Hasso Plattner Institute

Chair for Data Engineering Systems

Proposal Master Thesis

Hardware-Conscious SIMD-Accelerated Sort-Merge Joins in Multi Core In-Memory Database Systems

Finn Schöllkopf

Time frame: October 2024 - March 2024

Supervisor

Prof. Dr. Tilmann Rabl

Advisor

Florian Schmeller

1 Motivation

Traditional database systems have historically been designed for systems and architectures where I/O dominates performance. However, modern processors with multi-core architectures, advanced instruction sets, and other hardware accelerants like vector operations (SIMD) have significantly altered this landscape. Today's in-memory database systems are no longer I/O bound and, therefore, need high intra-operator parallelism to utilize the multi-core architecture fully. To achieve maximum performance, cache locality, NUMA awareness, and using SIMD instructions for higher data parallelism should be considered. The join operator is a fundamental component of every database system. In recent years, the difference in performance between the sort-merge and radix-hash join has been the subject of ongoing debate. Kim et al. [10] projected that Sort-Merge Join would outperform hash-based alternatives with a factor of 1.35 – 1.65 with 512-bit SIMD. Albutiu et al. [1] reinforced this claim with recent results reporting that their NUMA-aware implementation of sort-merge join is superior to that of hash joins (without leveraging SIMD). Balkesen et al. [2] experimentally show contradicting results by implementing optimized versions for sort-merge and radix-hash join, showing that their implementation of radix-hash join is still superior. They use AVX2 in their implementation, allowing further work to explore wider SIMD registers (e.g., AVX512).

Despite ongoing research, public implementations of join algorithms optimized for modern hardware are hard to find. Most existing implementations are proprietary or experimental, limiting their accessibility and usefulness to the research community and database developers. An open-source, state-of-the-art implementation of a sort-merge join optimized for different architectures would help address this need. Such an implementation serves as a valuable baseline for researchers looking to evaluate or improve upon existing methods, and it also contributes to advancing database system design by providing a solid foundation for future innovation.

2 Goal of Thesis

This thesis aims to efficiently implement the sort-merge join algorithm, explicitly optimized for specific architectures and hardware components. As Equi-joins are the most common type of join operation, particularly in relational databases, we will restrict ourselves to an Equi-join implementation and then optionally extend upon this later.

While multiple papers exist about modern implementation approaches for sort-merge joins in in-memory database systems and SIMD sorting generally, only some have public implementations[1]. Most SIMD sorting algorithms presented in the literature are not directly applicable to join operations as they usually use sorting

[1]Implementation of [2] published at `https://archive-systems.ethz.ch/node/334`

keys of only 32 bits. We must additionally track the row ID (rid) corresponding to the sorting key for a join, requiring at least 64-bit elements. The current implementations of sort-merge join in literature use SSE and AVX2 intrinsics, but to our knowledge, there has yet to be an implementation using AVX512.

Therefore, in the scope of this thesis, we want to integrate support for modern AVX512 sorting algorithms ([15], [16]) next to SSE and AVX2 into a complete sort-merge join operator. It would also be of value to see how new and existing approaches transfer to other CPU architectures like Arm with its Scalable Vector Extension (SVE) or Power with its Vector Scalar Extension (VSX). We are most interested in AWS Graviton 3/4, which offers 256-bit SVE, and Power9/10, which, unfortunately, only offers 128-bit vector registers.

While some public implementations exist for modern and optimized sort-merge join, they have usually isolated implementations with a strong focus on the sorting step using randomly chosen input data, often already in the required data format. Also, they often skip the lookup of matching rows and the construction of the joined table. Hence, in this thesis, we want to integrate our implementation of the sort-merge join into Hyrise [8], a research in-memory database. Hyrise contains both a radix-based Hash-Join and sort-merge join. The sort-merge join uses radix cluster sorting, which uses pattern-defeating quicksort (boost) but no explicit SIMD instructions. It fundamentally differs from the modern approaches in the literature. These differences allow us to test our implementation against the existing sort-merge and hash-based join. Complete integration into an in-memory database allows us to run decision support benchmarks like TCP-H, TCP-DS, and the Public BI benchmark to compare operators to other implementations in a more realistic scenario.

Benchmarks like TCP-H have schemas and datatypes carefully designed by experts in database design. Hence, they can fail to capture the chaotic nature of real-world applications [14]. For instance, TCP-H only uses integer values for keys. 32-bit integer values do not require any change in data format to be SIMD sortable. However, in many Business Intelligence applications, strings are used for various types, e.g., to deal with dirty data that is not parsable. String join keys complicate SIMD sorting, as multiple strings do not fit into SIMD registers. Therefore, we must reduce the key size by compression, prefix functions, or hashing. Shortening the key size can introduce false positives, which need to be filtered. We could accelerate traditional string sorting through SIMD in other ways (e.g., SIMD accelerated string comparison), not requiring any form of compression, but methods like sorting networks and bitonic merge networks will likely not be applicable.

As strings are variable in size, it makes sense to consider their internal representation and encoding. For instance, in a dictionary encoding, the indices can be used as a sorting key rather than the string itself. Other representations allow for cheap access to a prefix or short string. For instance, Umbra's "German String" [11] consists of a 128-bit struct and are adopted by more recent databases like CedarDB and DuckDB. The first 32 bits represent the length. The remaining bits hold the complete string if the length is at most 12. Otherwise, the struct consists of the 32-bit length, a 32-bit prefix, and a pointer to a storage location. Due to saving pointer dereferences, this can speed up comparison, lexicographical sorting, and other prefix operations.

Benchmarking should also include measuring the sorting throughput in tuples per second and all algorithmic steps: initial data construction in the format of (key, rid) from the input relations, sorting, finding join partners, and the final construction of the joined table. We want to test our implementation on different architectures and hardware, evaluating differences in core count, cache size, SIMD registers widths, NUMA regions, and other hardware-specific properties.

3 Approach

The sort-merge join involves sorting both input relations. It is the most crucial and time-consuming part of the sort-merge join operation. Therefore, optimization efforts should primarily focus on this step, as it largely determines the runtime.

Due to modern multi-core architectures, sorting should intensively utilize thread-level parallelism by multithreading. With the recent architectural trends of wider register widths for SIMD, sorting should also heavily use SIMD instructions to exploit data parallelism. In a multi-core context, merge sort is often preferred over quicksort, as the parallelization of the divide-and-conquer approach is straightforward, and it has other advantages over quicksort such as more predictable and cache-friendly memory access patterns and better load balancing through equal-sized partitioning.

Sorting through SIMD registers can be achieved through sorting networks [4]. The sorting network compares elements in parallel in each step using SIMD min/max operations. A final transposition is needed, which requires additional SIMD shuffle instructions to complete the sorting. We can build sorting kernels for various input sizes[2] depending on the data type and register size.

Merging can also benefit from SIMD acceleration. There are two standard merging networks: bitonic merge networks and odd-even merge networks ([4], SIMD accelerated [9]). Both scale poorly for bigger input sizes, with odd-even networks requiring slightly fewer comparisons but instead involving data movement and element masking. Therefore, we can use SIMD-accelerated merging networks as a kernel for small input sizes, e.g., by sequentially pulling already sorted data into SIMD registers and calling the merge kernel, which writes to the output and then fetches new data.

We can merge different subparts of the data in different threads as long as we have enough sorted sublists. In the later round of the merge tree, with only a few sorted sublists remaining, it becomes increasingly more challenging to parallelize

[2]`https://bertdobbelaere.github.io/sorting_networks.html`

efficiently. However, even at this point, we can parallelize. One way parallelization is made possible is through the Merge Path [13]. This conceptual path allows us to parallelize a two-way merge by splitting it into non-overlapping segments that form disjoint sets of elements. We can then sequentially merge these segments in parallel. The sequential merging can again benefit from SIMD acceleration [15]. In the later stages, out-of-cache merging becomes necessary, quickly resulting in the memory bandwidth becoming the bottleneck of even a single-threaded merge routine.

Therefore, multi-way merging [2]. The merge-tree consists of multiple two-way merge units (managed as tasks) connected via FIFO queues , is introduced. Only the leaves of the merge tree load data from memory. Blocking and task switching ensure the combined FIFO queues fit into the CPU cache. This way, memory bandwidth can be reduced with a slight CPU overhead. Optionally, we could explore merging through other primitives, such as tournament trees and priority queues.

We can exploit other hardware properties, for instance through NUMA-aware partitioning to speed up memory access. Before we can sort our input relations, the tuples need to be translated into a SIMD sortable format. Usually, a 64-bit pair (key, rid) (key & rid both 32-bit) is assumed. We, therefore, support a maximum relation size of 2^{32}. With most value types greater than 32 bits, we need to compress the values of the join columns to 32 bits. Methods like key-prefix [12] and XOR- and shift-based hash functions [6] have been used to represent keys using 32 bits.

After sorting both input relations, a final loop over both sorted input relations suffices to find all join candidates. The sorted data is of the form (key, rid). Hence, we can use the row ID (rid) to find the respective tuples. As meantioned before some compression was used to genereate the 32-bit sorting key from the join column value. Therefore, we might require additional validation and filtering in the final merge step.

4 Related Work

Many papers describe how sorting can be done efficiently in modern multi-core architectures for SIMD-accelerated sorting. Chhugani et al. [7] describe the concepts needed for efficient (SSE) SIMD sorting for both single- and multi-core execution, including sorting networks, bitonic- and odd-even merge networks, and how to deal with memory bandwidth limitations for large problem sizes through multiway merging. There are other ideas like MergePath [13] for merging only a few very large sublists in parallel and SIMD accelerated. Kim et al. [10] implemented a sort-merge join using SSE intrinsics using these same concepts, projecting performance for wider SIMD widths that would outperform hash joins. Albutiu et al. [1] present MPSM, a sort-merge join implementation designed for modern multi-core and multi-socket NUMA processors using their custom sorting routine without SIMD. They did an experimental evaluation on a 32-core (4 socket) system, concluding that their sort-merge join implementation is faster than the respective hash join implementation of Blanas et al. [5]. Recent studies show that parallel radix-hash join has the best overall performance [3].

Therefore, Balkesen et al. [2] experimentally studied the performance of main-memory, parallel, multi-core join, and NUMA-aware algorithms, focusing on sort-merge and radix-hash join. They claim to provide the fastest in-memory join processing algorithms using sorting and hashing, and that sort-merge join gets more comparable in performance to radix-hash join with very large input sizes. Still, they conclude that the radix-hash join exceeds the sort-merge join for 256-bit SIMD. None of the papers mentioned above take advantage of 512-bit SIMD. The hash join operator implemented in Hyrise is based on [3] and [2], allowing for good comparison between a sort-merge join operator inside the Hyrise system. Other in-memory databases like DuckDB also employ parallel radix-hash joins.

There is research on SIMD sorting using 512-bit SIMD ([15],[16]). Still, to our knowledge, no literature exists on implementing sort-merge join with the same optimizations and concepts like multiway merging for 512-bit SIMD. The much more extensive instruction set of AVX-512, with its gather/scatter intrinsics, improved masking, and new instruction for data manipulation like compress and expand, might allow for further improvements, besides just offering a wider vector width, by creatively using these new instructions.

5 Project Plan

Sketch of a time line for the thesis with major milestones, e.g.

Time	**Writing/Research**	**Prototype**

Table 1: Planned Time Table

References

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, jun 2012.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, sep 2013.

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373, 2013.

[4] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[5] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 37–48, New York, NY, USA, 2011. Association for Computing Machinery.

[6] S. Chen, A. Ailamaki, P.B. Gibbons, and T.C. Mowry. Improving hash join performance through prefetching. In *Proceedings. 20th International Conference on Data Engineering*, pages 116–127, 2004.

[7] 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. *Proc. VLDB Endow.*, 1(2):1313–1324, aug 2008.

[8] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. Hyrise re-engineered: An extensible database system for research in relational in-memory data management. In Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi, editors, *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 313–324. OpenProceedings.org, 2019.

[9] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 189–198, 2007.

[10] 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. *Proc. VLDB Endow.*, 2(2):1378–1389, aug 2009.

[11] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org, 2020.

[12] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alpha-

sort: a risc machine sort, 1994.

[13] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path - parallel merging made simple. pages 1611–1618, 05 2012.

[14] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. Get real: How benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems*, DBTest '18, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Alex Watkins and Oded Green. A fast and simple approach to merge and merge sort using wide vector instructions. 11 2018.

[16] Zekun Yin, Tianyu Zhang, André Müller, Hui Liu, Yanjie Wei, Bertil Schmidt, and Weiguo Liu. Efficient parallel sort on avx-512-based multi-core and many-core architectures. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 168–176, 2019.