Hasso Plattner Institute
Chair for Data Engineering Systems

Proposal Master's 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
Martin Boissier

1 Motivation

Historically, many systems were designed for computer systems and architectures where I/O dominates performance. However, modern systems with multi-core architectures, larger DRAM capacities, advanced instruction sets, and other hardware accelerants like vector operations (SIMD), which allow us to simultaneously perform the same operation on multiple data items, have significantly altered this landscape. Hence, we must reconsider how we implement modern systems.

In-memory database systems, which store data directly in the main memory rather than on disk, provide significantly faster data access and processing due to reduced latency. Therefore, they are no longer I/O bound and need high intra-operator parallelism. It is also crucial to use other hardware features, such as cache locality and SIMD instructions, for higher data parallelism.

In this thesis, we want to look at one common database operator: the join. The join operator is a fundamental component of a 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. [12] projected that sort-merge join would outperform hash-based alternatives with 512-bit SIMD. Albutiu et al. [1] reinforced this claim with recent results reporting that their 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.

Despite ongoing research, public implementations of join algorithms optimized for modern hardware are hard to find. Most existing implementations are proprietary or experimental[1], 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 would serve as a valuable baseline for researchers looking to evaluate or improve existing methods. It would also contribute 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 [5], we will restrict ourselves to an equi-join implementation and then optionally extend upon this. We will examine the join in isolation without considering the implications of the sorted results.

The goals of this thesis include an open-source implementation of a sort-merge

[1]Implementation by Balkesen et al.: `https://archive-systems.ethz.ch/node/334`

join algorithm integrated into the Hyrise in-memory database. The sorting should be accelerated through SIMD parallelism and support at least AVX2 and AVX-512. Possible input data types are int, float, and string data. We also want to experiment with other architectures like Arm and Power and run benchmarks to evaluate our approach.

2.1 Hyrise Integration

While some public implementations exist for modern and optimized sort-merge joins, 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 [9], 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 "pdqsort" (Boost C++ library) 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.

2.2 SIMD Sorting

Most SIMD sorting algorithms presented in the literature use sorting keys of only 32 bits. We must additionally track the row ID (rid) corresponding to the sorting key for a join, resulting in 64-bit elements. Most architectures only support 8-, 16-, 32-, and 64-bit elements for SIMD. Therefore, larger elements like 128-bit are usually not supported.

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 AVX-512. Therefore, in the scope of this thesis, we want to integrate support for modern AVX-512 sorting algorithms [20, 21].

2.3 Architectures

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).

Hence, in addition to x86 systems, we optinally want to experiment with AWS Graviton 4 or other ARM chips and Power10. This requires adapting the implementation to use the respective architecture's vector extension.

2.4 Benchmarking

Complete integration into an in-memory database allows us to run decision support benchmarks like TCP-H, TCP-DS, and the Join Order Benchmark (JOB) [13] 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 [19]. For instance, TCP-H only uses integer values for keys.

However, many business intelligence applications use strings for various data types, e.g., to deal with dirty data that is not parsable. Hence, we frequently encounter string-type join keys. For that, we chose JOB due to its real-world data and ease of use, as it has already been integrated into Hyrise. The Public BI benchmark[2] also provides a realistic setting but is more difficult to use, and we only consider it optional.

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.

3 Approach

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

We chose vectorized mergesort over vectorized quicksort because of the extensive foundational work in the literature. Some SIMD quicksort implementations vectorize only the partitioning step [10], and some newer implementations also use sorting networks using AVX-512 [6]. They often share similarities when compared to mergesort, such as sorting networks. While studying the differences between vectorized quicksort and merge sort is interesting, we will limit ourselves to mergesort due to the limited time scope.

3.1 SIMD Data Format

Before we can sort our input relations, the join attribute values need to be translated into a SIMD sortable format. Usually, a 64-bit pair (key, rid) is assumed. We, therefore, support a maximum relation size of 2^{32}. With most value types greater than 32 bits, e.g., strings, we need to compress the values of the join columns. Methods like key-prefix [16] and XOR- and shift-based hash functions [7] have been used to generate keys of 32 bits.

[2] `https://github.com/cwida/public_bi_benchmark`

As strings are variable in size, it makes sense to consider their internal representation and encoding. Certain string representations allow for cheap access to a prefix or short string. For instance, Umbra's string representation [15] consists of a 128-bit struct and is 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.

3.2 Partitioning

To make use of the multi-core architecture, the unsorted column data of both join relations are partitioned into chunks and distributed among the available threads. One option for partitioning is to divide the unsorted inputs into thread count many contiguous sections of equal size. Using this strategy, occurrences of the same value can happen in multiple partitions, always requiring a global merge over all partitions (even if global sorting is not required). Another option is range partitioning, e.g., through radix partitioning. We can often improve radix-partitioning through software-managed buffers and non-temporal streaming stores (for x86 provided by the AVX instruction set) [18]. For architectures with non-uniform memory access (NUMA), it can make sense to distribute equally over NUMA regions and let threads read from their closest NUMA region. However, for this thesis, we will keep NUMA optimizations optional.

3.3 Sorting and Merging

The sort and merge step aims to sort both input relations based on the join attribute. Usually, we aim for global sorting, but there might be cases where partial sorting is enough, e.g., inside a range partition.

3.3.1 SIMD Sort- and Merge-Networks

To efficiently utilize SIMD, we need an algorithm that aligns with the SIMD execution model. Sorting and merging networks meet these criteria. For a network of input size n, n items enter the network from left to right through a series of comparators that emit the larger value to the top and the smaller value to the bottom. We can implement such a network with only min/max operators. The input is in arbitrary order for a sorting network, leaving the network sorted. For a merging network, the input comprises two sorted halves that merge into one sorted list of size n. There are two standard merging network types: bitonic merge networks and odd-even merge networks [8].

We can replace each input item with a SIMD vector and use vectorized min/max operations to take advantage of SIMD. The input items are sorted across SIMD

registers, therefore requiring additional transposition. Due to the merge networks' poor scaling [14], we can use small merging networks within a merging algorithm for larger lists [11].

3.3.2 Merging Higher Levels

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, parallelizing becomes more challenging. However, parallelization is still possible through algorithms like Merge Path [17]. 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 merge these segments in parallel. In the later stages, out-of-cache merging becomes necessary, potentially creating a bottleneck through memory bandwidth limitations. Therefore, multi-way merging [2, 8] solves this problem by saving roundtrips to memory. It implements a merge-tree, with each node being a two-way merge unit. The nodes are connected via FIFO queues. Only the leave nodes directly access memory. By pausing and starting the execution of nodes, we always ensure that all combined FIFO queues still fit into the cache. By doing so, we step-by-step propagate buffers of merged data through the merge tree until all data is merged. Optionally, we could explore merging through other primitives, such as tournament trees or priority queues.

3.4 Joining

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 compression can result in false positives in the merging step, we might require additional validation and filtering. Further parallelization of this final join step is also possible.

4 Related Work

Several papers describe how SIMD-accelerated sorting can be done efficiently on modern multi-core architectures. Chhugani et al. [8] describe the concepts needed for efficient SIMD sorting for both single- and multi-core execution. MergePath [17] presents an algorithm for merging a few very large sublists in parallel. Kim et al. [12] implemented a sort-merge join using SSE intrinsics, projecting performance for wider SIMD widths that would outperform hash joins. Albutiu et al. [1] present MPSM, a sort-merge join implementation without SIMD, concluding that their sort-merge join implementation is faster than the hash join implementation of Blanas et al. [4]. Balkesen et al. [2] experimentally studied the performance of sort-merge and radix-hash join. They claim to provide the fastest in-memory join algorithms using

sorting and hashing. Still, they conclude that the radix-hash join exceeds the sort-merge join for 256-bit SIMD. They also claim their parallel radix-hash join is the most efficient hash-join implementation yet [3]. The hash join operator implemented in Hyrise is also based on [3] and [2].

None of the papers mentioned above take advantage of 512-bit SIMD and only target x86 platforms. There is research on SIMD sorting using AVX-512 [20, 21]. However, to the best of our knowledge, no literature exists on implementing sort-merge join with the same optimizations and concepts like multiway merging for AVX-512.

5 Project Plan

Table 1: Planned Time Table

Time	**Writing/Research**	**Prototype**
Oct - Nov	– Setup for different architectures – SIMD sorting building blocks – Single-threaded SIMD sorting – Adapt to all architectures	– Setup for x86, IBM Power, AWS Graviton. – Implement sorting- and bitonic-merge networks on different architectures. – Implement single-threaded SIMD sorting (for 64-bit keys and starting with 256-bit support)
Nov - Jan	– AVX-512 specific sorting – Multiway-Merging – Hyrise integration	– Scale up sorting- and bitonic-merge networks to wider bit width. – Explore further possible improvements through AVX-512 specific instructions. – Implement Multiway Merging (Explore alternatives: tournament-trees, priority-queue). – Working sort-merge-join implementation for integer/float types on different architectures.

	– SIMD sorting of string types	– Implement prefix- and hash-based key compressions (real string data from Public BI benchmark).
Jan-Feb	– Benchmarking & Evaluation	– Parameter tuning for different architectures. – Compare to hash-join and old sort-merge join (Hyrise). – Run TCP-H, TCP-DS and Public BI benchmark.
Feb-Mar	– Thesis Writing & Evaluation	– Draft of master thesis
Mar-Apr	– Thesis Writing & Evaluation	– Finished master thesis

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, 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, 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 Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373. IEEE Computer Society, 2013.

[4] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48. ACM, 2011.

[5] Spyros Blanas and Jignesh M. Patel. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 19:1–19:16. ACM, 2013.

[6] Bérenger Bramas. A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake. *International Journal of Advanced Computer Science and Applications*, 8, 11 2017.

[7] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 116–127. IEEE Computer Society, 2004.

[8] 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, 2008.

[9] 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.

[10] Shay Gueron and Vlad Krasnov. Fast quicksort implementation using AVX instructions. *Comput. J.*, 59(1):83–90, 2016.

[11] 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 Architectures and Compilation Techniques (PACT*

2007), Brasov, Romania, September 15-19, 2007, pages 189–198. IEEE Computer Society, 2007.

[12] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009.

[13] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[14] René Müller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *VLDB J.*, 21(1):1–23, 2012.

[15] 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.

[16] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. Alphasort: A RISC machine sort. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, pages 233–242. ACM Press, 1994.

[17] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path - parallel merging made simple. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 1611–1618. IEEE Computer Society, 2012.

[18] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the surprising difficulty of simple things: the case of radix partitioning. *Proc. VLDB Endow.*, 8(9):934–937, 2015.

[19] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. Get real: How benchmarks fail to represent the real world. In Alexander Böhm and Tilmann Rabl, editors, *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 1:1–1:6. ACM, 2018.

[20] Alex Watkins and Oded Green. A fast and simple approach to merge and merge sort using wide vector instructions. In *8th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3@SC 2018, Dallas, TX, USA, November 12, 2018*, pages 37–44. IEEE, 2018.

[21] 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 Zheng Xiao, Laurence T. Yang, Pavan Balaji, Tao Li, Keqin Li, and Albert Y. Zomaya, editors, *21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10-12, 2019*, pages 168–176. IEEE, 2019.