These Rows Are Made for Sorting and That's Just What We'll Do

Laurens Kuiper
CWI, Amsterdam, Netherlands
laurens.kuiper@cwi.nl

Hannes Mühleisen
CWI, Amsterdam, Netherlands
hannes.muehleisen@cwi.nl

***Abstract*—Sorting is one of the most well-studied problems in computer science and a vital operation for relational database systems. Despite this, little research has been published on implementing an efficient relational sorting operator. In this work, we aim to fill this gap. We use micro-benchmarks to explore how to sort relational data efficiently for analytical database systems, taking into account different query execution engines as well as row and columnar data formats. We show that, regardless of architectural differences between query engines, sorting rows is almost always more efficient than sorting columnar data, even if this requires converting the data from columns to rows and back. Sorting rows efficiently is challenging for systems with an interpreted execution engine, as interpreting rows at runtime causes overhead. We show that this overhead can be overcome with several existing techniques. Based on our findings, we implement a highly optimized row-based sorting approach in the DuckDB open-source in-process analytical database management system, which has a vectorized interpreted query engine. We compare DuckDB with four analytical database systems and find that DuckDB's sort implementation outperforms query engines that sort using a columnar data format, and matches or outperforms compiled query engines that sort using a row data format.**

***Index Terms*—relational databases, database query processing, sorting**

I. Introduction

Sorting is one of the most well-studied problems in computer science. Research on efficient sorting algorithms focuses on crucial issues such as cache-efficiency [1], reducing branch mispredictions [2], parallelism [3], and worst-case patterns [4], but is mostly limited to sorting large arrays of integers. Database systems research focuses on many of the same issues [5]–[7], and database systems have some of the most practical use-cases of sorting: The `ORDER BY` and `WINDOW` operators explicitly invoke sorting, but other operations such as building an index, merge joins, and inequality joins [8] may implicitly rely on sorting. Therefore, it is crucial to have an efficient sort implementation to provide fast query response times, especially for analytical (OLAP) data management systems. However, sorting relational data is more involved than sorting an array of integers, and very little research has gone into sorting relational data efficiently, especially compared to the amount of research on other operators, such as joins [9].

Although relational sorting implementations can reap the benefits from the sorting algorithms that research has produced [2]–[4], merely using such an algorithm will not make a relational sorting implementation efficient by default, as we will demonstrate in this paper. This is because the cost of sorting is dominated by two operations: Comparing and moving values [10]. Comparing and moving relational data is not as straightforward as comparing and moving integers and is more costly. These two operations must be implemented as efficiently as possible. The comparison function that arises from the `ORDER BY` clause can be arbitrarily complex and contain any of the types that the system supports.

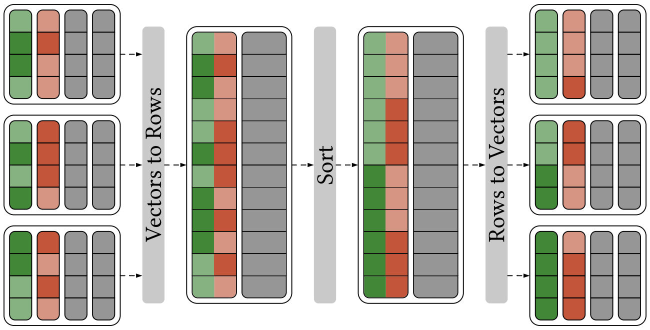

Fig. 1. Converting vectors to rows, sorting them, and converting them back to vectors. The columns that appear in the order clause are colored, and the other selected columns are in gray.

A system's query execution engine affects how these operations can be implemented, as well as the efficiency of these implementations. The engines of most modern OLAP systems are based on either vectorization, pioneered by VectorWise [5], or data-centric code-generation, pioneered by HyPer [11]. The strengths and weaknesses of these two processing paradigms have been researched in-depth [12], [13], but not in the context of sorting. This work investigates how to sort relational data efficiently under both paradigms. We find that sorting rows is almost always more efficient than sorting columnar data, even if this requires converting from a columnar to a row format and back, as illustrated for a vectorized engine in Figure 1.

To compare the differences between sorting a columnar and row data format, as well as the differences between sorting in vectorized and compiled query engines, we could compare the end-to-end runtime of database systems that implement these approaches. However, with full-fledged systems, it is challenging to create an apples-to-apples comparison, as these systems differ in many aspects unrelated to sorting. Therefore, to isolate the fundamental differences between these approaches, we implement a relational sorting micro-benchmark. Our experimental results show that compiled query engines can efficiently sort row data, while vectorized interpreted engines are hindered by interpretation and function call overhead. We demonstrate that this overhead can be overcome by implementing several existing techniques.

Finally, we implement a row-based sort that includes these techniques within DuckDB, our in-process OLAP database management system. DuckDB provides SQL, columnar storage and uses a vectorized interpreted execution engine. We compare our implementation with four high-performance analytical database systems: ClickHouse [14], MonetDB [15], HyPer [11], and Umbra [16]. The results of this comparison show that DuckDB's interpreted row-based sort implementation outperforms ClickHouse's and MonetDB's sort implementations, which use a columnar data format, and matches or outperforms HyPer's and Umbra's compiled row-based sort implementations.

The remainder of this paper is organized as follows. In Section II, we discuss the challenges of implementing relational sorting. After describing our methodology in Section III, we evaluate several approaches to sorting relational data in row and columnar format with micro-benchmarks in Section IV. Then, in Section V, we discuss how the query execution engine affects sorting performance. Section VI discusses and evaluates techniques for sorting in vectorized interpreted query engines. In Section VII, we describe DuckDB's sort implementation and evaluate it in an end-to-end benchmark. We summarize and draw our conclusions in Section VIII, and discuss future work in Section IX.

II. Sorting Relational Data

Database systems use sorting for many purposes [10]. Sorting can be invoked explicitly by specifying a sort order at the end of a query or a window specification, but also implicitly for many purposes such as join algorithms [8], improving run-length encoding compression [17], and zone maps [18]. Because it is so widely useful, any database system needs an efficient sorting implementation, especially OLAP systems that aim to have low query response times. Sorting relational data efficiently, however, is not as straightforward as sorting an array of integers.

Whether sorting is invoked explicitly in an ORDER BY clause or implicitly by other means, a relational sorting implementation must be able to sort all selected data by one or more predicates. Take the following query, for example:

```sql
SELECT * FROM customer
ORDER BY c_birth_country DESC NULLS LAST,
         c_birth_year ASC NULLS FIRST;
```

This query specifies which columns to return and how they should be sorted, i.e., it describes how rows should be compared to produce the ordered query result. All columns from the customer table should be returned, sorted by the column c_birth_country in descending order, with NULL values coming last, and where rows have the same value in that column, they should be sorted by c_birth_year in ascending order with NULL values coming first. We will refer to the columns that appear in the ORDER BY clause as *key* columns, and all other selected columns as *payload* columns.

As we can see for the example query, describing how rows should be compared is already complex, even though we specify only two key columns. In contrast, comparing integers is done using simply the '<'-operator. Naively comparing rows may result in a complex comparator with many branches. Because the comparator is so frequently used during sorting, it should be implemented as efficiently as possible. For example, eliminating branches from the comparison function speeds up quicksort significantly [2].

How a relational sort implementation physically represents data in memory also affects comparison efficiency. If the keys belonging to a single row are not stored consecutively in memory, which is the case for a columnar data format, comparing tuples causes random access for each compared value, which may cause cache misses. Improving cache locality speeds up many sorting algorithms significantly [1].

Besides determining the order in which to return the tuples by comparing and sorting them, the data must also be physically re-ordered. For an array of integers, this is trivial: We sort the array with an efficient sorting algorithm, which results in the data being in the correct order. For relational data, it is not as obvious because there are many ways to represent tuples. Sorting is inherently a row-wise operation, but systems with a vectorized engine use a columnar data format. For such systems, it might be beneficial to convert the data to a row format, also called the N-ary Storage Model (NSM), and then convert it back to a columnar format, also called the Decomposition Storage Model (DSM), after completing the sort. Vectorized engines already do this for hash tables in joins and aggregations [19].

Additionally, we can deal with the key and payload columns separately: We can retrieve the payload in the correct order after sorting the key columns. The question is then whether it is efficient to convert either, neither, or both to NSM. This is essentially a trade-off between cache locality and data movement. Converting from DSM to NSM results in a better memory access pattern when retrieving the payload in the correct order because values that belong together are close together in memory. This conversion, however, requires moving all the data from the columnar format to the row format. For the payload columns, the random access that is incurred when retrieving tuples in the correct order can be improved [20]. While payload handling is an essential part of relational sorting, it is outside the scope of this paper. In this work, we focus on sorting key columns.

Without benchmarking different sorting implementations, it is unclear what the most efficient approach will be for a given system. Differences in execution engine, hardware characteristics like CPU cache size, and functionality requirements all affect how to approach relational sorting. However, implementing a relational operator such as the sort operator in a database system is cumbersome, let alone implementing multiple variations to determine the best-performing one. Therefore, we use micro-benchmarks to evaluate several approaches for sorting relational data to determine their effectiveness.

III. Methodology

To isolate the fundamental properties of sorting data in row and columnar format, as well as the fundamental properties of sorting in compiled and vectorized interpreted query execution engines, we implement micro-benchmarks. By using micro-benchmarks, we do not measure the time it takes to parse and optimize queries and the overhead of result set serialization [21], which may differ across database systems. We have implemented several approaches to sorting relational data using in C++. All of the approaches use `std::sort`, an introspective sort [22] implementation. The `std::sort` algorithm is not the state-of-the-art, as it can be improved in various ways [2], [4]. This is not a problem, as we wish to measure the effect of different data formats, tuple comparison methods, and execution engine. We can directly compare this as long as we use the same sorting algorithm.

A. Workload

The data for our micro-benchmarks consists of columns of unsigned 32-bit integers. These are generated by sampling from three distributions[1]:

Random Random uniform

Unique128 Random uniform with 128 unique values

PowerLaw Power-law with 128 unique values and a distribution power of 5

We vary the number of key columns from 1 to 4, as well as the number of rows from 2^{10} to 2^{24}.

We choose these three data distributions because we want to know how the sorting approaches perform on different data distributions with varying degrees of skewness. The Random data distribution has virtually no duplicate values in each column. The Unique128 distribution has many duplicates in each column, with each unique value occurring at approximately the same rate. The PowerLaw distribution has the same number of unique values as the Unique128 distribution, but the groups of duplicates all have different sizes, and the group size is skewed by a power-law distribution. Having duplicate values in the key columns leads to ties when comparing values during sorting, which requires the comparison function to compare the next key column as well.

This is an oversimplification of real-world relational data, which can have many different data distributions and types. It is however, already more complex than sorting a single array of integers, as we will consider multiple key columns, the existence of a payload, row and columnar data formats, and different execution engines. As we will demonstrate in the following, memory access patterns, branch mispredictions, and interpretation overhead are the main differentiating factors in the performance of sorting implementations. The memory access patterns of sorting a row and columnar data format that we measure in our micro-benchmark are the same regardless of data type. The same holds for branch mispredictions and interpretation overhead. Although we purposely keep our micro-benchmarks simple, we include experiments with other data types in our end-to-end benchmarks in Section VII.

B. Experimental Setup

To measure the runtime of our micro-benchmarks, we use the `m5d.8xlarge` AWS EC2 instance unless noted otherwise. This instance has an Intel Xeon Platinum 8259CL CPU with 16 cores (32 threads), 128GB of RAM, and NVMe storage. We use Ubuntu 20 as OS and compile our code with Clang 12. We repeat each experiment five times and report only the median runtime.

We measure CPU performance counters to further analyze and understand the properties of different approaches. These counters are obtained using Xcode command-line tools on a 2020 MacBook Pro with an M1 ARM CPU, 16GB of RAM, and NVMe storage. On this machine, we compile our code with AppleClang 13. We use this machine because we cannot measure performance counters on the `m5d.8xlarge` instance type due to restrictions of the virtual machine. Note that this machine has a different CPU architecture and a larger L1 cache size than the AWS instance. We run the performance counter measurements just once. Detailed specifications of all hardware used in our experiments can be found in Table I.

TABLE I
Specifications of hardware used in experiments.

	`m5d.8xlarge`	`m6gd.8xlarge`	MacBook
CPU Brand	Intel	AWS	Apple
CPU Model	8259CL	Graviton2	M1
Architecture	x86	ARM	ARM
Cores/Threads	16/32	32/32	8/8
Clock Rate	2.5-3.5 GHz	2.5 GHz	3.2 GHz
L1 Cache	32 KB	64 KB	128 KB

IV. DSM vs. NSM

In this section, we experimentally evaluate the efficiency and performance characteristics of sorting relational data in columnar and row format. We sort only the key columns because we can retrieve the payload in the correct order after sorting the keys to create a fully sorted run. Note that to collect the payload, we need some way of tracking which keys correspond to which payload, e.g., using a pointer or a row ID. Furthermore, we assume that all input has been materialized, i.e., the sort operator has collected all input data already in a row or columnar format, such that we can further isolate raw sorting performance.

There are two obvious ways of comparing tuples of relational data when there are multiple key columns:

1) Iterate through the key columns with each comparison. Compare values until we find one that is not equal or until we have iterated through all columns in the order clause.

[1] The source code for our micro-benchmarks can be found at https://github.com/lnkuiper/experiments/sorting_simulation

2) Sort everything by the first key column, then identify the tuples that have an equal value in this column, and sort these by the second key column. Repeat until all key columns are sorted.

We apply these approaches to both the columnar and row data formats. We will refer to (1) as the *tuple-at-a-time* approach, and to (2) as the *subsort* approach.

A. Sorting Columnar Data

With a columnar data format, comparing tuples requires accessing their values in each column. This necessitates sorting indices rather than sorting the data directly because we need to use the indices to access the data in the columns. After sorting, the sorted indices are used to retrieve the payload in the correct order. An implementation of the *tuple-at-a-time* approach for the columnar format data using `std::sort` in C++ could look like the following:

```
// Compare two tuples using their row id
bool compare(l_id, r_id, cols) {
    size_t i = 0;
    for (; i < cols.size() - 1; i++) {
      if (cols[i][l_id] != cols[i][r_id])
        break;
    }
    return cols[i][l_id] < cols[i][r_id];
}
// Sort N row indices ’idxs’ by the values of the
     keys in ’cols’ using our ’compare’ function
std::sort(idxs, idxs + N, [&] (l_id, r_id) -> bool {
  return compare(l_id, r_id, cols);
});
```

Tuples are represented by their indices ‘`idxs`’, and compared using their respective values in ‘`cols`’. The *subsort* implementation works similarly: It also sorts by indices but has a less complex comparison function that only compares one column. Additionally, the *subsort* approach has to identify tied tuples after each pass and recurse until there are no more ties, or until all columns are done.

The *tuple-at-a-time* sorting approach for the columnar data format has three major deficiencies: 1) Comparing two tuples causes multiple random accesses with each comparison. If the tuples have the same value in the first key column, we need to compare their value in the second key column, and so forth. The more duplicate values there are, the more random access we have. Therefore, this approach suffers from data distributions with many duplicates, and skewed data distributions. 2) The comparison function has branches, namely, whether to compare the next key column or not. This may be difficult to predict, again depending on the distribution: If there are no duplicates, the branch predictor will correctly predict never to compare the next column. 3) It sorts indices rather than the data in the key columns itself, i.e., the data in the key columns never actually moves. Sorting algorithms move similar values next to each other in memory, which improves cache locality. However, by not moving the actual data, the *tuple-at-a-time* approach does not benefit from this.

The *subsort* approach improves on this by only causing random access in one column at a time, mitigating deficiency (1), and comparing a single column at a time and therefore having no branches in the comparison function, eliminating deficiency (2). Furthermore, sorting one column at a time reduces the cache pressure because a single memory region is accessed at a time, which mitigates deficiency (3) slightly.

We measure performance counters of both methods in our micro-benchmark and show the results in Table II. As expected, the *tuple-at-a-time* approach incurs more cache misses and branch mispredictions than the *subsort* approach on all data distributions, except for the Random distribution. In this data distribution, there are almost no duplicates. Therefore, the second column rarely has to be randomly accessed. For the same reason, the number of branch mispredictions on this data distribution is also similar between the two approaches because the branch predictor can almost perfectly predict to never compare the value in the second column for the *tuple-at-a-time* approach. For the other two data distributions, the *subsort* approach incurs significantly fewer cache misses and branch mispredictions.

TABLE II
L1 CACHE MISSES AND BRANCH MISPREDICTIONS (BOTH IN TRILLIONS, LOWER IS BETTER) OF SORTING 2^{24} ROWS OF 3 KEY COLUMNS IN COLUMNAR (C) DATA FORMAT, WITH THE *tuple-at-a-time* (T) AND *subsort* (S) APPROACHES.

Distribution	Cache Misses		Branch Misses	
	C/T	C/S	C/T	C/S
Random	**2.74**	2.87	1.54	**1.48**
Unique128	11.18	**6.13**	3.56	**1.54**
PowerLaw	11.37	**4.44**	2.45	**0.97**

CPU performance is affected negatively by cache misses and branch mispredictions, which should translate into a worse sorting performance. We measure the runtime of both approaches using our micro-benchmark and show the relative runtime of *subsort* compared to the *tuple-at-a-time* approach in Figure 2. A relative runtime of 2.00 means that the *subsort* approach was twice as fast as the *tuple-at-a-time* approach, i.e., it finished sorting in half the time. When there is only one key column, the approaches are virtually equal. For the Random data distribution, the relative runtime is close to 1 for all inputs, i.e., the approaches perform similarly. This is in line with our expectation because the *tuple-at-a-time* approach does not suffer as much from its deficiencies for this distribution.

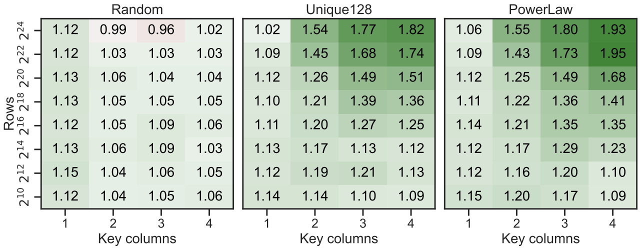

Fig. 2. Relative runtime (higher is better) of the *subsort* approach compared to the *tuple-at-a-time* approach on a columnar data format.

For the Unique128 and PowerLaw distributions, the *subsort* approach is better the more rows and the more key columns there are. For these distributions, the *tuple-at-a-time* approach performs relatively worse than the *subsort* approach because the columns no longer fit in the CPU's cache, and the increased cache pressure and branch mispredictions of the *tuple-at-a-time* approach hurt its performance. From these results, we conclude that the *subsort* approach is the best approach for data in columnar format.

B. Sorting Row Data

Unlike the columnar data format, we do not have to use row indices *while* sorting row format data. We can directly address a row rather than address it by its row index because all values that we compare are co-located. The row indices (or pointers) are still needed to retrieve the payload in the correct order after sorting. We can pack these within the row, which, for example, can be achieved with a struct in C++:

```cpp
struct OrderKey {
 // Key columns
 uint32_t col1;
 uint32_t col2;
 // Index (or pointer) to the payload
 size_t idx;
};
```

The row format data is sorted by calling, e.g., `std::sort` on an array of the `OrderKey` structs. The comparison function used to sort is then defined as a comparison function between the key column values in these structs.

Like the columnar data format, we sort the row data format with the *tuple-at-a-time* approach. We can sort rows with the *subsort* approach as well, although, at first glance, it is less intuitive to do this with rows. Like before, *subsort* simplifies the comparison function while sorting, as it does not need branches. The implementation of *subsort* for rows is the same as for columnar data: We sort, identify which tuples are tied, sort the tied tuples by the next column, and so forth.

We measure performance counters of both methods in our micro-benchmark and show the results in Table III. By comparing this with Table II, it is clear that sorting the row data format incurs fewer cache misses than sorting columnar format data by an order of magnitude. The number of branch misses is similar. Unsurprisingly, the *subsort* approach incurs fewer branch mispredictions than the *tuple-at-a-time* approach on every data distribution because there are no branches in the comparison function. The *subsort* approach incurs slightly more cache misses than *tuple-at-a-time*. This is caused by scanning the data for tied tuples after each sort.

TABLE III

L1 cache misses and branch mispredictions (both in trillions, lower is better) of sorting 10^{24} rows of 3 key columns in row (R) data format, with the *tuple-at-a-time* (T), and *subsort* (S) approaches.

Distribution	Cache Misses		Branch Misses	
	R/T	R/S	R/T	R/S
Random	0.41	**0.40**	0.93	**0.90**
Unique128	**0.49**	0.67	1.48	**0.91**
PowerLaw	**0.39**	0.50	0.89	**0.51**

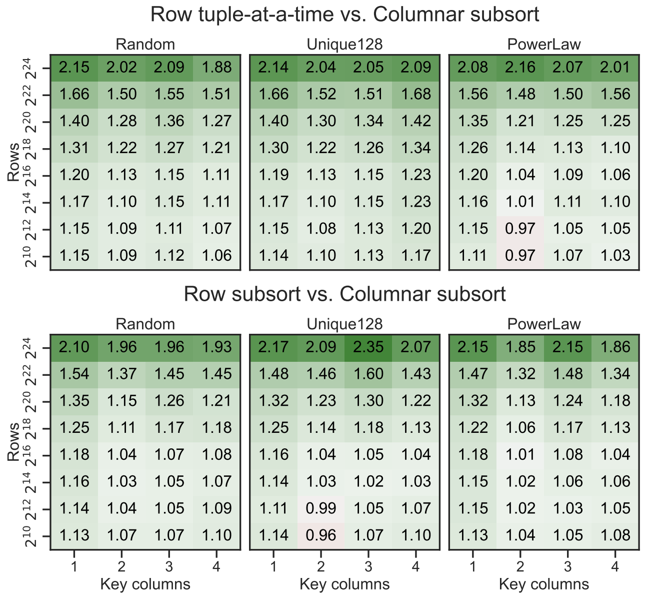

Fig. 3. Relative runtime (higher is better) of the *tuple-at-a-time* and *subsort* approaches on row format data compared to the *subsort* approach on a columnar data format.

We measure the runtime of the row format approaches and show the results in Figure 3. We use the best columnar sorting approach, the *subsort* approach, as a baseline to compare with the row sorting approaches. From the results, it is clear that sorting the row data format is more efficient than sorting the columnar data format, as the relative runtime of the row sorting approaches is greater than 1 for almost all inputs. For smaller input sizes, sorting rows has a similar performance because the data fits in the CPU's cache. Therefore, the random access incurred by the columnar approach does not affect the runtime by much. For larger input sizes, the data does not fit in the CPU's cache, and the improved cache locality of the row data format results in a better performance.

In summary, NSM tuple representation performs much better when sorting relational data than DSM. This is true regardless of data distribution, and the performance gain is especially noticeable when sorting a large number of tuples. It remains to be seen whether converting to rows, sorting, then converting back to columns is worth it in a system that uses the DSM tuple representation in its query execution engine. This will be evaluated in end-to-end benchmarks in Section VII.

V. Query Engines and Sorting

The Volcano [23] iterator model for pipelined processing leads to tuple-at-a-time query execution, which causes high interpretation overhead, making it unsuitable for OLAP systems that require low query response time. Vectorized query execution [5] amortizes this overhead with vector-at-a-time query execution. Compiled query execution [11] generates code specialized for the types present in the query, which eliminates interpretation overhead, and compiles the generated code to efficient machine code using just-in-time (JIT) compilation. These two fundamentally different models have a similar performance in OLAP workloads [12]. Although the concepts of vectorization and compilation are not orthogonal [13], most systems adopt one of the two models.

In a query execution pipeline, sorting is a pipeline breaker: The sort operator must consume all input data before being able to output any data since the very last input row may be the first row that has to be output. This necessitates materializing the input data. To a certain extent, operators that materialize their input have more ‘freedom’ than streaming operators, as they can internally represent the data in any way they see fit, as long as they output their data in a way that is compliant with the default of the execution engine. Although converting the data format from and to a different format does not come without a cost, it can significantly speed up query processing [19], [24] because it results in better memory access patterns.

Both vectorized and compiling query engines have to choose how to materialize input data for the sort operator. As we saw in the previous section, the data format directly affects sorting efficiency because it affects the cost of comparing and moving data. However, within a database system, sorting is used for many purposes. How other operators use sorted data internally, such as merge- and inequality joins [8] should also be taken into account. In this section, we discuss the effect that the fundamentally different query execution engines have on sorting.

A. Compiled Sorting

From the previous section, it is clear that sorting a row data format is significantly more efficient than sorting a columnar data format. This is especially true when many key columns and tuples are sorted. Compiling query engines use tuple-at-a-time processing on generated data types [12], such as the `OrderKey` struct that we used in our micro-benchmarks. An array of such structs is essentially relational data in row data format. This format allows compiling query engines to sort using an efficient random access iterator, such as the C++ iterator interface that is used for `std::sort` and many other efficient sort implementations [2], [4]. Furthermore, compiling engines can also compile the comparison function, which practically removes all interpretation and function call overhead.

In short, compiling query engines can use highly efficient compiled operations to compare and move values, which are the two main costs of sorting. Compiling query engines can adopt the *tuple-at-a-time* or *subsort* approaches and achieve a very efficient sorting performance, matching the best performance we have seen in our micro-benchmarks so far.

B. Vectorized (Interpreted) Sorting

Vectorization amortizes the interpretation overhead of tuple-at-a-time processing. This approach yields excellent performance in many cases because many relational operations are vectorizable. The same is true for sorting: In a system with a vectorized interpreted engine, the *subsort* approach can be used to interpret the type and `ASC`/`DESC`, `NULLS FIRST`/`LAST` order only once per key column. However, as we have seen, the columnar *subsort* approach is much less efficient than sorting row data for large input sizes.

Furthermore, some use cases in database systems cannot use the *subsort* approach. Examples of this are merge sort, merge joins, and inequality joins [8]. In these use cases, runs of sorted data are accessed with a random access iterator, and tuples are compared. For example, in a 2-way merge sort, we have a left and a right sorted run, both of which have a random access iterator. When merging the runs, the decision of incrementing either the left or right iterator relies on a full tuple comparison, i.e., comparing all key columns. Fully comparing tuples in an interpreted engine requires interpreting a type and a sort order for each key column or performing a function callback for each key column in the tuple comparison. The former creates interpretation overhead, and the latter creates function call overhead in the comparison function. This overhead is costly [5] and decreases CPU efficiency, especially when called for every tuple, which is necessary for comparing tuples of sorted data.

We illustrate this cost by comparing two *tuple-at-a-time* approaches to sorting the row data format in our micro-benchmark. The approaches are identical except that one has a statically compiled comparison function, while the other uses a dynamic function call to compare values, incurring function call overhead on every comparison. We show the results of this experiment in Figure 4. As expected, a dynamic function call is always slower than a statically compiled comparison. This effect is roughly similar across all three data distributions. We emphasize that this experiment is an oversimplification of what would be implemented in a database system with a vectorized interpreted engine. However, the experiment accurately illustrates the cost the overhead that interpreted systems deal with.

From our experiments in the previous section, it is evident that sorting data in a row format is much more efficient than sorting data in a columnar format. In this section, it has become clear that vectorized interpreted execution engines

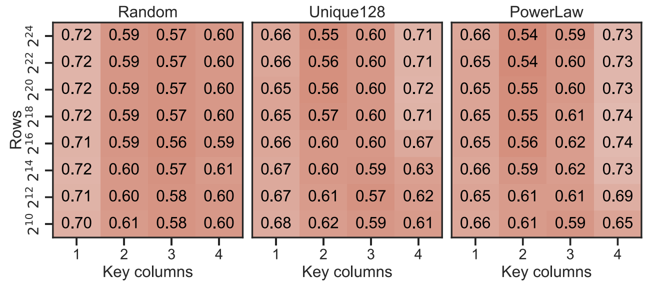

Fig. 4. Relative runtime (higher is better) of a *tuple-at-a-time* approach with a dynamic comparator compared to a static comparator on data in row format.

suffer from overheads that hurt the efficiency of sorting. The columnar *subsort* approach can be used to mitigate this. However, this approach suffers from cache inefficiency for large input sizes, as well as interpretation and function call overhead when tuples need to be compared fully, which is often the case when sorted data is used in other relational operations. Therefore, vectorized engines need to explore other solutions to overcome this overhead.

VI. Sorting Rows in an Interpreted Query Execution Engine

In this section, we discuss and evaluate techniques that can be used to improve the performance of sorting row format data in interpreted query execution engines. These existing techniques have been proposed in the literature, but not together and in the context of sorting relational data in an interpreted query engine, to the best of our knowledge.

A. Normalized Keys

Key normalization [25] is an encoding technique that produces a single order-preserving string from a sequence of values, which dates back to System R [26]. The technique is illustrated in Figure 5 for the example query in Section II, which orders the `customer` table by `c_birth_country DESC` and `c_birth_year ASC`. The first column, `c_birth_country`, is of type `VARCHAR`. The shorter string '`GERMANY`' is padded with '`0`' such that it has the same length as the longer string '`NETHERLANDS`' to ensure that the size of each normalized key is the same. Then, we flip the bits to get a descending order for this column. The second column, `c_birth_year`, is of type `INTEGER`. On a big-endian machine, the most significant byte comes last. To create an order-preserving encoding, we swap the bytes so that the most significant byte comes first. Then, we flip the sign bit such that negative integers appear before positive integers in the ascending sort order. The resulting normalized keys yield the correct order for the example query if we compare them if they are strings.

String collations are handled by evaluating the collation before encoding the string prefix. `NULL` values are handled by prefixing each value with an additional byte, denoting whether or not the value is `NULL`. This byte is then flipped depending on whether we are sorting by `NULLS FIRST` / `LAST`. If we have min / max statistics and know that the first bit of the column we are encoding is never set, it is possible to use a single bit instead.

(a)

c_birth_country	c_birth_year
NETHERLANDS	1992
GERMANY	1924

(b)

c_birth_country	c_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)

Normalized Key
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

Fig. 5. Key normalization. The original data in (a) is represented byte-by-byte as (b) in-memory on a little-endian machine. The data is encoded as (c) for `c_birth_country DESC` and `c_birth_year ASC`.

Although it is not strictly necessary to use fixed-size keys, having fixed-size keys allows keys to swap places during sorting, which improves cache efficiency, as we have seen in Section IV. We cannot generate the `OrderKey` struct without JIT compilation. Therefore we have to use `memcpy` to move the keys. If variable-size columns like strings appear in the order clause, we can only encode a prefix. The rest of the string only needs to be compared when prefixes are equal. The resulting fixed-size keys can be compared using `memcmp` rather than a complex comparison function.

The conversion from columnar data to normalized keys does not come without cost. However, this conversion can be done efficiently, in a vectorized way, amortizing interpretation overhead. The result is a key that can be compared generically without any interpretation or function call overhead.

B. The `mem*` Functions

Compiling query engines can rely on generating a data type that holds the key column values for arbitrary queries. This allows them to compile a comparison function and use the '='-operator to move data. This yields highly efficient machine code for comparing and moving tuples. Vectorized interpreted engines cannot do this and must deal dynamically with arbitrary `ORDER BY` clauses at runtime. To be able to compare and move rows of any given size, the `memcpy` and `memcmp` functions can be used.

The more information the compiler has at compile time, the more optimizations it can perform. Vectorized query engines have less information at compile-time but can benefit from pre-compiling specific procedures nonetheless. The `memcpy` and `memcmp` functions, which respectively move and compare a number of bytes, may benefit from knowing the exact number of bytes statically at compile-time rather than dynamically at runtime. We pre-compile many versions of these functions with different fixed sizes. The correct size is then chosen at runtime, like so:

```
// Static memcpy
void s_memcpy(void *dest, void *src, size_t size) {
  switch (size) {
  case 1:
    return memcpy(dest, src, 1);
  case 2:
    return memcpy(dest, src, 2);
  // And so forth
  default:
    memcpy(dest, src, size);
  }
}
```

Performing the `switch`-statement to choose the correct size at runtime causes a slight overhead. This overhead is negligible for sorting because the `size` parameter stays constant throughout, and therefore the CPU's branch predictor can predict perfectly.

We investigate the performance of calling these functions dynamically and choosing a static pre-compiled version at run-time. For `memcpy`, we sequentially copy 1.000.000 elements from a source array to a destination array, i.e., we copy the first element from the source array to the first element in the destination array, then the second, and so forth. We then take the average execution time of copying an element. We repeat this for elements of sizes 0 to 64.

Similarly, for `memcmp`, we generate two arrays, both containing 1.000.000 elements. Each element has an equal prefix, the length of which is randomly chosen between 0 and the size of the element. Then, we sequentially compare the first element of the arrays with each other, then the second, and so forth. We again take the average execution time comparing an element and repeat this for elements of sizes 0 to 64.

Copying and comparing regions of memory are highly optimized procedures. The behavior of these procedures may depend on CPU architecture. Therefore, we run this experiment on the `m6gd.8xlarge` AWS EC2 instance, which has ARM CPU architecture, as well as the instance that we have used for all our experiments so far, `m5d.8xlarge`, which has x86 CPU architecture. Detailed hardware specifications are found in Table I.

The results of this experiment are shown in Figure 6. For dynamic `memcpy`, we see a pattern that seems to be optimized for various fixed sizes, i.e., we see the same execution time for sizes 8 to 15 on both CPU architectures. For dynamic `memcmp`, we see a cyclic behavior, i.e., execution time is faster for sizes that are a multiple of 8 or 16.

The x86 CPU is slightly faster on average than the ARM CPU for both `mem*` functions. The difference between the static and dynamic version of `memcmp` is very low on this CPU. The static version is faster than the dynamic version when the size parameter is less than or equal to 16, by 5% on average. In many cases, the pre-compiled version is slower. The difference between the static and dynamic version of `memcpy` is larger: The static version is significantly slower when we copy more bytes.

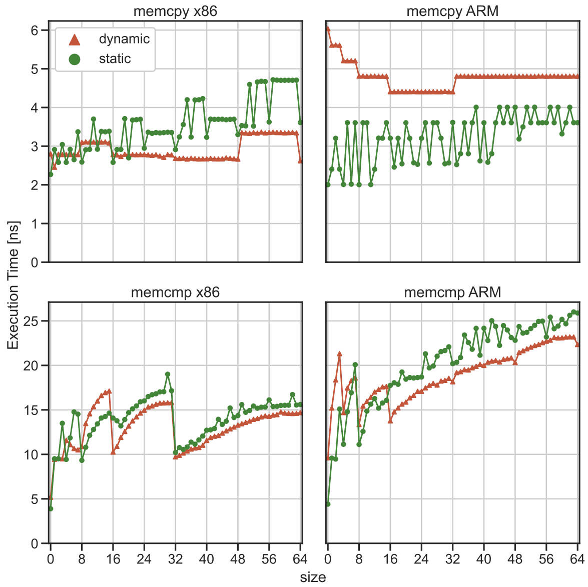

Fig. 6. Execution time of static and dynamic versions of the `memcpy` and `memcmp` functions averaged over 1.000.000 runs on x86 and ARM CPU architecture.

On the ARM CPU, the static and dynamic versions of `memcmp` differ more. When the size parameter is less than 16, the static version is 25% faster on average than the dynamic version. The most interesting result is `memcpy` on this CPU: The static version is, on average, 55% faster. This average increases to 92% when the size parameter is less than or equal to 16 and up to 121% for sizes less than or equal to 8.

C. Radix Sort

Quicksort is one of the most efficient sorting algorithms and the most commonly used one. Therefore, it is no surprise that most database systems opt for a variant of this algorithm. Quicksort, being a comparison-based sort, has a time complexity of $O(n \log n)$, where n is the number of rows to be sorted. Radix sort, on the other hand, is a distribution-based sort with a time complexity of $O(nk)$, where k is the key size. This is potentially much faster than quicksort when there are many tuples, as k does not depend on the input size. However, radix sort also has downsides. Radix sort may be much slower for large key size k, especially when n is small. Furthermore, radix sort uses twice as much memory because it needs auxiliary memory of the same size as the input, whereas quicksort sorts in place. Because normalized keys yield the correct sort order when comparing byte-by-byte with `memcmp`, the keys can also be sorted with a byte-by-byte radix sort.

In his survey on implementing sorting in database systems [10], Goetz Graefe remarks that while radix sort may seem like a good option, it has a few shortcomings for database systems: (1) if keys are long and the data contains duplicate keys, many of the passes over the data are unproductive, (2) radix sort is most effective if values are uniformly distributed, (3) if input records are nearly sorted, and the keys have a common prefix, the initial passes of radix sort are rather ineffective, and (4) radix sort has worse cache efficiency than comparison-based sorts.

We have implemented least significant digit (LSD) radix sort and most significant digit (MSD) radix sort that recurses to insertion sort for buckets with ≤ 24 tuples. We have added an optimization to both radix sorts that avoids copying data when all counted values belong to the same bucket, which helps slightly with shortcomings (1) and (3). LSD radix sort is selected when the key size is ≤ 4 bytes, and MSD radix sort is selected otherwise. Radix sort moves data every time it re-distributes, which is done using the pre-compiled `memcpy` as described in the previous subsection.

For a more fair comparison, we compare our radix sort implementation with pdqsort [4], rather than with `std::sort`. pdqsort is a highly optimized comparison-based sort that uses optimizations from BlockQuickSort [2] to reduce branch mispredictions, along with recognizing worst-case patterns

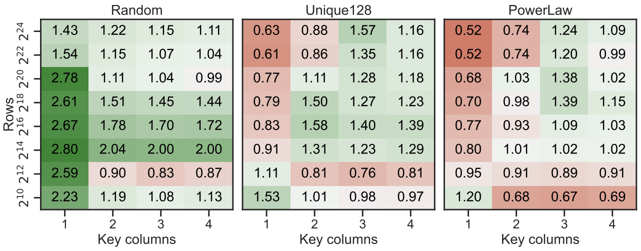

Fig. 7. Relative runtime (higher is better) of our radix sort implementation compared to pdqsort on normalized keys.

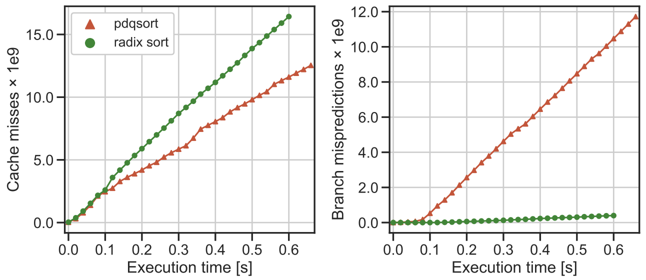

Fig. 8. Cumulative L1 cache misses and branch mispredictions of sorting 2^{24} rows with 3 key columns, PowerLaw distribution, with pdqsort and radix sort.

that quicksort traditionally does not perform well on. We compare sorting normalized keys with pdqsort and radix sort in our micro-benchmark and show the results in Figure 7. We compare these two to illustrate the differences between a comparison-based sort and radix sort on relational data.

For the Random distribution, radix sort performs better than pdqsort on almost every input and wins out by a considerable margin when there is only one key column. This is no surprise, as radix sort excels at uniform distributions with a high number of unique values, especially when the number of keys is low.

For the Unique128 distribution, pdqsort is faster than radix sort when sorting one key column. This distribution is sub-optimal for radix sort, as the number of unique values is low. pdqsort, on the other hand, performs well on this distribution: One of the worst-case patterns that pdqsort optimizes for is a low number of unique values. This effect becomes more pronounced as the data size increases. However, when we add more key columns, there are more unique tuples, as tuples are now a combination of multiple keys, making pdqsort less effective.

For the PowerLaw distribution, the results are similar to the Unique128 distribution. However, the results are even more in favor of pdqsort, as the skewed distribution generates many more tuples that are exactly the same. This makes pdqsort's optimizations for worst-case patterns more effective.

We measure performance counters of both sorting approaches for a single input and show them in Figure 8. For this input, the runtime of both algorithms is approximately the same. As expected, radix sort has a worse cache performance than pdqsort, but this difference is relatively small. We use 3 key columns; therefore, our radix sort implementation chooses MSD radix sort, which has much better cache performance than LSD radix sort. Radix sort performs better than pdqsort when it comes to branch mispredictions. It incurs virtually no branch mispredictions, as it is a mostly branchless algorithm.

VII. Evaluation

In the previous section, we applied several existing techniques to sorting relational data and evaluated them in micro-benchmarks. Our results suggest that these techniques can vastly improve the performance of relational sorting in systems with an interpreted query execution engine. Based on these findings, we have implemented an efficient relational sort within our analytical database system, DuckDB, which has a vectorized interpreted execution engine. In this section, we first describe DuckDB's sort implementation in detail. Then, we describe the sort implementations of the four other systems under benchmark. Finally, we evaluate and discuss their performance on sorting integers / floats and a relational sort benchmark based on the TPC-DS [27] data generator.

DuckDB's fully parallel sorting pipeline [28] is shown in Figure 9. DuckDB uses morsel-driven parallelism [6]; therefore, each thread collects roughly the same amount of data in parallel. Key and payload columns are converted separately to row formats, with key columns being converted to normalized keys. This conversion is performed by copying one 'group' of vectors at a time, one vector at a time, making this an efficient and mostly cache-resident process. We copy the values in a vector using tight loops templated on a data type. Both row formats use 8-byte alignment and have fixed-size rows; variable-sized types like strings are stored separately. Therefore, we encode only a prefix of variable-sized types like strings in our normalized keys: We compare the rest of the string only if the prefixes are equal. For strings, we encode the first 12 bytes. However, this could be chosen at runtime based on available statistics. When the amount of data collected by a thread reaches a threshold, we sort the normalized keys with a thread-local radix sort, or pdqsort if there are strings, with `memcmp` as the comparison function. We use a version of pdqsort that is modified to specifically deal with our normalized keys, such that we can use inlined tuple comparisons[2]. Then, we reorder the payload, creating runs of sorted data that are added to a thread-global state.

After all the input data has been added to the global state, we start a 2-way cascaded merge sort. This is trivial to parallelize when there are many more sorted runs than threads, as we can assign each thread to merge two sorted runs. However, as the runs get merged, parallelization falls apart, until the last two sorted runs are merged by a single thread. Therefore, we parallelize this phase using Merge Path [3]. Merge Path creates partitions that can be merged independently and, therefore, in parallel. The partition boundaries are efficiently computed with a binary search. During merge sort, we compare tuples with

[2] The DuckDB source code can be found at https://github.com/duckdb/duckdb

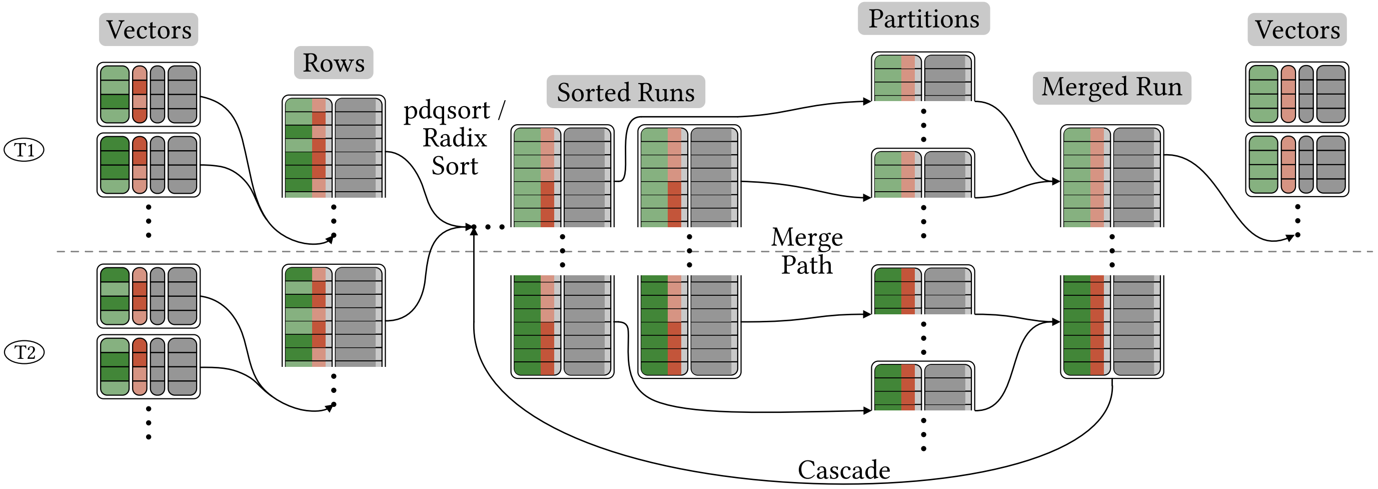

Fig. 9. Full sorting pipeline in DuckDB. Incoming vectors from worker threads (illustrated as T_1 and T_2 in the figure) are converted to 8-byte aligned row formats. Key columns are normalized and stored separately from the payload. The normalized keys are sorted with radix sort or pdqsort, creating sorted runs. The sorted runs are then partitioned and merged in parallel until one run remains. Finally, the result is converted back to vectors.

`memcmp`. We use static pre-compiled versions of the `mem*` functions as described in Section VI.

We compare DuckDB's sorting performance with four other OLAP/HTAP engines: ClickHouse [14], MonetDB [15], HyPer [11], and Umbra [16]. OLTP-focused systems like PostgreSQL and MySQL cannot deliver competitive performance on sorting large datasets. Despite their row-major data layouts, their execution engines suffer from large per-value overheads.

ClickHouse uses a columnar format throughout the sort and performs thread-local sorts with radix sort if sorting by a *single integer column*; otherwise, it uses pdqsort using a tuple-at-a-time comparison approach. JIT compilation is used to reduce some of the interpretation overhead. After the thread-local sorts are done, the sorted runs are merged using a k-way merge. MonetDB also uses a columnar format throughout the sort, using a *single-threaded* quicksort implementation. A subsort approach is used when sorting by multiple key columns. After sorting the key columns, the payload is collected in sorted order. HyPer and Umbra have a compiled, row-based sorting implementation similar to what is described in [6]. Threads perform a thread-local quicksort that is similar to pdqsort. The results are then merged using a parallel k-way merge. This merge is performed on pointers rather than physically moving the data. The data is physically collected in the sorted order when the output of the sort operator is read.

A. Benchmarking Relational Sorting

Although sorting can easily dominate the runtime of a query, *isolating* the sorting performance of a database system in a benchmark is difficult because we can only reliably observe end-to-end query runtime without access to the source code. In addition, streaming query execution interleaves the execution of multiple operators, which further complicates isolating sorting performance, even if the system has a query profiler. Measuring end-to-end query runtime introduces unwanted overhead unrelated to sorting, e.g., parsing, optimizing, scanning base tables, and transferring the result set through a client protocol. Especially the latter is costly and can easily dominate the query execution time for large result sets [21].

Therefore, we want to use a query that produces a small result set and where execution time is dominated by sorting. A full sort is often optimized out of the query plan if it is not strictly needed. For example, `ORDER BY ... LIMIT 1` will trigger a top N operator rather than the full sort operator. Furthermore, if we aggregate over a subquery that sorts, the sort will likely be discarded by the optimizer because it is irrelevant to the aggregate. We can circumvent this by disabling the optimizer, which is often possible using a configuration setting. This is inadvisable, however, as disabling it may impact performance differently across systems. Instead, we 'trick' the optimizer with the following query:

```sql
SELECT count(payload_column),
FROM (SELECT payload_column
      FROM input_table
      ORDER BY key_column1,
               key_column2,
               ...
      OFFSET 1);
```

In this query, the count aggregate reduces the size of the result set to 1, making the result set serialization negligible. The count aggregate reads the sorted subquery, forcing systems that lazily collect a sorted payload to collect it fully. This is an important detail that ensures that all systems under benchmark perform the same work. The count aggregate is cheap to compute and therefore does not add noticeable overhead to the query. Finally, we add the `OFFSET 1` so that the optimizers in the systems we benchmarked do not optimize the `ORDER BY` in the subquery away[3]. We repeat this query 5 times and report only the median end-to-end runtime.

[3] The source code for our end-to-end benchmarks can be found at https://github.com/lnkuiper/experiments/sorting

B. Random Integers & Floats

For our first benchmark, we generate two sets of 10 tables containing 10 to 100 million rows in increments of 10 million. The first set contains 32-bit integers from 0 to 99.999.999, shuffled. The second set contains 32-bit floating point numbers between $-1e9$ and $1e9$, taken from a random distribution. With this benchmark, we measure raw, single-key sorting performance. Comparing floats is more expensive than comparing integers; therefore, we expect integer sorting to be slightly faster. Following the findings in our micro-benchmarks, we also expect the performance of the columnar approaches of ClickHouse and MonetDB to degrade more quickly with the number of tuples than the row-based approaches due to a worse cache performance. We benchmark the sorting performance of all systems using the `m5d.8xlarge` instance.

We show the results of this benchmark in Figure 10. MonetDB is much slower than the other systems; therefore, we have cut off the figure to make the differences between the other systems more clearly visible. MonetDB's single-threaded sort takes 29.18s and 36.39s for 100 million integers and floats, respectively. For reference, when limited to a single thread, DuckDB takes 10.41s for the integers and 9.14s for the floats. As expected, all systems sort floats slightly slower than integers due to the more expensive comparison function of floats, except DuckDB. DuckDB does not suffer from this because it encodes both the floats and integers as normalized keys, then sorts them precisely the same. In this benchmark specifically, DuckDB sorts the floats faster than the integers because the floats have a more uniform distribution, making radix sort more effective.

ClickHouse's sort is competitive with DuckDB, HyPer, and Umbra at 10 million integers, but as the data size increases, its performance degrades more quickly than the other three systems. This is in line with our expectations, as the columnar format that ClickHouse uses has a worse cache performance. DuckDB, HyPer, and Umbra, all of which use a more cache-efficient row-based approach to sorting, show strong scaling with the number of tuples. DuckDB sorts 100 million integers in 1.55s. HyPer and Umbra, which have similar implementations, have a very similar performance on this benchmark, with Umbra being slightly faster than HyPer.

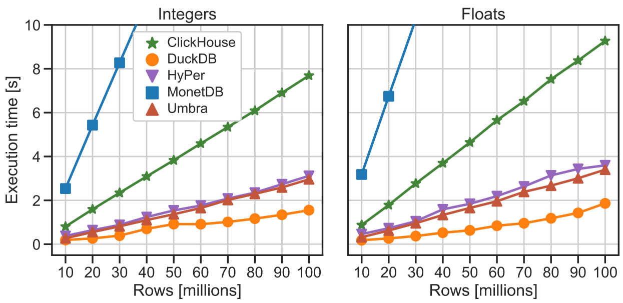

Fig. 10. Execution times (lower is better) of sorting 10 to 100 million random integers and floats in increments of 10 million.

C. TPC-DS Catalog Sales Table

For our second benchmark, we use the largest table from TPC-DS, `catalog_sales`, at scale factors 10 and 100. The cardinality of this table can be found in Table IV. We select only `cs_item_sk`, and sort it by up to four key columns: `cs_warehouse_sk`, `cs_ship_mode_sk`, `cs_promo_sk`, and `cs_quantity`. With this benchmark, we measure how well the systems deal with multiple key columns, which should increase the cost of comparing tuples. Our micro-benchmarks would suggest that the performance of columnar sorting approaches degrades more with additional key columns because they cause more random access than row-based approaches.

TABLE IV
CARDINALITY OF TPC-DS TABLES.

SF	`catalog_sales`	`customer`
10	14.401.261	-
100	143.997.065	2.000.000
300	-	5.000.000

We show the results of sorting the `catalog_sales` table in Figure 11. Again, MonetDB's performance is much slower than the other systems; therefore, we cut off the figure. MonetDB is around 3x slower when sorting by four key columns than by one key column. ClickHouse has a competitive sorting performance when sorting by one key column because it only has random access in one column and uses radix sort. However, when sorting by two key columns, ClickHouse slows down by around 4x compared to sorting by one column because it switches from radix sort to pdqsort with a tuple-at-a-time comparison function that has branches and causes random access in both key columns. As expected, DuckDB's, HyPer's, and Umbra's row-based sorting approaches lose less performance here, as they have good cache performance when comparing subsequent key columns. The cost of the comparison function, however, does still matter for the row-based approaches: Umbra is up to 2.43x and 2.96x slower when sorting four key columns than when sorting one key column at scale factors 10 and 100, respectively, while DuckDB and HyPer are only around 1.5x slower.

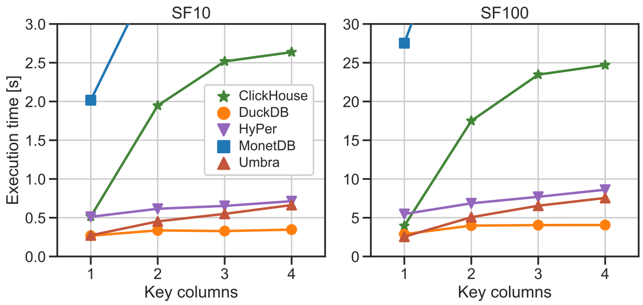

Fig. 11. Execution times (lower is better) at scale factor 10 and 100 of sorting 1 to 4 key columns (`cs_warehouse_sk`, `cs_ship_mode_sk`, `cs_promo_sk`, `cs_quantity`) of the `catalog_sales` table.

D. TPC-DS Customer Table

For our third benchmark, we use the customer table from TPC-DS at scale factors 100 and 300. The cardinality of this table can be found in Table IV. We select c_customer_sk from this table, and sort it by either c_birth_year, c_birth_month, and c_birth_day, all three of which are INTEGER columns, or by c_last_name and c_first_name, both of which are VARCHAR columns. With this benchmark, we measure how well the systems sort by fixed- vs. variable-sized types. Comparing strings is much more costly than comparing integers; therefore, we expect that sorting strings will be slower.

We show the results of sorting the customer table in Figure 12. We have again cut off the figure so the differences between the systems are more visible. On the x-axis we have the two categories: integer and string. As expected, sorting strings is slower than sorting integers for all five systems. For ClickHouse and MonetDB, this difference is around 3x. For HyPer and Umbra, this difference is much lower, between 1.24x and 1.62x. At scale factor 300, the difference between sorting the integers and strings is higher for DuckDB, at 2.03x. Despite this, DuckDB has a competitive string sorting performance. This larger difference between sorting integers and strings for DuckDB can be attributed to using radix sort for the integers while using pdqsort for the strings.

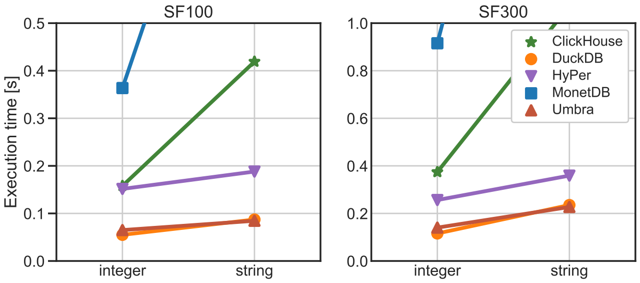

Fig. 12. Execution times (lower is better) at scale factor 100 and 300 of sorting the c_birth_year, c_birth_month, and c_birth_day columns (integer), and the c_last_name and c_first_name columns (string) of the customer table.

VIII. Conclusion

In this work, we have discussed the challenges of sorting relational data efficiently in OLAP systems. Following this discussion, we have evaluated different approaches to sorting data using micro-benchmarks. These approaches differed in their row and columnar data formats, their interpreted and compiled execution engines and tuple comparison method. We have shown that, for columnar data, sorting by one column at a time is much more efficient than sorting by all columns at once. This efficiency can be attributed to a better cache performance and fewer branches in the tuple comparison function. However, we have found that sorting data in a row format is almost always better than sorting columnar data, mostly due to an even better cache performance.

Following these findings, we have identified that database systems with a compiling query execution engine can efficiently sort row format data by generating query-specific data types and a comparison function. In systems with a vectorized interpreted engine, however, the efficiency of sorting rows is hindered by interpretation and function call overhead in the comparison function. To overcome this overhead, we have proposed using key normalization, pre-compiled memcpy and memcmp, and radix sort. Radix sort, however, is less effective when sorting strings. In this case, a comparison-based sorting algorithm like pdqsort is superior.

Finally, we implemented and released these techniques in DuckDB, which has a vectorized interpreted query execution engine. We evaluated our implementation with end-to-end sorting benchmarks. We compared our results on this benchmark with four other analytical database management systems, which have a different execution engine. This comparison showed that our sort implementation matches or outperforms all other systems under benchmark and, therefore, that the proposed techniques effectively overcome the interpretation and function call overhead that systems with an interpreted execution engine theoretically would be at a disadvantage of.

IX. Future Work

DuckDB uses pdqsort in its thread-local sorts when strings are present; otherwise, it uses radix sort. Variables other than the data type affect the efficiency of these algorithms, for example, key size, number of tuples, the estimated number of unique values, and other statistics. A heuristic that takes these variables into account could improve the algorithm choice. Additionally, pdqsort could be used within the recursive calls to MSD radix sort, which may improve sorting performance even further.

Besides the sort operator, the aggregate, join, and window operators are also blocking operators. They materialize their input because they cannot stream data, i.e., all input data must be consumed and processed before data can be output. In DuckDB, these operators use a unified row format. However, DuckDB's vectorized engine moves vectors between operators. When we chain blocking operators, for example, when we aggregate over the output of a join or sort the output of an aggregate, the data is converted from rows to vectors and back. We could prevent these conversions by allowing the execution engine to move row format data between operators.

The same blocking operators risk running out of memory because they must materialize their input. When the data size exceeds the memory limit, many main-memory database systems either cannot complete the query or switch to an out-of-core strategy, orders of magnitude slower than the in-memory strategy. This could be overcome by designing these operators so that their performance gracefully degrades as the data size exceeds the memory limit. Utilizing DuckDB's row format to be able to offload the data to storage in a unified way could facilitate this.

REFERENCES

[1] A. LaMarca and R. E. Ladner, "The Influence of Caches on the Performance of Sorting," *Journal of Algorithms*, vol. 31, no. 1, pp. 66–104, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677498909853

[2] S. Edelkamp and A. Weiß, "BlockQuicksort: Avoiding Branch Mispredictions in Quicksort," *ACM J. Exp. Algorithmics*, vol. 24, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3274660

[3] O. Green, S. Odeh, and Y. Birk, "Merge Path - A Visually Intuitive Approach to Parallel Merging," *CoRR*, vol. abs/1406.2628, 2014. [Online]. Available: http://arxiv.org/abs/1406.2628

[4] O. R. L. Peters, "Pattern-defeating Quicksort," *CoRR*, vol. abs/2106.05123, 2021. [Online]. Available: https://arxiv.org/abs/2106.05123

[5] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *Second Biennial Conference on Innovative Data Systems Research, CIDR, Online Proceedings*. Asilomar, CA, USA: www.cidrdb.org, 2005, pp. 225–237. [Online]. Available: http://cidrdb.org/cidr2005/papers/P19.pdf

[6] V. Leis, P. Boncz, A. Kemper, and 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*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 743–754. [Online]. Available: https://doi.org/10.1145/2588555.2610507

[7] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting Worst-Case Optimal Joins in Relational Database Systems," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 1891–1904, Jul. 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407797

[8] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis, "Lightning Fast and Space Efficient Inequality Joins," *Proc. VLDB Endow.*, vol. 8, no. 13, pp. 2074–2085, Sep. 2015. [Online]. Available: https://doi.org/10.14778/2831360.2831362

[9] M. Bandle, J. Giceva, and T. Neumann, "To Partition, or Not to Partition, That is the Join Question in a Real System," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 168–180. [Online]. Available: https://doi.org/10.1145/3448016.3452831

[10] G. Goetz, "Implementing Sorting in Database Systems," *ACM Comput. Surv.*, vol. 38, no. 3, pp. 10–es, Sep. 2006. [Online]. Available: https://doi.org/10.1145/1132960.1132964

[11] A. Kemper and T. Neumann, "HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11. USA: IEEE Computer Society, 2011, pp. 195–206. [Online]. Available: https://doi.org/10.1109/ICDE.2011.5767867

[12] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask," *Proc. VLDB Endow.*, vol. 11, no. 13, pp. 2209–2222, sep 2018.

[13] J. Sompolski, M. Zukowski, and P. Boncz, "Vectorization vs. Compilation in Query Execution," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, ser. DaMoN '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 33–40. [Online]. Available: https://doi.org/10.1145/1995441.1995446

[14] B. Imasheva, A. Nakispekov, A. Sidelkovskaya, and A. Sidelkovskiy, "The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse," in *Optimization of Complex Systems: Theory, Models, Algorithms and Applications, WCGO 2019, World Congress on Global Optimization, Metz, France, 8-10 July, 2019*, ser. Advances in Intelligent Systems and Computing, vol. 991. Springer, 2019, pp. 820–828. [Online]. Available: https://doi.org/10.1007/978-3-030-21803-4_82

[15] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012. [Online]. Available: http://sites.computer.org/debull/A12mar/monetdb.pdf

[16] T. Neumann and M. 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*. Amsterdam, Netherlands: www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[17] D. Lemire and O. Kaser, "Reordering Columns for Smaller Indexes," *Inf. Sci.*, vol. 181, no. 12, pp. 2550–2570, jun 2011. [Online]. Available: https://doi.org/10.1016/j.ins.2011.02.002

[18] G. Moerkotte, "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, ser. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 476–487.

[19] M. Zukowski, N. Nes, and P. Boncz, "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing," in *Proceedings of the 4th International Workshop on Data Management on New Hardware*, ser. DaMoN '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 47–54. [Online]. Available: https://doi.org/10.1145/1457150.1457160

[20] S. Manegold, P. Boncz, N. Nes, and M. Kersten, "Cache-Conscious Radix-Decluster Projections," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. USA: VLDB Endowment, 2004, pp. 684–695.

[21] M. Raasveldt and H. Mühleisen, "Don't Hold My Data Hostage: A Case for Client Protocol Redesign," *Proc. VLDB Endow.*, vol. 10, no. 10, pp. 1022–1033, Jun. 2017. [Online]. Available: https://doi.org/10.14778/3115404.3115408

[22] D. R. Musser, "Introspective Sorting and Selection Algorithms," *Softw. Pract. Exper.*, vol. 27, no. 8, pp. 983–993, aug 1997.

[23] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: Association for Computing Machinery, 1990, pp. 102–111. [Online]. Available: https://doi.org/10.1145/93597.98720

[24] Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," *SIGMOD Rec.*, vol. 26, no. 2, pp. 159–170, jun 1997. [Online]. Available: https://doi.org/10.1145/253262.253288

[25] M. W. Blasgen, R. G. Casey, and K. P. Eswaran, "An Encoding Method for Multifield Sorting and Indexing," *Commun. ACM*, vol. 20, no. 11, pp. 874–878, Nov. 1977. [Online]. Available: https://doi.org/10.1145/359863.359892

[26] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational Approach to Database Management," *ACM Trans. Database Syst.*, vol. 1, no. 2, pp. 97–137, jun 1976. [Online]. Available: https://doi.org/10.1145/320455.320457

[27] M. Poess, B. Smith, L. Kollar, and 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*, ser. SIGMOD '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 582–587. [Online]. Available: https://doi.org/10.1145/564691.564759

[28] L. Kuiper, "Fastest table sort in the West - Redesigning DuckDB's sort," 2021. [Online]. Available: http://bit.ly/duckdb-sort