# Analytical Query Execution Optimized for all Layers of Modern Hardware

**Orestis Polychroniou**

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

**COLUMBIA UNIVERSITY**

2018

©2018
Orestis Polychroniou
All Rights Reserved

# ABSTRACT

## Analytical Query Execution Optimized for all Layers of Modern Hardware

## Orestis Polychroniou

Analytical database queries are at the core of business intelligence and decision support. To analyze the vast amounts of data available today, query execution needs to be orders of magnitude faster. Hardware advances have made a profound impact on database design and implementation. The large main memory capacity allows queries to execute exclusively in memory and shifts the bottleneck from disk access to memory bandwidth. In the new setting, to optimize query performance, databases must be aware of an unprecedented multitude of complicated hardware features. This thesis focuses on the design and implementation of highly efficient database systems by optimizing analytical query execution for all layers of modern hardware. The hardware layers include the network across multiple machines, main memory and the NUMA interconnection across multiple processors, the multiple levels of caches across multiple processor cores, and the execution pipeline within each core. For the network layer, we introduce a distributed join algorithm that minimizes the network traffic. For the memory hierarchy, we describe partitioning variants aware to the dynamics of the CPU caches and the NUMA interconnection. To improve the memory access rate of linear scans, we optimize lightweight compression variants and evaluate their trade-offs. To accelerate query execution within the core pipeline, we introduce advanced SIMD vectorization techniques generalizable across multiple operators. We evaluate our algorithms and techniques on both mainstream hardware and on many-integrated-core platforms, and combine our techniques in a new query engine design that can better utilize the features of many-core CPUs. In the era of hardware becoming increasingly parallel and datasets consistently growing in size, this thesis can serve as a compass for developing hardware-conscious databases with truly high-performance analytical query execution.

# Table of Contents


**List of Figures** vi

**List of Tables** x

**1 Introduction** 1

1.1 Modern Hardware Hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.1.1 Mainstream CPUs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.1.2 Cache Hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.1.3 Main Memory and NUMA . . . . . . . . . . . . . . . . . . . . . . . . 6

1.1.4 Network Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.2 Additional Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.2.1 SIMD Vectorization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.2.2 Many Integrated Cores . . . . . . . . . . . . . . . . . . . . . . . . . . 10

1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

1.3.1 Distributed Joins with Minimal Network Traffic . . . . . . . . . . . . 10

1.3.2 In-Memory Partitioning and Large-Scale Sorting . . . . . . . . . . . . 11

1.3.3 Lightweight Compression Alongside Fast Scans . . . . . . . . . . . . 11

1.3.4 Advanced SIMD Vectorization Techniques . . . . . . . . . . . . . . . 12

1.3.5 Query Execution Engines for the Many-Core Era . . . . . . . . . . . 12

1.4 Additional Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

1.4.1 Heavy Hitter Aggregation for SIMD Processors . . . . . . . . . . . . 13

1.4.2 Energy Analysis of Hardware and Software Range Partitioning . . . 13

1.4.3 SIMD-accelerated Regular Expression Matching . . . . . . . . . . . . 14

1.5 Related Work . . . . . 14
1.5.1 Parallel and Distributed Databases . . . . . 14
1.5.2 Batch Distributed Data Processing . . . . . 15
1.5.3 Network-Optimized Databases . . . . . 15
1.5.4 Query Execution Engines . . . . . 16
1.5.5 Main-Memory Partitioning . . . . . 16
1.5.6 In-Memory Joins and Group-by Aggregation . . . . . 17
1.5.7 Lightweight In-Memory Compression . . . . . 17
1.5.8 SIMD Vectorization . . . . . 18
1.5.9 Databases on GPUs and FPGAs . . . . . 18

**2 Distributed Joins with Minimal Network Traffic** **20**
2.1 Introduction . . . . . 20
2.2 Track Join . . . . . 23
2.2.1 2-Phase Track Join . . . . . 23
2.2.2 3-Phase Track Join . . . . . 26
2.2.3 4-Phase Track Join . . . . . 29
2.2.4 Traffic Compression . . . . . 35
2.3 Query Optimization . . . . . 36
2.3.1 Network Cost Model . . . . . 36
2.3.2 Tracking-Aware Hash Join . . . . . 39
2.3.3 Semi-Join Filtering . . . . . 40
2.4 Experimental Evaluation . . . . . 42
2.4.1 Simulations . . . . . 42
2.4.2 Implementation . . . . . 49
2.5 Conclusion . . . . . 53

**3 In-Memory Partitioning and Large-Scale Sorting** **54**
3.1 Introduction . . . . . 54
3.2 Partitioning Variants . . . . . 58
3.2.1 Partitioning In-Cache . . . . . 58

3.2.2 Partitioning Out-of-Cache . . . . . . . . . . . . . . . . . . . . . . . . 59
3.2.3 Partitioning Across NUMA Regions . . . . . . . . . . . . . . . . . . 67
3.2.4 Radix / Hash Histogram . . . . . . . . . . . . . . . . . . . . . . . . 68
3.2.5 Range Histogram . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.3 Sorting Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
3.3.1 Setting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
3.3.2 Radixsort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
3.3.3 Comparison-Based Sorting . . . . . . . . . . . . . . . . . . . . . . 75
3.4 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

**4 Lightweight Compression Alongside Fast Scans 86**

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.2 Horizontal Bit Packing . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
4.2.1 Scalar Packing and Unpacking . . . . . . . . . . . . . . . . . . . . 88
4.2.2 Word-Aligned Scalar Scanning . . . . . . . . . . . . . . . . . . . . 89
4.2.3 SIMD Unpacking and Scanning . . . . . . . . . . . . . . . . . . . . 91
4.3 Vertical Bit Packing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
4.3.1 Scalar Packing and Unpacking . . . . . . . . . . . . . . . . . . . . 93
4.3.2 SIMD Packing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
4.3.3 SIMD Unpacking . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
4.3.4 Selection Scan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
4.4 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 99
4.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

**5 Advanced SIMD Vectorization Techniques 102**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.2 Fundamental Operations for Vectorization . . . . . . . . . . . . . . . . 106
5.2.1 Logical Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.2.2 Gather/Scatter Implementation . . . . . . . . . . . . . . . . . . . . 107
5.2.3 Implementing Selective Loads / Stores . . . . . . . . . . . . . . . . 109

5.2.4 Algorithmic Notation . . . . . . . . . . . . . . . . . . . . . . . . 110
5.3 Selection Scans . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.4 Hash Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
5.4.1 Linear Probing . . . . . . . . . . . . . . . . . . . . . . . . 114
5.4.2 Double Hashing . . . . . . . . . . . . . . . . . . . . . . . . 119
5.4.3 Cuckoo Hashing . . . . . . . . . . . . . . . . . . . . . . . 121
5.5 Bloom Filters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
5.6 Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
5.6.1 Radix / Hash Histogram . . . . . . . . . . . . . . . . . . . 128
5.6.2 Range Histogram . . . . . . . . . . . . . . . . . . . . . . . 129
5.6.3 Data Shuffling . . . . . . . . . . . . . . . . . . . . . . . . 130
5.6.4 Buffered Data Shuffling . . . . . . . . . . . . . . . . . . . 133
5.7 Sorting and Hash Joins . . . . . . . . . . . . . . . . . . . . . . . . 136
5.8 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . 137
5.8.1 Selection Scans . . . . . . . . . . . . . . . . . . . . . . . . 138
5.8.2 Hash Tables . . . . . . . . . . . . . . . . . . . . . . . . . . 139
5.8.3 Bloom Filters . . . . . . . . . . . . . . . . . . . . . . . . . 142
5.8.4 Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . . 143
5.8.5 Sorting and Hash Joins . . . . . . . . . . . . . . . . . . . . 144
5.9 SIMD CPUs and SIMT GPUs . . . . . . . . . . . . . . . . . . . . . 148
5.10 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148

**6 Query Execution Engines for the Many-Core Era 149**
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
6.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
6.2.1 Mainstream & Many-Core Processors . . . . . . . . . . . . . 151
6.2.2 Analytical Query Execution Engines . . . . . . . . . . . . . 154
6.3 Vectorized Code Generation . . . . . . . . . . . . . . . . . . . . . . 155
6.3.1 Scalar Code Generation . . . . . . . . . . . . . . . . . . . . 155
6.3.2 SIMD Code Generation . . . . . . . . . . . . . . . . . . . . 156
6.4 Proposed Query Engine Design . . . . . . . . . . . . . . . . . . . . 160

6.4.1 Selection Scans . . . 161
6.4.2 Compression . . . 164
6.4.3 Hash Joins . . . 167
6.4.4 Partitioning . . . 171
6.4.5 Materialization . . . 174
6.4.6 Group-by Aggregation . . . 176
6.5 Experimental Evaluation . . . 183
6.6 Conclusion . . . 190

**7 Conclusions and Future Work** **191**

**Bibliography** **198**

# List of Figures

1.1 SIMD instruction examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1 Example of network transfers in hash join and track join per distinct join key 29
2.2 Example of 4-phase track join optimal schedule generation per distinct join key . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.3 Synthetic $10^9$ unique $R$ tuples $\bowtie$ $10^9$ unique $S$ tuples ($1 \times 1$ key matches) . 42
2.4 Synthetic $2 \cdot 10^8$ unique $R$ tuples $\bowtie$ $10^9$ non-unique $S$ tuples ($1 \times 5$ key matches) 43
2.5 Synthetic $4 \cdot 10^7$ unique $R$ tuples $\bowtie$ $2 \cdot 10^8$ $S$ tuples ($1 \times 5$ key matches) . . 43
2.6 Synthetic $2 \cdot 10^8$ $R$ tuples $\bowtie$ $2 \cdot 10^8$ $S$ tuples with $4 \cdot 10^7$ unique keys ($5 \times 5$ key matches intra-table collocated) . . . . . . . . . . . . . . . . . . . . . . . . 44
2.7 Synthetic $2 \cdot 10^8$ $R$ tuples $\bowtie$ $2 \cdot 10^8$ $S$ tuples with $4 \cdot 10^7$ unique keys ($5 \times 5$ matches inter-table collocated and intra-table collocated) . . . . . . . . . . 45
2.8 Slowest join of slowest query (Q1) of workload $X$ (original data placement) 46
2.9 Slowest join of slowest query (Q1) of workload $X$ (shuffled data placement) 46
2.10 Common slowest join in slowest queries 1–5 of $X$ (dictionary compressed) . 47
2.11 Slowest join in slowest query of workload $Y$ using variable byte encoding . . 48

3.1 Partitioning variants and contributions . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.2 SIMD comb-sort (4-way) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
3.3 Throughput of out-of-cache radix partitioning ($10^{10}$ tuples) . . . . . . . . . 79
3.4 Partitioning throughput with uniform random and Zipf data ($\theta = 1.2$) . . . 80
3.5 Scalability of out-of-cache radix partitioning ($10^9$ tuples, 1024-way) . . . . . 80
3.6 Throughput of histogram generation ($10^{10}$ keys) . . . . . . . . . . . . . . . . 81

3.7 Throughput of all three sorting methods . . . . . . . . . . . . . . . . . . . . 81
3.8 Sorting scalability ($10^9$ tuples, 32-bit keys and 32-bit payloads) . . . . . . . 82
3.9 Execution time per sorting algorithm phase ($10^{10}$ tuples) . . . . . . . . . . . 83
3.10 NUMA-aware vs. NUMA-oblivious sorting ($10^{10}$ tuples) . . . . . . . . . . . 83
3.11 Performance of scalar and SIMD comb-sort (32-bit keys and 32-bit payloads) 84

4.1 Throughput of packing, unpacking, and scanning for horizontal and vertical bit packing using scalar and SIMD code (multi-threaded) . . . . . . . . . . 100
4.2 Throughput of packing, unpacking, and scanning for horizontal and vertical bit packing using scalar and SIMD code (single-threaded) . . . . . . . . . . 101

5.1 Selective store operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.2 Selective load operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.3 Gather operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.4 Scatter operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.5 Selection scan throughput on KNC and Haswell (32-bit key and 32-bit payload) 139
5.6 Throughput of hash table probing on KNC and Haswell using linear probing (LP) and double hashing (DH) (shared, 32-bit key input $\rightarrow$ 32-bit payload output) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
5.7 Throughput of hash table probing on KNC and Haswell using cuckoo hashing (shared, 32-bit key input $\rightarrow$ 32-bit payload output) . . . . . . . . . . . . . . 140
5.8 Throughput of building and probing hash tables on KNC using linear probing (LP), double hashing (DP), and cuckoo hashing (CH) (1:1 build–probe, private, 32-bit keys and 32-bit payloads both inputs) . . . . . . . . . . . . 141
5.9 Throughput of building and probing on KNC using linear probing (LP), double hashing (DH), and cuckoo hashing (CH) (1:10 build–probe, L1-resident, private, 32-bit keys and 32-bit payloads both inputs) . . . . . . . . . . . . 141
5.10 Throughput of Bloom filter probing on KNC (5 hash functions, 10 Bloom filter bits / tuple, 5% selectivity, 32-bit key and 32-bit payload) . . . . . . . 142
5.11 Throughput of radix and hash histogram generation on KNC (32-bit keys) . 143
5.12 Throughput of range partitioning function on KNC and Haswell (32-bit keys) 143

5.13 Throughput of data shuffling for radix partitioning on KNC (private, out-of-cache, 32-bit key and 32-bit payload) . . . . . . . . . . . . . . . . . . . . . . 144
5.14 Performance of sorting (LSB radixsort) on KNC . . . . . . . . . . . . . . . 145
5.15 Hash join performance on KNC ($2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key and 32-bit payload) 145
5.16 scalability of LSB radixsort and partitioned hash join on KNC ($4 \cdot 10^8$ and $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key and 32-bit payload, log/log scale) . . . . . . . . . 146
5.17 LSB radixsort and partitioned hash join on 1 61-core KNC co-processor (Intel Xeon Phi 7120P) vs. 4 8-core Sandy Bridge CPUs (Intel Xeon E5 4620) (sort $4 \cdot 10^8$ tuples, join $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ tuples, 32-bit key and 32-bit payload) . . 146
5.18 Radixsort with varying payloads on KNC ($2 \cdot 10^8$ tuples, 32-bit key) . . . . 147
5.19 Hash join with varying payloads on KNC ($10^7 \bowtie 10^8$ tuples, 32-bit keys) . . 147

6.1 Throughput of non-sequential accesses . . . . . . . . . . . . . . . . . . . . . 153
6.2 Running time of example query using code generation (10% selectivity) . . 159
6.3 Running time of example query using code generation (90% selectivity) . . 159
6.4 Example of selection scan (already determined tuples are not skipped) . . . 163
6.5 Hash join F⋈A from the example query with and without partitioning . . . 175
6.6 Example of mapping hashes to group-ids . . . . . . . . . . . . . . . . . . . 176
6.7 Example of resolving hash collisions . . . . . . . . . . . . . . . . . . . . . . 179
6.8 Selection scans on uncompressed data (1–3 32-bit key columns, 1 predicate per key column, 1 32-bit payload column) . . . . . . . . . . . . . . . . . . . 183
6.9 Selection scans on compressed data (1–3 32-bit key columns, 1 predicate per key column, 1 32-bit payload column) . . . . . . . . . . . . . . . . . . . . . 184
6.10 Expression tree for selection on part table from TPC-H Q19 (0.24% selectivity) 184
6.11 Selection on part table from TPC-H Q19 . . . . . . . . . . . . . . . . . . . 185
6.12 Partition of multiple columns and types . . . . . . . . . . . . . . . . . . . . 185
6.13 Hash join on synthetic data (1 key, 1–3 payloads, 128M ⋈ 32M tuples) . . . 186
6.14 Hash joins using the core TPC-H tables . . . . . . . . . . . . . . . . . . . . 187
6.15 Group-by aggregation on synthetic data (1 32-bit group-by key, 1–3 32-bit payloads, 1 sum aggregate per payload, count(*), 256M tuples) . . . . . . . 188
6.16 Step breakdown for join and aggregation . . . . . . . . . . . . . . . . . . . 189

6.17 Group-by aggregation step from TPC-H Q1 . . . . . . . . . . . . . . . . . . . 190

# List of Tables

2.1 Column details of $R$ and $S$ tables joined in Q1 . . . . . . . . . . . . . . . . . 45
2.2 CPU and network time of the slowest join of the slowest query of $X$ and $Y$ 50
2.3 Distributed hash join time per step (in seconds) . . . . . . . . . . . . . . . . . 51
2.4 4-phase track join time per step (in seconds) . . . . . . . . . . . . . . . . . . . 52

5.1 Experimental platforms for advanced SIMD vectorization experiments . . . 137

First, I would like to thank my advisor Kenneth Ross for accepting me as a PhD student, discussing with me for endless hours, giving me complete intellectual freedom, and generally tolerating me over these last 6 years. I would also like to thank Luis Gravano and Eugene Wu for all the discussions that took place in the Columbia Database Group. I would also like to thank Martha Kim and Stratos Idreos for their participation in the committee and interest in the work.

Second, I would like to thank current and former members of the Columbia Database Group at Columbia: Eva Sitaridi, Fotis Psallidas, John Paparrizos, Pablo Barrio, Bingyi Cao, Jiacheng Yang, and Wandga Zhang, as well as other friends from Columbia: Dimitris Paparas, Evangelia Skiani, Marios Pomonis, and Vasileios P. Kemerlis.

Third, I would like to thank Yannis Ioannidis, Yannis Smaragdakis, Alex Delis, and Panagiotis Stamatopoulos for introducing me to Computer Science and Databases and generally nurturing me during my undergraduate years in the University of Athens.

Fourth, I would like to thank Eric Sedlar, Nipun Agarwal, Rajkumar Sen, Arun Raghavan, and John Kowtko for the collaboration during my internships at the Oracle Labs.

Finally, I would like to thank Ippokratis Pandis and Michalis Petropoulos from Amazon Web Services.

To my family and closest friends.

# Chapter 1

# Introduction

In the era with large volumes of data easily accessible and business decisions being data driven, analytical database queries are at the core of business intelligence and decision support. Database management systems have traditionally focused on transactions, supporting analytical queries, but executing them at very low efficiency compared to what is achievable by modern hardware. The quality of decision support depends on the amount of data we process in time. Thus, the need for highly efficient analytical query execution is apparent.

Modern hardware advances have fundamentally reshaped the design and implementation of database management systems. The advent of large main memory capacity has allowed small to medium-sized databases to remain main-memory resident. The bottleneck shifts from disk access, the metric that database systems have traditionally optimized for, to memory bandwidth [**?**; Manegold *et al.*, 2000a]. In the context of distributed databases, the bottleneck is the network bandwidth. The potential performance improvement supported by the hardware is now orders of magnitude higher compared to disk-oriented databases and marks the one-size-fits-all database model as obsolete [Stonebraker *et al.*, 2007].

Optimizing database systems by focusing on specific workloads is necessary to achieve the efficiency potential provided by modern hardware [Stonebraker *et al.*, 2005; Stonebraker *et al.*, 2007; Cudre-Mauroux *et al.*, 2009; Raman *et al.*, 2013; Gupta *et al.*, 2015; Dageville *et al.*, 2016; Verbitski *et al.*, 2017]. Workload specialization include transactional (OLTP) databases, analytical (OLAP) databases, and others types such as array databases. Analytical queries are fundamentally different than transactional queries, the classic focus

of database systems. An OLAP database varies in both the storage and the execution model from an OLAP database. Transactional queries favor row-oriented storage, since they update a small number of tuples. On the other hand, analytical queries favor column-oriented storage, since they read a small number of attributes from a large number of tuples.

In this thesis, we describe techniques for the design and implementation of efficient analytical query execution on modern hardware. Each chapter discusses optizations to make query execution aware of (or utilize) different *layers* of the underlying hardware. The hardware layers are main memory, the multiple levels of caches in modern processors, the NUMA interconnection layer across multiple CPUs in shared memory platforms, the network layer, and, finally, the layer above the caches commonly described as the processor registers, but more accurately described as the throughput within each processor core.

The memory layers of a mainstream platform include three levels of CPU caches, the main memory connected to each CPU socket, and the NUMA interconnection layer combining multiple CPUs in a single shared memory platform. Memory access during query execution is optimized not only via column-oriented storage and compression, but also by using main memory partitioning to divide or cluster the data in smaller pieces with better memory locality. Cache-conscious execution [Manegold *et al.*, 2000b] can be used on multiple database operators such as joins [Manegold *et al.*, 2002; Kim *et al.*, 2009; **?**; Blanas *et al.*, 2011; Albutiu *et al.*, 2012; Balkesen *et al.*, 2013b; Balkesen *et al.*, 2013a; Schuh *et al.*, 2016], group-by aggregation [Müller *et al.*, 2015], and materialization [Manegold *et al.*, 2004], in order to eliminate random memory accesses and cache misses altogether. Adapting to the multiple levels of memory hierarchy by minimizing cache misses and transfers across CPUs is instrumental to efficiency. Chapter 3 describes multiple main memory partitioning variants optimized for the CPU caches, main memory, and the NUMA interconnection. The partitioning variants are used as building blocks in the implementation of multiple large-scale sorting algorithms with different properties.

Lightweight compression reduces the memory footprint and increases the data access rate. Contrary to generic lossless compression, lightweight compression focuses on decompression efficiency over compression ratio. Dictionary encoding is the simplest and most common approach used in databases and allows query execution to use compressed data

directly without decompressing [Willhalm *et al.*, 2009; Raman *et al.*, 2013]. Dictionary encoding supports multiple schemes for storing the compressed bits and each scheme provides different trade-offs [Ross and Cieslewicz, 2009; Li and Patel, 2013]. Optimizing and evaluating multiple storage schemes is the focus of Chapter 4. The work can be used to choose the most suitable compression storage scheme per case to fully optimize memory access.

Shared-memory platforms can scale up to a few terabytes of RAM using multiple CPUs. However, for datasets that exceed that threshold, database management systems need to scale out to multiple servers, using high speed network. If the data is memory resident across multiple servers, distributed query execution is bound by the network bandwidth that is an order of magnitude lower than main memory bandwidth. Designing distributed algorithms that reduce network traffic by burdening the CPUs can reduce the end-to-end execution time. Chapter 2 introduces track join, a distributed join algorithm that minimizes network traffic by generating optimal transfer schedules for each distinct join key.

The memory hierarchy layers determine the throughput of loading data into the CPU and storing data back to memory. However, the processing throughput within the CPU cores is determined by the implementation of the database operators. Modern processors take advantage of three sources of parallelism to maximize performance. First, instruction-level parallelism optimizes the instruction throughput of the pipeline. Second, thread parallelism involves multiple cores per CPU as well as multiple hardware threads per core executing concurrently. Third, data parallelism is supported via SIMD vector registers that execute the same operation for multiple data items per instruction. The primary source of core pipeline efficiency based on the implementation, is SIMD vectorization. Databases already utilize SIMD instructions for linear-access operators such as scans and compression but not for non-linear-access operators such as hash joins. Chapter 5 describes advanced SIMD vectorization techniques that cover most database operators including non-linear-access cases such as hash tables, partitioning, and Bloom filters. The techniques are evaluated on both mainstream CPUs and co-processors based on the first generation of the Intel many-integrated-core (MIC) architecture. Compared to mainstream CPUs, the MIC architecture uses a larger number of smaller cores with wide SIMD registers and advanced SIMD instructions, stressing thread and data parallelism over instruction-level parallelism.

Analytical query engines dictate how to implement each individual operator based on holistic query engine design goals. For instance, code generation [Kim *et al.*, 2010; Neumann, 2011; Gupta *et al.*, 2015] minimizes data materialization by pipelining operators and eliminate interpretation costs by compiling specific code per query. However, cache-conscious execution is precluded and diminish the impact of SIMD vectorization is diminished. In research, hardware-oriented optimizations such as SIMD vectorization and partitioning, are typically applied to databases operators assuming simple data types and predicates. In order to apply such optimizations in a realistic setting, a careful redesign of each operator is necessary, in order to support multiple data types and complex predicates, as well as contain the cost of data materialization. Chapter 6 describes an analytical query engine design focused on SIMD vectorization. The design is tailored to the latest generation of many-integrated-core processors. Notably, many-core processors include an additional memory layer of on-chip high-bandwidth memory that is faster than main memory but slower than the caches and can be used to facilitate cache-conscious execution.

We now briefly outline the chapters of the thesis. In Chapter 1, we present the necessary background, detail our contributions, and discuss related work. In Chapter 2, we introduce distributed joins with minimal network traffic. In Chapter 3, we design partitioning optimized for the CPU caches and the NUMA layer. In Chapter 4 we discuss lightweight compression. In Chapter 5, we introduce advanced SIMD vectorization techniques to optimize the efficiency within the processor core. In Chapter 6, we describe a query engine design tailored to many-core CPUs. In Chapter 7, we conclude and discuss future work.

Each contribution focuses on specific hardware layers but uses techniques from other layers. To optimize query execution in an analytical database, we have to apply them in unison. Partitioning, the focus of Chapter 3, is reused in Chapter 2 to implement a cache-conscious track join, in Chapter 5 to implement cache-conscious hash joins and sorting using SIMD, and in Chapter 6 for joins, aggregation, and materialization. Compression, the focus of Chapter 4, is reused in Chapter 6. SIMD vectorization, the main focus of 5, is used in Chapter 3 for range partitioning, in Chapter 4 to accelerate compression and decompression, and in Chapter 6 as the basis of the query engine design. Finally, Chapter 6 combines the basic principles of Chapters 3, 4, and 5 in the context of a query engine.

## 1.1 Modern Hardware Hierarchy

The main hardware advances over the last decade that fundamentally reshaped database design and implementation are (i) the increase in main-memory capacity and bandwidth, and (ii) the performance of modern multi-core CPUs. The increase in main-memory capacity allows medium to large databases to remain memory resident. Even when the dataset is too large, query execution can operate in a reduced memory-resident working set. In other words, database operators look like high-performance in-memory algorithms, in contrast to traditional databases where disk access dominated execution time. Modern CPUs have increased their performance throughput, driving novel designs and implementations for most database operators, in order to take advantage of the complex features of modern CPUs.

### 1.1.1 Mainstream CPUs

Modern mainstream CPUs share specific characteristics. Up to about a decade ago, processors had been doubling the frequency clock rate every few years. At the same time, the processors had become increasingly complicated. Besides the tens of stages, the pipeline became *super-scalar*, executing multiple instructions per cycle, with instructions executing mostly *out-of-order*. These advances are the basis of *instruction-level parallelism*. When the clock frequency race came to a halt, the solution was to place multiple cores per CPU chip. Each core would retain most advanced instruction-level parallelism capabilities, combined now with *thread parallelism*. The last source of parallelism, *data parallelism*, was supported in mainstream CPUs via SIMD instructions, before the advent of multi-cores.

We outline the characteristics of modern CPUs: (i) massively super-scalar execution that reaches 6–8 instructions per cycle, (ii) aggressive out-of-order execution with a reorder buffer window that fits hundreds of instructions, (iii) at least a dozen cores for server-class platforms, (iv) simultaneous multi-threading (SMT) supporting multiple hardware threads per physical core, (v) multiple levels of caches with each lower level increasing in capacity and latency, and (vi) advanced SIMD instructions and wide 256-bit or 512-bit SIMD registers. The latest mainstream CPU micro-architectures by Intel are, in chronological order, Westmere, Sandy Bridge, Haswell, and Skylake.

### 1.1.2 Cache Hierarchy

The cache hierarchy in modern processors has remained relatively stable among the latest micro-architectures. The L1 cache is private per core and is split between data and instruction cache. The L1 capacity is 32–64 KB and the latency is 1–5 cycles. The L2 cache is also private per core, the capacity of 256–512 KB, and the latency is 10 cycles or less. The L3 cache has a capacity in the range 8–128 MB and the latency is 25–50 cycles. The L3 is shared among all the cores in the chip. Deviations from this scheme exist in the many-integrated-cores (MIC) architecture outside of the spectrum of mainstream CPUs. We discuss MIC platforms and their relevance to mainstream CPUs in Chapters 5 and 6.

CPU caches are important in databases, especially for analytical queries. The disk–RAM pair of traditional databases has been substituted by the RAM–cache pair. In the past, the working set was a subset of the dataset and was loaded from disk to memory. Nowadays, the entire dataset is already in memory and the working set is kept small enough to remain cache-resident. Cache-conscious execution refers to database operator algorithms that take into account the cache capacity to manage the working set accordingly. Interestingly, as we show in Chapter 3, main-memory partitioning, the most important operator of cache-conscious algorithms, is particularly unfavorable to caches if implemented naïvely.

### 1.1.3 Main Memory and NUMA

The capacity of main memory is currently in the scale of terabytes for high-end servers. Small to medium-sized databases can easily fit in memory on server-class platforms. The memory bandwidth has also increased via a quad-channel interconnection between the memory modules and the CPU. The RAM bandwidth today can reach 50 GB/s using a single CPU. Sequential memory accesses take advantage of hardware prefetching, a mechanism that automatically requests the following cache lines before they are explicitly requested by the software. Databases can easily saturate the memory bandwidth for selection scans on uncompressed data. In Chapter 4, we show that lightweight compression not only reduces the data footprint, but also allows for skipping a constant percentage of the input.

Data are transferred from memory into the CPU in *cache lines*. Cache lines are 64–128 bytes and are the minimum unit of transfer across the levels of the cache as well as

between the CPU and main memory. The cache line units further reduces the throughput of random accesses. For instance, accessing a large array of 4-byte integers in random order, is at least 16 times slower than sequential accesses given 64-byte cache lines, without taking into account the absence of hardware prefetching. Random accesses are further burdened by TLB (*transaction-lookahead-buffer*) misses during the translation of virtual memory addresses to physical memory addresses. Virtual addressing is the indirection level for memory translation imposed by the operating system (OS). In Chapter 3, we show that transferring the data between the caches and main memory in cache line units is essential when partitioning inputs larger than the cache.

To scale the memory bandwidth while retaining the shared-memory multi-processing model, we connect multiple CPUs via the NUMA (*non-uniform-memory-access*) interconnection layer. Each CPU is connected to local RAM modules, collectively known as the NUMA region. In order for a CPU to access data from a memory module connected to another CPU, the request travels through the NUMA interconnection layer. These transfers across NUMA boundaries result in additional latency. In Chapter 3, we show that NUMA-aware algorithms can achieve significant speedups over their NUMA-oblivious counterparts.

### 1.1.4 Network Layer

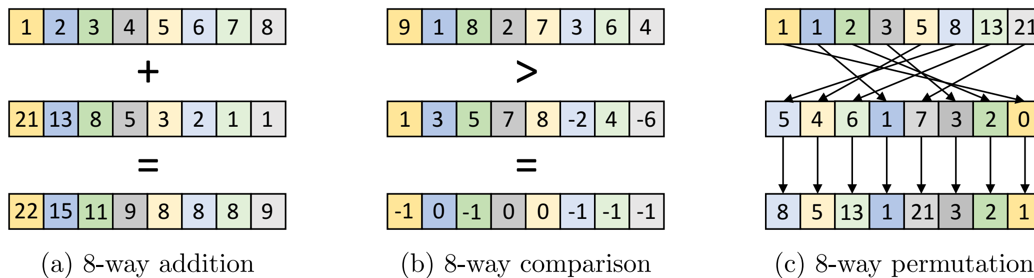
To *scale up* memory capacity and bandwidth, we can connect 2–8 CPUs in a shared memory platform. Beyond that limit, we *scale out* to multiple nodes, potentially each node having multiple CPUs, connected via fast local-area network (LAN). The fastest LAN networks today use the InfiniBand protocol and transfer $\sim$ 10 GB/s node-to-node. Databases take advantage not only of the increased network bandwidth but also of the reduced latencies via RDMA (*remote-direct-memory-access*), to execute distributed transactions efficiently.

In analytical queries, bandwidth is the relevant factor for network efficiency. Nevertheless, the fastest networks are still significantly slower than RAM. For nodes with multiple CPUs, the total memory bandwidth can exceed 200 GB/s, which is at least an order of magnitude higher than the bandwidth of the fastest network. In Chapter 2, we discuss how to trade-off network traffic for CPU cycles via a new distributed join algorithm that minimizes the volume of network transfers by placing extra work on the CPUs.

## 1.2 Additional Background

### 1.2.1 SIMD Vectorization

SIMD (*single-instruction-multiple-data*) refers to special CPU instructions that operate on special registers and execute the same instruction for multiple elements at once. The operations for all 8 elements are computed in a single instruction with a throughput of one cycle. SIMD is synonymous to data parallelism. The term data parallelism is derived from the fact that the data must be spatially adjacent to execute the same operation for multiple elements, using as many instructions as we would use for a single element in scalar code.




(a) 8-way addition (b) 8-way comparison (c) 8-way permutation

Figure 1.1: SIMD instruction examples

Writing code in SIMD requires the use of SIMD instructions directly. We can invoke SIMD instructions by using *intrinsics* provided by the compiler. Intrinsics are part of the official ISA documentation and their descriptions are available online.[1]

```
void add(const int* a, const int* b, int* c, size_t n) {
  size_t i = 0;
  do {
    c[i] = a[i] + b[i];
  } while (++i != n);
}
```

We show an example of a simple loop adding the elements of two arrays of 32-bit integers using scalar code and SIMD code. In modern CPUs, SIMD registers are 128-bit, 256-bit, or 512-bit wide. For this example, we assume 256-bit SIMD registers and use the AVX2 instruction set. Each 256-bit register processes 8 32-bit elements. The functions we invoke are intrinsics mapped to single assembly instructions and are inlined by the compiler.

[1] http://software.intel.com/sites/landingpage/IntrinsicsGuide/

```
void add_simd(const int* a, const int* b, int* c, size_t n) {
  size_t i = 0;
  do {
    // load 8 32-bit integers from array "a" into 256-bit SIMD register "x"
    __m256i x = _mm256_load_si256((__m256i*) &a[i]);
    // load 8 32-bit integers from array "b" into 256-bit SIMD register "y"
    __m256i y = _mm256_load_si256((__m256i*) &b[i]);
    // add 8 32-bit integers from registers "x" and "y" into 256-bit SIMD register "z"
    __m256i z = _mm256_add_epi32(x, y);
    // store 8 32-bit integers from 256-bit SIMD register "z" to array "c"
    _mm256_store_si256((__m256i*) &c[i], z);
  } while ((i = i + 8) != n);  // we assume for simplicity that n is a multiple of 8
}
```

SIMD instructions specify the size of each element they process. Assuming 256-bit SIMD registers, the `AVX2` SIMD ISA specifies instructions to add 32 8-bit integers, 16 16-bit integers, 8 32-bit integers, 4 64-bit integers, 8 32-bit floating point numbers, and 4 64-bit floating point numbers. The degree of data parallelism is dependent on the hardware that determines the size of the SIMD register, and on the software that specifies the size of each element. In Chapter 4, we show how to utilize 8-bit integer as part of 32-bit integer decompression to increase the degree of parallelism through the use of shorter integers.

SIMD vectorization is a fundamental optimization for modern CPUs since it enables one of the two sources of parallelism triggered by the software. Thread parallelism is enabled in software via the use parallel algorithms. Instruction-level parallelism is already provided by the hardware unless we suffer from performance pitfalls such as branch mispredictions and virtual function calls. Data parallelism is enabled in software via SIMD.

Automatic SIMD vectorization refers to the compilation of scalar source code without SIMD intrinsics into assembly with SIMD instructions. The previous example is simple enough to be automatically vectorized by popular compilers. However, automatic vectorization is limited to simple cases and does not cover database operators with more complex control flow and non-linear memory accesses.

In database research, the most common SIMD vectorization approach is to implement individual operators in an ad-hoc way, using techniques applicable only to one specific problem. In Chapter 5, we show that generic SIMD vectorization requires the transformation of control flow into data flow. In Chapter 6, we show that vectorization affects the algorithmic design of individual operators as well as the design of the whole query execution engine.

### 1.2.2 Many Integrated Cores

The trade-off between fewer complex cores versus many simple cores has regained interest since the advent of many-integrated-core (MIC) platforms. The first generation of MIC platforms codenamed Knights Corner (KNC) were co-processors connected via PCI-e like GPUs and used P54C (Pentium 1) cores without super-scalar or out-of-order execution. However, the simpler cores had reduced energy and area footprint and had $\sim 60$ cores per chip. Still, each core supported 512-bit SIMD ahead of mainstream CPUs at the time. The MIC design reduced instruction-level parallelism in favor of thread and data parallelism. In Chapter 5, we use KNC co-processors to evaluate advanced SIMD vectorization techniques.

The latest MIC platform codenamed Knights Landing (KNL) is also available as a stand-alone CPU. KNL (or *many-core*) CPUs have 60–70 cores per chip, with super-scalar pipelines, out-of-order execution, with AVX-512 SIMD, same as the latest mainstream CPUs. In addition to regular DRAM, many-core CPUs have access to 16 GB of high-bandwidth on-chip multi-channel DRAM (MCDRAM) that is faster than regular DRAM and slower than the caches, albeit much larger in capacity. In Chapter 6, we discuss the design of analytical query engines for many-core CPUs.

## 1.3 Contributions

### 1.3.1 Distributed Joins with Minimal Network Traffic

Network communication is the slowest component of many operators in distributed parallel databases. Existing parallel DBMSs rely on algorithms designed for disks with minor modifications for networks. A more complicated algorithm may burden the CPUs, but could avoid redundant transfers of tuples across the network. In Chapter 2 [Polychroniou *et al.*, 2014], we introduce *track join*, a novel distributed join algorithm that minimizes network traffic by generating an optimal transfer schedule per distinct join key. Track join extends the trade-off options between CPU and network. Our evaluation based on real and synthetic data shows that track join adapts to diverse cases and degrees of locality. Considering both network traffic and execution time, even with no locality, track join outperforms hash join on the most expensive queries of real workloads.

### 1.3.2 In-Memory Partitioning and Large-Scale Sorting

High-throughput in-memory query execution relies on partitioning to cluster or divide data into smaller pieces and thus achieve better parallelism and memory locality. In Chapter 3 [Polychroniou and Ross, 2014a], we consider a comprehensive collection of variants of main-memory partitioning tuned for various layers of the memory hierarchy. We revisit the pitfalls of in-cache partitioning, and utilizing the crucial performance factors, we introduce new variants for partitioning out-of-cache. Besides non-in-place variants where linear extra space is used, we introduce large-scale in-place variants, and propose NUMA-aware partitioning that guarantees locality on multiple processors. Also, we make range partitioning comparably fast with hash or radix, by designing a novel cache-resident index to compute ranges. All partitioning variants are combined to build three NUMA-aware sorting algorithms: a stable LSB radixsort; an in-place MSB radixsort using different variants across memory layers; and a comparison-based sort using high-fanout range partitioning.

### 1.3.3 Lightweight Compression Alongside Fast Scans

Database systems employ compression, not only to fit the data in main memory, but also to address the memory bandwidth bottleneck. Lightweight compression schemes focus on efficiency over compression rate and allow query operators to process the data in compressed form. Dictionary compression keeps the distinct column values in a sorted dictionary and stores the values as index codes with the minimum number of bits. Packing the bits of each code contiguously, namely horizontal bit packing, has been optimized by using SIMD instructions for unpacking and by evaluating predicates in parallel per processor word for selection scans. Interleaving the bits of codes, namely vertical bit packing, provides faster scans, but incurs prohibitive costs for packing and unpacking. In Chapter 4 [Polychroniou and Ross, 2015], we improve packing and unpacking for vertical bit packing using SIMD instructions, achieving more than an order of magnitude speedup. Also, we optimize horizontal bit packing on the latest CPUs and compare all approaches. While no single variant is better in all cases, vertical bit packing now offers a good overall trade-off by combining the fastest scans with comparably fast packing and unpacking.

### 1.3.4 Advanced SIMD Vectorization Techniques

Analytical databases are continuously adapting to the underlying hardware in order to saturate all sources of parallelism. Many-core platforms stray from the mainstream CPU design by packing a larger number of simpler cores per chip, relying on SIMD instructions to fill the performance gap. In Chapter 5 [Polychroniou and Ross, 2014b; Polychroniou *et al.*, 2015], we present vectorized designs and implementations of database operators, based on advanced SIMD operations. We study selections, hash tables, Bloom filters, partitioning, sorting and joins. Our evaluation on many-core co-processors and mainstream CPUs shows that our vectorization techniques are an order of magnitude faster than the state-of-the-art scalar and vectorized implementations. We highlight the impact of SIMD vectorization on the algorithmic design of database operators, as well as the architectural design and power efficiency of hardware, by making simple cores comparably fast to complex cores.

### 1.3.5 Query Execution Engines for the Many-Core Era

Hardware-conscious query execution engines, even after specializing to transactions or analytics, are based on multiple distinctive designs. Due to the continuous evolution of hardware, new platforms can favor or disfavor existing designs. For example, the recent many-core platforms revisit the trade-off between many simple cores and fewer complex cores. In Chapter 6, we focus on analytical query execution on many-core CPUs. We extend a state-of-the-art design used on mainstream CPUs by combining code generation and operator pipelining with SIMD vectorization, and show that the SIMD vectorization speedup is diminished when query execution is memory-bound. To better utilize the hardware, we introduce VIP, an analytical query execution engine built bottom-up from pre-compiled column-oriented data-parallel sub-operators, implemented entirely in SIMD, and utilizing the high-bandwidth on-chip memory of many-core CPUs to facilitate cache-conscious execution. We discuss selection scans, compression, hash joins, and group-by aggregation, and extend the operators to support multiple data types and complex predicates. In our evaluation using synthetic and TPC-H queries, we show that VIP outperforms hand-written optimized query-specific code without including the runtime compilation overhead, and highlight the efficiency of VIP at utilizing the hardware features of many-core CPUs.

## 1.4 Additional Work

In this section, we discuss additional work of the author that is not presented in detail in the following chapters. The work is within the topic of analytical query execution.

### 1.4.1 Heavy Hitter Aggregation for SIMD Processors

Heavy hitters are data items that occur at high frequency in a data set. They are among the most important items for an organization to summarize and understand during analytical processing. In data sets with sufficient skew, the number of heavy hitters can be relatively small. We take advantage of this small footprint to compute aggregate functions for the heavy hitters in fast cache memory in a single pass. In [Polychroniou and Ross, 2013], we design cache-resident, shared-nothing structures that hold only the most frequent elements. Our algorithm works in three phases. It first samples and picks heavy hitter candidates. It then builds a hash table and computes the exact aggregates of these elements. Finally, a validation step identifies the true heavy hitters from among the candidates. We identify trade-offs between the hash table configuration and performance. Configurations consist of the probing algorithm and the table capacity that determines how many candidates can be aggregated. The probing algorithm can be perfect hashing, cuckoo hashing and bucketized hashing to explore trade-offs between size and speed. We optimize performance by the use of SIMD instructions, utilized in novel ways beyond single vectorized operations, to minimize cache accesses and the instruction footprint.

### 1.4.2 Energy Analysis of Hardware and Software Range Partitioning

Data partitioning is a critical operation for manipulating large datasets because it subdivides tasks into pieces that are more amenable to efficient processing. It is often the limiting factor in database performance and represents a significant fraction of the overall runtime of large data queries. In [Wu *et al.*, 2014], we measure the performance and energy of state-of-the-art software partitioners, and describe and evaluate a hardware range partitioner that further improves efficiency. The software implementation is broken into two phases, allowing separate analysis of the partition function computation and the cost of data shuffling.

Although range partitioning is commonly thought to be more expensive than simpler strategies such as hash partitioning, our measurements indicate that careful data movement and optimization of the partition function can allow it to approach the throughput and energy consumption of hash or radix partitioning. For further acceleration, we describe a hardware range partitioner, a streaming framework that offers a seamless execution environment for this and other streaming accelerators, and a detailed analysis of a 32nm physical design that matches the throughput of 4–8 software threads while consuming just 6.9% of the area and 4.3% of the power of a Xeon core in the same technology generation.

### 1.4.3 SIMD-accelerated Regular Expression Matching

String processing tasks are common in analytical queries powering business intelligence. Besides substring matching, provided in SQL by the like operator, popular DBMSs also support regular expressions as selective filters. Substring matching can be optimized by using specialized SIMD instructions on mainstream CPUs, reaching the performance of numeric column scans. However, generic regular expressions are harder to evaluate, being dependent on both the DFA size and the irregularity of the input. In [Sitaridi *et al.*, 2016], we optimize matching string columns against regular expressions using SIMD-vectorized code. Our approach avoids accessing the strings in lockstep without branching, to exploit cases when some strings are accepted or rejected early by looking at the first few characters. On common string lengths, our implementation is up to 2X faster than scalar code on a mainstream CPU and up to 5X faster on the Xeon Phi co-processor, improving regular expression support in DBMSs.

## 1.5 Related Work

### 1.5.1 Parallel and Distributed Databases

The foundations of distributed query processing and parallel database implementation [Bernstein and Chiu, 1981; DeWitt *et al.*, 1984; DeWitt and Gray, 1992; Epstein *et al.*, 1978; Mackert and Lohman, 1986] have been laid some decades back. The fundamental distributed hash join algorithm was first presented as part of the Grace database machine

[Kitsuregawa *et al.*, 1983] and was later parallelized by the Gamma project [DeWitt *et al.*, 1990]. Distributed algorithms that filter tuples to reduce network traffic have been extensively studied, namely semi-joins [Bernstein and Chiu, 1981; Roussopoulos and Kang, 1991; Stamos and Young, 1993], Bloom-joins [Mullin, 1990], and other approaches [Li and Ross, 1995]. Contemporary databases shift from the one-size-fits-all paradigm and are optimized for specific workloads, the most notable cases being transactions and relational analytics. Transactional distributed databases are targeting higher transaction throughput, by eliminating unnecessary network communication [Stonebraker *et al.*, 2007] due to latency induced delays of distributed commit protocols. [Binnig *et al.*, 2016] discussed changes to both transactional and analytical workloads for high-end networks, by improving the distributed 2-phase-commit protocol using RDMA as well as Grace hash join and group-by aggregation using local preprocessing and semi-join filtering.

### 1.5.2 Batch Distributed Data Processing

Batch distributed data processing systems such as MapReduce [Dean and Ghemawat, 2004] are popular platforms for executing analytical database workloads [Thusoo *et al.*, 2009]. [Pavlo *et al.*, 2009] evaluated MapReduce against distributed databases and finds that found databases to be faster at execution but slow when data loading times are taken into account. [Afrati and Ullman, 2010] proposed a single-pass algorithm for shuffling data across the network for multiple joins. [Chu *et al.*, 2015] evaluated the algorithm alongside Grace hash join alongside multiple approaches for executing the joins in memory. [Okcan and Riedewald, 2011] proposed MapReduce joins with minimized network transfers by collocating the data with the relevant processing operations to reduce the network traffic.

### 1.5.3 Network-Optimized Databases

[Kim *et al.*, 2012] proposed a distributed sorting algorithm where CPU and network time are overlapped by transferring keys and payloads separately. [Frey *et al.*, 2009] discussed distributed joins on faster networks where the network is not the bottleneck by broadcasting relations using a ring-like topology. [Rödiger *et al.*, 2014] discussed network time optimization for distributed joins by formulating the placing of partitions in network nodes as an

integer optimization problem. However, the approach is NP-complete and therefore cannot be applied per-key. In contrast, the track join algorithm we propose in Chapter 2 minimizes network volume using linear scheduling applied per key, achieving a finer granularity optimum. While completion time is not necessarily a better metric than network volume when partial results can be consumed as soon as they are generated, track join can utilize the same temporal scheduling to reduce end-to-end time. [Zamanian *et al.*, 2015] discussed data placement to maximize locality for database workloads. [Rödiger *et al.*, 2015] measured the impact of high-speed networks on query optimization and in-memory execution. [Rödiger *et al.*, 2016] proposed a distributed join algorithm that deals with skew building on top of hash join and broadcast join. [Barthels *et al.*, 2015] optimized distributed hash joins to run on low-area high-end networks using RDMA and [Barthels *et al.*, 2017] extended the approach in the latest hardware using thousands of cores.

### 1.5.4 Query Execution Engines

High-performance in-memory query execution originated with the advent of large main-memory capacities. Column-oriented query execution [Manegold *et al.*, 2000a] and cache-conscious database operators [Manegold *et al.*, 2002; Manegold *et al.*, 2004] were proposed even before the advent of multi-core CPUs. Analytical database systems adopted column-oriented storage, focusing on columnar compression [Stonebraker *et al.*, 2005] and complex materialization strategies [Abadi *et al.*, 2007] to further optimize memory access. Notable commercial analytical databases use column-oriented storage and compression as their basic design principles [Raman *et al.*, 2013; Willhalm *et al.*, 2009]. Storage managers have also been improved since [Johnson *et al.*, 2009] to keep up with the increasing speed. Column-oriented block-at-a-time execution [Boncz *et al.*, 2005] as well as per-query code generation and compilation [Gupta *et al.*, 2015; Kim *et al.*, 2010; Neumann, 2011] are both state-of-the-art analytical query engine designs. To combine transactions and analytics, [Grund *et al.*, 2010; Alagiannis *et al.*, 2014; Arulraj *et al.*, 2016] proposed hybrid-stores and adaptive stores over column stores. [Menon *et al.*, 2017] combined block-at-a-time execution with prefetching to reduce the cost of cache misses in code generation. We will discuss analytical query execution engines further in Section 6.2.2. While most engine designs dictate a

particular way to build each operator, optimized implementations of isolated individual operators are also popular. These implementations typically use key–rid pairs and omit payload materialization. In Chapter 6 we generalize from key–rid pairs to realistic database operators that support multiple columns, data types, and complex predicates.

### 1.5.5 Main-Memory Partitioning

We first present related work on in-memory partitioning which is the focus of Chapter 3. [Manegold *et al.*, 2000b] identified that TLB thrashing occurs when naïvely partitioning to a large number of outputs that exceed the TLB capacity. [Satish *et al.*, 2010] introduced efficient out-of-cache partitioning using cache-resident buffers and [Wassenberg and Sanders, 2011] identified the significance of write-combining to avoid cache pollution. Both works applied efficient partitioning to radixsort. [Wu *et al.*, 2013] proposed hardware-accelerated range partitioning for performance and power efficiency. [Li *et al.*, 2013] evaluated data shuffling on the NUMA layer for platforms with multiple CPUs. [Cho *et al.*, 2015] discussed in-place radix-sort. [Psaroudakis *et al.*, 2016] studied data placement on NUMA platforms.

### 1.5.6 In-Memory Joins and Group-by Aggregation

[Manegold *et al.*, 2002] proposed the first cache-conscious design of hash join and [Kim *et al.*, 2009] applied the design to multi-core CPUs. [Blanas *et al.*, 2011] argued that hardware-oblivious joins are fast enough. [Balkesen *et al.*, 2013b] argued in favor of partitioning-based hash join implementations. [Albutiu *et al.*, 2012] proposed a NUMA-aware sort-merge-join algorithm for mulitple CPUs and later [Balkesen *et al.*, 2013a] evaluated sort-merge join against hash joins on NUMA platforms. [Lee *et al.*, 2014] descibed the execution of joins on compressed data. [Schuh *et al.*, 2016] re-evaluated multiple hash join implementations, pointing out that payload materialization eventually becomes more expensive than the hash join itself and thus optimizing a stand-alone hash join is of limited use. We will use these observations in Chapter 6 to discuss hash joins in the context of the whole query engine. [Cieslewicz and Ross, 2007; Cieslewicz *et al.*, 2010; Ye *et al.*, 2011] discussed how to scale group-by aggregation on multi-core processors by avoiding contention and [Müller *et al.*, 2015] discussed cache-conscious aggregation via partitioning.

### 1.5.7 Lightweight In-Memory Compression

Modern analytical databases are tuned for main-memory accesses [**?**; Manegold *et al.*, 2000a]. [Stonebraker *et al.*, 2005] highlighted the importance of columnar compression for in-memory databases. [Zukowski *et al.*, 2006] discussed RAM to CPU-cache decompression. [Abadi *et al.*, 2006] incorporated compression into query execution. [Holloway *et al.*, 2007] studied schema-tuned query code generation. [Johnson *et al.*, 2008] studied selection scans on bit packed data. [Raman and Swart, 2006] discussed entropy compression including complex dictionary encoding schemes, later applied in commercial database systems [Raman *et al.*, 2013]. [Afrati and Ullman, 2010] proposed pattern-based compression in processor words. [Li and Patel, 2013] proposed vertical bit packing and denormalization with compression to convert complex queries into scans [Leis *et al.*, 2014]. [Feng *et al.*, 2015] proposed vertical byte packing. [Feng and Lo, 2015] showed how to compute aggregation functions on bit packed data. [Satish *et al.*, 2010] studied skew-friendly compression. [Stepanov *et al.*, 2011] studied vectorized variable-byte encoding. [Lemire and Boytsov, 2015] discussed vectorized frame-of-reference compression and delta encoding. [Lemire *et al.*, 2016; Wang *et al.*, 2017] evaluated such compression techniques on inverted lists and bitmaps.

### 1.5.8 SIMD Vectorization

Optimizing database operators to use SIMD instructions is essential to take advantage of data parallelism provided by modern processors. [Zhou and Ross, 2002] applied SIMD to linear scans, index scans, and nested loop joins. [Ross, 2007] proposed hash tables with multiple keys per bucket probed via SIMD instructions, and applied the idea to cuckoo hashing. [Willhalm *et al.*, 2009] optimized decompression of bit packed data. In Chapter 4 we discuss the optimization of other bit packing layouts using SIMD instructions. [Inoue *et al.*, 2007] proposed data-parallel combsort and merging. [Chhugani *et al.*, 2008] optimized bitonic merging for mergesort. [Kim *et al.*, 2010] proposed multi-way trees tailored to the SIMD layout by comparing multiple keys per node. [Roy *et al.*, 2012] proposed SIMD-based pre-filtering for frequent itemset mining. [Mühlbauer *et al.*, 2013] optimized data loading from CSV files using SIMD instructions for efficient parsing. [Idreos *et al.*, 2007] proposed database cracking, a dynamic partitioning for selection scans. [Pirk *et al.*, 2014b] proposed

a SIMD implementation for cracking. [Inoue *et al.*, 2014] redesigned sorted set intersection using SIMD to reduce branch mispredictions. [Inoue and Taura, 2015] studied the sorting of structures instead of key–rid pairs. Hash joins were evaluated on first generation MIC co-processors by [Jha *et al.*, 2015] and on second generation MIC processors by [Cheng *et al.*, 2017]. [Sidirourgos and Kersten, 2013] presented in-memory secondary indexes based on bitmaps and zonemaps, implemented in SIMD by [Sidirourgos and Mühleisen, 2017].

Compilers can translate simple scalar code to use SIMD instructions based on loop unrolling [Larsen and Amarasinghe, 2000] and control flow predication [Shin, 2007]. However, the vectorized operators we present in Chapter 5 are too complex to be automatically vectorized by any compiler as of yet. Also, the vectorized code executes the same database operation but is not always equivalent to the scalar code.

### 1.5.9 Databases on GPUs and FPGAs

GPUs are somewhat similar to SIMD vectorization since the SIMT model is loosely similar to the SIMD model. [Bakkum and Skadron, 2010] described the implementation of database operators on GPUs. [Kim *et al.*, 2010] applied the multi-way index optimization to both CPUs and GPUs. [Satish *et al.*, 2010] also evaluated optimized SIMD radixsort on both CPUs and GPUs. [Pirk *et al.*, 2011] discussed the asymmetries in the memory layout of GPUs compared to CPUs. [Sitaridi and Ross, 2012] discussed memory contention for group-by aggregation on GPUs. [Kim *et al.*, 2012] discussed hash joins in GPUs, focusing in efficient data transfers via the PCI-e channel. [Sitaridi and Ross, 2013] studied selective predicates and proposed an optimization model for disjunctive selections on GPUs. [Pirk *et al.*, 2014a] presented techniques towards co-processing database workloads in multiple platforms including GPUs. [Pirk *et al.*, 2015] measured several hardware traits of GPUs and MIC co-processors. [Pirk *et al.*, 2016] proposed a vectorization algebra for mapping SIMD operations to high-level constructs, encompassing both CPUs and GPUs. [Sitaridi and Ross, 2016] discussed regular expression matching on GPUs and [Sidler *et al.*, 2017] proposed using FPGAs for the same problem. [Stehle and Jacobsen, 2017] revisited data partitioning on GPUs by focusing on TLB misses. [Kara *et al.*, 2017] proposed off-loading in-memory partitioning from the CPUs to FPGAs.

# Chapter 2

# Distributed Joins with Minimal Network Traffic

## 2.1 Introduction

The processing power and storage capacity of a single server can be large enough to fit small to medium scale databases. Servers with memory capacity of more than a terabyte are now common. Packing a few multi-core CPUs on top of shared non-uniform access (NUMA) RAM provides substantial parallelism, where we can run in-memory database operations (i.e. sort, join, and group-by) at rates of a few gigabytes per second [Albutiu *et al.*, 2012; Balkesen *et al.*, 2013a; Satish *et al.*, 2010; Wassenberg and Sanders, 2011; Ye *et al.*, 2011].

Database research has also evolved to catch up to the hardware advances. Fundamental design rules of the past on how a DBMS should operate are now being revised due to their inability to scale and achieve good performance on modern hardware. Special purpose databases are now popular against the one-size-fits-all approach [Stonebraker *et al.*, 2007], while accelerators [Raman *et al.*, 2013] are the implicit manifestation of the same concept.

The advances in database design for storage and execution on modern hardware have not been met by similar advances in distributed parallel database design. When the most fundamental work on distributed and parallel databases was published [Bernstein *et al.*, 1981; Epstein *et al.*, 1978; Mackert and Lohman, 1986], hardware advances of today like multi-core parallelism had not yet occurred. Techniques to speed up short-lived distributed

transactions [Stonebraker *et al.*, 2007] target distributed commit protocols, which suffer from network latencies rather than throughput. Queries where communication is inevitable are less popular research topics or are left for data-centric generic distributed systems for batch-processing [Dean and Ghemawat, 2004; Pavlo *et al.*, 2009].

The latest network technologies may be slow relative to main-memory-resident processing. A 40 Gbps InfiniBand measured less than 3 GB/s real data rate per node during hash partitioning. If done in RAM, partitioning to a few thousand outputs runs close to the memory copy bandwidth [Satish *et al.*, 2010; Wassenberg and Sanders, 2011]. For instance, a server using 4X 8-core CPUs and 1333 MHz quad-channel DDR3 DRAM achieves a partition rate of 30–35 GB/s, more than an order of magnitude higher than the InfiniBand network. [Balkesen *et al.*, 2013a] achieves a hash join rate of 4.85 GB/s of 32-bit key, 32-bit payload tuples on 4X 8-core CPUs. Such high-end specifications are common in commercial hardware platforms for large-scale analytics.

Network optimization is important for both low-end and high-end hardware. In low-end platforms where the network is relatively slow compared to local in-memory processing, we expect the execution time to be dominated by network transfers. Thus, any network traffic reduction directly translates to faster execution. In high-end platforms, given that the network still cannot be as fast as the RAM bandwidth, completion times are also reduced if the reduction in network traffic is comparable with the increase in CPU cycles.

In order to show how much time databases can spend on the network, we give an example of a real analytical workload from a large commercial vendor, using a market-leading commercial DBMS. Using 8 machines connected through 40 Gbps InfiniBand (see Section 2.4 for more details of the configuration), we found that the five most expensive queries spend $\sim$ 65–70% of their time transferring tuples on the network and account for 14.7% of the total time required to execute the entire analytical workload with more than 1500 queries. All five queries have a non-trivial query plan (4–6 joins), but spend 23%, 31%, 30%, 42%, and 43% of their total execution time on a single distributed hash join.

A sophisticated DBMS should have available options to optimize the trade-off between network and CPU utilization. One solution would be to apply network optimization at a higher level treating the network as a less desired route for data transfers, without modifying

the underlying algorithms. These approaches are common in generic distributed processing systems [Dean and Ghemawat, 2004]. A second solution would be to employ data compression before sending data over the network. This solution is orthogonal to any algorithm but can consume a lot of CPU resources without always yielding substantial compression. A third solution is to create novel algorithms for database operator evaluation that minimize network communication by incurring local processing cost. This approach is orthogonal and compatible with compression and other higher level network transfer optimizations.

Grace hash join [DeWitt *et al.*, 1990; Kitsuregawa *et al.*, 1983] is the predominant method for executing distributed joins and uses hash partitioning to split the initial problem into shared-nothing sub-problems that can proceed locally per node. Partitioning both tables works almost independently of the table sizes. However, hash join is far from network-optimal because it transfers almost the full size of both tables over the network. Using pre-determined hash functions guarantees load balancing, but limits the probability that a hashed tuple will not be transferred over the network to $1/N$ on $N$ nodes.

We introduce track join, a novel algorithm for distributed joins that minimizes transfers of tuples across the network. The main idea of track join is to decide where to send rows on a key by key basis. The decision uses information about where records of the given key are located. Track join has the following properties: it (i) is orthogonal to data-centric compression, (ii) can co-exist with semi-join optimizations, (iii) does not rely on favorable schema properties, such as foreign key joins, (iv) is compatible with both row-store and column-store organization, and (v) does not assume favorable pre-existing tuple placement. We implement track join to evaluate the most expensive join operator in the most expensive queries of real workloads. We found that track join reduces the network traffic significantly over known methods, even if pre-existing data locality is removed and all data are used in optimally compressed form throughout the join.

The rest of this chapter is organized as follows. In Section 2.2, we describe the track join algorithm by using three variants with increasing complexity. In Section 2.3, we discuss query optimization costs, tracking-aware hash joins, and semi-join filtering. In Section 2.4, we present our experimental evaluation using both synthetic datasets and real workloads. In Section 2.5, we discuss our conclusions.

## 2.2 Track Join

Formally, the problem under study is the general distributed equi-join. The input consists of tables $R$ and $S$ split arbitrarily across $N$ nodes. Every node can send data to all other nodes and all links are considered to have the same network bandwidth. We present three versions of track join gradually evolving in complexity.

The first version of track join (2-phase track join) discovers network locations that have at least one matching tuple for each unique key. We then pick one side of the join and broadcast each tuple to all locations that have matching tuples of the other table for the same join key. We use the term *selective broadcast* for this process.

The second version of track join (3-phase track join) gathers not only a boolean indicator whether a node has tuples of a specific join key or not, but also the total size of these tuples. Using this information at runtime, we pick the cheapest side to be selectively broadcast, a decision taken independently for each distinct join key.

The third and full version of track join (4-phase track join) generates an optimal join schedule for each distinct key achieving minimum payload transfers, without hashing or broadcasting. The schedule migrates tuples from one table to potentially fewer nodes, before selectively broadcasting tuples from the other table.

The algorithm is described here using a pipelined approach. We assume three processes ($R$, $S$ operating on tables $R$ and $S$, and $T$ that generates schedules) running simultaneously on all nodes. A non-pipelined approach is also possible (see Section 2.4).

We now explain the algorithmic convention used throughout this section. Types are shown next to variables when their values are being assigned. Unless a new value assignment is explicitly written, the variable already has a value and is used as input.

### 2.2.1 2-Phase Track Join

The 2-phase track join is the simplest version of track join. The first phase is the tracking phase, while the second phase is the selective broadcast phase. In the first phase, both $R$ and $S$ are projected to their join key and sent over the network. The destination is determined by hashing the key, in the same way that hash join sends tuples. Duplicate keys

**Algorithm 1:** 2-Phase Track Join: $\text{process}_R - R$ to $S$

1. $T_R \leftarrow \{\}$
2. **for all** $\langle \text{key}_{R|S}\ k,\ \text{payload}_R\ p_R \rangle$ **in** table $R$ **do**
3.     **if** $k$ **not in** $T_R$ **then**
4.         $n \leftarrow \text{hash}(k) \bmod N$
5.         **send** $k$ **to** $\text{process}_T\ n$
6.     $T_R \leftarrow T_R \cup \langle k, p_R \rangle$
7. **barrier**
8. **while any** $\text{process}_T\ n_T$ **sends do**
9.     **for all** $\langle \text{key}_{R|S}\ k,\ \text{process}_S\ n_S \rangle$ **from** $n_T$ **do**
10.         **for all** $\langle k,\ \text{payload}_R\ p_R \rangle$ **in** $T_R$ **do**
11.             **send** $\langle k, p_R \rangle$ **to** $n_S$

on each node are redundant and are eliminated locally. Every node receives unique keys and stores them alongside the id of the source node. Then, transfer schedules are generated for each join key based on the locations of matching tuples. Hash partitioning the join keys ensures load balancing of the schedule generation process across all nodes.

In the second phase, we transfer payloads from one input only. When $R$ tuples are transferred, $\text{process}_T$ sends messages to each location with matching $R$ tuples. Each message includs the join key and the set of locations with matching $S$ tuples. Algorithm 1 describes $\text{process}_R$ of 2-phase track join that selectively broadcasts tuples from table $R$.

Algorithm 2 describes the transferring of $R$ tuples during 2-phase track join. Note that the algorithm is not identical to the $R$-processing part. When $R$ tuples are transferred, the

**Algorithm 2:** 2-Phase Track Join: $\text{process}_S - R$ to $S$

1. $T_S \leftarrow \{\}$
2. **for all** $\langle \text{key}_{R|S}\ k,\ \text{payload}_S\ p_S \rangle$ **in** table $S$ **do**
3.     **if** $k$ **not in** $T_S$ **then**
4.         $n \leftarrow \text{hash}(k) \bmod N$
5.         **send** $k$ **to** $\text{process}_T\ n$
6.     $T_S \leftarrow T_S \cup \langle k, p_S \rangle$
7. **barrier**
8. **while any** $\text{process}_R\ n_R$ **sends do**
9.     **for all** $\langle \text{key}_{R|S}\ k,\ \text{payload}_R\ p_R \rangle$ **from** $n_R$ **do**
10.         **for all** $\langle k,\ \text{payload}_S\ p_S \rangle$ **in** $T_S$ **do**
11.             **commit** $\langle k, p_R, p_S \rangle$

**Algorithm 3:** 2-Phase Track Join: process$_T$ − $R$ to $S$

```
1 T_{R|S} ← {}
2 while any process_R or any process_S n_{R|S} sends do
3     for all ⟨key_{R|S} k⟩ from n_{R|S} do
4         T_{R|S} ← T_{R|S} ∪ ⟨k, n_{R|S}⟩
5 barrier
6 for all distinct key_{R|S} k in T_{R|S} do
7     for all ⟨k, process_R n_R⟩ in T_{R|S} do
8         for all ⟨k, process_S n_S⟩ in T_{R|S} do
9             send ⟨k, n_S⟩ in n_R
```

$R$-processing algorithm sends the payloads, while the $S$-processing algorithms receives the $R$ tuples and joins them locally. The inverse would happen if $S$ tuples were transferred. Both algorithms use a distributed barrier to synchronize all processes between the two phases.

Algorithm 3 describes the process of receiving the join keys, generating the transfer schedules, and distributing the broadcast locations. The decision whether to selectively broadcast $R$ tuples to $S$ locations, or $S$ tuples to $R$ locations, is decided by the query optimizer before the 2-phase track join starts executing, similarly to hash joins where we decide beforehand which relation to build in a hash table and which relation to probe.

The tracking phase of 2-phase track join resembles a hash join on the join keys, after eliminating duplicates. The selective broadcast phase resembles a broadcast join, since only the tuples of one table are transferred over the network. However, compared to a broadcast join that sends the $R$ tuples to all nodes, track join selectively broadcasts the $R$ tuples to nodes that are known to contain matching $S$ tuples. For instance, consider a join with two equally sized tables with unique join keys and low join selectivity. The total cost of track join would be the cost of the tracking phase plus the size of the smallest table $\min(|R|, |S|)$, taking into account the average payload size times the number of tuples. In the same scenario, a broadcast join would cost $(N-1) \cdot \min(|R|, |S|)$, and a hash join would cost $(N-1)/N \cdot (|R| + |S|)$. Having entirely unique keys maximizes the cost of the tracking phase to $w_{key}/(w_{key} + w_{payload})$ times that of hash join, where $w_{key}$ and $w_{payload}$ are the key and payload widths. As long as $w_{key} < w_{payload}$ and the key is smaller than half of the tuple, track join will transfer less data than hash join.

### 2.2.2 3-Phase Track Join

3-phase track join improves over 2-phase track join by deciding whether to broadcast $R$ tuples to the locations of $S$ tuples, or vice versa. The decision is taken while executing the join, independently for each distinct join key. In order to decide which selective broadcast direction is cheaper, we need to know how many tuples will be transferred in each case. We gather this information during the tracking phase. Instead of only tracking nodes with at least one matching tuple, as done in 2-phase track join, here we also track the number of matches for each distinct join key per node. Thus, we know not only which nodes have matching tuples per join, but also how many tuples. To generalize for tuples of variable length, instead of using a single count, we sum the widths of the matching tuples per node.

Algorithm 4 describes the processing of $R$ tuples during 3-phase track join. The algorithms for processing $R$ and $S$ are identical in 3-phase track join. The additional step is

**Algorithm 4:** 3-Phase Track Join: process$_R$

```
1  T_R ← {}
2  for all ⟨ key_{R|S} k, payload_R p_R⟩ in table R do
3      T_R ← T_R ∪ ⟨k, p_R⟩
4  barrier
5  for all distinct key_{R|S} k in T_R do
6      c ← |k in T_R|
7      n ← hash(k) mod N
8      send ⟨k, c⟩ to process_T n
9  barrier
10 while any process_S or any process_T sends do
11     if source is process_T n_T then
12         for all ⟨ key_{R|S} k, process_S n_S⟩ from n_T do
13             for all ⟨k, payload_R p_R⟩ in T_R do
14                 send ⟨k, p_R⟩ to n_S
15     else if source is process_S n_S then
16         for all ⟨key_{R|S} k, payload_S p_S⟩ from n_S do
17             for all ⟨k, payload_R p_R⟩ in T_R do
18                 commit ⟨k, p_R, p_S⟩
```

**Algorithm 5:** 3-Phase Track Join: process$_S$

```
1  [...]                        // symmetric with process_R of 3-phase track join
```

the count aggregation of the join keys. The unique join keys and the aggregated counts are hash partitioned over the network. Duplicate key elimination is part of the aggregation. The first of the two barriers can be omitted here since the first phase sends nothing over the network. Nevertheless, we include the barrier to clarify that the tracking phase will practically start only after the local count aggregation on each node is completed.

Algorithm 6 describes the process of receiving the join keys, generating the bi-directional transfer schedules, and driving of bi-directional selective broadcasts. The hash partitioning of keys also distributes the matching tuple counts used to pre-compute the selective broadcast cost. We compute the cost for both directions and pick the cheapest. Then the suitable key and node-id messages are relayed to source nodes to drive the selective broadcast.

The improvement of 3-phase over 2-phase track join is that bi-directional selective broadcast can distinguish cases in which moving $S$ tuples would transfer fewer bytes than moving

**Algorithm 6:** 3-Phase Track Join: process$_T$

```
1  barrier
2  T_{R|S} ← {}
3  while any process_R or any process_S n_{R|S} sends do
4      for all ⟨key_{R|S} k, count c⟩ from n_{R|S} do
5          T_{R|S} ← T_{R|S} ∪ ⟨k, n_{R|S}, c⟩
6  barrier
7  for all distinct key_{R|S} k in T_{R|S} do
8      R, S ← {}, {}
9      for all ⟨k, process_R n_R, count c⟩ in T_{R|S} do
10         R ← R ∪ ⟨n_R, c · width_R⟩
11     for all ⟨k, process_S n_S, count c⟩ in T_{R|S} do
12         S ← S ∪ ⟨n_S, c · width_S⟩
13     c_{RS} ← broadcast R to S
14     c_{SR} ← broadcast S to R
15     if c_{RS} < c_{SR} then
16         for all ⟨k, process_R n_R⟩ in T_{R|S} do
17             for all ⟨k, process_S n_S⟩ in T_{R|S} do
18                 send ⟨k, n_S⟩ to n_R
19     else
20         for all ⟨k, process_S n_S⟩ in T_{R|S} do
21             for all ⟨k, process_R n_R⟩ in T_{R|S} do
22                 send ⟨k, n_R⟩ to n_S
```

**Algorithm 7:** Track Join: Broadcast $R$ to $S$ (used in process$_T$)

```
1  r_total ← 0                      // the total number of matching R tuples (or their total size)
2  r_local ← 0                      // the total number of matching R tuples in nodes with matching S tuples (local sends)
3  n_R ← 0                          // the number of nodes with matching R tuples (excluding self)
4  n_S ← 0                          // the number of nodes with matching S tuples (including self)
5  for all process_{R|S} i do
6      r_total ← r_total + R_i      // add the total number of matching R tuples in node i
7      if R_i ≠ 0 and i ≠ n_self then
8          n_R ← n_R + 1            // count nodes with both R and S tuples
9      if S_i ≠ 0 then
10         r_local ← r_local + R_i n_S ← n_S + 1
11 return r_total · n_S − r_local + n_R · n_S · M    // M: the size of the schedule message
```

$R$ tuples. Because selective broadcasts occur from both directions, the algorithm for processing $R$ and $S$ includes both cases. The decision whether to selectively broadcast $R \to S$ or $S \to R$ is taken independently per distinct join key and we do not pick a single direction for the entire join. 3-phase track join also simplifies query optimization since we do not need to predetermine which input to selectively broadcast.

The cost of a selective broadcast ($R \to S$) is shown in Algorithm 7. The cost is the same as a Cartesian product excluding all nodes that have no tuples of the particular join key. We compute the total size of $R$ ($r_{all}$), the size of $R$ tuples in nodes with matching $S$ tuples ($r_{local}$), the number of nodes with matching $R$ tuples ($n_R$), and the number of nodes with matching $S$ tuples ($n_S$). The total cost is then computed by subtracting the local $R$ tuples ($r_{local}$) from all $R$ tuples ($r_{all}$). We also include the cost of sending location messages that drive the selective broadcast. The size of these messages denoted by ($M$, is the length of the join key, the length of the node-id, and the length of the tuple count. In Section 2.2.4, we discuss simple ways to reduce the size of such messages to reduce the network traffic.

Computing the cost of selective broadcast has $O(n)$ complexity, where $n$ is the number of nodes with at least one matching tuple for the particular join key in either input. Nodes with keys from neither input are ignored. Since the number of steps per join key is bounded by the number of matching tuples, the total number of steps for the entire join is also bounded by the total number of tuples in both join inputs. Thus, the complexity of the join algorithm is in the worst case linear to the number of input tuples, similarly to hash join.

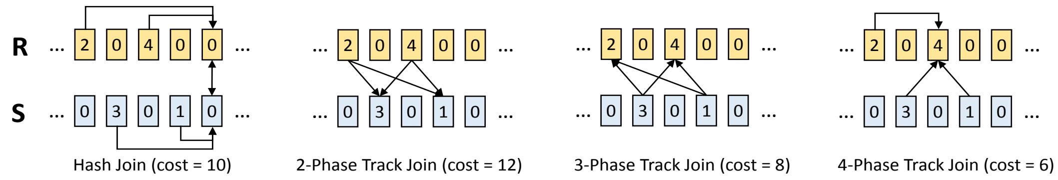



Figure 2.1: Example of network transfers in hash join and track join per distinct join key

### 2.2.3 4-Phase Track Join

In the full version of track join, the join is logically decomposed into single key (Cartesian product) joins for each distinct join key. By minimizing the network cost of each Cartesian product, we reduce the network cost of the entire join. If we omit the cost of tracking which cannot be optimized in a generic way, 4-phase track join transfers the minimum amount of payload data possible for an early-materialized distributed join.

4-phase track join extends 3-phase track join by using a more complicated scheduling algorithm that includes a migration phase. The additional phase ensures that when tuples from one table are selectively broadcast to matching tuple locations from the other table, we have already migrated tuples to a subset of nodes, minimizing the total network transfers for the particular join key. Note that track join does the scheduling of tuples transfers independently for each distinct join key, the finest granularity level.

In Figure 2.1 we show an example of how the different join algorithms behave for a single key. In hash join, we assume the hash destination is the fifth node. In 2-phase track join we selectively broadcast $R$ tuples to matching $S$ tuple locations and in 3-phase track join we choose the opposite direction, since it is cheaper. The 4-phase track join first migrates $R$ tuples, before selectively broadcasting $S$ tuples.

The migration phase allows tuples from one table to be moved to fewer nodes before the selective broadcast from the other table. The schedules do not only decide which side to selectively broadcast, but also which nodes will migrate migrate to other nodes, so that the selective broadcast can skip nodes. Even if we end up migrating all tuples to a single node, track join still performs better than hash join, because the destination is determined by hashing but is the node with the most pre-existing matching tuples. Algorithm 8 describes the migration of $R$ tuples. Processing $S$ is identical.

**Algorithm 8:** 4-Phase Track Join: Process$_R$

```
1  [...]                                                  // identical to the 1st phase of 3-phase track join
2  barrier
3  [...]                                                  // identical to the 2nd phase of 3-phase track join
4  barrier
5  while any process_T or any process_R sends do
6      if source is process_T n_T then
7          for all ⟨key_{R|S} k, process_R n_R⟩ from n_T do
8              for all ⟨k, payload_R p_R⟩ in T_R do
9                  send ⟨k, p_R⟩ to n_R
10                 T_R ← T_R \ ⟨k, p_R⟩
11     else
12         for all ⟨key_{R|S} k, payload_R p_R⟩ from n_R do
13             T_R ← T_R ∪ ⟨k, p_R⟩
14 barrier
15 [...]                                                  // identical to the 3rd phase of 3-phase track join
```

**Algorithm 9:** 4-Phase Track Join: Process$_S$

```
1  [...]                                                  // symmetric with process_R of 4-phase track join
```

We now show the driving of the migration phase in Algorithm 10. During the migration phase, we send key and node-id messages to source nodes to drive the migration matching tuples. Note that we never send the same message to multiple destinations. Every node that receives a $\langle\, k,\, n_R \,\rangle$ message, sends all $R$ tuples with key equal to $k$ to the node $n_R$ (and only that node). The same is true for $S$ tuples.

We are still missing two pieces of the puzzle to complete 4-phase track join. First, we need to explain how to generate transfer schedules that include migration. Second, we have to prove that these transfers schedules are optimal for single key joins.

We formalize the problem of network traffic minimization for a single key (Cartesian product) join, assuming for simplicity (but without loss of generality) that the plan is known to every node and there is no need to send any location messages. Assume that $x_{ij}$ is the binary decision of sending $R$ tuples from node $i$ to node $j$, while $y_{ij}$ is the binary decision of sending $S$ tuples from node $j$ to node $i$. $R_i$ is the total size of $R$ tuples in node $i$ and $|S_j|$ is the total size of $S$ tuples in node $j$. Since each schedule is on a single key, we have to join every $R_i$ data part with every $S_j$ data part. One way is by sending $R$ tuples from $i$ to

**Algorithm 10:** 4-Phase Track Join: Process$_T$

```
1  barrier
2  [...]                                  // identical to the 2nd phase of 3-phase track join
3  barrier
4  for all distinct key_{R|S} k in T_{R|S} do
5      R, S ← {}, {}
6      for all ⟨k, process_R n_R, count c⟩ in T_{R|S} do
7          R ← R ∪ ⟨n_R, c · width_R⟩
8      for all ⟨k, process_S n_S, count c⟩ in T_{R|S} do
9          S ← S ∪ ⟨n_S, c · width_S⟩
10     c_RS, S_migrate ← migrate S and broadcast R
11     c_SR, R_migrate ← migrate R and broadcast S
12     if c_RS < c_SR then
13         for all ⟨k, process_S n_S⟩ in S_migrate do
14             send ⟨k, d_S⟩ to n_S
15             T_{R|S} ← T_{R|S} \ ⟨k, n_S, count c_src⟩
16             T_{R|S} ← T_{R|S} \ ⟨k, d_S, count c_dst⟩
17             T_{R|S} ← T_{R|S} ∪ ⟨k, d_S, c_src + c_dst⟩
18     else
19         for all ⟨k, process_R n_R⟩ in R_migrate do
20             T_{R|S} ← T_{R|S} \ ⟨k, n_R, count c_src⟩
21             T_{R|S} ← T_{R|S} \ ⟨k, d_R, count c_dst⟩
22             T_{R|S} ← T_{R|S} ∪ ⟨k, d_R, c_src + c_dst⟩
23 barrier
24 [...]                                  // identical to the 3rd phase of 3-phase track join
```

$j$, setting $x_{ij} = 1$. Another option is to send $S$ tuples from $j$ to $i$, setting $y_{ij} = 1$. The last option is to send both $R_i$ and $S_j$ to a common third node and join them there. In short, we need some node $k$, where $x_{ik} = 1$ and $y_{kj} = 1$. Local sends do not affect the network traffic, thus all $x_{ii}$ and $y_{jj}$ are excluded from the cost.

$$\text{minimize: } \sum_i \sum_{j \neq i} x_{ij} \cdot R_i + y_{ij} \cdot S_j \qquad \text{subject to: } \forall i, j \sum_k x_{ik} \cdot y_{kj} \geq 1$$

We solve the traffic minimization problem by allowing tuples to be migrated before the selective broadcasts. Before the $R \to S$ selective broadcast, we migrate $S$ tuples, by testing how the cost of the selective broadcast is affected. Since the mechanism allows only $S$ tuples to be transferred, we can easily decide whether the $S$ tuples of some node should stay in place, or be transferred somewhere with more $S$ tuples. Also, the destination can be any

**Algorithm 11:** Track Join: Migrate $S$ and Broadcast $R$ (used in process$_T$)

```
1  r_total, r_local, n_R, n_S ← 0                    // same as in Algorithm 7
2  c_max ← 0            // the most matching tuples on a node with some S tuples
3  for all process_{R|S} i do
4      r_total ← r_total + R_i
5      if R_i ≠ 0 and i ≠ n_self then
6          n_R ← n_R + 1
7      if S_i ≠ 0 then
8          r_local ← r_local + R_i
9          n_S ← n_S + 1
10         if R_i + S_i > c_max then
11             c_max ← R_i + S_i
12             i_max ← i     // the node with the most matching tuples and some S tuples
13 c ← r_total · n_S − r_local + n_R · n_S · M
14 S_migrate ← {}                     // the set of nodes that will migrate S tuples
15 for all process_{R|S} i do
16     if S_i ≠ 0 and i ≠ i_max then
17         Δ ← R_i + S_i − r_total − n_R · M
18         if i ≠ i_self then
19             Δ ← Δ + M
20         if Δ < 0 then
21             c ← c + Δ
22             S_migrate ← S_migrate ∪ {i}
23 return c, S_migrate
```

node that has $S$ tuples (that will not be migrated), thus deciding the destination node is not an issue. The cost of the selective broadcast is considered alongside the cost of migration. If migrating tuples from a node $n$ reduces the selective broadcast cost but increases the total cost, then the migration from node $n$ is rejected.

Algorithm 11 shown above, computes the optimal schedule of $S$ tuple migration followed by the $R \rightarrow S$ selective broadcast. The key and node-id messages have size $m$. We first compute the cost of the selective broadcast $R \rightarrow S$. Then, we iterate over every $S$ node with matching tuples and check whether migrating the $S$ tuples would reduce the overall cost, since we will have to selective broadcast to one less node later. The decision can be taken greedily without considering any other nodes.

Since we consider each node with matching tuples exactly once, schedule generation is in the worst case linear to the number of matching tuples, not to the total number of nodes.

**Theorem 1.** *We can generate optimal selective broadcast $R \to S$ schedules for any single join key by migrating $S$ tuples. The optimal schedule can be determined in worst-case $O(n)$ time, where $n$ is the total number of $R$ and $S$ tuples.*

*Proof.* We want to compute a set of nodes $S_{migrate} \subseteq S$, where any node $i$ in $S_{migrate}$ migrates local matching tuples. Observe that the migration destination does not affect the network cost, thus, can be any node in $S \setminus S_{migrate}$. Let $m_i$ be the binary decision whether node $i$ keeps all local matching $S$ tuples ($m_i = 0 \Leftrightarrow i \in S \setminus S_{migrate}$) or migrates them ($m_i = 1 \Leftrightarrow i \in S_{migrate}$). The cost of migrating selective broadcast from $R$ to $S$ is:

$$\sum R_i \cdot \sum (1 - m_i) - \sum R_i \cdot (1 - m_i) + \sum S_i \cdot m_i$$

The $\sum R_i$ term is the total size of $R$ tuples, the $\sum(1-m_i)$ is the number of locations with $S$ tuples, the $\sum R_i \cdot (1-m_i)$ are the $R$ tuples that were local during broadcast and $\sum S_i \cdot m_i$ is the cost of migrating $S$ tuples. Since $\sum R_i (\equiv R)$ and $\sum S_i (\equiv S)$ are independent of all $m_i$, the formula can be minimized by checking each $m_i$ independently. Finally, since $S_{migrate}$ cannot contain all nodes, we never consinder adding the node $i$ with the largest $R_i + S_i$ (and $S_i > 0$) in $S_{migrate}$ ($m_i = 0$). □

**Theorem 2.** *Between the two selective broadcast schedules ($R \to S$ or $S \to R$), optimized by adding a migration phase, the one with the lowest network cost is also the schedule with the minimum network traffic for the single key (Cartesian product) join.*

*Proof.* Given the set $S_{migrate}$, after migrating $S$ from $S_{migrate}$ nodes and before executing the selective broadcast of $R$ tuples to $S$ locations, migration of any $R$ tuples cannot reduce the network cost further. Assume we migrate $R$ tuples from node $x$ to node $y$. If $y \in S \setminus S_{migrate}$, then the network cost remains the same, since we moved the $R$ tuples from node $x$ once, increasing the network cost by $R_x$. However, now the $R$ tuples are local to the $S$ tuples at node $y$ and we can skip node $y$ during the selective broadcast, reducing the network cost by $R_x$. Across the two phases, the network cost remains the same. On the other hand, if $y \in S_{migrate}$, the network cost of the migration phase increases by $R_x$ without reducing the network cost of the selective broadcast phase. Thus, the network cost across the two phases increases. Thus, migrating tuples from both tables is meaningless. □

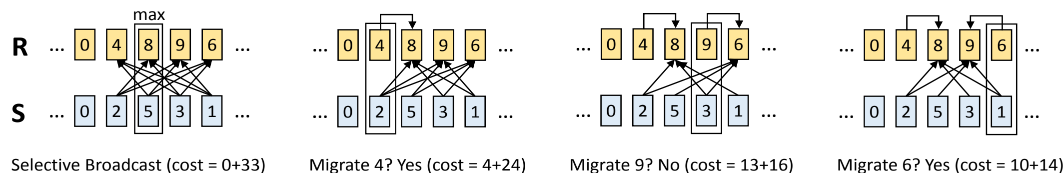



Figure 2.2: Example of 4-phase track join optimal schedule generation per distinct join key

What the last proof means, is that the migrate and broadcast mechanism is sufficient to generate optimal schedules for single key joins. There are no plans that would migrate data from both tables with a lower network cost. Similarly, there are no other tuple transferring mechanisms that can achieve a lower network cost, without taking into account the cost of the tracking phase. Thus, the mechanism of 4-phase track join is sufficient to minimize the network traffic at the finest granularity possible.

In Figure 2.2 we show an example of 4-phase track join schedule generation for a single join key. Initially, we compute the cost of the selective broadcast and check for each node whether migrating its matching tuples reduces the total network cost. The nodes that have no matching tuples from either $R$ or $S$ are not considered at all during schedule generation. The time to generate each schedule depends on the number of tuples per join key. The input of schedule generation is an array of triplets with (i) key, (ii) node identifier, and (iii) tuple size (width). If a node does not contain a key, the triplet for that node is not generated at all. Thus, schedule generation is linear to the number of input tuples from both tables. Note that track join is a greedy algorithm, thus we use the term *minimal* instead of the term *minimum*. While we proved that the amount of payload data transferred is indeed minimum, we cannot always guarantee minimal network traffic throughout all phases of the join, since the tracking phase cannot be optimized.

Using multiple threads is essential to achieve good performance on all CPU intensive tasks, even when the complexity is linear. The optimization becomes more important if the data remain main-memory resident throughout the join. Track join allows all in-process operations across keys to be parallelized across threads freely, since all local operations combine tuples with the same join key only. Recent work showed that dynamic thread sharing [Leis *et al.*, 2014] is essential to achieve good performance when multiple operators are pipelined. Such techniques are usable in track join.

Finally, while the algorithmic descriptions of track join in this section implement track join by pipelining different operators, track join can also be implemented by fully separating the steps. Each step materializes the results in memory before the following step. In such an implementation, we do not separate processes $R$, $S$, and $T$, since they run in successive steps and not in parallel. Each step can be implemented to use multiple threads internally. In Section 2.4, we use this approach to implement track join and evaluate its running times.

### 2.2.4 Traffic Compression

Track join, as described so far, makes absolutely no effort to reduce the data footprint by compressing the data before sending them over the network. Modern analytical database systems employ distinct column value dictionaries [Raman *et al.*, 2013; Willhalm *et al.*, 2009]. The compression process can occur offline, and data can be processed in compressed form throughout the join. In this section, we briefly discuss some techniques that can further reduce network traffic on top of track join.

A simple compression technique is delta encoding [Lemire and Boytsov, 2015; Stonebraker *et al.*, 2005]. With sufficient spare CPU cycles, we can sort all columns before sending them over the network. Track join imposes no specific message ordering, besides the barriers between phases. The highest compression ratio is achieved by sorting.

A second approach is to create common prefixes by partitioning the data at the source nodes before sending anything over the network. For instance, we can radix partition the first $b_p$ out of $b$ bits and pack $(b - b_p)$-bit suffix with a common prefix. We can tune the compression rate by employing more partition passes to create wider prefixes, since each radix partitioning pass is very fast and runs as fast as a memory-to-memory copy [Satish *et al.*, 2010; Wassenberg and Sanders, 2011]. If the column values are dictionary encoded throughout the join, the number of distinct dictionary indexes that share the same prefix is maximized and the common prefix method is more effective.

The messages sent over the network that drive the migration and the selective broadcast phases are a pair of a key and a node id each. We can group the pairs that have the same node id together and avoid transferring the node ids on every message. Instead, we send all keys tagged with the same node id as a group and reduce the network footprint.

## 2.3 Query Optimization

The formal model of track join is used by the query optimizer to decide whether to use track join, hash join, or broadcast join. We prove track join superior to hash join, even after making the latter tracking-aware using globally-unique record identifiers (rids). Finally, we study how track join interacts with Bloom filters in semi-joins.

### 2.3.1 Network Cost Model

In this section, we assume a uniform distribution of tuples across nodes. This is the worst case for track join since it precludes locality. Track join optimization is not limited to one specific distribution as we will explain in this section.

The network cost of the standard hash join with early materialized payloads is:

$$\frac{N-1}{N} \cdot (t_R \cdot (w_k + w_R) + t_S \cdot (w_k + w_S))$$

where $t_R$ and $t_S$ are the tuple counts of tables $R$ and $S$, $w_k$ is the total width of the join key columns used in conjunctive equality conditions, while $w_R$ and $w_S$ are the total width of relevant payload columns projected for the later steps of the query.

To define the network cost of 2-phase track join, we first determine the cost of tracking. The number of nodes that contain matches for each key is upper bounded by $N$ and is $t/d$, in the worst case when equal keys are randomly distributed across nodes, where $t$ is the number of tuples and $d$ is the number of distinct values. We use the terms $n_R \equiv \min(N, t_R/d_R)$ and $n_S \equiv \min(N, t_S/d_S)$ for these commonly reused quantities. We also use $\mathcal{N} = \frac{N-1}{N}$ to exclude keys that are already in their hash destination.

We define the input selectivity ($s_R$ and $s_S$) as the percentage of tuples of one table that have matches in the other table, after applying all other selective predicates. The number of distinct nodes with matching payloads also includes the input selectivity factor. We assume that the selective predicates do not use the key column and the number of distinct keys is unaffected. Again, we synopsize these commonly reused quantities using the terms $m_R \equiv \min(N, (t_R \cdot s_R)/d_R)$ and $m_S \equiv \min(N, (t_S \cdot s_S)/d_S)$.

Using the terms described above, the cost of 2-phase track join is:

$$\begin{aligned} \mathcal{N} \cdot (d_R \cdot n_R + d_S \cdot n_S) \cdot w_k & \quad \text{(track R and S keys)} \\ + d_R \cdot m_R \cdot m_S \cdot w_k & \quad \text{(transfer S locations)} \\ + t_R \cdot s_R \cdot m_S \cdot (w_k + w_R) & \quad \text{(transfer R} \rightarrow \text{S tuples)} \end{aligned}$$

For 3-phase track join, we transfer counts during tracking alongside the keys. The $t/(d \cdot s)$ fraction gives the average repetition of keys on each node if the distribution and node placement is assumed to be uniform random. We use this metric to estimate how many bits to use for representing counters (or tuple widths). We refer to the counter lengths using $c_R \equiv \log(t_S/(d_S \cdot n_S))$ and $c_S \equiv \log(t_S/(d_S \cdot n_S))$. Even if some keys occur frequently enough to exceed the maximum count, we can still aggregate at the destination before the schedule generation. The cost of 3-phase track join is:

$$\begin{aligned} \mathcal{N} \cdot d_R \cdot n_R \cdot (w_k + c_R) + d_S \cdot n_S \cdot (w_k + c_S) & \quad \text{(tracking)} \\ + d_{R_1} \cdot m_{S_1} \cdot w_k + t_{R_1} \cdot s_{R_1} \cdot m_{S_1} \cdot (w_k + w_{R_1}) & \quad (R_1 \rightarrow S_1) \\ + d_{S_2} \cdot m_{R_2} \cdot w_k + t_{S_2} \cdot s_{S_2} \cdot m_{R_2} \cdot (w_k + w_{S_2}) & \quad (S_2 \rightarrow R_2) \end{aligned}$$

In this formula, we assume the tuples are split into two separate classes $R_1$, $S_1$ and $R_2$, $S_2$. The tuples of $R_1$ are defined based on whether the $R \rightarrow S$ direction is less expensive than the $S \rightarrow R$ direction at distinct join key granularity.

Picking which variant of track join to use is a much simpler task than estimating the network cost of the join. 2-phase track join suffices when both tables have almost entirely unique keys, or one table has more repeats per value than the number of nodes. In the latter case, a broadcast join would be used. 3-phase track join is preferred when we cannot estimate the cardinality of the relations and want to avoid the inner–outer table distinction. If the number of output tuples exceeds the number of input tuples, such as in $m$-$n$ joins, 4-phase track join should be used to generate the optimal transfer schedules. The same is true if significant pre-existing locality is detected in the workload or if locality is generated artificially prior to executing the joins.

If the query optimizer must estimate the join cost rather than a relative comparison of available algorithms, we can estimate the network cost of track join by splitting the input in classes with different degrees of correlation. For example, the $R_1$ and $R_2$ classes used in the cost of 3-phase track join represent the percentages of tuples of $R$ that are joined using the $R \rightarrow$ (for $R_1$), or the $S \rightarrow R$ (for $R_2$) selective broadcast direction.

To populate the classes, we can use *correlated sampling* [Yu *et al.*, 2013], a technique that preserves the join relationships of tuples. Correleated sampling works independently of the distribution, and can be generated off-line. The sample is augmented with initial placements of tuples. Besides computing the exact track join cost, we incrementally classify the distinct join keys to correlation classes based on estimated network traffic per schedule.

The cost of 4-phase track join is shown below, using two classes for 3-phase track join ($R_1$, $S_1$, $R_2$, $S_2$) and one for hash join ($R_3$, $S_3$):

$$\begin{aligned}
\mathcal{N} \cdot d_R \cdot n_R \cdot (w_k + c_R) + d_S \cdot n_S \cdot (w_k + c_S) &\quad \text{(tracking)} \\
+d_{R_1} \cdot m_{R_1} \cdot m_{S_1} \cdot w_k + t_{R_1} \cdot s_{R_1} \cdot m_{S_1} \cdot (w_k + w_{R_1}) &\quad (R_1 \rightarrow S_1) \\
+d_{S_2} \cdot m_{R_2} \cdot m_{S_2} \cdot w_k + t_{S_2} \cdot s_{S_2} \cdot m_{R_2} \cdot (w_k + w_{S_2}) &\quad (S_2 \rightarrow R_2) \\
+d_{R_3} \cdot n_{R_3} \cdot w_k + t_{R_3} \cdot s_{R_3} \cdot (w_k + w_{R_3}) &\quad (R_3 \rightarrow \text{h(k)}) \\
+d_{S_3} \cdot n_{S_3} \cdot w_k + t_{S_3} \cdot s_{S_3} \cdot (w_k + w_{S_3}) &\quad (S_3 \rightarrow \text{h(k)})
\end{aligned}$$

The three correlation classes that combine hash join with 3-phase track join combine broadcasting as well as partitioning to a common destination. If higher estimation accuracy is required, we can create more classes as step functions between the above.

Track join invests significant time working with keys compared to early materialized hash join. As as result, when the payloads are small, we expect track join to perform worse than hash join in the absence of locality. For instance, if the joined tables have the same number of tuples ($t_R = t_S$), both tables have entirely unique keys, and the join selectivity is 0.5 (all tuples match), then the the larger payload must be twice larger than the join key, in order for track join to send less data than hash join:

$$4 \cdot w_k + \min(w_R, w_S) \leq 2 \cdot w_k + w_R + w_S \quad \Leftrightarrow \quad 2 \cdot w_k \leq \max(w_R, w_S)$$

### 2.3.2 Tracking-Aware Hash Join

Late materialization is a popular approach for designing main memory databases because it allows for operators to be performed on the keys alone. The payloads are only accessed when needed using rids. The technique is typically used in column stores [Raman *et al.*, 2013; Stonebraker *et al.*, 2005], where each column is stored as a fixed width array and any column value can be accessed as an array slot.

Late materialization can also be viewed as a query optimizer decision of whether to defer payload accesses or to process payloads alongside the keys. Interleaving late materialized operators to exploit high selectivities is out of the scope of this work.

In the simple late materialized hash join, join keys are hashed, rids are implicitly generated (0 to $N$), and payloads are fetched afterwards. The network cost is:

$$\mathcal{N} \cdot (t_R + t_S) \cdot w_k + t_{RS} \cdot (w_R + w_S + \log t_R + \log t_S)$$

where $t_R S$ is the number of joined tuples. The join (output) selectivity is also defined as $s_{RS} = t_{RS}/(t_R + t_S)$. Note that $s_{RS}$ can be arbitrarily higher than 1 in $m$-$n$ joins.

When distributed, rids contain a local rid and a node identifier. To use the implicit tracking information carried in an rid would be to migrate the result to the tuple location, instead of fetching the tuples where the rid pair is. For instance, assume a join of $R$ and $S$, where fetching $S$ payloads is costlier than $R$. Using the $R$ rid, we migrate the $R$ rid where the $R$ tuple resides. The $S$ tuple is sent to the location of the $R$ tuple, after a request with the $S$ rid and the $R$ destination.

Assuming that no later operation precedes payload fetching, we can further elaborate the improvement by redoing the join on the final destination. In the above example, we send one local id to the $R$ node and the local id to the $S$ node, alongside the node id of the $R$ node. Sending the node id can be avoided as in track join. The payload brought from the $S$ node will be coupled with the join key. The $R$ rid will access the tuple and rejoin it with the incoming tuple from the $S$ node. The network cost is:

$$\mathcal{N} \cdot (t_R + t_S) \cdot w_k + t_{RS} \cdot (\min(w_R, w_S) + w_k + \log t_R + \log t_S)$$

We show that late materialized hash join is less effective than track join in minimizing the total network traffic. In the initial phase, late materialized hash join transfers the key column alongside rids, without eliminating any duplicate keys. In contrast, track join transfers either the distinct keys on each node, or pairs of distinct keys counts, as a result of the aggregation step. In the later phases, track join has minimized the network transfers.

The rid-based hash join transfers payloads from the shorter side to all locations of the larger side where there are matching tuples. This is the same schedule as 2-phase track join. On the other hand, track join resends the join keys have shorter representation than rids when compressed. As a result, the simplest 2-phase track join transfers less data over then network than the tracking-aware hash join using network-global rids.

The extra cost of transferring rids is non-trivial in most cases. As shown in our experiments, real workloads may use less than 8 bytes of payload data while globally unique rids must be at least 4 bytes. A pair of rids from both input tables may be wider than the payloads with smaller size, making late materialization less effective for distributed query execution compared to local in-memory query execution.

### 2.3.3 Semi-Join Filtering

When join operations are coupled with selections, we can prune tuples that do not qualify from the selective conditions, both individually per table and across tables. To that end, databases use semi-joins [Bernstein and Chiu, 1981; Mullin, 1990] implemented via Bloom filters [Bloom, 1970]. Bloom filters are already optimized towards network traffic due to their minimal size. In our analysis here, we assume a two-way Bloom-join. In two-way Bloom-joins, both input tables prune tuples from the opposite table using a Bloom filter, prior to sending any tuples over the network to execute the distributed join.

When a false positive occurs, hash join transfers the whole tuple in vain. In track join, the matches are determined when the keys are tracked. All join keys with no matches are discarded up front using the key projection; no location or tuples are transferred thereafter. Whereas late materialized hash join reduces the penalty of filter errors by using key and rid pairs, track join sends less than the key column alone.

Assuming the length per qualifying tuple in the filter to be $w_{bf}$, the cost of early mate-

rialized hash join that uses Bloom filters as an initial step is:

$$\begin{aligned} (t_R \cdot s_R + t_S \cdot s_S) \cdot N \cdot w_{bf} & \quad \text{(broadcast filters)} \\ +\mathcal{N} \cdot t_R \cdot (s_R + e) \cdot (w_k + w_R) & \quad \text{(transfer S tuples)} \\ +\mathcal{N} \cdot t_S \cdot (s_S + e) \cdot (w_k + w_S) & \quad \text{(transfer R tuples)} \end{aligned}$$

where $e$ is the relative error of the Bloom filters.

The cost of late materialized hash join that uses Bloom filters as an initial step is:

$$\begin{aligned} (t_R \cdot s_R + t_S \cdot s_S) \cdot N \cdot w_{bf} & \quad \text{(broadcast filters)} \\ +\mathcal{N} \cdot t_R \cdot (s_R + e) \cdot (w_k + \log t_R) & \quad \text{(transfer S pairs)} \\ +\mathcal{N} \cdot t_S \cdot (s_S + e) \cdot (w_k + \log t_S) & \quad \text{(transfer R pairs)} \\ +t_{RS} \cdot (w_R + w_S + \log t_R + \log t_S) & \quad \text{(fetch payloads)} \end{aligned}$$

where $t_{RS}$ includes the standard output selectivity factor.

The cost of 2-phase track join that uses Bloom filters as an initial step is:

$$\begin{aligned} (t_R \cdot s_R + t_S \cdot s_S) \cdot N \cdot w_{bf} & \quad \text{(broadcast filters)} \\ +\mathcal{N} \cdot d_R \cdot (s_R + e) \cdot me_R \cdot w_k & \quad \text{(track filtered R)} \\ +\mathcal{N} \cdot d_S \cdot (s_S + e) \cdot me_S \cdot w_k & \quad \text{(track filtered S)} \\ +d_R \cdot s_R \cdot m_S \cdot w_k & \quad \text{(transfer S locations)} \\ +t_R \cdot s_R \cdot m_S \cdot (w_k + w_R) & \quad \text{(transfer R tuples)} \end{aligned}$$

where $me_R \equiv \min(N, t_R \cdot (s_R + e)/d_R)$ and similarly for $me_S$.

The cost of broadcasting the Bloom filters may exceed the cost of transferring a few payload columns over the network for a reasonable number of nodes $N$. Track join implicitly executes perfect semi-join filtering during the tracking phase. Consequently, by using track join, we are more likely to not need the initial Bloom filter step compared to both early materialized hash join, that sends all key and payload columns during hash partitioning, and late materialized hash join, that transfers only keys and rids initially.

## 2.4 Experimental Evaluation

Our evaluation is split into three parts. First, we simulate distributed joins to measure network traffic across encoding schemes (Section 2.4.1). Then, we implement and measure track join against hash join execution times (Section 2.4.2).

### 2.4.1 Simulations

In the simulations shown in this section, we measure the total network traffic. We assume 16 network nodes and show 7 cases of distributed join algorithms: broadcast join $R \to S$ (`BJ_R`) and $S \to R$ (`BJ_S`), hash join (`HJ`), 2-phase track join $R \to S$ (`2TJ_R`) and $S \to R$ (`2TJ_S`), 3-phase track join (`3TJ`), and 4-phase track join (`4TJ`).

In the three experiments shown in Figure 2.3, the width of $R$ tuples varies between 20 and 60 bytes, while the $S$ tuples are fixed at 60 bytes. The key width (included in the tuple width) is always 4 bytes. The tables have equal cardinality ($10^9$) and have (both the same) unique keys. Thus, track join selectively broadcasts tuples from the table with smaller payloads to the one matching tuple from the table with larger payloads and 2-phase track join is sufficient to minimize the network traffic.

In the general case, hash join has $1/N^k$ probability in order for all $k$ matching tuples to be on the same node that matches the hash destination of the join key. For track join, this probability is $1/N^{k-1}$ because the collocation can occur in any node. When both tables have almost unique keys (Figure 2.3), the network cost gap between hash join and track join is maximized since $k \approx 2$ is the lower bound for equi-joins.

In the three experiments shown in Figure 2.4, we vary the tuple width ratio $R$ and $S$ as

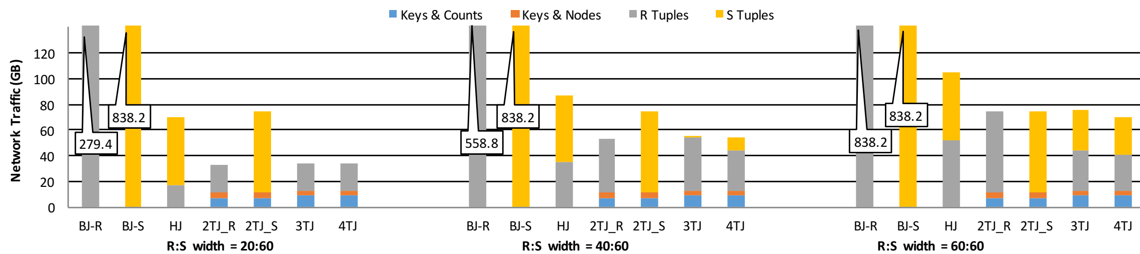



Figure 2.3: Synthetic $10^9$ unique $R$ tuples $\bowtie$ $10^9$ unique $S$ tuples ($1 \times 1$ key matches)

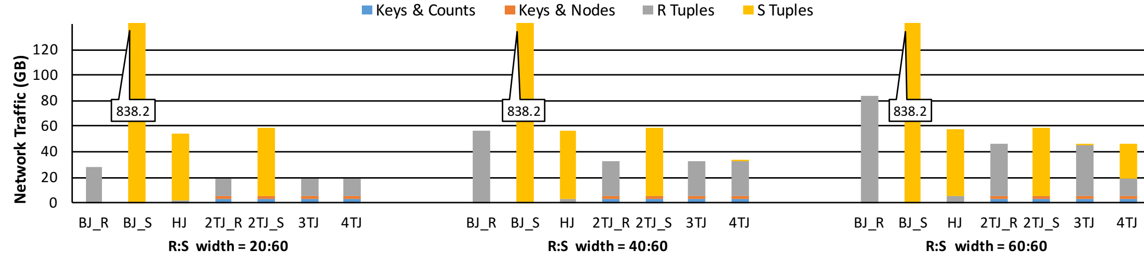



Figure 2.4: Synthetic $2 \cdot 10^8$ unique $R$ tuples $\bowtie$ $10^9$ non-unique $S$ tuples ($1 \times 5$ key matches)

in Figure 2.3, but assume a foreign-key join where $R$ has $10^8$ unique tuples and $S$ has $10^9$ tuples. The match rate is 100%, thus each unique tuple in $R$ is repeated 10 times in $S$. The cost of broadcasting $R$ is competitive and is actually better than hash join when the tuple width ratio of $R$ to $S$ is $1/3$. Again, the network cost is minimized using 2-phase track join that broadcasts the $R$ tuples to all locations with matching $S$ tuples.

In Figure 2.5, we show how locality can affect track join performance. We assume $R$ to have 200 million 30-byte tuples with unique keys and $S$ to have 1 billion 60-byte tuples. Each distinct join key in $S$ is repeated 5 times but key locations follow the patterns that are specified below each chart. The pattern `5,0,0,...` denotes that all 5 tuples with the same keys are on the same node. The pattern `2,2,1,0,0,...` represents 2 nodes with 2 matching tuples per key and 1 node with 1 matching tuple. The pattern `1,1,1,1,1,0,0,...` denotes that all tuples with matching keys are on different nodes. Naturally, the nodes for placing the keys to satisfy each pattern, are chosen randomly per distinct join key. Overall, these placements are artificially generated to represent different degrees of locality.

When all repeats are collocated on the same node, track join will only transfer matching keys to a single location. When the tuple repeats follow the `2,2,1,0,0,...` pattern, still

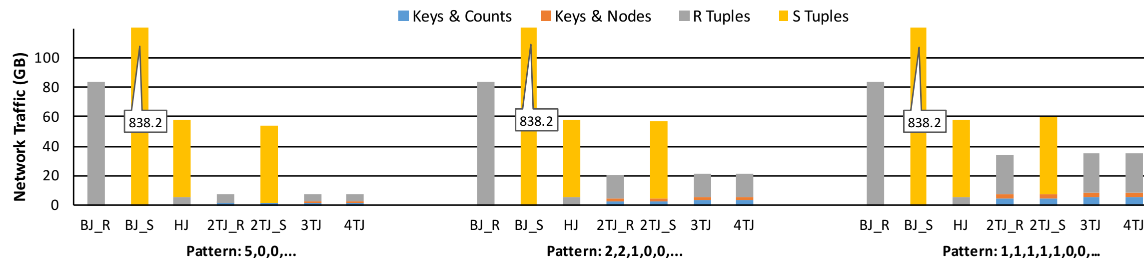



Figure 2.5: Synthetic $4 \cdot 10^7$ unique $R$ tuples $\bowtie$ $2 \cdot 10^8$ $S$ tuples ($1 \times 5$ key matches)

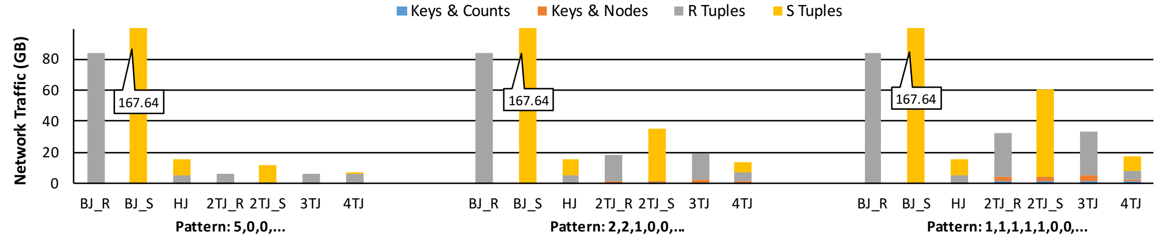



Figure 2.6: Synthetic $2 \cdot 10^8$ $R$ tuples $\bowtie$ $2 \cdot 10^8$ $S$ tuples with $4 \cdot 10^7$ unique keys ($5 \times 5$ key matches intra-table collocated)

better than uniform placement, the traffic is still reduced compared to hash join. Note that random placement collocates repeats to some degree.

We now do the same experiment using small tables. Each table has 40 million tuples with unique keys and repeats each key 5 times. Since each key has 5 repeats on each table, we generate 25 output tuples per distinct join key. The datasets were generated to have the same number of output tuples (1 billion). In Figures 2.6 and 2.7 we use 200 million tuples with 40 million distinct join keys for both tables, thus each distinct join key produces 25 output tuples. The tuple widths are 30 bytes for $R$ and 60 bytes for $S$.

The difference between Figure 2.6 and Figure 2.7 is whether the tuples across tables are collocated. In the first, the `5,0,0,...` configuration denotes that all repeats from $R$ are on a single node and that all repeats from $S$ are also on a (different) single node. We term this case, shown on Figure 2.6, *intra-table* collocation. When same key tuples from across tables are collocated, locality is further increased. We term this case, shown on Figure 2.7, *inter-table* collocation. When all key repeats are collocated, track join transfers no payloads. The messages used during the tracking phase to track the tuple locations can only be affected by the same case of locality as hash join.

Figures 2.4, 2.5, 2.6, and 2.7 show the weakness of 2-phase and 3-phase track join to handle all possible equi-joins. If both tables have values with repeating keys shuffled randomly with no repeats in the same node, 4-phase track join is similar to hash join. However, track join sends far less data when the matching tuples are collocated.

In the rest of our experimental evaluation for both simulation and the implementation of track join (Sections 2.4.1 and 2.4.2), we will use queries from real analytical workloads. We did extensive profiling among real commercial workloads with one sophisticated commercial

| ***R*** **column** | **Cardinality** | **Bits** | | ***S*** **column** | **Cardinality** | **Bits** |
|---|---|---|---|---|---|---|
| J.ID (key) | 769,785,856 | 30 | | J.ID (key) | 788,463,616 | 30 |
| O.U.AMT | 26,308,608 | 25 | | T.ID | 53 | 6 |
| C.ID | 359 | 9 | | S.B.ID | 95 | 7 |
| T.B.C.ID | 233,040 | 18 | | T.ID | 53 | 6 |
| | | | | J.T.AMT | 9,824,256 | 24 |
| | | | | T.C.ID | 297,952 | 19 |
| | | | | S.C.AMT | 11,278,336 | 24 |
| | | | | M.U.AMT | 54,407,160 | 26 |
| | 769,845,120 | | | | 790,963,741 | |

Table 2.1: Column details of $R$ and $S$ tables joined in Q1

DBMS. The commercial hardware where we did the profiling is an 8-node cluster connected by 40 Gbps InfiniBand. Each node has 2X Intel Xeon E5-2690 CPUs at 2.9 GHz, 256 GB of quad-channel DDR3 RAM at 1600 MHz, 1.6 TB flash storage with 7 GB/s bandwidth, and 4X 300 GB hard disks at 10,000 RPM.

In the next experiment we use the slowest query (`Q1`) from a real workload ($X$). All query plans were generated by the same commercial DBMS. Query `Q1` includes a hash join that we profiled to take 23% of its total execution time. Details about the columns of two inputs participating in the slowest join are shown in table 2.1. They are both intermediate relations and consist of 769,845,120 and 790,963,741 tuples respectively. The join output has 730,073,001 tuples. Overall, query (`Q1`) executes 7 joins, after applying selective conditions on 4 relations, and then performs a single final aggregation.

All columns of the `Q1` join are of number type and would use variable byte encoding by default, if left uncompressed. Sending uncompressed data over the network is expensive. Compression helps all join algorithms reduce their network traffic, since the column values

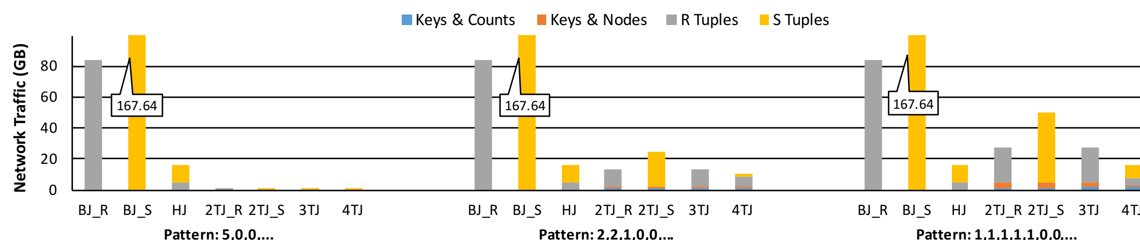



Figure 2.7: Synthetic $2 \cdot 10^8$ $R$ tuples $\bowtie$ $2 \cdot 10^8$ $S$ tuples with $4 \cdot 10^7$ unique keys ($5 \times 5$ matches inter-table collocated and intra-table collocated)

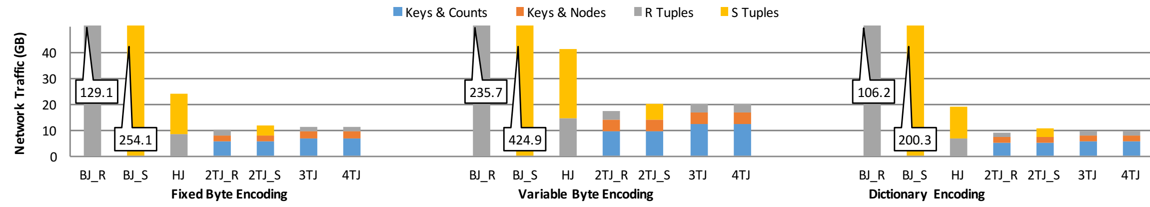



Figure 2.8: Slowest join of slowest query (Q1) of workload $X$ (original data placement)

are compressed using a global dictionary and use the minimum number of bits. Specifically for track join, we send keys over the network more often than we do payloads. Also, the tracking phase cannot be optimized since the key locations have not yet been tracked. Thus, for track join, compressing the key columns will have a greater impact in reducing the network traffic compared to compressing the payloads.

In Figure 2.8, we show the network traffic of `Q1`. We use 3 different encoding schemes, fixed byte (1, 2 or 4 byte) encoding, fixed bit dictionary encoding, and variable byte encoding. The variable byte encoding is base-100 and stores two decimal digits per byte. The input of `Q1` exhibits pre-existing locality, since broadcasting $R$ transferred $\approx 1/3$ of $R$ tuples compared to hash join. We would expect both hash join and track join to transfer the whole $R$ table over the network. We shuffle the placement of the tuples across the cluster to remove any data locality and repeat the experiments in Figure 2.9. As expected, the network traffic increases as we now have to transfer most $R$ payloads over the network. Nevertheless, track join still outperforms hash join.

In the $X$ workload, the key columns have less than 1 billion distinct values, and thus, fit in a 32-bit integer if dictionary encoded. However, the actual values do not fit in the 32-bit range. In some cases, we can effectively reduce the data width using simple compression schemes [Lemire and Boytsov, 2015] that allow fast decompression

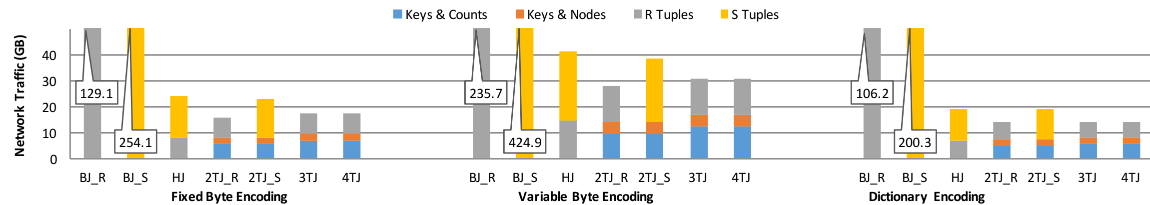



Figure 2.9: Slowest join of slowest query (Q1) of workload $X$ (shuffled data placement)

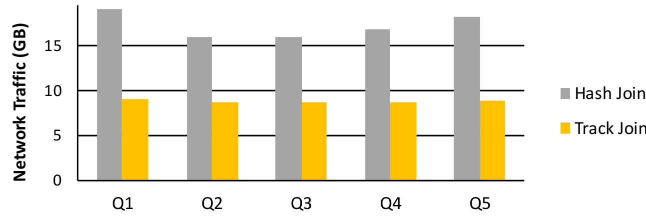
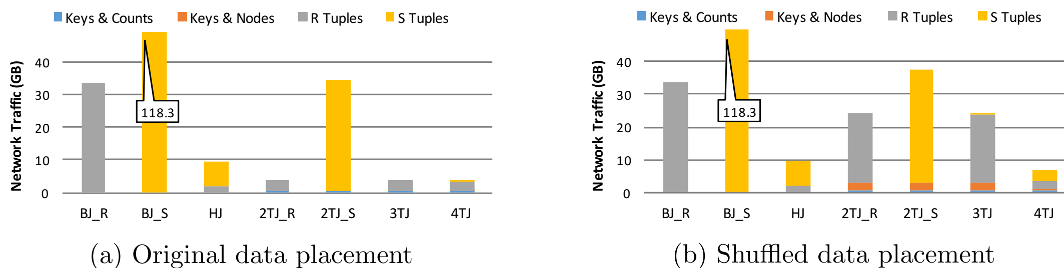



Figure 2.10: Common slowest join in slowest queries 1–5 of $X$ (dictionary compressed)

and do not require accessing large dictionaries that are unlikely to remain CPU-cache-resident. On the other hand, dictionaries reduce the width of data types and, if sorted, can perform equality and ordering operations with the dictionary index. Decoding arbitrary $n$-bit dictionary indexes can be implemented efficiently [Lemire and Boytsov, 2015; Willhalm *et al.*, 2009]. However, in many operations, including non-selective equi-joins, dictionary accesses are redundant. In our results for compressed workload $X$, we omit dictionary dereference since the join can use the encoded values of the join keys, given that the join key columns are encoded using the same dictionary.

In all experiments we omit dictionary dereference traffic. Dictionaries are also assumed to be compressed before the join, using the minimum number of bits required to encode the distinct values of the intermediate relation, even if some join keys have been filtered before the join. This is the optimal encoding scheme in cases where most keys are unique and there is specific value ordering. If the dictionaries are not compressed after every selection, which is the expected scenario, we would still rely on the original dictionary that encodes all distinct values of original tables. Encoding to the optimal compression before every join in order to achieve the minimal network traffic is possible, but the total gain in network traffic may be outweighed by the extra CPU time needed to re-compress.

Workload $X$ contains more than 1500 queries and the same most expensive hash join appears in the five slowest queries, taking 23%, 31%, 30%, 42%, and 43% of their execution time. The slowest five queries require 14.7% of the total time required for all queries in the workload ($> 1500$) cumulatively and spend $\approx$ 65–70% of their time on the network. Time was measured by running the entire $X$ workload on the commercial DBMS we use. The hash join in all five queries operates on the same intermediate result for the key columns, but each query uses different payloads. Queries Q2–Q5 are similar to Q1 and do 4–6 joins




(a) Original data placement

(b) Shuffled data placement

Figure 2.11: Slowest join in slowest query of workload $Y$ using variable byte encoding

followed by a final aggregation step.

In Figure 2.10, we compare hash and track join on the slowest five queries, using dictionary coding with the minimum number of bits, which is also the optimal compression scheme that we can apply here. The total bits per tuple for $R$:$S$ are 79:145, 67:120, 60:126, 67:131, and 69:145 respectively. The network traffic reduction is 53%, 45%, 46%, 48%, and 52% respectively. Here, both inputs have almost entirely unique keys. Thus, assuming we pick the table with shorter tuples ($R$ here) to selectively broadcast during 2-phase track join, all track join versions have similar performance.

In our last experiment we use a second real workload ($Y$) that has $\approx$ 90 analytical queries. We use the slowest query in $Y$ that executes 9 joins and isolate the most expensive hash join, based on profiling results from the commercial DBMS. The most expensive query takes 20% of the time required to run all queries of $Y$ cumulatively and the hash join takes about 36% of the total execution time of the most expensive query. In Figure 2.11a we show the network traffic of the join using the original data placement. In Figure 2.11b, we shuffle the inputs to remove all pre-existing locality.

The $R$ table has 57,119,489 rows and the $S$ table has 141,312,688 rows. The join output consists of 1,068,159,117 rows. The data are uncompressed and use variable byte encoding. The tuples are 37 and 47 bytes wide, the largest part of which is due to a character 23-byte character column. The large output of the query makes it a good example of high join selectivity that has different characteristics from foreign and candidate key joins. Late materialization could cause this query to be excessively expensive, if payloads are fetched at the output tuple cardinality.

When the input is shuffled, the 4-phase version is better than hash join, while the

other versions almost broadcast $R$ due to key repetitions. The 2-phase track join can be prohibitive if there is no locality and we choose to broadcast $S$ tuples to $R$ tuple locations. This is due to sending $S$ tuples to too many nodes to match with $R$ and is shown in Figure 2.11b. The opposite broadcast direction is not as bad, but is still three times more expensive than hash join. 4-phase track join adapts to the shuffled case and transfers 28% less data than hash join. This experiment is a good example of the adaptiveness of track join. We need many repeated keys since the output cardinality is 5.4 times the input cardinality per distinct join key. The naïve selective broadcast of 2-phase and 3-phase track join are similar to broadcasting $R$ to all nodes.

### 2.4.2 Implementation

For the next set of experiments, we implement hash join and track join in C++ using portable operating system calls (POSIX) without any extra libraries. The network communication uses the TCP protocol and all local processing is done on RAM. We use four identical machines connected through an 1 Gbit Ethernet switch, each with 2X Intel Xeon X5550 CPUs at 2.67 GHz and 24 GB ECC RAM. Each CPU has 4 physical cores and 2 hardware threads per core. We measured the RAM bandwidth at 28.4 GB/s for loading data from RAM, at 16 GB/s for storing data to RAM, and at 12.4 GB/s for RAM to RAM copying. We compile the C++ source code using GCC 4.8 with `-O3` optimization. The operating system is the Ubuntu 14.04 GNU/Linux distribution and the Linux kernel version is 3.2. In all of our experiments, we use all 16 hardware threads of each machine.

Our platform is severely network bound since each edge can transfer 0.093 GB/s, if used exclusively. Although track join was presented using a pipelined approach in Section 2.2, we separate CPU and network utilization by de-pipelining all operations of track join. Thus, we can estimate performance on perfectly balanced and much faster networks by dividing the time of network transfers accordingly. We assume the data are initially on RAM and the join results are also materialized on RAM. If the result is too large, we write it in large parts. In Table 2.2 we show the CPU and network cost for hash join and track join.

Our implementation uses fixed byte widths for keys, payloads, and node identifiers. For the dataset of workload $X$, we use 4-byte keys, 7-byte $R$payloads, and 18-byte $S$ payloads.

| Dataset | Placement | Time | HJ | 2TJ | 3TJ | 4TJ |
|---|---|---|---|---|---|---|
| $X$ | Original | CPU time (s) | 4.308 | 5.396 | 6.842 | 7.500 |
| | | Network time (s) | 87.754 | 38.857 | 44.432 | 44.389 |
| | Shuffled | CPU time (s) | 4.598 | 6.457 | 7.601 | 8.290 |
| | | Network time (s) | 87.828 | 61.961 | 67.117 | 67.518 |
| $Y$ | Original | CPU time (s) | 2.301 | 2.279 | 3.355 | 2.400 |
| | | Network time (s) | 30.097 | 10.800 | 11.145 | 10.476 |
| | Shuffled | CPU time (s) | 2.331 | 2.635 | 3.536 | 2.541 |
| | | Network time (s) | 30.191 | 28.674 | 29.520 | 18.230 |

Table 2.2: CPU and network time of the slowest join of the slowest query of $X$ and $Y$

For the dataset of workload $Y$, we use 4-byte keys, 33-byte $R$ payloads, and 43-byte $S$ payloads. The node ids use 1 byte and the counts use 1 byte for $X$ and 2 bytes for $Y$.

Using workload $X$ in the original order, 2-phase track join (using the $R \to S$ selective broadcast direction) increases the CPU time by 25%, but reduces the network time by 56% compared to hash join. 3-phase and 4-phase track join increases the CPU time by 59% and 74% without further reducing the network traffic. If $X$ is shuffled, 2-phase track join, increases the CPU time by 40% and reduces the network time by 29%.

Using $Y$ in the original order, 2-phase track join suffices to reduce the network time by 64% compared to hash join without affecting the CPU time significantly. If $Y$ is shuffled, 2-phase and 3-phase track join increase the CPU time without saving any network traffic, due to excessive key repetitions. In fact, we would transfer more data than hash join, as shown Figure 2.11b, if we used more nodes, because the number of nodes is less than the key repeats. On the other hand, 4-phase track join increases the CPU time by 9% compared to hash join but reduces the network traffic by 40%.

We can project track join performance on 10 Gbit Ethernet by scaling the network time. In $X$, track join is $\approx$ 29% faster than hash join, and in $Y$, track join is $\approx$ 37% faster. Note that the hardware bundle of the commercial DBMS uses both faster CPUs (2X 8-core CPUs at 2.9GHz) and network (40 Gbit InfiniBand). If the network is too fast compared to the CPU, track join becomes less useful. Still, if the network is slow, track join is likely to reduce the total execution time, even if there is no locality in the dataset. If the dataset exhibits locality, track join will further increase the performance gap with hash join.

Table 2.3 shows the execution times (in seconds) per step for hash join. To execute

| **Step description** | **Dataset $X$** | | **Dataset $Y$** | |
|---|---|---|---|---|
| | Original | Shuffled | Original | Shuffled |
| Hash partition $R$ tuples by key | 0.347 | 0.350 | 0.054 | 0.054 |
| Hash partition $S$ tuples by key | 0.478 | 0.477 | 0.167 | 0.167 |
| Transfer $R$ tuples | 29.464 | 29.925 | 7.197 | 7.392 |
| Transfer $S$ tuples | 57.199 | 57.142 | 22.550 | 22.945 |
| Local copy tuples | 0.115 | 0.115 | 0.039 | 0.039 |
| Sort received $R$ tuples by key | 1.145 | 1.288 | 0.176 | 0.179 |
| Sort received $S$ tuples by key | 1.627 | 1.777 | 0.535 | 0.572 |
| Final merge-join | 0.601 | 0.602 | 1.322 | 1.321 |

Table 2.3: Distributed hash join time per step (in seconds)

hash join, we first hash partition both tables based on the join key. Then, we transfer each partition of both tables to the respective network node. Once the tables have been transferred over the network, each node has keys with the same hash destination. We then sort the tables locally using MSB radix-sort and perform a final merge-join on RAM. The transfer steps that use the network are significantly slower since the experiment is measured on 1Gbit Ethernet. Nevertheless, since the steps are completely de-pipelined, we can scale the network time for faster networks independently of the other steps.

Table 2.4 shows the execution time per step for 4-phase track join. First, we sort both tables locally. Then, we aggregate the tuples by key and count the number of tuples per group, to gather the information needed in the tracking phase. Then, we hash partition the keys and the counts by key and distribute them over the network. Then, we merge the keys and counts to create a mapping of matching tuples for each distinct join key. The mapping is used to generate the schedule of transfers that minimizes the network traffic for each Cartesian product join. We transfer keys and counts that participate in the $R \rightarrow S$ selective broadcast and pairs that participate in the $S \rightarrow R$ selective broadcast separately. In practice, we distinguish between two types of messages per direction, those about the initial migration and those about the later selective broadcast. After the messages are merged on each node by key, the keys are joined with payloads and are transferred over the network. At this point we transfer tuples for both the migration and the selective broadcast phases together. In the last step, we merge by key and perform the final merge-join.

The cost of scheduling and the local join depend on the dataset. Since $X$ has mostly

| **Step description** | **Dataset $X$** | | **Dataset $Y$** | |
|---|---|---|---|---|
| | Original | Shuffled | Original | Shuffled |
| Sort local $R$ tuples by key | 0.979 | 1.300 | 0.182 | 0.182 |
| Sort local $S$ tuples by key | 1.401 | 1.792 | 0.534 | 0.565 |
| Aggregate (local) keys $\Rightarrow$ counts | 0.229 | 0.227 | 0.022 | 0.025 |
| Hash partitition keys and counts by key | 0.373 | 0.372 | 0.011 | 0.018 |
| Transfer keys and counts | 26.800 | 27.339 | 0.977 | 1.378 |
| Local copy keys and counts | 0.034 | 0.034 | 0.093 | 0.001 |
| Merge received keys and counts | 0.506 | 0.507 | 0.015 | 0.022 |
| Schedule and partition by node id | 1.627 | 1.650 | 0.035 | 0.047 |
| Transfer $R \to S$ keys and node ids | 7.277 | 10.913 | 0.346 | 0.532 |
| Transfer $S \to R$ keys and node ids | 6.046 | 1.562 | 0.135 | 0.247 |
| Local copy keys and node ids | 0.016 | 0.016 | 0.000 | 0.000 |
| Merge received keys and node ids | 0.237 | 0.235 | 0.007 | 0.012 |
| Merge-join $R \to S$ keys and node ids $\Rightarrow$ payloads and partition by node id | 0.315 | 0.456 | 0.068 | 0.098 |
| Merge-join $S \to R$ keys and node ids $\Rightarrow$ payloads and partition by node id | 0.355 | 0.204 | 0.067 | 0.082 |
| Transfer $R \to S$ tuples | 2.664 | 27.532 | 6.086 | 9.600 |
| Transfer $S \to R$ tuples | 0.001 | 0.001 | 3.235 | 6.462 |
| Local copy $R \to S$ tuples | 0.067 | 0.017 | 0.007 | 0.009 |
| Local copy $S \to R$ tuples | 0.138 | 0.037 | 0.021 | 0.008 |
| Merge received $R \to S$ tuples | 0.161 | 0.531 | 0.045 | 0.067 |
| Merge received $S \to R$ tuples | 0.141 | 0.066 | 0.043 | 0.045 |
| Final merge-join $R \to S$ | 0.419 | 0.555 | 0.822 | 0.793 |
| Final merge-join $S \to R$ | 0.342 | 0.161 | 0.518 | 0.556 |

Table 2.4: 4-phase track join time per step (in seconds)

unique keys, scheduling is faster than the local join. On the other hand, in $Y$, schedule generation is more expensive, but remains negligible compared to the total execution time. The final merge-join in $Y$ is expensive since the $M - N$ join produces a larger number of output tuples compared to the sum of both inputs.

All track join versions, as shown in Tables 2.2 and 2.4, initially sort the tables locally and then maintain the tables in sorted order via merging tuples received over the network in sorted order. This approach guarantees that we will remain cache-conscious throughout the track join variants and we can avoid relying on large hash tables that will be larger than the cache and thus incur cache misses. Since track join needs the data grouped by key in multiple steps, merging by key is cheaper than scattering to a new large hash table after

every time a tuple (or a key and count pair, or a key and node id pair) is transferred.

The original ordering of tuples in dataset $X$ exhibits locality skew among nodes. In the absence of load balancing, the improvement of track join over hash join by taking advantage of locality is reduced. If some nodes exhibit more locality than others, we need to take into account the balancing of transfers and not only aim for minimal network traffic. The approach of [Rödiger *et al.*, 2015] can help towards balancing the transfers across nodes by re-scheduling the transfers to minimize the end-to-end execution time.

Using large hash tables allows the algorithm to work in a pipelined fashion, as described in Section 2.2. However, track join cannot overlap steps across separate *phases*; pipelining takes place within each phase. A pipelined implementation can reduce end-to-end time by overlapping CPU and network [Kim *et al.*, 2012]. Track join is more complex than hash join, offering more choices for overlap. Thus, investigating alternative implementations that overlap CPU and network to minimize time is a crucial problem.

## 2.5 Conclusion

We presented track join, a novel algorithm for distributed joins that minimizes communication and therefore, network traffic. To minimize the number of tuples transferred across the network, track join generates an optimal transfer schedule for each distinct join key after tracking initial locations of tuples. Track join makes no assumptions about data organization in the DBMS and does not rely on schema properties or pre-existing favorable data placement to show its merits.

We described three versions of track join, studied cost estimation for query optimization, compared the interaction of track join with semi-join filtering, and showed that track join is better than tracking-aware hash join. Our experimental evaluation shows the efficiency of track join at reducing network traffic and shows its adaptiveness on various cases and degrees of locality. Our experimental evaluation shows that we can reduce both network traffic and total execution time. The workloads were extracted from a corpus of commercial analytical workloads and the queries were profiled as the most expensive out of all the queries in each workload using a market-leading commercial DBMS.

# Chapter 3

# In-Memory Partitioning and Large-Scale Sorting

## 3.1 Introduction

The increasing main-memory capacity of contemporary hardware allows query execution to occur entirely in memory. If the entire database also fits in RAM, analytical query workloads that are typically read-only need no disk access after the initial load. Thus, memory bandwidth is the primary performance bottleneck for query execution. Relational analytics are at the core of business intelligence tasks today, thus, the need for high-throughput in-memory query execution is apparent.

To maximize memory bandwidth and capacity, a few CPUs can be combined in a shared-memory system using a fast interconnection. Such hardware combines the parallelism of multiple multi-core CPUs with a higher aggregate memory bandwidth. The shared-memory functionality is provided by the non-uniform-memory-access (NUMA) interconnection, adding an additional layer in the memory hierarchy.

In a modern multi-core processor, the best performance is achieved when all cores work in a shared-nothing fashion and the working set is small enough to fit in the higher levels of the CPU cache that are also private per core. The same *cache-conscious* approach was more efficient even before the advent of multi-core CPUs, since random memory accesses are too expensive out of the cache.

Query execution is decomposed into a series of operations, the most time consuming of which are typically joins and aggregations. To speed up these operations using hardware parallelism, we partition into small pieces using the keys, then process each piece independently. For instance, an efficient algorithm for joins is to hash partition in parallel until the input is split into cache resident pieces before we execute the join using a hash table [Manegold *et al.*, 2002]. In fact, even in-cache join can further partition into trivial parts with very few distinct items, before executing a nested loop to join them [Kim *et al.*, 2009].

This chapter considers a comprehensive menu of partitioning options across several dimensions. The three *types* of partitioning are hash, radix and range partitioning, depending on the function that takes the key as an input and outputs the destination partition. Partitioning also depends on the layer of the memory hierarchy that it targets, namely in-cache, out-of-cache and across NUMA regions. Finally, we distinguish partitioning variants based on whether they use auxiliary space that is linear to the input size or not.

Until recently, prior work used the in-cache versions of partitioning and, if parallel, the non-in-place variant, which can be trivially distributed across threads. In all cases, when the input is larger than the cache, [Manegold *et al.*, 2000b] showed that the performance is throttled by TLB misses. [Satish *et al.*, 2010] highlighted that partitioning is a pathological case for caches and suggested using the cache as a buffer to mitigate TLB misses. [Wassenberg and Sanders, 2011] suggested using non-temporal writes on buffers that are as large as a cache line, to facilitate hardware write-combining.

Efficient out-of-cache partitioning assumes free access to auxiliary space in order to write the output. For cases where extra memory space in unavailable, we introduce several in-place out-of-cache partitioning variants utilizing the same crucial performance factors: an in-place method analogous to the shared-nothing non-in-place method; a modified non-in-place method that generates the output as a list of blocks that overwrites the input; and a parallel non-in-place method that combines the previous two.

To support scaling to multiple CPUs, we consider how the NUMA layer affects partitioning performance and modify both in-place and non-in-place out-of-cache partitioning to guarantee minimal transfers across NUMA boundaries. Also, we ensure that memory accesses that cross NUMA boundaries are always sequential, in order for hardware pre-fetching

to hide the latency of the NUMA interconnection layer [Albutiu *et al.*, 2012].

All of the algorithms mentioned so far target the *data shuffling* part of partitioning and implicitly assume that the *partition function* is cheap and can be computed at virtually no cost. While this assumption holds for radix and hash partitioning given a suitable hash function choice, the cost of computing a range partition function is higher than the cost to transfer the tuple, especially when the number of partitions increases beyond the TLB capacity. The naïve implementation uses binary search on a sorted array of delimiters that define each range, incurring a logarithmic number of cache accesses. We introduce a specialized SIMD-based cache-resident index that speeds up range functions up to 6 times, making range partitioning a practical choice for many applications.

All partitioning variants discussed in this chapter are shown in Figure 3.1, where we also mark our contributions. We apply all variants to design and implement three large-scale NUMA-aware sorting algorithms. We use sorting, rather than joins or aggregations for two reasons. First, because it is a wider problem that can be a sub-problem for both join and aggregation. Second, we can apply *all* partitioning variants to build *unique* sorting algorithms, such that each is more scalable in distinct cases depending on input size, key domain size, space requirements, and skew efficiency.

The first sorting algorithm we propose is stable least-significant-bit (LSB) radixsort. The algorithm is based on non-in-place out-of-cache radix partitioning where we add two innovations. We use hybrid range-radix partitioning to guarantee load balancing among threads, and ensure each tuple crosses NUMA boundaries at most once, even if the default algorithm would re-shuffle the whole array in each partitioning pass.

The second sorting algorithm we propose is an in-place most-significant-bit (MSB) radixsort. The algorithm utilizes all variants of in-place partitioning that we introduce, one for each distinctive level in the memory hierarchy: shared out-of-cache in-place partitioning, shared-nothing out-of-cache partitioning, and in-cache partitioning. We reuse the range-radix idea to minimize transfers across NUMA boundaries and guarantee load balancing.

The third sorting algorithm we propose is a comparison-sort that uses the newly optimized range partitioning. We perform very few out-of-cache range partitioning passes with a very wide fanout until we reach the cache, providing NUMA optimality. In the cache, we

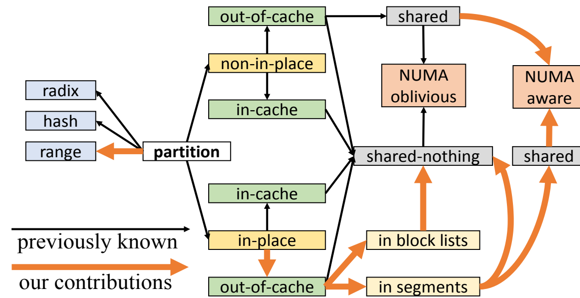



Figure 3.1: Partitioning variants and contributions

employ SIMD-assisted sorting [Inoue *et al.*, 2007], modified to use the cache more effectively.

We use fixed length integer keys and payloads, typical of analytical database applications. We evaluate on both dense and sparse key domains. If order-preserving compression is used [Raman *et al.*, 2013; Willhalm *et al.*, 2009], any sparse or dense domain with fixed or variable length data is compacted into a dense integer domain.

Skewed workload distribution can reduce parallelism in some naïve approaches. In our context, when we statically distribute partitions to threads, we ensure that the partitions are as balanced as possible. Specifically, we never use radix partitioning to any range of bits to divide the workload. Instead, we combine range with radix partitioning to guarantee that, if specific bit ranges have very few distinct values, we can find delimiters that split the workload equally among threads, independently of the key value range.

In this chapter, we make the following contributions:

- We introduce several new variants for main-memory partitioning, most notably large-scale in-place partitioning and efficient range partitioning, and also guarantee minimal NUMA transfers across multiple CPUs.
- We combine partitioning variants to design three sorting algorithms, all the fastest of their class for billion-scale inputs, and evaluate the best options depending on input size, key domain, available space, and skew.

The rest of this chapter is organized as follows. In Section 3.2, we describe all partitioning variants, starting from the top level of the memory hierarchy. In Section 3.3, we discuss the three sorting algorithms. In Section 3.4, we present our experimental evaluation and we conclude in Section 3.5.

## 3.2 Partitioning Variants

### 3.2.1 Partitioning In-Cache

We start by considering the versions that best operate when the table fits in the cache. Each partition is generated in a single contiguous segment. The non-in-place version in Algorithm 12 uses a separate array from the input to store the output. The simplest non-in-place version does only two random accesses per item. When operating in the cache, we need the output and the offset array to be cache-resident.

A slightly more complicated version of the algorithm allows the partitioning to occur in-place, by swapping tuples across locations instead of copying out to a separate output array. This variant is shown in Algorithm 12. In the in-place version, we start by loading a tuple, find the correct partition and the output location through the partition offset. Then, we swap the loaded tuple with the item stored at the target location and continue with the next tuple we just loaded. The loop stops when the cycle of swaps returns to the original location. Each tuple is moved exactly once between locations of the same table and the process stops when every tuple has been moved. In the simple version of this algorithm [Albutiu *et al.*, 2012], each cycle swaps tuples until we return to the partition encompassing the initial location, incurring $O(|T|/\mathcal{P})$ branch mispredictions for shuffling $|T|$ tuples across $\mathcal{P}$ partitions. In Algorithm 13, the tuples are written in reverse order and each swap cycle shuffles all tuples from at least one partition, incurring only $O(\mathcal{P})$ branch mispredictions.

**Algorithm 12:** Non-in-place in-cache partitioning

```
1  i, o, p ← 0
2  do
3  |  O[p] ← o        // store the offset to the start of each partition in O (formally O is the prefix sum of H)
4  |  o ← o + H[p]    // the starting offset of the next partition is the ending offset of the current partition
5  |  p ← p + 1                                                        // iterate over the 𝒫 partitions
6  while p ≠ 𝒫
7  do
8  |  t ← T[i]                                                         // load tuple from input
9  |  i ← i + 1
10 |  p ← f(t)                  // compute the output partition using the partition function f
11 |  o ← O[p]                                                         // load the output location of tuple
12 |  O[p] ← o + 1                                       // store back incremented offset of partition
13 |  T'[o] ← t                                                        // store tuple to output
14 while i ≠ |T|
```

**Algorithm 13:** In-place in-cache partitioning

1. $o_1, o_2, p_1, p_2 \leftarrow 0$
2. **do**
3.     $o \leftarrow o + H[p_1]$ // compute and store the offset to the end of each partition in $O$
4.     $O[p_1] \leftarrow o$
5.     $p_1 \leftarrow p_1 + 1$
6. **while** $p_1 \neq \mathcal{P}$
7. **while** $H[p_2] = 0$ **do**
8.     $p_2 \leftarrow p_2 + 1$ // skip initial empty partitions
9. **do**
10.     $t \leftarrow T[o_2]$ // load tuple to initiate cycle of tuple swaps
11.     **do**
12.         $p_1 \leftarrow f(t)$ // find the output partition using the partition function $f$
13.         $o_1 \leftarrow O[p_1] - 1$ // find the output location of the tuple
14.         $O[p_1] \leftarrow o_1$ // store back decremented offset of partition
15.         $T[o_1] \longleftrightarrow t$ // swap tuple in registers with tuple in table
16.     **while** $o_1 \neq o_2$
17.     **do**
18.         $o_2 \leftarrow o_2 + H[p_2]$ // go over partitions with all tuples already shuffled
19.         $p_2 \leftarrow p_2 + 1$
20.     **while** $p_2 \neq \mathcal{P}$ **and** $o_2 = O[p_2]$
21. **while** $p_2 \neq \mathcal{P}$

### 3.2.2 Partitioning Out-of-Cache

As mentioned in Section 3.1, out-of-cache performance is throttled by increased cache conflicts [Satish *et al.*, 2010] and cache pollution with output tuples [Wassenberg and Sanders, 2011]. Also, TLB thrashing occurs when the number of partitions exceeds the capacity of the TLB [Manegold *et al.*, 2000b], unless the entire dataset can be placed in equally few large operating system pages in order to be TLB resident.

#### 3.2.2.1 Non-in-place Partitioning

To mitigate these problems, we can use the cache as an intermediate buffer and write back to memory in groups of tuples [Satish *et al.*, 2010]. Writes can bypass the higher cache levels via non-temporal stores [Wassenberg and Sanders, 2011]. Non-temporal stores are the only mechanism to avoid cache pollution with data we do not need to re-access soon.

Buffering data for each partition reduces the working set size and eliminates the TLB problem when operating in the buffer. TLB misses still occur, but $1/\mathcal{B}$ of the time, if $\mathcal{B}$ is the number of tuples buffered for each partition before writing to output. If the buffer

for each partition is exactly as big as a cache line, writing the full cache line to memory is accelerated by write-combining and avoids polluting the higher cache levels with output data. The partitioning fanout is now bounded by the number of cache lines in the fast core-private cache, rather than the TLB entries. To maximize the cache use, we use the last buffer slot to save the output offset and access one cache line per iteration. Algorithm 14 shows this method. The $\mathcal{P}$ buffers with $\mathcal{B}$ slots each are contiguous in the $B$ array. The last loop flushes any remaining tuples in the buffers to the output. We need to rewrite the last group of tuples of each partition in case it was overwritten by the of the following partition.

**Algorithm 14:** Non-in-place out-of-cache partitioning

```
1  i, o, p, p' ← 0
2  do
3  │  B[(p₁ + 1) · 𝓑 − 1] ← o            // store output offset of partition in last buffer slot
4  │  o ← o + H[p₁]
5  │  p₁ ← p₁ + 1
6  while p₁ ≠ 𝓟
7  do
8  │  t ← T[i]                            // load tuple from input
9  │  i ← i + 1
10 │  p ← f(t)                            // find the output partition using partition function f
11 │  o ← B[(p + 1) · 𝓑 − 1]              // load output offset from last buffer slot
12 │  B[(p + 1) · 𝓑 − 1] ← o + 1          // store back the incremented output offset
13 │  b ← o mod 𝓑                         // 𝓑 is a power of 2 and we can use masking instead of modulo
14 │  B[p · 𝓑 + b] ← T[i]                 // copy tuple from input table to buffer
15 │  if b = 𝓑 − 1 then
16 │  │  b ← 0
17 │  │  do
18 │  │  │  T'[o + b − 𝓑] ← B[p · 𝓑 + b]  // (non-temporal) copy tuple from buffer to output
19 │  │  │  b ← b + 1
20 │  │  while b ≠ 𝓑
21 │  └  B[(p + 1) · 𝓑 − 1] ← o + 1       // store output offset in last buffer slot
22 while i ≠ |T|
23 do
24 │  o ← B[(p' + 1) · 𝓑 − 1]             // load output offset of partition from last slot of buffer
25 │  b₁ ← 0                              // 1st valid tuple in the buffer
26 │  b₂ ← o mod 𝓑                        // last valid tuple in the buffer
27 │  if b₂ > H[p'] then
28 │  └  b₁ ← b₂ − H[p']                  // in case the partition has very few tuples
29 │  while b₁ ≠ b₂ do
30 │  │  T'[o − (b₂ − b₁)] ← B[p' · 𝓑 + b₁]  // copy tuple from buffer to output
31 │  └  b₁ ← b₁ + 1
32 │  p' ← p' + 1
33 while p' ≠ 𝓟
```

If we have multiple columns stored in separate arrays, we use one cache line per column in the buffer for each partition. We can use one cache line per column flushed separately. We can also pack multiple columns as a single tuple in order to reduce the number of cache accesses, and de-interleave to separate columns before we flush to memory. For instance, to partition 32-bit keys and 32-bit payloads, we can store each tuple as a 64-bit pair in the buffer. Each time the buffer is flushed, we can use SIMD instructions to de-interleave.

To execute the non-in-place out-of-cache partitioning variant in parallel given $\mathcal{T}$ threads, we can split the input among $\mathcal{T}$ equal pieces and process each piece in one thread. Each thread computes the histogram of a different portion of the input and, by computing the prefix sum of the interleaved histograms, we can ensure that each partition is split to $\mathcal{T}$ partition with no memory overlaps between different threads. We need to synchronize the $\mathcal{T}$ threads twice, once to wait for the histograms to be generated before we interleave them, and once before we flush the buffers to memory. Without the latter synchronization, race conditions can occur at the cache lines spanning across multiple partition boundaries.

**Algorithm 15:** In-place out-of-cache partitioning

1. $\mathcal{O} \leftarrow (|T| + \mathcal{B}) - ((|T| + \mathcal{B}) \mod \mathcal{B})$ // any multiple of $\mathcal{B}$ that is larger than $|T|$
2. $o, p \leftarrow 0$ // loop over the $\mathcal{P}$ partitions to setup the buffers
3. **do**
4. $\quad b_1 \leftarrow 0$
5. $\quad$ **if** $H[p] + (o \mod \mathcal{B}) > \mathcal{B}$ **then**
6. $\quad\quad b_2 \leftarrow (o + H[p] - 1) - (o + H[p] - 1) \mod \mathcal{B})$
7. $\quad\quad$ **do**
8. $\quad\quad\quad B[p \cdot \mathcal{B} + b_1] \leftarrow T[b_1 + b_2]$ // buffer tuple from cache line at the end of partition
9. $\quad\quad\quad b_1 \leftarrow b_1 + 1$
10. $\quad\quad$ **while** $b_1 \neq \mathcal{B}$
11. $\quad\quad B_0[p] \leftarrow B[p \cdot \mathcal{B}]$ // store tuple in 1st slot of buffer
12. $\quad\quad B[p \cdot \mathcal{B}] \leftarrow o + H[p]$ // overwrite the 1st slot of the buffer with the output offset
13. $\quad$ **else**
14. $\quad\quad$ **do**
15. $\quad\quad\quad B[p \cdot \mathcal{B} + b_1] \leftarrow T[b_1 + o]$ // buffer tuple from (only) cache line at the end of partition
16. $\quad\quad\quad b_1 \leftarrow b_1 + 1$
17. $\quad\quad$ **while** $b_1 \neq H[p]$
18. $\quad\quad B_0[p] \leftarrow B[p \cdot \mathcal{B}]$ // store tuple in 1st slot of buffer
19. $\quad\quad B[p \cdot \mathcal{B}] \leftarrow b_1 + \mathcal{O}$ // output offset of last cache line stored using $\mathcal{O}$
20. $\quad o \leftarrow o + H[p]$
21. $\quad O[p] \leftarrow o$ // store ending offset of each partition in $O$
22. $\quad p \leftarrow p + 1$
23. **while** $p \neq \mathcal{P}$

**Algorithm 15:** In-place out-of-cache partitioning (continued)

```
24 p₁, p₂, o₂ ← 0
25 while H[p₂] = 0 do
26 │ p₂ ← p₂ + 1                                    // skip initial empty partitions
27 t ← T[0]                       // load 1st tuple of 1st partition to start swap cycle
28 loop
29 │ do
30 │ │ p₁ ← f(t)         // compute the output partition using the partition function f
31 │ │ o₁ ← B[p₁ · B] − 1                 // load and decrement the output offset
32 │ │ B[p₁ · B] ← o₁                     // store back the decremented output offset
33 │ │ b ← o₁ mod B
34 │ │ B[p₁ · B + b] ⟷ t           // swap tuple from registers with tuple from buffer
35 │ while b ≠ 0
36 │ if o₁ ≠ O then
37 │ │ do
38 │ │ │ t ← T[o₁ + b − B]                      // load tuple from following cache line
39 │ │ │ B[p₁ · B + b] ⟷ t    // swap with tuple from following cache line in the buffer
40 │ │ │ T[o₁ + b] ← t          // (non-temporal) store tuple from buffer to output
41 │ │ │ b ← b + 1
42 │ │ while b ≠ B
43 │ │ t ← B₀[p₁]     // load tuple in 1st slot from last cache line to restart swap cycle
44 │ │ b₂ ← o₁ − O[p₁]               // compute remaining tuples for current partition
45 │ │ if b₂ > B then
46 │ │ │ B₀[p₁] ← B[p₁ · B]  // save tuple in 1st slot of current (buffered) cache line to reuse the slot
47 │ │ │ B[p₁ · B] ← o₁          // overwrite the 1st slot of buffer with the output index
48 │ │ else
49 │ │ │ b₁ ← 0
50 │ │ │ while b₁ ≠ b₂ do
51 │ │ │ │ B[p₁ · B + b₁] ← B[(p₁ + 1) · B + b₁ − b₂]  // move tuples to the front of the buffer
52 │ │ │ │ b₁ ← b₁ + 1
53 │ │ │ B₀[p₁] ← B[p₁ · B]  // save tuple in 1st slot of current (buffered) cache line to reuse the slot
54 │ │ │ B[p₁ · B] ← b₂ + O       // output offset larger than O denotes the last cache line
55 │ else
56 │ │ t ← B₀[p₁]            // restore tuple from 1st slot in case swap cycle is not closed
57 │ │ B₀[p₁] ← B[p₁ · B]            // save tuple in 1st slot of buffer to reuse the slot
58 │ │ B[p₁ · B] ← O     // overwrite 1st slot of buffer with O to denote finished partition
59 │ │ if p₁ = p₂ then
60 │ │ │ do
61 │ │ │ │ o₂ ← o₂ + H[p₂]         // go over partitions with all tuples already shuffled
62 │ │ │ │ p₂ ← p₂ + 1
63 │ │ │ │ if p₂ = P then
64 │ │ │ │ │ break loop     // exit (outer) loop if we have processed all partitions
65 │ │ │ while B[p₂ · B] = O
66 │ │ │ t ← T[o₂]                // load 1st tuple of partition to restart swap cycle
```

**Algorithm 15:** In-place out-of-cache partitioning (continued)

```
67 o, p ← 0                                      // flush the buffers for all 𝒫 partitions
68 do
69     b ← min(ℬ − (o mod ℬ), H[p])
70     B[p · ℬ] ← B_0[p]                          // restore tuple in 1st slot of buffer
71     while b ≠ 0 do
72         T′[o + b] ← B[p · ℬ + b]               // copy tuple from buffer to output
73         b ← b − 1
74     o ← o + H[p]
75     p ← p + 1
76 while p ≠ 𝒫
```

#### 3.2.2.2 In-place Partitioning, Contiguous Segments, Shared-Nothing

Adapting the out-of-cache buffering technique to in-place partitioning requires a more complicated approach. The basic idea is to perform the swaps between the buffers, so that the sparse RAM locations are accessed only $1/\mathcal{B}$ of the time, reducing the overhead from TLB misses. Compared with non-in-place out-of-cache partitioning, which uses the buffer as an intermediate layer to group tuples before writing them, in-place out-of-cache partitioning performs all tuples swaps in the buffer, and accesses RAM at cache line units. Effectively the buffers map the current cache line per partition where the tuple swaps are performed.

Before the main loop, we load the cache lines in the starting partition locations to the buffers. We use the first slot of each buffer to store the output offset, similarly to the non-in-place variant where we used the last slot. The offsets are stored inside the buffer and the last $\mathcal{B}$ items of each partition are handled differently to eliminate branching in the inner loop. Item swaps between partitions occur using the last $\mathcal{B}$ tuples that are stored in the buffer. When a buffer has swapped all $\mathcal{B}$ tuples, the cache line is flushed to the location where it was originally loaded from and the buffer is refilled with the next $\mathcal{B}$ tuples of the same partition. Thus, all operations occur in the buffers $(\mathcal{B}-1)/\mathcal{B}$ of the time without any cache or misses. The entire process is shown in Algorithm 15.

Given $\mathcal{T}$ threads and $\mathcal{P}$ partitions, if having $\mathcal{T} \times \mathcal{P}$ contiguous segments is acceptable, then we can run in-place partitioning in parallel in a shared-nothing fashion. However, unlike the non-in-place variant, generating $\mathcal{P}$ contiguous segments across using multiple threads is impossible unless we use some coarse grain synchronization mechanism.

#### 3.2.2.3 In-place Partitioning, List of Blocks

The requirement of generating all partitions in $\mathcal{P}$ contiguous segments can be relaxed. Instead of storing the tuples of each partition sequentially to the output, we can generate large blocks that contain tuples from a single partition. While shuffling, we fill $\mathcal{P}$ blocks at a time. Once a block is full, we allocate a new block for the partition. The block size must be large enough to amortize the cost of non-sequential writes acros blocks, but not be too large to avoid external fragmentation from non-full blocks.

To access the tuples from a single partition, we generate linked lists to connect all the blocks that contain tuples of the same partition. While the access is not entirely sequential compared to the single segment case, the non-sequential accesses incurred by the list hops after scanning an entire block sequentially, are amortized by using sufficiently large blocks.

This method can be extended to work in-place, if we remove $\mathcal{P}$ blocks of tuples from the start of the input table and save them in private space. We start the partitioning of the input from the $(\mathcal{P} \times \mathcal{B})^{\text{th}}$ tuple, assuming we have $\mathcal{B}$ tuples per block. By the time any partition is filled, the input pointer will have advanced enough for the output to safely use the space of input we have already processed, without overwriting tuples we have not yet accessed. After we process the remaining input, we also process the tuples that we initially copied out. For each partition, only the last block of the block list can be non-full. Thus, unused space has an upper bound of $\mathcal{P} \times \mathcal{B}$ and is negligible compared to size of the input.

Storing the partitioning output in linked lists of blocks has a number of strong properties. First, it uses the fast non-in-place out-of-cache partitioning. Second, it does not require the pre-computation of a histogram. Third, it works in-place by ensuring no overlap between input and output data. Finally, thread parallelism is straight-forward as we only need to connect the $\mathcal{T}$ linked lists of blocks per partition generated among the $\mathcal{T}$ threads.

#### 3.2.2.4 In-place Partitioning, Contiguous Segments, Shared

In order to partition and shuffle data in parallel inside the same segment, we need fine-grain synchronization. Since using OS latches are overly expensive, we use atomic instructions. Atomic `fetch_and_add` reads a memory location, adds some value to it, and stores it back atomically, while returning its previous value. Imagine an array of items and multiple

threads where each item must be processed by exactly one thread. Each thread can safely use item at index $i$, returned by invoking `fetch_and_add`$(c, 1)$ on a shared counter $c$. Once finished, the thread asks for the next item or terminates if $i$ exceeds the number of items. We apply the same idea to in-place partitioning using one shared counter for each partition to represent the number of tuples swapped so far.

We use atomic `fetch_and_add` instructions on the shared counter of partition $p$ to "lock" the cell of the next yet unread item of partition $p$. First, we store the location of the tuple that initiates the swap cycle. After swapping an arbitrary number of tuples, at some point we will return to the original partition $p$. Then, we store the last tuple back to the initial location, closing the swap cycle. Only the $\mathcal{P}$ counters are shared across threads.

As mentioned in Section 3.2.1, we define a *swap cycle* as a sequence of swaps that starts by reading a key from a specific partition and after a number of swaps, returns to the same partition to write a key in the initially read location. We cannot know in advance how large a swap cycle will be, thus we cannot lock all locations the cycle will go through before actually moving tuples. Imagine a scenario where the first partition has only one item found in the last cell of the array. Then, one thread would perform a single swap cycle covering all items before the last cell is reached and the cycle is closed.

To solve this problem, threads lock only one location at a time for a single swap. However, when the process is close to completion, multiple threads may compete for the same swap cycles causing deadlocks. For example, assume we have one tuple per partition. If thread $t_1$ reads tuple $k_x$ (must go to location $l_x$) from location $l_y$ (must bring $k_y$ here) and $t_2$ reads $k_z$ (must go to $l_z$) from $l_x$ (must bring $k_x$ here), thread $t_1$ will find no space for $k_x$, because the offset of partition $X$ was incremented by $t_2$ when it read $k_z$. If $t_1$ waits, $t_2$ eventually reaches $k_y$. Then, a deadlock occurs, since $t_1$ holds ($k_x$, $l_y$) and $t_2$ holds ($k_y$, $l_x$).

To solve the deadlock problem, threads never wait for any other threads. When a thread reaches a full partition while swapping tuples, it saves both the last tuple of the swap cycle and the initial (locked) location of the swap cycle. In the previous example, $t_1$ records ($k_x$, $l_y$) and $t_2$ records ($k_y$, $l_x$). Algorithm 16 describes the overall process. We omit the final step of shuffling the tuples that were recorded to avoid deadlocks. The step takes negligible time as the number of recorded tuples is bounded by $\mathcal{P} \times \mathcal{T}$ for $\mathcal{P}$ partitions and $\mathcal{T}$ threads.

So far, we presented a way for multiple threads to partition items in-place concurrently, but this solution is impractical if used as is. First, we make no use of buffering to improve out-of-cache performance. Second, for each key we move to its destination, we update a shared variable triggering cache invalidations on every tuple move. To make it practical, we change the unit of transfer from tuples to blocks. Each block must have a fixed size and all tuples must belong to the same partition. We generate such blocks using the technique described in the previous section. Out-of-cache accesses are amortized by the block size, as is the synchronization cost of accessing shared variables.

**Algorithm 16:** Synchronized in-place partitioning

```
1  o, p ← 0
2  A ← {}                                    // set of yet active (unfinished) partitions
3  do
4      A ← A ∪ {p}
5      O[p] ← o                              // O the prefix sum of the histogram H (same for all threads)
6      o ← o + H[p]
7      p ← p + 1
8  while p ≠ 𝒫
9  D ← {}                                    // set of tuple and location pairs that would deadlock
10 loop
11     p1 ← any ∈ A
12     o1 ← atomic_fetch_and_add(C[p1], 1)   // atomically increment shared counter to lock swap location
13     if o1 < H[p1] then
14         o2 ← o1 + O[p]                    // compute (initial) location using offset within partition
15         t ← T[o2]                         // load tuple from input to start swap cycle
16         p2 ← f(t)                         // compute output partition of 1st tuple
17         if p1 ≠ p2 then
18             do
19                 o1 ← atomic_fetch_and_add(C[p1], 1)   // lock swap location
20                 if o1 ≥ H[p2] then
21                     D ← D ∪ {t, o2}       // record the tuple and initial location to avoid deadlocks
22                     continue loop         // break out of the inner loop and continue the outer (main) loop
23                 o1 ← o1 + O[p2]           // compute location using offset within partition
24                 T[i] ⟷ t                  // swap tuple
25                 p2 ← f(t)                 // compute output partition of next tuple
26             while p1 ≠ p2
27             T[o1] ← t
28     else
29         A ← A \ {p}                       // already shuffled all tuples in current partition
30         if A = {} then
31             break loop                    // exit if all partitions are shuffled
```

### 3.2.3 Partitioning Across NUMA Regions

Moving RAM-resident data across multiple CPUs raises questions about the effectiveness of NUMA RAM transfers. Accessing remote memory locations goes through an interconnection channel that issues operations to remote RAM modules, increasing the latency. Normally, random accesses are much slower than sequential access and the gap increases when the accesses reference remote RAM regions and go through the CPU interconnection. Prior work [Albutiu *et al.*, 2012] proposed doing sequential accesses to remote memory, since hardware pre-fetching hides the latency. To avoid imbalanced use of the NUMA layer when all transfers are directed to a subset of CPUs, we can pre-schedule the transfers and supervise them via synchronization to ensure load balancing [Li *et al.*, 2013].

One way to make NUMA-oblivious code scale on multiple CPUs is to allocate both arrays to be physically interleaved across all RAM regions. The OS can support interleaved allocation, where the physical locations of a single array are interleaved across all NUMA regions. Randomization of page placement balances accesses across the NUMA interconnection, but precludes NUMA locality. Thus, if we do random accesses, we pay the extra NUMA latency. Cache-line buffering, used by out-of-cache partitioning to avoid TLB misses and facilitate write-combining, also mitigates the NUMA overhead. Still, we measured out-of-cache partitioning to be up to 55% slower on four NUMA regions on interleaved space. The overhead for accessing single tuples across NUMA regions is even higher.

A more NUMA-friendly allocation is to split space into large segments bound to a specific region. We can have one segment per thread bound to its local NUMA region, or one segment per NUMA region. We use the latter approach for sorting (see Section 3.3.1).

#### 3.2.3.1 Non-in-place Partitioning

Using NUMA-bound segmented allocation for threads or CPUs and if extra space is allowed, we can ensure that all tuples will cross the NUMA boundaries at most once. We use shared-nothing partitioning locally and then use a separate step to shuffle across CPUs. We can use the NUMA interconnection in a balanced way without manual schedules [Li *et al.*, 2013]. We distribute each segment across all threads of the destination CPU, and do the transfers in a per thread random order. Given $\mathcal{N}$ NUMA regions, the expected number of times a

tuple will be transferred across NUMA regions is $\frac{\mathcal{N}-1}{\mathcal{N}}$. This number is less than 1 because $\frac{1}{\mathcal{N}}$ of tuples that are already on the right NUMA region never cross NUMA boundaries.

The NUMA-oblivious partitioning may perform faster than the 2-step method of NUMA-local shared-nothing partitioning followed by shuffling across NUMA nodes, since out-of-cache partitioning mitigates the latency cost by buffering. Depending on the number partitioning passes, the number of NUMA regions, and the hardware, enforcing minimal transfers across NUMA regions may not outperform NUMA-oblivious approaches.

#### 3.2.3.2 In-place Partitioning

Assuming NUMA-bound segmented allocation, the only in-place variant where threads do not work in a shared-nothing fashion is during block shuffling. During the phase of block shuffling on multiple NUMA regions, threads can read and write blocks from all regions, but all accesses are sequential, since the block is large enough to amortize the random access penalty. In the worst case, the operating thread CPU, the source, and the destination of the swapped block, will be on three different regions. Thus, the tuples can cross the interconnection at most twice. Since collocations occur, the expected number of transfers is $\frac{2\mathcal{N}^2-3\mathcal{N}+1}{\mathcal{N}^2}$ for $N$ NUMA regions, assuming $\mathcal{N} > 2$. For example, with $\mathcal{N} = 4$, we perform 1.3125 transfers per tuple on average, 75% more than the non-in-place variant.

### 3.2.4 Radix / Hash Histogram

Hash and radix partitioning are similar because the time required to compute the partition function is trivial. Complicated hash functions are used to decrease the collisions in hash tables. In our context, we need partitions of almost equal size that separate the keys randomly. Thus, employing hash functions designed to minimize hash table collisions, would waste CPU cycles without offering any advantage.

The radix function is a shift operation followed by a logical `and`. To isolate bit range $[x, y)$, we shift right by $x$, then mask with $2^{y-x}-1$. Using multiplicative hashing and $\mathcal{P} = 2^n$ partitions, we multiply with an odd factor and then shift by $b - n$ for $b$-bit keys. In both cases, computing the partition function only marginally affects performance.

### 3.2.5 Range Histogram

For hash and radix partitioning, computing the destination partition comes almost at zero cost, whereas range partitioning is more expensive [Wu *et al.*, 2013]. To split into $\mathcal{P}$ ranges, we need $\mathcal{P}$-1 delimiters that define the lower and upper bounds of each partition. We sort the $\mathcal{P}$-1 delimiters and do a binary search for each input key. The difference from textbook binary search is that we search ranges rather than keys. Thus, we omit equality comparisons and do not exit the loop in less than $\log \mathcal{P}$ steps even if an exact match is found earlier.

Scalar binary search is one example where being cache-resident is not fast enough to give good performance. The array of $\mathcal{P} - 1$ delimiters always remains resident in the L1 cache. However, the delays from waiting to load a new delimiter to proceed to the next comparison and the logarithmic number of loads that have to be performed, slow down range function computation almost by an order of magnitude compared to hash and radix. Replacing branches with conditional moves performs even worse, proving that the problem here is not the branches, but the logarithmic number of cache accesses that are tightly coupled due to data (or conditional control flow) dependencies and thus incur their full latency cost.

#### 3.2.5.1 Range Partitioning Function, In SIMD Registers

We first show how the computation of range partitioning functions can be accelerated via SIMD instructions, by doing small fanout range partitioning. Each SIMD register can hold multiple values and perform comparisons with all the values at the same time.

In the first approach, we holds different delimiters in each SIMD register and we broadcast each input key to all lanes of another SIMD register. We can do a SIMD comparison, convert the result to a bit-mask and search the least-significant-set-bit to see which delimiter is actually larger than the key. To extend the fanout, we can store $\mathcal{W} \times \mathcal{R}$ delimiters in $\mathcal{R} \times \mathcal{W}$-wide SIMD registers. We perform $R$ SIMD comparisons and $\mathcal{R} - 1$ SIMD packs, then extract and append the bitmasks. The fanout depends on the number of available registers. In practice, we do not have that many registers before stack spilling occurs and, unlike binary search that requires $O(\log \mathcal{P})$ time, this *horizontal* approach requires $O(\mathcal{P}/\mathcal{W})$ time. An example of 17-way range partitioning is shown below. The 16 range delimiters (32-bit) are stored in 4 SIMD registers (128-bit): `del_ABCD`, `del_EFGH`, `del_IJKL`, and `del_MNOP`.

```
__m128i key = _mm_shuffle_epi32(_mm_cvt32_si128(keys[i++]), 0);  // broadcast next 1 key
__m128i cmp_ABCD = _mm_cmpgt_epi32(key, del_ABCD);  // compare with 16 delimiters
__m128i cmp_EFGH = _mm_cmpgt_epi32(key, del_EFGH);
__m128i cmp_IJKL = _mm_cmpgt_epi32(key, del_IJKL);
__m128i cmp_MNOP = _mm_cmpgt_epi32(key, del_MNOP);
__m128i cmp_A_to_H = _mm_packs_epi32(cmp_ABCD, cmp_EFGH);  // pack to 1 SIMD register
__m128i cmp_I_to_P = _mm_packs_epi32(cmp_IJKL, cmp_MNOP);
__m128i cmp_A_to_P = _mm_packs_epi16(cmp_A_to_H, cmp_I_to_P);
int bitmask = _mm_movemask_epi8(cmp_A_to_P);  // extract the bitmask of comparisons
int res = _bit_scan_forward(bitmask | 0x10000);  // find the lowest set bit from bitmask
```

The transposed approach is broadcast each delimiter in a separate SIMD register and compare against $\mathcal{W}$ keys from the input. We use the results from earlier comparisons to blend delimiters into new custom delimiters. For instance, suppose we want to do 4-way range partitioning using 3 delimiters $d_1 < d_2 < d_3$. We first compare the keys with (the SIMD register with broadcast) $d_2$, then blend $d_1$ with $d_3$ to create a new delimiter $d_{1|3}$. Each lane will either have $d_1$ or $d_3$ based on the comparison of the input key with $d_2$. Then, we compare the keys with $d_{1|3}$ and combine the results. This *vertical* approach can be seen as a binary tree structure of depth $\log \mathcal{P}$ with $\mathcal{P} - 1$ delimiters. On each comparison, the first half nodes of each level are blended with the other half, creating a new tree of depth $\log \mathcal{P} - 1$. The result is generated by bit-interleaving the $D$ comparisons into ranges $\in [0, \mathcal{P})$. We show an 8-way range function for 4 32-bit keys using 7 delimiters `del_1`, ..., `del_7`.

```
__m128i key = _mm_load_si128((__m128i*) &keys[i]);  i += 4;  // load next 4 32-bit keys
__m128i cmp_lvl_1 = _mm_cmpgt_epi32(key, del_4);  // 1st level comparison
__m128i del_15 = _mm_blendv_epi8(del_1, del_5, cmp_lvl_1);
__m128i del_26 = _mm_blendv_epi8(del_2, del_6, cmp_lvl_1);
__m128i del_37 = _mm_blendv_epi8(del_3, del_7, cmp_lvl_1);
__m128i cmp_lvl_2 = _mm_cmpgt_epi32(key, del_26);  // 2nd level comparison
__m128i del_1357 = _mm_blendv_epi8(del_15, del_37, cmp_lvl_2);
__m128i cmp_lvl_3 = _mm_cmpgt_epi32(key, del_1357);  // 3rd level comparison
__m128i res = _mm_sub_epi32(_mm_setzero_si128(), cmp_lvl_1);  // combine the results
res = _mm_sub_epi32(_mm_add_epi32(res, res), cmp_lvl_2);
res = _mm_sub_epi32(_mm_add_epi32(res, res), cmp_lvl_3);
```

The two variants are largely equivalent. The vertical approach requires $\mathcal{P} - 1$ SIMD registers to hold the delimiters and requires $O(\mathcal{P})$ instructions for $\mathcal{W}$ keys. The horizontal approach requires $\mathcal{P}/\mathcal{W}$ registers to hold the delimiters and executes $O(\mathcal{P}/\mathcal{W})$ instructions per key. We use both variants in the following section. We also use the vertical version to combine range with radix partitioning functions (see Section 3.3.2.1).

#### 3.2.5.2 Range Partitioning Function, In-Cache Index

Register-resident range partitioning is very fast, but the number of partitions we generate is not enough to saturate the partitioning fanout. [Kim *et al.*, 2010] studied fast tree-index search using SIMD. To search each node, [Kim *et al.*, 2010] performs a horizontal search with one SIMD register. The node fanout increases from 2 to $\mathcal{W}$ making the tree shorter, while node accesses are implemented with $O(1)$ SIMD instructions rather than $O(\mathcal{W})$ scalar instructions. The scalar binary tree search performs very poorly in comparison.

We extend the idea of larger fanout to optimize range partitioning functions. A cache-resident index tree does not need pointers thus we can remove them; each level of the tree is a single sorted array. For example, a tree with 24 delimiters can be represented in 2 levels. The first level holds delimiters $[d_5, d_{10}, d_{15}, d_{20}]$ and the 2$^{\text{nd}}$ level holds delimiters $[d_1, d_2, d_3, d_4]$, $[d_6, d_7, d_8, d_9]$, $[d_{11}, d_{12}, d_{13}, d_{14}]$, $[d_{16}, d_{17}, d_{18}, d_{19}]$, and $[d_{21}, d_{22}, d_{23}, d_{24}]$. The node distinction implied by the parentheses is not explicit, we simply accessing the array at a different offset denotes a different node. Delimiters are never repeated across levels.

The array pointers representing the tree levels are kept in scalar registers and all tree node accesses are "hard-coded" with ad-hoc code per tree level. Thus, we can design trees with different fanout on each level based on the total number of range partitions. We generate a menu of tree fanout configurations in order to be able to choose the best partitioning fanout, similar to picking the number of bits for radix or hash partitioning. The sensible configurations are the ones where we load and horizontally search $k$ SIMD registers, giving tree node fanout of the form $k \cdot \mathcal{W} + 1$. We can hold the zero level (root) of the tree in registers during the operation using horizontal register search, but we found that using vertical SIMD search to access the root is faster.

Assuming 32-bit keys and 128-bit SIMD registers ($\mathcal{W} = 4$), we use 5-way or 9-way fanouts on tree levels. A level with 17-way fanout is outperformed by two consecutive levels with 5-way fanout, also increasing the fanout from 17 to 25. The best configurations that work well with suitable out-of-cache partitioning fanouts are: (i) a 360-way range partitioning function using a 3-level $(8 \times 5 \times 9)$-way tree search, (ii) a 1000-way function using a 4-level $(8 \times 5 \times 5 \times 5)$-way tree, and (iii) a 1800-way function using an $(8 \times 5 \times 5 \times 9)$-way tree. We show the code accessing the 360-way tree omitting the 8-way root.

```
__m128i lvl_1 = _mm_load_si128((__m128i*) &index[res_0 << 2]);  // access 1st level (5-way fanout)
__m128i cmp_1 = _mm_cmpgt_epi32(lvl_1, key);
int res_1 = _mm_movemask_ps(_mm_castsi128_ps(cmp_1));
res_1 = _bit_scan_forward(res_1 ^ 0x1FF);
res_1 += res_0 + (res_0 << 2);
// access 2nd level (9-way fanout) with an offset of 32 (the 1st level delimiters)
__m128i lvl_2_A = _mm_load_si128((__m128i*) &index[(res_1 << 3) + 32]);
__m128i lvl_2_B = _mm_load_si128((__m128i*) &index[(res_1 << 3) + 36]);
__m128i cmp_2_A = _mm_cmpgt_epi32(lvl_2_A, key);
__m128i cmp_2_B = _mm_cmpgt_epi32(lvl_2_B, key);
__m128i cmp_2 = _mm_packs_epi32(cmp_2_A, cmp_2_B);
__m128i cmp_2 = _mm_packs_epi16(cmp_2, _mm_setzero_si128());
int res_2 = _mm_movemask_epi8(cmp_2);
res_2 = _bit_scan_forward(res_2 ^ 0x1FFFF);
res_2 += res_1 + (res_1 << 3);
```

To maximize the instruction-level parallelism and overlap the cache accesses across different input keys, we unroll the loops of all tree level accesses to process 4 input keys at a time. The root uses the vertical approach with an 8-way fanout and is unrolled to process 16 input keys at a time. We evaluated the best degree of loop unrolling experimentally assuming 32-bit keys in a platform with 128-bit SIMD ($\mathcal{W} = 4$).

## 3.3 Sorting Algorithms

### 3.3.1 Setting

Like most prior work, we operate on fixed-length keys and payloads. In read-only workloads typical of data analytics, more complex types can be dictionary encoded into compact integer types that preserve the key order [Raman *et al.*, 2013; Willhalm *et al.*, 2009]. Keys and payloads reside in separate arrays, as would be typical of a column-store analytical database. When the algorithm starts, we assume that our input is evenly divided among the $\mathcal{N}$ NUMA regions in contiguous segments. All columns of the same tuple reside in the same NUMA region. The requirement for $\mathcal{N}$ separate segments is because the operating system either interleaves allocated memory across regions, or allocates space from one region exclusively. To maximize performance, we use $\mathcal{N}$ contiguous segments, one per NUMA region. The output is also a collection of $\mathcal{B}$ segments of sorted data, each residing in a distinct NUMA region. All keys in region $N_i$ have a smaller key than all keys in region $N_j$ for $i < j$. The sorted output is a logical concatenation of the $\mathcal{N}$ segments.

### 3.3.2 Radixsort

Radixsort is one of the most efficient sorting algorithms and is based on radix partitioning. The number of partitioning passes is dependent on the size of the key. Radixsort is typically misrepresented as an $O(n)$ worst-case algorithm. Theoretically, we need $\log n$ bits per value to store $n$ distinct values. Radixsort then needs $O(\log n)$ radix partitioning passes to sort the data, taking $O(n \log n)$ time in total, same as comparison-based sorting. The advantage of radixsort over comparison-based sorting is not due to its complexity, but because radix partitioning is faster than range partitioning, especially if implemented naïvely.

#### 3.3.2.1 Stable LSB Radixsort

The first sorting algorithm we present in this chapter is NUMA-aware least-significant-bit (LSB) radixsort. If the size of key domain is $\mathcal{D}$, we radix partitioning using the $\lceil \log \mathcal{D} \rceil$ bits. Building on the partitioning techniques described in Section 3.2, we can readily present how a NUMA-aware LSB radixsort works. First, we must ensure that during partitioning, we never swap items with equal keys. The order of the items is defined by the bit range of the radix function. All passes, with the exception of the initial pass, must be *stable*, i.e. preserve the ordering of equal items. In-place partitioning is inherently not stable.

If we allocate the array across multiple NUMA regions at random, each tuple will be transferred across NUMA boundaries several times. This design is clearly suboptimal. We ensure that each tuple is transferred through the NUMA interconnection network exactly once. In the first partitioning step, we use $\mathcal{B}$-way range partitioning. The $\mathcal{B} - 1$ delimiters are acquired by uniform sampling. Since $\mathcal{B}$ is small and the $\log \mathcal{B}$ bits cannot saturate the partitioning fanout, we fill up with low order radix bits. The steps are described below:

1. Sample $\mathcal{N} - 1$ range delimiters dividing the data equally across $\mathcal{N}$ NUMA regions.
2. Partition the data locally on each NUMA region using a combined range ($\mathcal{N}$-way) and radix function (see Section 3.2.2.1). Use enough bits to optimize the fanout per pass.
3. Shuffle the data across $\mathcal{N}$ NUMA regions (see Section 3.2.3.1).
4. Perform multiple radix partitioning passes locally per NUMA region to cover all $\log \mathcal{D}$ key domain bits (see Section 3.2.2.1). Use enough bits to optimize the fanout per pass.

All data movement during any partitioning pass are based on parallel out-of-cache partitioning (see Section 3.2.2.1) between the threads of the *same* NUMA region only. Data transfers across NUMA nodes happen only during the shuffling of the $\mathcal{N}$ ranges (see Section 3.2.3.1). Threads operating on the same NUMA nodes always split the workload in equal pieces, regardless of partition sizes and skew. The $\mathcal{N} - 1$ range delimiters ensure equalized segment sizes across the $\mathcal{N}$ NUMA regions. Finally, to compute the mixed range-radix partitioning function of the first phase, we use register-resident delimiters (see Section 3.2.5.1) and concatenate the range function result with the low order radix bits. The number of passes does not change; we only add a minor $\log \mathcal{N}$ range partition bits. We can also support balancing of workload across non-uniform CPUs by dividing the data to $\mathcal{B}$ non-equal range partitions, if we can estimate the relative performance between CPUs.

When the key domain is dense ($\mathcal{D} \sim n$), LSB radixsort performs the minimum number of passes. When the key domain is larger than the cardinality of the array, LSB radixsort needs than $\log n$ passes, eliminating its advantage over other approaches.

#### 3.3.2.2 In-place MSB Radixsort

The second sorting algorithm we present is in-place most-significant-bit (MSB) radixsort. In MSB radixsort, the bits are processed high-to-low, rather than low-to-high as in LSB. The partitioning passes are dividing the data to non-overlapping key ranges, a radix-based analog to quicksort. The outputs of each pass are never interleaved. Since each pass does not maintain the ordering of previous passes, stable partitioning is irrelevant.

The MSB radixsort can work in-place. As with large-scale in-place partitioning, we are the first to propose an algorithm for large-scale in-place sorting. LSB radixsort is faster than MSB radixsort on dense key domains, but the small performance gap justifies paying a small price in time and in exchange save space to be used for other concurrent operators.

In the initial pass, we use shared segment in-place partitioning across all NUMA regions (see Section 3.2.3.2). If the initial bit range does not manage to divide the data in a balanced fashion, the later phases will either operate on unbalanced sub-array sizes, or transfer data across NUMA regions more than once to re-balance work. Mixing radix with range partitioning during the initial pass is a robust solution. Range partitioning ensures

load balancing is unaffected even if the high-order bits of the keys are always the same. Such cases are not uncommon for databases that use fixed data types or schemas with larger data types than necessary for the domain. The steps are described below:

1. Sample $\mathcal{T} - 1$ range delimiters that divide the data equally across $\mathcal{T}$ threads. Add $\mathcal{T}'$ range delimiters ($\mathcal{T}' \geq \mathcal{T}$) generated from the $\log \mathcal{T}'$ high-order bits.
2. Range partition (in parallel) generating linked lists of blocks (see Section 3.2.2.3).
3. Shuffle the blocks across all NUMA regions in-place (see Sections 3.2.2.4 and 3.2.3.2).
4. Sort each segment recursively in a shared-nothing fashion:
   (a) If the (sub-)array is larger than the cache, use out-of-cache shared-nothing radix partitioning (see Section 3.2.2.1) and continue recursively for each partition.
   (b) Otherwise, use in-place in-cache radix partitioning (see Section 3.2.1). Continue recursively to cover the key bits, or use insertsort for tiny sub-arrays ($< 10$ tuples).

We divide the array across $\mathcal{T}$ ranges, where $\mathcal{T}$ is the number of threads, not the NUMA regions. The $\mathcal{T} - 1$ range delimiters are determined via sampling. We add $\log \mathcal{T}$ bits in the same way we added $\log \mathcal{N}$ bits for LSB radixsort. We "hide" the extra bits by overlapping with $2^{\lceil \log \mathcal{T}' \rceil}$ radix partitions generated from the high $\lceil \log \mathcal{T}' \rceil$ radix bits $\mathcal{T}' \geq \mathcal{T}$.

After range partitioning and shuffling across NUMA regions, each thread sorts a partition recursively using both out-of-cache (see Section 3.2.2.1) and in-cache (see Section 3.2.1) partitioning. When the sub-array of size $n$ is smaller than the cache, we use $\approx \log n$ radix bits to generate parts of trivial size [Kim *et al.*, 2009]. We use $\lceil \log n \rceil - 2$ bits and generate partitions of 4–8 tuples, sorted via insertsort. As a result, MSB radixsort covers $\log \min(\mathcal{D}, n)$ bits, in contrast to LSB radixsort that covers $\log \mathcal{D}$ bits.

### 3.3.3 Comparison-Based Sorting

The third sorting algorithm we present is comparison-based and builds on range partitioning. In contrast to radixsort, we rely solely on comparisons to remain independent of the key domain size. Generally, comparison-based sorting allows for perfect load balancing and higher efficiency under skew. The most important comparison-based algorithms are quicksort and mergesort. The latter has been optimized using SIMD [Chhugani *et al.*, 2008].

#### 3.3.3.1 Sorting In The Cache

Prior work [Balkesen *et al.*, 2013a; Chhugani *et al.*, 2008; Inoue *et al.*, 2007; Satish *et al.*, 2010] has suggested SIMD-aware sorting. The proposed method [Balkesen *et al.*, 2013a; Chhugani *et al.*, 2008; Satish *et al.*, 2010] starts from sorting in SIMD registers, then combines multiple registers using sorting networks or bitonic sort, then switches to merging. Outside the database community, an approach that scales to the SIMD length has been proposed [Inoue *et al.*, 2007], initially using $W$-way comb-sort in-cache without comparing keys across lanes, then transpose and merge all parts out-of-cache.

Optimally, we need $O(n/\mathcal{W})\log n$ SIMD instructions to execute $O(n\log n)$ comparisons on $\mathcal{W}$-way SIMD. Sorting networks and bitonic sorting scale to the SIMD length by requiring $O((n/\mathcal{W})\log^2 n)$ instructions. If we view the array as $n/W$ vectors rather than $n$ keys and never compare keys across lanes, we need $O((n/\mathcal{W})\log(n/\mathcal{W}))$ SIMD comparisons. Afterwards, we merge the $W$ arrays with $n\log\mathcal{W}$ comparisons. In total, we need $O((n/\mathcal{W})\log(n/\mathcal{W}) + n\log\mathcal{W})$ comparisons, which is very close to $O((n/\mathcal{W})\log n)$ and better than bitonic sort. Combsort can be extended to operate on vectors of $\mathcal{W}$ tuples.

```
__m128i x = _mm_load_si128(&input_keys[i]);
__m128i y = _mm_load_si128(&input_keys[i + gap]);
_mm_store_si128(&input_keys[i], _mm_min_epu32(x, y));
_mm_store_si128(&input_keys[i + gap], _mm_max_epu32(x, y));
```

After $\mathcal{W}$-wide combsort finishes, we merge the $\mathcal{W}$ lanes. We keep the "last" key from the $W$ arrays in a SIMD register. We also keep the location offsets and the payloads. On each iteration, we find the lane of the minimum key, extract the payloads and the offset, write the minimum tuple to the output, read the next tuple from $i + \mathcal{W}$ where $i$ is the location of the last minimum key, and replace the SIMD lane. We show one iteration with 32-bit keys on 128-bit SIMD below, omitting payloads for simplicity.

```
__m128i min_key = min_across(key);  // find the min key and output it
_mm_stream_si32(output_keys++, _mm_cvtsi128_si32(min_key));
__m128i min_loc = min_across(_mm_or_si128(loc,_mm_xor_si128(mask_I,_mm_cmpeq_epi32(key, min_key))));
size_t i = _mm_cvtsi128_si32(min_loc);  // load the next key and update the index
__m128i new_key = _mm_shuffle_epi32(_mm_cvtsi32_si128(input_keys[i + 4]), 0);
__m128i new_loc = _mm_add_epi32(min_loc, mask_4);
min_loc = _mm_cmpeq_epi32(min_loc, loc);  // insert new key and location
key = _mm_blendv_epi8(key, new_key, min_loc);
loc = _mm_blendv_epi8(loc, new_loc, min_loc);
```

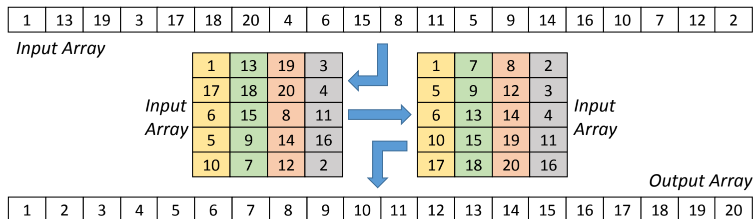



Figure 3.2: SIMD comb-sort (4-way)

To compute the minimum key across a SIMD cell holding $\mathcal{W}$ keys we need $\log W$ comparisons and $\log W$ shuffle instructions. We show the implementation of min-across on Intel SSE below for 4 32-bit keys ($X$, $Y$, $Z$, $W$, where $A = \min(X, Y)$ and $B = \min(Z, W)$):

```
__m128i min_across(__m128i key_XYZW) {
  __m128i key_YXWZ = _mm_shuffle_epi32(key_XYZW, _MM_SHUFFLE(2,3,0,1));
  __m128i key_AABB = _mm_min_epu32(key_XYZW, key_YXWZ);
  __m128i key_BBAA = _mm_shuffle_epi32(key_AABB, _MM_SHUFFLE(1,0,3,2));
  return _mm_min_epu32(key_AABB, key_BBAA); }
```

All steps are shown in Figure 3.2. Given $\mathcal{W} = 4$, the merging step takes significantly less time than the $\mathcal{W}$-way combsort and skips the need for 2-way merging [Inoue *et al.*, 2007] or bitonic sort [Chhugani *et al.*, 2008]. The interleaved arrays are transposed during merging without using auxiliary space. The extra space is used to flush the sorted output, bypassing the cache. The algorithm performs $O((n/\mathcal{W}) \log (n/\mathcal{W}) + n \log \mathcal{W})$ comparisons.

#### 3.3.3.2 Out-of-cache

Prior work [Balkesen *et al.*, 2013a; Chhugani *et al.*, 2008; Satish *et al.*, 2010] argued merge-sort as the most effective comparison-based sorting algorithm. Since merging two streams on each pass is bounded by the RAM bandwidth, merging a larger number of arrays is a better approach. Quick-sort splits the array using 2-way range partitioning optimized to avoid histograms, but is also bounded by the RAM bandwidth when running out-of-cache.

We introduce a new variant for comparison sorting based on range partitioning that uses a large fanout on each pass. Since radixsort can do a lot of radix partitions on each pass, we could do the same for range partitioning. Until recently, this idea hit the wall of slow out-of-cache partitioning, and, until now, the wall of slow range function computation.

The comparison-based algorithm we propose is quite similar to radixsort, but replaces

radix with range partitioning. The version we present here uses non-in-place partitioning, thus, uses linear auxiliary space, similarly to merge-sort. The algorithm decides on a number of range partitioning passes to range-split the data to cache-resident range parts.

To compute the range function, we build a specialized index in the cache (see Section 3.2.5.2) and, alongside computing the histogram, we store the range partition for each tuple to avoid re-computing it. While the tuples are actually moved, we (sequentially) read the range partitions from the array where they are stored during range histogram generation. The data movement part of partitioning performs almost as fast as radix partitioning, since scanning an additional short range array has a small cost. The boost in performance is due to the fast range partition function. We describe the recursive steps below:

1. If in-cache, sort using SIMD combsort and return.
2. Sample range delimiters and generate histogram using the range function index.
3. Use out-of-cache range partitioning locally (see Section 3.2.2.1).
4. If $1^{\text{st}}$ pass, shuffle across the $\mathcal{N}$ NUMA regions (see Section 3.2.3.1).
5. Continue recursively for each range partition.

Using the large fanout of partitioning is essential to avoid doing many passes bounded by the RAM bandwidth. The alternative would be to use merging. *Incremental merging* proposed in [Chhugani *et al.*, 2008] is CPU-bound, but we transfer tuples $\log \mathcal{P}$ times between small intermediate buffers. More recent work that compares merging with partitioning [Balkesen *et al.*, 2013a], shows 16-way merging to be as fast as non-in-place out-of-cache radix partitioning for up to 11-bits. As we will see experimentally, this advantage holds also for range partitioning, using the efficient cache-resident range index that we propose.

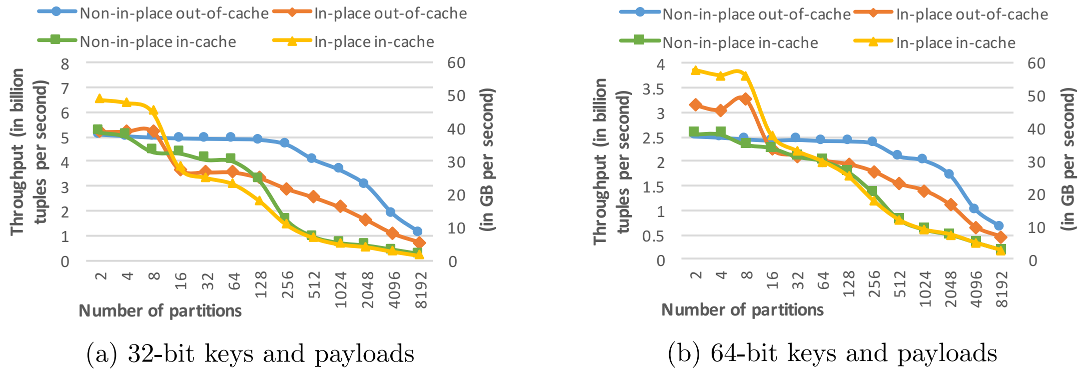
To avoid skew and load balancing problems, we create more range partitions than the total threads before we shuffle across NUMA regions. The threads can fine-grain share the workload and the $\mathcal{B}$ segment sizes can have almost equal sizes. Also, to achieve range partitions of balanced size [Ross and Cieslewicz, 2009], when key $X$ is sampled twice or more as a delimiter, we substitute the multiple repeats with $X$ and $X + 1$, generating a partition with only $X$ tuples. If too many repeated delimiters are discarded, we switch to a smaller range index. The single-key range partitions need no further processing.

## 3.4 Experimental Evaluation

All experiments are run on 4 Intel Xeon E5-4620 8-core CPUs at 2.2 GHz based on the Sandy Bridge micro-architecture with 512 GB quad-channel DDR3 ECC RAM at 1333 MHz. Each CPU provides 16 hardware threads via 2-way simultaneous multi-threading (SMT). Each core has a 32 KB L1 data cache and a 256 KB L2 cache. Each CPU has 8 MB L3 cache shared across the 8 cores. We measured the RAM bandwidth at 122 GB/s for reading, 60 GB/s for writing, and 37.3 GB/s for copying.

The operating system is Linux 3.2 and the compiler is GCC 4.8 with `-O3` optimization. We use SSE (128-bit SIMD) instructions on AVX registers (256-bit), in order to use non-destructive 3-operand instructions that improve pipeline performance. We do use 256-bit SIMD because the platform supports 256-bit SIMD for floating point operations only. Unless stated otherwise, we use all 64 hardware threads and the keys are uniform random. Also, the keys and the payloads are stored on separate arrays.

In Figure 3.3a, we show the performance of the four variants of partitioning using 32-bit keys and 32-bit payloads run in a shared-nothing fashion. The in-cache variants are bound by the TLB capacity, thus, have poor performance for large fanout. When running out-of-cache, the optimal cases are 5–6 bits (32–64 partitions). The out-of-cache variants increase the optimal fanout to 10–12 bits, when non-in-place, and 9–10 bits, when in-place. Using out-of-cache variants for small cache-resident array sizes incurs unnecessary overheads. The optimal fanout is the one with the highest performance per partitioning bit ($= \log \mathcal{P}$, for




(a) 32-bit keys and payloads

(b) 64-bit keys and payloads

Figure 3.3: Throughput of out-of-cache radix partitioning ($10^{10}$ tuples)

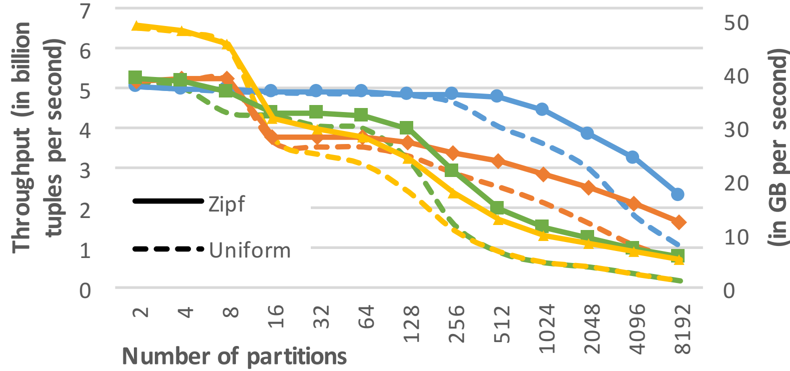
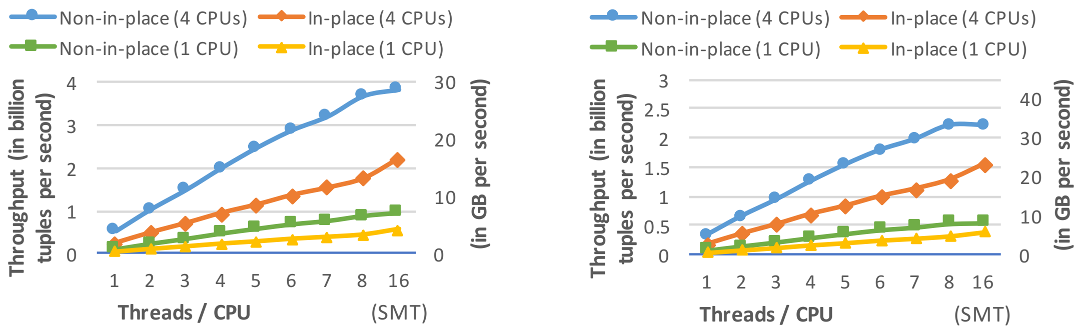



Figure 3.4: Partitioning throughput with uniform random and Zipf data ($\theta = 1.2$)

$\mathcal{P}$-way partitioning) ratio. In Figure 3.3b, we show the performance of partitioning for 64-bit keys and 64-bit payloads. Compared to the 32-bit case, partitioning is actually slightly faster (in GB/s), since RAM accesses and computation overlap more effectively.

In Figure 3.4, we repeat the experiment of Figure 3.3a including runs with data that follow the Zipf distribution with $\theta = 1.2$. Under skew, some partitions are accessed more often than others. Implicit caching of these partitions decreases the probability of cache misses and TLB thrashes, improving performance. With less skew ($\theta < 1$), we found no significant difference in partitioning performance.

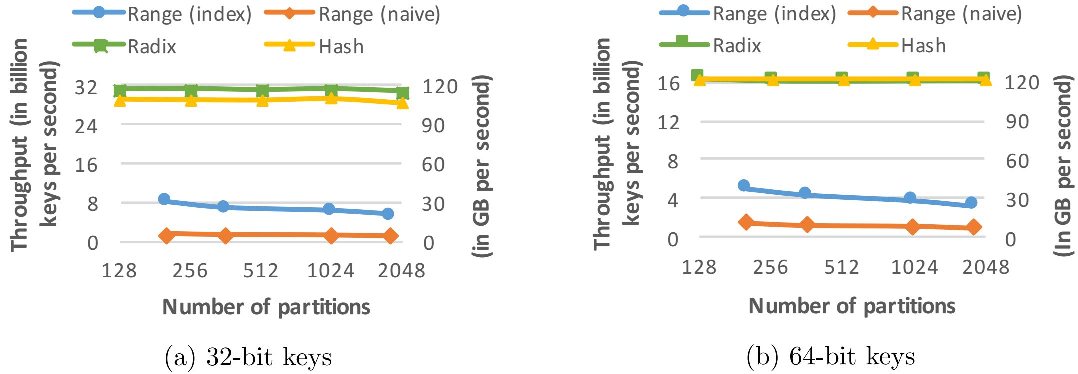
In Figure 3.5, we show the thread scalability of out-of-cache partitioning variants assuming a fanout of 1024. The algorithm scales almost linearly with the number of cores. Noticeably, SMT threads give in-place partitioning a bigger performance boost compared to non-in-place partitioning.




(a) 32-bit keys and 32-bit payloads

(b) 64-bit keys and 64-bit payloads

Figure 3.5: Scalability of out-of-cache radix partitioning ($10^9$ tuples, 1024-way)



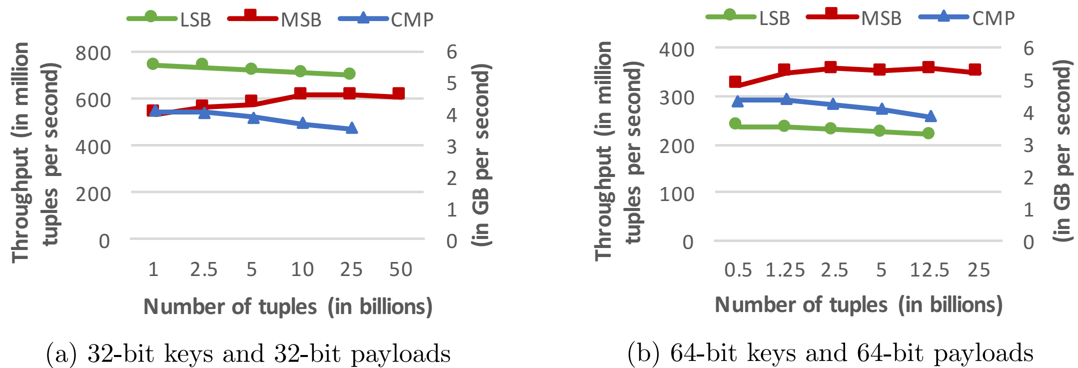

(a) 32-bit keys

(b) 64-bit keys

Figure 3.6: Throughput of histogram generation ($10^{10}$ keys)

In Figure 3.6, we show the performance of histogram generation for all variants of partitioning. Radix and hash partitioning operate roughly at the memory bandwidth. Range partitioning using the configured range function index (see Section 3.2.5.2) improves 4.95–5.8X compared to binary search with 32-bit keys (Figure 3.6a). With 64-bit keys (Figure 3.6b), the range function index speeds-up the process 3.17–3.4X over scalar code, despite the limitation of only 2-way 64-bit operations. The speed-up decreases because scalar binary search doubles its performance (in GB/s), fully shadowing the RAM accesses.

In Figure 3.7a, we show the performance of the three sorting variants that we propose, using tuples with a 32-bit key and a 32-bit payload. LSB denotes LSB radixsort, MSB denotes MSB radixsort and CMP denotes comparison-sort. MSB is 10–20% slower than the fastest LSB and maximizes when the array exceeds the domain size, using only radix partitioning in the cache. Comparison-based sorting is slower but still comparable.




(a) 32-bit keys and 32-bit payloads

(b) 64-bit keys and 64-bit payloads

Figure 3.7: Throughput of all three sorting methods

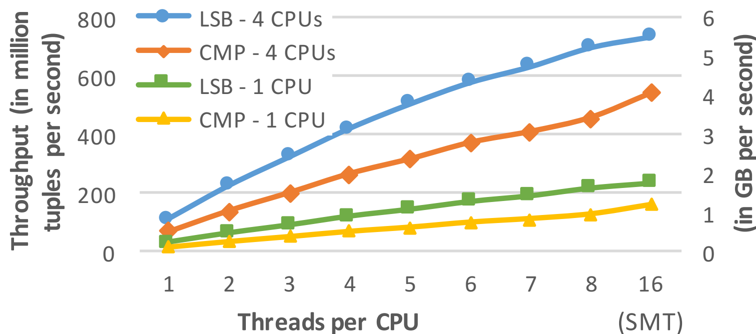
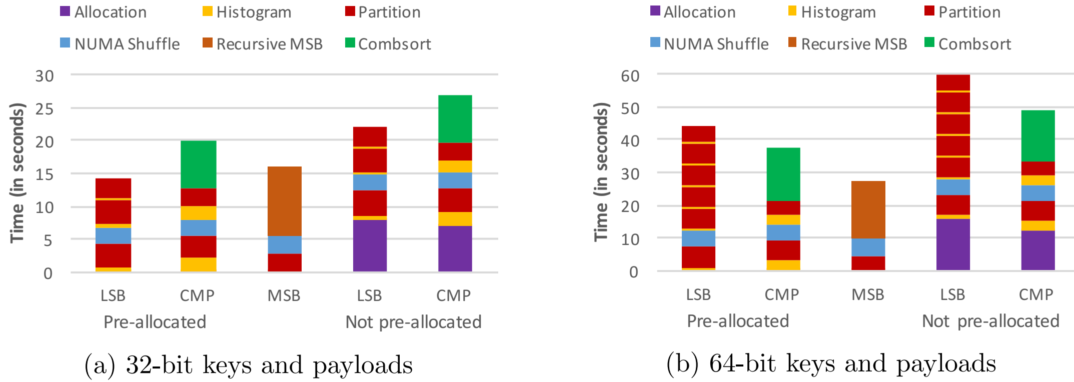



Figure 3.8: Sorting scalability ($10^9$ tuples, 32-bit keys and 32-bit payloads)

In Figure 3.7b, we show the speed of all variants for 64-bit keys and 64-bit payloads. These algorithms are most useful in cases where data-to-query time is important and compression is too expensive, or when tuples change rapidly and cannot be compressed. MSB radixsort is faster than LSB, as it needs less partition passes. When we reach the cache, we create $n/4$ to $n/8$ partitions and then sort 4–8 items using insert-sort. Our advantage over hybrid approaches, such as MSB switching to LSB [Wassenberg and Sanders, 2011], is that the latter will do more passes for sparse keys that are not free, even if cache-resident. Comparison-sort, like MSB radixsort, needs fewer passes than LSB. Range partitioning performs closer to radix, since the data movement costs twice, while the range histogram is less affected. On the other hand, in-cache SIMD sorting is more expensive, since each 128-bit SIMD register can hold two 64-bit keys and cannot be much faster than scalar code.

In Figure 3.8, we show the scalability of both LSB and CMP. CMP benefits more from SMT threads than LSB. Using all hardware threads, the speed-up of 4 CPUs over 1 CPU is 3.13X for LSB and 3.29X for CMP. On 4 CPUs, we cannot achieve a 4X speedup due to the extra step of shuffling across NUMA regions. The step is omitted on 1 CPU.

LSB radixsort is immune to skew due to always splitting the input equally among threads. Comparison-based sorting is in fact significantly faster under skew. For $10^{10}$ tuples following the Zipf distribution, LSB is 5% faster for $\theta = 1$ and 15% faster for $\theta = 1.2$. CMP is 30% faster for $\theta = 1$ and 80% faster for $\theta = 1.2$ outperforming both variants of radixsort. MSB can be affected negatively, but not until high skew ($\theta \geq 1.2$). If excessive key repeats affect the balance across the $\mathcal{N}$ NUMA regions, we can identify these keys during sampling and create single key partitions across NUMA regions [Ross and Cieslewicz, 2009].




(a) 32-bit keys and payloads

(b) 64-bit keys and payloads

Figure 3.9: Execution time per sorting algorithm phase ($10^{10}$ tuples)

In Figure 3.9, we show the execution time spent on each step of the algorithm and also show the additional cost of each algorithm when auxiliary memory is not pre-allocated. MSB outperforms LSB radixsort when auxiliary space is not already available. CMP needs only two range partitioning passes to split into sub-arrays that are smaller than the cache. However, it still spends $\sim$ 40% of the total execution time to sort the sub-arrays in the cache using SIMD combsort.

In Figure 3.10, we compare NUMA-aware and NUMA-oblivious variants of sorting algorithms. The effect of the NUMA layer is dependent on the number of partitioning passes. A pass can be more than 50% slower on NUMA-interleaved memory. Using an extra pass for NUMA shuffling always helps. LSB radixsort is $\approx$ 25% faster when NUMA-aware, even

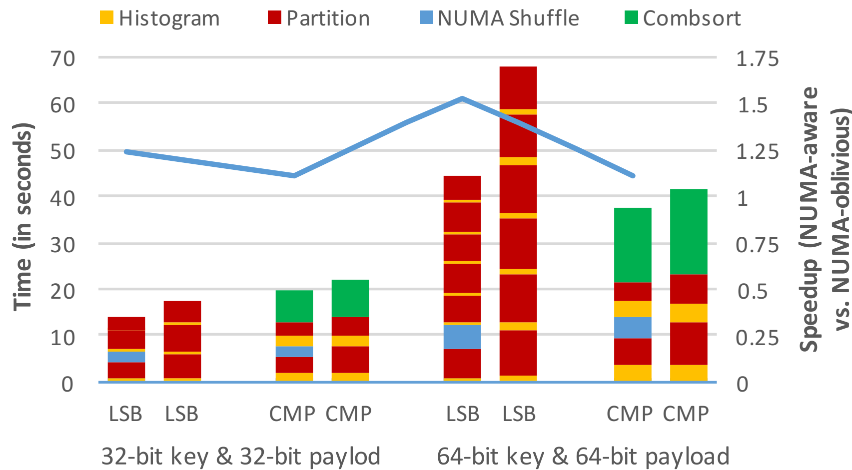



Figure 3.10: NUMA-aware vs. NUMA-oblivious sorting ($10^{10}$ tuples)

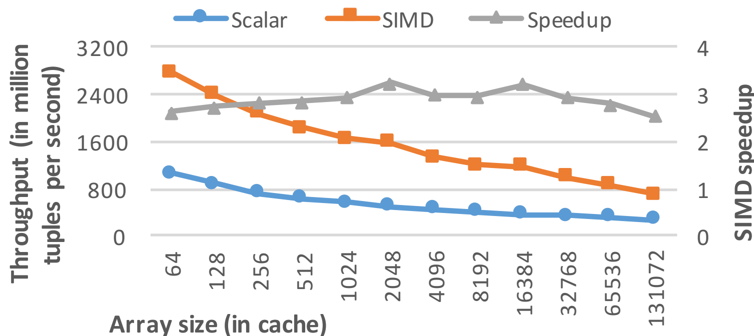



Figure 3.11: Performance of scalar and SIMD comb-sort (32-bit keys and 32-bit payloads)

if only 3 passes are required for 32-bit keys. When using 64-bit keys, being NUMA-aware is more than 50% faster. Finally, the effect on comparison sort is smaller (10–15%), since range histograms are CPU-bound and the number of passes is minimal.

In Figure 3.11, we show the performance of SIMD combsort using 32-bit keys and 32-bit payloads. The input is split in smaller sub-arrays and we sort each sub-array independently in a single thread. The process is the same as the last phase of CMP. The speedup of SIMD combsort over the scalar version is 2.9X on average, which is fairly close to the 4X upper bound ($\mathcal{W} = 4$) for 32-bit keys and 128-bit SIMD registers.

Large-scale sorting has been recently used for sort-merge-join operators. [Albutiu *et al.*, 2012] performs an in-place MSB radix partitioning pass using a simpler variant of in-place in-cache radix partitioning, then uses introsort [Musser, 1997] to sort the partitions. This hybrid approach operates in-place, but is neither purely comparison-based nor purely a radixsort. In-place radixsort is 2–3X faster than introsort on 32-bit keys, and a single radix partitioning step still leaves most of the work for introsort. [Balkesen *et al.*, 2013a], that improves over [Albutiu *et al.*, 2012], uses similar hardware to ours (4 Intel Sandy Bridge CPUs, 32 cores with SMT, NUMA memory). For 1 billion tuples, [Balkesen *et al.*, 2013a] shows a sorting rate of 350 million tuples per second using merging, compared to our comparison-based sorting that sorts 540 million tuples per second. Again for 1 billion tuples, [Balkesen *et al.*, 2013a] shows a sorting rate of 675 million tuples per second using non-in-place MSB radix partitioning out of the cache, compared to our non-in-place LSB radixsort that sorts 740 million tuples per second. Thus, we outperform prior work on both radix- and comparison-based sorting. Furthermore, [Albutiu *et al.*, 2012] and [Balkesen

*et al.*, 2013a] implicitly assume that the key value range covers the entire key domain, as MSB radix partitioning is not combined with range partitioning to guarantee load balancing across threads. The MSB radix partitioning pass can be rendered useless if the high-order bits are not uniform random. Neither work considers large key domains.

The performance of SIMD-optimized mergesort in [Chhugani *et al.*, 2008] drops by 12.4% when sorting $2^{28}$ keys (1 GB) compared to $2^{27}$, and in [Balkesen *et al.*, 2013a] by 25% when sorting $2^{30}$ tuples (8 GB) compared to $2^{29}$. The performance of our comparison-based sorting algorithm drops by 13% when sorting 25 billion tuples (186.26 GB) compared to 1 billion (7.45 GB). Thus, our optimized range partitioning is more scalable than merging on large-scale comparison-based sorting.

## 3.5 Conclusion

We studied a wide menu of partitioning variants across all layers of the memory hierarchy, introduced large-scale in-place partitioning, and provided guarantees for minimal transfers across NUMA regions on platforms with multiple processors. We designed a SIMD-base range index to optimize the computation of range partitioning function, making range partitioning comparably fast with radix or hash partitioning.

By combining multiple partitioning variants, we designed three large-scale sorting algorithms that are the fastest to date: a stable LSB radixsort, an in-place MSB radixsort, and a comparison-based sorting algorithm based on range partitioning. Our evaluation on large-scale arrays with billions of tuples suggests that (i) LSB radixsort is best on dense key domains and is immune to skew, (ii) MSB radixsort is best on sparse key domains or when saving memory space is important, and (iii) comparison-based sorting is guarantees optimal load balancing and higher performance under skew.

Our work can serve as a tool for designing other large-scale database operations by combining the most suitable partitioning variants. In-place partitioning offers a valuable trade-off between space and time. Range partitioning guarantees load balancing regardless of skew or the key domain. Enforcing minimal transfers across NUMA boundaries guarantees efficiency and scalability on future even more parallel hardware.

# Chapter 4

# Lightweight Compression Alongside Fast Scans

## 4.1 Introduction

The increase in main-memory capacity has allowed small to medium-sized databases to fit in main memory and the bottleneck has shifted from disk access to the memory bandwidth. In-memory query execution has raised the bar and vendors strive to provide real-time evaluation of analytical queries, even if a large portion of the database is accessed.

Column stores are a structural evolution of analytical database systems in order to optimize for the memory bandwidth bottleneck. By storing each column separately, we maximize the portion of useful data transferred for queries that access a few columns. To optimize the data fetching rate, we have to maximize both the useful data per unit of transfer and the rate of transfers by avoiding random accesses.

Sequential memory scans are preferred over random accesses. Dropping the indexes altogether in favor of linear table scans is now a sensible design [Raman *et al.*, 2013]. Cache-conscious joins, sorting, and aggregation also eliminate non-sequential accesses, while facilitating thread scalability. By tuning each operator to the underlying hardware, database systems maximize performance close to the hardware limit. In this context, memory bandwidth is the most important bottleneck.

Analytical databases are compressed, not only to fit the data in main memory, but

also to improve beyond the memory bandwidth. Compression schemes allow operators to process the data directly on the compressed format [Raman *et al.*, 2013; Willhalm *et al.*, 2009]. The most common scheme is dictionary compression. In its pure form, the $n$ distinct values of each column are kept in a sorted dictionary and instead of storing the actual column values of each tuple, we store the dictionary indexes as codes using $\lceil \log n \rceil$ bits. Joins and selections can process the compressed codes instead of the actual values. The predicate constants present in the query are also converted into dictionary codes using the sorted dictionary, maintaining the same comparison order.

There are multiple approaches regarding the storage layout of dictionary codes. In the standard horizontal bit packing, we can accelerate scans by wasting space to ensure word-alignment and allow for in-register data-parallel predicate evaluation [Johnson *et al.*, 2008]. Otherwise, we can still unpack the fully packed codes efficiently using SIMD instructions [Willhalm *et al.*, 2009]. To achieve optimal scan performance by even skipping some parts of the data, we must pack the bits vertically [Li and Patel, 2013] by interleaving them. However, the vertical layout is too expensive to generate and incurs prohibitive costs for extracting a large portion of tuples for cases with high selectivity.

In this chapter, we study both the horizontal and the vertical bit packing layout, focusing on three operations: packing, unpacking, and selection scans. We optimize the operations on the horizontal layout, including the word-aligned variant, by using instructions that are only available in recent mainstream CPUs and achieve significantly better instruction efficiency. Our novel contribution is the design and implementation of the packing and unpacking operations on the vertical bit packing layout using SIMD instructions. Our approach increases performance by more than an order of magnitude. Finally, we compare all approaches and highlight respective strengths and weaknesses and conclude that while horizontal bit packing is highly competitive, vertical bit packing combines the fastest selection scans while being comparably fast for packing and unpacking. Our work offers deep insights on these layouts and extends the trade-off options for lightweight database compression.

The rest of the chapter is organized as follows. In Sections 4.2 and 4.3, we discuss horizontal and vertical bit packing respectively. In Section 4.4, we present our experimental evaluation and we conclude in Section 4.5.

## 4.2 Horizontal Bit Packing

Horizontal bit packing stores an array of integer codes using a constant number of bits. Each integer code points to the same column value (by index) in a dictionary of unique values for the column. We need $b = \lceil logn \rceil$ bits to represent $n$ distinct values. In horizontal encoding, the bits per code are stored contiguously. For example, if $b = 4$ and we want to pack codes 3 (0011), 5 (0101), 7 (0111), and 9 (1001), we would pack them as: 1001011101010011.

### 4.2.1 Scalar Packing and Unpacking

Packing 32-bit codes to $b$-bit codes is shown below in C. We iterate over the unpacked codes and construct the next packed word using a 64-bit scalar register. We branch once the processor word with packed codes is full, store the word to output, and restart filling the word with new codes. We also handle cases where codes span across two processor words.

```
void compress(const uint32_t* uncompressed, int8_t bits, size_t tuples, uint64_t* compressed) {
  uint64_t word = 0;  // the word of packed codes
  int8_t word_bits = 0;  // current number of used bits in the word of packed codes
  size_t i = 0, o = 0;
  do {  // pack one code per iteration
    uint64_t code = uncompressed[i++];  // load next code from input
    word |= code << word_bits;  // insert code in word of packed codes
    word_bits += bits;  // update the number of used bits in the word
    if (word_bits >= 64) {  // if we exceed the size of the word
      compressed[o++] = word;  // store the previous word of packed codes
      word_bits -= 64;
      word = code >> (bits - word_bits);  // reset the word with remaining bits from the last code
    }
  } while (i != tuples);
}
```

We can optimize this methods for each code size individually. We can loop unroll the packing of the next 32 codes, use constant stride shifts and bitwise operations to pack the codes, and remove the branch instructions completely [Lemire and Boytsov, 2015; Zukowski *et al.*, 2006]. We do not evaluate such optimizations here.

The scalar code for unpacking the horizontal layout is shown below. We load and process one 64-bit processor word at a time. Once we process all packed words from the 64-bit processor word, we perform a branch to fetch the next 64-bit word. The bits of a dictionary code may span across two words so we may to stitch pieces.

```
void decompress(const uint64_t* compressed, int8_t bits, size_t tuples, uint32_t* uncompressed) {
  uint64_t word = 0;  // the word of packed codes
  int8_t word_bits = 0;  // current number of used bits in the word of packed codes
  const uint64_t max_code = (1 << bits) - 1;  // mask to clear the high-order bits of unpacked codes
  size_t i = 0, o = 0;
  do {  // unpack one code per iteration
    uint64_t code = word & max_code;  // copy the code and clear the high-order bits
    word >>= word_bits;  // shift the word of packed codes
    word_bits -= bits;  // update the number of used bits in the word of packed codes
    if (word_bits < 0) {  // if the word is out of bits
      uint64_t next_word = compressed[i++];  // read next word of packed codes
      word = next_word >> (0 - word_bits);  // ignore bits of last code from word
      word_bits += 64;
      code |= (next_word << word_bits) >> (64 - bits);  // stitch bits for the last code
    }
    uncompressed[o++] = code;
  } while (o != tuples);
}
```

Similarly to packing, we can also unroll the loop of unpacking, use constant stride shifts, and eliminate the branches [Lemire and Boytsov, 2015; Zukowski *et al.*, 2006].

### 4.2.2 Word-Aligned Scalar Scanning

Based on earlier work [Johnson *et al.*, 2008; Lamport, 1975], scanning the packed format of dictionary codes to evaluate selective predicates can be done directly on the packed format without unpacking using scalar code. The method uses two observations on binary arithmetic first made by [Lamport, 1975]. First, the $C_1 \neq C_2$ boolean expression in $b$-bit arithmetic is equal to the overflow bit of $(C_1 \texttt{ xor } C_2) + (2^b - 1)$. Second, the $C_1 < C_2$ expression is equal to the overflow bit of $(C_1 \texttt{ xor } (2^b - 1)) + C_2$.

To parallelize the predicate evaluation, we pack $b' = \lfloor 64/(b+1) \rfloor$ codes per 64-bit word. Each code uses $b + 1$ bits. The extra bit per code is the overflow bit that allows predicate evaluation to occur in parallel for all the $b'$ codes packed in the processor word. The overflow bit also ensures that the addition does not propagate a carry to the following $b + 1$ bits of the next packed code. Packing and unpacking this format are simplified by using an inner loop to pack the $b'$ codes per 64-bit word. However, the remaining $64 - b' \cdot (b + 1)$ bits at the end of each word are wasted, in addition to the extra bit per $b$-bit code that we utilize as an overflow bit.

The scalar code for evaluating the $C_1 \leq column \leq C_2$ selective predicate on the word-aligned horizontal packed layout is shown below. $C_1$ and $C_2$ are query constants.

```
void scan(const uint64_t* compressed, int8_t bits, size_t tuples,
    uint64_t* bitmap, uint32_t min, uint32_t max) {
  const uint64_t mask_min = repeat(min, bits + 1);  // repeat min every b+1 bits
  const uint64_t mask_max = repeat(max, bits + 1);  // repeat max every b+1 bits
  const uint64_t mask_1_0b = repeat(1 << bits, bits + 1);  // set highest bit every b+1 bits
  const uint64_t mask_0_1b = mask_1_0b - (mask_1_0b >> bits);  // reset highest bit every b+1 bits
  const int8_t bm_bits = 64 / (bits + 1);  // packed codes per 64-bit word
  const size_t words = tuples / bm_bits;  // number of packed 64-bit words
  uint64_t bm_word = 0;
  int8_t bm_word_bits = 0;  // horizontal bit packing for the output bitmap
  size_t i = 0, o = 0;
  do {  // scan one packed word per iteration
    uint64_t word = compressed[i++];
    uint64_t cmp_lt_min = mask_min + (word ^ mask_0_1b);  // evaluate x < min
    uint64_t cmp_gt_max = word + (mask_max ^ mask_0_1b);  // evaluate max < x
    // combine the results: (x < min or x > max) = not(x >= min and x <= max)
    word = _pext_u64(cmp_lt_min | cmp_gt_max, mask_1_0b);  // extract highest bit per b+1 bits
    bm_word |= word << bm_word_bits;  // pack the predicate result bits into the bitmap
    bm_word_bits += b_bits;  // this is the same as regular horizontal packing
    if (bitmap_word_bits >= 64) {
      bitmap[o++] = ~bm_word;  // not(value < min or value > max) = (value >= min and value <= max)
      bm_word_bits -= 64;
      bm_word = word >> (bm_bits - bm_word_bits);
    }
  } while (i != words);
  [...]  // process (up to) one more 64-bit word with outstanding tuples
}
```

The helper function `repeat(x,y)` replicates `x` every `y` bits in a 64-bit word. The `_pext_u64(x,y)` instruction extracts and compacts bits from `x`. The target bits are denoted by `y`. We use the instruction to extract the overflow bits of the packed codes and compact them in the low-order bits of the output. The instruction is supported in all mainstream CPUs based on the Haswell micro-architecture or more recent.

We execute very few scalar instructions per processor word to evaluate the selective predicate for $b'$ codes. As a result, selection scans are bound by the memory bandwidth, and performance is reduced proportionally to the percentage of wasted space.

We can also use this technique on the fully packed format by using special masks to avoid bits spilling across the packed words. We do not evaluate this approach since horizontal layouts can be efficiently unpacked in SIMD code.

### 4.2.3 SIMD Unpacking and Scanning

Based on earlier work [Willhalm *et al.*, 2009], unpacking the horizontal layout can be done efficiently in SIMD code without wasting space. We use the same layout we used in Section 4.2.1. We can replicate the packed bytes of the packed format in SIMD lanes and unpack one code per lane. If each SIMD register fits $\mathcal{W}$ unpacked codes, we unpack $\mathcal{W}$ codes at a time using the following steps:

1. Load the next $\mathcal{W} \cdot b$ bits from the packed input in a SIMD register. If $\mathcal{W} \equiv 0 \mod 8$, load the next $b$ bytes using an unaligned (byte-aligned) SIMD load.
2. Shuffle the bytes to align codes at specific byte lanes of the SIMD register. Assuming 32-bit (unpacked) codes, bytes $4i$ to $4i + 3$ of the $i$-th 32-bit lane are pulled from bytes $j$ to $j + 3$ of the loaded vector, where $j = \lfloor i \times b \div 8 \rfloor$.
3. Shift the (unpacked) codes per SIMD lane to clear low-order bits from other (packed) codes. The $i$-th lane is shifted right by $((i \times b) \mod 8)$.
4. Bitwise-`and` the (unpacked) codes per SIMD lanes with $2^b - 1$ to clear the high-order bits that belong to the next (packed) codes.
5. Store the $\mathcal{W}$ unpacked codes or use them to evaluate selective predicates.

We show how to generate the constant masks that can be pre-computed and placed in the source file. We use 256-bit (`AVX2`) SIMD and assume 32-bit (unpacked) codes ($\mathcal{W} = 8$).

```
int32_t shift[31][8];  // shift stride to clear low-order bits of 32-bit (unpacked) words
int8_t shuffle_lo[31][32];  // byte permutation masks for lower half of AVX register
int8_t shuffle_hi[31][32];  // byte permutation masks for lower half of AVX register
for (size_t b = 0; b != 31; ++b) {  // for each possible number of bits
  for (size_t i = 0; i != 8; ++i) {  // for each SIMD lane with an (unpacked) word
    for (size_t j = 0; j != 4; ++j) {  // for each byte in the (unpacked) word
      int8_t byte_dest = (i * bits) / 8 + j;  // the destination of the byte
      if (byte_dest < 16) {  // if byte is part of the lower half in the AVX register
        shuffle_lo[b][i * 4 + j] = byte_dest;  // destination of lower half byte
        shuffle_hi[b][i * 4 + j] = -1;  // clear byte during AVX byte permutation
      } else {  // if byte is part of the upper half in the AVX register
        shuffle_lo[b][i * 4 + j] = -1;  // clear byte during AVX byte permutation
        shuffle_hi[b][i * 4 + j] = byte_dest - 16;  // destination of upper half byte
      }
    }
    shift[b][i] = (i * bits) / 8;
  }
}
```

```
void decompress_16(const uint8_t* compressed, int8_t bits, size_t tuples, uint32_t* uncompressed) {
  const __m256i mask_shuffle = _mm256_loadu_si256((__m256i*) shuffle_lo[bits]);  // constant masks
  const __m256i mask_shift = _mm256_loadu_si256((__m256i*) shift[bits]);
  const __m256i mask_and = _mm256_set1_epi32((1 << bits) - 1);
  size_t i = 0, o = 0;
  do {
    __m256i data = _mm_loadu_si128((__m128i*) &compressed[i]);  i += bits;  // load 128 bits
    data = _mm256_permute4x64_epi64(data, _MM_SHUFFLE(1,0,1,0));  // broadcast to 256 bits
    data = _mm256_shuffle_epi8(data, mask_shuffle);  // place codes in the right SIMD lanes
    data = _mm256_srlv_epi32(data, mask_shift);  // shift left to clear low-order bits
    data = _mm256_and_si256(data, mask_and);  // mask (bitwise-and) to clear high-order bits
    _mm256_store_si256((__m256i*) &uncompressed[o], data);  // store unpacked codes
  } while ((o = o + 8) != tuples);
}
```

The SIMD code for unpacking horizontally bit packed codes using 256-bit (`AVX2`) SIMD is shown above. In our evaluation, we manually unroll the loop four times to eliminate most instruction latencies and maximize IPC. Specifically for `AVX2` SIMD, byte shuffling permutes bytes within each 128-bit half of the 256-bit register but not across the two halves. To shuffle bytes across the entire 256-bit register, we broadcast each half and shuffle it separately. We then blend the two halves. If $b \leq 16$, we only need the lower 128-bit half of the 256-bit register. If $16 < b \leq 26$ (or $b = 28$), we need to process both halves of the `AVX2` register. This case is shown below. If $b \in \{27, 29, 30, 31\}$, we need more instructions to unpack 8 codes, because the 32-bit shifts exceed the boundaries of 32-bit SIMD lanes.

```
void decompress_26(const uint8_t* compressed, int8_t bits, size_t tuples, uint32_t* uncompressed) {
  const __m256i mask_shuffle_lo = _mm256_loadu_si256((__m256i*) shuffle_lo[bits]);  // masks
  const __m256i mask_shuffle_hi = _mm256_loadu_si256((__m256i*) shuffle_hi[bits]);
  const __m256i mask_shift = _mm256_loadu_si256((__m256i*) shift[bits]);
  const __m256i mask_and = _mm256_set1_epi32((1 << bits) - 1);
  size_t i = 0, o = 0;
  do {
    // input progresses by 8 (packed) codes = 8b bits = b bytes
    __m256i data = _mm256_loadu_si256((__m256i*) &compressed[i]); i += bits;  // load 256 bits
    // broadcast lower (128-bit) half of (256-bit) AVX register to both halves
    __m256i data_lo = _mm256_permute4x64_epi64(data, _MM_SHUFFLE(1,0,1,0));
    // broadcast upper (128-bit) half of (256-bit) AVX register to both halves
    __m256i data_hi = _mm256_permute4x64_epi64(data, _MM_SHUFFLE(3,2,3,2));
    data_lo = _mm256_shuffle_epi8(data_lo, mask_shuffle_lo);  // permute bytes of lower half
    data_hi = _mm256_shuffle_epi8(data_hi, mask_shuffle_hi);  // permute bytes of upper half
    data = _mm256_or_si256(data_lo, data_hi);  // merge halves
    data = _mm256_srlv_epi32(data, mask_shift);  // align to 32-bit integer boundaries
    data = _mm256_and_si256(data, mask_and);  // clear high-order bits
    _mm256_store_si256((__m256i*) &uncompressed[o], data);  // store unpacked codes
  } while ((o = o + 8) != tuples);
}
```

Evaluating the $C_1 \leq column \leq C_2$ predicate and extracting a bitmask adds 4 more instructions. Previous work using 128-bit SIMD (SSE) reported 10 instructions per code [Li and Patel, 2013], which is also due to SIMD variable stride shifts being emulated via SIMD multiplications that are considerably more expensive instructions.

By viewing the input as $\mathcal{W}$ interleaved streams of packed codes, we can extend the optimized per $b$ compressing and decompressing [Zukowski *et al.*, 2006] to use SIMD instructions [Lemire and Boytsov, 2015]. Each stream is processed by one of the $\mathcal{W}$ SIMD lanes. Compression is much faster, assuming we are not memory bound. On the other hand, decompression saves very few instructions compared to the version presented above.

## 4.3 Vertical Bit Packing

In vertical bit packing [Li and Patel, 2013], the $b$ bits per code are not stored contiguously, but are interleaved for groups of $k$ codes. The bits of the next $k$ codes are interleaved in $b$ $k$-bit words and the $i$-th word has the $i$-th bit of each code. The layout allows predicate evaluation on the packed format and executes at most the same order of instructions as word-aligned horizontal bit packing (Section 4.2.2), but without wasting any bits. Processor word alignment is guaranteed by setting $k$ to be a multiple of a processor word length.

### 4.3.1 Scalar Packing and Unpacking

Scalar C code for vertical bit packing with $k = 64$ is shown below. We extract one bit at a time and add it to the packed word. The process is repeated $b$ times per code. The algorithm runs in $O(nb)$ time because we pack one bit per code at a time.

```
void compress(const uint32_t* uncompressed, int8_t bits, size_t tuples, uint64_t* compressed) {
  size_t i, o, b1, b2, b = bits;
  for (i = o = 0; i != tuples; i += 64, o += b) {
    for (b1 = b - 1; 0 <= (ssize_t) b1; --b1) {
      uint64_t word;
      for (b2 = 63; 0 <= (ssize_t) b2; --b2) {
        word = (word << 1) + ((uncompressed[i + b2] >> b1) & 1);
      }
      compressed[o + b1] = word;
    }
  }
}
```

```
void decompress(const uint64_t* compressed, int8_t bits, size_t tuples, uint32_t* uncompressed) {
  size_t i, o, b1, b2, b = bits;
  for (i = o = 0; o != tuples; i += b, o += 64) {
    for (b1 = 0; b1 != 64; ++b1) {
      uint32_t code = 0;
      for (b2 = 0; b2 != b; ++b2) {
        code += code + ((compressed[i + b2] >> b1) & 1);
      }
      uncompressed[o + b1] = code;
    }
  }
}
```

Unpacking the vertical layout is symmetrical and is shown above. We build each unpacked code by extracting one bit per packed word at a time. No matter how much we optimize the scalar code here, the vertical layout is still at least an order of magnitude slower than the horizontal layout, for both packing and unpacking.

### 4.3.2 SIMD Packing

The simplest version of SIMD vertical bit packing holds all $k$ unpacked codes in SIMD registers and extract one bit per lane from each SIMD register. The process is repeated $b$ times for $b$-bit codes. If the CPU core provides 16 256-bit registers, we can use 8 to hold $k = 64$ unpacked codes. We process $k$ unpacked codes at a time with the following steps:

1. Load the next $k$ words in $k/\mathcal{W}$ SIMD registers.
2. Shift left the loaded $k/\mathcal{W}$ SIMD registers by $32 - b$ for 32-bit (unpacked) codes.
3. Repeat $b$ times:
   (a) Extract the $\mathcal{W}$ sign bits per register into a bitmask for all $k/\mathcal{W}$ registers.
   (b) Store (append) the $k/\mathcal{W}$ bitmasks ($k$ bits) to the output.
   (c) Double the $k/\mathcal{W}$ SIMD registers.

SIMD code is significantly faster than scalar code. The time complexity is reduced from $O(nb)$ to $O(nb/\mathcal{W})$. Additionally, we avoid reloading the unpacked codes several times from the L1 cache but keep them resident in SIMD registers instead. We extract multiple bits per instruction, executing $b/\mathcal{W}$ bit extraction instructions per code rather than $b$. To further improve performance, we pack each byte of the packed codes in fewer SIMD registers and extract bits using 8-bit lanes. The steps to unpack $k$ codes are described below:

1. Load the next $k$ words in $k/\mathcal{W}$ SIMD registers.
2. Shift the $k/\mathcal{W}$ SIMD registers left by $32 - b$ bits.
3. Repeat $\lceil b \div 4 \rceil$ times:
   (a) Pack the high-order byte per 32-bit lane from $k/\mathcal{W}$ SIMD registers into $k/4\mathcal{W}$ registers.
   (b) Repeat 8 times or until out of bits:
      i. Extract the $k/4\mathcal{W}$ sign bits per register into a bitmask into for $k/4\mathcal{W}$ registers.
      ii. Store (append) the $k/4\mathcal{W}$ (scalar) bitmasks ($k$ bits) to the output.
      iii. Double the $k/4\mathcal{W}$ SIMD registers.
   (c) Shift the $k/\mathcal{W}$ registers right by 8 bits per 32-bit lane.

SIMD vertical packing (`AVX2`) is shown below for $k = 64$. By using 8-bit SIMD lanes to extract bits, we reduce the SIMD instructions to pack 64 codes from $(16b+8)$ to $(10b+30)$.

```
void compress(const uint32_t* uncompressed, int8_t bits, size_t tuples, uint64_t* compressed) {
  const __m128i mask_shift = _mm_cvtsi32_si128(32 - bits);
  const __m256i mask_order = _mm256_set_epi32(7,3,6,2,5,1,4,0);
  const int8_t bytes = (bits + 7) / 8, bits_in_last_byte = bits & 7;
  size_t i = 0, o = 0;
  do {  // pack k = 64 codes per iteration
    __m256i r1 = _mm256_load_si256((__m256i*) &uncompressed[i]);  // load 64 32-bit integers
    r1 = _mm256_sll_epi32(r1, mask_shift);  // shift left to move b-th bit as MSB (sign bit)
    [...]  // repeat for r2, r3, ..., r8 (14 lines omitted)
    int8_t outer_loops = bytes;
    do {  // process codes using 8-bit (byte) SIMD lanes
      __m256i s1 = _mm256_srli_epi32(r1, 24);  // isolate the high order byte as the low byte
      [...]  // repeat for s2, s3, ..., s8 (7 lines omitted)
      __m256i s12 = _mm256_packus_epi32(s1, s2);  // pack 32-bit into 16-bit in 1/2 of registers
      [...]  // repeat for s34, s56, and s78 (3 lines omitted)
      __m256i s1234 = _mm256_packus_epi16(s12, s34);  // pack 16-bit into 8-bit in 1/4 of registers
      __m256i s5678 = _mm256_packus_epi16(s56, s78);
      s1234 = _mm256_permutevar8x32_epi32(s1234, mask_order);  // restore order polluted by packs
      s5678 = _mm256_permutevar8x32_epi32(s5678, mask_order);
      int8_t inner_loops = (outer_loops != 1) ? 8 : bits_in_last_byte;
      do {  // extract one bit at a time per byte lane
        uint64_t lo = (unsigned) _mm256_movemask_epi8(s1234);
        uint64_t hi = (unsigned) _mm256_movemask_epi8(s5678);
        s1234 = _mm256_add_epi8(s1234, s1234);  // shift bytes left by one bit
        s5678 = _mm256_add_epi8(s5678, s5678);
        compressed[o++] = lo | (hi << 32);  // store compressed format 64-bit word at a time
      } while (--inner_loops);
      r1 = _mm256_slli_epi32(r1, 8);  // shift the input registers left by 1 byte
      [...]  // repeat for r2, r3, ..., r8 (7 lines omitted)
    } while (--outer_loops);
  } while ((i = i + 64) != tuples);
}
```

### 4.3.3 SIMD Unpacking

Unpacking the vertical bit packed format is based on the same principles as packing but performs the inverse operation. Instead of extracting one bit per SIMD lane, we convert the packed bits to masks in SIMD lanes and insert one bit per SIMD lane in the SIMD registers that hold the $k$ unpacked codes.

1. Set the next $k$ codes in $k/\mathcal{W}$ SIMD registers to zero.
2. Repeat $b$ times:
   (a) Load the next $k$ bits from the input.
   (b) Convert $k$ bits to {0,-1} masks in $k/\mathcal{W}$ registers.
   (c) Double the $k/\mathcal{W}$ registers and subtract the masks.
3. Append the $k/\mathcal{W}$ registers to the output.

Similarly to bit packing, we do not reload the input multiple times but keep the output unpacked codes in SIMD registers. To improve the number of bits inserted per instruction, we use smaller SIMD lanes again. We extract bits in byte lanes that are then up-converted and inserted in the $k/\mathcal{W}$ output registers. The steps per $k$ codes are shown below:

1. Set the next $k$ codes in $k/\mathcal{W}$ SIMD registers to zero.
2. Repeat $\lceil b/4 \rceil$ times:
   (a) Reset $k/4\mathcal{W}$ SIMD registers to zero.
   (b) Repeat 8 times or until out of bits:
      i. Load the next $k$ bits from the (packed) input.
      ii. Convert $k$ bits to {0,-1} byte masks in $k/4\mathcal{W}$ registers.
      iii. Double the $k/4\mathcal{W}$ registers and subtract the byte masks.
   (c) Shift the $k/\mathcal{W}$ registers left by 8 bits per 32-bit lane.
   (d) Up-convert 8-bit lanes in $k/4\mathcal{W}$ registers to 32-bit lanes.
   (e) Merge (bitwise-`or`) the up-converted registers with the unpacked codes.
3. Store (append) the $k/\mathcal{W}$ registers holding the unpacked codes to the output.

The code for unpacking using 256-bit SIMD (AVX) intrinsic is shown below. We execute $18b + 30$ SIMD instructions per 64 codes, which is $O(nb)$. However, because we are, we compress the codes as fast as $O(n)$ SIMD horizontal unpacking.

```
void decompress(const uint64_t* compressed, int8_t bits, size_t tuples, uint32_t* decompressed) {
  const uint64_t m4 = 0x4040404040404040ll, m8 = 0x8080808080808080ll, mC = 0xC0C0C0C0C0C0C0C0ll;
  const __m256i mask_shuffle_lo = _mm256_set_epi64x(m8, 0, m8, 0);
  const __m256i mask_shuffle_hi = _mm256_set_epi64x(mC, m4, mC, m4);
  const __m256i mask_bit = _mm256_set1_epi64x(0x8040201008040201ll);
  size_t i = 0, o = 0;
  do {
    __m256i r1 = _mm256_setzero_si256();  // reset registers holding unpacked words
    [...]  // repeat for r2, r3, ..., r8 (7 lines omitted)
    int8_t bit = bits;
    do {  // iterate over bytes of packed words
      __m256i r_lo = _mm256_setzero_si256();  // reset registers holding last byte of unpacked words
      __m256i r_hi = _mm256_setzero_si256();
      do {  // iterate over bits per byte of packed words
        __m256i b = _mm256_cvtepu8_epi32(_mm_loadl_epi64((__m128i*) &packed[i++]));
        __m256i b_lo = _mm256_shuffle_epi8(b, mask_shuffle_lo);  // convert bits to {0/-1} bytes
        __m256i b_hi = _mm256_shuffle_epi8(b, mask_shuffle_hi);
        b_lo = _mm256_cmpeq_epi8(_mm256_and_si256(b_lo, mask_bit), mask_bit);
        b_hi = _mm256_cmpeq_epi8(_mm256_and_si256(b_hi, mask_bit), mask_bit);
        r_lo = _mm256_sub_epi8(_mm256_add_epi8(r_lo, r_lo), b_lo);  // append bit to byte
        r_hi = _mm256_sub_epi8(_mm256_add_epi8(r_hi, r_hi), b_hi);
      } while (--bit & 7);
      r1 = _mm256_slli_epi32(r1, 8);  // shift the output registers left by 1 byte
      [...]  // repeat for r2, r3, ..., r8 (7 lines omitted)
      // merge 1 byte per code after up-converting first 8 out of 32 bytes per register
      r1 = _mm256_or_si256(r1, _mm256_cvtepu8_epi32(_mm256_castsi256_si128(r_lo)));
      r2 = _mm256_or_si256(r2, _mm256_cvtepu8_epi32(_mm256_castsi256_si128(r_hi)));
      r_lo = _mm256_permute4x64_epi64(r_lo, _MM_SHUFFLE(0,3,2,1));  // rotate for next 8/32 bytes
      r_hi = _mm256_permute4x64_epi64(r_hi, _MM_SHUFFLE(0,3,2,1));
      [...]  // repeat for rest 24 out of 32 bytes per register (18 lines omitted)
    } while (bit);
    _mm256_store_si256((__m256i*) &uncompressed[o], r1);  // store (unpacked) codes
    [...]  // repeat for r2, r3, ..., r8 (7 lines omitted)
  } while ((o = o + 64) != tuples);
}
```

### 4.3.4 Selection Scan

Predicate evaluation on vertically bit packed data can proceed directly on the packed format without unpacking. Equality to a constant is evaluated as the bitwise-`and` of the bitwise-`xnor` of all $b$ bits of the $k$ words with the bits of the constant. To evaluate less-than or greater-than, we find the highest order bit that the column value differs from the constant and copy the target bit of the constant, for less-than, or its inverse, for greater-than.

Using $w$-bit processor words, we need $O(nb/w)$ instructions to evaluate selective predicates with simple comparisons. However, we can avoid accessing all $b$ $w$-bit words per $k$

codes. Instead, once we have found a mismatching bit per code for *all* $w$ codes, we can stop without looking into the lower-order bits.

For the selective predicate $column = C$ ($C$ a query constant) the probability for one code to reach the $i$-th word, assuming uniform random data, is $2^{1-i}$. For *any* of the $w$ codes to reach the $i$-th word, the probability $P(X \geq i)$ is:

$$P(X \geq i) = 1 - (1 - 2^{1-i})^w$$

For the predicate $C_1 \leq column \leq C_2$, the probability for one code to reach the $i$-th word is $1 - (1 - 2^{1-i})^2$ and for *any* of the $w$ codes:

$$P(X \geq i) = 1 - (1 - [1 - (1 - 2^{1-i})^2])^w = 1 - (1 - 2^{1-i})^{2w}$$

The expected number of $w$-bit words accessed for $w$ codes using $b$ bits is:

$$E[X] = \sum_{i=1}^{b} P(X = i) \cdot i = \sum_{i=1}^{b} P(X \geq i)$$

```
void scan(const uint64_t* compressed, int8_t bits, size_t tuples,
    uint64_t* bitmap, uint32_t min, uint32_t max) {
  const int64_t mask_min = (int64_t) (min << (64 - bits));  // shift query constants ...
  const int64_t mask_max = (int64_t) (max << (64 - bits));  // placing target bit as sign bit
  size_t i = 0, o = 0, b = bits;
  do {
    uint64_t gt_min = 0;  // indicates if code is greater than the min constant
    uint64_t lt_max = 0;  // indicates if code is less than the max constant
    uint64_t eq_min = -1;  // indicates if code is (yet) equal to min constant
    uint64_t eq_max = -1;  // indicates if code is (yet) equal to max constant
    uint64_t min_bit = mask_min;  // used to broadcast bit of min constant
    uint64_t max_bit = mask_max;  // used to broadcast bit of max constant
    size_t j = 0;
    do {  // iterate over all bits
      uint64_t word = compressed[i + j];
      uint64_t neq_min_bit = word ^ (uint64_t) (min_bit >> 63);  // compare with broadcast sign bit
      uint64_t neq_max_bit = word ^ (uint64_t) (max_bit >> 63);
      gt_min |= neq_min_bit & min_bit & word;  // update greater-than and less-than bitmaps
      lt_max |= neq_max_bit & max_bit & ~word;
      eq_min &= ~neq_min_bit;  // update equality condition
      eq_max &= ~neq_max_bit;
      if ((eq_min | eq_max) == 0) break;  // early termination
      bit_min += bit_min;  // shift constants to bring next bit as sign bit
      bit_max += bit_max;
    } while (++j != b);
    bitmap[o++] = (gt_min | eq_min) & (lt_max | eq_max);  // combine comparison results
  } while (o != tuples);
}
```

Scalar code for scanning to evaluate $C_1 \leq column \leq C_2$ is shown above. The shifts by 63 broadcast the sign bit. Extending the code to use SIMD instructions is trivial if we use a larger $k$. In such a case, the number of instructions per code is reduced, but the register width $w$ is increased and so is the number of $w$-bit words accessed per $w$ codes.

Scanning less than $b$ words does not improve performance, due to cache line granularity of RAM accesses and hardware prefetching. Bitweaving [Li and Patel, 2013] addresses this problem by vertically packing $b' < b$ bits at a time separately. The most extreme approach is to store each bit separately [Wong *et al.*, 1985]. Here, we increase $k$ to place low-order and high-order bits of codes in distant cache lines.

Setting $k > w$ can be handled by all vertical bit packing operations with little overhead, if $w|k$. For packing, we store the $w$-bit packed words with a $k/w$ stride. For unpacking and scanning, we load the packed $w$-bit words with a $k/w$ stride. Processing the first $w$ codes out of $k$ would place packed words for the few next runs of $w$ codes in the L1 cache and would be loaded from the L1 in the next runs. When $k$ exceeds one cache line of $\mathcal{L}$ bits, in order to find the number of cache line accesses, we use the same $E[X]$ formula by setting $w = \mathcal{L}$. Also, to compensate for hardware prefetching of neighbor cache lines, we increase $k$ beyond one cache line up to the available L1 capacity. Finally, we can use SIMD for scanning without increasing the false prefetches.

When fewer tuples are extracted, performance differs when selectivity is low enough to skip cache lines. Then, vertical bit packing with $k \geq \mathcal{L}$ can be $O(b)$ times slower than horizontal, since we access 1 cache line per code bit rather than 1–2 cache lines per code.

## 4.4 Experimental Evaluation

The platform we use has one Intel Xeon E3-1275v3 CPU at 3.5 GHz based on the Haswell micro-architecture. The CPU has 4 cores with 2-way SMT with 256-bit `AVX2` SIMD. The RAM is dual-channel ECC DDR3 at 1600 MHz with 21.8 GB/s load bandwidth. We compile with ICC 15 using `-O3`. We use uniform random data since most methods are invariant to the input value distribution. Simple integer compression schemes that vary $b$ across groups of codes (e.g. frame-of-reference), are easy extensions adding negligible overhead.

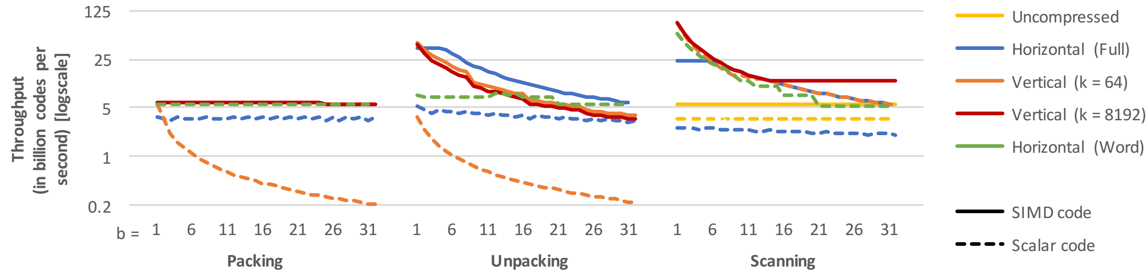



Figure 4.1: Throughput of packing, unpacking, and scanning for horizontal and vertical bit packing using scalar and SIMD code (multi-threaded)

In Figure 4.1, we measure the throughput of packing, unpacking, and scanning using both the horizontal and the vertical bit packing layout. We vary $b$ and use all hardware threads. If we materialize the output in memory, we saturate the RAM bandwidth for both packing and unpacking, thus, we exclude it and compute a checksum instead. For scanning, we evaluate the $C_1 \leq column \leq C_2$ predicate and materialize the ouput bitmap in memory.

Packing typically saturates the bandwidth by loading the unpacked input. Word-aligned horizontal packing is faster than the fully packed version. Packing to the vertical bit packed layout in SIMD code executes $O(nb)$ operations, but still saturates the bandwidth, even if $b = 32$. The vectorization speedup is 1.1–27.4X and is maximized for $b = 32$.

Unpacking the horizontal layout is faster than unpacking the vertical layout ($k = 64$) by 1.4–1.6X, if both are written in SIMD code, except for $b = 9$ and $b = 10$ that get a speedup of 1.8X and 1.7X respectively. The SIMD speedup is 1.9–8.4X for horizontal and 11–20X for the vertical layout. Increasing $k$ to 8192 makes unpacking 10–15% slower. For larger $b$, vertical unpacking in scalar code is prohibitively slow. For smaller $b$, unpacking one code at a time in scalar code is compute-bound.

Scanning the vertical layout in scalar code is faster than scanning the horizontal in SIMD by 1.1–3.4X for $b \leq 5$, while the opposite is true by 1.05–1.25X for $6 \leq b \leq 12$. For larger $b$, both methods saturate the bandwidth. The word-aligned layout loses performance by accessing a larger data footprint. Evaluating the predicates after horizontally unpacking in SIMD reduces performance by $\sim$ 30% for $b \leq 5$ down to $\sim$ 5% for $b > 16$. With $k = 64$, even if we access 8–9 words in 1–2 cache lines per 64 codes, we prefetch all cache lines.

By increasing $k$ beyond a single cache line, we process 10–11 bits per code, but minimize

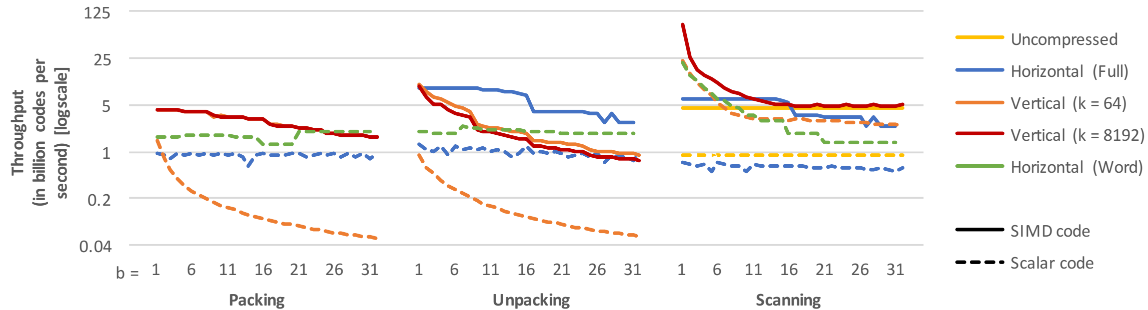



Figure 4.2: Throughput of packing, unpacking, and scanning for horizontal and vertical bit packing using scalar and SIMD code (single-threaded)

the false prefetches. Performance is constant for larger $b$ and outperforms all other methods. For $b = 32$, scanning the vertical layout effectively (logically) doubles the RAM bandwidth. Also, with $b = 32$, we can also encode the actual (32-bit) column values, not the dictionary codes, while still allowing selection scans to be executed twice as fast.

In Figure 4.2, we repeat the experiment of Figure 4.1 in a single thread to simulate higher memory bandwidth, making most methods compute-bound. Using multiple threads makes us memory-bound and reduces the gap between horizontal and vertical unpacking. Vertical scanning gets 1.9–3.6X speedup from SIMD and exposes the early exit for $b \geq 9$ if $k = 64$ and for $b \geq 12$ for $k = 8192$, matching the analysis of Section 4.3.4. Scanning uncompressed data gets a 5X speedup from SIMD and is close to the fastest, highlighting that lightweight compression is of limited use when we are not memory-bound.

## 4.5 Conclusion

We studied multiple bit packing layouts employed for light- weight compression. For horizontal bit packing, we considered both the word-aligned and the fully packed variant, and optimized previous techniques using the latest scalar and SIMD instructions. For vertical bit packing, we designed and implemented efficient packing and unpacking, by placing multiple codes in SIMD registers and by maximizing the number of bit movements per instruction. All variants are compared for packing, unpacking, and scanning. While the horizontal schemes are highly competitive, vertical bit packing combines the fastest scans with comparably fast packing and unpacking, extending the available trade-off options.

# Chapter 5

# Advanced SIMD Vectorization Techniques

## 5.1 Introduction

Real time analytics are the steering wheels of big data driven business intelligence. Database customer needs have extended beyond OLTP with high ACID transaction throughput, to interactive OLAP query execution across the entire database. As a consequence, vendors offer fast OLAP solutions, either by rebuilding a new DBMS for OLAP [Raman *et al.*, 2013; Stonebraker *et al.*, 2005], or by improving within the existing OLTP-focused DBMS.

The advent of large main-memory capacity is one of the reasons that blink-of-an-eye analytical query execution has become possible. Query optimization used to measure blocks fetched from disk as the primary unit of query cost. Today, the entire database can often remain main-memory resident and the need for efficient in-memory algorithms is apparent.

The prevalent shift in database design for the new era are column stores [Manegold *et al.*, 2000a; Raman *et al.*, 2013; Stonebraker *et al.*, 2005]. They allow for higher data compression in order to reduce the data footprint, minimize the number of columns accessed per tuple, and use column oriented execution coupled with late materialization [Abadi *et al.*, 2007] to eliminate unnecessary accesses to RAM resident columns.

Hardware provides performance through three sources of parallelism: *thread parallelism*, *instruction level parallelism*, and *data parallelism*. Analytical databases have evolved to

take advantage of all sources of parallelism. Thread parallelism is achieved, for individual operators, by splitting the input equally among threads [**?**; Balkesen *et al.*, 2013a; Blanas *et al.*, 2011; Cieslewicz *et al.*, 2010; Kim *et al.*, 2009; Satish *et al.*, 2010; Ye *et al.*, 2011], and in the case of queries that combine multiple operators, by using the pipeline breaking points of the query plan to split the materialized data in chunks that are distributed to threads dynamically [Leis *et al.*, 2014; Raman *et al.*, 2013]. Instruction level parallelism is achieved by applying the same operation to a block of tuples [Boncz *et al.*, 2005] and by generating and compiling query specific code, converting pipelined operators into tight nested loops [Krikellas *et al.*, 2010; Neumann, 2011]. Data parallelism is achieved individually for each operator by using SIMD instructions [Chhugani *et al.*, 2008; Kim *et al.*, 2010; Ross, 2007; Willhalm *et al.*, 2009; Zhou and Ross, 2002].

The different sources of hardware parallelism were developed as a means to deliver more performance within the same power budget available per chip. Mainstream CPUs have evolved regarding all sources of parallelism, featuring massively superscalar pipelines, out-of-order execution of tens of instructions, and advanced SIMD capabilities, all replicated across multiple cores on each CPU chip. For example, the Intel Haswell micro-architecture can issue up to 8 micro-operations per cycle, has 192 reorder buffer entries for out-of-order execution, supports 256-bit SIMD instructions, has two levels of private caches per core as well as a large shared cache, and scales up to 18 cores per chip.

Concurrently with the evolution of mainstream CPUs, a new approach on processor design has surfaced. The design, named the many-integrated-cores (MIC) architecture, uses cores with a smaller area (transistors) and power footprint by removing the massively superscalar pipeline, out-of-order execution, and the large L3 cache. Each core is based on a Pentium 1 processor with a simple in-order pipeline, but is augmented with wide SIMD registers, advanced SIMD instructions, and simultaneous multi-threading that is essential to hide the high load and instruction latency of SIMD instructions. Since each core has a smaller area and power footprint, more cores can be packed in the chip.

The MIC design was originally intended as a GPU [Seiler *et al.*, 2008], but now targets high performance computing applications. Using a high FLOPS machine to execute compute-intensive algorithms with superlinear complexity is self-evident. Executing ana-

lytical queries in memory, however, consists of data-intensive linear algorithms that mostly "move" rather than "process" data. Earlier work to add SIMD in database operators has either focused on sequential access operators such as index scans [Zhou and Ross, 2002], linear scans [Willhalm *et al.*, 2009], bucketized hash probes [Raman and Swart, 2006], multi-way trees with nodes that match the SIMD register layout [Kim *et al.*, 2010], or optimized specific operators, such as sorting [Chhugani *et al.*, 2008; Inoue *et al.*, 2007], by using *ad-hoc* SIMD vectorization techniques that are useful only for a specific problem.

Here, we present the basic design principles for extendible SIMD vectorization of in-memory database operators, without modifying the logic or the data layout of the baseline scalar algorithm. The baseline algorithm is defined here as the standard scalar implementation with the best time complexity. Formally, assume an algorithm that solves a problem with optimal complexity, a basic scalar implementation, and a SIMD vectorized implementation. We say that the algorithm can be *fully vectorized*, if the vectorized implementation executes $O(f(n)/\mathcal{W})$ SIMD instructions instead of $O(f(n))$ scalar instructions where $\mathcal{W}$ is the vector length, excluding random memory accesses that are by definition not data-parallel. We define fundamental vector operations that are frequently reused in the vectorizations and are implemented using advanced SIMD instructions, such as non-contiguous loads (*gathers*) and stores (*scatters*). The fundamental operations that are not directly supported by the hardware can be emulated at a performance penalty.

We implement vectorized operators in the context of main-memory databases: selection scans, hash tables, and partitioning, which are combined to build more advanced operators: sorting and joins. These operators cover a large portion of the time needed to execute analytical queries in main memory. For selection scans, we show branchless tuple selection and in-cache buffering. For hash tables, we study both building and probing across using multiple hashing schemes. For partitioning, we describe histogram generation, including all partitioning function types: radix, hash, and range. We also describe data shuffling, including inputs larger than the cache. Many of the above operators are combined to build sorting and hash join operators, in multiple variants, in order to highlight the impact of SIMD vectorization on determining the best algorithmic design for each operator.

We compare our vectorized implementations of in-memory database operators against

the respective state-of-the-art scalar and vector techniques, by evaluating on a Intel Xeon Phi co-processor codenamed Knights Corner (KNC) and on mainstream CPUs based on the Haswell micro-architecture. We use the sorting and join operator to compare an Intel Xeon Phi 7120P (KNC) co-processor (61 P54C cores at 1.238 GHz, 300 Watts TDP) against four high-end CPUs based on the Sandy Bridge micro-architecture ($4 \times 8$ Sandy Bridge cores at 2.2 GHz, $4 \times 130$ Watts TDP), and found that they have similar performance, but on a different power budget, since the co-processor spends almost half the energy.

The second generation of Intel Xeon Phi codenamed Knights Landing (KNL) is both available as a PCI-e co-processor and as a stand-alone CPU, and supports more advanced SIMD instructions (AVX-512) that are also supported by the latest mainstream CPUs. Here, we do not focus on evaluating Intel Xeon Phi platforms as co-processing accelerators bound by the PCI-e bandwidth, such as GPGPUs, but as alternative CPU designs that are both fast and power efficient for executing analytical database workloads.

We summarize our contributions for this chapter:

- We introduce design principles for efficient vectorization of in-memory database operators and define fundamental vector operations that are frequently reused.
- We design and implement vectorized selection scans, hash tables, and partitioning, that are combined to design and build sorting and multiple join variants.
- We compare our implementations against state-of-the-art scalar and vectorized techniques. We achieve up to an order of magnitude speedups by evaluating on Xeon Phi as well as on mainstream CPUs.
- We show the impact of vectorization on the algorithmic design of in-memory operators, as well as the architectural design and power efficiency of hardware, by making simple cores comparably fast to complex cores.

The rest of the chapter is organized as follows. In Sections 5.3, 5.4, 5.5, and 5.6, we discuss the vectorization of selection scans, hash tables, Bloom filters, and partitioning. In Section 5.7, we discuss algorithmic designs for sorting and multiple hash join variants. In Section 5.8, we present our experimental evaluation. In Section 5.9, we discuss how SIMD vectorization relates to SIMT in GPGPUs and we conclude in Section 5.10.

## 5.2 Fundamental Operations for Vectorization

### 5.2.1 Logical Design

We define the fundamental vector operations used vectorize database operators. The first two operations, termed *selective load* and *selective store*, are spatially contiguous memory accesses that load or store values using a *subset* of vector lanes. The last two operations are spatially non-contiguous memory loads and stores, termed *gathers* and *scatters* respectively.

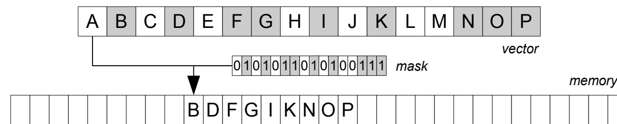



Figure 5.1: Selective store operation

Selective stores write a specific subset of the vector lanes to a memory location contiguously. The subset of vector lanes to be written is decided using a vector or scalar register as the mask, which must not be limited to a constant.

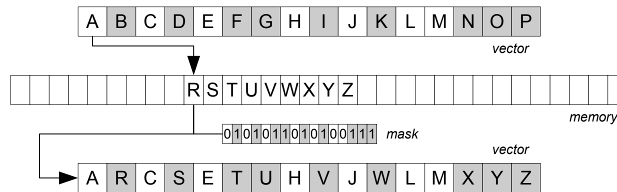



Figure 5.2: Selective load operation

Selective loads are the symmetric operation that involves loading from a memory location contiguously to a subset of vector lanes based on a mask. The lanes that are active in the mask are overwritten. The inactive lanes in the mask retain their previous values.

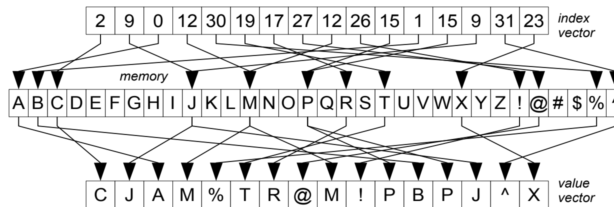



Figure 5.3: Gather operation

Gather operations load values from non-contiguous locations. The inputs are a vector of indexes and an array pointer. The output is a vector with the values of the respective array

cells. By adding a mask as an input operand, we define the selective gather that operates on a subset of lanes. The inactive mask lanes retain their previous contents.

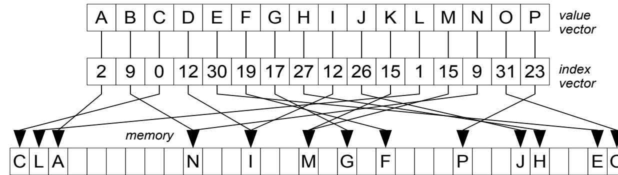



Figure 5.4: Scatter operation

Scatter operations store values to non-contiguous memory locations. The inputs are (i) a vector of indexes, (ii) the base pointer of the array, and (iii) a vector of values to store. If multiple vector lanes point to the same location, we assume that the rightmost value is eventually written. By adding a mask as an input we can store lanes selectively.

Gathers and scatters are not executed in parallel because the (L1) cache allows 1–2 accesses per cycle. Executing $\mathcal{W}$ cache accesses per cycle is an impractical hardware design. Thus, non-contiguous memory accesses are excluded from the $O(f(n)/\mathcal{W})$ rule. In fact, some hardware implementations for gather and scatter operations invoke $O(\mathcal{W})$ instructions.

Gathers are supported on mainstream CPUs based on the Haswell and the Skylake micro-architectures via the *AVX2* (256-bit) SIMD instruction set but scatters are not. CPUs based on the Sandy Bridge micro-architecture or older support neither. We can emulate gathers and scatters using scalar loads and stores respectively. The overhead may not be forbidding since hardware gathers and scatters are also $O(\mathcal{W}$ operations. Selective loads and stores are not directly supported in AVX2, but they can be implemented via vector permutations in $O(1)$ SIMD instructions. The latest CPUs directly support all four operations via the *AVX-512* (512-bit) SIMD instruction set.

### 5.2.2 Gather/Scatter Implementation

In the AVX2 256-bit SIMD instruction set, gathers are invoked via a single hardware instruction (`vpgatherdd`) loading up to 8 32-bit values in 8 32-bit SIMD lanes. On first generation Xeon Phi co-processors (KNC) with 512-bit SIMD (neither AVX2 nor AVX-512), the same instruction will access only one cache line each time we invoke it. To load values from multiple cache lines, we repeat until all $\mathcal{W}$ lanes are processed. If all memory

locations are in the same cache line, we invoke the instruction only once. If all memory locations are in different cache lines, we invoke the instruction once per lane.

An implementation for loading 16 32-bit values on a 512-bit SIMD register in a KNC co-processor is shown below. In this example, the `rax` scalar register holds the array pointer. The 512-bit `zmm0` SIMD register holds the indexes and `zmm1` holds the loaded values. `k1` is a 16-bit bitmask (or boolean vector) set by `kxnor`. Each call to `vpgatherdd` finds the leftmost set bit in `k1`, extracts the index of the respective 32-bit lane from `zmm0`, identifies all other 32-bit lanes that point to the same cache line, loads the values from the cache line in `zmm1`, and resets the respective bits in `k1`. The `jknzd` branches back to re-invoke the instruction until `k1` has no set bits. Note that gathers in AVX-512 need no such loop.

```
      kxnor        %k1, %k1
loop: vpgatherdd   (%rax, %zmm0, 4), %zmm1 {%k1}
      jknzd        loop, %k1
```

Scatters are implemented similarly to gathers, although they are not supported in AVX2. During a scatter operation, if multiple lanes of the index vector point to the same location, the value in the rightmost lane is written last overwriting any values from other lanes. In any case, all respective bits in `k1` are reset. In KNC where the scatter operations are implemented as loops of scatter instructions, we can detect multiple references to the same cache line by checking if multiple bits of `k1` were reset, but we cannot distinguish between conflicts and distinct writes in the same cache line.

Non-selective gathers and scatters first set all bits in `k1`. Selective gathers and scatters use `k1` and `zmm1` as both inputs and outputs to load a subset of lanes determined by `k1`. The rest lanes in `zmm1` not set in `k1` are unaffected. In AVX2, `k1` is also a 256-bit vector where the highest bit denotes whether the lane is active. In AVX-512, `k1` is a bitmask.

Both latency and throughput of gathers and scatters is determined by the number of cache lines accessed. Assuming all cache lines are in L1, the operation is faster if the locations are assembled in fewer cache lines. On KNC, gathering values to fewer lanes from the same cache line reduces the latency of the instruction [Hofmann *et al.*, 2014]. On the other hand, executing multiple gather and scatter operations in the same loop by invoking multiple (KNC) gather and scatter instructions offers no performance benefit.

Gathers and scatters can be emulated via scalar loads and stores. We did a careful

implementation of gathers using other vector instructions and used it in linear probing and double hashing tables. We found that we lose up to 13% probing performance compared to the hardware gathers, which reduces as the table size increases. When the loads are not cache resident, there is no noticeable difference, which is expected since gathers and scatters go through the same memory access path as loads and stores to a single location.

### 5.2.3 Implementing Selective Loads / Stores

In AVX-512, selective loads and stores are implemented via a single SIMD instruction that issues an unaligned (byte-aligned) load or store to memory. For instance, the `vpexpandd` instruction is a selective load of 32-bit lanes. The `vpcompressd` instruction is a selective store of 32-bit SIMD lanes. KNC only supports aligned memory accesses. Selective loads and stores are split to two instructions, once for the lower and one for the upper cache line, assuming the selective load spans across two cache lines. The implementation of the selective load operation for KNC co-processors assuming 32-bit SIMD lanes is shown below.

```
void _mm512_mask_packstore_epi32(int32_t* ptr, __mmask16 mask, __m512i vec) {
  _mm512_mask_packstorelo_epi32(&ptr[0],  mask, vec):
  _mm512_mask_packstorehi_epi32(&ptr[16], mask, vec); }
```

The symmetric selective store implementation for KNC co-processors is shown below.

```
__m512i _mm512_mask_loadunpack_epi32(__m512i vec, __mmask16 mask, const int32_t* ptr) {
  vec = _mm512_mask_loadunpacklo_epi32(vec, mask, &ptr[0]):
  vec = _mm512_mask_loadunpackhi_epi32(vec, mask, &ptr[16]);
  return vec; }
```

In AVX2, selective loads and stores are not mapped to specific instructions. The basic SIMD operation we need is the ability to shuffle a register using lane indexes provided by another register. For example, if our vector of values is `[A,B,C,D,E,F,G,H]`, and the boolean vector of active lanes is `[0,1,0,1,0,1,1,1]`, we re-order the input vector to `[B,D,F,G,H,A,C,E]`. For this operation, we need the permutation vector `[1,3,5,6,7,0,2,4]`. The permutation mask is loaded from a lookup table using the boolean vector bitmask as an index (`11101010` $= 234$ here). If the SIMD register has $\mathcal{W}$ lanes, the lookup table needs $2^{\mathcal{W}}$ entries. In AVX2, assuming 32-bit lanes ($\mathcal{W} = 8$), we need $2^8 = 256$ permutation vectors. We store each vector using 8 bytes. Thus, the lookup table has a footprint of 2 KB and remains L1 resident. For 64-bit lanes ($\mathcal{W} = 4$), the lookup table is merely 128 bytes.

### 5.2.4 Algorithmic Notation

We now explain the notation we use for vectorized algorithmic descriptions in this chapter. Vector operations with boolean results generate bitmasks (boolean vectors) as output, which are regular scalar variables. For example, $m \leftarrow \vec{x} < \vec{y}$ generates bitmask $m$. The $m \leftarrow$ true assignment sets all $\mathcal{W}$ bits of $m$. Vectors used as array indexes denote a gather or a scatter. For example, $\vec{x} \leftarrow A[y]$ is a contiguous vector load, while $\vec{x} \leftarrow A[\vec{y}]$ denotes a gather. Selective loads and stores as well as selective gathers and scatters use a bitmask as a subscript in the assignment. For example, $\vec{x} \leftarrow_m A[y]$ is a selective load, while $\vec{x} \leftarrow_m A[\vec{y}]$ is a selective gather. $|m|$ denotes the bit population count of $m$ and $|T|$ is the cardinality of table $T$. If-then-else statements, such as $\vec{x} \leftarrow m \ ? \ \vec{y} : \vec{z}$, use vector blending. Finally, scalar values in vector expressions assume constant vectors generated out of the main loop. For example, $\vec{x} \leftarrow \vec{x} + k$ and $m \leftarrow \vec{x} > k$ uses a vector with $k$ broadcast to all $\mathcal{W}$ lanes.

## 5.3 Selection Scans

Linear scans are replacing traditional unclustered indexes as the means of executing selections in modern in-memory DBMSs [Raman *et al.*, 2013]. Advanced optimizations for selection scans include using lightweight bit compression [Willhalm *et al.*, 2009] to reduce the data footprint and save RAM bandwidth, generating statistics to skip data regions [Raman *et al.*, 2013], and scanning of bitmaps or zonemaps to skip cache lines [Sidirourgos and Kersten, 2013]. Nevertheless, basic linear scans remain the core operation of selections. We show a simple example of a selection scan using the naive branching approach. We assume $X \leq T_k \leq Y$ is the selective predicate and $X$, $Y$ are query constants.

**Algorithm 17:** Selection Scan (Scalar - Branching)

```
1 i, o ← 0                                  // input and output index
2 do
3     k ← T_k[i]                            // load key columns from input
4     if k ≥ X and k ≤ Y then
5         T'_k[o] ← k                       // store key columns to the output
6         T'_v[o] ← T_v[i]                  // copy payload columns to output
7         o ← o + 1
8     i ← i + 1
9 while i ≠ |T|
```

**Algorithm 18:** Selection Scan (Scalar - Branchless)

```
1  i, o ← 0                                   // input and output index
2  do
3      k ← T_k[i]                             // load key columns from input
4      v ← T_v[i]                             // load payload columns from input
5      i ← i + 1
6      T'_k[o] ← k                            // store key columns to output (may be overwritten)
7      T'_v[o] ← v                            // store payload columns to output
8      m_1 ← k ≥ X ? 1 : 0                    // get the boolean result as {0,1} conditional flag
9      m_2 ← k ≤ Y ? 1 : 0                    // get the boolean result as {0,1} conditional flag
10     o ← o + (m_1 & m_2)                    // update the output index without branching (bitwise-and)
11 while i ≠ |R|
```

The performance of linear scans for selections has been associated with branch mispredictions, given any implementation similar to Algorithm 17. [Ross, 2004] showed that converting control flow to data flow can improve performance by removing branches to eliminate branch mispredictions. Nevertheless, if the selectivity is either too low or too high, branch mispredictions are rare and branches are preferable. As a result, the best approach depends on the selectivity. Algorithm 18 shows how branches can be eliminated altogether. However, since we fully eliminate branches inside the main loop, we have to access both key and payload columns and evaluate all selective predicates eagerly for every tuple.

Vectorized selection scans use selective stores to store the lanes that satisfy the selection predicates. We use SIMD instructions to evaluate the predicates resulting in a bitmask of the qualifying lanes. Partially vectorized selection extracts one bit at a time from the bitmask and accesses the corresponding tuple. Instead, we use the bitmask to selectively store the qualifying tuples to the output vector at once.

When the selectivity is low, we should avoid accessing payload columns to save memory bandwidth. Note that when the branch is speculatively executed, the hardware may still needlessly load payloads incurring the bandwidth cost. Instead, we use a small cache resident buffer to store the indexes of qualifying tuples (rids) instead of the actual tuples. When the buffer is full, we reload the rids from the buffer, gather the tuples from the base columns, and flush to the output. This approach is shown in Algorithm 19.

When we store data on RAM with no intent to reuse it soon, we use non-temporal stores to save bandwidth. Mainstream CPUs provide non-temporal stores that bypass the higher cache levels and increase the RAM bandwidth for storing data. KNC co-processors do not

support non-temporal stores in scalar code, but provide an instruction to overwrite a cache line with a vector without loading the cache line first. This technique requires the vector length to match the cache line but eliminates the need for write-combining buffers.

The code for selection scans assuming 32-bit keys and payloads in KNC. The selective condition is $X \leq T_k \leq Y$. The buffer must be small enough to be L1 resident. The analog of non-temporal stores on KNC co-processors is the `vmovnrngoaps` instruction (with intrinsic `_mm512_storenrngo_ps`) that overwrites a complete cache line in RAM without first loading it in the cache. For all such mechanisms to work, the input and output columns must be aligned in cache line boundaries. The same loop is almost identical in AVX-512 code. In AVX2, the implementation of selective loads and stores is based on permutations.

**Algorithm 19:** Selection Scan (Vector)

```
1  r⃗ ← {0, 1, 2, 3, ..., W − 1}                    // input indexes (rids) in vector
2  i, o, b, b'' ← 0                                 // input, output, and buffer index
3  do
4  |  k⃗ ← T_k[i]                                    // load vectors of key columns
5  |  m ← (k⃗ ≥ X) & (k⃗ ≤ Y)                         // predicates to mask
6  |  B[b] ←_m r⃗                                    // (selective) store rids to buffer
7  |  b ← b + |m|
8  |  if b > |B| − 𝒲 then
9  |  |  b' ← 0
10 |  |  do
11 |  |  |  i⃗' ← B[b']                              // load input indexes from buffer
12 |  |  |  k⃗ ← T_k[r⃗']                             // dereference (gather) column values
13 |  |  |  v⃗ ← T_v[r⃗']
14 |  |  |  T'_k[o] ← k⃗                             // flush to output using non-temporal stores
15 |  |  |  T'_v[o] ← v⃗
16 |  |  |  b' ← b' + 𝒲
17 |  |  |  o ← o + 𝒲                               // update output index
18 |  |  while b' ≠ |B| − W
19 |  |  p⃗ ← B[b']                                  // move extra indexes to front of buffer
20 |  |  B[0] ← p⃗
21 |  |  b ← b − b'                                 // update buffer index
22 |  r⃗ ← r⃗ + 𝒲                                     // update vector of rids
23 |  i ← i + 𝒲
24 while i ≠ |T|
25 while b'' ≠ b do
26 |  i ← B[b'']                                    // flush items remaining in buffer
27 |  b' ← b'' + 1
28 |  T'_k[o] ← T_k[i]
29 |  T'_v[o] ← T_v[i]
30 |  o ← o + 1
```

```
int32_t buffer_rids[BUF_SIZE] __attribute__((aligned(64)));  // aligned buffer to store rids
__m512i rid = _mm512_set_epi32(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0);
size_t i = 0, o = 0, b = 0;
do {
  __m512i key = _mm512_load_epi32(&input_keys[i]);  // load key column
  __mmask16 k = _mm512_cmpge_epi32_mask(key, mask_X);  // evaluate column >= X
  k = _mm512_mask_cmple_epi32_mask(k, key, mask_Y);  // evaluate and merge column <= Y
  _mm512_mask_packstore_epi32(&buffer_rids[j], k, rid);  // (selective) store to buffer
  b += _mm_countbits_64(_mm512_mask2int(m));  // update the buffer index
  if (b > BUF_SIZE - 16) {  // flush the buffer
    size_t bb = 0;
    do {
      __m512i rid_b = _mm512_load_epi32(&buffer_rids[bb]);  // load back rids from buffer
      __m512 key_b = _mm512_i32gather_ps(rid_b, input_keys, 4);  // dereference 32-bit keys
      __m512 val_b = _mm512_i32gather_ps(rid_b, input_vals, 4);  // dereference 32-bit payloads
      _mm512_storenrngo_ps(&output_keys[bb + o], key_b);  // (non-temporal) store keys to output
      _mm512_storenrngo_ps(&output_vals[bb + o], pay_b);  // (non-temporal) store payloads to output
    } while ((bb = bb + 16) != BUF_SIZE - 16);
    _mm512_store_epi32(rids_buf, _mm512_load_epi32(&rids_buf[b]));  // move to the front of buffer
    o += bb, b -= bb;  // update the output and the buffer index
  }
  rid = _mm512_add_epi32(rid, mask_16);  // update the rids
} while ((i = i + 16) != tuples);
[...]  // flush tuples from the buffer
```

## 5.4 Hash Tables

Hash tables are essential for the in-memory execution of joins and group-by aggregations since they allow for constant time key lookups. In hash joins, the "inner" relation is used to build a hash table and the "outer" relation probes the hash table to find matching tuples by hashing the join keys. In group-by aggregation, hash tables are used either to map tuples to unique group ids or to update partial aggregates by hashing the group-by attributes.

SIMD instructions have been proposed as a way to probe *bucketized* hash tables [Ross, 2007], i.e. hash tables with multiple keys and payloads per bucket. Instead of comparing against a single key at a time, we place multiple keys per bucket and compare them to the probing key with SIMD instructions. We use the term *horizontal vectorization* to describe the approach of comparing a single input key with multiple keys per bucket. Scalar loads should be as fast as contiguous vector loads, thus, the cost of bucketized probing diminishes to extracting the right payload. [Ross, 2007] showed that bucketized hash tables based on cuckoo hashing [Pagh and Rodler, 2004] can support load factors close to 100%.

Horizontal vectorization, if we expect to probe fewer than $\mathcal{W}$ buckets on average per key, is wasteful. For example, a 50% full hash table with one match per key accesses $\approx 1.5$ buckets on average to find the match if we use linear probing. In such a case, comparing one input key against multiple keys in the hash table bucket cannot yield high performance improvement and takes no advantage of the increasing SIMD register size.

Here, we propose a generic form of hash table vectorization termed *vertical vectorization* that can be applied to any hash table variant without altering the hash table layout. The fundamental principle is to process a different input key per vector lane. All vector lanes process different keys from the input and access different hash table locations.

The variants we discuss are linear probing (Section 5.4.1), double hashing (Section 5.4.2), and cuckoo hashing (Section 5.4.3). For the hash function, we use multiplicative hashing, which requires two multiplications, or for $2^n$ buckets, a multiplication a one shift instruction. Multiplication costs very few cycles in modern CPUs and is supported in SIMD.

### 5.4.1 Linear Probing

Linear probing is an open addressing scheme that traverses the table linearly until an empty bucket is found. In the hash table buckets, we store keys and payloads but no pointers. The scalar code for probing the hash table is shown in Algorithm 20.

**Algorithm 20:** Linear Probing - Probe (Scalar)

```
1  i, j ← 0                                    // input and output index
2  do
3  |  k ← S_k[i]                               // load key and payload from outer (probing) relation
4  |  v ← S_v[i]
5  |  i ← i + 1
6  |  h ← (k · f) ×↑ |H|                       // multiplicative hashing
7  |  while H_k[h] ≠ k_E do
8  |  |  if k = H_k[h] then
9  |  |  |  T_Rv[o] ← H_v[h]                   // (joined) inner values
10 |  |  |  T_Sv[o] ← v                        // (joined) outer values
11 |  |  |  T_k[o] ← k                         // (joined) keys
12 |  |  |  o ← o + 1
13 |  |  h ← h + 1                             // go to next bucket (linear probing)
14 |  |  if h = |H| then
15 |  |  |  h ← 0                              // reset if last bucket
16 while i ≠ |S|
```

**Algorithm 21:** Linear Probing - Probe (Vector)

1. $i, o \leftarrow 0$ // input and output index ($o$ and $\vec{o}$ are different variables)
2. $m \leftarrow$ true // use all vector lanes
3. **do**
4. $\quad \vec{k} \leftarrow_m S_k[i]$ // (selective) load new outer keys
5. $\quad \vec{v} \leftarrow_m S_v[i]$ // (selective) load new outer payloads
6. $\quad i \leftarrow i + |m|$
7. $\quad \vec{h} \leftarrow (\vec{k} \cdot f) \times\uparrow |H|$ // multiplicative hashing
8. $\quad \vec{o} \leftarrow m\ ?\ 0 : (\vec{o} + 1)$ // increment (or reset) linear probing offsets
9. $\quad \vec{h} \leftarrow \vec{h} + \vec{o}$ // add linear probing offsets
10. $\quad m \leftarrow \vec{h} < |H|$ // find overflowing lanes
11. $\quad \vec{h} \leftarrow m\ ?\ \vec{h} : (\vec{h} - |H|)$ // fix overflows (vector blend)
12. $\quad \vec{k}_H \leftarrow H_k[\vec{h}]$ // gather keys from hash table buckets
13. $\quad \vec{v}_H \leftarrow H_v[\vec{h}]$ // gather payloads from hash table buckets
14. $\quad m \leftarrow \vec{k}_H = \vec{k}$ // find matching keys
15. $\quad T_k[o] \leftarrow_m \vec{k}$ // (selective) store matching keys
16. $\quad T_{Sv}[o] \leftarrow_m \vec{v}$ // (selective) store outer payloads for matches
17. $\quad T_{Rv}[o] \leftarrow_m \vec{v}_H$ // (selective) store inner payloads for matches
18. $\quad o \leftarrow o + |m|$
19. $\quad m \leftarrow \vec{k}_H = k_E$ // reuse lanes that reached an empty bucket
20. **while** $i + |m| \leq |S_k|$

The vectorized procedure for probing a hash table assuming linear probing for collisions is shown in Algorithm 21. Our vectorization principle is to process a different key per SIMD lane using gathers to access the hash table. Assuming $\mathcal{W}$ vector lanes, we process $\mathcal{W}$ different input keys on each loop. Instead of using a nested loop to find all matches for the $\mathcal{W}$ keys before loading the next $\mathcal{W}$ keys, we reuse vector lanes as soon as we know there are no more matches in the table, by (selective) loading new keys from the input to replace finished keys. Thus, each key executes the same number of loops as in scalar code. Every time a match is found, we use selective stores to write to the output the vector lanes that have matches. In order to support each key having executed an arbitrary number of loops already, we keep a vector of offsets that maintain how far each key has searched in the table. When a key is overwritten, the offset is reset to zero.

A simpler approach would be to process $\mathcal{W}$ keys at a time and use a nested loop to find all matches. However, the inner loop would be executed as many times as the maximum number of buckets accessed by any one of the $\mathcal{W}$ keys, underutilizing the SIMD lanes, since the average number of accessed buckets of $\mathcal{W}$ keys can be significantly smaller than the maximum. By reusing SIMD lanes dynamically, we are reading the input "out-of-order".

```
__mmask16 k = _mm512_int2mask(0xFFFF);  // set to use all vector lanes initially
__m512i key, val, off;  // state (keys, payloads, and offsets)
size_t i = 0, j = 16, o = 0;
do {  // process 16 tuples at a time in vectorized code
  key = _mm512_mask_loadunpack_epi32(key, k, &outer_keys[i]);  // load new keys reusing vector lanes
  val = _mm512_mask_loadunpack_epi32(val, k, &outer_vals[i]);  // load new payloads
  i += j;  // increment input using the number of reused lanes
  __m512i loc = _mm512_mullo_epi32(key, mask_factor);  // hash keys (multiplicative hashing)
  loc = _mm512_mulhi_epu32(loc, mask_buckets);
  off = _mm512_mask_xor_epi32(off, k, off, off);  // reset linear probing offset for reused lanes
  loc = _mm512_add_epi32(loc, off);  // add linear probing offset
  off = _mm512_add_epi32(off, mask_1);  // increment linear probing offsets
  k = _mm512_cmpge_epi32_mask(loc, mask_buckets);  // find overflowing lanes
  loc = _mm512_mask_sub_epi32(loc, k, loc, mask_buckets);  // fix overflowing lanes
  __m512i key_H = _mm512_i32gather_epi32(loc, &hash_table[0].key, 8);  // gather hash bucket keys
  __m512i val_H = _mm512_i32gather_epi32(loc, &hash_table[0].val, 8);  // gather payloads
  k = _mm512_cmpeq_epi32_mask(key, key_H);  // find matching lanes
  _mm512_mask_packstore_epi32(&join_keys[j], k, key);  // (selective) store join keys to output
  _mm512_mask_packstore_epi32(&join_outer_vals[j], k, val);  // (selective) store outer payloads
  _mm512_mask_packstore_epi32(&join_inner_vals[j], k, val_H);  // (selective) store inner payloads
  o += _mm_countbits_64(_mm512_mask2int(k));  // update the output index
  k = _mm512_cmpeq_epi32_mask(key_H, mask_empty);  // find which lanes reached an empty bucket
  j = _mm_countbits_64(_mm512_mask2int(k));  // count the reusable lanes
} while (i + j <= tuples);  // check if at least 16 tuples remain to process in SIMD code
[...]  // process the final (15 or less) tuples in scalar code
```

The KNC implementation for hash table probing is shown above. The input has 32-bit keys and payloads and each hash table bucket has a 32-bit key and a 32-bit payload.

Building a hash table assuming linear probing to deal with collisions is similar to probing. In order to insert a new tuple, we iterate over buckets starting from the hash location until we reach an empty bucket. The standard scalar approach is shown in Algorithm 22.

**Algorithm 22:** Linear Probing - Build (Scalar)

1 $i \leftarrow 0$ // input index
2 **do**
3 $\quad k \leftarrow R_k[i]$ // load inner keys
4 $\quad v \leftarrow R_v[i]$ // load inner payloads
5 $\quad i \leftarrow i + 1$
6 $\quad h \leftarrow (k \cdot f) \times\uparrow |H|$ // multiplicative hashing
7 $\quad$ **while** $H_k[h] \neq k_E$ **do**
8 $\quad\quad h \leftarrow h + 1$ // go to next hash bucket (linear probing)
9 $\quad\quad$ **if** $h = |H|$ **then**
10 $\quad\quad\quad h \leftarrow 0$ // reset to first hash bucket if last
11 $\quad H_k[h] \leftarrow k$ // set hash table keys
12 $\quad H_v[h] \leftarrow v$ // set hash table payloads
13 **while** $i \neq |R|$

**Algorithm 23:** Linear Probing - Build (Vector)

```
1  l⃗ ← {1, 2, 3, ..., W}                    // vector with unique values per lane
2  i ← 0
3  m ← true                                  // use all vector lanes
4  do
5      k⃗ ←_m R_k[i]                          // (selective) load new inner keys
6      v⃗ ←_m R_v[i]                          // (selective) load new inner payloads
7      i ← i + |m|
8      h⃗ ← (k · f) ×↑ |H|                    // multiplicative hashing
9      o⃗ ← m ? 0 : (o⃗ + 1)                   // increment or reset linear probing offsets
10     h⃗ ← h⃗ + o⃗                             // add linear probing offsets
11     m ← h⃗ < |H|                           // fix overflows
12     h⃗ ← m ? h⃗ : (h⃗ − |H|)
13     k⃗_H ← H_k[h⃗]                          // gather keys from hash buckets
14     m ← k⃗_H = k_E                         // find empty hash buckets
15     T[h⃗] ←_m l⃗                            // detect scatter conflicts
16     l⃗_H ←_m H_k[h⃗]
17     m ← m & (l⃗ = l⃗_H)
18     H_k[h⃗] ←_m k⃗                          // scatter inner keys to hash buckets
19     H_v[h⃗] ←_m v⃗                          // scatter inner payloads to hash buckets
20 while i + |m| ≤ |R|
```

The vector code for building a hash table is shown in Algorithm 23. The basics of vectorized probe and build of linear probing hash tables are the same. We process different input keys per vector lane and on top of gathers, we now also use scatters to store the keys non-contiguously. We access the input "out-of-order" to reuse lanes as soon as keys are inserted. To insert tuples, we first gather to check if the buckets are empty and then scatter the tuples only if the bucket is empty. The tuples that accessed a non-empty bucket increment an offset vector in order to search the next bucket in the next loop. We assume a bucket is empty if the key is equal to the special value $k_E$.

In order to ensure that multiple tuples will not try to fill the same empty bucket, we add a conflict detection step before scattering the tuples. Two lanes are conflicting if they point to the same location. However, we do not need to identify both lanes but rather the leftmost one that would get its value overwritten by the rightmost during the scatter. To identify these lanes, we scatter arbitrary values using a vector with unique values per lane (e.g., $[1, 2, 3, \ldots, \mathcal{W}]$). Then, we gather using the same index vector. If the gathered value matches the (unique) scattered value, we can safely scatter to that lane. The conflicting lanes are handled like non-matching lanes; they probe the next bucket in the following loop.

If the input keys are unique (e.g., join on a candidate key), we can scatter the keys to the table and gather them back to find the conflicting lanes instead of a constant vector with unique values per lane, saving one scatter operation. Also, the AVX-512 SIMD instruction set provides instructions to implement conflict detection directly in registers with $O(1)$ instructions. For instance, the `vpconflictd` instruction performs an all-to-all comparison across all 32-bit lanes of the SIMD register. Thus, we can save the extra scatter and gather.

The hash table uses a row-oriented format storing interleaved keys and values per bucket. To reduce the number of cache accesses, we can pack multiple gathers into fewer wider gathers. For example, assuming 32-bit keys and 32-bit payloads, the two 16-way 32-bit gathers can be replaced with two 8-way 64-bit gathers. The same applies to scatters.

For both probing and building, selective loads and stores assume there are enough items in the input to saturate the vector register. To process the last items in the input, we switch to scalar code. The last items are bounded in number by $\mathcal{W}$, which is negligible compared to the total number of input tuples. Thus, the overall throughput is unaffected.

The KNC implementation for building a linear probing hash table with 32-bit keys and payloads is shown above. Conflict detection uses gathers and scatters as in Algorithm 23.

```
__mmask16 k = _mm512_int2mask(0xFFFF);  // set to use all vector lanes initially
__m512i key, val, off;  // state (keys, payloads, and offsets)
size_t i = 0, j = 16;
do {  // process 16 input tuples at a time
  key = _mm512_mask_loadunpack_epi32(key, k, &inner_keys[i]);  // load new keys reusing vector lanes
  val = _mm512_mask_loadunpack_epi32(val, k, &inner_vals[i]);  // load new payloads
  i += j;  // increment input using the number of reused lanesxr
  __m512i loc = _mm512_mullo_epi32(key, mask_factor);  // hash keys (multiplicative hashing)
  loc = _mm512_mulhi_epu32(loc, mask_buckets);
  off = _mm512_mask_xor_epi32(off, k, off, off);  // reset linear probing offset for reused lanes
  loc = _mm512_add_epi32(loc, off);  // add linear probing offsets
  off = _mm512_add_epi32(off, mask_1);  // increment linear probing offsets
  k = _mm512_cmpge_epi32_mask(off, mask_buckets);  // find overflowing lanes
  off = _mm512_mask_sub_epi32_mask(off, k, off, mask_buckets);  // fix overflowing lanes
  __m512i key_H = _mm512_i32gather_epi32(loc, &hash_table[0].key, 8);  // gather keys
  k = _mm512_cmpeq_epi32_mask(key_H, mask_empty);  // check if buckets are empty
  _mm512_mask_i32scatter_epi32(&hash_table[0].key, k, loc, mask_unique, 8);  // detect conflits
  key_H = _mm512_mask_i32gather_epi32(key_H, k, loc, table, 8);
  k = _mm512_mask_cmpeq_epi32_mask(k, key_H, mask_unique);
  _mm512_mask_i32scatter_epi32(&hash_table[0].key, k, loc, key, 8);  // scatter keys
  _mm512_mask_i32scatter_epi32(&hash_table[0].val, k, loc, val, 8);  // scatter payloads
  j = _mm_countbits_64(_mm512_mask2int(k));  // count number of reused lanes
} while (i + j <= tuples);  // check if at least 16 tuples remain to process in SIMD code
[...]  // process last (15 or less) tuples in scalar code
```

### 5.4.2 Double Hashing

Duplicate keys in hash tables can be handled by storing the payloads in a separate table, or by repeating the keys. The first approach works well when most matching keys are repeated. The second approach works well with mostly unique keys, but suffers from clustering duplicate keys in the same region, if linear probing is used. Double hashing uses a second hash function to distribute collisions so that the number of accessed buckets is close to the number of true matches. We can use the latter approach in both cases. Comparing multiple hash table layouts based on the number of repeats is not the focus of this work.

In double hashing, the primary hash function $h_1$ always takes values in $[0, b)$ for $b$ buckets. For $0 \leq i < b$ collisions, the hash bucket locations are given by the formula $h = (h_1 + i \cdot h_2) \bmod b$. If $b = 2^n$ and $h_2$ is odd, then $h$ takes *every* value in $[0, b)$ for $0 \leq i < b$ and the same bucket is *never* repeated. Because enforcing $b = 2^n$ may waste too much space, we can ensure the same property if $b$ is prime. If $h_2$ takes values in $[1, b)$, then $h_2$ is always co-prime with $b$ and $h$ also takes every value in $[0, b)$. Since each bucket is unique within $[0, b)$, we can avoid modulos and fix the overflows by subtracting $b$ from $h$ when $h \geq b$. Algorithm 24 describes scalar hash table probing using this double hashing scheme.

**Algorithm 24:** Double Probing - Probe (Scalar)

```
1  i, o ← 0
2  do
3      k ← S_k[i]                                  // load outer key
4      v ← S_v[i]                                  // load outer payload
5      i ← i + 1
6      h ← (k · f_1) ×↑ |H|                        // primary hash function (|H| is a prime)
7      if H_k[h] ≠ k_E then
8          h' ← (k · f_2) ×↑ (|H| − 1) + 1          // secondary hash function
9          do
10             if k = H_k[h] then
11                 T_Rv[o] ← H_vals[h]             // store inner payload
12                 T_Sv[o] ← v                     // store outer payload
13                 T_k[o] ← k                      // store (joined) key
14                 o ← o + 1
15             h ← h + h'                          // combine primary and secondary function
16             if h ≥ |H| then
17                 h ← h − |H|                     // fix overflow
18         while H_k[h] ≠ k_E
19 while i ≠ |S|
```

**Algorithm 25:** Double Hashing - Probe (Vector)

```
1  i, o ← 0
2  m ← true                                              // use all vector lanes
3  do
4  |  k⃗ ←_m S_k[i]                                       // (selective) load new outer keys
5  |  v⃗ ←_m S_v[i]                                       // (selective) load new outer keys
6  |  i ← i + |m|
7  |  f⃗ ← m ? f_1 : f_2                                  // pick multiplicative factor
8  |  b⃗ ← m ? |H| : (|H| − 1)                            // pick number of buckets
9  |  h⃗' ← (k⃗ × ↓ f⃗) × ↑ b⃗
10 |  h⃗ ← m ? h⃗' : (h⃗ + h⃗' + 1)                        // combine primary and secondary function
11 |  m ← h⃗ < |H|                                        // find overflows
12 |  h⃗ ← m ? h⃗ : (h⃗ − |H|)                              // fix overflows
13 |  k⃗_H ← H_k[h⃗]                                       // gather keys from hash table buckets
14 |  v⃗_H ← H_v[h⃗]                                       // gather payloads from hash table buckets
15 |  m ← k⃗_H = k⃗                                        // find matching keys
16 |  T_k[o] ←_m k⃗                                       // (selective) store keys for matches
17 |  T_Sv[o] ←_m v⃗                                      // (selective) store outer payloads for matches
18 |  T_Rv[o] ←_m v⃗_T                                    // (selective) store inner payloads for matches
19 |  o ← o + |m|
20 |  m ← k⃗_H = k_E                                       // reuse lanes that reached an empty bucket
21 while i + |m| ≤ |S|
```

Algorithm 25 shows the double hashing scheme in vectorized code. Here, $m$ is the subset of vector lanes that have probed at least one bucket. Linear probing uses the linear probing offsets $\vec{o}$ as part of the state, double hashing uses the last hash bucket locations $\vec{h}$ instead. To switch between the primary and the secondary hash function, we simply switch to a different multiplicative hash factor. To ensure that the hash functions are pairwise independent, we pick random odd multiplicative hash factors that satisfy: $(f_1 - f_2) \mod 4 = 2$.

In each iteration of the vectorized implementation, we either compute the primary or the secondary hash function for each input key in the SIMD vector but not both. To use stronger hash functions (e.g. to avoid flooding), the two functions must use the same instructions but remain pairwise independent. We can either generate multiple instances by changing constants or by initializing with different seeds.

Building the hash table with double hashing is largely equivalent to linear probing. We omit showing the algorithm since the modification can be derived by comparing Algorithms 21 and 25. For completeness, we show the KNC implementation of probing a double hashing table below. The changes compared to linear probing are minimal.

```
__mmask16 k = _mm512_int2mask(0xFFFF);  // set to use all vector lanes initially
__m512i key, val, loc;  // state (keys, payloads, and previous hash bucket locations)
size_t i = 0, j = 16, o = 0;
do {  // process 16 tuples at a time in vectorized code
  key = _mm512_mask_loadunpack_epi32(key, k, &outer_keys[i]);  // load new keys reusing vector lanes
  val = _mm512_mask_loadunpack_epi32(val, k, &outer_vals[i]);  // load new payloads
  i += j;  // increment input using the number of reused lanes
  __m512i fac = _mm512_mask_blend_epi32(k, mask_factor_2, mask_factor_1);  // pick hash factor
  __m512i buc = _mm512_mask_blend_epi32(k, mask_buckets_minus_1, mask_buckets);  // pick buckets
  __m512i loc_0 = _mm512_add_epi32(loc, mask_1);  // increment old hashes
  loc = _mm512_mulhi_epu32(_mm512_mullo_epi32(key, fac), buc);  // compute hashes
  loc = _mm512_mask_add_epi32(loc, _mm512_knot(k), loc, loc_0);  // combine hashes
  k = _mm512_cmpge_epi32_mask(loc, mask_buckets);  // find overflows
  loc = _mm512_mask_sub_epi32(loc, k, loc, mask_buckets);  // fix overflows
  __m512i key_H = _mm512_i32gather_epi32(loc, &hash_table[0].key, 8);  // gather hash bucket keys
  __m512i val_H = _mm512_i32gather_epi32(loc, &hash_table[0].val, 8);  // gather payloads
  k = _mm512_cmpeq_epi32_mask(key, key_H);  // find matching lanes
  _mm512_mask_packstore_epi32(&join_keys[j], k, key);  // (selective) store join keys to output
  _mm512_mask_packstore_epi32(&join_outer_vals[j], k, val);  // (selective) store outer payloads
  _mm512_mask_packstore_epi32(&join_inner_vals[j], k, val_H);  // (selective) store inner payloads
  o += _mm_countbits_64(_mm512_mask2int(k));  // update the output index
  k = _mm512_cmpeq_epi32_mask(key_H, mask_empty);  // find which lanes reached an empty bucket
  j = _mm_countbits_64(_mm512_mask2int(k));  // count the reusable lanes
} while (i + j <= tuples);  // check if at least 16 tuples remain to process in SIMD code
[...]  // process the final (15 or less) tuples in scalar code
```

### 5.4.3 Cuckoo Hashing

Cuckoo hashing [Pagh and Rodler, 2004] is another hashing scheme that uses multiple hash functions. and is the only hash table scheme that has been vectorized in previous work [Ross, 2007], as a means to allow multiple keys per bucket (horizontal vectorization). Here, we study cuckoo hashing to compare our (vertical vectorization) approach against previous work [Ross, 2007; Zukowski *et al.*, 2006]. We also show that complicated control flow logic, such as cuckoo table building, can be transformed to data flow vector logic.

Algorithm 26 shows the scalar algorithm for probing cuckoo hashing tables. For each input key, we probe up to two hash buckets in order to find a match. Since we search up to two buckets, we only need a branch to handle collisions, instead of an inner loop.

When probing cuckoo tables in scalar code, we can avoid the branching by always accessing both buckets and combining their values using bitwise logic. This approach eliminates branching at the expense of always accessing both buckets, but [Zukowski *et al.*, 2006] showed it to be faster than other implementations on mainstream CPUs at the time.

**Algorithm 26:** Cuckoo Hashing - Probe (Scalar)

1 $i, o \leftarrow 0$
2 **do**
3 $\quad k \leftarrow S_k[i]$ // load outer key
4 $\quad h \leftarrow (k \cdot f_1) \times\uparrow |H|$ // $1^{\text{st}}$ hash function
5 $\quad$ **if** $k \neq H_k[h]$ **and** $H_k[h] \neq k_E$ **then**
6 $\quad\quad h \leftarrow (k \cdot f_2) \times\uparrow |H|$ // $2^{\text{st}}$ hash function
7 $\quad$ **if** $k = H_k[h]$ **then**
8 $\quad\quad T_k[o] \leftarrow k$ // store (joined) key
9 $\quad\quad T_{Sv}[o] \leftarrow S_v[i]$ // store outer payload
10 $\quad\quad T_{Rv}[o] \leftarrow H_v[h]$ // store inner payload
11 $\quad\quad o \leftarrow o + 1$
12 $\quad i \leftarrow i + 1$
13 **while** $i \neq |S|$

Vectorized cuckoo table probing is shown in Algorithm 27. We do not reuse a subset of vector lanes per iteration, we always process $\mathcal{W}$ new keys, accessing the input "in-order". As a result, vectorized cuckoo table probing is *stable* to the order of probing relation. To probe the hash table, we gather the bucket based on the first hash function for all keys. For the lanes where the keys do not match and the buckets are empty, we gather based on the second function. Eagerly accessing both buckets is wasteful since selective gathers are already branchless. Note that we skip the second bucket if the first bucket is empty.

**Algorithm 27:** Cuckoo Hashing - Probe (Vector)

1 $i, o \leftarrow 0$
2 **do**
3 $\quad \vec{k} \leftarrow S_{keys}[i]$ // load outer keys
4 $\quad \vec{v} \leftarrow S_{vals}[i]$ // load outer payloads
5 $\quad i \leftarrow i + \mathcal{W}$
6 $\quad \vec{h}_1 \leftarrow (\vec{k} \cdot f_1) \times\uparrow |H|$ // $1^{\text{st}}$ hash function
7 $\quad \vec{h}_2 \leftarrow (\vec{k} \cdot f_2) \times\uparrow |H|$ // $2^{\text{nd}}$ hash function
8 $\quad \vec{k}_H \leftarrow H_k[\vec{h}_1]$ // gather keys from $1^{\text{st}}$ hash bucket
9 $\quad \vec{k}_H \leftarrow H_k[\vec{h}_2]$ // gather payloads from $1^{\text{st}}$ hash bucket
10 $\quad m \leftarrow (\vec{k} \neq \vec{k}_H)$ & $(\vec{k}_H \neq k_E)$ // find lanes to access the $2^{\text{nd}}$ bucket
11 $\quad \vec{k}_H \leftarrow_m H_k[\vec{h}_2]$ // gather keys from $2^{\text{nd}}$ hash bucket if $1^{\text{st}}$ not matching
12 $\quad \vec{v}_H \leftarrow_m H_v[\vec{h}_2]$ // gather payloads from $2^{\text{nd}}$ hash bucket if $1^{\text{st}}$ not matching
13 $\quad m \leftarrow \vec{k} = \vec{k}_H$ // find matching keys
14 $\quad T_k[o] \leftarrow_m \vec{k}$ // (selective) store (joined) keys
15 $\quad T_{Sv}[o] \leftarrow_m \vec{v}$ // (selective) store outer payloads
16 $\quad T_{Rv}[o] \leftarrow_m \vec{v}_H$ // (selective) store inner payloads
17 $\quad o \leftarrow o + |m|$
18 **while** $i \neq |S|$

```
size_t i = 0, o = 0;
do {  // process 16 (new) probing keys per iteration
  __m512i key = _mm512_load_epi32(&outer_keys[i]);  // load input keys
  __m512i val = _mm512_load_epi32(&outer_vals[i]);  // load input payloads
  // compute both hash functions using different multiplicative hash factors
  __m512i loc_1 = _mm512_mulhi_epu32(_mm512_mullo_epi32(key, mask_factor_1), mask_buckets);
  __m512i loc_2 = _mm512_mulhi_epu32(_mm512_mullo_epi32(key, mask_factor_2), mask_buckets);
  __m512i key_H = _mm512_i32gather_epi32(loc_1, &hash_table[0].key, 8);  // gather 1st bucket keys
  __m512i val_H = _mm512_i32gather_epi32(loc_1, &hash_table[0].val, 8);  // gather payloads
  __mmask16 k = _mm512_cmpneq_epi32_mask(key, key_H);  // find lanes where keys are not matching ...
  k = _mm512_mask_cmpneq_epi32_mask(k, key, mask_empty);  // and also do not point to empty buckets
  // (selective) gather for keys and payloads from bucket using 2nd hash function
  key_H = _mm512_mask_i32gather_epi32(key_H, k, loc_2, &hash_table[0].key, 8);
  val_H = _mm512_mask_i32gather_epi32(val_h, k, loc_2, &hash_table[0].val, 8);
  k = _mm512_cmpeq_epi32_mask(key, key_H);  // find lanes with matching keys
  _mm512_mask_packstore_epi32(&join_keys[j], k, key);  // (selective) store join keys to output
  _mm512_mask_packstore_epi32(&join_outer_vals[j], k, val);  // (selective) store outer payloads
  _mm512_mask_packstore_epi32(&join_inner_vals[j], k, val_H);  // (selective) store inner payloads
  o += _mm_countbits_64(_mm512_mask2int(k));  // update the output index
} while ((i = i + 16) != tuples);
```

The KNC implementation of cuckoo table probing with 32-bit keys and 32-bit payloads is shown above. In contrast to linear probing and double hashing, we do not use selective loads as the input is read "in order". We simplify the code using separate 32-bit gathers for keys and payloads. In practice, we access key–payload pairs using 64-bit gathers.

Building a cuckoo hashing table, shown in Algorithm 28, is more involved than probing.

**Algorithm 28:** Cuckoo Hashing - Build (Scalar)

```
1  i ← 0
2  do
3      k ← R_k[i]                                   // load inner key
4      v ← R_v[i]                                   // load inner payload
5      i ← i + 1
6      h ← (k · f_1) ×↑ |H|                          // 1st hash function
7      if H_k[h] ≠ k_E then
8          h ← (k · f_2) ×↑ |H|                      // 2nd hash function
9          while H_k[h] ≠ k_E do
10             H_k[h] ⟷ k            // swap current key with key in hash bucket
11             H_v[h] ⟷ v    // swap current payload with payload in hash bucket
12             h_1 ← (k · f_1) ×↑ |H|                // 1st hash function
13             h_2 ← (k · f_1) ×↑ |H|                // 2nd hash function
14             h ← h_1 + h_2 − h           // find alternative hash bucket
15     H_k[h] ← k                           // store key in hash bucket
16     H_v[h] ← v                       // store payload in hash bucket
17 while i ≠ |S|
```

We always place the tuple in the first hash bucket if empty. Otherwise, we try the 2nd bucket. If both buckets are full, we have to create space for the new tuple. We choose one of the two buckets that the new tuple can be placed (here the 2nd) and swap it with the tuple there. Now we have to insert the new tuple that we swapped out by searching the alternative bucket. We may swap many tuples before we reach an empty bucket. To avoid infinite loops, we set a bound on the total iterations and, if reached, we rebuild the table.

Vectorized cuckoo table building is described in Algorithm 29. In each iteration, we reuse vector lanes to load new tuples from the input. The lanes we cannot reuse hold tuples that were either displaced from the hash table or were conflicting. In each iteration, the new tuples look for an empty bucket using the first and then the second hash function. The tuples that were carried from the previous iteration use the alternative hash function. After we scatter the tuples to the buckets, we gather back the keys to detect conflicts, taking advantage of the fact that keys are unique. The conflicting lanes and the lanes with newly displaced tuples are passed through to the next iteration. The remaining lanes are reused.

**Algorithm 29:** Cuckoo Hashing - Build (Vector)

```
1  i ← 0
2  m ← true                                    // initially reuse all lanes
3  do
4  |  k⃗ ←_m R_k[i]                            // (selective) load new inner keys reusing lanes
5  |  v⃗ ←_m R_v[i]                            // (selective) load new inner payloads
6  |  i ← i + |m|                              // increment input index by number of reused lanes
7  |  h⃗_1 ← (k⃗ · f_1) ×↑ |H|                 // compute 1st hash function
8  |  h⃗_2 ← (k⃗ · f_2) ×↑ |H|                 // compute 2nd hash function
9  |  h⃗ ← h⃗_1 + h⃗_2 − h⃗                    // compute alternative hash function (for old tuples)
10 |  h⃗ ← m ? h⃗_1 : h                         // use 1st hash function for new tuples
11 |  k⃗_H ← H_k[h⃗]                           // gather keys from hash buckets for new and old tuples
12 |  v⃗_H ← H_v[h⃗]                           // gather payloads from hash buckets for new and old tuples
13 |  m ← m & (k⃗_H ≠ k_E)                      // use 2nd function if new tuple and 1st bucket not empty
14 |  k⃗_H ←_m H_k[h⃗_2]                       // (selective) gather keys from 2nd hash bucket
15 |  v⃗_H ←_m H_v[h⃗_2]                       // (selective) gather payloads from 2nd choice hash bucket
16 |  h⃗ ← m ? h⃗_2 : h⃗                        // merge locations
17 |  H_k[h⃗] ← k⃗                              // scatter keys to hash buckets
18 |  H_v[h⃗] ← v⃗                              // scatter payloads to hash buckets
19 |  m ← k⃗ ≠ H_k[h⃗]                          // gather (unique) keys back to detect conflicts
20 |  k⃗ ← m ? k⃗_H : k⃗                        // retain keys of conflicting lanes
21 |  v⃗ ← m ? v⃗_H : v⃗                        // retain payloads of conflicting lanes
22 |  m ← k⃗ = k_E                              // reuse lanes where swapped buckets were empty
23 while i + |m| ≤ |R|
```

```
__mmask16 k = _mm512_int2mask(0xFFFF);  // set to use all vector lanes initially
__m512i key, val, loc;  // state (keys, payloads, and last hash locations)
size_t i = 0, j = 16, l = [...];  // upper bound in iterations to avoid infinite loops
do {  // process 16 tuples at a time
  key = _mm512_mask_loadunpack_epi32(key, k, &inner_keys[i]);  // load new keys reusing vector lanes
  val = _mm512_mask_loadunpack_epi32(val, k, &inner_vals[i]);  // load new payloads
  i += j;  // update input index and compute both hash functions
  __m512i loc_1 = _mm512_mulhi_epu32(_mm512_mullo_epi32(key, mask_factor_1), mask_buckets);
  __m512i loc_2 = _mm512_mulhi_epu32(_mm512_mullo_epi32(key, mask_factor_2), mask_buckets);
  loc = _mm512_sub_epi32(_mm512_add_epi32(loc_1, loc_2), loc);  // alternative bucket for old tuples
  loc = _mm512_mask_blend_epi32(k, loc, loc_1);  // search 1st hash bucket for new tuples
  __m512i key_H = _mm512_i32gather_epi32(loc, &hash_table[0].key, 8);  // gather keys
  __m512i val_H = _mm512_i32gather_epi32(loc, &hash_table[0].val, 8);  // gather payloads
  k = _mm512_mask_cmpneq_epi32_mask(k, key_H, mask_empty);  // search 2nd if new and 1st was full
  // (selective) gather of keys and payloads from the 2nd hash bucket
  key_H = _mm512_mask_i32gather_epi32(key_H, k, loc_2, &hash_table[0].key, 8);
  val_H = _mm512_mask_i32gather_epi32(val_H, k, loc_2, &hash_table[0].val, 8);
  loc = _mm512_mask_blend_epi32(k, loc, loc_2);  // merge 2nd hash function
  _mm512_i32scatter_epi32(&hash_table[0].key, loc, key_H, 8);  // scatter keys to table
  _mm512_i32scatter_epi32(&hash_table[0].val, loc, val_H, 8);  // scatter payloads to table
  __m512i key_C = _mm512_i32gather_epi32(loc, &hash_table[0].key, 8);  // gather back (unique) keys
  k = _mm512_cmpneq_epi32_mask(key, key_C);  // detect conflicting lanes
  key = _mm512_mask_blend_epi32(k, key, key_H);  // retain keys of conflicting lanes
  val = _mm512_mask_blend_epi32(k, val, val_H);  // retain payloads of conflicting lanes
  k = _mm512_cmpeq_epi32_mask(key, mask_empty);  // reuse lanes where buckets were empty
  j = _mm_countbits_64(_mm512_mask2int(k));  // count number of reused lanes
} while (i + j <= tuples && --l);  // check if at least 16 tuples remain to process in SIMD code
[...]  // process last tuples in scalar code if upper bound of iterations was not reached
```

The KNC implementation for building a cuckoo table with 32-bit keys and 32-bit payloads is shown above. Build failures a cuckoo table can occur even for low load factors. If we know the number of inner tuples in advance, we can set a suitable load factor. If we still fail, we modify the hash function and retry without resizing. Note that cuckoo hashing does not require the hash functions to be homomorphic as we always compute both.

## 5.5 Bloom Filters

Bloom filters [Bloom, 1970] are an essential data structure for *semi-joins*, i.e. applying selective conditions across tables before the join. A tuple qualifies from the Bloom filter, if $k$ specific bits are set in the filter, based on $k$ hash functions. In the scalar implementation, we process one input key at a time $k$ bits by computing hash functions iteratively. If a bit is not set, we proceed directly to the next tuple. This *short-circuit* logic is essential for high performance, as most tuples will fail in the first or second bit test, assuming high selectivity.

```
size_t i = 0, o = 0;
do {
  int32_t key = input_keys[i];  // load input key
  size_t f = 0;  // check one hash function at a time
  do {
    size_t h = (key * hash_factor[f]) >> hash_shift;  // multiplicative hashing
    asm goto(  // inline assembly in C code
    "btl %1,(%0)\n\t"  // load bit from memory and test if set
    "jnc %l[bit_not_set]"  // jump to label if bit is not set
    :: "r"(bloom_filter), "r"(h) : "cc" : bit_not_set);  // compiler jargon for inline assembly
  } while (++f != hash_functions);  // tuple qualifies since all bits are set
  output_vals[o] = input_vals[i];  // copy payload to output
  output_keys[o++] = key;  // store key to output
bit_not_set:;  // label to jump to if any key fails
} while (++i != tuples);
```

We show the scalar implementation of probing a Bloom filters in the (scalar) assembly of mainstream CPUs (`x86`). The input has 32-bit keys and 32-bit payloads. We use multiplicative hashing and assume the Bloom filter has $2^n$ bits with $n \leq 32$. To test a specific bit in the bitmap, we use the `x86` instruction `bt` and avoid doing shift and bitwise-and to separate the word index and the bit offset. In contrast to hash tables, building a Bloom filter is uninteresting and should take negligible time compared to probing.

To access a bitmap in SIMD, we use the smallest possible SIMD gather lane supported by the hardware. We test the lane against a bitmask with the single target bit set. For instance, if the smallest gather lane is 32-bit integers, we gather the 32-bit word containing the target bit using the bit offset divided by 32. We also create a bitmask using the bit offset modulo 32. Checking the target bit in the gathered word is trivial.

Eagerly testing all $k$ bits per input tuple using all $k$ hash functions to determine whether the tuple qualifies the Bloom filter or not, is a very wasteful approach. Early aborting of tuples that failed a bit test are essential to performance for cases with high selectivity. To maintain this property, we must not vectorize accesses for the same key. To vectorize accesses across keys, we use the same approach of *vertical vectorization* that we used for hash tables, where we process different input keys per SIMD vector lane. Also, different vector lanes are not necessarily tied to the same hash function. In each iteration, each key may be checking a different hash function, thus we keep the hash function index as part of the state. Generally, we cannot assume a specific pattern of failures, as the flow is dependent on both the input data and the configuration.

```
__mmask16 k = _mm512_int2mask(0xFFFF);  // set to use all lanes initially
__m512i key, val, fun;  // state (keys, payloads, and hash function index)
size_t i = 0, j = 16, o = 0;
do {  process 16 tuples at a time
  key = _mm512_mask_loadunpack_epi32(key, k, &input_keys[i]);  // load new keys reusing vector lanes
  val = _mm512_mask_loadunpack_epi32(val, k, &input_vals[i]);  // load new payloads
  i += j;  // increment input using the number of reused vector lanes
  fun = _mm512_mask_xor_epi32(fun, k, fun, fun);  // reset hash function index for reused lanes
  __m512i fac = _mm512_permutevar_epi32(fun, mask_factors);  // pick multiplicative hash factor
  __m512i bit = _mm512_srl_epi32(_mm512_mullo_epi32(key, fac), mask_shift);  // hash function
  // get 32-bit word containing target bit from Bloom filter by dividing bit offset by 32
  __m512i bit_div = _mm512_i32gather_epi32(_mm512_srli_epi32(bit, 5), bloom_filter, 4);
  // create 32-bit bitmask with only target bit set using the bit offset modulo 32
  __m512i bit_mod = _mm512_sllv_epi32(mask_1, _mm512_and_epi32(bit, mask_31));
  k = _mm512_test_epi32_mask(bit_div, bit_mod);  // test target bit in Bloom filter
  fun = _mm512_add_epi32(fun, mask_1);  // increment hash function index
  // find qualifying lanes that checked all bits by using all hash functions
  __mmask16 kk = _mm512_mask_cmpeq_epi32_mask(k, fun, mask_hash_functions);
  _mm512_mask_packstore_epi32(&output_keys[j], kk, key);  // (selective) store qualifying keys
  _mm512_mask_packstore_epi32(&output_vals[j], kk, val);  // (selective) store qualifying payloads
  k = _mm512_kor(_mm512_knot(k), kk);  // reuse lane if bit was not set or tuple qualified
  j = _mm_countbits_64(_mm512_mask2int(k));  // count number of reused lanes
} while (i + j <= tuples);  // check if at least 16 tuples remain to process in SIMD code
[...]  // process (15 or less) tuples remaining in scalar code
```

The KNC implementation for probing Bloom filters is shown above. As in the scalar code, we assume 32-bit keys, 32-bit payloads, and $2^n$ bits in the Bloom filter. Using stronger hash functions than multiplicative hashing is also possible, but, similarly to double hashing, they need to have the same code. We can vary the hash function seed or some function constants, provided they guarantee that the instances are pairwise independent.

## 5.6 Partitioning

Partitioning is essential for query execution in modern hardware to execute in-memory database operators in a cache-conscious way. Both hash join and group-by aggregation can use hash partitioning to split the input into non-overlapping parts that can be distributed among threads and fit in the small but fast caches [**?**; Balkesen *et al.*, 2013a; Blanas *et al.*, 2011; Kim *et al.*, 2009; Manegold *et al.*, 2002]. In Chapter 3, we studied various types of partitioning depending on the partition function, the layout of the output, the use of auxiliary space, and the memory layer that the partitioning variant is designed to run on. We also showed that partitioning is synonymous to efficient sorting.

### 5.6.1 Radix / Hash Histogram

Prior to moving any data, in order to partition into contiguous segments, we use a histogram to set the boundaries for each partitioning. To compute the histogram, we increment a count based on the partition function of each key. Provided we use a hash function such as multiplicative hashing, hash partitioning becomes equally fast to radix partitioning.

**Algorithm 30:** Hash Partitioning - Histogram

1 $\vec{o} \leftarrow \{0, 1, 2, 3, ..., \mathcal{W} - 1\}$
2 $i, p \leftarrow 0$
3 **do**
4     $\vec{k} \leftarrow T_k[i]$ // load keys
5     $i \leftarrow i + \mathcal{W}$
6     $\vec{h} \leftarrow (\vec{k} \cdot f) \times \uparrow \mathcal{P}$ // hash partitioning function
7     $\vec{h} \leftarrow (\vec{h} \cdot \mathcal{W}) + \vec{o}$ // split index to point to replicated histograms
8     $\vec{c} \leftarrow H'[\vec{h}]$ // gather the histogram counts
9     $H'[\vec{h}] \leftarrow \vec{c} + 1$ // increment and scatter the histogram counts
10 **while** $i \neq |T|$
11 **do**
12     $\vec{c} \leftarrow H'[p \cdot \mathcal{W}]$ // load W counts of partition
13     $H[p] \leftarrow \text{sum_across}(\vec{c})$ // reduce into single result
14     $p \leftarrow p + 1$
15 **while** $p \neq \mathcal{P}$

Vectorized histogram generation, shown in Algorithm 30, uses gathers and scatters to increment counts. However, if $k > 1$ lanes scatter to the same location, the count will be incremented by 1 instead of $k$. To avoid conflicts, we replicate the histogram and isolate each lane. Lane $j$ increments $H'[i \cdot \mathcal{W} + j]$ instead of $H[i]$. In the end, the $\mathcal{W}$ histograms are merged. If the histograms do not fit in the L1, we use 1-byte counts and flush to wider counts when they overflow. The KNC implementation assuming 32-bit keys is shown below.

```
const __m512i mask_lane_offset = _mm512_set_epi32(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0);
size_t i = 0, p = 0;
do {  // process 16 tuples at a time
  __m512i key = _mm512_load_epi32(&keys[i]);  // load keys
  __m512i pid = _mm512_mulhi_epu32(_mm512_mullo_epi32(key, mask_factor), mask_partitions);
  // compute locations in replicated histograms by multiplying by 16 and adding the lane offset
  __m512i loc = _mm512_or_epi32(_mm512_slli_epi32(pid, 4), mask_lane_offset);
  __m512i cnt = _mm512_i32gather_epi32(loc, histogram_x16, 4);  // update the replicated histograms
  _mm512_i32scatter_epi32(histogram_x16, loc, _mm512_add_epi32(cnt, mask_1), 4);
} while ((i = i + 16) != tuples);
do {  // merge partial histograms
  histogram[p] = _mm512_reduce_add_epi32(_mm512_load_epi32(&histogram_x16[p << 4]));
} while (++p != partitions);
```

### 5.6.2 Range Histogram

Radix and hash partitioning functions are significantly faster than range partitioning functions. In range partitioning, we use binary search over a sorted array of delimiters than divide the ranges. Although the array is cache-resident, binary search is $O(\log \mathcal{P})$ and the accesses are dependent on each other exposing the cache hit latency in the critical path. Branch elimination in scalar code only marginally improves performance.

In Section 3.2.5.2, we proposed a range index where each node has multiple splitters that are compared against one input key using SIMD comparisons. Each node is at least as wide as a vector and scalar code is used for index arithmetic and to access the nodes. The SIMD index can be viewed as horizontal vectorization of binary search.

The vertical vectorization of binary search processes $\mathcal{W}$ input keys in parallel and uses gather instructions to load the splitters from the sorted array, as shown in Algorithm 31. The search path is computed by blending vectors of "low" and "high" indexes.

**Algorithm 31:** Range Partitioning - Histogram

```
1  o⃗ ← {0, 1, 2, 3, ..., W − 1}
2  i, p ← 0
3  do
4  |   k⃗ ← T_k[i]                                  // load the next keys
5  |   i ← i + W
6  |   r⃗_L ← 0                                     // "low" index of binary search
7  |   r⃗_H ← P                                     // "high" index of binary search
8  |   j ← ⌈log P⌉                                 // binary search across P range partitions
9  |   do
10 |   |   r⃗ ← (r⃗_L + r⃗_H) >> 1                    // compute "mid" index
11 |   |   d⃗ ← D[r⃗ − 1]                            // gather delimiters from sorted array D
12 |   |   m ← k⃗ > d⃗                               // compare input keys with delimiters
13 |   |   r⃗_L ← m ? r⃗ : r⃗_L                       // merge "low" index with "mid" index
14 |   |   r⃗_H ← m ? r⃗_H : r⃗                       // merge "high" index with "mid" index
15 |   |   j ← j − 1
16 |   while j ≠ 0
17 |   r⃗ ← o⃗ + (r⃗_L · W)                          // location in replicated histograms
18 |   c⃗ ← H′[r⃗]                                  // gather the histogram counts
19 |   H′[r⃗] ← c⃗ + 1                              // increment and scatter the histogram counts
20 while i ≠ |T|
21 do
22 |   c⃗ ← H′[p · W]                              // load W counts of partition
23 |   H[p] ← sum_across(c⃗)                       // reduce into single result
24 |   p ← p + 1
25 while p ≠ P
```

```
const __m512i mask_lane_offset = _mm512_set_epi32(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0);
size_t i = 0, p = 0;
do {  // process 16 tuples at a time
  __m512i key = _mm512_load_epi32(&keys[i]);  // load keys
  __m512i pid_L = _mm512_setzero_epi32();  // low index for binary search
  __m512i pid_H = mask_partitions;  // high index for binary search
  size_t j = ceil_log_partitions;
  do {  // inner loop for binary search
    __m512i pid = _mm512_srli_epi32(_mm512_add_epi32(pid_L, pid_H), 1);  // mid index
    __m512i del = _mm512_i32gather_epi32(loc, &delimiters[-1], 4);  // gather delimiters
    __mmask16 k = _mm512_cmpgt_epu32(key, del);  // compare against delimiters
    loc_L = _mm512_mask_blend_epi32(k, pid_L, pid);  // update low index
    loc_H = _mm512_mask_blend_epi32(k, pid, pid_H);  // update high index
  } while (--j);
  // compute locations in replicated histograms by multiplying by 16 and adding the lane offset
  __m512i pid = _mm512_or_epi32(_mm512_slli_epi32(pid_L, 4), mask_lane_offset);
  __m512i cnt = _mm512_i32gather_epi32(loc, histogram_x16, 4);  // update the replicated histograms
  _mm512_i32scatter_epi32(histogram_x16, loc, _mm512_add_epi32(cnt, mask_1), 4);
} while ((i = i + 16) != tuples);
do {  // merge partial histograms
  histogram[p] = _mm512_reduce_add_epi32(_mm512_load_epi32(&histogram_x16[p << 4]));
} while (++p != partitions);
```

The KNC implementation of range partitioning assuming 32-bit keys is shown above. Note that these implementations are identical using AVX-512, the latest SIMD instruction set supported by the latest mainstream CPUs and Intel Xeon Phi platforms.

### 5.6.3 Data Shuffling

The data shuffling phase of partitioning involves the actual movement of tuples. To generate the output partitions in contiguous space, we maintain an array of partition offsets, initialized by the prefix sum of the histogram. The offset array is updated for every tuple transferred to the output. We described many variants of shuffling in Section 3.2.

Vectorized data shuffling uses gathers and scatters to increment the offset array and scatters the tuples to the output. However, if $k > 1$ vector lanes have tuples that belong to the same partition, the offset would be incremented by 1 instead of $k$. Also, the $k$ input tuples would be (over)written to the same output location.

To resolve the conflicts, we compute a vector of offsets that serialize the tuples in the same input vector that belong to the same output partition. The process is described in Algorithm 32 and uses multiple rounds of gathers and scatters, assuming that the hardware does not provide specialized vector instructions for detecting conflicts.

**Algorithm 32:** Conflict Serialization w/o Hardware Support

1 $\vec{v} \leftarrow \{\mathcal{W}-1, \mathcal{W}-2, \mathcal{W}-3, \ldots, 0\}$ // mask with unique values also used to reverse vector lanes
2 $\vec{h}' \leftarrow$ permute$(\vec{h}, \vec{m})$ // reverse lanes of hashes
3 $\vec{c} \leftarrow 0$ // serialization offsets (output)
4 $m \leftarrow$ true
5 **do**
6 $\quad T[\vec{h}'] \leftarrow_m \vec{v}$ // detect conflicts using array $T$
7 $\quad \vec{v}' \leftarrow_m T[\vec{h}']\ m \leftarrow m\ \&\ (\vec{v} \neq \vec{v}')$ // update conflicting lanes
8 $\quad \vec{c} \leftarrow m\ ?\ (\vec{c}+1) : \vec{c}$ // increment offsets for conflicting lanes
9 **while** $m \neq$ false
10 **return** permute$(\vec{c}, \vec{v})$ // reverse counts to original order

We scatter unique values per lane to an array with $\mathcal{P}$ entries for $\mathcal{P}$ partitions. Then, we gather using the same indexes and compare against the scattered vector to find conflicts. We increment the conflicting lanes and repeat the process for these lanes only until no lanes conflict. Note that even if $\mathcal{W}$ iterations are executed, the total number of accesses to distinct memory locations is always $\mathcal{W}$, i.e., if $a_i$ is the number of accesses to distinct memory locations in iteration $i$, then $\sum a_i = \mathcal{W}$. Thus, conflict detection runs in up to $O(\mathcal{W})$ CPU cycles, same as the worst case for any individual $\mathcal{W}$-way gather or scatter.

```
__m512i serialize_offsets(__m512i pid, int32_t* array, [...]) {  // inlined with masks as arguments
  __mmask16 k = _mm512_int2mask(0xFFFF);  // use all lanes initially
  __m512i val, off = _mm512_setzero_epi32();  // reset offsets
  __m512i loc = _mm512_permutevar_epi32(pid, mask_reverse);  // reverse lanes with partition ids
  do {
    _mm512_mask_i32scatter_epi32(array, k, loc, mask_reverse, 4);  // scatter unique values per lane
    val = _mm512_mask_i32gather_epi32(val, k, loc, array, 4);  // gather back scattered values
    k = _mm512_cmpneq_epi32_mask(val, mask_reverse);
    off = _mm512_mask_add_epi32(off, k, off, mask_1);  // increment offsets
  } while (!_mm512_kortestz(k, k));  // until no conflicts remain
  return _mm512_permutevar_epi32(off, mask_reverse);  // reverse offsets back to original order
}  // code for KNC (Intel Xeon Phi 1st generation co-processors)
```

The KNC implementation of the above loop is shown above. In AVX-512, we can serialize the conflicts via conflict detection instructions without gathers and scatters. The `vpconflictd` instruction (with intrinsic `_mm512_conflict_epi32`) compares all 32-bit lanes to all previous 32-bit lanes in the 512-bit SIMD register. The output register has a 16-bit bitmask per 32-bit lane denoting the conflicts. Lane $i$ conflicts with lane $j$ if $i > j$ and their values match. Then the bitmask in lane $i$ will have the $j^{\text{th}}$ bit set. We need 8 AVX-512 instructions to serialize conflicts; 7/8 perform the bit population count, if not directly supported. AVX defines `_mm512_popcnt_epi32` but it is not supported in KNL processors.

```
__m512i serialize_conflicts(__m512i pid, [...]) {  // inlined with masks as arguments
  __m512i x = _mm512_conflict_epi32(pid);  // create 15-bit bitmasks of conflicts per 32-bit lane
  __m512i y = _mm512_srli_epi32(x, 5);  // shift bits 6 to 10 to the low-order bits
  __m512i z = _mm512_srli_epi32(x, 10);  // shift bits 11 to 15 to the low-order bits
  // count set bits in ranges using 32-way permutation using 5 low-order bits of index register
  x = _mm512_permutex2var_epi32(mask_bit_pop_count_L, x, mask_bit_pop_count_H);  // bit range [1,5]
  y = _mm512_permutex2var_epi32(mask_bit_pop_count_L, y, mask_bit_pop_count_H);  // bit range [6,10]
  z = _mm512_permutex2var_epi32(mask_bit_pop_count_L, z, mask_bit_pop_count_H);  // bit range [11,15]
  return _mm512_add_epi32(_mm512_add_epi32(x, y), z);  // add the counted set bits
}  // code in AVX-512 SIMD (latest mainstream CPUs and latest Intel Xeon Phi platforms)
```

In the KNC implementation for conflict serialization, we reverse the lanes before the rounds of scatters and gathers. Since the value of the rightmost lane per distinct scatter location persists, but we want the value of the leftmost lanes (in the original order) to be written. For $k$ lanes pointing to the same partition, the rightmost lane will have a serialization offset of $k-1$. Thus, by writing the serialization offset plus 1, we can also update histograms correctly without replicating the counts. In the case of data shuffling, we update the prefix sum of histograms. Furthermore, because the conflicts are written in leftmost to rightmost lane order, the shuffling operation is *stable*. Stable partitioning is essential when we execute multiple partitioning passes iteratively and use the same partition function from low-order to high-order bits. The most prominent example is LSB radixsort. Vectorized data shuffling is shown in Algorithm 33, extending the scalar version in Algorithm 12. Note that we transfer tuples from the input to the output table directly without buffering, thus, this algorithm works well only when both the input and the output are cache-resident.

**Algorithm 33:** Radix Partitioning - Shuffling

1 $O \leftarrow \text{prefix_sum}(H)$ // partition offsets from histogram
2 $i \leftarrow 0$
3 **do**
4 $\quad \vec{k} \leftarrow T_k[i]$ // load input keys
5 $\quad \vec{v} \leftarrow T_v[i]$ // load input payloads
6 $\quad i \leftarrow i + \mathcal{W}$
7 $\quad \vec{h} \leftarrow (\vec{k} << s_1) >> s_2$ // compute partition function
8 $\quad \vec{o} \leftarrow O[\vec{h}]$ // gather output offsets
9 $\quad \vec{c} \leftarrow \text{serialize_conflicts}(\vec{h}, O)$
10 $\quad \vec{o} \leftarrow \vec{o} + \vec{c}$ // add serialization offsets
11 $\quad O[\vec{h}] \leftarrow \vec{o} + 1$ // scatter incremented offsets
12 $\quad T'_k[\vec{o}] \leftarrow \vec{k}$ // scatter keys to output
13 $\quad T'_v[\vec{o}] \leftarrow \vec{v}$ // scatter payloads to output
14 **while** $i \neq |T|$

### 5.6.4 Buffered Data Shuffling

Data shuffling, as described so far, is fast if the input and the output is cache resident, but falls into performance pitfalls if out of the cache. As we explained in Section 3.1, (i) we suffer from TLB thrashing when the partitioning fanout exceeds the TLB capacity [Manegold *et al.*, 2000b], (ii) we trigger many cache conflicts [Satish *et al.*, 2010] and in the worst case, may be bound by the size of the cache associativity set, and (iii) we pollute the cache with output data we will not reuse, reducing the effective memory bandwidth [Wassenberg and Sanders, 2011]. The vectorized implementation of simple non-buffered shuffling improves performance, but suffers from the same performance pitfalls as the scalar version. Here, vectorization improves performance compared to its scalar counterpart, but does not overcome cache inefficiencies. In short, vectorization is orthogonal to other optimizations.

To address these inefficiencies when shuffling data larger than the cache, we apply the *out-of-cache* partitioning techniques described in Section 3.2. We use cache-resident buffers with a footprint small enough to remain cache-resident. If the buffers can hold $\mathcal{B}$ tuples per partition, we reduce the cache and TLB misses to $1/\mathcal{B}$. If the buffers are flushed with non-temporal stores, we facilitate hardware write-combining and avoid polluting the cache with output data. The partitioning fanout is bounded by the cache capacity in order to ensure the buffer remains cache-resident. In vectorized code, we set $\mathcal{B} = \mathcal{W}$ to ensure we move data to memory in cache line units while maximizing the partitioning fanout.

The improvement of vectorized buffered shuffling shown in Algorithm 34 over vectorized unbuffered shuffling shown in Algorithm 33, is scattering the tuples to the cache resident buffer rather than directly to the output. For each vector of tuples, once the tuples are scattered, we iterate over the partitions that the current $\mathcal{W}$ input tuples belong to, and flush to the output when all available buffer slots are filled. The scalar version is Algorithm 14.

Since multiple tuples can be written to the buffer for the same partition on each iteration, writing up to $\mathcal{W}$ tuples to buffers with $\mathcal{W}$ slots can cause overflows. We identify which vector lanes will not cause overflow and store tuples there using selective scatters. After flushing buffers that are full, we scatter the remaining tuples. We identify which vector lanes point to partitions with full buffers using the output index by inferring which lanes scattered to the last slots of buffers per partition. Thus, we also ensure that we will not flush the same buffer

twice. Flushing occurs "horizontally" for one partition at a time, after selectively storing the partitions in the stack, and is done by streaming stores to avoid polluting the cache with output data [Wassenberg and Sanders, 2011]. Note that we are storing to multiple outputs, thus, output buffering (see Section 5.3) is already part of the algorithm. If partitioning does not need to be stable, instead of conflict serialization, we detect and process conflicting lanes during the next loop. Performance is slightly increased since few conflicts normally occur per loop. Finally, as in hash tables, if we have keys and rids on separate arrays, we use fewer and wider scatters by interleaving the two columns before scattering to the buffers.

**Algorithm 34:** Radix Partitioning - Buffered Shuffling

```
1  O ← prefix_sum(H)                              // partition offsets from histogram
2  i ← 0
3  do
4      k⃗ ← T_k[i]                                  // load tuples from input
5      v⃗ ← T_v[i]
6      i ← i + 𝒲
7      h⃗ ← (k⃗ << s_1) >> s_2                      // compute radix function
8      o⃗ ← O[h⃗]                                   // gather partition offsets
9      c⃗ ← serialize_conflicts(h⃗, O)
10     o⃗ ← o⃗ + c⃗                                  // add serialization offsets
11     O[h⃗] ← o⃗ + 1                               // scatter incremented offsets
12     o⃗ ← o⃗ mod 𝒲                                // buffer offsets in partition
13     m ← o⃗ < 𝒲                                  // find lanes that do not overflow the buffers
14     o⃗′ ← o⃗ + (h⃗ · 𝒲)                           // buffer offsets across partitions
15     B_k[o⃗′] ←_m k⃗                              // scatter tuples to buffer if not overflowing
16     B_v[o⃗′] ←_m v⃗
17     m ← o⃗ = (𝒲 − 1)                            // find lanes pointing to full buffers to be flushed
18     if m ≠ false then
19         P[0] ←_m h⃗                             // (selective) store partition ids with full buffers
20         j ← 0
21         do
22             p ← P[j]
23             j ← j + 1
24             o ← O[p] − (O[p] mod 𝒲) − W         // output location
25             k⃗′ ← B_k[p · 𝒲]                     // load tuples from buffer
26             v⃗′ ← B_v[p · 𝒲]
27             T′_k[o] ← k⃗′                        // (non-temporal) store tuples to output
28             T′_v[o] ← v⃗′
29         while j ≠ |m|
30         m ← o⃗ ≥ 𝒲                              // find overflowing lanes
31         B_k[o⃗′ − 𝒲] ←_m k⃗                      // scatter tuples to buffer
32         B_v[o⃗′ − 𝒲] ←_m v⃗
33 while i ≠ |T|
```

```
int32_t flush_pids[16] __attribute__((aligned(64)));  // buffer to store flushing partition ids
size_t i = 0;
do {  // process 16 tuples at a time
  __m512i key = _mm512_load_epi32(&input_keys[i]);  // load keys and payloads from input
  __m512i val = _mm512_load_epi32(&input_vals[i]);
  __m512i pid = _mm512_srlv_epi32(_mm512_sllv_epi32(key, mask_shift_1), mask_shift_2);  // radix
  __m512i off_O = _mm512_i32gather_epi32(par, offsets, 4);  // gather output offsets
  __m512i off_S = serialize_conflicts(pid, offsets, [...]);  // serialize conflicts (inlined)
  __m512i off_B = _mm512_and_epi32(off_O, mask_15);  // in-buffer offsets
  off_O = _mm512_add_epi32(off_O, mask_1);  // increment output offsets
  off_O = _mm512_add_epi32(off_O, off_S);  // add serialization offsets to output offsets
  off_B = _mm512_add_epi32(off_B, off_S);  // add serialization offsets to buffer offsets
  _mm512_i32scatter_epi32(offsets, par, off_O, 4);  // scatter back incremented output offsets
  __mmask16 k = _mm512_cmplt_epi32_mask(off_B, mask_15);  // find non-overflowing lanes
  buf_O = _mm512_or_epi32(buf_O, _mm512_slli_epi32(pid, 4));  // find offsets across buffers
  // (selective) scatter of keys and payloads to buffer for non-overflowing lanes
  _mm512_mask_i32scatter_epi32(buffer_keys, k, off_B, key, 4);
  _mm512_mask_i32scatter_epi32(buffer_vals, k, off_B, val, 4);
  __mmask16 kk = _mm512_cmpeq_epi32_mask(buf_off, mask_15);  // scattered to last buffer slot
  if (_mm512_kortestz(kk, kk)) continue;  // skip if nothing to flush
  _mm512_mask_packstorelo_epi32(flush_pids, kk, par);  // (selective) store partitions to flush
  size_t j = 0, f = _mm_countbits_64(_mm512_mask2int(kk));  // count flushing lanes
  do {  // flush data from buffer to one partition at a time
    size_t p = flush_pids[j];  // which partition to flush
    size_t o = (offsets[p] & -16) - 16;  // compute output offset of flushing partition
    __m512 key_B = _mm512_load_ps(&buffer_keys[p << 4]);  // load keys from the buffers
    __m512 val_B = _mm512_load_ps(&buffer_vals[p << 4]);  // load payloads from the buffers
    _mm512_storenrngo_ps(&output_keys[o], key_B);  // (non-temporal) store keys to output
    _mm512_storenrngo_ps(&output_vals[o], val_B);  // (non-temporal) store payloads to output
  } while (++j != f);
  k = _mm512_knot(k);  // lanes that would overflow the buffers
  // (selective) scatter of keys and payloads to buffer for overflowing lanes
  _mm512_mask_i32scatter_epi32(&buffer_keys[-16], k, off_B, key, 4);
  _mm512_mask_i32scatter_epi32(&buffer_vals[-16], k, off_B, val, 4);
} while ((i = i + 16) != tuples);
[...]  // cleanup buffers by flushing any remaining tuples
```

The KNC implementation of stable buffered data shuffling for stable radix partitioning of 32-bit keys and 32-bit payloads is shown above. Both the input and the output have to be aligned at cache line boundaries. After the main loop, we synchronize (for multiple threads) and flush any tuples remaining in the buffers. Depending on the data type, the number of items we store in the cache-resident buffer varies. To partition 8-bit, 16-bit, 32-bit, and 64-bit columns, we buffer 64, 32, 16, and 8 items in the buffer of each partition. Specifically for 64-bit columns, since we process $\mathcal{W} = 16$ tuples per iteration but 16 64-bit values can span across two cache lines, we scatter the tuples to the buffers in 3 phases instead of 2. The code is almost identical in AVX-512 except for the change in conflict serialization.

## 5.7 Sorting and Hash Joins

Sorting is used in databases as a subproblem for join and aggregation. Sorting is also used for declustering, index building, compression, and duplicate elimination. Recent work showed that large-scale sorting is synonymous to partitioning. Radixsort and comparison sorting based on range partitioning have comparable performance, by maximizing the fanout to minimize the number of partition passes. Here, we implemented vectorized LSB radixsort, the fastest method for 32-bit keys (see Section 3.4). We enable thread parallelism by executing partitioning in a shared-nothing. By using vectorized buffered partitioning, we also enable data parallelism.

Hash joins are the most frequent operators in analytical queries and can be expensive enough to dominate query execution time. Recent work has focused on comparing main-memory equi-joins, namely sort-merge join and hash join. The former is dominated by sorting [Balkesen *et al.*, 2013a; Kim *et al.*, 2009]. In the baseline hash join, the inner relation is built into a hash table using the join key and the outer relation is probed through the hash table to find matches. Partitioning can be applied to hash join forming multiple variants with different strengths and weaknesses. Here, we design three hash join variants using different degrees of partitioning that also allow for different degrees of vectorization. Because the inputs are much larger than the cache, we use buffered shuffling during partitioning (see Section 5.6.4).

In the first hash join variant termed *no partition*, we completely avoid the use of partitioning. Tuples from the inner relation are inserted in a single hash table shared across all threads. To avoid race conditions, we use atomic `compare_and_swap` operations. The threads are synchronized once after building the hash table. The probing phases does not modify the hash table and does not require atomics. The overall operation is only partially vectorized because atomic operations are not supported in SIMD instructions.

In the second hash join variant termed *min partition*, we use partitioning to eliminate the need for atomics and allow for full vectorization. Before the hash table is built, we partition the inner relation into $\mathcal{T}$ parts, one per thread. Each thread builds its own hash table without atomics. During probing, we pick both which hash table (out of $\mathcal{T}$) to search and which bucket to search within that hash table. The algorithm can be fully vectorized.

We only slightly modify the code to probe across the $T$ hash tables.

In the third hash join variant termed *max partition*, we partition both input relations until the inner relation parts are small enough to fit in a cache-resident hash table. In the initial partitioning phase, we split both relations to $\mathcal{T}$ parts, one per thread. Then, each thread further partitions a part in one or more passes. The resulting parts are used to build and probe hash tables in the cache. The algorithm can be fully vectorized.

## 5.8 Experimental Evaluation

We use three platforms throughout our evaluation shown in Table 5.1. The 61-core KNC co-processor hosts most experiments, providing 512-bit SIMD including both gathers and scatters. We use a mainstream Haswell CPU to compare our SIMD vectorized implementations against scalar and state-of-the-art SIMD implementations. The CPU supports AVX2 256-bit SIMD which includes gathers. However, because a single 4-core Haswell CPU is not the same scale of the 61-core KNC co-processor we use four 8-core Sandy Bridge CPUs with comparable processing power and memory bandwidth in order to measure aggregate performance and power efficiency.

| **Platform Processor(s)** | **1 co-processor** | **1 CPU** | **4 CPUs** |
|---|---|---|---|
| Processor Model (Intel) | Xeon Phi 7120P | Xeon E3-1275v3 | Xeon E5-4620 |
| (Core) Clock Frequency | 1.238 GHz | 3.5 GHz | 2.2 GHz |
| Hardware Threads (Cores × SMT) | $61 \times 4$ | $4 \times 2$ | $(4 \times 8) \times 2$ |
| (Core) Micro-Architecture | KNC (P54C) | Haswell | Sandy Bridge |
| Pipeline Issue Width | 2-way | 8-way | 6-way |
| Reorder Buffer Entries | N/A | 192 | 168 |
| L1 Cache Size per Core | 32 + 32 KB | 32 + 32 KB | 32 + 32 KB |
| L2 Cache Size per Core | 512 KB | 256 KB | 256 KB |
| L3 Cache Size (Total) | N/A | 8 MB | 4 × 16 MB |
| RAM Capacity | 16 GB | 32 GB | 512 GB |
| RAM Load Bandwidth | 212 GB/s | 21.8 GB/s | 122 GB/s |
| RAM Copy Bandwidth | 80 GB/s | 9.3 GB/s | 38 GB/s |
| SIMD Register Width | 512-bit | 256-bit | 128-bit |
| Gather / Scatter Support | Yes / Yes | Yes / No | No / No |
| Thermal Design Power | 300 W | 84 W | 4 × 130 W |

Table 5.1: Experimental platforms for advanced SIMD vectorization experiments

To compile for the KNC co-processor, we use ICC 15 with the `-mmic` flag. We also set the `-no-vec` flag to avoid automatic vectorization. To compile for mainstream CPUs, we use either ICC 15 or GCC 4.9, the fastest per case. We always use `-O3` optimization. For the Haswell CPU, we use `-mavx2` for AVX2 256-bit SIMD. For the Sandy Bridge CPUs we use `-mavx` for `SSE4` 128-bit SIMD with VEX encoding that supports non-destructive 3-operand instructions. The KNC co-processor runs Linux 2.6 as an embedded OS, the Haswell platform runs Linux 3.13 and the Sandy Bridge platform runs Linux 3.2.

Unless specified otherwise, we use all available hardware threads, including SMT, to minimize memory load and instruction latencies. All data are synthetically generated to be uniform random, which matches the most common analytical database benchmarks, such as TPC-H, and are not particularly favorable to any specific operation. As a matter of fact, we already showed that partitioning and sorting are both faster under skew (see Section 3.4).

### 5.8.1 Selection Scans

Figure 5.5 shows the performance of selection scans on the KNC co-processor and the Haswell CPU. The input table consists of 32-bit keys and 32-bit payloads and the selective condition on the key column is $k_{min} \leq k \leq k_{max}$, as shown in Section 5.3, where $k_{min}$ and $k_{max}$ are query constants. We vary the selectivity and measure the throughput of six selection scan versions, two scalar with and without branching [Ross, 2004], and four vectorized using two orthogonal design choices. First, we either select the qualifying tuples by extracting one bit at a time from the bitmask, or use vector selective stores. Second, we either access both keys and payloads during predicate evaluation and buffer all the columns of qualifying tuples, or we load the key column only and buffer the rids of qualifying tuples, used later during buffer flushing to dereference the actual column values.

On the KNC co-processor, vector code is an order of magnitude faster than scalar code, whereas on the Haswell CPU, the speedup is $\sim$ 2X. Low selectivities are faster because loading from RAM is faster than copying, and payload accesses are skipped. Branch elimination improves performance on the Haswell CPU, but decreases performance on the KNC co-processor. On Haswell, all vector variant are similar and saturate the memory bandwidth, while the branchless scalar code catches up on 10% selectivity or more. On KNC,

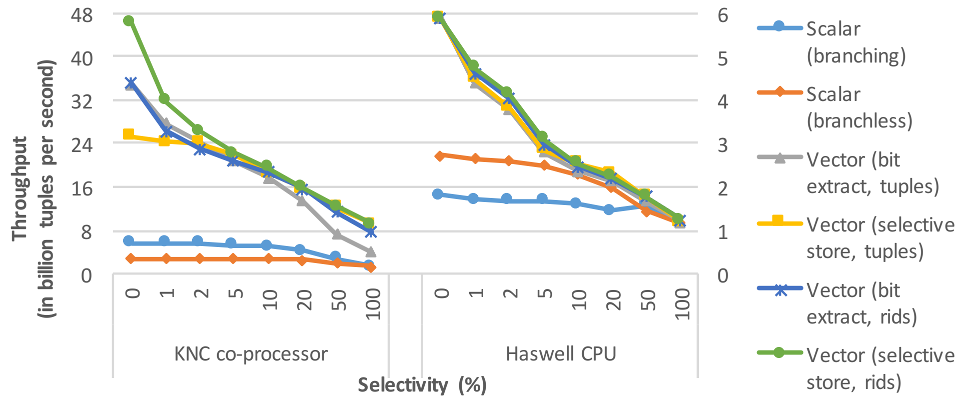



Figure 5.5: Selection scan throughput on KNC and Haswell (32-bit key and 32-bit payload)

accessing the payload columns alongside the keys dominates performance if the selectivity is low, while using selective stores instead of extracting one tuple at a time dominates performance if the selectivity is high. On the simple P54C (Pentium 1) cores of the KNC co-processor, vectorization is essential to saturate the memory bandwidth. On the complex cores of mainstream CPUs, vectorization remains useful if the selectivity is lower than 5%.

### 5.8.2 Hash Tables

In the next sets of experiments, we evaluate hash table vectorization. The first set of experiments measures the probing throughput and compares our approach with other state-of-the-art scalar and vectorized approaches, on both the KNC co-processor and the Haswell CPU. The second set of experiments evaluates building and probing of private (per thread) hash tables, similarly to the last phase of partitioned hash joins, on the KNC co-processor.

In Figure 5.6, we measure the hash table probing throughput using linear probing (LP) and double hashing (DH). The input are the probing 32-bit keys and the output are the 32-bit payload stored in the hash table. The hash table buckets store a 32-bit key and a 32-bit payload and the load factor is 50%. All keys from the input have a matching key in the hash table. We evaluate both horizontal and vertical vectorization. Horizontal vector code refers to bucketized hash table where each input key is compared against multiple table keys [Ross, 2007]. Vertical vector code refers to processing a different input key per vector lane. Our vertical vectorization approach is up to 6X faster on the KNC co-processor, and we achieve a smaller speedup for cache-resident hash tables on the Haswell CPU.

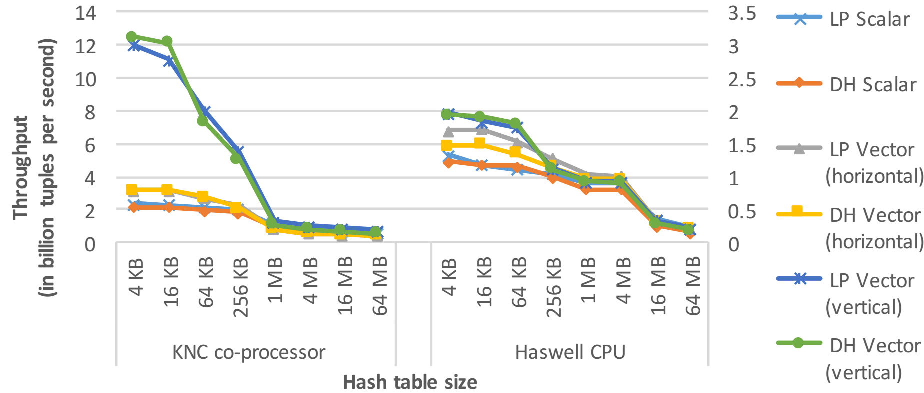



Figure 5.6: Throughput of hash table probing on KNC and Haswell using linear probing (LP) and double hashing (DH) (shared, 32-bit key input → 32-bit payload output)

Figure 5.7 shows the probing throughput of cuckoo hashing, with the same setting as Figure 5.6. The scalar versions are either branching or branchless [Zukowski *et al.*, 2006], and the vector versions are either horizontal or vertical. In the vertical approach, we evaluate two vertical techniques, loading both buckets for every tuple and blending them, or loading the second bucket only if needed. In scalar code, the branchless version [Zukowski *et al.*, 2006] is slower than the branching version on both KNC and Haswell. Notably, the branching version would be faster than the branchless on Haswell if we only compiled with ICC; ICC produced higher quality branching code while GCC produced higher quality branchless code. The version based on vertical vectorization is 5X faster than scalar code on KNC and 1.7X on Haswell, while being consistently better than horizontal vectorization.

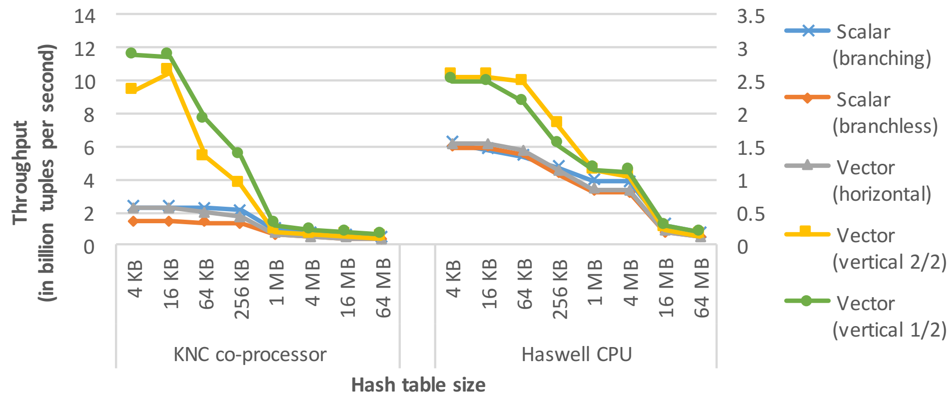



Figure 5.7: Throughput of hash table probing on KNC and Haswell using cuckoo hashing (shared, 32-bit key input → 32-bit payload output)

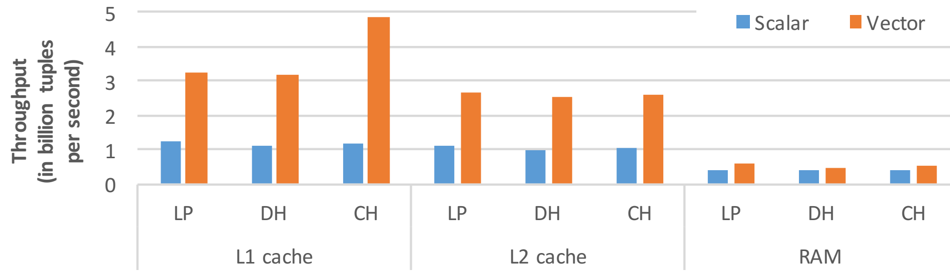



Figure 5.8: Throughput of building and probing hash tables on KNC using linear probing (LP), double hashing (DP), and cuckoo hashing (CH) (1:1 build–probe, private, 32-bit keys and 32-bit payloads both inputs)

In Figure 5.8, we iteratively build and probe private tables on the KNC co-processor. We vary the size of the hash table based on multiple levels of the memory hierarchy. The actual hash table size is 4 KB in L1, 64 KB in L2, and 1 MB out of the cache. The load factor for the hash table is set to 50%. The build to probe ratio is 1:1 and 100% of the input keys have matching keys in the hash table. Both the building and the probing inputs are comprised of 32-bit keys and 32-bit payloads. The output has the matching keys and the two payloads per tuple. In this experiment, throughput is defined as $(|R| + |S|)/t$ ($t$ is the time), including both the probing and the building side in the equation. The speedup we found is 2.6–4X when the hash table is L1-resident, 2.4–2.7X when the table is in the L2, and 1.2–1.4X when the table is larger than the cache.

In Figure 5.9, we repeat the experiment of Figure 5.8 by varying the number of key repeats with the same output size. All tables are L1-resident and the build to probe ratio

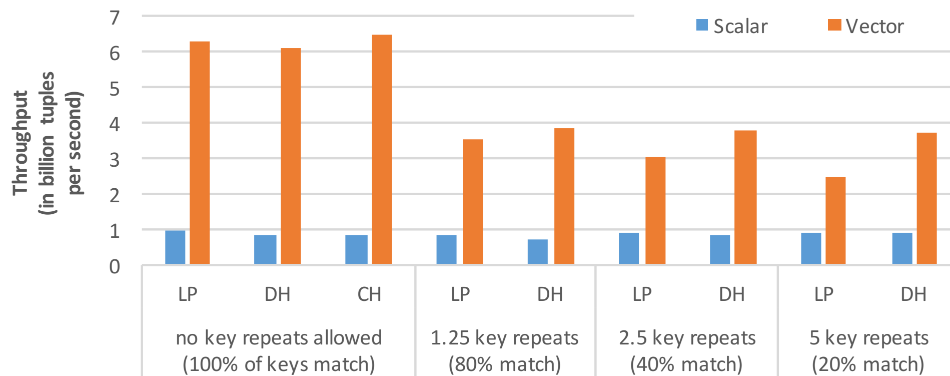



Figure 5.9: Throughput of building and probing on KNC using linear probing (LP), double hashing (DH), and cuckoo hashing (CH) (1:10 build–probe, L1-resident, private, 32-bit keys and 32-bit payloads both inputs)

is changed to 1:10. The other settings are the same as in Figure 5.8. When no keys are repeated, the speedup is 6.6–7.7X and is higher than the speedup in Figure 5.8, since we build 10 times less tuples than we probe and building is generally more expensive than probing. For building alone we measured a vectorization speedup of 2.5–2.7X. If the keys are repeated 5 times, the speedup is 4.1X for DH and 2.7X for LP. Generally, we show that double hashing is more resilient to repeating keys than linear probing.

Notably, the throughput of hash table probing is significantly higher for no repeats compared to 1.25 key repeats for linear probing and double hashing. The reason is that when no repeats are allowed, we optimize the hash table to expect up to a single match per probing key. As a result, we search fewer buckets on average. Note that cuckoo hashing does not support repeating keys without another level of indirection.

### 5.8.3 Bloom Filters

In Figure 5.10, we measure the throughput of Bloom filter probing in both the KNC co-processor and the Haswell CPU. The input comprises of 32-bit keys and 32-bit payloads. The selectivity is set to 5%, we use $k = 5$ hash functions, and the Bloom filter uses approximately 10 bits per inserted tuple. The vectorization speedup we measured is 3.6–7.8X on KNC and 1.3–3.1X on Haswell and is maximized when the Bloom filter is cache-resident. In this experiment the Bloom filter has $2^n$ bits. Otherwise, we need additional instructions to compute the hash functions, widening the performance gap. For instance, without the $2^n$ assumption, the vectorization speedup for L1-resident Bloom filters exceeds 10X on KNC.

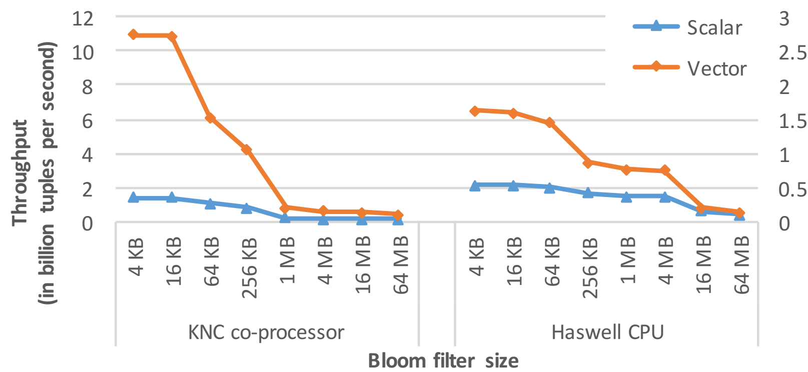



Figure 5.10: Throughput of Bloom filter probing on KNC (5 hash functions, 10 Bloom filter bits / tuple, 5% selectivity, 32-bit key and 32-bit payload)

### 5.8.4 Partitioning

In Figure 5.11, we show radix and hash histogram generation on the KNC co-processor. On the Haswell CPU, the memory load bandwidth is saturated (see Section 3.4). On KNC, scalar code is dominated by the partition function. By replicating each count $\mathcal{W}$ times to avoid scatter conflicts, we get a 2.55X speedup over scalar radix. Performance degrades when we exceed the L1, but we can increase the fanout $\mathcal{P}$ by compressing the replicated histogram to 8-bit counts to reduce its footprint. We can also update the histogram using conflict serialization without replicating the histogram, but this approach is slower, especially if $\mathcal{P} \leq \mathcal{W}$. If $\mathcal{P}$ is large, both count replication and conflict serialization are slow.

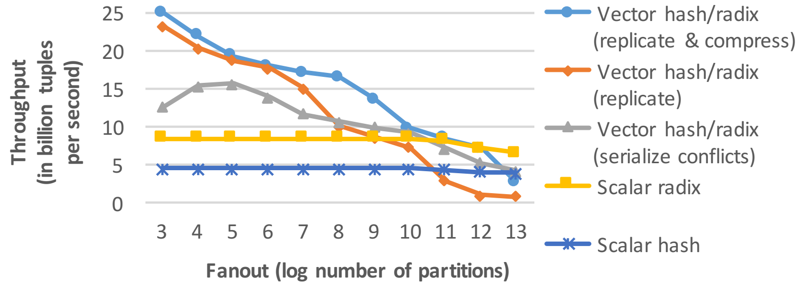



Figure 5.11: Throughput of radix and hash histogram generation on KNC (32-bit keys)

In Figure 5.12, we show the throughput of computing the range partitioning function. Vertical vectorization of binary search achieves a speedup of 7–15X on KNC and 2.4–2.8X on Haswell. The range index (see Section 3.2.5.2) is consistently faster than vertical vectorization on Haswell. On KNC, they are comparable for larger fanout and vertical vectorization

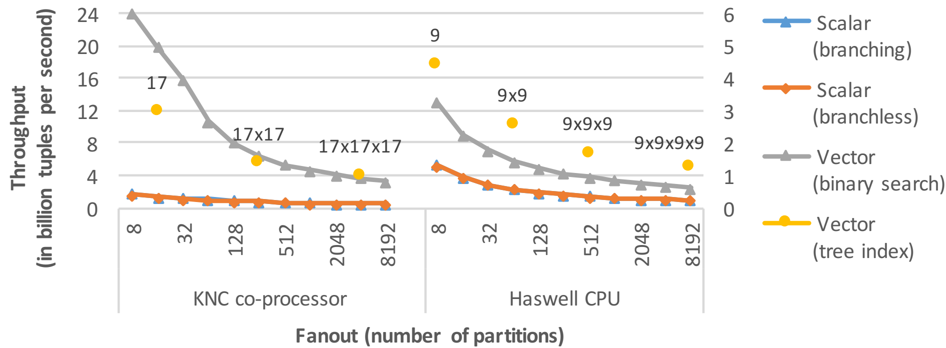



Figure 5.12: Throughput of range partitioning function on KNC and Haswell (32-bit keys)

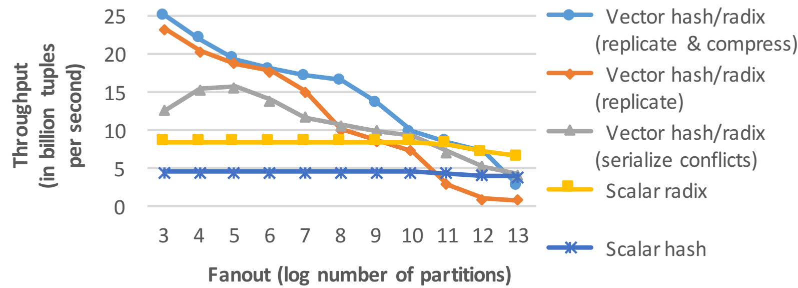



Figure 5.13: Throughput of data shuffling for radix partitioning on KNC (private, out-of-cache, 32-bit key and 32-bit payload)

if faster for lower fanout. The range index interleaves many scalar and SIMD instructions. On the massively superscalar Haswell cores, the scalar instructions are overlapped. On the the non-superscalar KNC cores, simple scalar instructions can saturate the pipeline.

In Figure 5.13, we measure the throughput of data shuffling on KNC, using inputs larger than the cache. On Haswell, data shuffling cannot be fully vectorized without scatters. Nevertheless, partitioning saturates the memory copy bandwidth even for large fanout (see Section 3.4). On KNC, between the unbuffered versions, the vectorized version achieves up to 1.95X speedup. Between the scalar versions, the buffered version achieves up to 1.8X speedup. Between the buffered versions, the vectorized version achieves up to 2.85X speedup and utilizes up to 60% of the memory bandwidth. The optimal fanout that maximizes throughput $\times$ bits, is 5–8 radix bits per pass. Unstable hash partitioning which avoids conflict serialization, can be up to 17% faster than stable radix partitioning.

### 5.8.5 Sorting and Hash Joins

In this part of the experiments, we evaluate sorting and hash joins in three part. First, we measure the performance on KNC and highlight the impact of vectorization on different algorithmic designs for database operations. Second, we compare against four high-end mainstream CPUs with similar aggregate performance bandwidth, in order to highlight the impact of vectorization on power efficiency. Third, we study the impact on performance for a generic vectorized implementation of sorting and hash joins, aiming to support multiple columns with multiple data types of different sizes.

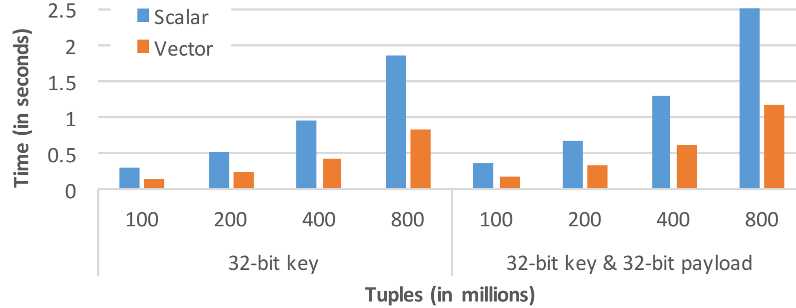



Figure 5.14: Performance of sorting (LSB radixsort) on KNC

#### 5.8.5.1 Vectorization Speedup and Algorithmic Designs

In Figure 5.14, we show the performance of LSB radixsort on KNC. The vectorization speedup is 2.2X over state-of-the-art scalar code. In Figure 5.15, we evaluate the three hash join variants (see Section 5.7) on KNC. The "no-partition" and the "min-partition" variants get small 1.05X and 1.25X speedups, while the fully partitioned versions gets a 3.3X speedup and becomes the fastest that is 2.25X faster than all other methods. The performance gap does not justify hardware-oblivious hash joins on KNC [Jha *et al.*, 2015]. Moreover, hash join is faster than sort-merge join [Balkesen *et al.*, 2013a; Kim *et al.*, 2009], since we sort $4 \cdot 10^8$ in 0.6 seconds and join $2 \times 2 \cdot 10^8$ tuples in 0.54 seconds.

In Figure 5.16, we show the thread scalability of LSB radixsort and partitioned hash join on KNC. The speedup is almost linear to the number of threads, even when we exceed the number of cores by using 2-way and 4-way SMT. On KNC, using 4-way SMT is essential to hide the 4-cycle vector instruction latencies. On mainstream CPUs, LSB radixsort gains only marginal speedup from 2-way SMT.

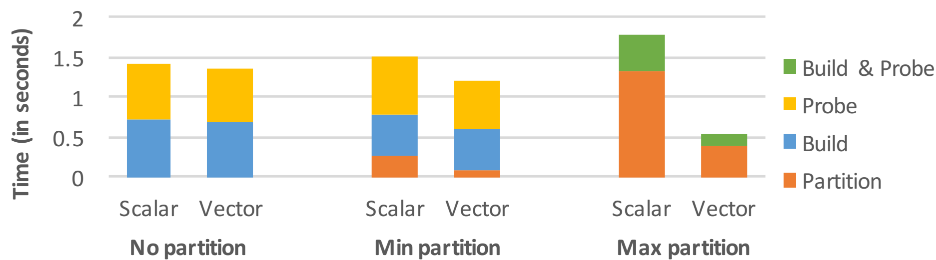



Figure 5.15: Hash join performance on KNC ($2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key and 32-bit payload)

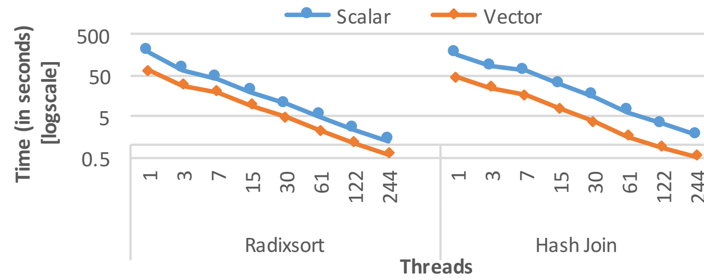



Figure 5.16: scalability of LSB radixsort and partitioned hash join on KNC ($4 \cdot 10^8$ and $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key and 32-bit payload, log/log scale)

#### 5.8.5.2 Aggregate Performance and Power Efficiency

In Figure 5.17, we compare the KNC co-processor with four Sandy Bridge CPUs to get comparable performance, using LSB radixsort and partitioning hash join. Partitioning passes on the Sandy Bridge CPUs are memory bound, thus do not benefit further from vectorization. Both algorithms on the Sandy Bridge CPUs are NUMA aware and transfer the data at most once across NUMA regions (see Section 3.3. Radixsort and hash join are both $\approx$ 14% slower on the Phi compared to the 4 SB CPUs. If we assume the operating power of both platforms to be equally proportional to TDP, both radixsort and hash join are $\approx$ 1.5X more power efficient on the Phi. We also include results after equalizing the two platforms. We set the frequency of the SB CPUs to 1.2 GHz and halve the bandwidth of the Phi to 40 GB/s for copying, which is done by adjusting the code to access twice as

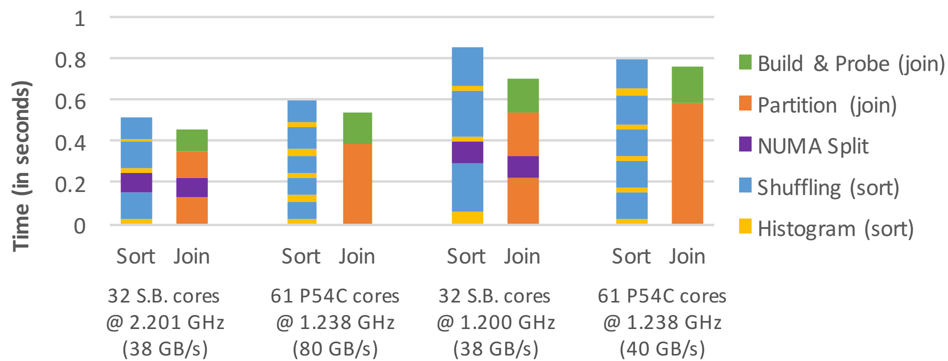



Figure 5.17: LSB radixsort and partitioned hash join on 1 61-core KNC co-processor (Intel Xeon Phi 7120P) vs. 4 8-core Sandy Bridge CPUs (Intel Xeon E5 4620) (sort $4 \cdot 10^8$ tuples, join $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ tuples, 32-bit key and 32-bit payload)

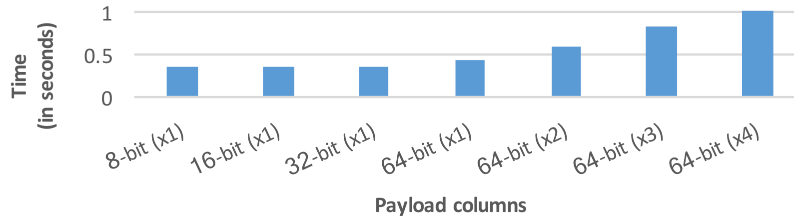



Figure 5.18: Radixsort with varying payloads on KNC ($2 \cdot 10^8$ tuples, 32-bit key)

many bytes as are processed. Phi is then 7% faster for radixsort and 8% slower for hash join. Again, we show that hash joins are faster than sort-merge joins, since we join $2 \times 2 \cdot 10^8$ tuples faster than sorting $4 \cdot 10^8$ tuples alone, also materializing the join output.

#### 5.8.5.3 Materialization of Multiple Columns and Data Types

Vector code cannot handle multiple types as easily as scalar code. So far we used 32-bit columns, which suffices for sorting orders and join indexes of 32-bit keys. Figure 5.18 measures radixsort with 32-bit keys by varying the number and width of payload columns. Per pass, we generate the histogram once and shuffle one column at a time. Shuffling 8-bit or 16-bit columns costs as much as 32-bit columns since we are compute-bound. Also, Xeon Phi upcasts 8-bit and 16-bit operations to 32-bit vector lanes. This approach scales well with wider columns, as we sort 8-byte tuples in 0.36 seconds and 36-byte tuples in 1 second.

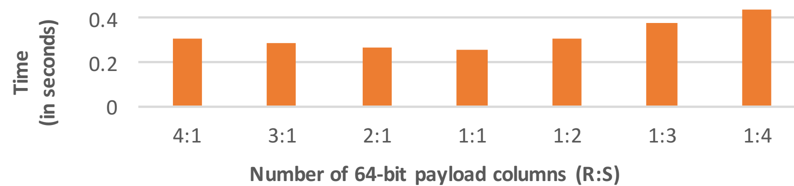



Figure 5.19: Hash join with varying payloads on KNC ($10^7 \bowtie 10^8$ tuples, 32-bit keys)

Figure 5.19 shows partitioned hash join with 32-bit keys and multiple 64-bit payload columns. Out of the cache, we shuffle one column at a time as in Figure 5.18. In the cache, we store rids in hash tables and then dereference the columns. To materialize the payloads, we can partition on the order of the smaller input, and then re-partition to the order of the larger input. This approach is an alternative to radix-decluster [Manegold *et al.*, 2004]. Nevertheless, evaluating the fastest materialization strategy is out of the scope of this work.

## 5.9 SIMD CPUs and SIMT GPUs

The SIMT model in GPUs exhibits some similarity to SIMD in CPUs. SIMT GPUs run scalar code, but "tie" all threads in a *warp* to execute the same instruction per cycle. For instance, gathers and scatters are written as scalar loads and stores to non-contiguous locations. Horizontal SIMD instructions can be supported via special shuffle instructions that operate across the warp. Thus, CPU threads are analogous to GPU warps and GPU threads are analogous to SIMD lanes. However, while conditional control flow is eliminated in SIMD code "manually", SIMT transforms control flow to data flow automatically, by executing all paths and nullifying instructions from paths that are not taken per thread.

One-to-one conversion from SIMD to SIMT code is of limited use for in-memory database operators, due to the vastly different memory hierarchy dynamics. The speedup from SIMD vectorization is maximized for cache-conscious processing, which is achieved by partitioning. On the other hand, GPUs are fast even without partitioning [Kim *et al.*, 2012; Pirk *et al.*, 2011], due to good memory latency hiding. Sequential operators, such as selection scans, have already been studied in detail [Sitaridi and Ross, 2013].

We view KNC and other Xeon Phi platforms as CPU designs. We study the impact of vectorization on making it more suitable for analytical databases. Thus, we do not compare against GPUs and we do not transfer data through the PCI-e bus of KNC in our evaluation.

## 5.10 Conclusion

We introduced advanced SIMD vectorization techniques for in-memory analytical query execution. We defined fundamental vector operations and presented the fundamental vectorization principles. We implemented selection scans, hash tables, and partitioning using entirely vector code, and built sorting and hash join operators. Our implementations were evaluated against scalar and state-of-the-art vector code on mainstream CPUs, as well as the Intel Xeon Phi "Knights Corner" co-processor. In the context of in-memory database operators, we highlighted the impact of vectorization on algorithmic designs, as well as architectural designs and power efficiency, by making simple cores comparable to complex cores. Our work is applicable to all SIMD processors, using either simple or complex cores.

# Chapter 6

# Query Execution Engines for the Many-Core Era

## 6.1 Introduction

Hardware-conscious database design and implementation is a topic of continuous research due to the profound impact of modern hardware advances on query execution. Large memory capacity and multi-core CPUs raised the bar for efficient in-memory execution. Database systems diverged from the one-size-fits-all paradigm to focus on transactional, analytical, scientific, or other workloads. Storage and execution, narrowed down to specific workloads, were redesigned by adapting to the new hardware dynamics.

In analytical databases, column-oriented storage is now a standard design choice, since most queries access a small number of columns from a large number of tuples, in contrast to transactional databases where each transaction update a small number of tuples. However, analytical query execution engines are based on multiple distinctive designs, including column-oriented and row-oriented execution, interpretation and runtime compilation, cache-conscious execution and operator pipelining.

Efficient in-memory execution requires low interpretation cost, optimized memory access, and high CPU efficiency. Low interpretation cost is coupled with high instruction-level parallelism and is achieved by processing entire columns [Manegold *et al.*, 2000a], batches of tuples per iterator call [Boncz *et al.*, 2005], or by generating and compiling query-specific

code [Kim *et al.*, 2010; Neumann, 2011; Gupta *et al.*, 2015]. Memory access can be optimized by pipelining operators to avoid materializing the pipelined stream [Graefe, 1994], or by using partitioning to avoid cache and TLB misses [Manegold *et al.*, 2000b]. Data parallelism is achieved via SIMD vectorization. Linear-access operators such as scans and compression [Zhou and Ross, 2002; Willhalm *et al.*, 2009; Lang *et al.*, 2016], are naturally data-parallel and easy to vectorize. For some operators such as sorting, the common approach is to use ad-hoc SIMD optimizations [Inoue *et al.*, 2007; Chhugani *et al.*, 2008; Inoue and Taura, 2015]. In Chapter 5, we introduced advanced SIMD vectorization that cover non-linear-access operators, such as hash tables and partitioning.

Since the advent of many-core platforms known as Intel Xeon Phi, the design tradeoff between many simple cores and fewer complex cores is being revisited. Fewer complex cores are the basis of mainstream CPUs, whereas many-core platforms are optimized for high FLOPS and marketed for HPC. In Chapter 5, we showed that data parallelism via advanced SIMD vectorization in software, can make up for the lack of instruction-level parallelism provided by super-scalar out-of-order cores in hardware. However, we focused on vectorizing core algorithms, such as hash tables, and basic database operators using key–rid pairs. Going from operators over key–rid pairs to a more realistic setting, where selection scans, hash joins, and group-by aggregation operators support multiple columns, data types, compression, as well as complex predicates and expressions, requires us to redesign the operators and the query engine as a whole.

The latest Intel Xeon Phi CPUs codenamed MIC Knights Landing are more interesting than the first generation of many-core co-processors that we used in Chapter 5, since they (i) are not co-processors connected via PCI-e but stand-alone CPUs that access DRAM directly, (ii) provide a multi-channel DRAM (MCDRAM) memory on-chip component that has higher bandwidth than regular DRAM and larger capacity than the caches, and (iii) are more similar to the upcoming mainstream CPUs compared to the first generation of many-core co-processors since they have super-scalar out-of-order cores and support the same 512-bit SIMD instruction set (AVX-512).

In this chapter, we explore analytical query engine designs using the latest many-core CPUs. Initially, we show that code generation and operator pipelining, despite being suc-

cessful on mainstream CPUs, diminish the benefit of SIMD vectorization due to being memory-bound. Also, random memory accesses that are inherent in operator pipelining underutilize the on-chip MCDRAM, since MCDRAM has higher bandwidth than regular DRAM but similar latency. To better utilize the hardware features of the many-core platform, we introduce VIP[1], an analytical query execution engine built bottom-up from pre-compiled data-parallel sub-operators, implemented entirely in SIMD, and utilizing the high-bandwidth memory to facilitate cache-conscious execution. We discuss selection scans with compression, hash joins, and group-by aggregation operators. In order to support multiple columns, data types, and complex predicates efficiently, without generating query-specific code at runtime, operators invoke pre-compiled sub-operators. Each sub-operator is designed and implemented using advanced SIMD vectorization techniques, and processes one column at a time [Manegold *et al.*, 2000a] from one block of tuples at a time [Boncz *et al.*, 2005], in order to amortize the interpretation cost and remain cache-resident. In our experimental evaluation using synthetic and TPC-H queries, we outperform hand-written query-specific code based on state-of-the-art designs [Kim *et al.*, 2010; Neumann, 2011], without including the runtime compilation overhead of code generating engines that is non-existent in the VIP design.

The rest of the chapter is organized as follows. In Section 6.2, we summarize the necessary background on many-core CPUs and analytical query engines. In Section 6.3 we discuss vectorized code generation. In Section 6.4, we describe VIP, our proposed query engine design. In Section 6.5, we present our evaluation and conclude in Section 6.6.

## 6.2 Background

### 6.2.1 Mainstream & Many-Core Processors

We now briefly summarize mainstream and many-core CPUs. Intel Xeon Phi CPUs, the latest generation of many-core (MIC) CPUs codenamed Knights Landing (KNL) offer 64–72 cores and use 215–260 Watts TDP. The latest mainstream CPUs offer up to 22 cores per socket and use 145 Watts TDP. Mainstream CPUs evolved from single-core CPUs (Pentium

---

[1] Based on ***V**ectorization*, ***I**nterpretation*, and ***P**artitioning*.

4) focused on instruction-level parallelism. MIC cores initially focused on data parallelism with wider SIMD registers and an advanced SIMD ISA, but were not super-scalar and had no out-of-order execution. Both KNL CPUs and the upcoming mainstream CPUs support the same SIMD ISA (AVX-512). Also, KNL cores are super-scalar with out-of-order execution, even if not as aggressively as mainstream cores, e.g. a KNL core has 72 reorder-buffer entries, while a Skylake core has 224. Mainstream CPUs still have fewer and faster cores than many-core CPUs, but the two designs are more similar today compared to the first generation of MIC platforms.

The memory hierarchy of the two platforms differs considerably. Mainstream CPUs use private L1 and L2 caches, and up to 128 MB of L3 shared across all the cores per socket. Many-core CPUs use a private L1, an L2 that is shared across 2 cores, and 16 GB of multi-channel DRAM (MCDRAM) on-chip. MCDRAM has ∼4X higher bandwidth than regular DRAM. On a Intel Xeon Phi 7210 (KNL) CPU, we measured the bandwidth of DDR4 DRAM at ∼70 GB/s and the bandwidth of the on-chip MCDRAM at ∼295 GB/s. Note that the cores of a single many-core CPU do not access all regions of memory at the same speed, similarly to multiple mainstream CPUs with multiple NUMA regions.

Measuring the memory bandwidth using contiguous loads and stores is easy. Measuring random accesses both in and out of the cache, however, is trickier when we also need to account for the effect of SIMD non-contiguous loads (gathers) and stores (scatters). The micro-benchmark we use to measure random accesses is shown below. It computes the `xor` of an array with $2^k$ elements, traversed via an odd stride. We vectorize the same loop by accessing $\mathcal{W}$ elements at a time using SIMD gathers.

```
int32_t random_traversal(const int32_t* a, size_t n) {
  const size_t m = n - 1;  // mask to fix array bounds overflows
  const size_t o = rand() * 2 + 1;  // random odd stride
  size_t i = 0;  // initial (and final) index
  int32_t x = 0;  // xor value of array elements
  do {  // access one array element at a time
    x ^= a[i];  // load and xor the current array element
    i += o;  // add odd stride to index
    i &= m;  // fix array bounds overflow
  } while (i);  // until we traverse all 2^k array elements
  return x;
}
```

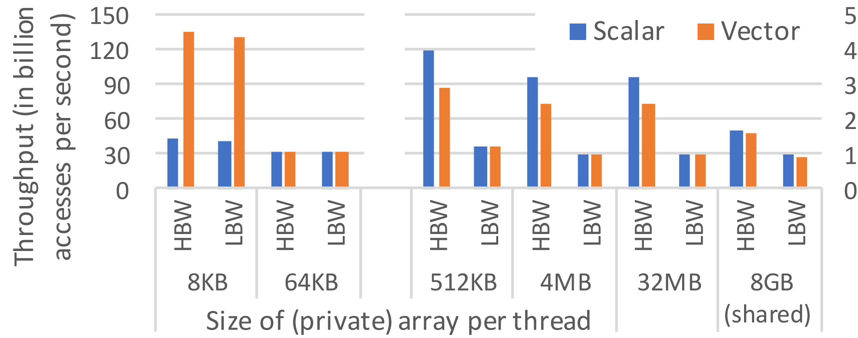



Figure 6.1: Throughput of non-sequential accesses

Figure 6.1 shows the throughput of non-sequential memory accesses using the described micro-benchmark on a Xeon Phi 7210 CPU. We use all 256 hardware threads (see Section 6.5 for platform details). Each thread traverses a private array to minimize the memory latency, except for the last two columns where the arrays are shared across all threads. The arrays are either on DRAM (`LBW`) or on MCDRAM (`HBW`). If the arrays fit in the L1 cache, SIMD gathers are ∼3X faster than scalar loads. If the arrays are in the L2 cache, there is virtually no speedup from SIMD gathers compared to scalar loads.

If the arrays exceed the cache capacity, the throughput drops by an order of magnitude (using right $y$ axis in Figure 6.1). The high-bandwidth on-chip MCDRAM is at least 3X faster than off-chip regular DRAM. However, scalar loads are now up to 40% faster than SIMD gathers because out-of-order execution overlaps instructions with scalar loads. On the other hand, SIMD gathers issue 16 loads per thread, immediately saturating the load buffers of the core. The out-of-order execution window is too small to overlap with the hundreds of cycles to fetch all $\mathcal{W}$ cache lines from memory and the pipeline stalls.

When the arrays reside on MCDRAM, traversing a shared 8GB array reduces the random access throughput by half compared to traversing 256 thread-private 32MB arrays, highlighting the NUMA effect between cores of the same many-core CPU.

The results of this benchmark alongside with the results of Section 6.3 will explain why cache-conscious query execution is favored by many-core platforms and why it can utilize the high-bandwidth on-chip MCDRAM memory more effectively than engine designs based on non-partitioned operators bound by random memory accesses out of the cache.

### 6.2.2 Analytical Query Execution Engines

Transactional databases prefer row-oriented storage and execution, since each transaction updates a small number of tuples. Analytical databases have largely standardized the column-oriented storage since analytical queries access a small number of columns from a large number of tuples. However, state-of-the-art analytical query engines can use either column-oriented or row-oriented execution.

Disk-based DBMSs typically use the row-oriented Volcano model [Graefe, 1994] where each operator is implemented as an iterator. Each call to the iterator of the operator yields one output tuple, in turn calling iterators of the lower operators to fetch input tuples. This pull-based design avoids materialization by pipelining operators, but suffers from high interpretation overhead due to relying on virtual function calls that perform poorly on modern CPUs [Neumann, 2011], as well as shared hash tables that typically exceed the cache size and thus incur excessive cache and TLB misses.

In MonetDB [Manegold *et al.*, 2000a], queries are split into operators that process entire columns and materialize intermediate results in RAM. Column-at-a-time execution strictly applied on every micro-operation can incur prohibitive materialization costs, e.g., `select sum(l_extendedprice * (1 + l_tax) * (1 - l_discount)) from lineitem`. Cache and TLB misses can be eliminated via cache-conscious execution [Manegold *et al.*, 2000b], although the materialization cost is increased due to additional partitioning passes.

MonetDB/X100 [Boncz *et al.*, 2005] extends the Volcano iterators to process batches of tuples (termed *vectors*), amortizing the interpretation cost. The design can be either column-oriented or row-oriented, depending on the processing order of tuples and columns within each batch. The batches passed between iterators can remain cache-resident. However, cache and TLB misses due to using large hash tables are still present.

HyPer [Neumann, 2011] and Redshift [Gupta *et al.*, 2015] pipeline operators and convert them into push-based nested loops by generating query-specific code. The pipelined stream is not materialized and there is no interpretation cost. Then, random memory accesses become the dominant performance factor. Compilation time is mitigated in Hyper by generating LLVM code. This design generates row-oriented code, but can use column-oriented pre-compiled code for indexes and table scans [Lang *et al.*, 2016].

## 6.3 Vectorized Code Generation

We now evaluate whether SIMD vectorization works well alongside code generation and operator pipelining by using an example query. Note that most queries cannot be fully vectorized and SIMD code generation cannot support the same properties as scalar code, such as *register resident* execution [Neumann, 2011], while maintaining data parallelism.

The example query template, shown below, performs a range selection on each table using constants and joins a fact table `F` with multiple dimension tables based on a different attribute of the fact table. The output is a single aggregate using columns from every table. The fact table is larger than the dimension tables and the query plan pipelines the joins.

```
select sum(f.val * a.val * b.val * [...])  -- single group
  from F f, A a, B b, [...]  -- any number of dimension tables
 where f.key_A = a.key and f.key_B = b.key and [...]  -- joins
   and f.val between x0 and y0  -- selection on fact table
   and a.val between x1 and y1  -- selection on dimension table
   and b.val between x2 and y2 and [...];
```

We generate $k + 1$ loops to execute the left-deep plan of the example query assuming $k$ dimension tables. The first $k$ loops evaluate the selection on each dimension table and materialize the projection $\langle$`key`, `val`$\rangle$ in a hash table. The last loop evaluates the selection on the fact table, joins every dimension using $k$ nested loops, and computes the sum aggregate on an arithmetic expression that combines attributes from all tables.

### 6.3.1 Scalar Code Generation

The generated code for the above query assuming two dimension tables is shown below. We use linear probing to handle hash table collisions. The data structure of the hash buckets is also part of code generation. The new data type per operator holds the attributes of the output projection for that operator, as dictated by the query plan. During compilation, the assembly that is generated, is specialized to the data type, similarly to C++ templates.

```
typedef struct {
  int set:1;  // indicates if the bucket is set (empty otherwise)
  A_key_t A_key;  // type & value of A.key column
  A_val_t A_val;  // type & value of A.val column
} hash_join_bucket_A;  // bucket of hash join table (HJT_a)
```

```
size_t i = 0;  // index to F base columns
do {  // linear scan over F
  F_val_t f_val = F_val[i];  // load f.val
  if (f_val >= x0 && f_val <= y0) {  // selection predicate
    A_key_t f_key_A = F_key_A[i];  // load f.key_A
    size_t h1 = hash(f_key_A, buckets_HJT_a);
    while (HTJ_a[h1].set) {  // hash join f with a
      if (HTJ_a[h1].key == f_key_A) {  // join predicate of F and A
        B_key_t f_key_B = F_key[i];  // load f.key_B
        size_t h2 = hash(f_key_B, buckets_HJT_b);
        while (HJT_b[h2].set) {  // hash join f with b
          if (HJT_b[h2].key == f_key_B) {  // join predicate of F and B
            sum += f_val * HJT_a[h1].val * HJT_b[h2].val;  // update sum(f.val * a.val * b.val)
          }
          if (++h2 == buckets_HJT_b) h2 = 0;  // go to next hash bucket for B
      } }
      if (++h1 == buckets_HJT_a) h1 = 0;  // go to next hash bucket for A
  } }
} while (++i != tuples);
```

The hash buckets contain the `key` and the `val` attribute of each dimension table. Besides the projected columns, we can have additional fields, such as `null` bits or next pointers if we use hash chaining instead of linear probing. The `set` field is used to atomically fill the bucket via atomic compare-and-swap during the hash table building phase, since the hash table is shared across all threads.

To utilize thread parallelism, we divide the input equally among threads. Dynamic sharing achieves better parallelism on mainstream CPUs [Leis *et al.*, 2014], but on a many-core CPU, there is no uniform memory access latency and we do not control the placement. To maximize the memory throughput, we also split the shared hash table among threads. During probing, we determine which part of the hash table to probe by accessing the L1, having marginal effect on the total latency.

### 6.3.2 SIMD Code Generation

Data-parallel instructions execute the same operation for $\mathcal{W}$ tuples, assuming we fit $\mathcal{W}$ values per SIMD register. While selection scans are inherently data parallel, hash joins probe one tuple at a time from hash tables and pipeline multiple operators in nested loops. This tight integration of nested loops allows the outer stream of tuples to remain *register resident* [Neumann, 2011], i.e., the column values are kept in CPU registers.

Register-resident execution is impractical in SIMD. In this example, $0 < \mathcal{W}' \leq \mathcal{W}$ tuples satisfy the selective predicate, then $0 < \mathcal{W}'' \leq \mathcal{W}'$ find a match in the first probed bucket, and so on. Assuming the last $\mathcal{W}$ tuples loaded from `F` are pushed through all pipelined operators, if the selectivity of any operator is low, or we do not have nearly perfect hashing for joins, most SIMD lanes will be processing invalid tuples.

To ensure we retain data parallelism throughout query execution, each operator stores intermediate results in cache-resident buffers. Each operator loads data from the buffer of the previous step. The buffers are smaller than block-at-a-time execution [Boncz *et al.*, 2005] to remain in the L1 cache, but larger than $\mathcal{W}$ to guarantee data parallelism. The operators are nested similarly to scalar code without any function calls, albeit the next (inner) operator is executed once the output buffer of the previous (outer) operator is full.

The code for the `F` selection is shown below using AVX-512 SIMD intrinsics. We scan `F.val`, evaluate the predicates and store the rids of qualifying tuples in a cache-resident buffer. Here, we set the `key` columns to be 64-bit integers and the `val` columns to be 64-bit floating point numbers to simplify the computation of the aggregate expression.

```
const __m512d m_x0 = _mm512_set1_pd(x0);  // vector masks
[...] __m512i f_rid_0 = _mm512_set_epi64(7,6,5,4,3,2,1,0);
size_t i = 0, o1 = 0, o2 = 0, o3 = 0;  // number of tuples held in each buffer
do {  // scan over F
  __m512i f_val_0 = _mm512_load_pd(&F_val[i]);  // load F.val column
  __mmask8 k0 = _mm512_cmp_pd_mask(f_val_0, mask_x0, _CMP_GE_OQ);  // evaluate x0 <= F.val <= y0
  k0 = _mm512_mask_cmp_pd_mask(k0, f_val_0, mask_y0, _CMP_LE_OQ);
  _mm512_mask_compressstoreu_epi64(&L1_f_rid[o1], k0, f_rid_0);
  o1 += _mm_popcnt_u64(k0);  // store rids of qualifying tuples
  if (o1 > buffer_size - 8) { [...] }  // go to next operator if the buffer is full
  f_rid_0 = _mm512_add_epi64(f_rid_0, mask_8);  // update input rids
} while ((i = i + 8) != F_tuples);
```

Hash probing in SIMD uses *vertical vectorization*; we process a different input item and gather a different hash bucket per SIMD lane (see Section 5.4). To avoid SIMD lane underutilization, we replace the finished items with new input items on each iteration. To avoid accessing the keys and computing the hash function on multiple iterations, we load the rids from the buffer, gathers the keys, compute the hashes, and store the hash table locations and the keys in the buffer. The SIMD code that probes the `A` hash table buckets is shown below. For simplicity, we assume that negative keys denote empty buckets. We store both rids to F and pointers to the hash table of `A` to the output buffer in order to

materialize the `val` columns. The following join with dimension `B` is almost identical, with the addition of pointers to the `A` hash table as an extra payload, alongside `F` rids.

```
__m512i f_rid, f_key_A, a_ptr;  // operator state
size_t i1 = 0, j1 = 8;  __mmask8 k1 = 255;  // set mask
do {  // while all SIMD lanes are used
  a_ptr = _mm512_mask_expandloadu_epi64(a_ptr, k1, &L1_a_ptr[i1]);  // load pointer to A hash table
  f_rid = _mm512_mask_expandloadu_epi64(f_rid, k1, &L1_f_rid[i1]);  // load rids to F
  f_key_A = _mm512_mask_expandloadu_epi64(f_key_A, k1, &L1_A_key[i1]);  // load f.key_A values
  i1 += j1;  // update input buffer (level 1 buffer) index
  __m512i a_key = _mm512_i64gather_epi64(a_ptr, 0, 8);  // gather a.key from hash table of A
  k1 = _mm512_cmpeq_epi64_mask(a_key, f_key_A);  // evaluate f.key_A = a.key
  _mm512_mask_compressstoreu_epi64(&L2_f_rid[o2], k1, f_rid);  // store F rids for qualifiers
  _mm512_mask_compressstoreu_epi64(&L2_a_ptr[o2], k1, a_ptr);  // store A pointers for qualifiers
  o2 += _mm_popcnt_u64(k1);  // update output buffer (level 2 buffer) index
  if (o2 > buffer_size - 8) { [...] }  // go to next operator (level 2) if the buffer is full
  a_ptr = _mm512_add_epi64(a_ptr, mask_16);  // point to next hash table bucket (linear probing)
  k1 = _mm512_kor(k1, _mm512_cmplt_epi64_mask(a_key, mask_0));  // find finished lanes
  j1 = _mm_popcnt_u64(k1);  // count finished lanes to reuse
} while (i1 + j1 <= o1);  // move remaining items to front of the buffer after the loop
```

In the last and inner-most loop, we load the rids to `F` and the pointers to the hash tables of `A` and `B` from the buffers, access the colmn values, compute the arithmetic expression, and update the single sum aggregate in SIMD registers.

```
size_t i3 = 0;
do {
  __m512i f_rid = _mm512_load_epi64(&L3_f_rid[i3]);  // load rids to F
  __m512i f_val = _mm512_i64gather_epi64(f_rid_3, F_val, 8);  // gather F.val column
  __m512i a_ptr = _mm512_load_epi64(&L3_a_ptr[i3]);  // load hash table pointers to A
  __m512i a_val = _mm512_i64gather_pd(a_ptr, 8, 8);  // gather A.val column
  __m512i b_ptr = _mm512_load_epi64(&L3_b_ptr[i3]);  // load hash table pointers to B
  __m512i b_val = _mm512_i64gather_pd(b_ptr, 8, 8);  // gather B.val column
  sum = _mm512_fmadd_pd(_mm512_fmadd_pd(f_val, a_val), b_val, sum);  // update the sum aggregate
} while ((i3 = i3 + 8) != buffer_size - 8);
```

We now run the example query by varying the number of dimension tables to highlight operator pipelining. For 1 join, `F` has $2^{29}$ tuples and `A` has $2^{27}$ tuples. For multiple joins, each dimension has half the tuples from the previous dimension. The smallest dimension is joined first. We place the whole dataset either on DRAM (`LBW`) or on the on-chip MCDRAM (`HBW`) and compare. The table sizes were chosen so that the working set fits in the 16GB on-chip MCDRAM. The column values are uniform random with no correlation between the attributes. The hash function we use is fully vectorized. We measure the last loop that scans `F`, joins with 1–4 dimension tables in 1–4 inner loops, and computes the sum aggregate. The

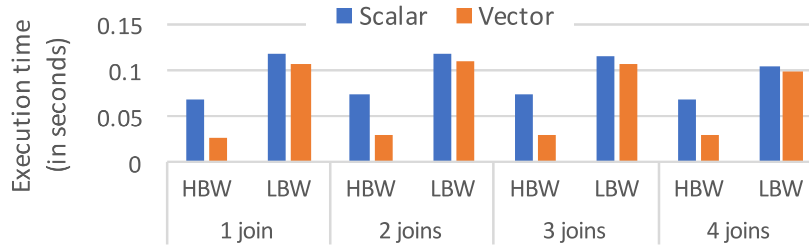



Figure 6.2: Running time of example query using code generation (10% selectivity)

loops that build the hash tables are not included in the comparison, since atomics are not supported in SIMD. Another notable operation that cannot be fully vectorized is group-by aggregation using a shared hash table to update the partial aggregates. In general, not all database operators are data-parallel and thus cannot be fully vectorized.

In Figure 6.2, we set the selectivity for all selections to 10%. The selection scan now dominates the execution time. When the dataset resides on the on-chip MCDRAM, SIMD code is 2.5X faster than scalar code. However, when the dataset resides on regular DRAM, execution becomes memory bound and the SIMD speedup is less than 10%. For scalar code, the speedup due to using MCDRAM compared to regular DRAM is up to 1.75X. For SIMD code, the speedup due to using MCDRAM is 3.93X.

In Figure 6.3, we repeat the experiment with 90% selectivity. Execution times are now an order of magnitude higher. Since the hash tables are larger than the cache, we suffer almost as many cache misses as the tuples in `F` times the number of joins. SIMD code is up to 20% better than scalar code. The SIMD speedup is marginal here because random memory accesses dominate the execution time. Storing the dataset on the on-chip MCDRAM instead of DRAM, improves performance by up to 60%. The speedup matches the result in Figure 6.1 for random accesses on shared arrays.

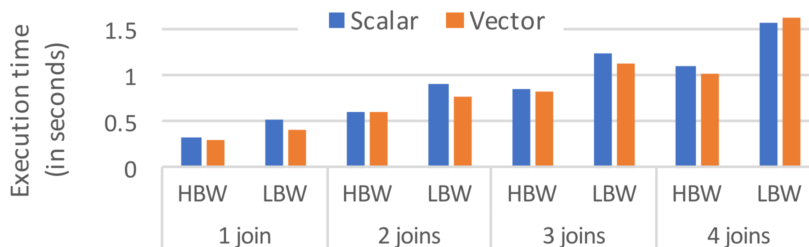



Figure 6.3: Running time of example query using code generation (90% selectivity)

Query engines based on code generation coupled with operator pipelining have been used extensively, even if the design has been shown to exhibit very low IPC on mainstream CPUs [Sirin *et al.*, 2016]. Nevertheless, since query execution in the design is predominately memory bound, the potential speedup due to SIMD vectorization is diminished. On many-core CPUs, the on-chip MCDRAM memory is also underutilized since the 1.6X speedup we can achieve for executing random accesses to shared memory is much lower than the 4.3X difference in bandwidth that is achievable for sequential accesses.

## 6.4 Proposed Query Engine Design

In this section, we present the design of VIP, our proposed analytical query execution engine, alongside core implementation details. In VIP, we use column-oriented execution [Manegold *et al.*, 2000a; Stonebraker *et al.*, 2005], but never process an entire column at once. Instead, we process one column for the next block of tuples, retaining intermediate results in the cache [Boncz *et al.*, 2005]. Therefore, we amortize the interpretation cost, since we use pre-compiled manually optimized SIMD code instead of per-query code generation [Gupta *et al.*, 2015; Kim *et al.*, 2010; Neumann, 2011]. In contrast to the original block-at-a-time design [Boncz *et al.*, 2005], which is based on Volcano iterators [Graefe, 1994], we extend the column-at-a-time model that separates operators [Manegold *et al.*, 2000a]. Allocation of work units between threads is static due to the NUMA effect. Finally, we use cache-conscious hash joins and aggregations to avoid random memory accesses, using the on-chip MCDRAM to accelerate the partitioning passes.

In the VIP design, each operator is built bottom-up from column-oriented data-parallel *sub-operators* that are implemented entirely in SIMD. During query execution, each query operator invokes pre-compiled sub-operators to processes one column at a time for one block of tuples at a time. The block size is chosen to amortize the cost of interpretation and retain the working set in the fast private cache of the CPU core. Sub-operators are not just implemented in SIMD but are designed based on advanced vectorization principles, and execute a simple operation on one payload column of a specific type. This allows for extreme manual optimization per sub-operator without affecting the rest of the engine.

To explain how sub-operators work across columns, consider a hash join on a composite key from multiple columns. To hash the key columns, we invoke the hash sub-operator for the 1st column based on its data type and store the hash values in the cache. Then, we call the hash sub-operator for the 2nd column and update the hash values for the same block of tuples. If the block size is small enough, the hashes remain in the cache. The sub-operator to compute the FNV hash [Fowler *et al.*, 2017] of a 32-bit column is shown below.

```
void hash_int32(int32_t* data, uint32_t* hash, size_t tuples) {
  size_t i = 0;
  if (tuples >= 16) {
    const __m512i mask_255 = _mm512_set1_epi32(255);  // constant mask to isolate mask byte
    const __m512i mask_fnv = _mm512_set1_epi32(16777619);  // constant mask with FNV prime
    size_t tuples_16 = tuples & -16;
    do {  // process 16 column values at a time in SIMD code
      __m512i h = _mm512_load_epi32(&hash[i]);  // load 32-bit hashes (from the cache)
      __m512i d = _mm512_load_epi32(&data[i]);  // load data column (32-bit integer here)
      size_t j = 4;  // FNV hashing hashes 1 byte at a time
      do {  // process 1 byte at a time (unrolled)
        h = _mm512_ternarylogic_epi32(h, d, mask_255, 120);  // isolate lower byte and xor with hash
        h = _mm512_mullo_epi32(h, mask_fnv);  // multiply with FNV prime
        d = _mm512_srli_epi32(d, 8);  // shift next byte of key
      } while (--j);
      _mm512_store_epi32(&hash[i], h);  // store back 32-bit hashes (to the cache)
    } while ((i = i + 16) != tuples_16);
  }
  [...] // process remaining 15 or less values in scalar code
}
```

The hash sub-operator must be implemented for every supported data type. The sub-operators that execute the same operation across data types have similar prototypes: `void hash_T(T* data, uint32_t* hash, size_t tuples)`.

### 6.4.1 Selection Scans

In modern analytical databases, linear scans are preferred them over indexes for selections [Raman *et al.*, 2013]. Branch mispredictions have been at the core of selection scan performance [Ross, 2004]. Selection scans are by definition data-parallel and thus easy to vectorize. Code generating engines also prefer pre-compiled SIMD code for scans and decompression [Lang *et al.*, 2016]. In column-at-a-time execution [Manegold *et al.*, 2000a], each predicate produces an rid list (i.e., array) for qualifying tuples. Conjunctions and disjunctions are computed via the union or intersection of rid lists.

In our design, we use bitmaps to represent intermediate results for combinations of predicates. In contrast to unions and intersections, bitwise operations are fast and maximally data-parallel. To maximize the memory bandwidth given the NUMA effect, we statically partition among threads. The output tuples retain the same order as the input. Dynamic work allocation via a shared counter is also possible.

For complex expressions of predicates, exhaustively evaluating every selective predicate on every column and then merging the bitmaps is trivial. However, for some queries, always evaluating every column that appears on the selective predicate can be orders of magnitude slower. Instead, we want to use the output of earlier predicates to skip evaluating later predicates and skip loading columns for subsets of the input. In conjunctions, we can skip the tuples for which an earlier predicate was evaluated as false. In disjunctions, we can skip tuples for which an earlier predicate was evaluated as true.

On linear scans, performance is dictated by the number of accessed cache lines. SIMD naturally handles predicates on $W$ contiguous values very efficiently, thus, we evaluate or skip in units of $W$ tuples. The sub-operator to evaluate comparison predicates on 32-bit integer columns is shown below.

```
void select_int32(const uint16_t* bitmap_in,  // bitmap of undetermined tuples
    uint16_t* bitmap_out,  // bitmap of qualifiers (not strict subset of input bitmap)
    const int32_t* data, size_t tuples, int32_t constant, int op) {
  const __m512i mask_constant = _mm512_set1_epi32(constant);  // mask of query constant
  size_t i = 0;
  switch (op) {  // separate code per comparison operator
    case '>':  // code for ">" operator
      do {  // scan over block
        __m256i bit = _mm256_load_si256(bitmap_in);  // check 256 bits of input bitmap
        if (!_mm256_testz_si256(b, b)) {  // skip 256 tuples if all 256 bits are 0
          __m512i bit_E = _mm512_cvtepu16_epi32(bit);
          uint64_t m = _mm512_test_epi32_mask(bit_E, bit_E);  // find groups of 16 tuples not all 0
          do {  // evaluate 16 tuples at a time that are not all 0
            size_t j = _tzcnt_u64(m);  // locate the 16 next tuples within 256 tuples
            __m512i key = _mm512_load_epi32(&data[j << 4]));  // load column values
            bitmap_out[j] = _mm512_cmpgt_epi32_mask(key, mask_constant);  // evaluate the predicate
          } while (m = _blsr_u64(m));  // go to next 16 tuples not all 0
        }
        bitmap_in += 16, bitmap_out += 16, data += 256;  // progress pointers
      } while ((i = i + 256) != tuples);
      break;
    case '=': [...]  // code for "=", "<", "<>", ">=", and "<="
} }
```

The input bitmap denotes which tuples can be skipped and which tuples we still need to evaluate. The output bitmap is the evaluation result. Note that the output bitmaps do not have to be strict subsets of the input bitmaps. We express complex predicates as a tree and process one predicate at a time for the same block of tuples while traversing the tree.

Rid lists are better than bitmaps for evaluating very few tuples. However, we can also skip 256 or 512 consecutive tuples by checking the bitmap in SIMD. We merge the bitmaps using only `and`, `or`, and `and not` instructions, which come for free if the bitmaps are cache-resident. Thus, we avoid unions and intersections of rid lists. Notably, state-of-the-art sorted set intersection [Inoue *et al.*, 2014] reduces branch mispredictions from $O(n)$ to $O(\frac{n}{W})$ but requires $O(n)$ SIMD instructions to execute, same as the of $O(n)$ scalar instructions of the standard scalar implementation.

In conjunction nodes, the input bitmap represents the tuples that satisfy all previous predicates (siblings of the current node); the remaining tuples can be skipped. In disjunction nodes, the input bitmap represents the tuples that failed all previous predicates. In the example of Figure 6.4, we show the input bitmap denoting the tuples we need to evaluate per predicate where we do not skip any tuples. The expression tree does not have to be in CNF or DNF. We can support arbitrary levels of alternating conjunctions and disjunctions with either as root, which is not that uncommon. For example, TPC-H `Q19` has a 3-level expression and converting into either CNF or DNF would increase the number of leaf predicates. Generating the best tree is part of query optimization.

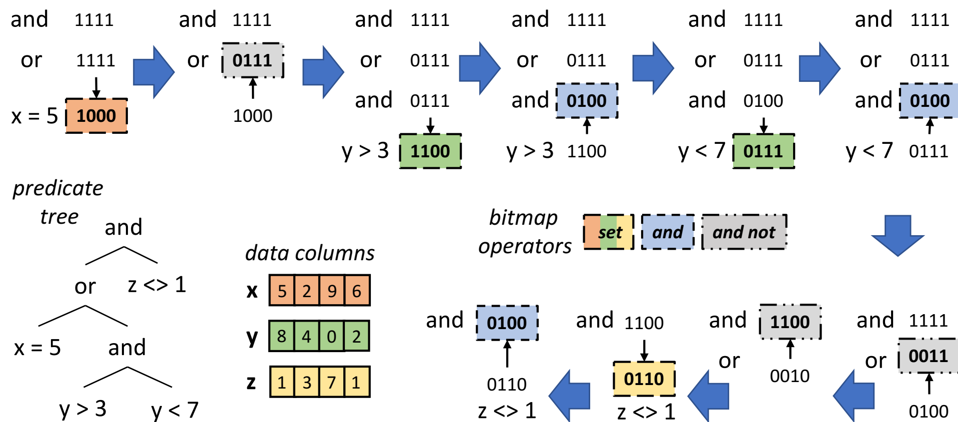



Figure 6.4: Example of selection scan (already determined tuples are not skipped)

The output of the selection scan is a sorted list of rids used to access column values from the selected table. In case of late materialization, we can simply append the rids to output without gathering any columns. Materialization strategies are part of query optimization and depend on many factors, such as selectivity, cardinality, columns, and data types [Abadi *et al.*, 2007] as well as hardware factors such as memory bandwidth. Our design supports both early and late materialization or a combination of both. The sub-operator to materialize a 32-bit column by dereferencing a list of rids is shown below.

```
void materialize_int32(const int32_t* rids, size_t tuples,
    const int32_t* data_in, int32_t* data_out) {
  size_t i = 0;  // store 1 tuple at a time until output is aligned
  while ((63 & (size_t) &out[i]) && i != tuples) { [...] }
  if (tuples - i >= 16) {
    size_t tuples_16 = ((tuples - i) & -16) + i;
    do {  // process 16 column values at a time
      __m512i rid = _mm512_loadu_si512(&rids[i]);  // load rids
      __m512i val = _mm512_i32gather_epi32(rid, data_in, 4);  // gather column values
      _mm512_stream_si512(&data_out[i], val);  // store to memory (non-temporal)
    } while ((i = i + 16) != tuples_16);
  }
  while (i != tuples) { [...] }  // store 1 tuple at a time to output
}
```

### 6.4.2 Compression

Some analytical databases prefer linear scans over secondary indexes due to compression [Raman *et al.*, 2013]. Our goal is to evaluate selection scans directly on compressed columns without decompressing. For complex predicate expressions, we still want to skip tuples that are determined by earlier predicates.

Dictionary encoding uses a sorted dictionary of distinct values per column and substitutes the column values with dictionary indexes. We need $\lceil \log n \rceil$ bits per index to represent $n$ distinct values and predicates are evaluated using the indexes alone. As described in Chapter 4, there are many ways to store the bits of the indexes. Here, we use *horizontal bit packing*, also used in commercial databases [Willhalm *et al.*, 2009]. In this scheme, we can skip tuples, although skipping cache lines is not guaranteed due to misalignment.

The sub-operator that unpacks the bits of indexes and evaluates the predicate is shown below. We need 5 SIMD instructions to unpack. The permutations, using masks `m_per_1`

and `m_per_2`, isolate the two 32-bit words that contain the packed bits. The lower word is shifted right to align the bits, using mask `m_srl`, and the upper word is shifted left to make space for the lower word, using `m_sll`. The words are merged and the high-order bits are reset via a ternary bitwise instruction. The predicate is evaluated in registers.

```
const int32_t decode_permute_1_mask[31][16] = { [...] }  // permutation masks for decoding (lower)
const int32_t decode_permute_2_mask[31][16] = { [...] }  // permutation masks for decoding (upper)
const int32_t decode_shift_1_mask[31][16] = { [...] }  // shift right masks for decoding
const int32_t decode_shift_2_mask[31][16] = { [...] }  // shift left masks for decoding
void select_encoded(const uint16_t* bitmap_in, uint16_t* bitmap_out,
    const void* data, size_t tuples, int32_t constant, int op, int bits) {
  const __m512i mask_constant = _mm512_set1_epi32(constant);  // mask of query constant
  const __m512i mask_max_code = _mm512_set1_epi32((1 << bits) - 1);  // to clear high order bits
  const __m512i mask_permute_1 = _mm512_load_epi32(decode_permute_masks[bits]);
  const __m512i mask_permute_2 = _mm512_load_epi32(decode_permute_masks[bits]);
  const __m512i mask_shift_1 = _mm512_load_epi32(shift_1_masks[bits]);
  const __m512i mask_shift_2 = _mm512_load_epi32(shift_2_masks[bits]);
  size_t i = 0, bits_x2 = bits + bits;
  switch (op) {  // separate code per comparison operator
    case '>':  // code for ">" operator
      do {  // scan over block
        __m256i bit = _mm256_load_si256(bitmap_in);  // check 256 bits of input bitmap
        if (!_mm256_testz_si256(b, b)) {  // skip 256 tuples if all 256 bits are 0
          __m512i bit_E = _mm512_cvtepu16_epi32(bit);
          uint64_t m = _mm512_test_epi32_mask(bit_E, bit_E);  // find groups of 16 tuples not all 0
          do {  // evaluate 16 tuples at a time that are not all 0
            size_t j = _tzcnt_u64(m);  // locate the 16 next tuples within 256 tuples
            __m512i key = _mm512_load_epi32(data + j * bits_x2);  // load 16 column values
            // permute to a pair of 32-bit words containing the packed bits per lane
            __m512i key_1 = _mm512_permutevar_epi32(mask_permute_1, key);
            __m512i key_2 = _mm512_permutevar_epi32(mask_permute_2, key);
            key_1 = _mm512_srlv_epi32(key_1, mask_shift_1);  // shift right to clear low-order bits
            key_2 = _mm512_sllv_epi32(key_2, mask_shift_2);  // shift left to clear high-order bits
            // merge (bitwise-or) the two words and clear (bitwise-and) the high-order bits
            key = _mm512_ternarylogic_epi32(key_1, key_2, mask_max_code, 168);
            bitmap_out[j] = _mm512_cmpgt_epi32_mask(key, mask_constant);  // predicate result
          } while (m = _blsr_u64(m));  // go to next 16 tuples that are not all 0
        }
        bitmap_in += 16, bitmap_out += 16, data += 256;  // progress the input pointers
      } while ((i = i + 256) != tuples);
      break;
    case '=': [...]  // code for "=", "<", "<>", ">=", and "<="
  }
}
```

Using *horizontal bit packing* not only allows us to skip parts of the input, but allows for efficient access of dictionary codes using rids. We only need to perform a 64-bit gather to load the bits of the index. The sub-operator that gathers the dictionary code of an encoded column is shown below.

```
void materialize_encoded(const int32_t* rids, size_t tuples,
    const void* data_in, int32_t* data_out, int bits) {
  const int64_t max_code = (1 << bits - 1);  // used to mask the high-order bits
  size_t i = 0;  // store 1 tuple at a time until output is aligned
  while ((63 & (size_t) &out[i]) && i != tuples) {
    size_t bit = rids[i] * bits;
    int64_t key = *(int64_t*) (data_in + (bit >> 3));  // load 64-bits containing the packed bits
    key >>= bit & 7;  // align to clear low-order bits
    key &= max_code;  // mask to clear high-order bits
    _mm_stream_si32(&out[i++], (int32_t) key);  // (non-temporal) store to output
  }
  if (tuples - i >= 16) {
    const __m512i mask_max_code = _mm512_set1_epi32(max_code);
    size_t tuples_16 = ((tuples - i) & -16) + i;
    do {  // process 16 column values at a time
      __m512i rid = _mm512_loadu_si512(&rids[i]);  // load rids
      __m512i bit_1 = _mm512_mul_epi32(rid, m_bit);  // compute bit offsets
      __m512i bit_2 = _mm512_mul_epi32(_mm512_srli_epi64(rid, 32), m_bit);
      // gather 64-bit words containing the packed bits per dictionary index
      __m512i key_1 = _mm512_i64gather_epi64(_mm512_srli_epi64(bit_1, 5), data_in, 4);
      __m512i key_2 = _mm512_i64gather_epi64(_mm512_srli_epi64(bit_2, 5), data_in, 4);
      // align 64-bit words by shift with bit offset modulo 32
      key_1 = _mm512_srlv_epi64(key_1, _mm512_and_si512(bit_1, mask_31));
      key_2 = _mm512_srlv_epi64(key_2, _mm512_and_si512(bit_2, mask_31));
      __m512i key = _mm512_permutex2var_epi32(key_lo, m_mix, key_hi);  // mix two halves
      key = _mm512_and_si512(key, mask_max_code);  // clear high-order bits
      _mm512_stream_si512(&out[i], key);  // (non-temporal) store decoded codes
    } while ((i = i + 16) != tuples_16);
  }
  while (i != tuples) { [...] }  // process 15 or less remaining tuples (same as scalar loop above)
}
```

Depending on the number of bits, we can unpack to the nearest CPU word size to avoid bit packing overheads. For example, 7-bit indexes can be materialized as 8-bit integers in memory. Saving bit space speeds up selection scans by saving bandwidth, but the overhead of packing and unpacking before and after every operator is expected to be higher.

The speed of decompression depends on the size of the dictionary. For columns with many distinct values, dereferencing large dictionaries incurs cache misses for every value and is in theory as expensive as a hash join. Thus, compression should be used when slow decompression is not required. For instance, attributes that are compressed with the same dictionary can be joined without any decompression. On the other hand, compressing numeric columns used in aggregate functions only adds decompression overhead.

Vertical bit packing, discussed in Chapter 4, can also be implemented, although the uncompressed should be kept as well to avoid expensive materialization via rids.

### 6.4.3 Hash Joins

Hash joins dominate the execution time of queries in data warehouses. As mentioned in Section 6.2.2, Volcano-based designs pipeline joins to avoid materializing intermediate results. The exception is column-at-a-time execution that executes each join separately. We showed that non-partitioned joins [Blanas *et al.*, 2011] perform poorly on many-core CPUs due to being memory latency bound. On the other hand, cache-conscious joins [Manegold *et al.*, 2002] eliminate random accesses out of the cache. If the workload is cache resident and we are compute-bound rather than memory-bound, SIMD code provides better speedups. Moreover, we can take advantage of the high-bandwidth on-chip MCDRAM that favors sequential accesses over random accesses.

The hash join operator is frequently optimized as a stand-alone operator that involves a key–rid pair [Balkesen *et al.*, 2013a; Blanas *et al.*, 2011]. While we can achieve very good performance in such a setting, the results are misleading, primarily because they ignore the cost of materialization of payload columns [Schuh *et al.*, 2016]. Furthermore, in situations where the join involves composite keys or the join key is not necessarily a 32-bit integer, many optimizations are either not applicable or are not nearly as efficient. In practice, we have to support not only multiple data types and composite keys, but also additional predicates. For example, the query `select * from R, S where R.x = S.x and R.y > S.y` should evaluate both predicates during the join. We do not discuss disjunctions in join predicates here since they are less common.

We outline how hash join operators work in our design. First, we hash the columns that appear in equality predicates. We execute the hash join using the hash values instead of the actual key columns. This allows us to map any data type into an integer including composite keys and optimize for the integer join. After we perform the hash join using the hash values, we generate rid lists for joined tuples pointing to the two inputs. We use the rids to evaluate the join predicates by accessing the actual columns, including non-equality predicates, and store back the rids of qualifiers. In the end, we can store the qualifying rids as the output of the join, or use them to materialize payload columns during the join, similarly to selection scans. The sub-operator for building a hash table using the hash values of the join key columns is shown below.

```
typedef struct {
  uint32_t hash;  // the 32-bit hash of the join key columns in the tuple
  int32_t rid;  // the location of the tuple (in the partitioned input)
} join_bucket_t;
```

The hash buckets are located by masking the hash values directly, we do not rehash. Note that since we map all key columns into integers by hashing, this sub-operator does not have to support multiple types.

```
void build_hashes(const uint32_t* hashes, size_t tuples,
    join_bucket_t* table, size_t buckets) {
  const __m512i mask_lanes = _mm512_set_epi32(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0);
  __m512i key, rid, loc;  // keys, rids, and hash bucket locations (state)
  __mmask16 k;  k = _mm512_kxnor(k, k);  // set mask to use all lanes initially
  size_t i = 0, j = 16;
  while (i + j <= tuples) {  // process 16 hashes at a time
    key = _mm512_mask_expandloadu_epi32(key, k, &hashes[i]);  // load new hashes from input
    __m512i rid_B = _mm512_broadcast_epi32(_mm_cvtsi64_si128(i));  // generate rids
    rid = _mm512_mask_expand_epi32(rid, k, _mm512_add_epi32(rid_B, mask_lanes));
    i += j;  // advance input pointer by number of reused lanes
    loc = [...];  // extract hash bucket locations from hash value without rehashing
    __m512i rid_T = _mm512_i32gather_epi32(off, &hash_table[0].rid, 8);  // gather rids
    k = _mm512_cmplt_epi32_mask(rid_T, m_0);  // find empty buckets
    __m512i con = _mm512_conflict_epi32(off);  // compare all lanes to detect conflicts
    k = _mm512_mask_testn_epi32_mask(k, con, con);  // find non-conflicting lanes
    j = _mm_popcnt_u64(k);  // count number of reused lanes
    __m512i key_lo = _mm512_permutex2var_epi32(key, m_pak_1, rid);  // pack keys and rids
    __m512i key_hi = _mm512_permutex2var_epi32(key, m_pak_2, rid);
    _mm512_mask_i32loscatter_epi64(table, k, off, key_lo, 8);  // scatter 64-bit pairs
    _mm512_mask_i32loscatter_epi64(table, k >> 8, _mm512_alignr_epi32(off, off, 8), hi, 8);
  }
  [...]  // process last 15 or less hashes in scalar code
}
```

As discussed in Chapter 5, when two SIMD lanes from the same scatter instruction write to the same memory location, *conflicts* occur. To avoid this hazard, we use special SIMD instructions that detect conflicts by performing all-to-all comparisons within the SIMD register, otherwise requiring $O(W^2)$ scalar or $O(W)$ regular SIMD instructions. In Section 5.4, we described two approaches, one for KNC co-processors using an additional round of gathers and scatters, and one for AVX-512 using specialized instructions. To reduce the cache accesses, we even pack hashes and rids into pairs and use 64-bit scatters. These ad-hoc optimizations are possible because we reduce the join key columns to hash values and execute the join using simple data types.

The sub-operator for probing the hash table of hash values is shown below. We cannot use unique-key hash table algorithms such as cuckoo hashing [Pagh and Rodler, 2004], even for foreign-key joins where unique inner keys are guaranteed, because distinct inner keys may still be mapped to the same hash value.

```
size_t probe_hashes(const uint32_t* hashes, size_t tuples,
    const join_bucket_t* hash_table, size_t buckets,  // hash table
    int32_t* inner_rids, int32_t* outer_rids) {  // outputs
  const __m512i mask_lanes = _mm512_set_epi32(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0);
  __mmask16 k;  k = _mm512_kxnor(k, k);  // set mask to use all lanes initially
  size_t i = 0, j = 16, o = 0;
  while (i + j <= tuples) {  // 16 tuples at a time
    key = _mm512_mask_expandloadu_epi32(key, k, &hashes[i]);  // load new keys from input
    __m512i rid_B = _mm512_broadcast_epi32(_mm_cvtsi64_si128(i));  // generate rids
    rid = _mm512_mask_expand_epi32(rid, k, _mm512_add_epi32(rids_B, mask_lanes));
    i += j;  // advance input pointer by number of reused lanes
    loc = [...];  // extract hash bucket locations from hashes
    __m512i buc_1 = _mm512_i32logather_epi64(loc, hash_table, 8);  // gather buckets
    __m512i buc_2 = _mm512_i32logather_epi64(_mm512_alignr_epi32(loc, loc, 8), hash_table, 8);
    __m512i key_H = _mm512_permutex2var_epi32(buc_1, m_pak_1, buc_2);  // unpack to hashes and gids
    __m512i rid_H = _mm512_permutex2var_epi32(buc_1, m_pak_2, buc_2);
    k = _mm512_cmpeq_epi32_mask(key, key_H);  // find matching hashes
    _mm512_mask_compressstoreu_epi32(&outer_rids[o], k, rid);  // (selective) store outer rids
    _mm512_mask_compressstoreu_epi32(&inner_rids[o], k, rid_H);  // (selective) store inner rids
    o += _mm_popcnt_u64(k);
    k = _mm512_cmplt_epi32_mask(rid_T, m_0);  // reuse lanes pointing to empty buckets
    j = _mm_popcnt_u64(k);  // count reused lanes
  }
  [...]  // process last 15 or less tuples using scalar code
  return o;  // joining tuples based (whose hashes of join keys match)
}
```

To make the final join operation cache-resident, we partition the inputs. Partitioning is discussed in Section 6.4.4. Each partition is processed by a different thread. Thus, the hash tables are private and no atomics are needed. If the inner relation is very small, we can use a shared hash table built with atomics in scalar code, but the cost of building is nevertheless irrelevant since the inner relation is small.

After the hash probing, we evaluate the predicates using rid lists. Hash collisions are rare because the hash join uses partitioning until the inner table fits in the cache. Also, the hash function we use during partitioning is different from the one used during the hash join. In a conjunction of join predicates, a sub-operator evaluates each predicate and filters the rid lists. The hash collisions are resolved by evaluating the equality predicates. The sub-operator to evaluate equality join predicates on 32-bit integer columns is shown below.

```
size_t join_filer_int32(int32_t* rids_1, int32_t* rids_2, size_t tuples,
    const int32_t* data_1, const int32_t* data_2, int op) {
  size_t i = 0;
  switch (op) {  // seperate code for each operator
    case '=':  // code for equality join predicate (R.x = S.y)
      if (tuples >= 16) {
        size_t tuples_16 = tuples & -16;
        do {  // process 16 tuples at a time
          __m512i ptr_1 = _mm512_load_epi32(&rids_1[i]);  // load inner rids
          __m512i ptr_2 = _mm512_load_epi32(&rids_2[i]);  // load outer rids
          __m512i val_1 = _mm512_i32gather_epi32(ptr_1, data_1, 4);
          __m512i val_2 = _mm512_i32gather_epi32(ptr_2, data_2, 4);
          __mmask16 k = _mm512_cmpeq_epi32_mask(val_1, val_2);
          _mm512_mask_compressstoreu_epi32(&rids_1[o], k, ptr_1);
          _mm512_mask_compressstoreu_epi32(&rids_2[o], k, ptr_2);
          o += _mm_popcnt_u64(k); }  // append inner & outer rids
        } while ((i = i + 16) != tuples_16);
      }
      while (i != tuples) { [...] }  // process last tuples
      break;
    case '>': [...]  // code for other operators
  }
  return o;  // number of tuples that satisfy the predicate
}
```

The inner keys are hashed and built into a hash table one block of tuples a time. The outer keys are also hashed, probed through the hash table, and the join predicates evaluated via rids one block of tuples at a time. Thus, we minimize the cache misses when evaluating the join predicates. Notably, neither the inner rids nor the outer rids are strictly sorted. The inner rids point to a cache-resident subset of the inner table. The outer rids are generated from vectorized hash probing out of order but are nevertheless clustered.

The hash function is chosen at runtime based on the key columns. If the join is on a single 32-bit key, we use an 1-to-1 32-bit hash function to avoid collisions. If the total width of multiple key columns is less than 4 bytes we avoid collisions by using FNV hashing [Fowler *et al.*, 2017]. If hash collisions cannot occur, we skip evaluating the equality predicates using the rids. For foreign key joins without hash collisions, we reuse the SIMD lanes when a match is found during hash probing.

After we evaluate the join predicates, we can materialize the rids directly or materialize a set of payload columns accessed via the rids. For instance, if the payload columns are used in the subsequent operator in the query plan, we can avoid storing and reloading the rids to access the payloads.

### 6.4.4 Partitioning

Partitioning is used in hash join and group-by aggregation to ensure cache-conscious execution. Many-core CPUs are favorable to cache-conscious execution, since the high-bandwidth on-chip MCDRAM can be used to accelerate partitioning. In our design, we store the partitioned output contiguously so we first compute a histogram. To compute the histogram, we load and hash the next block of tuples from the key columns, convert the hashes to partition ids based on the number of partitions, and increment the histogram counters. To avoid SIMD scatter conflicts, we replicate the histogram $W$ times for $W$ SIMD lanes (see Section 5.6.1). The sub-operator to update the histograms is shown below.

```
void histogram(const uint32_t* hashes, int32_t* histogram_x16,
    int bit_lo, int bit_hi, const uint8_t* pids) {  // bit range
  size_t i = 0;
  if (tuples >= 16) {
    const __m512i mask_1 = _mm512_set1_epi32(1); [...]  // set constant masks
    size_t tuples_16 = tuples & -16;
    do {  // process 16 tuples at a time in SIMD code
      __m512i pid = _mm512_load_epi32(&hashes[i]);  // load hashes
      pid = _mm512_srl_epi32(pid, mask_srl);  // isolate partition ids
      pid = _mm512_sll_epi32(pid, mask_sll);
      _mm_stream_si128(&pids[i], _mm512_cvtepi32_epi8(pid));  // store partition ids
      __m512i off = _mm512_add_epi32(pid, mask_rep);  // replicated histogram access
      __m512i cnt = _mm512_i32gather_epi32(off, histogram_x16, 4);
      cnt = _mm512_add_epi32(cnt, mask_1);  // increment histogram counts
      _mm512_i32scatter_epi32(histogram_x16, off, cnt, 4);  // scatter histogram counts
    } while ((i = i + 16) != tuples_16);
  }
  while (i != tuples) {  // process last tuples in scalar code
    uint32_t h = hashes[i];
    size_t p = [...];  // extract partition id from hash
    histogram_x16[p << 4] += 1;  // increment histogram
    pids[i++] = (uint8_t) p;
  }
}
```

To find the partition boundaries, we compute the prefix sum of the histograms. Then, we invoke the sub-operator shown below to compute the output offset of each tuple and store it in the cache. The output offsets will be reused by the sub-operators that shuffle each payload column and avoid computing the output offsets multiple times. Alongside the output offset, we store an offset that serializes scatter conflicts. Conflict serialization was discussed in Section 5.6.3 alongside an implementation for serializing conflicts in AVX-512.

```
void shuffle_core(const uint8_t* pids, size_t tuples,
    int32_t* partition_offset,  // output offset per partition (prefix sum of histograms)
    int8_t* conflict_offset,  // conflict serialization offset per tuple
    int32_t* output_offset) {  // output offset per tuple
  size_t i = 0;
  if (tuples >= 16) {
    const __m512i mask_1 = _mm512_set1_epi32(1); [...]  // set constant masks
    size_t tuples_16 = tuples & -16;
    do {
      __m512i pid = _mm512_cvtepu8_epi32(_mm_load_si128(&pids[i]));  // load partition ids
      __m512i off = _mm512_i32gather_epi32(pid, partition_offset, 4);  // gather partition offsets
      __m512i ser_1 = _mm512_conflict_epi32(pid);  // create bitmasks of conflicts per 32-bit lane
      __m512i ser_2 = _mm512_srli_epi32(x, 5);  // shift bits 6 to 10 to the low-order bits
      __m512i ser_3 = _mm512_srli_epi32(x, 10);  // shift bits 11 to 15 to the low-order bits
      // count set bits in ranges using 32-way permutation using 5 low-order bits of index register
      ser_1 = _mm512_permutex2var_epi32(mask_bitcount_L, ser_1, mask_bitcount_H);  // bits [1,5]
      ser_2 = _mm512_permutex2var_epi32(mask_bitcount_L, ser_2, mask_bitcount_H);  // bits [6,10]
      ser_3 = _mm512_permutex2var_epi32(mask_bitcount_L, ser_3, mask_bitcount_H);  // bits [11,15]
      __m512i ser = _mm512_add_epi32(_mm512_add_epi32(ser_1, ser_2), ser_3);
      _mm_store_si128(&conflict_offset[i], _mm512_cvtepi32_epi8(ser));
      off = _mm512_add_epi32(off, ser);  // update output offset per tuple
      _mm512_store_epi32(&output_offset[i], off);  // store output offset per tuple for next block
      off = _mm512_add_epi32(off, mask_1);  // increment partition offsets
      _mm512_i32scatter_epi32(partition_offset, pid, off, 4);  // scatter offsets
    } while ((i = i + 16) != tuples_16);
  }
  while (i != tuples) {  // process last tuples in scalar code
    size_t p = pids[i];  // load partition id
    size_t o = partition_offset[p]++;  // update output offset of partition
    output_offset[i++] = o;  // store output offset of tuple
  }
}
```

After computing the offsets, we invoke sub-operators to shuffle one column at a time for the next block of tuples. To avoid re-accessing the array of offsets per partition for each column we shuffle, we store the output offset per tuple of the next block in the cache and reload it for each column that we shuffle. To mitigate the cost of cache and TLB misses, we store the data in a cache-resident buffer and flush to output in batches using non-temporal stores [Satish *et al.*, 2010; Wassenberg and Sanders, 2011].

In our implementation, the size of the buffer for each partition is equal to the size of two cache lines. Once the lower half of the buffer is full, we flush the cache line to output using a non-temporal SIMD store and shift the data from the upper half. Thus, we avoid buffer overflows and can also process 8-bit or 16-bit columns using 32-bit SIMD scatters, since 8-bit or 16-bit scatters are not supported in AVX-512. This process is shown below.

```
void shuffle_int32(const uint8_t* pids, size_t tuples,
    const int8_t* conflict_offsets, const int32_t* output_offsets,
    const int32_t* data_in, int32_t* data_buf, int32_t* data_out) {
  size_t i = 0;
  if (tuples >= 16) {
    const __m512i mask_15 = _mm512_set1_epi32(15); [...]  // set constant masks
    size_t tuples_16 = tuples & -16;
    do {
      __m512i off = _mm512_load_epi32(&output_offsets[i]);
      __m512i ser = _mm512_cvtepu8_epi32(_mm_load_si128(&conflict_offsets[i]));
      off = _mm512_sub_epi32(off, ser);  // remove serialization offset
      off = _mm512_and_epi32(off, mask_15);  // find location in buffer
      off = _mm512_add_epi32(off, ser);  // add serialization offset
      __mmask16 k = _mm512_cmpeq_epi32_mask(off, mask_15);  // find flushing partitions
      __m512i pid = _mm512_cvtepu8_epi32(_mm_load_si128(&pids[i]));  // load partition ids
      off = _mm512_or_epi32(off, _mm512_slli_epi32(pid, 5));  // compute offset in buffers
      __m512i val = _mm512_stream_load_si512(&data_in[i]);  // load data
      _mm512_i32scatter_epi32(data_buf, off, val, 4);  // scatter to buffer
      if (_mm512_kortestz(k, k)) continue;  // skip if nothing to flush
      uint64_t m = k;  // bitmask of lanes pointing to full buffers
      do {  // flush one full buffer at a time to memory
        size_t j = i + _tzcnt_u64(m);  // location of tuple in input
        size_t o = output_offsets[j];  // location of tuple in output
        size_t p = pids[j];  // the partition we need to flush
        int32_t* b = data_buf[p << 5];  // buffer of partition
        // flush lower half to output and shift upper half of buffer
        __m512i buf_1 = _mm512_load_si512(b);  // load buffer
        __m512i buf_2 = _mm512_load_si512(&b[16]);
        // store lower half of buffer to output using non-temporal stores
        _mm512_stream_si512(&data_out[o - 15], buf_1);
        _mm512_store_si512(b, buf_2);  // shift upper half in buffer
      } while (m = _blsr_u64(m));  // get next set bit of bitmask
    } while ((i = i + 16) != tuples_16);
  }
  while (i != tuples) { [...] }  // process remaining tuples
}
```

Depending on the data type of the column we are shuffling, a different number of values is needed to fill the cache-resident buffer of each partition. Since the sub-operators are specialized per data-type, this logic is hardcoded using constants, for instance, how many items we need before we flush a buffer, or the right memory offsets for the scatters. In some cases, we optimize the sub-operator even within the same data type. For instance, long strings are shuffled (e.g., `char(100)`) one tuple per iteration similar to scalar code (horizontal vectorization of Section 5.4), but use contiguous SIMD loads and stores to move 64 bytes at a time. On the other hand, short strings are shuffled similar to integers by processing $\mathcal{W}$ tuples per iteration (vertical vectorization of Section 5.4).

During a partitioned hash join, each thread partitions a portion of the input. If we need to split into many partitions that exceed the fanout that guarantees cache-conscious execution, we split into multiple passes like LSB radixsort [Satish *et al.*, 2010]. Once we have more partitions than threads, we shuffle the data across threads, similar to a NUMA shuffling step [?]. Then, we can continue partitioning until the partitions of the inner table fits in the cache. In the final phase, each thread executes the final hash join for its local partitions. To find the boundaries of partition $p$ out of $2^k$ partitions, we do not store the histograms explicitly. We perform a binary search over the join key columns and search for the earliest occurrence of hash $p$ and $p{+}1$, after masking with $2^k - 1$.

### 6.4.5 Materialization

Column-at-a-time execution introduced late materialization via rid dereferencing, but this step can be more expensive than the actual hash join [Schuh *et al.*, 2016]. In radix-decluster [Manegold *et al.*, 2004], we execute the join using keys and rids only, and then decluster the inner payloads to the outer relation order using rids [Manegold *et al.*, 2004].

Interestingly, code-generating operator-pipelining designs avoid rid dereferencing altogether. When multiple hash joins are pipelined, the attributes of the inner tables are early materialized in the hash table, while the attributes of the outer table are loaded on demand during the linear scan. Assume the example code in Section 6.3.1 with two hash joins as nested loops. We access `f.key_A` column only for `F` tuples that satisfy the selection. Then, we access the `f.key_B` column only for `F` tuples that satisfy the outer join. The outer side achieves late materialization without rid dereferencing.

We support both early and late materialization since different strategies are better per case [Abadi *et al.*, 2007]. Our goal is to always guarantee both data parallelism and thread parallelism. To ensure thread parallelism, instead of using radix-decluster, we generate intermediate rids and sort to reshuffle the inner payloads based on the order of the outer table. In radix-decluster, we lose thread parallelism, as splitting each window (the cache size) across all threads is impractical.

To decluster the payloads of the inner side, we avoid sorting the rids to the outer relation directly, as we would then need to merge rid lists across multiple joins. Instead, the output

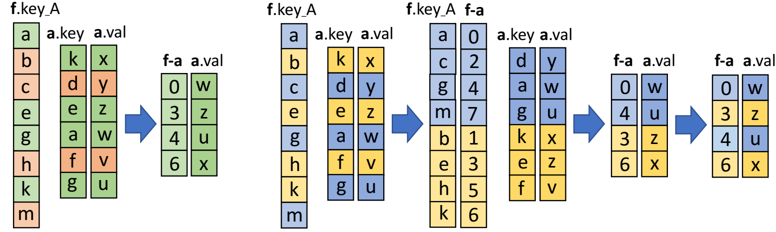



Figure 6.5: Hash join F⋈A from the example query with and without partitioning

of each hash join generates pointers to its inputs in range $[0, n)$, where $n$ is the number of tuples in the outer relation. Then, we use $\lceil \log n \rceil$ partitioning passes to decluster the inner payloads back to the original order of the outer side. The outer payloads are gathered using the generated pointers in $[0, n)$ and no merging of rid lists is required.

We use the query of Section 6.3 as an example for late materialization. The join order is (`F` ⋈ `A`) ⋈ `B`. Initially, we execute the selections on `F` and `A`. From the selection on `F`, we materialize `f.key_A` and `f`, an rid list to `F` tuples that satisfy the predicates. We will use `f` to materialize `f.val`. From the selection on `A`, we materialize ⟨`a.key, a.val`⟩ without generating any rids. To join `f` with `a`, we determine the number of partitioning passes to ensure cache-conscious execution. We partition ⟨`f.key_A`⟩ on the hash of ⟨`f.key_A`⟩ and ⟨`a.key, a.val`⟩ on the hash of ⟨`a.key`⟩. While partitioning ⟨`f.key_A`⟩, we generate `f-a`, an array of pointers to the input (the outputs of the previous join). The `f-a` rids are pointers to `f` that satisfy `f.key_A = a.key`. For multiple partitioning passes, we treat `f-a` as a payload column. We join the two sides, materialize ⟨`f-a, a.val`⟩, and sort the output on ⟨`f-a`⟩ using LSB radix-sort. Figure 6.5 shows the hash join between `F` and `A` with and without partitioning. We have the rids of `F` tuples that satisfy the selective predicates in `f`, and the pointers to `f` that satisfy the join predicate in `f-a`. To get the `f` rids that satisfy both the selection on `F` and the join of `F` with `A`, we dereference `f-a`→`f` and replace `f`. For the hash join on `f.key_B = b.key`, we invoke a selection scan on `B` and materialize ⟨`b.key, b.val`⟩. The `f.key_B` column is materialized via the updated `f`. We join the two sides, with or without partitioning, and generate `f-a-b`, which points to tuples that qualify from the previous join with `A`. We sort ⟨`b.val`⟩ on `f-a-b` to match the `f` order. For the final aggregate, we materialize `f.val` using `f` and `a.val` using `f-a-b`, but first we update `f` by dereferencing `f-a-b` → `f`. The `b.val` column is already in the right order.

### 6.4.6 Group-by Aggregation

In our design, group-by aggregation is split in multiple steps, (i) compute the number of groups, (ii) partition the input to ensure cache-conscious execution, (iii) determine the group id (gid) per tuple, (iv) compute intermediate expressions, and (v) use the gids to update the partial aggregates. In contrast to hash joins where we can use cardinality of the smallest input to determine the number of partitions for cache-conscious execution, in group-by aggregation with a group-by clause, we estimate the cardinality of the output.

To estimate the number of groups, we can use the PCSA algorithm [Flajolet and Martin, 1985], built from data-parallel sub-operators and written entirely in SIMD. We scan over the group-by key attributes, compute a hash function for the next block of tuples, and update a cache-resident array of bitmasks by `or`-ing the least significant set bit of the hash value. The lowest set bit of the hash value is computed using `x & -x`. We keep an array of bitmasks and pick which bitmask to update using the high-order bits of the hash value, similarly to updating a histogram. To avoid SIMD scatter conflicts, we replicate the array of bitmasks. At the end, we merge the bitmasks and combine the estimates across threads. This step can be considered a separate operator, similar to partitioning and NUMA shuffling.

After estimating the number of groups, we determine the number of partitioning passes to ensure cache-resident execution. If there are no partitions, each thread processes the local portion of tuples and we merge the partial aggregates at the end. Otherwise, we create more partitions than threads, shuffling the data across threads, and each thread processes its local partitions in a shared-nothing fashion.

After partitioning, the first step of the main group-by aggregation operator is to map of the hash values of the group-by keys to gids. We use a hash table to store pairs of hash values and gids. Each group is currently identified by a unique hash value. We compute the hash value of the group-by columns for each input tuple, search two possible hash buckets

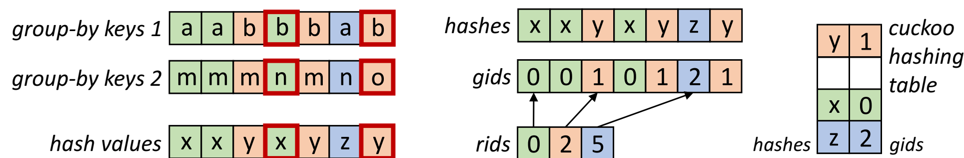



Figure 6.6: Example of mapping hashes to group-ids

in the cuckoo table for a hash value match. If no match is found, we create a new group, assign the next gid, and store the rid of the tuple. Figure 6.6 shows an example.

```
typedef struct {
  uint32_t hash;  // hash value of group-by keys
  int32_t gid;  // implicitly generated group-id
} aggr_bucket_t;
```

In contrast to hash joins where cuckoo tables are not used due to hash collisions, cuckoo tables are useful here since they store the mapped gids in the same order as the input tuples, since other vectorized hashing schemes produce the output out-of-order. The SIMD code here is complicated due to many edge cases. We present a brief outline here. On each iteration, we load new hashes from the input and process them alongside old hash–gid pairs in the SIMD registers that were displaced as part of cuckoo hashing insertion. The displaced pairs probe the alternative hash bucket while the new hashes may probe both. For unique hashes not found in the table, new gids are generated and their rids are stored.

```
size_t hashes_to_gids(const uint32_t* hashes, int32_t* gids, size_t tuples,
    size_t groups, int32_t* rids, aggr_bucket_t* hash_table, [...]) {
  size_t i = 0, j = 16, g = 0, l = [...];  // upper bound in iterations
  __m512i key, gid, loc;  // state (keys aka hash values, gids, and hash bucket locations)
  __mmask16 k;  k = _mm512_kxnor(k, k);  // use all lanes initially
  while ((i = i + j) <= tuples) {  // process 16 tuples at a time
    if (--l == 0) { [...] }  // resize and rebuild cuckoo table
    key = _mm512_mask_loadu_epi32(key, k, &hashes[i - 16]);  // load new hashes in rightmost lanes
    loc = _mm512_ternarylogic_epi32(loc, loc_1, loc_2, 150);  // compute alternative hash
    loc = _mm512_mask_mov_epi32(loc, k, loc_1);  // use 1st hash for new tuples
    __m512i con = _mm512_conflict_epi32(key);  // find unique hashes via conflict detection
    __mmask16 k1 = _mm512_testn_epi32_mask(con, con);  // find (leftmost) unique hashes
    __m512i loc_1 = [...];  // map hash value to 1st hash bucket without rehashing
    // gather buckets from hash table for hashes that are unique
    __m512i buc_1;  buc_1 = _mm512_mask_i32gather_epi64(buc_1, k1, hash_table, loc, 8);
    __m512i col = _mm512_alignr_epi32(loc, loc, 8);
    __m512i buc_2;  buc_2 = _mm512_mask_i32gather_epi64(buc_2, k1 >> 8, hash_table, col, 8);
    __m512i key_H = _mm512_permutex2var_epi32(buc_1, mask_unpack_1, buc_2);  // unpack hashes
    __m512i gid_H = _mm512_permutex2var_epi32(buc_1, mask_unpack_2, buc_2);  // unpack gids
    // find for which lanes we need to gather the 2nd hash bucket
    __mmask16 k2 = _mm512_mask_cmpge_epi32_mask(k1, gid_H, mask_0);  // if 1st bucket is empty
    k2 = _mm512_mask_cmpneq_epi32_mask(k2, key, key_H);  // or if hash is found in 1st bucket
    k2 = _mm512_kand(k2, k1);  // exclude displaced tuples that gather once
    __m512i loc_2 = [...];  // map hash value to 2nd hash bucket without rehashing
    loc = _mm512_mask_mov_epi32(loc, k2, loc_2);  // replace with 2nd hash bucket
    buc_1 = _mm512_mask_i32gather_epi64(buc_1, k2, loc, hash_table, 8); // (re-)gather buckets
    col = _mm512_alignr_epi32(loc, loc, 8);
    buc_2 = _mm512_mask_i32gather_epi64(buc_2, k2 >> 8, col, hash_table, 8);
```

```
    key_H = _mm512_permutex2var_epi32(buc_1, mask_unpack_1, buc_2);  // (re-)unpack hashes
    gid_H = _mm512_permutex2var_epi32(buc_1, mask_unpack_2, buc_2);  // (re-)unpack gids
    // lanes without unique hash values must copy gid from (leftmost) lanes with unique hashes
    con = _mm512_and_epi32(con, _mm512_sub_epi32(mask_0, con));  // isolate leftmost conflict
    // we implicitly implement a vectorized trailing zero count via leading zero count here
    con = _mm512_sub_epi32(mask_31, _mm512_lzcnt_epi32(con));  // point to leftmost conflict
    // hashes not unique in the SIMD register copy the gid of the leftmost lane
    con = _mm512_mask_blend_epi32(k1, con, mask_lanes);  // point to self if no conflict
    // find which lanes must generate new groups altogether
    __mmask16 k3 = _mm512_mask_cmpge_epi32_mask(k_U, gid_H, mask_0);  // find non-empty buckets
    k2 = _mm512_mask_cmpneq_epi32_mask(k3, key, key_H);  // find non-matching hashes
    k2 = _mm512_kor(k2, _mm512_kandn(k3, k1));  // bucket empty or different hash
    if (!_mm512_kortestz(k2, k2)) {  // all (unique) hashes were found in the hash table
      // store gids from leftmost lane with same hash if not unique and store to output
      _mm512_storeu_epi32(&gids[i - 16], _mm512_permutevar_epi32(con, gid_H));
      j = 16, k = _mm512_kxnor(k, k);  // reused all lanes in the next loop
    } else {
      k1 = _mm512_kand(k, k2);  __m512i gid_O;
      if (_mm512_kortestz(k1, k1)) {  // no new group ids
        gid_O = _mm512_mask_blend_epi32(k, gid, gid_T);
      } else {  // generate new group ids and store rids to output
        __m512i gid_B = _mm512_broadcast_epi32(_mm_cvtsi64_si128(g));  // broadcast groups
        __m512i rid_B = _mm512_broadcast_epi32(_mm_cvtsi64_si128(i));  // broadcast index
        // generate new gids for unique hashes not found in the table and store to output
        gid = _mm512_mask_expand_epi32(gid, k1, _mm512_add_epi32(gid_B, mask_lanes));
        _mm512_mask_compressstoreu_epi32(&rids[g], k1, _mm512_add_epi32(rid_B, mask_lanes));
        g += _mm_popcnt_u64(k1);  // update number of groups (index to array of rids)
        gid_O = _mm512_mask_blend_epi32(_mm512_kandn(k1, k), gid, gid_T);
      }
      // copy gid from leftmost lanes with same hash and store to output
      _mm512_mask_storeu_epi32(&gids[i - 16], k, _mm512_permutevar_epi32(con, gid_O));
      con = _mm512_conflict_epi32(_mm512_mask_blend_epi32(k_I, mask_minus_1, loc));
      k1 = _mm512_mask_testn_epi32_mask(k2, con, con);  // exclude scatter conflicts
      buc_1 = _mm512_permutex2var_epi32(key, mask_pack_1, gid);  // pack hashes and gids
      buc_2 = _mm512_permutex2var_epi32(key, mask_pack_2, gid);
      // scatter to hash table pairs of (unique) hashes and gids (new or displaced)
      _mm512_mask_i32loscatter_epi64(hash_table, k1, loc, buc_1, 8);
      _mm512_mask_i32loscatter_epi64(hash_table, k1 >> 8, col, buc_2, 8);
      key = _mm512_mask_mov_epi32(key, k1, key_H);  // retain conflicting hashes for next loop
      gid = _mm512_mask_mov_epi32(gid, k1, gid_H);  // retain conflicting gids for next loop
      k = _mm512_mask_cmpge_epi32_mask(k1, gid, mask_0);  // lanes with displaced pairs
      k = _mm512_kor(k, _mm512_kandn(k1, k2));  // include conflicting lanes
      key = _mm512_mask_compress_epi32(key, k, key);  // pack hashes to the leftmost lanes
      gid = _mm512_mask_compress_epi32(gid, k, gid);  // pack (assigned) gids
      loc = _mm512_mask_compress_epi32(loc, k, loc);  // pack hash bucket locations
      j = 16 - _mm_popcnt_u64(k);  // count the (rightmost) lanes to be reused
      k = 0xFFFF0000 >> (int8_t) j;  // set a bitmask for the rightmost lanes to be reused
    }
  }
  [...]  // process last 15 or less tuples in scalar code (also avoiding infinite loops)
  return g;  // number of groups found based on unique hash values
}
```

We scatter both new and displaced pairs back to the cuckoo table, in turn displacing new tuples in case of collision. The displaced hash–gid pairs are kept in the leftmost SIMD lanes to detect matching hashes from new input tuples using the SIMD conflict detection instructions. We show the sub-operator that maps hashes to gids below, skipping the part where we insert new groups into the table.

Unlike hash joins where the number of inner tuples are known, here we do not know the exact number of groups. Thus, the load factor may increase enough for the hash table to fail during building. Also, cuckoo hashing may inherently fail due to hash function collisions [Pagh and Rodler, 2004]. To cover both these cases, we bound the number of iterations. If we reach the iteration threshold, we resize and rebuild the cuckoo table.

On average, we expect to have much fewer groups than tuples, so in most cases we will have already inserted all the hash–gid pairs in the hash table and will simply probe the hash table and store the matching gids sequentially in the output. Being able to map the hashes to gids in-order is the main reason that we use cuckoo hashing here. On the other hand, because inserting tuples in cuckoo hashing tables is not particularly simple, we need excessively complicated code to cover cases where new groups are being inserted into the table. New gids are generated as soon as the new hash values are encountered, even if we cannot insert all the new pairs in the hash table during the current iteration.

Mapping unique hashes to gids does not guarantee correctness due to hash collisions. In hash joins, we re-evaluate the predicates, including the equalities. Here, we use the rid of the first tuple per group to dereference the group-by columns and verify that the group-by columns of tuples in the same group indeed match. If not, we create new groups on the fly. Figure 6.7 fixes the hash collisions using the same example as Figure 6.6.

To fix hash collisions, we scan each group-by column exactly once. We scan the column sequentially and compare the latest column value with the column value of the first tuple

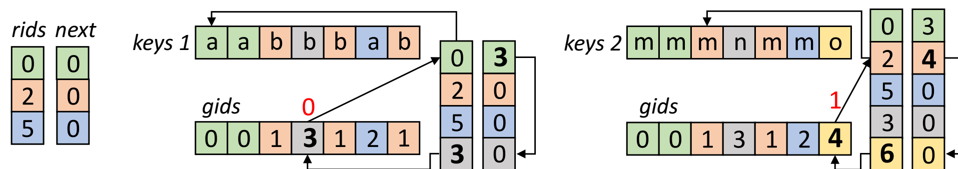



Figure 6.7: Example of resolving hash collisions

of the group. If the columns do not match, we traverse a list of gid–rid pairs belonging to distinct groups with the same hash and compare against other column values dereferenced via rid. If we find a match, we only update the gid of the latest tuple. Otherwise, we generate a new gid–rid pair and add it in the list of groups with the same hash.

Since we process one column at a time, we can use type-specific sub-operators. The sub-operator used to validate a 32-bit group-by column is shown below. We load the group-by column and the gids sequentially. We gather the rids of the first tuple per group using the gids, and then gather the earlier group-by column value of the first tuple per group. Because hash collisions are rare, given that we partition the input to ensure cache-conscious execution, we switch to scalar code to fix the collisions.

```
size_t group_by_fix_int32(const int32_t* data, int32_t* gids, size_t tuples,
    int32_t* rids, int32_t* next, size_t groups) {
  size_t i = 0;
  if (tuples >= 16) {
    size_t tuples_16 = tuples & -16;
    do {
      __m512i gid = _mm512_load_epi32(&gids[i]);  // load gids
      __m512i rid = _mm512_i32gather_epi32(gid, rids, 4);  // gather rid of 1st tuple per group
      __m512i key_1 = _mm512_i32gather_epi32(rid, data, 4);  // gather column of 1st tuple per group
      __m512i key_2 = _mm512_load_epi32(&data[i]);  // load column from new tuples
      uint64_t m = _mm512_cmpeq_epi32_mask(key_1, key_2);  // compare column value
      if (_mm512_kortestz(m, m)) continue;  // no collisions to fix
      do {
        size_t j = i + (size_t) _tzcnt_u64(m);  // input offset
        size_t g = gids[j];  // load gid from input
        int8_t k = keys[j];  // (re-)load key from input in scalar register
        do {  // iterate over list of groups with same hash of group-by keys
          size_t ng = next[g];
          if (ng == 0) {  // if new group must be added
            next[g] = groups;  // connect old group with current (new) group
            g = groups++;  // get next gid by incrementing number of groups
            next[g] = 0;  // set current group as the end of list
            rids[g] = (int32_t) j;  // store rid of new group
            break;
          }
          g = ng;  // go to next gid in list
        } while (k != keys[(size_t) rids[g]]);  // or stop if group is found
        gids[j] = g;  // fix group id
      } while (m = _blsr_u64(m));  // next tuple to fix
    } while ((i = i + 16) != tuples_16);
  }
  while (i != tuples) { [...] }  // process last tuples in scalar code
  return groups;  // return (updated) number of groups
}
```

As discussed in Section 6.4.3, if we rule out hash collisions based on the width of the group-by attributes, this step is bypassed. If we start with one group for all tuples (all gids are 0) and scan each column to fix collisions, we would compute the correct gids without using hashes. However, we would need $O(ng)$ time for $n$ tuples and $g$ groups, not $O(n)$.

```
void update_min_float(const float* data, const int32_t* gids,
    size_t tuples, float* min_aggr_x16, size_t groups) {
  if (tuples >= 16) {
    const __m512i mask_lanes = _mm512_set_epi32(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0);
    size_t tuples_16 = tuples & -16;
    do {  // process 16 tuples at a time
      __m512 val = _mm512_load_ps(&data[i]);  // load source data
      __m512i gid = _mm512_loadu_si512(&gids[i]);  // load gids
      // compute offset of replicated partial aggregate from gids
      gid = _mm512_or_epi32(_mm512_slli_epi32(gid, 4), m_inc);
      // load the min and write back current value if smaller
      __m512 min = _mm512_i32gather_ps(gid, min_aggr_x16, 4);
      __mmask16 k = _mm512_cmp_ps_mask(val, min, _CMP_LT_OQ);
      _mm512_mask_i32scatter_ps(min_aggr_x16, k, gid, val, 4);
    } while ((i = i + 16) != tuples_16);
  }
  while (i != tuples) { [...] }  // process last tuples in scalar code
}
```

We use the gids as direct mapping pointers to compute the aggregate functions. If we need to compute functions across multiple columns, we compute them block at a time. For example, to compute `sum(x*y)`, we compute `x*y` for the next block of tuples, store the intermediate result in the cache and then use it to update the partial sum aggregates. To avoid conflicts, we replicate the array of sums, since the sum is commutative. Sum aggregates are updated similar to histograms for partitioning. We show the sub-operator that computes the `min()` aggregate for a `float` column above. If we have only one group, we update the (replicated) partial aggregates directly in registers, as shown below.

```
void update_min_float_single(const float* data, [...]) {
  if (tuples >= 16) {
    __m512 min = _mm512_load_ps(min_aggr_x16);  // load aggregate
    size_t tuples_16 = tuples & -16;
    do {  // process 16 tuples at a time
      min = _mm512_min_ps(min, _mm512_load_ps(&data[i]));
    } while ((i = i + 16) != tuples_16);
    _mm512_store_ps(min_aggr_x16, min);  // store back aggregate
  }
  while (i != tuples) { [...] }  // process last tuples in scalar code
}
```

Each thread processes a separate partition. The hash table of hashes to gids, the rids per group and the partial aggregates remain in the cache until we process all the tuples of the partition. Since we know the number of groups, we allocate output slots by atomically incrementing a shared counter. If the input is not partitioned, each thread processes one block at a time and all the partial aggregates are kept in the cache. We use a special-purpose shared hash table to link the partial aggregates of the same group across all threads. Finally, the threads distribute, merge, and store the partial aggregates.

Arithmetic expressions used in aggregate functions is done in cache-resident buffers. Sub-operators process each arithmetic operation on two columns or using a constant. Decimals are harder because we need ensure no precision is lost by detecting arithmetic overflows and also ensure rounding is done on decimal digits. In the VIP engine, we optimize decimal operations using both integer and floating point operators. For additions and subtractions, we use integer operations and compute overflows using bitwise operations. The sub-operator to add 64-bit decimals and check for overflows is shown below.

```
bool add_decimal(const int64_t* in_1, const int64_t* in_2, int64_t* out, size_t tuples) {
  size_t i = 0;
  bool overflow = false;
  if (tuples > 7) {
    const __m512i m_0 = _mm512_setzero_si512();
    __mmask8 k = 255;  // overflow mask
    do {  // process 8 tuples at a time (can be unrolled further)
      __m512i x = _mm512_load_epi64(&in_1[i]);  // load inputs
      __m512i y = _mm512_load_epi64(&in_2[i]);
      __m512i z = _mm512_add_epi64(x, y);  // compute z = x + y
      x = _mm512_ternarylogic_epi64(x, y, z, 66);  // compute bitwise (x & z) ^ (y & z)
      k = _mm512_mask_cmpge_epi64_mask(k, x, m_0);  // detect overflow by checking sign bit
      _mm512_store_epi64(&out[i], z);  // store back result
    } while ((i = i + 8) != tuples_8);
    if (k != 255) overflow = true;  // overflow was detected
  }
  while (i != tuples) { [...] }
  return overflow;
}
```

We support integer, floating point, and numeric types. Numeric types are stored as integers, e.g., `numeric(12, 2)` 0.05 is stored as 5, and we use 64-bit floating point SIMD instructions for multiplications and divisions. To compute `x * y`, we convert from integer to double, evaluate `x * y * 0.01`, check for overflow or precision loss, and round back to integer. In case of overflow or potential precision loss, we branch to scalar code.

## 6.5 Experimental Evaluation

We perform all experiments in an Intel Xeon Phi 7210 CPU, based on the latest generation of many-core platforms codenamed Knights Landing. The CPU has 64 cores running at 1.3 GHz with 4-way simultaneous multi-threading (SMT) for a total of 256 hardware threads. The CPU also includes 16 GB of high-bandwidth on-chip MCDRAM memory and has access to 192 GB DDR4 DRAM at 2133 MHz. We measured the load bandwidth of MCDRAM at ∼295 GB/s, the (non-temporal) store bandwidth ∼220 GB/s and the copy bandwidth ∼170 GB/s. For DRAM, the same are 70 GB/s, 41 GB/s and 34 GB/s respectively. We compile using ICC 18 with `-O3` optimization. The OS is Linux 3.10.

In all experiments, we compare operators based on our design against query-specific scalar code. The baseline code is hand-written (see Section 6.3.1), and is equivalent to the output of code-generating engines. Register-resident execution [Neumann, 2011] is achieved via `-O3` optimization after we eliminate all function calls and inline the hash functions. Finally, we ignore the compilation times, favoring the baseline, since the VIP engine does not compile new code per query at runtime.

In Figure 6.8, we show the selection scan throughput using synthetic uniform data. We use 1–3 integer columns (32-bit) with one predicate per column. The selectivity per predicate is $\sqrt{s}$ for 2 columns and $\sqrt[3]{s}$ for 3 columns ($s$ the overall selectivity). The query-specific code (baseline) evaluates one tuple at a time using a `x && y && z` expression, thus skipping columns when a predicate is false. In the VIP design, during selection scans,

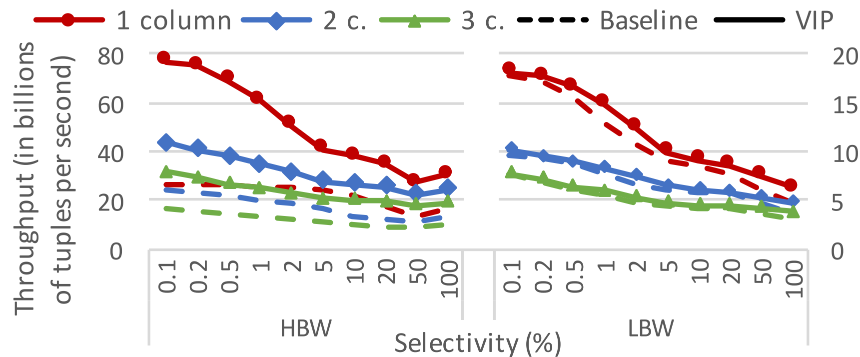



Figure 6.8: Selection scans on uncompressed data (1–3 32-bit key columns, 1 predicate per key column, 1 32-bit payload column)

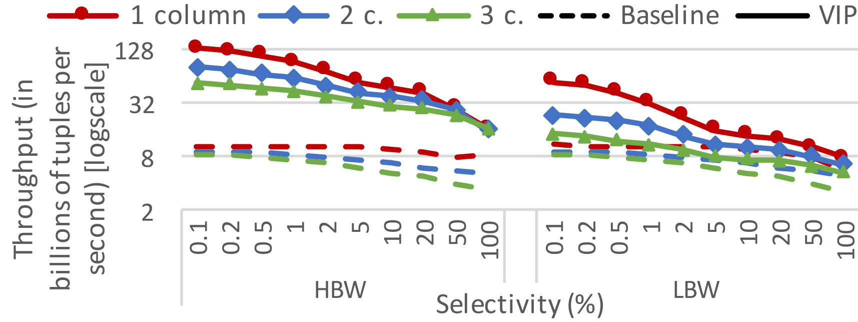



Figure 6.9: Selection scans on compressed data (1–3 32-bit key columns, 1 predicate per key column, 1 32-bit payload column)

columns are evaluated separately and bitmaps are merged in the cache. When the dataset is loaded from fast memory (HBW), VIP is 1.7–2.9X faster and saturates the bandwidth. The per-tuple code is compute-bound, even though it evaluates fewer predicates in total. When the dataset is loaded from slow memory (LBW), the throughput is reduced by 4X and both the baseline and VIP saturate the memory bandwidth.

In Figure 6.9, we repeat the experiment on dictionary compressed data using 10, 15, and 20 bits for the 3 columns. On fast memory, VIP is 12X faster than the baseline due to fast decompression and uses ∼60% of the memory bandwidth. On slow memory, the performance gap is reduced since we now saturate the 70 GB/s bandwidth. The baseline is always compute-bound and performs the same in both cases.

To evaluate complex expressions, we pick the selection from TPC-H that combines the most predicates, the selection on table `part` from `Q19`. The expression tree in optimal evaluation order, based on selectivities, is shown in Figure 6.10 and is neither in CNF nor in DNF. In Figure 6.11, we show the selection throughput using both uncompressed and

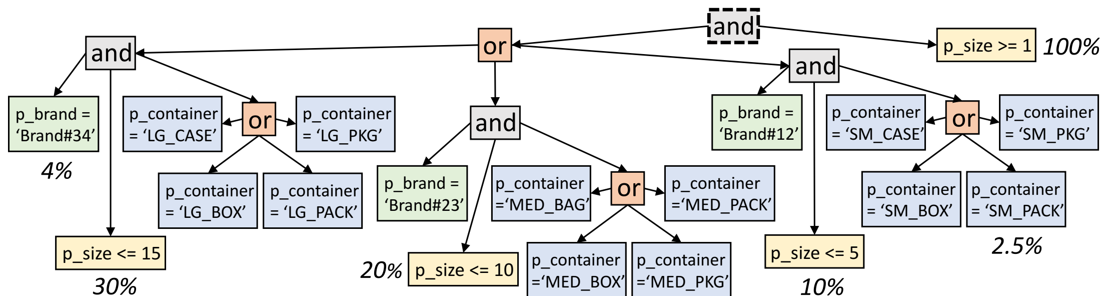



Figure 6.10: Expression tree for selection on part table from TPC-H Q19 (0.24% selectivity)

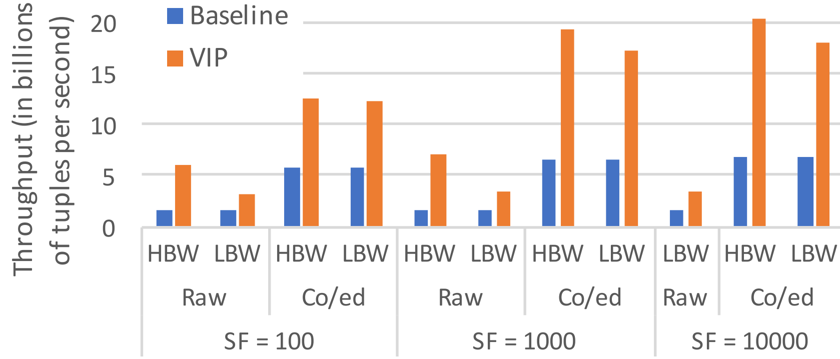



Figure 6.11: Selection on part table from TPC-H Q19

compressed data. The `p_brand` and `p_container` columns are `char(10)` with 25 and 40 distinct values respectively. The `p_size` column is a 32-bit integer with 50 distinct values. When compressed, we need 5, 6, and 5 bits respectively, reducing the footprint from 24 bytes to 2 bytes (16 bits) per tuple. The payload column is `p_partkey`, which is accessed for the qualifying 0.24% of tuples. We vary the scale factor (SF) of the dataset. For SF = 10000, the uncompressed data exceed the size of MCDRAM (HBW).

Our approach is 2.1–4.5X faster than the baseline and the best speedup is observed for scanning uncompressed data on MCDRAM. The large number of predicates highlights the efficiency of our design in handling complex expressions by handling bitmaps in the cache instead of compiling query-specific code. Since the selectivities are low, skipping predicates is crucial to performance, on both uncompressed and compressed data.

Both hash join and group-by aggregation use partitioning to ensure cache-conscious execution. In Figure 6.12, we show the partitioning throughput for multiple columns and

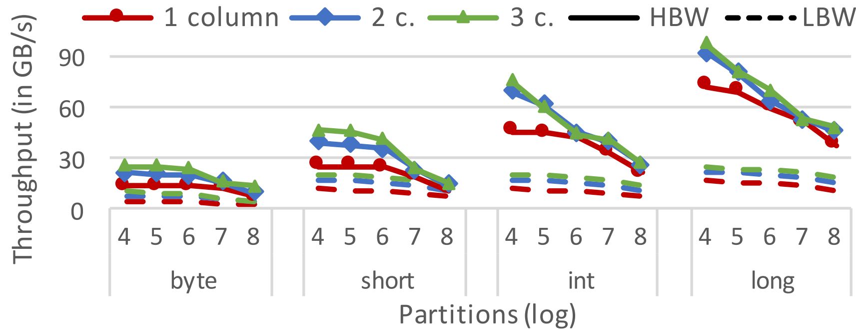



Figure 6.12: Partition of multiple columns and types

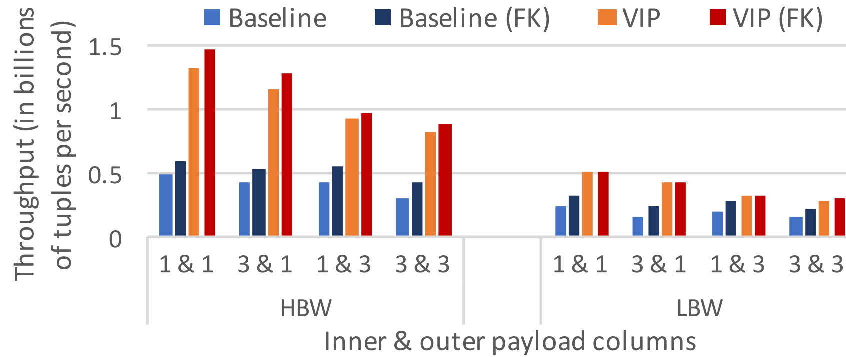



Figure 6.13: Hash join on synthetic data (1 key, 1–3 payloads, 128M ⋈ 32M tuples)

data types. The throughput improves with more payload columns, since we reuse the same output offsets. On fast memory (HBW), partitioning is compute-bound. As a result, payload columns with larger types use a higher portion of the bandwidth. Since random access occur in the cache, partitioning is up to 4X faster on MCDRAM (HBW) compared to DRAM (LBW), matching the 4X gap in bandwidth.

In Figure 6.13, we show the throughput of hash joins for 1 integer (32-bit) key column and 1–3 integer payload columns for both input tables. The data are synthetically generated to be uniform random. The join uses a foreign key so the inner keys are unique and the outer keys always find a match. The throughput is measured by the cardinality of the outer table. We use 128M tuples for the outer table and 32M tuples for the inner table. The output includes all the payload columns. In the baseline method, the payload columns are built in the hash table in a row-oriented layout to avoid additional cache misses when accessing each payload column. The hash bucket layout is query-specific as shown in Section 6.3.1. In the baseline method, executing a key–rid join and accessing the payloads via rid would be wasteful. In general, materializing payloads via simple rid lookup can in some cases be prohibitively expensive [Schuh *et al.*, 2016]. On fast memory (HBW), VIP is 1.9–2.4X faster, if we also utilize the fact that the inner keys are unique and terminate the hash probe per tuple as soon as a match is found (denoted FK). Without assuming a foreign key join, VIP is 2.1–2.8X faster. On fast memory, VIP is always compute-bound. On slow memory (LBW), VIP is 61%, 84%, 13% and 38% faster although both methods are memory-bound. On slow memory, we use fewer partitioning passes with larger fanout per pass.

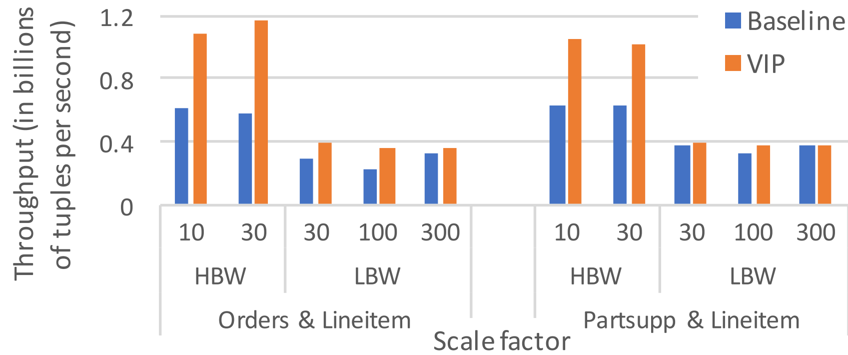



Figure 6.14: Hash joins using the core TPC-H tables

In Figure 6.14, we join the core tables of TPC-H. We join the tables **lineitem** with **orders** and **lineitem** with **partsupp**. The payloads are the foreign keys used to join the core tables with the next smaller dimension tables in the schema. These joins are part of every query that navigates multiple tables.

```
select l_partkey, l_suppkey, o_custkey
  from lineitem, orders
 where l_orderkey = o_orderkey;
```

The baseline method materializes the payloads using a query-specific layout for the hash table buckets and, since the inner keys are unique, we stop hash probing on the first match. In our approach, we use partitioning until the inner table fits in the cache and execute the hash join in blocks of tuples to remain cache-resident. On fast memory (HBW), VIP is 1.8–2X faster for the join of table **lineitem** with **orders** and 1.6–1.7X faster for the join of **lineitem** with **partsupp**.

```
select l_orderkey, l_partkey, l_suppkey
  from lineitem, partsupp
 where l_partkey = ps_partkey and l_suppkey = ps_suppkey;
```

The speedup is smaller on the join with **partsupp**, because in our design we evaluate each column of the composite key separately using rids after joining on the hash values. On slow memory (LBW), the two methods are largely equivalent. VIP is memory-bound due to the partitioning passes while the baseline is bound by random memory accesses. Notably, the baseline is noticeably slower for SF = 100 because the hash table is slightly larger than the L2 cache and causes excessive cache conflicts. By using a 4X larger hash table, we can achieve the throughput that we show for SF = 300.

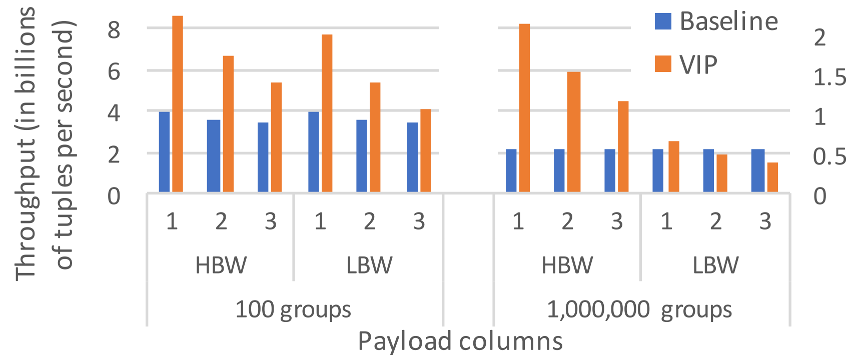



Figure 6.15: Group-by aggregation on synthetic data (1 32-bit group-by key, 1–3 32-bit payloads, 1 sum aggregate per payload, count(*), 256M tuples)

In Figure 6.15, we show the throughput of group-by aggregation on synthetic data. All columns are 32-bit integers, we have 1 group-by key column and 1–3 payload columns. We show the query for the case of 3 payload columns below.

```
select count(*), sum(x), sum(y), sum(z)
  from table group by x, y, z;
```

The baseline approach uses a query-specific hash table to store the group-by keys and update the partial aggregates. After the main loop has terminated, we scan through the hash table and finalize the aggregates. If the hash table is shared, the buckets are updated by locking the bucket via atomic instructions and spin locks. To lock the bucket we use the counter for the `count(*)` aggregate. The locking mechanism incorporates both shared and exclusive locks to avoid locking buckets before a match is found during hash probing.

For 100 groups, each thread updates a private hash table of partial aggregates. In VIP, group-by aggregation maps the group-by keys to group ids and then computes the aggregates by processing a block of tuples at a time. For 1 column and 1 sum, VIP is 2.2X faster. The speedup is reduced to 1.2X for 3 columns and 3 sums. The baseline performs the same on both fast and slow memory. VIP is up to 30% faster on fast memory.

For 1 million groups, the baseline method uses shared hash tables, while VIP uses partitioning to fit the partial aggregates in the cache. VIP is 3.9X faster for 1 payload column and 1 sum aggregate and 2.1X faster for 3 payloads and 3 sums. While we do not evaluate skew here, we can extend the operator to find frequent keys by sampling and then perform an extra pass to filter and aggregate them separately.

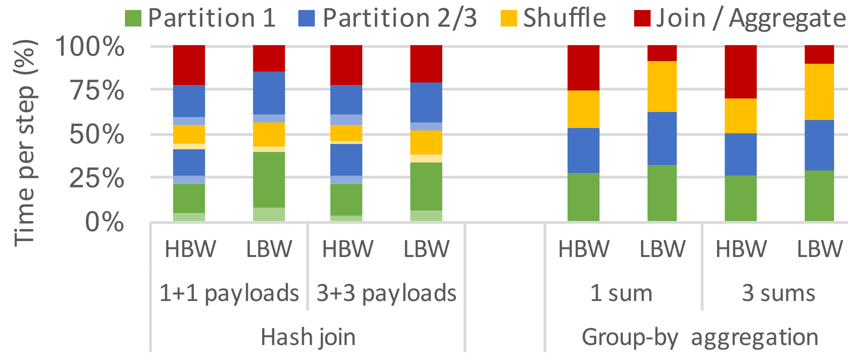



Figure 6.16: Step breakdown for join and aggregation

In Figure 6.16, we show the percentage of time spent on each step for hash join and group-by aggregation on synthetic data. We use the same settings as Figures 6.13 and 6.15 and vary the payload columns. On slow memory (LBW), hash joins perform fewer partitioning passes and the last join step takes ∼20% of the total execution time. For group-by aggregation, the last aggregation step takes a smaller fraction of the total time on slow memory, since the aggregates are very simple.

To evaluate VIP on a group-by aggregation with many aggregates and arithmetic expressions, we use TPC-H `Q1`. The group-by aggregation step of `Q1` is shown below. We use the smallest data type that fit the values of each column, which is a 16-bit integer for `l_quantity`, an 8-bit decimal for `l_tax`, an 8-bit decimal for `l_discount`, and a 32-bit decimal for `l_extendedprice`. The `l_returnflag` and `l_linestatus` group-by keys are both 1-byte strings. Since there are no `null` values in TPC-H, we can reuse the `count(*)` aggregate to compute the compute the `avg` aggregates as well. These optimizations are applied to both the baseline query-specific code and in the VIP input.

```
  select l_returnflag, l_linestatus,
         sum(l_quantity),  -- 64-bit sum
         sum(l_extendedprice),  -- 64-bit sum
         sum(l_extendedprice * (1 - l_discount)),  -- 64-bit sum expression rounded
         sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),  -- 64-bit sum expression rounded
         avg(l_quantity),  -- reuse the sum and count
         avg(l_extendedprice),  -- reuse the sum and count
         avg(l_discount)  -- compute separate sum and reuse count
         count(*)  -- the count used for average
    from lineitem
group by l_returnflag, l_linestatus;
```

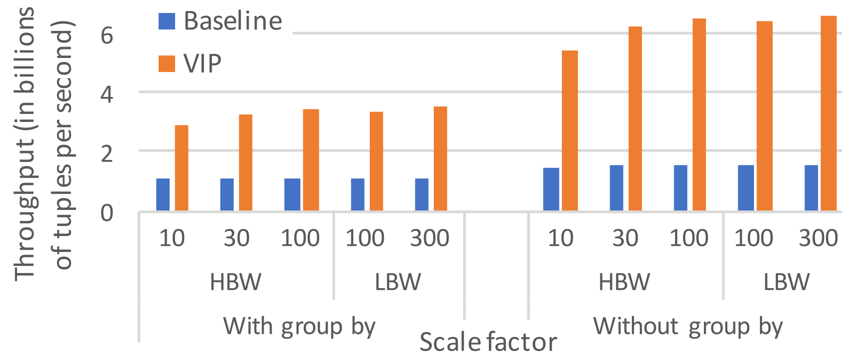



Figure 6.17: Group-by aggregation step from TPC-H Q1

In the baseline method, we compute the expressions one tuple at a time, keep the column values in registers, and update the partial aggregates in thread-private hash tables. In the VIP design, we compute one expression at a time for the next block of tuples and keep the intermediate results in the cache. We avoid recomputing common sub-expressions by reusing intermediate results across aggregates. As shown in Figure 6.17, VIP is 2.7–3.2X faster regardless of memory type. Estimating the groups is an order of magnitude faster here. Finally, we execute the same query without the group-by clause. Both methods now keep the partial aggregates in registers. In VIP, we compute each expression and aggregate separately for the next block of tuples. In the baseline method, we compute all aggregates for one tuple a time at once. VIP is 3.6–4.3X faster on both memory types.

## 6.6 Conclusion

We studied analytical query execution engine designs on the latest many-core CPUs. Combining a state-of-the-art design from mainstream CPUs with SIMD vectorization underutilizes the features of many-core CPUs due to being bound by random memory accesses. Instead, we proposed VIP, a query engine built bottom-up from pre-compiled data-parallel sub-operators, implemented entirely in SIMD, while utilizing the high-bandwidth on-chip MCDRAM of many-core CPUs to facilitate cache-conscious execution. VIP outperforms query-specific hand-written operators even if we ignore the compilation overhead. The VIP design effectively utilizes the hardware features of many-core CPUs and will also be relevant for the upcoming mainstream CPUs that will provide at least some of these features.

# Chapter 7

# Conclusions and Future Work

In this thesis, we optimized analytical query execution for all layers of modern hardware. Each chapter focused on a set of hardware layers, network, NUMA, main memory, processor caches, and the pipeline within each processor core. In each chapter, we also reuse ideas from other chapters, highlighting the fact that hardware conscious query execution optimizations are tightly coupled. In order to provide high-performance analytical query execution, while effectively utilizing modern hardware features, we need to apply the optimizations across layers in unison.

In Chapter 2, we introduced track join, a distributed join algorithm for minimal network traffic. Track joins avoids redundant transfers of tuples over the network by logically decomposing into Cartesian products and by generating optimal transfer schedules for each distinct join key. We motivated our work using a commercial database system where we showed that we can spend a significant amount of execution time transferring tuples over the network. Using track join, we provide an additional algorithm to trade-off CPU time for network traffic. We compared track join against hash join using both synthetic and real workloads and showed that track join always outperforms hash join in terms of network traffic. Furthermore, track join takes advantage of pre-existing locality to widen the gap against hash join.

Track join can facilitate complex strategies for data placement since it automatically identifies and takes advantage of locality. The typical placement strategies use hash distribution on one or more attributes but still need to redistribute the data to join on any

other attribute. Hash join eliminates all network traffic only if the data are pre-partitioned based on hashing. On the other hand, track join eliminates most network traffic as long as the matching tuples are on the same nodes. Thus, we can potentially use more advanced placement strategies to cluster matching tuples across multiple dimensions and then use track join to detect the collocated matching tuples and avoid transferring them over the network. Even in the absence of locality, track join naturally adapts between broadcast join and hash join. Thus, we can use it for joins on intermediate results when the cardinalities cannot be estimated with sufficient confidence.

To reduce the end-to-end execution time, we can stress the overlap of CPU processing with network transfers by splitting the input in parts that are executed in parallel. Then, we can pipeline the different phases of different parts, executing a different phase of each part at the same time. Thus, if a phase is CPU-bound while another phase is network-bound, we overlap them to better utilize the hardware. Additionally, if the network transfers are not balanced, we can relax the requirement for minimal network traffic and transfer data to other nodes to ensure their CPUs do not remain idle.

In Chapter 3, we studied a multiple partitioning variants across all layers of the memory hierarchy. Partitioning is a vital operator for in-memory query execution since it facilitates cache-resident execution by eliminating cache misses. Prior to our work, certain types of partitioning were prohibitively expensive and there was no practical space-time trade-off since any in-place partitioning algorithm would be significantly slower than the non-in-place counterpart. In our work, we introduced the first large-scale in-place partitioning algorithm, showed how to minimize the transfers across CPUs via the NUMA interconnection layer, and designed a SIMD-based range index to optimize the computation of range partitioning function, making range partitioning comparably fast with radix or hash partitioning.

We show that all partitioning variants can be efficient and provide practical trade-offs between time and space. This work can serve as a tool for building other operations such as joins and aggregation by combining the most suitable partitioning variants that best meet the design goals. We highlight the versatility of the different partitioning variants by using them to build multiple sorting algorithms that have different advantages and disadvantages. We build time-efficient sorting using LSB radix-sort, space-efficient sorting using in-place

MSB radix-sort, and skew-efficient sorting using comparison-based sorting via range partitioning. All these sorting algorithms are the fastest to date in their respective category, highlighting the efficiency of our algorithms and implementations across all partitioning variants.

In Chapter 4, we studied lightweight in-memory compression. Databases use dictionary compression that allows queries to be executed directly on the compressed data. The dictionary encoding bits can be stored in multiple variants with different trade-offs. The two main variants is horizontal bit packing, where the bits are stored contiguously, and vertical bit packing, where the bits are interleaved. In our work, we optimized the compression and decompression of vertical bit packing beyond an order of magnitude using SIMD instructions. We also reimplemented introducing slight improvements to other variants. In our evaluation, we compare the compression, decompression, and selection scan speed of horizontal and vertical bit packing across multiple configurations. Our evaluation highlights the trade-offs between the encoding variants, showing that horizontal bit packing is easier to decompress but vertical bit packing allows for very fast scans without incurring prohibitive cost in compressing or decompressing the layout.

In Chapter 5, we presented advanced SIMD vectorization techniques for the most fundamental in-memory database operators including random-access operators that are not sequential and hence not inherently data parallel. Selection scans and compression are standard use cases for SIMD since the data access is contiguous and in most cases already data parallel. However, for operations such as hash tables, the memory access pattern is non-sequential, thus limiting the effectiveness of basic SIMD instructions. In this work, we extend SIMD vectorization to cover non-sequential operators covering almost all in-memory query execution operators. Initially, we define a set of fundamental vectorization operations for contiguous and non-contiguous memory access that we reuse in our vectorized implementations of multiple database operators. Then, we describe vectorized algorithms for building and probing hash tables with multiple hashing schemes as well as other non-sequential access data structures such as Bloom filters. We also describe vectorized algorithms for in-memory partitioning. Finally, we combine our vectorized algorithms to build sorting and multiple hash join variants.

Our vectorized algorithms maximize efficiency for compute-bound query execution for any SIMD-capable hardware without relying on special instructions besides for the basic non-contiguous memory accesses, namely gathers and scatters. In contrast to ad-hoc vectorized SIMD implementations by earlier work, our vectorized algorithms utilize all vector lanes regardless of the size of the vector and are designed to be more efficient as the vector size increases and outperform scalar code by almost an order of magnitude even for operations that are not data parallel. Our vectorized algorithms transform complicated control flow dependencies into data parallel data flow dependencies without utilizing operator-specific ad-hoc optimizations and without changing the layout of the data structures to match the SIMD layout. As a result, they can be used to implement vectorized implementations for the most important database operators across multiple SIMD instruction sets or hardware generations with different SIMD register sizes.

In our work, we also highlight the impact of SIMD vectorization in determining the best algorithm for each database operator. For instance, we show that non-partitioned hash joins are faster than partitioned hash joins using scalar code but partitioned hash joins become significantly faster if SIMD is used for both. In general, compute-bound algorithms are expected to get a substantial speedup from SIMD vectorization potentially outperforming the alternatives that were measured to be faster in scalar code. Moreover, our work shows that simple CPU cores can be comparably fast to complex cores for in-memory query execution, raising the question for which type of hardware is more efficient or more energy efficient for database workloads.

In Chapter 6, we proposed VIP, a new query engine design to more effectively utilize the features of the latest many-core CPUs, namely large SIMD vectors and high-bandwidth memory. Query compilation and operator pipelining form the most popular design for analytical database execution engines on mainstream CPUs. We combine this design with advanced SIMD vectorization using the vectorization techniques described in Chapter 5. Nevertheless, the mainstream CPU design diminishes the impact of both SIMD vectorization and high-bandwidth memory due to being dominated by random memory accesses and cache misses. As a result, we propose a new design, VIP, based on the decomposition of database operators into pre-compiled data-parallel sub-operators that are designed to be data parallel

and are implemented entirely in SIMD. The design favors partitioning and cache conscious execution by taking advantage of the high-bandwidth on-chip memory. The high-bandwidth memory has significantly faster bandwidth than DRAM but not as much better latency. Thus, partitioning and cache conscious execution take advantage of the bandwidth boost and are unaffected by memory latency since they eliminate cache misses.

The VIP design uses vectorization in the context of a realistic query engine. We support multiple data types and complex predicates, in contrast to earlier work that uses keys and record identifiers and is typically limited to a single basic predicate per operator. VIP is the first fully vectorized query execution engine and was designed bottom up from sub-operators designed to be data parallel, in contrast to SIMD re-implementations of operators designed and implemented in scalar code. VIP executes operators for a block of tuples at a time but each sub-operator processes a single column for that block. The sub-operators are pre-compiled and are not query specific, thus avoiding the per-query compilation overhead. In VIP, we also introduce the first columnar group-by algorithm, we show how to combine selection scans with compression while avoiding to decompress data that can be skipped, and we describe how to evaluate numeric operators in SIMD. Finally, our evaluation shows that, on many-core CPUs, the VIP design achieves significantly higher performance than the design used in mainstream CPUs by utilizing their additional hardware features, advanced SIMD vectorization and high-bandwidth memory, more effectively.

In VIP, we used early materialization of payloads as part of each operator. Nevertheless, the engine design can also support late materialization as well as hybrid approaches. Late materialization may be preferable when the number of payload columns is high or the data types are large. We plan to further study materialization strategies and operators built in the VIP design using data-parallel sub-operators. Furthermore, we can extend the design by pipelining multiple operators when cache misses cannot occur. For instance, we can combine the output of the selection scan with the input of the first partitioning pass, hash join, or group-by aggregation, while still processing one block of tuples at a time with sub-operators.

Overall, our work optimizes analytical query execution by focusing on a different layer of the hardware hierarchy in each chapter. Certain optimizations can be either orthogonal

to optimizations on other layers and can be applied independently of other design decisions. Other optimizations require a design that enforces specific design choices on other layers. Even in the latter case, using each optimization as proposed here in isolation will still improve performance. For example, even if we do not use partitioning and cache resident execution, SIMD vectorization will still marginally improve performance for queries whose execution time is dominated by cache misses, and will still provide significant speedups to queries with high selectivity and small joins. Track join can also work in a pipelined and a non-pipelined fashion without affecting the goal of minimizing the network traffic. A final example is the use of NUMA-aware partitioning to minimize transfers across CPUs. While this optimization works in synergy with cache resident execution, we can minimize the partitioning fanout to minimize transfers across CPUs without enforcing cache-resident execution for each CPU and NUMA region.

This thesis presents holistic solutions to well defined problems that we believe to be substantial steps towards saturating the hardware capabilities for analytical query execution. We present a generic distributed join algorithm for minimizing network traffic that is orthogonal to compression, schema properties, or data organization within the database. The present partitioning variants we study cover all types of partitioning and offer practical time and space trade-offs without limiting to a specific partitioning variant which might not be suitable to meet certain design goals. The compression algorithms also cover multiple variants and present practical trade-offs. Our work on vectorized algorithms is designed to take advantage of future hardware with stronger SIMD capabilities for nearly all database operators and the fundamental operations to the control flow to data flow transformation can be reused to extend vectorization to other algorithms. Finally, the VIP design uses these advanced optimizations in the premise of a realistic analytical query engine combining optimizations from multiple hardware layers into a single synergistic design.

Finally, we believe that the algorithms and implementations presented in this thesis transcend the current and near-future generation of hardware advances and are not bound by artifacts specific to the current generation of hardware capabilities, including network and memory technologies as well as mainstream and non-mainstream processors and co-processors. For instance, we introduced a distributed join algorithm that introduces a

trade-off between CPU and network. Future processors and network technologies will always present trade-offs between scale-up to more advanced shared-memory machines and scale-out to larger numbers of machines. Furthermore, we presented techniques for optimal usage of the memory hierarchy layers and as the number of memory layers increases over time, the same trend is likely to continue and introduce additional layers. Advances on modern processors are focused on extracting additional performance in the same energy budget by evolving on three sources of parallelism: instruction level parallelism, thread parallelism, and data parallelism. Here, we described generic vectorized algorithms that fully utilize the data parallelism capabilities provided by modern and future processors. Providing the platform for query execution engines to effectively utilize data parallelism is a vital step towards hardware-conscious database systems.

# Bibliography

[Abadi *et al.*, 2006] Daniel J. Abadi, Samuel R. Madden, and Miguel C. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[Abadi *et al.*, 2007] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, pages 466–475, 2007.

[Afrati and Ullman, 2010] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[Alagiannis *et al.*, 2014] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2o: A hands-free adaptive store. In *SIGMOD*, pages 1103–1114, 2014.

[Albutiu *et al.*, 2012] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, June 2012.

[Arulraj *et al.*, 2016] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *SIGMOD*, pages 583–598, 2016.

[Bakkum and Skadron, 2010] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, pages 94–103, 2010.

[Balkesen *et al.*, 2013a] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Ozsu. Multicore, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, September 2013.

[Balkesen *et al.*, 2013b] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[Barthels *et al.*, 2015] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using rdma. In *SIGMOD*, pages 1463–1475, 2015.

[Barthels *et al.*, 2017] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, January 2017.

[Bernstein and Chiu, 1981] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, January 1981.

[Bernstein *et al.*, 1981] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. Query processing in a system for distributed databases. *TODS*, 6:602–625, 1981.

[Binnig *et al.*, 2016] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, March 2016.

[Blanas *et al.*, 2011] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48, 2011.

[Bloom, 1970] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, July 1970.

[Boncz *et al.*, 2005] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[Cheng *et al.*, 2017] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture. In *CIKM*, pages 657–666, 2017.

[Chhugani *et al.*, 2008] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*, pages 1313–1324, 2008.

[Cho *et al.*, 2015] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. PARADIS: An efficient parallel algorithm for in-place radix sort. *PVLDB*, 8(12):1518–1529, August 2015.

[Chu *et al.*, 2015] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[Cieslewicz and Ross, 2007] John Cieslewicz and Kenneth A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[Cieslewicz *et al.*, 2010] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, pages 483–494, 2010.

[Cudre-Mauroux *et al.*, 2009] Philippe Cudre-Mauroux, Hideaki Kimura, K.-T. Lim, Jennie Rogers, R. Simakov, Emad Soroush, Pavel Velikhov, Daniel L. Wang, Magdalena Balazinska, Jacek Becla, David J. DeWitt, B. Heath, D. Maier, Samuel R. Madden, Jignesh M. Patel, Michael R. Stonebraker, and Stan B. Zdonik. A demonstration of sciDB: A science-oriented DBMS. *PVLDB*, 2(2):1534–1537, August 2009.

[Dageville *et al.*, 2016] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *SIGMOD*, pages 215–226, 2016.

[Dean and Ghemawat, 2004] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[DeWitt and Gray, 1992] David J. DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *CACM*, 35:85–98, 1992.

[DeWitt *et al.*, 1984] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.

[DeWitt *et al.*, 1990] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I. Hsiao, and Rick Rasmussen. The Gamma database machine project. *TKDE*, 2(1):44–62, March 1990.

[Epstein *et al.*, 1978] Robert Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In *SIGMOD*, pages 169–180, 1978.

[Feng and Lo, 2015] Ziqiang Feng and Eric Lo. Accelerating aggregation using intra-cycle parallelism. In *ICDE*, pages 291–302, 2015.

[Feng *et al.*, 2015] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. ByteSlice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*, pages 31–46, 2015.

[Flajolet and Martin, 1985] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, September 1985.

[Fowler *et al.*, 2017] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald Eastlake, and Tony Hansen. The FNV non-cryptographic hash algorithm. Technical report, 2017. http://www.ietf.org/internet-drafts/draft-eastlake-fnv-13.txt.

[Frey *et al.*, 2009] Philip W. Frey, Romulo Goncalves, Martin L. Kersten, and Jens Teubner. Spinning relations: high-speed networks for distributed join processing. In *DaMoN*, pages 27–33, 2009.

[Graefe, 1994] Goetz Graefe. Volcano: An extensible and parallel query evaluation system. *TKDE*, 6(1):120–135, February 1994.

[Grund *et al.*, 2010] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel R. Madden. HYRISE: A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, November 2010.

[Gupta *et al.*, 2015] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.

[Hofmann *et al.*, 2014] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips. *CoRR*, arXiv:1401.7494, 2014.

[Holloway *et al.*, 2007] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: Compression and bandwidth trade offs for database scans. In *SIGMOD*, pages 389–400, 2007.

[Idreos *et al.*, 2007] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[Inoue and Taura, 2015] Hiroshi Inoue and Kenjiro Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8(11):1274–1285, July 2015.

[Inoue *et al.*, 2007] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, pages 189–198, 2007.

[Inoue *et al.*, 2014] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, November 2014.

[Jha *et al.*, 2015] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB*, 8(6):642–653, February 2015.

[Johnson *et al.*, 2008] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, August 2008.

[Johnson *et al.*, 2009] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[Kara *et al.*, 2017] Kaan Kara, Jana Giceva, and Gustavo Alonso. Fpga-based data partitioning. In *SIGMOD*, pages 433–445, 2017.

[Kim *et al.*, 2009] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, August 2009.

[Kim *et al.*, 2010] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[Kim *et al.*, 2012] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. CloudRAMsort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *SIGMOD*, pages 841–850, 2012.

[Kitsuregawa *et al.*, 1983] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1:63–74, 1983.

[Krikellas *et al.*, 2010] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[Lamport, 1975] Leslie Lamport. Multiple byte processing with full-word instructions. *CACM*, 18(8):471–475, August 1975.

[Lang *et al.*, 2016] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid OLTP and OLAP on com-

pressed storage using both vectorization and compilation. In *SIGMOD*, pages 311–326, 2016.

[Larsen and Amarasinghe, 2000] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, pages 145–156, 2000.

[Lee *et al.*, 2014] Jae-Gil Lee, Gopi Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy Lohman, Konstantinos Morfonios, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Vincent Kulandai Samy, Richard Sidle, Knut Stolze, and Liping Zhang. Joins on encoded and partitioned data. *PVLDB*, 7(13):1355–1366, August 2014.

[Leis *et al.*, 2014] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[Lemire and Boytsov, 2015] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper.*, 45(1):1–29, January 2015.

[Lemire *et al.*, 2016] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exper.*, 46(6):723–749, June 2016.

[Li and Patel, 2013] Yinan Li and Jignesh M. Patel. BitWeaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[Li and Ross, 1995] Zhe Li and Kenneth A. Ross. PERF join: an alternative to two-way semijoin and Bloomjoin. In *CIKM*, pages 137–144, 1995.

[Li *et al.*, 2013] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[Mackert and Lohman, 1986] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, pages 149–159, 1986.

[Manegold *et al.*, 2000a] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *J. VLDB*, 9(3):231–246, 2000.

[Manegold *et al.*, 2000b] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What happens during a join? dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[Manegold *et al.*, 2002] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. *TKDE*, 14(4):709–730, July 2002.

[Manegold *et al.*, 2004] Stefan Manegold, Peter A. Boncz, Niels Nes, and Martin L. Kersten. Cache-conscious radix-decluster projections. In *VLDB*, pages 684–695, 2004.

[Menon *et al.*, 2017] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 2017.

[Mühlbauer *et al.*, 2013] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, September 2013.

[Müller *et al.*, 2015] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*, pages 1123–1136, 2015.

[Mullin, 1990] James K. Mullin. Optimal semijoins for distributed database systems. *TSE*, 16(5):558–560, May 1990.

[Musser, 1997] David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993, August 1997.

[Neumann, 2011] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, June 2011.

[Okcan and Riedewald, 2011] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.

[Pagh and Rodler, 2004] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[Pavlo *et al.*, 2009] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel R. Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[Pirk *et al.*, 2011] Holger Pirk, Stefan Manegold, and Martin L. Kersten. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS*, 2011.

[Pirk *et al.*, 2014a] Holger Pirk, Stefan Manegold, and Martin L. Kersten. Waste not... efficient co-processing of relational data. In *ICDE*, pages 508–519, 2014.

[Pirk *et al.*, 2014b] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Database cracking: Fancy scan, not poor man's sort! In *DaMoN*, 2014.

[Pirk *et al.*, 2015] Holger Pirk, Samuel R. Madden, and Michael Stonebraker. By their fruits shall ye know them: A data analyst's perspective on massively parallel system design. In *DaMoN*, 2015.

[Pirk *et al.*, 2016] Holger Pirk, Oscar Moll, Matei Zaharia, and Samuel R. Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *PVLDB*, 9(14):1707–1718, October 2016.

[Polychroniou and Ross, 2013] O. Polychroniou and Kenneth A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, 2013.

[Polychroniou and Ross, 2014a] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.

[Polychroniou and Ross, 2014b] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *DaMoN*, 2014.

[Polychroniou and Ross, 2015] Orestis Polychroniou and Kenneth A. Ross. Efficient lightweight compression alongside fast scans. In *DaMoN*, 2015.

[Polychroniou *et al.*, 2014] Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. Track join: Distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494, 2014.

[Polychroniou *et al.*, 2015] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[Psaroudakis *et al.*, 2016] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 10(2):37–48, October 2016.

[Raman and Swart, 2006] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *VLDB*, pages 858–869, 2006.

[Raman *et al.*, 2013] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, August 2013.

[Rödiger *et al.*, 2014] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014.

[Rödiger *et al.*, 2015] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, December 2015.

[Rödiger *et al.*, 2016] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, pages 1194–1205, 2016.

[Ross and Cieslewicz, 2009] Kenneth A. Ross and John Cieslewicz. Optimal splitters for database partitioning with size bounds. In *ICDT*, pages 98–110, 2009.

[Ross, 2004] Kenneth A. Ross. Selection conditions in main memory. *TODS*, 29(1):132–161, March 2004.

[Ross, 2007] Kenneth A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[Roussopoulos and Kang, 1991] Nick Roussopoulos and Hyunchul Kang. A pipeline n-way join algorithm based on the 2-way semijoin program. *TKDE*, 3(4):486–495, December 1991.

[Roy *et al.*, 2012] Pratanu Roy, Jens Teubner, and Gustavo Alonso. Efficient frequent item counting in multi-core hardware. In *SIGKDD*, pages 1451–1459, 2012.

[Satish *et al.*, 2010] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[Schuh *et al.*, 2016] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976, 2016.

[Seiler *et al.*, 2008] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graphics*, 27(3), August 2008.

[Shin, 2007] Jaewook Shin. Introducing control flow into vectorized code. In *PACT*, pages 280–291, 2007.

[Sidirourgos and Kersten, 2013] Lefteris Sidirourgos and Martin L. Kersten. Column imprints: A secondary index structure. In *SIGMOD*, pages 893–904, 2013.

[Sidirourgos and Mühleisen, 2017] Lefteris Sidirourgos and Hannes Mühleisen. Scaling column imprints using advanced vectorization. In *DaMoN*, 2017.

[Sidler *et al.*, 2017] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *SIGMOD*, pages 403–415, 2017.

[Sirin *et al.*, 2016] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. Micro-architectural analysis of in-memory OLTP. In *SIGMOD*, pages 387–402, 2016.

[Sitaridi and Ross, 2012] Evangelia A. Sitaridi and Kenneth A. Ross. Ameliorating memory contention of olap operators on gpu processors. In *DaMoN*, pages 39–47, 2012.

[Sitaridi and Ross, 2013] Evangelia A. Sitaridi and Kenneth A. Ross. Optimizing select conditions on GPUs. In *DaMoN*, 2013.

[Sitaridi and Ross, 2016] Evangelia A. Sitaridi and Kenneth A. Ross. Gpu-accelerated string matching for database applications. *J. VLDB*, 25(5):719–740, October 2016.

[Sitaridi *et al.*, 2016] Evangelia A. Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. SIMD-accelerated regular expression matching. In *DaMoN*, 2016.

[Stamos and Young, 1993] James W. Stamos and Honesty C. Young. A symmetric fragment and replicate algorithm for distributed joins. *TPDS*, 4(12):1345–1354, December 1993.

[Stehle and Jacobsen, 2017] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *SIGMOD*, pages 417–432, 2017.

[Stepanov *et al.*, 2011] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. SIMD-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.

[Stonebraker *et al.*, 2005] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch F. Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[Stonebraker *et al.*, 2007] Michael Stonebraker, Samuel R. Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[Thusoo *et al.*, 2009] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, August 2009.

[Verbitski *et al.*, 2017] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, pages 1041–1052, 2017.

[Wang *et al.*, 2017] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, pages 993–1008, 2017.

[Wassenberg and Sanders, 2011] Jan Wassenberg and Peter Sanders. Engineering a multi core radix sort. In *EuroPar*, pages 160–169, 2011.

[Willhalm *et al.*, 2009] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, August 2009.

[Wong *et al.*, 1985] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed files. In *VLDB*, pages 448–457, 1985.

[Wu *et al.*, 2013] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, pages 249–260, 2013.

[Wu *et al.*, 2014] Lisa Wu, Orestis Polychroniou, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Energy analysis of hardware and software range partitioning. *TOCS*, 32(3), August 2014.

[Ye *et al.*, 2011] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, pages 1–9, 2011.

[Yu *et al.*, 2013] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. CS2: a new database synopsis for query estimation. In *SIGMOD*, pages 469–480, 2013.

[Zamanian *et al.*, 2015] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, pages 17–30, 2015.

[Zhou and Ross, 2002] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[Zukowski *et al.*, 2006] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.