelined join algorithm that is much more efficient on SSDs than previous algorithms such as hybrid-hash join.

Our results show performance improvements of up to a factor of six for scans, multiway joins, and complex queries from the TPC-H benchmark. These improvements stem from three factors: First, FlashJoin only reads attributes as they are required by each operator, which reduces the amount of data read as compared to hybrid-hash join. Second, FlashJoin has a reduced memory footprint and therefore increases the range of memory sizes for which each join can be computed in one pass; one-pass joins require much less I/O than two-pass joins. Third, it employs a late materialization strategy to access the minimum set of attributes needed at any point in the query execution plan, thus reducing the amount of data accessed from base relations and the partitioning cost when the joins are executed in two passes.
Chapter 7

Analyzing the Energy Efficiency of a Database Server

7.1 Introduction

In the same way performance (e.g. measuring completed tasks per unit of time) has been central to systems evaluation, energy-efficiency (e.g. completed tasks per unit of energy) is quickly growing in importance for minimizing IT cost. Most recent work in improving energy-efficiency is either hardware-platform oriented or workload management oriented. However, the role of data management software in affecting and, ultimately, improving energy efficiency has not been explored.

Our goal is to understand the energy characteristics of database systems on modern hardware. We first characterize the power-use profiles of database operators under different configuration parameters. We find that common database operations can exercise the full dynamic power range of a server, and that the CPU power consumption of different operators, for the same CPU utilization, can differ by as much as 60%. We also find that for these operations CPU power does not vary linearly with CPU utilization.

We then experiment with several classes of database systems and storage managers, varying parameters that span from different query plans to compression algorithms and from physical layout to CPU frequency and operating system scheduling. Contrary to what recent work has suggested, we find that within a single node intended for use in scale-out (shared-nothing) architectures, the most energy-efficient configuration is typically the highest performing one. We explain under which circumstances
this is not the case, and argue that these circumstances do not warrant a re-targetting of database system optimization goals. Further, our results reveal opportunities for cross-node energy optimizations and point out directions for new scale-out architectures.

### 7.1.1 Where does power go?

To assess the opportunities for optimization, we start by measuring the power of system components from idle to fully utilized. Figure 7.1 shows the power break down (measurement details are in Sections 2 and 3) of one configuration of our 8-core (dual-CPU) test machine. An important factor for any study in energy efficiency is the hardware setup used; Section 7.2 discusses hardware trends and argues why this particular setup’s power profile is close to what we expect to be a typical node in a cluster.

The right half of the pie-chart (grayed out) is the idle power consumption and accounts for about half of the peak power. As the breakdown shows, the three main components contributing to the waste are the idle power of the two CPUs, the fixed power of RAM and the rest of the system-board components. The idle power of the HDDs is relatively small, and the consumption of the SSDs is close to zero. The left half of the pie-chart shows the additional power consumed when all CPUs and disks are fully utilized. The SSDs and HDDs draw similar power, however, the dominant components are the two CPUs (+112W). Next, we examine in more detail the components that exhibit measurable dynamic power range, i.e., the difference between idle and peak power.

![Figure 7.1: Power breakdown of test machine.](image-url)
7.1.2 Power use and resource utilization

The largest power consumer in our system are the CPUs. These are two quad-core Intel Xeon E5430 CPUs (Harpertown) which use aggressive power management techniques. To our knowledge, there are no studies that show the CPU power use of database-like operators in modern processors as the CPU utilization varies. The literature provides several power models, however, as a recent study shows [89], these models begin to break down in CPUs with shared resources and aggressive power management techniques. Rather than attempting to derive yet another power model, we focus on computations, data access patterns, core and cache sharing patterns, and data sizes that are directly relevant to database operators. To factor out overheads that a complete DBMS may carry, we implement high-performance, micro-benchmark kernels for cache-conscious hash joins and parallel sorting. For scan operators we use an open-source minimum overhead row/column storage manager.

Figure 7.2 shows the total CPU power consumption for several operators, as we vary the number of cores used, from 1 to 8 (each core is 100% utilized, point 0 is the idle power). When we refer to CPU utilization, we mean the fraction of total cores used since each of our micro-benchmark processes will fully utilize a core from the OS perspective. Figure 7.2(a) corresponds to a performance-oriented process scheduling policy whereas Figure 7.2(b) uses an energy-conserving policy (details are in Section 7.3).

Surprisingly, different operators can vary by more than 60% in power consumption (e.g., sort and row-scan using two cores). Section 7.3 explains this result and presents further details and performance results with these micro-benchmarks. We also find that in SSDs, power-use is nearly proportional to utilization (Section 7.3), something we were not able to find in the literature. These results reveal opportunities for cross-node energy optimizations (e.g., the last 30% of a node’s CPU computation capacity comes essentially for free) and inefficiencies in server CPUs that new/alternative hardware should address (e.g., need for smaller CPU caches). Section 7.5 discusses these and other implications in detail.

7.1.3 Affecting energy efficiency: what “knobs”?

We define energy efficiency as the ratio of useful work done to the energy used, which is the same as the ratio of performance to power:
As database software is rich in tunable parameters, from system-level constants to query planning and execution, and our experiments so far point to several options that can affect power-use, these parameters can potentially affect energy efficiency. The most promising knobs are the ones that can directly trade CPU cycles for disk access time since these are two resources with significantly different power-use profiles. Such tradeoffs exist in access methods (sequential scans vs. clustered and non-clustered index scans), column-oriented vs. row-oriented record access, compression techniques (lightweight vs. heavyweight), and join algorithms (hash-join or sort-merge vs. nested loop join). Recent work has speculated that some of these knobs may be promising [41, 51], and others have shown specific cases where energy efficiency is improved [76, 65, 104] at the expense of performance.

With all these available knobs at hand, we set to find regions where performance may be sacrificed for energy efficiency. Such tradeoffs would make the case for considering energy apart from pure performance inside a modern, scale-out server node.

We focus on operators and queries that commonly appear in data analysis tasks and data warehousing workloads. In online transaction processing (OLTP) workloads, most of the software knobs we consider

\[
EE = \frac{\text{Work done}}{\text{Power}} = \frac{\text{Work done}}{\text{Time}} = \frac{\text{Perf}}{\text{Power}} \quad (7.1)
\]
are not usually applicable (e.g., long-running scans, joins, different query plans, columnar storage, compression). Although our results do not directly apply, recent research shows [48] that modern OLTP applications can significantly improve transaction throughput, and thus energy efficiency, by redesigning the entire OLTP engine, rather than just tuning it.

7.1.4 Energy efficiency vs. performance

To increase the generality of our results and ensure that we do not miss out on opportunities for improving energy efficiency due to system-specific limitations, we experimented with a total of five systems/database kernels. We implemented two minimum overhead kernels, one for performing parallel, cache-conscious hash joins, and one for sorting, using Alphasort and a custom implementation of parallel merging. We then chose a high-performance, open-source database storage engine that supports both column-oriented and row-oriented database scans along with light-weight compression [50]. Lastly, for full-fledged DBMSs, we chose a popular open-source DBMS (PostgreSQL) and a popular commercial DBMS (System-X).

Figure 7.3: Energy efficiency vs. performance in System-X.

In the overwhelming majority of our experiments (we collected over 1000 data points) we observe that, for any given database task (from simple scans to multi-user query workloads), the most energy-
efficient configuration is the highest performing one. We found isolated cases where small gains in energy efficiency can be achieved by choosing a lower performing operating point. Even if we were to assume a zero-power system board (to derive an upper limit) these gains extend to only 10-12%.

Figure 7.3 shows the energy efficiencies of TPC-H query 5 in a System-X database for 128 different configurations. Note, this graph is a parametric graph in which both axes – energy efficiency (y-axis) and performance (x-axis) — are dependent variables; they both depend on the configuration. We see that energy efficiency goes hand-in-hand with performance. As Section 7.4 discusses, there is almost always a configuration that can further improve energy efficiency by simply improving performance. Even for the points near peak performance, their energy efficiency varies less than 10%.

Figure 7.4 shows microbenchmark results that are representative of the picture we see repeatedly in our experiments. The left graph shows the energy efficiency for a set of SSD disk-resident column-store scans. Again, each point corresponds to a different configuration (we varied number of CPUs, CPU frequency, and compression algorithms). As before, the highest performing configuration is the most energy efficient. The right graph isolates three points and shows how the performance and energy efficiency change when employing two different compression methods (“light” stands for FOR and “heavy” for FOR-delta [50]). This is a case where energy efficiency and performance improve at different rates. As a result, if all components other than CPU and disks consumed zero power (depicted as “energy efficiency for CPU only”–we discuss this in Sections 4 and 5), we would have seen a small drop in energy efficiency when switching from a “light” compression method to a faster “heavy” one.

7.1.5 Analysis of results and implications

Our results come in direct contrast to recent work [51, 65, 104] that has suggested that energy efficiency and performance are often two different optimization goals. Taking a closer look at the results in those papers we can see that the differences stem from computing energy efficiency without taking into consideration the power of peripheral components and the idle power of the CPU. This is a result of the fixed up-front power that most system components exhibit at their lowest performance points, which we explain in Section 7.5.

Although our results seem to discourage any further software-level energy reduction methods within a single-node DBMS, we believe that they point to opportunities in (a) cross-node energy efficiency tech-
niques for co-locating computation and optimizing data placement and movement, and (b) alternative single-node hardware architectures that remove inefficiencies in current designs.

![Energy Efficiency vs. Performance](image1)

**Figure 7.4:** Scan results (including compression).

### 7.2 Background and motivation

Energy efficiency of computing equipment in data centers is an important concern for several reasons. First, powering and cooling costs are starting to overtake the cost of hardware [44]. Second, increased energy use has negative implications for density, reliability, and scalability within a data center. Finally, increased data center energy use has prompted environmental concerns leading governments across the world seeking to regulate enterprise IT power. For these reasons, we are starting to see a shift in industry and research towards optimizing for energy efficiency.

This shift is broad, ranging from chips to data-centers. Chip designers have considered a variety of power-saving techniques such as dynamic frequency and voltage scaling (DVFS), clock routing optimization, asymmetric multi-cores, and so on. System architects have suggested strategies for dynamically managing DRAM power states, disk speed control, or spinning down disks. We are seeing new energy-efficient platform redesigns that meet the performance SLAs of a small, but important class of
workloads, e.g., web servers and data analysis [6, 44, 90]. Finally, recent work proposes ensemble-level optimizations, for example, shifting workloads around for meeting power and temperature constraints or consolidation for improved energy use [101]. Going forward, data-center architects are investigating holistic redesigns that treat the data center as a single computer [32, 85].

Towards this end, some are calling for energy proportional hardware [11]. A perfectly proportional component uses no power when not in use and only uses power in constant proportion to its performance. Since energy efficiency is the ratio of performance to power, energy-proportional hardware provides constant energy efficiency at all performance regimes. With such hardware, we need not employ higher-level techniques to adjust for the most efficient point. The hardware designers provide the best efficiency possible, and the software designers continue to worry about performance.

Today, however, components are hardly energy proportional, but their dynamic power range and proportionality are steadily improving. CPU vendors initially introduced low-power versions of their processors for the mobile market. To improve proportionality for server-class CPUs, vendors started to introduce increasingly finer active and sleep power states. Commercial products already include hooks to control power states of DRAM, and new memory controllers are on the way that dynamically adjust these states. There are also new types of non-volatile memory around the corner like PCRAM [79] and memristor [99], vying to replace DRAM altogether. For drives, we see new drives with multiple spin rates as well as lower power drives with more sleep states. Finally, SSDs are on track to replace hard drives in contexts where capacity is not the limiting factor. Our experiments later will show that these drives exhibit near linear proportionality.

<table>
<thead>
<tr>
<th>Component</th>
<th>Min (W)</th>
<th>Max (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2xIntel Xeon E5430 Quad Core 2.66GHz</td>
<td>48W</td>
<td>160W</td>
</tr>
<tr>
<td>4x4GB FB-DIMMS (RAM)</td>
<td>40W</td>
<td>40W</td>
</tr>
<tr>
<td>4x300GB Seagate Savvio 10K.3 2.5”</td>
<td>14W</td>
<td>24W</td>
</tr>
<tr>
<td>4x64GB Intel X-25E 2.5” (SSD)</td>
<td>0.2W</td>
<td>10W</td>
</tr>
<tr>
<td>System board (remaining components)</td>
<td>54W</td>
<td>54W</td>
</tr>
</tbody>
</table>

Using these newer components, we still are left with the question of how to best assemble these
into larger, energy-efficient systems. There are two main approaches for building data-management architectures: scale-up (shared memory and shared disk) and scale-out (shared-nothing). The former has many connected components: servers and disks, all managed as a single whole. For these, the most energy-efficient approach involves choosing the right balance of components for a task and powering down unneeded resources. Today, however, we are limited to high-end, typically more power-hungry components that are used in such architectures.

On the other hand, scale-out approaches involve picking the best single node configuration and then connecting them via a scalable network. This approach seems more amenable to energy efficiency optimizations for data-centric workloads. We can pick from a variety of components for more aggressive energy optimizations on single node, and then horizontally scale the task to achieve needed performance while maintaining the same energy efficiency. Other projects like FAWN [6] and microslice servers [44] have taken this approach but not for database workloads. Thus, in this work, we focus on understanding the power-performance tradeoffs for a single database node in a scale-out setting. Others have started to look at such architectures [65, 104], but those analyses fall short. They neither provide a detailed study of database operators on modern CPUs nor do they account for total system power.

Our current single node setup uses modern (as of 2009) CPUs, DRAM, and SSDs. We chose this setup because it is inline with future trends in components: the CPU has several active power states and the SSDs are state-of-the-art. Moreover, the system is balanced for performance; the disks when fully utilized can keep all the cores busy on simple operations like sorting and hashing.

### 7.3 Where Does Power Go?

In this section, we analyze the power profiles of different hardware components in the context of database operations. The goal is to increase our understanding of how these operations affect power consumption and, ultimately, to reveal the energy saving potential. For that purpose, we designed a set of micro-benchmarks to exercise the hardware components of a database server using typical database-centric operations such as scans, joins, and sorts.

In this study we used an HP xw8600 workstation running a 64-bit Fedora 4 Linux with kernel 2.6.29. The configuration of our server is presented in Table 7.1. The system has two Intel Xeon E5430
2.66GHz quad core processors, for a total of 8 cores. Each processor has an 32K L1 instruction and data cache, and two 6MB L2 caches shared by two cores.

In all of our experiments, we used a Brand Electronics 20-1850 CI to measure the total system power. This power meter has ±1.5% accuracy and collects readings once a second. To isolate the power drawn by the SSDs, HDDs, and the CPUs, we attached a clamp meter to the 5V and 12V lines to the components from the power supply. We used an Extech MA120 200A AC/DC mini clamp-on meter which had a resolution of 100mA and an accuracy of ±2.8%. By multiplying the current measured with the line voltage, we derive our power measurements. To increase the resolution of the clamp meter, when possible, we wrap the 5V or 12V lines multiple times around the clamp meter’s loop, and divide the measurement with the number of loops. All reported power and performance measurements are an average over multiple trials.

The last two columns in Table 7.1 denote the minimum and maximum power consumption of each component. To estimate the total CPU power at load, we run a CPU-intensive computation with a small memory footprint on each core, and attribute the increase in system power from idle entirely to the CPU. Although memory is responsible for 20% of idle power consumption (Figure 7.1, Section 7.1), we were not able to measure noticeable variations in power consumption using different workloads that varied the amount of memory accessed and the access patterns applied (sequential vs random memory accesses). Therefore, throughout this study, we assume that the only way to vary memory power use is by physically removing memory modules.

All components other than CPUs, disks, and RAM, are represented as “system board” in Table 7.1 and collectively consume 54W (again, we were not able to measure noticeable differences between minimum and maximum power). While these components include fans, the network card, the GPU, and the motherboard, the majority of the power budget, according to manufacturer specs, is attributed to the memory and IO controller units on the motherboard (we did not exercise the full capabilities of the GPU as we were always remotely connected to the test machine). The total power consumption of our system in idle state was measured at 156W.

Note that, from a power consumption perspective, we believe our hardware setup closely approximates a node in production-use scale-out architectures. We purposely configured the system with 16GB of RAM and not 32GB, as we only had available DIMMs of 4GB each; these DIMMs consume the
same power as the more expensive 8GB ones, which would make it possible to have 32GB of RAM at a 40W power budget for main memory. Next-generation chip processes could easily double that amount for the same power budget. Our 8 disks provide an aggregate of 1.5GB/sec bandwidth, and adding four more SSDs could yield an additional 1GB/sec at only a 3.5% increase in total server power.

For each row in Table 7.1, the difference between maximum and minimum power consumption denotes the dynamic power range (maximum increase in power consumption) attributed to a specific component. Note however, that dynamic power range does not necessarily reveal opportunities for energy saving, it only indicates the energy saving potential. In general, energy inefficiency stems from systems’ poor energy proportionality, i.e., power use does not increase in proportion with performance, which is due to the poor proportionality of individual hardware components. In order to analyze the energy efficiency of database systems, it is important to study first how power consumption varies as a function of each component’s utilization. Next, we examine in isolation the two components that exhibit dynamic power range: disks and CPUs.

### 7.3.1 Disks

Our database server utilizes both enterprise hard drives (HDDs) and solid state drives (SSDs). Unlike memory, these components exhibit higher power variability, being responsible for 15% of total active power. One common characteristic in both storage technologies is the existence of two operational states, idle and active, each consuming a different amount of power. What is not clear from past work is the energy proportionality (if any) of these devices while executing database-centric operations.

To study the energy proportionality of SSDs and HDDs, we use a minimum overhead row-store/column-store kernel [50] that emulates a row-based or column-based database-storage engine and can also perform simple predicate evaluation and aggregations. The kernel utilizes direct IO to reduce the CPU overhead while accessing data blocks and asynchronous IO to improve device throughput.

To measure the power consumption of SSDs as a function of device utilization, we configured these devices in RAID-0 (striping) and we used our kernel to read a 100GB file sequentially; the block size was set at 128KB. Device utilization was computed as the ratio of measured to maximum throughput. Predicates of increasing complexity were progressively applied to introduce CPU overhead that effectively reduced the throughput measured. Again, we isolated SSD power using the clamp meter. In
Figure 7.5, we plot the power use of the four SSD devices as a function of device utilization. As Figure 7.5 clearly demonstrates, SSDs exhibit perfect energy proportionality; they consume almost nothing with no load and exhibit linear power increases with additional load.

Figure 7.5: Power consumption of SSDs and HDDs as a function of device utilization.

Figure 7.5 also shows the power profile of HDDs measured using the same procedure. HDDs are clearly not energy proportional because they consume nearly half the power at idle state and approximately 80% of maximum power when the device becomes active. The initial power cost comes from spinning the disk platter, and we speculate the second jump comes from exercising additional circuitry and caches. Thereafter, HDDs exhibit a linear increase with utilization. As shown in previous work, the poor proportionality of drives can significantly reduce energy efficiency, especially when these devices are under-utilized [76].

7.3.2 CPUs

Interestingly, the CPUs are responsible for 85% of the power increase at load in our system. Several power models have been proposed that assume a linear correlation between CPU utilization and power consumption [89]. However, linear models for total CPU power are good approximations under very specific conditions: a) operations are CPU-bound, b) there are no shared resources among CPU cores, and c) no power management techniques are applied.

These conditions, however, do not hold for database operations as these are not always CPU-bound. Additionally, in modern multi-core CPUs the processing elements (cores) typically share several re-
sources such as part of the cache hierarchy and the memory bus. Furthermore, hardware and software power management techniques are commonly applied. Modern CPUs dynamically vary CPU frequency depending on the workload applied. At the software level, operating systems utilize energy-saving scheduling techniques that minimize CPU power consumption.

Given the aforementioned conditions, it should come as no surprise that linear power models do not accurately predict the power use of modern CPUs running data management tasks [89]. Our goal in this work is not to develop a new power model for modern CPUs. Instead, we are interested in understanding a) how CPU power is affected by database operations and b) the efficacy of hardware and software power management methods in this context. Towards that goal, we developed a set of micro-benchmarks that perform three classes of database operators, namely hashing, sorting, and scans. The micro-benchmarks are designed to exercise all cores as well as their shared resources, such as CPU cache and memory bus.

Micro-benchmarks

The first micro-benchmark is a custom join kernel that implements a CMP-adaptation of a cache-conscious hash join algorithm [96] for computing the join of two memory-resident relations in parallel. Like the grace hash join, our join kernel proceeds in three steps: a) partitioning, b) build, and c) probe. In the partitioning step, we divide the relations into partitions such that each build partition with its hash index fits into L2 cache (3MB per core). For each partition, we populate a hash index with the build tuples and probe the index with the other relation. Each step is executed in parallel by all cores via horizontal partitioning of the input relations and the intermediate partitions. The build relation holds 100K tuples, and the probe relation holds 20M 100-byte tuples.

The second micro-benchmark consists of a sort kernel that implements two in-memory parallel sorting algorithms tailored to CMP architectures. Both sorting algorithms are cache-conscious variants of AlphaSort [81]. The first variant, called AlphaSort-S, produces cache-sized sorted runs in parallel and then merges these runs using a serial merging phase. The second variant, called AlphaSort-P, applies a parallel merging phase. The main difference between AlphaSort-S and AlphaSort-P is that the latter exercises all processing elements throughout the entire execution of the sorting task. The input to both sorts are 120M integers (480MB total).

Our last micro-benchmark for measuring CPU power is the scan kernel used in Section 7.3.1. In
this section, we use it to scan both uncompressed rows in memory and compressed columns on disk. To run these scans in parallel, we simply issue multiple simultaneous scans on different files. The row scan operates over 60M 32-byte rows (1.9 GB total, in-memory), and the column scan runs over 60M 4-byte integers compressed using FOR-delta [50], for a total 64MB on-disk. Each run includes multiple scan iterations (for disk-resident files, direct IO ensures no Linux-cache buffering), to allow our Watt meter to collect several measurements.

Analyzing Power Consumption

In this section, we focus mainly on CPU power consumption of the micro-benchmarks running under different scheduling strategies and frequency settings. Section 7.4 examines energy efficiency in greater detail. To show the effects of process scheduling, we try two different policies: energy-efficient and performance-oriented. The former is a policy in the latest Linux kernel that first co-locates processes on one CPU before using the other to save power. The latter is one we implement by spreading processes across CPUs to maximize memory bandwidth. Figure 7.6 shows the difference between energy-efficient (top) and performance-oriented (bottom) scheduling on our CPUs. In all figures, the performance-oriented policy is the default, and the energy-efficient policy is labeled.

![Diagram of process scheduling policies]

Figure 7.6: Different process scheduling policies.

Figure 7.2 (Section 7.1) shows the CPU power for our database operators as we vary the number of cores used. In all cases, each core was fully-utilized when used, and the server automatically adjusted
the frequency. Note the sudden power increases when cores 1 and 2 are used in the performance-oriented policy and when cores 1 and 5 are used in the energy-efficient policy. These are the fixed power costs as the CPUs transition from idle states to active. Looking closer at (b), we see that these up-front costs exist for all operators but are much more pronounced for the row scan and hash join.

Although we cannot provide a conclusive explanation for this effect, using Intel’s Vtune performance profiling software, we found that operators with those high up-front costs also exhibit higher memory bus utilization. Figure 7.7 presents the bus utilization of each operator as we vary the number of cores and the scheduling policy. It shows discontinuities at the same points as Figure 7.2, and the bus is nearly saturated once both CPUs are active for row scan and hash join. We speculate that these operators activate and put more stress on the memory subsystem of the CPU, leading to increased power consumption.

These experiments lead to two important conclusions. First, CPU power is not a linear function of the number of cores used (which is what we mean by CPU utilization). Second, for a fixed configuration (number of cores, scheduling policy), different operators may differ significantly (60% in our experiment) in power consumption. Thus, simple models based purely on utilization are not suitable for predicting the CPU power, let alone system power, when running database operators.

In the next set of experiments, we focus on the power consumption of each individual operator, starting with the join kernel. We experimented with different number of cores, scheduling policies, and CPU frequencies. Our CPU can operate at two frequencies: 2GHz and 2.6GHz, which we manually adjusted and fixed for these experiments. Figure 7.8 shows the power profiles of the hash join as we varied these parameters. It also shows the same hash join but with a larger build relation of 20M tuples (640 MB), causing each partition to be larger than the L2 cache (non-cache conscious). This case simulates both a non-cache conscious hash join and an in-memory random index scan. Note, unlike the previous graphs, these are parametric graphs in which both axes depend on the independent variable, number of cores used. The performance (x-axis) is the inverse of the runtime.

There are some interesting effects to note. First, the curves are all concave down (bend down) except when the energy efficient policy activates the second CPU. This is because once one core on one of our CPUs is used, all cores on that CPU transition from idle to active power states. In addition, the CPU activates and exercises other needed shared components like caches and memory interfaces.
Thus, we see large power increases when a CPU becomes active, and additional performance comes at little added power cost. Second, lower frequency uses less power but also runs slower, which does not translate into much higher efficiencies as we show later in Section 7.4. Third, for the non-cache conscious join, the power use is lower for the high frequency curve compared to the cache-conscious join, but the low frequency curves are in a similar power range. This suggests that the cores on the non-cache conscious join are probably stalled on the cache and thus cannot effectively use the additional cycles at high frequencies.

Figure 7.9 shows our experiments with the sort kernel. We omit the graphs with energy efficient scheduling as it did not make much of a difference. AlphaSort-P exhibits an almost-linear increase in power consumption as performance increases. On the other hand, AlphaSort-S consumes comparatively less power because when more cores are used, the fraction of time spent on the serial phase increases, and so most of the cores remain underutilized. In both cases, the lower frequency does not make a significant enough difference in power to justify the decreased performance.

Figure 7.10 shows the same experiments for the scan operators. We see that in-memory row scan has a highly non-linear profile, with several performance points overlapping with each other, unlike any other operator. For clarity we only plot results for the two scheduling policies under high frequency; the
low frequency results have a similar shape, shifted towards left and down. This graph now fully explains the power-use profile of in-memory row scan: For performance-oriented CPU scheduling (solid square points, “Hi-freq”), going from one to two parallel scans (1 core to 2 cores in two different CPUs) doubles performance with a large increase in power (40W, which is comparable to the 44W increase from idle to 1 core). The next two points offer smaller increases in performance and power, as the memory bus to
each CPU becomes the bottleneck. With four parallel scans, the memory bus in both CPUs saturates and additional scans offer no increases in performance or power (hence the overlap of all solid square points for 4,5,6,7, and 8 cores). With energy-efficient scheduling (empty square points, “Hi-freq EE-sched”) the point overlap comes early, for 2,3, and 4 cores, as all these scans share the memory bus of the same CPU. For 5 cores there is again a jump in CPU power as the second CPU (and its memory bus) are activated. Subsequent small increases in performance, when close to peak power, are due to the fact we measure average scan completion time, and some scans finish earlier in the case of EE-scheduling.

Note that hash joins, like row scans, also highly utilize the memory bus (Figure 7.7), but are able to continue increasing performance with each additional core used, as each core can work on a cache-resident data set. Compressed disk-resident column scans (right part of Figure 7.10) behave like the sorts, again with the scheduling policy not making much difference. These scans are not bound by the memory bus (they hardly saturate the I/O bus on the four SSDs) as they require more CPU cycles for each byte read (due to both columnar storage and compression). Once again, though, in all scan cases the best performing high-frequency point is the most energy efficient.
7.4 Energy vs Performance

In this section, we take a more holistic approach and study how hardware and software “knobs” affect the energy efficiency of database workloads. Towards that goal, we not only revisit our micro-benchmarks from an energy efficiency perspective, but also experiment with a variety of queries on full-fledged database engines: PostgreSQL and commercial System-X.

Although database systems are notorious for the plethora of configurable parameters that they support, in this section, we select those that potentially have the greatest impact on energy efficiency. In particular, we vary the following database-level “knobs”: a) algorithm/plan selection, b) intra-operator parallelism (# of cores running a single operator), c) inter-query parallelism (# of independent queries running in parallel), d) physical layout (row vs column scans), e) storage layout (striping), and f) choice of storage medium (HDD vs. SDD). We also vary the platform-level knobs from before: scheduling policies and frequency settings.

7.4.1 Algorithm/Plan Selection

Recent work has suggested that to find plans that achieve improved energy efficiency, we will need to redesign optimizers to model energy costs and re-target their plan selection criteria [51, 65]. In this section, we test this assumption by measuring the energy efficiency of a wide range of queries that exercise all components of a database server. We use two popular, feature-rich databases: PostgreSQL, an open-source engine, and System-X, a commercial product. These engines implement a wide selection of query evaluation algorithms and utilize a cost-based optimizer. Additionally, they provide interfaces or “hints” for influencing choices during query optimization. After running many complex queries including those from the TPC-H benchmark, we present a sample of the results that summarize our most important findings.

Access Methods

Access methods are the fundamental building blocks of every query plan, and, in many cases, they determine the performance of a query. So naturally, we start by studying how different access methods affect energy efficiency. We measured the performance and energy efficiency of a single (non-parallelized)
scan query: “select * from LINEITEM where L.ORDERKEY < value”. We ran all queries over TPC-H tables against a 10GB TPC-H database. We ran this query using different selectivities, access methods (index scan, sequential scan) and storage configurations on both systems. We tried six storage configurations, listed in order of increasing sequential bandwidth: HDD-1, HDD-2, SSD-1, SSD-2, HDD-4, SSD-4, where the number indicates the number of drives in RAID-0 (striping).

Figure 7.11: Performance and energy-efficiency of access methods in PostgreSQL.

Figure 7.11 presents the energy efficiency and performance of the scan query running in PostgreSQL. Note, this again is a parametric graph in which the axes depend on the storage configuration and access method. The left and right graphs show the results for selectivities 10% and 100%, respectively. These graphs clearly show that energy efficiency and performance are linearly correlated. This happens because the dynamic power range among these points is small, only 19% of the minimum 165W. Thus, power remains relatively constant and energy efficiency varies directly with performance. Interestingly, this graph shows isolated cases that deviate from this relationship. For example, consider the second and third best performing points in the right graph, corresponding to sequential scans on HDD-4 and SSD-2, respectively. HDD-4 provides marginally better bandwidth, but uses 7W more power. The net effect is that the additional power is not worth the bandwidth gains. Nonetheless, the overall efficiency gains are small, less than 6%. We saw similar results for this query in System-X.
Compression

Figure 7.4(b) from Section 1 shows the energy efficiency of our scan kernel running disk-resident column scans using different levels of compression. Again, we see that when we consider total power, better performance implies better efficiency. If, however, we consider only CPU and disk power, the added power from heavy compression is not worth the performance benefit, though the efficiency gains for the slower algorithm are small.

![Figure 7.12: Performance and energy-efficiency of join algorithms in PostgreSQL.](image)

Join Algorithms

Since joins are one of the central operations in database systems, we next study the efficiency of different join algorithms. To do so, we issued the following (non-parallelized) equi-join query: “select * from LINEITEM, ORDERS where L.ORDERKEY = O.ORDERKEY and L.ORDERKEY < k”. We used \( k \) to control the join selectivity. We ran this query on both systems using different selectivities, join algorithms (sort merge and hash join), and hardware configurations (CPU frequency, storage system).

Figure 7.12 shows the energy efficiency and performance of these experiments in PostgreSQL for 10% selectivity. Once again we see a linear relationship because the dynamic power range among these configurations is small, only 14% of the minimum (169W) of these. These results are consistent across different selectivities and database engines. Although deviations from the monotonic relationship
between performance and energy efficiency exist (e.g., see circle in Figure 7.12), the difference in efficiency is less than 4%.

Complex Queries and Join Orderings

Given that we saw strong linear relationships for access methods and joins, we hypothesized that more complex queries with different join orders would exhibit the same behavior. To test this assumption, we considered TPC-H Q5 which is a complex five-way join query with sorting and aggregation. We ran this single (non-parallel) query on PostgreSQL and System-X and varied CPU frequency, storage layout (HDD-2, HDD-4, SSD-4), and join orders. Figure 7.13 shows, as expected, a linear relationship for System X. All points with the same shape are different join orders for a particular hardware setting. Again, the dynamic power range among all these configurations was small, 13% of the minimum 174W. PostgreSQL showed similar results.

For all non-parallel queries (single process using only one core), variations in CPU power from different single core utilizations combined with variations in storage power do not move the needle compared to the fixed system power costs. These results suggest that in this case, there is no need to change the optimizer.

![Figure 7.13: Performance and energy efficiency of different query plans for executing TPC-H Q5 on System-X.](image-url)
7.4.2 Intra-operator Parallelism

With the advent of chip multiprocessors, there has been a continuous effort to leverage on-chip parallelism in order to improve the performance of computationally intensive tasks. In this context, several database operators have been optimized for CMP architectures. Utilizing more cores typically improves the performance of a database operator, but it also increases power consumption. Hence, we next examine the energy efficiency of intra-operator parallelism in the context of CMPs. To do so, we revisit the parallel micro-benchmarks presented in Section 7.3 from an energy efficiency perspective.

Figure 7.14: Energy efficiency vs. performance for parallel in-memory hash join, as the number of CPU cores used varies from 1 to 8.
Parallel hash join

Figure 7.14 shows the energy efficiency and performance of our cache-conscious hash join as we varied the number of cores used. The left and right graphs compute energy efficiency using only the CPU power and total system power, respectively. The right graph shows a strong linear correlation between energy efficiency and performance, and for points near any given performance level, the efficiency hardly varies.

The left graph shows the energy efficiency curve that only includes CPU power. Again, we see a slightly weaker, but still linear relationship. Also, even though the best performing is not quite the most efficient, the difference is small (see top circle). The other circle shows the points where the energy efficient scheduling moves from using one CPU to two (4 cores to 5). Although performance improves in this case, energy efficiency remains the same because the added power of the other CPU cancels out the performance benefits.

![Figure 7.15: Energy efficiency of parallel in-memory sorting.](image)

Parallel Sorts

In the previous experiment, our join kernel was able to utilize all cores for the entire execution, but this is not always the case. Serial computation and load balancing issues may underutilize CMPs. To understand the energy efficiency implications for these cases, we revisit the parallel sort micro-benchmarks from Section 7.3. Figure 7.15 shows the energy efficiency and performance of AlphaSort-S
and AlphaSort-P, the serial and parallel merge versions of the sort kernel. Although AlphaSort-S uses less power than AlphaSort-P, it also further underutilizes the CPU as the number of cores increases, which wastes power and leads to much worse energy efficiency. This experiment shows that to reach maximum efficiency, we not only should optimize for performance on a single core, but also optimize for performance by fully utilizing all cores.

Although all these parallel operators show roughly the same energy efficiency-performance behavior as the non-parallel ones, their dynamic power range is large, nearly $92W$. This is $60\%$ of system idle power and $191\%$ of CPU idle power. We initially expected large dynamic range to allow opportunities for trading energy efficiency for performance. Section 7.5 explains, however, that we see the same behavior because of the concave down nature of the power-performance curves (see Figures 7.8 and 7.9) as well as the fixed power costs. For such curves, the relative performance increases are worth the added relative power, so energy efficiency improves with performance.

### 7.4.3 Inter-Query Parallelism

In all the previous sections, we studied energy efficiency during the execution of a single query. However, database systems commonly execute multiple queries at the same time. In this section we study how energy efficiency is affected as we increase the concurrency (# of queries running parallel) in the system. Initially, we consider simple scan queries that mainly exercise I/O and the memory sub-system, and then we progressively examine more complex queries that stress all the components of a database server.

We first study the effects of running multiple scan queries concurrently. Figure 7.4(a) from Section 1 shows the energy efficiency and performance of our compressed column scan kernel from Section 7.3 on four SSDs. In this experiment, we varied the frequency, compression method, and number of simultaneous scans from 1-8 (i.e., number of cores used). Once again, we see a linear energy efficiency-performance relationship even though the dynamic power range is large.

We next ran many concurrent PostgreSQL scan queries from Section 7.4.1 under various hardware and software configurations. Figure 7.16 presents the most interesting results. It shows the performance and energy efficiency of a main-memory clustered index scan (top) and an unclustered index scan on 4 SSDs (bottom). In both cases, we see that at a certain concurrency level (8 for clustered, and 16
for unclustered) the throughput peaks and then drops because we reach the capacity of the database software. Once again, energy efficiency, whether including CPU power only or total power, basically follows performance. Note, for the main memory scan, the energy efficiency peaks at slightly higher concurrency when considering CPU power only, but the difference is small.

Finally, to understand the energy efficiency of concurrent complex queries running on our system, we ran two TPC-H queries, Q12 and Q5 on System X. The former is a single join query with aggregation, and we ran it under various concurrency levels (1-64), storage layouts, and frequencies. Figure 7.17 shows the results for 1% and 10% selectivity. Apart from a few deviations, we see the same linear relationship between energy efficiency and performance.

Figure 7.3 from Section 7.1 shows the results for Q5 as we varied frequency, CPU scheduling, storage layout (in memory, SSD4, HDD2, HDD4), and number of cores used (1-8). We again see a
strong correlation between energy efficiency and performance. Although near peak performance we see some variation in energy efficiencies, the spread is small, less than 10%.

Figure 7.17: Performance and energy-efficiency of parallel TPC-H Q12 queries in System-X.

To summarize, regardless of query complexity and what knobs we use, i.e., access methods, operator algorithms, and type and level of parallelism, energy efficiency and performance go hand in hand.

### 7.5 Analysis and implications

Although Section 7.3 showed that power-use profiles vary significantly from operator to operator, Sections 7.1 and 7.4 showed that regardless of the configuration and execution strategy the most energy-efficient operating points were also the best performing. These results contradict the recent speculation about the need and opportunity for DBMS software to optimize for energy efficiency apart from performance [41, 51], and also contradict recent work that agrees with this speculation [65, 76, 104].

In this section, we explain these disagreements by understanding the conditions under which energy efficiency is optimal and contrast our results with past work in this light. Although our conclusion implies that software-level knob tuning specifically to improve energy efficiency within a single node is not useful, we discuss new challenging single-node problems and promising directions for multi-node techniques.
7.5.1 The balance of power and performance

Recall from equation 7.1 that energy efficiency, defined as the ratio of useful work done to the energy used, is equivalent to the ratio of performance to power. As we add system components one at a time, or use more resources, both the power and performance of the system change. Total power increases, and we expect performance to also improve. At any point, energy efficiency improves when the relative improvement of performance is greater than the relative increase of power:

\[
\frac{\Delta \text{Perf}}{\text{Perf}} > \frac{\Delta \text{Power}}{\text{Power}} \tag{7.2}
\]

When the two sides are equal, we have a balance of the relative improvements of each, and thus maintain the same efficiency. When the relation reverses, we are in a region of decreasing efficiency — a region of diminishing returns. That is, the increased performance from the additional resource does not justify its power cost.

In our system, there are two main reasons for why the most efficient configuration was typically the best performing. First, as Figure 7.1 shows, almost 50% of peak power is consumed at idle. This fixed power cost adds a large constant term to the denominator in equations (7.1) and (7.2) which makes all subsequent relative power increases worth the added performance, especially when the dynamic power...
range is small.

Although significant, idle power is not the sole cause. Figure 7.18(a) recalculates the energy efficiency of System-X from Figure 7.3, this time discounting system board power-use. Even in this case, the opportunities for affecting the energy efficiency are small. Figure 7.18(b) shows the efficiencies using only “active power,” that is by subtracting out all idle power. Even in this unrealistic case, although the points are more dispersed, the spread in energy efficiency among the best performing points is only 30%. Section 7.4 also shows similar results when only considering CPU power.

The second reason for this effect is the shape of the power-performance curves for these workloads. Rearranging the terms in the above equation helps explain why:

$$\frac{1}{EE} = \frac{\text{Power}}{\text{Perf}} > \frac{\Delta \text{Power}}{\Delta \text{Perf}}.$$  (7.3)

The left hand side (inverse of energy efficiency) is the slope of the line from the origin to the current point (configuration) on a power-performance curve. The right hand side is the “tangent” slope, the slope of the line from the current point to the next point that reflects the added resource. It tells whether the curve bends down, remains steady, or bends up. When the curve is concave down (bends down) or remains steady (linear with non-zero power intercept), the tangent slope is smaller, thus, increasing energy efficiency. But, when the power-performance curve is concave up (bends up), there is an optimal balanced efficiency point after which the tangent slope is larger, thus decreasing energy efficiency.

The power-performance curves in Section 7.3 of our components are all concave down or steady. The CPU profiles are concave down, especially in the cases of poorly scheduled joins and row scans. This concavity occurs because once the CPUs are active, they activate all cores simultaneously and other resources. As a result, the power “jumps” with each activated CPU and the remaining power increments are smaller. Combining these profiles with proportional SSD profiles still results in concave down profiles. Thus, we find for all our workloads, the most efficient configurations are also the best performing.

Examining two recent efforts that argue for energy efficiency optimizations [65, 104], we see that the analyses are based either on CPU-power only or on the system’s “active power.” In the former, the margin of improvements is greatly reduced by factoring in the power costs of all components. The effect is even more pronounced in the latter because all fixed costs are omitted.
Lastly, the authors in [76] experimented with a large shared-memory production server running TPC-H. The improvements in energy efficiency came when they removed up to four fifths of their 200+ HDDs. Their power-performance curve as the number of disks varied was concave up. We suspect that as disks were added, average disk utilization dropped leading to only marginal improvements in performance. Meanwhile, each disk contributed a fixed power cost, and after a point, the added performance was not worth it. In contrast, all fixed power costs were upfront in our system, not as additional resources were used. We speculate that if that shared memory system had used SSDs, the energy proportionality of SSDs would have enabled them to avoid these inefficiencies, and it too would exhibit maximum energy efficiency at best performance.

### 7.5.2 Implications for database computing

**One (hardware) size fits all?**

Although the notion of “one-size-does-not-fit-all” in database software design is steadily gaining acceptance, it comes as a surprise that server hardware designs show a striking adherence to the one-size-fits-all philosophy. The last significant change in data-management server architecture was the shift to shared-nothing clusters, largely motivated by price-performance considerations. For modern internet-scale applications, scaling out by orders of magnitude is significantly cheaper than scaling up a large SMP server.

Energy costs, however, have the potential to be the catalyst towards yet another turn in hardware design of large-scale platforms. Recent work [6, 90] has demonstrated that, for random retrieval and data analysis application, significantly different node architectures would be better from a power-performance perspective. If indeed we are heading towards a collection of heterogeneous nodes in a data center, each cluster optimized for specific classes of applications, then what characteristics should these nodes have for database applications? We believe this is a promising direction for future work. Based on the results from Section 7.3 we can add the following to the list of desired characteristics:

(a) **CPUs:** high-parallelism, low-frequency, small caches, simpler designs. Database systems are already well-equipped to handle available hardware parallelism. Since CPUs pay a high start-up power cost, after which, each additional active core comes at smaller relative increases in power, database-
optimized architectures should leverage high levels of parallelism within and across CPUs. Although in our experiments lowering the CPU frequency did not improve energy efficiency, it did not worsen it either. This is important, because a lower operating frequency can reduce the fixed CPU power cost, which may allow for more energy tradeoffs. Lastly, database operators do not always need the CPU cache (or they can be made to work with smaller caches). Therefore, simpler CPU designs with fewer levels in the memory hierarchy may lead to more efficient designs for database workloads.

(b) Solid state storage. SSDs (and we expect any other upcoming electronic non-volatile storage technology) are already close to ideal energy proportionality, as we saw in Section 7.3. Therefore any energy-efficient node architecture should be based on SSDs and not on mechanical HDDs which necessarily carry a fixed power cost (due to disk spinning). Since we expect HDDs to remain for at least a few years the cheapest option for raw storage capacity, it will be important to focus on techniques that reduce storage requirements to lower the initial investment in solid state storage. Example of such techniques are: compression (even lossy), de-duplication, giving up meta-data storage structures, and storing data summaries instead of exact data.

Shared-nothing, everything, or in-between?

Since we have shown limited opportunity for software-based energy optimizations within a node, we look to the multi-node case. There are two traditional designs to consider: shared nothing and shared disk. A promising research direction is to consider energy optimizations across an entire cluster. Clusters provide extreme scalability. They are, however, often over-provisioned for most tasks, and resources go unused wasting power [32]. Unfortunately, data availability is important, and in a shared nothing environment storage and computing power are coupled. We must find a way to turn off nodes and still maintain data access when the cluster is underutilized. Physical design, replication, hardware-level mechanisms, and data movement are all ripe for investigation. Although current ongoing work in this area considers physical design [68], little work considers dynamic data movement. Databases can easily reassign work among nodes, but the cost of moving data around may exceed the gain in energy by suspending nodes.

An alternative is to use a shared disk design, which decouples storage from compute. If equipped with SSDs, the storage system becomes more energy proportional, and what remains is affecting com-
compute energy proportionality through consolidation [101]. Our experiments show that operators like hash join can force a CPU to consume power disproportionately to its utilization, so consolidation would help to avoid waste power. To enable this, we would need accurate models for predicting power consumption. Finally, since shared disk is limited in scalability but can offer simplified energy management, a hybrid approach with clusters of shared disk systems might prove to be the most practical.

Controlling peak power

Another important consideration in data centers is peak power consumption. Racks, especially older ones, have limited capacity to deliver power and, when overdrawn, can cause fuses to trip. Peak power use can also increase temperature beyond the capacity to cool or significantly increase the cost of the cooling. As a result, data centers often need to enforce power budgets and rely on coordinated controllers at all levels from the rack to the node to enforce them [85]. At the node level, hardware mechanisms can regulate power quickly, but have little insight into application performance.

A challenging problem is for database software to tune execution to work within a target power envelope while still maximizing performance. Figure 7.19 re-plots the points from the various configurations for Q5 on System-X, this time by showing only configurations that consume power just below 200W. Although any of those configurations lies within the target for power consumption, performance can vary up to 4x. Using software mechanisms to cap power consumption while maximizing performance shows promising potential for future research.

7.6 Conclusions

We have studied the power-performance characteristics of various database analytic workloads on a modern server intended for scale-out architectures. We are the first to detail the power profile of core database operations like scans, hash joins, and sorts. Our results show that the CPU power used by different operators can vary widely, by up 60% for the same CPU utilization, and that CPU power is not linear with utilization. We also experiment with two widely deployed database systems, PostgreSQL and a popular commercial DBMS, varying both software and hardware knobs to understand their energy efficiency effects. In most of our experiments, we found that the best performing configuration was also
the most energy efficient. In the few cases where this did not hold, energy efficiency did not improve by more than 10%. This relationship is a result of large up-front power costs in modern server components.
Chapter 8

Conclusions

In this thesis we presented data processing techniques tailored to modern hardware architectures. We introduced in-memory list intersection algorithms for sorted and unsorted lists designed for the characteristics of chip multiprocessors. We analyzed the performance trade-offs in the execution of complex database queries on CMP architectures. We presented in-memory suffix tree construction algorithms for chip multiprocessors. We proposed query processing techniques for flash-based storage technologies and demonstrated how these techniques can be integrated in existing database systems. Finally, we analyzed the power consumption of database systems and revealed opportunities for improving the energy efficiency of data management systems.

The results from this thesis demonstrate that:

- Existing data processing techniques are not optimized for modern hardware architectures, thereby severely underutilizing hardware resources.

- Hardware-conscious techniques realize the potential for performance improvement offered by modern processor and storage technologies and satisfy the increasing needs for accelerating complex data processing tasks.

- Optimizing the utilization of hardware resources is imperative not only for improving performance but also for improving the energy-efficiency of data management systems.
8.1 Extensions

8.1.1 Designing a Database Query Engine for Chip Multiprocessors

With the number of cores in CMP architectures doubling almost every year, database query engines need to be re-designed to harness on-chip parallelism and overcome limiting performance bottlenecks. In Chapter 4, we presented the design of a database query engine tailored to chip multiprocessors. Apart from the query execution engine, many components of database management systems must be re-designed for modern processor architectures. Of utter importance is the design of architecture-conscious query optimizers.

Traditional query optimizer consider primarily the I/O cost while searching for the optimal plan to execute a query. In the context of modern systems with chip multiprocessors and large memories, I/O cost is not an effective proxy for the performance of many complex data processing tasks. Consequently, it is imperative to revise existing performance models in order to incorporate the performance characteristics of modern hardware architectures.

In addition to query optimization, the problem of resource allocation (e.g. allocating hardware resources to queries) must be re-considered in the context of chip multiprocessors. The utilization of intra-query and inter-query parallelism warrant the design of a new class of resource scheduling algorithms that take into account the characteristics of queries and operators to make optimal resource allocation decisions statically or online.

8.1.2 Database Systems on Flash-based Storage Systems

In Chapter 6, we discussed storage formats and query processing techniques tailored to the performance characteristics of solid state drives. We also demonstrated how these techniques can be integrated into existing database systems. One new consideration for query optimization is that many query execution plans that were not appropriate for HDDs become appropriate for SSDs. In addition to the algorithms presented in this thesis, query optimization may need to reconsider when to use index navigation, including index-nested-loops join and other forms of nested iteration. We expect SSDs to expand the cases in which index-based query execution plans are competitive with set-oriented query execution plans using hash join, hash aggregation, and sorting. In addition, the new efficiency trade-offs of late versus early
materialization greatly expand the set of interesting join permutations for a given query.
Bibliography


[29] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on adaptive set inter-
Verlag.

[30] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and 
David Wood. Implementation techniques for main memory database systems. SIGMOD Rec., 

[31] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-
nothing architecture using probabilistic splitting. In Proceedings of the First International Con-


[33] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of 

[34] Philip Garcia and Henry F. Korth. Database hash-join algorithms on multithreaded computer 

’07: Proceedings of the 3rd international workshop on Data management on new hardware, 
2007.

[36] Buğra Gedik, Philip S. Yu, and Rajesh R. Bordawekar. Executing stream joins on the cell pro-
cessor. In VLDB ’07: Proceedings of the 33rd international conference on Very large data bases, 

[37] Amol Ghoting and Konstantin Makarychev. Serial and parallel methods for i/o efficient suffix tree 
construction. In SIGMOD ’07: Proceedings of the 2009 ACM SIGMOD international conference 
on Management of data, 2009.


[40] Goetz Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In DaMoN ’07: Proceedings of the 3rd international workshop on Data management on new hardware, pages 1–9, New York, NY, USA, 2007. ACM.


