Efficient External Sorting in DuckDB

Laurens Kuiper, Mark Raasveldt and Hannes Mühleisen

CWI, Amsterdam

Abstract

Interactive data analysis is often conveniently done on personal computers that have limited memory. Current analytical data management systems rely almost exclusively on main memory for computation. When the data size exceeds the memory limit, many systems cannot complete queries or resort to an external execution strategy that assumes a high I/O cost. These strategies are often much slower than the in-memory strategy. However, I/O cost has gone down: Most modern laptops have fast NVMe storage. We believe that the difference between in-memory and external does not have to be this big. We implement a parallel external sorting operator in DuckDB that demonstrates this. Experimental results with our implementation show that even when the data size far exceeds the memory size, the performance loss is negligible. From this result, we conclude that it is possible to have a graceful degradation from in-memory to external sorting.

Keywords

Sorting, Parallel Sorting, In-Memory Sorting, Disk-Based External Sorting, Relational Databases

1. Introduction

It is not uncommon for database systems to have hundreds or even thousands of gigabytes of RAM at their disposal. High-performance systems such as HyPer [1], and ClickHouse [2] fully utilize the available memory and perform much better on analytical workloads than their traditional disk-based counterparts. Because these systems usually run on machines with such large memory capacities, the assumption is often that the workload fits in memory.

While laptops have also enjoyed increased memory capacity, their physical design has limited space. Therefore they typically have only 16GB of memory. Laptops are often used in interactive data analysis, with tools like Pandas [3] and dplyr [4], showing that there is a need for analytical data management technology that runs on a laptop. However, these tools operate only in memory. As a result, users cannot process datasets that are slightly larger than memory, on their own machine.

Disk-based database systems, on the other hand, have long solved the problem of processing larger-than-memory datasets. These systems are generally much slower than in-memory systems on analytical workloads. When a user wants to process a larger-than-memory dataset using an in-memory system, usually one of two things happens 1) The system throws an error stating it is out of memory, 2) The system switches to an external strategy that is much less efficient than the in-memory strategy, which results in a slow execution time, even when, for example, the input is only 10% larger than memory. Fast queries may become slow or run into an error when a table grows in size, creating a frustrating experience for users.

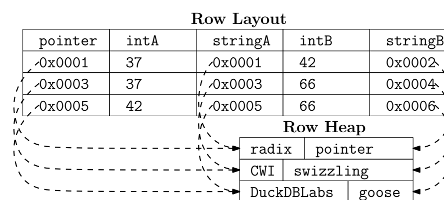

Figure 1: DuckDB's row layout and row heap.

We can mitigate this problem by implementing operators such that they optimally use the amount of available memory and only write data to disk when this is necessary. I/O quickly becomes the bottleneck on machines with low-bandwidth storage devices. However, most modern laptops have nVME storage with high write speeds, making I/O less of a limiting factor.

We have implemented a parallel, external sorting operator in DuckDB [5] that demonstrates this. Our implementation seamlessly transitions from in-memory to external sorting by storing data in buffer-managed blocks that are offloaded to disk using a least-recently-used queue, similar to LeanStore [6].

Transitioning from in-memory to disk is made possible by DuckDB's unified internal row layout, shown in Figure 1, which can be spilled to disk using *pointer swizzling* [7].

We compare our implementation against four other systems using an improvised relational sorting benchmark on two tables from TPC-DS [8]. Our implementation achieves excellent performance when data fits in memory and shows a graceful degradation in performance as we go over the limit of available memory.

BICOD'21: British International Conference on Databases, December 9–10, 2021, London, UK
laurens.kuiper@cwi.nl (L. Kuiper); raasveld@cwi.nl (M. Raasveldt); hannes.muehleisen@cwi.nl (H. Mühleisen)
© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

2. Sorting Relational Data

Sorting is one of the most well studied problems in computing science. Research in this area forms the basis of sorting in database systems but focuses mostly on sorting large arrays or key/value pairs. Sorting is more complex for relational data as many different types need to be supported, as well as NULL values. There can also be multiple order clauses. Besides sorting the order clause (key) columns, all other selected (payload) columns need to be re-ordered as well.

In 2006, Goetz Graefe surveyed sorting in database systems [9]. The most important takeaway from this survey when it comes to performance is that the cost of sorting is dominated by comparing values and re-ordering data. Anything that makes either of these two operations cheaper will have an impact on the overall speed.

There are two obvious ways to go about implementing a comparator in a column-store when we have multiple `ORDER BY` clauses:

1. Iterate through the clauses: Compare columns until we find one that is not equal, or until we have compared all columns. This comparator jumps between columns, causing random access in memory.
2. Sort the data entirely by the first clause, then sort by the second clause, but only where the first clause was equal, and so on. This approach requires multiple passes over the data.

The binary string comparison technique [10] improves sorting performance by simplifying the comparator. It encodes all columns in the ORDER BY clause into a single binary sequence that, when compared using `memcmp` will yield the correct overall sorting order. This encoding also yields the correct order with a byte-by-byte Radix sort. Although this technique has existed since the days of System R, not many systems use it today and opt for one of the ways listed above.

We implement this comparator and opt for fixed-size encodings, which can be more easily re-ordered. For variable-size types such as strings, we can therefore only encode a prefix. We compare the whole string only when prefixes are equal. The encoding is shown in Figure 2.

Not shown in the figure are NULL values and collations. NULL values are handled by prefixing each value with an additional byte denoting whether the value is NULL. Collations are handled by evaluating the collation function before encoding the prefix of the string.

The other high cost of sorting is re-ordering data. Systems that use columnar storage must re-order all selected columns, which causes a random access pattern for each. Row-based systems only have to pay the cost of this random access pattern once. When we select many columns, this becomes a considerable advantage.

(a)

birth_country	birth_year
NETHERLANDS	1992
GERMANY	1924

(b)

birth_country	birth_year
78 69 84 72 69 82 76 65 78 68 83 0	200 7 0 0
71 69 82 77 65 78 89 0	132 7 0 0

(c)

binary string
177 186 171 183 186 173 179 190 177 187 172 255 128 0 7 200
184 186 173 178 190 177 166 255 255 255 255 255 128 0 7 132

Figure 2: Binary string encoding. The original data in (a) is represented as (b) on a little-endian machine. It is encoded as (c) when ordered for a query with order clauses `c_birth_country DESC`, `c_birth_year ASC`. Descending order flips the bits. The string "GERMANY" is padded to ensure fixed size.

A few relational operators are inherently row-based, such as joins and aggregations. For vectorized execution engines, it is common practice to physically convert vectors to and from a row layout for these operators using *scatter-gather*. We argue that this layout should be used for sorting, as sorting is also a row-based operator.

We show DuckDB's row layout in Figure 1. Rows have a fixed size that can be easily re-ordered while sorting. We represent variable-sized columns with pointers into a string heap where the data resides. The heap data does not have to be re-ordered while sorting in memory. Each fixed-size row has an additional `pointer` field that points to its "heap row". This pointer is not used for in-memory sorting but is crucial for external sorting, which we will explain in section 2.2.

2.1. Parallel Sorting

DuckDB uses Morsel-Driven Parallelism [11], a framework for parallel query execution. For the sorting operator, this means that multiple threads collect a roughly equal amount of data, in parallel, from the input table. We use this parallelism by letting each thread sort its collected data using Radix sort. After this initial sorting phase, each thread has one or more sorted blocks of data, which must be combined into the final sorted result using merge sort.

There are two main ways of implementing merge sort: K-way and Cascade merge. The K-way merge merges K lists into one sorted list in one pass and is traditionally used for external sorting because it minimizes I/O [12]. Cascade merge is used for in-memory sorting because it is more efficient than K-way merge. It merges two lists of sorted data at-a-time until only one sorted list remains.

Recent work on K-way external merge sort [13] on devices with flash memory reduces execution time by 20% to 35% compared to standard external merge sort. Salah et al. [14]show that K-way merge can achieve better performance than cascaded merge when it comes to in-place sorting. This work on K-way merging may seem

attractive for our implementation, but cascaded merge is still a more attractive option when it comes to in-memory sorting. We also need to deal with variable-size data, which complicates in-place sorting. We do not want to compromise in-memory sorting and choose cascade merge.

Cascade merge is embarrassingly parallel when there are many more sorted blocks than available threads. As the blocks get merged, there will not be enough blocks to keep all threads busy. In the final round, when we merge two blocks to create the final sorted result, there is no parallelism: One thread processes all the data. We parallelize this using Merge Path [15]. Merge path pre-computes where blocks intersect, creating partitions that can merge independently of each other. Binary search efficiently computes these partitions.

2.2. External Sorting

To sort more data than fits in memory, we write blocks of sorted data to disk. Rather than actively doing this in our sorting implementation, DuckDB's buffer manager decides when to do this: When memory is full.

Writing data to disk is trivial for fixed-size types but non-trivial for variable-size types, as the pointers in our row layout will be invalidated. To be able to offload variable-sized types as well, we use *pointer swizzling*. When we are sorting externally, we convert the pointers in the row layout to offsets, shown in Figure 3.

Row Layout

offset	intA	stringA	intB	stringB
0	37	0	42	5
12	37	0	66	3
24	42	0	66	10

Row Heap

radix	pointer
CWI	swizzling
DuckDBLabs	goose

Figure 3: DuckDB's internal row layout (swizzled).

We replace the 8-byte pointer field with an 8-byte offset, which denotes where the strings of this row reside in the heap block. We also replace the pointers to the string values within the row with an 8-byte *relative* offset. This offset denotes how far this particular string is located from the start of this row's *heap row*. Using relative offsets within rows rather than absolute offsets is very useful during sorting: These relative offsets stay constant and do not need to be updated when we copy the row.

With this dual-purpose row-wise representation, we achieve an almost seamless transition between in-memory and external sorting. The only difference between the two is swizzling and unswizzling each pointer once and re-ordering the heap during sorting.

3. Evaluation

In this section, we evaluate DuckDB's sorting implementation. We compare against ClickHouse [2], HyPer [1], Pandas [3] and SQLite [16]. HyPer and ClickHouse are full-blown analytical database systems that focus on parallel in-memory computation. Pandas is single-threaded and in-memory, and SQLite is a more traditional (single-threaded) disk-based system.

To demonstrate how the systems perform in an environment with limited RAM, we run our experiments on a 2020 MacBook Pro. It has 16GB of memory and a fast SSD with a write speed of over 3GB/s. HyPer does not yet run natively on ARM CPUs, so we emulate it using Rosetta 2. See our blog [17] for an experiment on x86 CPU architecture.

Benchmarking sorting in database systems is not straightforward. We would like to measure only the time it takes to sort, with as little noise as possible. Therefore, we cannot use `SELECT` queries, as the client-server protocol will quickly dominate query runtime [18].

To approach a fair comparison, we measure the end-to-end time of queries that sort the data and write the result to a temporary table, i.e., `CREATE TEMPORARY TABLE output AS SELECT ... FROM ... ORDER BY ...;`. For Pandas we will use `sort_values` with `inplace=False` to mimic this query. To measure stable end-to-end query time, we run each query 5 times and report the median run time. Scripts for our experiments are available on GitHub[1].

We have created a relational sorting benchmark on the `customer` and `catalog_sales` tables from TPC-DS [8]. The row counts at different scale factors are shown in Table 1.

scale factor	catalog_sales	customer
10	14.401.261	500.000
100	143.997.065	2.000.000
300	260.014.080	5.000.000

Table 1
TPC-DS table row count for `customer` and `catalog_sales` at scale factor 10, 100, and 300.

TPC-DS tables are challenging for sorting implementations because they are wide (many columns, unlike the tables in TPC-H) and have a mix of fixed- and variable-sized types. `catalog_sales` has 34 columns, all fixed-size types: integer and double. `customer` has 18 columns, fixed- and variable-size: 10 integers, 8 strings.

We sort the `catalog_sales` table on `cs_quantity` and `cs_item_sk`, and select an increasing number of payload columns. This experiment tests both the system's ability to sort and re-order the payload. We show the results of this experiment in Figure 4.

[1]https://github.com/lnkuiper/experiments/tree/master/sorting

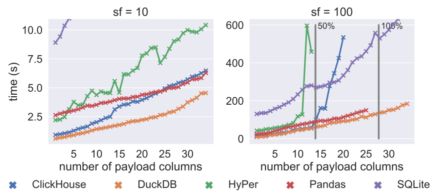

Figure 4: `catalog_sales` ordered by `cs_quantity, cs_item_sk`, with an increasing number of payload columns. These columns have few unique values, creating a difficult challenge for sorting algorithms. The grey vertical lines in the SF100 plot indicate at which points the payload columns take up 50% and 100% of the amount of available memory.

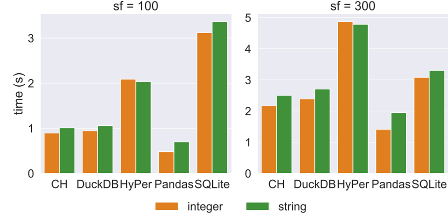

Figure 5: `customer` ordered by different column types: Integer columns (`c_birth_year, c_birth_month, c_birth_day`), and string columns (`c_first_name, c_last_name`).

The table fits in memory at SF10, and the systems' performances are in the same ballpark, except for SQLite, and DuckDB is the clear winner. At SF100, around 14 selected payload columns, the input table takes up 50% of memory. It is common for systems to not sort data in place, but copy it to a new location, which requires double the amount of memory. The figure clearly shows this: All systems except DuckDB and SQLite run into an error due to running out of memory and are unable to complete the benchmark.

ClickHouse switches to an external sorting strategy, which is much slower than its in-memory strategy. Therefore, adding a few payload columns results in a runtime that is orders of magnitude higher. Despite switching strategy, ClickHouse runs into an out-of-memory error. HyPer uses the `mmap` system call, which creates a mapping between a block of memory and a file, which allows HyPer to continue for a while when the data no longer fits in memory, before running into an error as well. As we can see, the runtime becomes very slow before HyPer runs out of memory: Random memory access becomes random disk access.

Surprisingly, Pandas can load the dataset at SF100 because macOS dynamically increases swap size. Most operating systems do not do this, and Pandas will not load the dataset at all. Pandas relies on NumPy's [19] single-threaded quicksort implementation. Pandas shows impressive performance, partly because it already has the input data fully materialized in memory and does not have to stream data through an execution pipeline from the input to the output table like most DBMSes.

Meanwhile, DuckDB and SQLite do not show a visible difference in performance when where data no longer fits in memory. SQLite always opts for a traditional external sorting strategy, resulting in a robust, but overall slower performance than DuckDB.

In our next benchmark, on the `customer` table, we test how well the systems can sort by strings and by integers. Both comparing and re-ordering strings are much more expensive than comparing and re-ordering numeric types. We show the result of this experiment in Figure 5.

As expected, sorting by strings is slower than sorting by integers for most systems. In this experiment, the payload also includes string columns. Pandas has an advantage here because it already has the strings in memory, and most likely only needs to re-order pointers to these strings. The database systems need to copy strings twice: Once when reading the input table, and again when creating the output table. Profiling in DuckDB reveals that the actual sorting takes less than a second at SF300, and most time is spent on (de)serializing strings. See [17] for more details on the difference between integers and strings.

4. Conclusion and Future Work

In this paper, we presented our parallel external sorting implementation in DuckDB. We compared it against four data management systems using a relational sorting benchmark based on TPC-DS. Three of the four systems perform well in memory, but crash as the data goes over the amount of available memory. DuckDB is the only system under benchmark that performs well both in memory and external. These results demonstrate that it is possible to implement a sorting operator that is efficient in memory and has a graceful degradation in performance as the input size exceeds the memory limit.

4.1. Future Work

It is unclear how each technique contributed to end-to-end performance. Quantifying these contributions e.g. through simulation is an area of future research.

Our sorting implementation uses a row layout that can be offloaded to storage using pointer swizzling. Other blocking operators could benefit from this layout. For instance, join, aggregation and window. Incorporating this layout would enable external computation for these operators as well. Implementing these operators such that their performance degrades gracefully as the input size exceeds the memory limit is an area of future research.

References

[1] A. Kemper, T. Neumann, HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots, in: S. Abiteboul, K. Böhm, C. Koch, K. Tan (Eds.), Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, IEEE Computer Society, 2011, pp. 195–206. URL: https://doi.org/10.1109/ICDE.2011.5767867. doi:10.1109/ICDE.2011.5767867.

[2] B. Imasheva, A. Nakispekov, A. Sidelkovskaya, A. Sidelkovskiy, The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse, in: H. A. L. Thi, H. M. Le, T. P. Dinh (Eds.), Optimization of Complex Systems: Theory, Models, Algorithms and Applications, WCGO 2019, World Congress on Global Optimization, Metz, France, 8-10 July, 2019, volume 991 of *Advances in Intelligent Systems and Computing*, Springer, 2019, pp. 820–828. URL: https://doi.org/10.1007/978-3-030-21803-4_82. doi:10.1007/978-3-030-21803-4_82.

[3] W. McKinney, et al., Data structures for statistical computing in Python, in: Proceedings of the 9th Python in Science Conference, volume 445, Austin, TX, 2010, pp. 51–56.

[4] H. Wickham, R. François, L. Henry, K. Müller, dplyr: A Grammar of Data Manipulation, 2021. URL: https://CRAN.R-project.org/package=dplyr, r package version 1.0.7.

[5] M. Raasveldt, H. Mühleisen, DuckDB: An Embeddable Analytical Database, in: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1981–1984. URL: https://doi.org/10.1145/3299869.3320212. doi:10.1145/3299869.3320212.

[6] V. Leis, M. Haubenschild, A. Kemper, T. Neumann, LeanStore: In-Memory Data Management beyond Main Memory, in: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018, IEEE Computer Society, 2018, pp. 185–196. URL: https://doi.org/10.1109/ICDE.2018.00026. doi:10.1109/ICDE.2018.00026.

[7] A. Kemper, D. Kossmann, Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis, VLDB J. 4 (1995) 519–566. URL: http://www.vldb.org/journal/VLDBJ4/P519.pdf.

[8] M. Poess, B. Smith, L. Kollar, P. Larson, TPC-DS, Taking Decision Support Benchmarking to the next Level, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, Association for Computing Machinery, New York, NY, USA, 2002, p. 582–587. URL: https://doi.org/10.1145/564691.564759. doi:10.1145/564691.564759.

[9] G. Graefe, Implementing Sorting in Database Systems, ACM Comput. Surv. 38 (2006) 10–es. URL: https://doi.org/10.1145/1132960.1132964. doi:10.1145/1132960.1132964.

[10] M. W. Blasgen, R. G. Casey, K. P. Eswaran, An Encoding Method for Multifield Sorting and Indexing, Commun. ACM 20 (1977) 874–878. URL: https://doi.org/10.1145/359863.359892. doi:10.1145/359863.359892.

[11] V. Leis, P. Boncz, A. Kemper, T. Neumann, Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 743–754. URL: https://doi.org/10.1145/2588555.2610507. doi:10.1145/2588555.2610507.

[12] D. Knuth, The Art Of Computer Programming, vol. 3: Sorting And Searching, Addison-Wesley, 1973.

[13] R. Jackson, J. Gresl, R. Lawrence, Efficient External Sorting for Memory-Constrained Embedded Devices with Flash Memory, ACM Trans. Embed. Comput. Syst. 20 (2021). URL: https://doi.org/10.1145/3446976. doi:10.1145/3446976.

[14] A. Salah, K. Li, Q. Liao, M. Hashem, Z. Li, A. T. Chronopoulos, A. Y. Zomaya, A Time-Space Efficient Algorithm for Parallel k-Way In-Place Merging Based on Sequence Partitioning and Perfect Shuffle, ACM Trans. Parallel Comput. 7 (2020). URL: https://doi.org/10.1145/3391443. doi:10.1145/3391443.

[15] O. Green, S. Odeh, Y. Birk, Merge Path - A Visually Intuitive Approach to Parallel Merging, CoRR abs/1406.2628 (2014). URL: http://arxiv.org/abs/1406.2628. arXiv:1406.2628.

[16] R. D. Hipp, SQLite, 2021. URL: https://www.sqlite.org/index.html.

[17] L. Kuiper, Fastest table sort in the West - Redesigning DuckDB's sort, 2021. URL: https://duckdb.org/2021/08/27/external-sorting.html.

[18] M. Raasveldt, H. Mühleisen, Don't Hold My Data Hostage: A Case for Client Protocol Redesign, Proc. VLDB Endow. 10 (2017) 1022–1033. URL: https://doi.org/10.14778/3115404.3115408. doi:10.14778/3115404.3115408.

[19] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming

with NumPy, Nature 585 (2020) 357–362. URL: https://doi.org/10.1038/s41586-020-2649-2. doi:10.1038/s41586-020-2649-2.