Fast Multi-Column Sorting in Main-Memory Column-Stores*

Wenjian Xu[§] Ziqiang Feng[§] Eric Lo[†]
[§]Department of Computing, The Hong Kong Polytechnic University
[†]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[§]{cswxu, cszqfeng}@comp.polyu.edu.hk [†]ericlo@cse.cuhk.edu.hk

ABSTRACT

Sorting is a crucial operation that could be used to implement SQL operators such as GROUP BY, ORDER BY, and SQL:2003 PARTITION BY. Queries with multiple attributes in those clauses are common in real workloads. When executing queries of that kind, state-of-the-art main-memory column-stores require one round of sorting per input column. With the advent of recent fast scans and denormalization techniques, that kind of *multi-column sorting* could become a bottleneck. In this paper, we propose a new technique called "code massaging", which manipulates the bits across the columns so that the overall sorting time can be reduced by eliminating some rounds of sorting and/or by improving the degree of SIMD data level parallelism. Empirical results show that a main-memory column-store with code massaging can achieve speedup of up to 4.7X, 4.7X, 4X, and 3.2X on TPC-H, TPC-H skew, TPC-DS, and real workload, respectively.

1. INTRODUCTION

In modern main-memory column-stores like SAP HANA [13], MonetDB [8], Vectorwise [42], and Oracle Exalytics [34], queries are read-mostly, data columns are encoded [30, 29] for memory efficiency, and their performance goal is to support real-time analytic.

Sorting, which could be used to implement SQL operators such as `GROUP BY`, `ORDER BY`, and `PARTITION BY`[1], is a crucial operation in main-memory column-stores. There are a plethora of works in main-memory databases on making sorting run faster. Works including [21] and [10] pioneered the discussion of accelerating sorting on modern CPU architectures by leveraging SIMD (Single Instruction Multiple Data) instruction set (e.g., SSE 128-bits, AVX2 256-bits), whose instructions can execute one multi-operand operation per cycle (each operand is b bits, where b, the *bank size*, equals 8, 16, 32, or 64 in AVX2). Subsequent works further improved the efficiency of in-memory sorting by reducing memory access through multi-way merging [37], increasing IPC (instructions per cycle) through bitonic merge network [26], and enhancing scalability on multi-core through partitioning [3, 5, 36]. In what follows, we refer any SIMD-enabled sorting implementation as *SIMD-sort*.

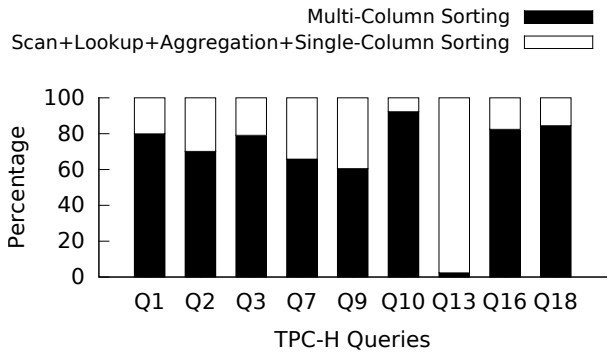


Figure 1: Time breakdown of TPC-H queries with ByteSlice [14] fast scans and WideTable [31] implemented. Queries with multiple attributes in their `GROUP BY` and/or `ORDER BY` clauses. 10GB data. Time spent on multi-column sorting is significant in general, except one exceptional case (Q13).

In state-of-the-art column-store implementations, when there are more than one column needed to be sorted, they are typically sorted *column-at-a-time*. Consider a TPC-H query with an `ORDER BY order_date, retail_price` clause. Column-stores like MonetDB [8] would first sort column `order_date` in the first round; then in the second round, it sorts column `retail_price` for each group of (tied) `order_date` values. With the recent advent of denormalization techniques that eliminate joins [31, 29] and fast scan techniques that perform filtering with early stopping [30, 14], that kind of *multi-column sorting* could take up a significant portion of the query execution time. Figure 1 shows the time breakdown of executing TPC-H queries that have multiple attributes in their `GROUP BY` and/or `ORDER BY` clauses, which can trigger multi-column sorting (experimental setting is given in Section 6). In that experiment, WideTable denormalization [31, 29] and ByteSlice [14] fast scan were used, and we used one of the most efficient SIMD-sort implementations to-date: merge-sort with sorting-network kernel [5]. We see that **the time spent on multi-column sorting is significant** — taking up 60% (Q9) to 92% (Q10) of time in general, except for Q13, which carries out a multi-column sorting after `GROUP BY` on a single attribute (so that the

*This work is partly supported by the Research Grants Council of Hong Kong (GRF 521012, 15200715), Hong Kong Polytechnic University, and Research Committee of CUHK.

†Work done while at Hong Kong Polytechnic University.

[1]`PARTITION BY` is part of SQL:2003 standard for the use of window functions.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915205

query execution is actually dominated by the single-column sorting instead).

In this paper, we take the first step to study the issues of optimizing multi-column sorting in main-memory column-stores. Queries that could get benefit include all those containing `GROUP BY`, `ORDER BY`, or `PARTITION BY` clauses with multiple attributes, which are not uncommon in real workloads. In TPC-H, we found 9 out of 22 such queries. In TPC-DS benchmark, we found 71 out of 99 such queries.

Our key technique is called **code massaging**, which leverages the properties of multiple available bank sizes in SIMD instructions and forms sort keys that best utilize SIMD's data parallelism. Using the encoding scheme in [30], the TPC-H attributes `order_date` and `retail_price` can be respectively encoded as 12-bit column and 17-bit column. For the example query above, state-of-the-art main-memory column-store implementations would carry out sorting in two rounds: first sort `order_date` using SIMD-sort with 16-bit banks (i.e., $b = 16$); then sort `retail_price` using SIMD-sort with 32-bit banks (i.e., $b = 32$), as `retail_price` column is wider than 16 bits. A main-memory column-store, if equipped with code massaging, could do a lot more better. For example, code massaging would consider first "stitching" the two columns as one (12+17)-bit super-column "`order_date-retail_price`" and then sorting that using a 32-bit bank SIMD-sort. In that *code massage plan*, the two columns are sorted in one go, thereby eliminating one round of sorting. Code massaging would also allow "bit-borrowing" among columns. For example, another feasible plan is column `order_date` "borrows" one bit from column `retail_price` so that the query execution engine can first sort the (12+1)-bit `order_date` column using a 16-bit bank SIMD-sort , and then sort the (17–1)-bit `retail_price` column using another 16-bit bank SIMD-sort. While this plan still requires two rounds of sorting, a 16-bit bank SIMD-sort is used in place of a 32-bank SIMD-sort in the second round, attaining 2X data parallelism there.

To the best of our knowledge, this is the first work to pinpoint and tackle the issue of multi-column sorting, which is emerging as a time significant phase in recent main-memory column-store implementations. This paper makes four contributions around this new issue. The principal contribution is code massaging, a technique that manipulates the bits across the columns to accelerate multi-column sorting. In order to choose the most promising code massage plan, our second contribution is an *architectural-aware cost model* that quantifies the cost of different code massage plans, with various architectural-sensitive parameter values (e.g., latency of a cache miss) calibrated from controlled experiments. Our third contribution is a *round-based greedy plan search algorithm*, which efficiently explores the huge search space to identify the most promising plan. Our last contribution is the integration of code massaging into a column-store prototype and the experimental evaluation on TPC-H, TPC-H Skew, TPC-DS and real data. Empirical results report a query speedup of up to 4.7X, 4.7X, 4X, 3.2X on TPC-H, TPC-H Skew, TPC-DS, and real workload, respectively.

The remainder of this paper is organized as follows: Section 2 presents some essential background information and discusses the related work. Section 3 presents our code massaging technique. Section 4 presents an architectural-aware cost model. Section 5 presents the plan search algorithm. Section 6 presents our experimental results. Section 7 contains our concluding remarks. For easy reading, Table 3 in the Appendix summarizes the most frequently used symbols in this paper.

2. BACKGROUND AND RELATED WORK

SIMD SIMD instructions interact with S-bit SIMD registers as a *vector of banks*. A bank is a continuous section of b bits. In AVX2, $S = 256$ and b is 8, 16, 32, or 64. We adopt these values in this paper but remark that our techniques can be straightforwardly extended to other models (e.g., 512-bit AVX-512 [24] and Larrabee [38]). The choice of b, the *bank width*, is on per instruction basis. A SIMD instruction carries out the same operation on the vector of banks simultaneously. For example, the `_mm256_add_epi32()` instruction[2] performs an 8-way addition between two SIMD registers, which adds eight pairs of 32-bit integers simultaneously. Similarly, the `_mm256_add_epi16()` instruction performs 16-way addition between two SIMD registers, which adds sixteen pairs of 16-bit short integers simultaneously. The degree of such *data-level parallelism* is S/b.

Column Encoding In modern main-memory column-stores, data is often stored in an encoded form [6, 12, 13, 27]. For example, in dictionary encoding, native column values are compressed as fixed-length order-preserving *codes*. All data types including numeric and strings are encoded as unsigned integer codes. For example, strings are encoded by building a sorted dictionary of all strings in that column [7, 27]. Floating point numbers with limited precision can be scaled to integers by multiplication with a certain factor [12]. From now on, we adopt fixed-length encoding and use the terms **code** and **value** interchangeably.

Fast Scan/Lookup and Denormalization Scan and lookup are two core operations in main-memory column-stores. A scan operation scans a column and returns a result bit vector that indicates which records satisfy a filter. Once a column scan is completed, the result bit vector is converted into a list of record numbers, which is then used to look up values from other columns of interest for a query. Recent studies [25, 30, 14] discuss how to carry out scans at the speed of the processing core. Their main idea is to pack multiple codes from the same column into the same CPU word so as to leverage SIMD to carry out parallel predicate evaluation. The latest work is ByteSlice [14], which chops codes into multiple bytes, so that scans can be executed very efficiently through early stopping while lookups can still be very efficient through byte stitching.

A recent trend of main-memory column-stores is to pre-join tables upfront and materialize join results as one or more *WideTables* [31, 29]. Queries on the original database, even complex join queries, can then be handled by fast scans on WideTables. Such a denormalization approach would not incur much storage overhead because of the various effective encoding schemes enabled by columnar storage [31, 29].

Sorting Sorting is known to be both computational and memory intensive in main-memory column-stores and has attracted great attention in optimizing its performance. This line of works started with [21] and [10], which pioneered the discussion of SIMD-enabled sorting. Then, sorting had been revisited in the context of *sort merge join* [26, 37, 3, 5]. It was first shown that a merge-sort implementation based on bitonic sorting network and bitonic merge yields superior performance [26, 37]. Later, merge-sort implementations based on multi-way merge and data partitioning for NUMA architectures were introduced [3, 5]. Recently, to mitigate data skew, a merge-sort implementation with range-partitioning and sampling was proposed [36]. All these works are based on the merge-sort framework because the structure of the merge-sort is intrinsi-

[2]Technically, it is a C function supported by SIMD instructions. We use the C function names in place of the SIMD instructions for simplicity.

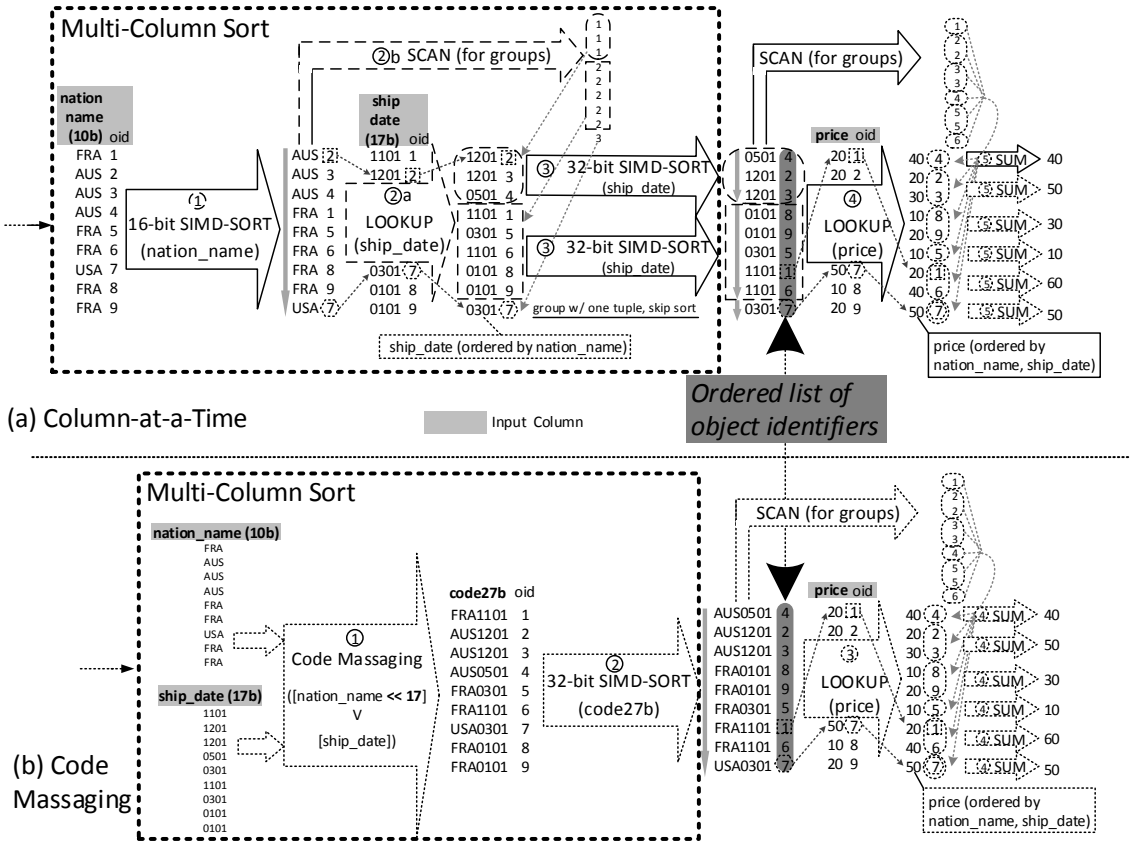


Figure 2: Multi-column sorting. Labels ①, ②, ... denote the execution sequences. Steps with the same label mean they can be executed in parallel. (a) Without code massaging (column-at-a-time) (b) With code massaging (1 round)

cally easy to be parallelized on multi-cores, compared with other comparison-based sorts. The internal sorting operation of merge-sort is usually based on *sorting network*, as sorting networks can be implemented as a series of branch-free SIMD `min/max` instructions, thereby minimizing branch mis-predictions. Non-comparison based sorting such as radix-sort had also been discussed [37, 11, 36, 22]. These works focused on exploiting SIMD to parallelize radix calculation, and using streaming load/store instructions to reduce cache pollution.

Overall, the performances of merge-sort and radix-sort are data-dependent and comparable [37, 36]. Recently, the discussion of sorting has also been extended to sorting an array of structures [22]. In [35], the discussion of sorting has been extended to the emerging MIC (many integrated core) platforms (e.g., Intel Xeon Phi).

Multi-Column Sorting State-of-the-art column-store implementations sort multiple columns column-at-a-time [39, 13, 20, 42, 34]. Figure 2a illustrates an example of that. The example illustrates the execution of the query below $Q1$, whose result is listed on the right:

```
SELECT SUM(price) //Query Q1
FROM ...
GROUP BY nation_name, ship_date
```

nation_name	ship_date	SUM (price)
AUS	0501	40
AUS	1201	50
...	...	...
USA	0301	50

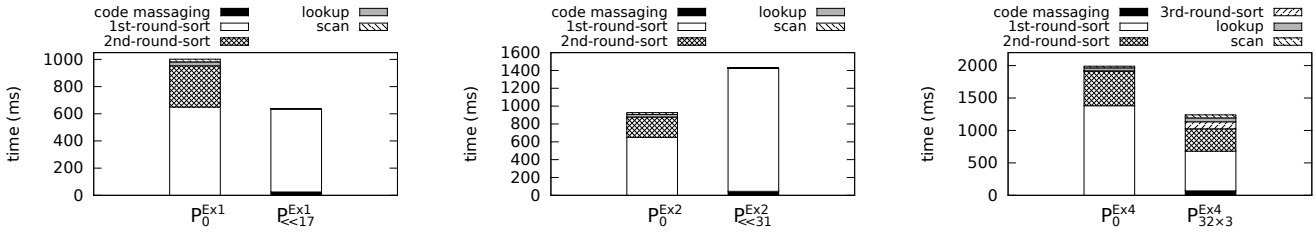
In the example, the first round of sorting is carried out on column `nation_name` (Step ①). Then, the list of (rearranged) object identifiers (oid's) of the sorted column is extracted and passed to a lookup operator, which aims to rearrange the codes in the other column, `ship_date`, based on the order predefined by sorted `nation_name` (Step ②a). Based on the grouping information obtained by scanning the sorted `nation_name` (Step ②b), the second round of sorting is carried out on column `ship_date` per `nation_name` group (e.g., AUS) (Step ③). After that round of sorting, each ⟨`nation_name`, `ship_name`⟩ group (e.g., ⟨AUS,0501⟩) invokes the lookup operator to fetch the values belonging to that group from column `price` (Step ④). Finally, price values within each group are aggregated through the SUM operator (Step ⑤).

3. CODE MASSAGING

The goal of code massaging is to speed up multi-column sorting triggered by the existence of `ORDER BY`, `GROUP BY`, or `PARTITION BY` clauses in a SQL query. Its key idea is to manipulate the bits across the columns to be sorted to form new sort keys.

For correctness, we have to ensure that sorting multiple columns with code massaging would return the same ordered list of object identifiers as with the one without using code massaging (i.e., sorting m columns in m rounds). Without loss of generality, we focus on sorting in ascending order. Denote C_i as a column/attribute, w_i as the code width of C_i, and $t.C_i$ as the code/value of attribute C_i in tuple t. For an `ORDER BY` C_1, C_2, ..., C_m clause, the correct sorting order of two tuples t_1 and t_2 is defined as $t_1 \prec t_2$, where $t_1.C_k < t_2.C_k$ for some $1 \leq k \leq m$ and $t_1.C_i = t_2.C_i$ for all $i < k$.

Now, let us denote $y||z$ as the code formed by concatenating code y and code z, and $|x|$ be the width of code x. For instance, given $y = (111)_2$ and $z = (0000)_2$, $y||z = (1110000)_2$. So,



(a) [Example Ex1]
ORDER BY 10-bit-column, 17-bit column

(b) [Example Ex2]
ORDER BY 15-bit-column, 31-bit column

(c) [Example Ex4]
ORDER BY 48-bit-column, 48-bit column

Figure 3: Performance of Various Code Massage Plans

given two tuples $t_1 = \langle x_1, y_1||z_1\rangle$ and $t_2 = \langle x_2, y_2||z_2\rangle$, where $|x_1| = |x_2|, |y_1| = |y_2|, |z_1| = |z_2|$, we have:

LEMMA 1 (CORRECTNESS OF CODE MASSAGING).

$$\begin{aligned} & t_1 \prec t_2 \\ \Leftrightarrow & \langle x_1, y_1||z_1\rangle \prec \langle x_2, y_2||z_2\rangle \\ \Leftrightarrow & \langle x_1||y_1, z_1\rangle \prec \langle x_2||y_2, z_2\rangle \end{aligned}$$

PROOF. If $\langle x_1, y_1||z_1\rangle \prec \langle x_2, y_2||z_2\rangle$, then there are two possible relationships between x_1 and x_2: either $x_1 < x_2$ or $x_1 = x_2$.

Case 1 ($x_1 < x_2$):

$$\begin{aligned} & x_1 < x_2 \\ \Rightarrow & x_1||y_1 < x_2||y_2 \\ \Rightarrow & \langle x_1||y_1, z_1\rangle \prec \langle x_2||y_2, z_2\rangle \end{aligned}$$

Case 2 ($x_1 = x_2$):

Then there must be $y_1||z_1 < y_2||z_2$. Hence two possible relationships between y_1 and y_2: either $y_1 < y_2$ or $y_1 = y_2$.

Case 2a: ($y_1 < y_2$):

$$\begin{aligned} & x_1 = x_2, y_1 < y_2 \\ \Rightarrow & x_1||y_1 < x_2||y_2 \\ \Rightarrow & \langle x_1||y_1, z_1\rangle \prec \langle x_2||y_2, z_2\rangle \end{aligned}$$

Case 2b: ($y_1 = y_2$):

$$\begin{aligned} & y_1||z_1 < y_2||z_2, y_1 = y_2 \\ \Rightarrow & z_1 < z_2 \\ & x_1 = x_2, y_1 = y_2, z_1 < z_2 \\ \Rightarrow & x_1||y_1 = x_2||y_2, z_1 < z_2 \\ \Rightarrow & \langle x_1||y_1, z_1\rangle \prec \langle x_2||y_2, z_2\rangle \end{aligned}$$

The proof of the opposite direction is similar. □

Lemma 1 essentially states that for ORDER BY clause, the sort order remains intact even though we shift the bits across the columns.

[Example Ex1] Figure 2b shows an example based on query $Q1$. With code massaging, the two columns nation_name (10-bit) and ship_date (17-bit) are "stitched" together as one 27-bit column. Concretely, the code massaging process first left-shifts ($\ll$) column nation_name by 17 bits, and then carries out a bitwise-OR ($\vee$) with column ship_date to form the 27-bit new column. From the figure, we see that we can sort the 27-bit column using one round, and by Lemma 1 we see that the ordered list of object identifiers *is the same as* the one obtained through two rounds of sorting in Figure 2a.

Figure 3a shows the performance results for this example (experimental setting is given in Section 6). In this experiment and the rest of this section, we generated $N = 2^{24}$ tuples and a w-bit column with 2^{13} distinct values[3] uniformly distributed on a $[0, 2^w - 1]$ domain. Since the steps after sorting are the same no matter code massaging is used or not (e.g., see Step ④/③ in Figure 2a/b), we only report the running time up to the point where all sortings are done. We use the notation R_i: $w/[b]$ to denote that a b-bit-bank SIMD-sort is used to sort a w-bit column in the i-th round of sorting. So, we denote the original column-at-a-time plan, which sorts the two columns nation_name (10-bit) and ship_date (17-bit) in two rounds, as plan

$$P_0^{Ex1} = \{R_1 : 10/[16], R_2 : 17/[32]\}$$

The code massage plan, which stitches the two columns into one by left-shifting 17-bits from the right column to the left, is denoted as $P_{\ll 17}^{Ex1}$={R_1: 27/[32]}.

From Figure 3a, we observe that $P_{\ll 17}^{Ex1}$ has improved the performance of such multi-column sorting by 44%. That attributes to not only the elimination of one round of sorting, but also the eliminations of one lookup operation and one scan operation (Steps ②a and ②b in Figure 2a). From Figure 3a, we also observe that the bitwise and memory operations required by code massaging (Step ① in Figure 2b) incur insignificant cost. □

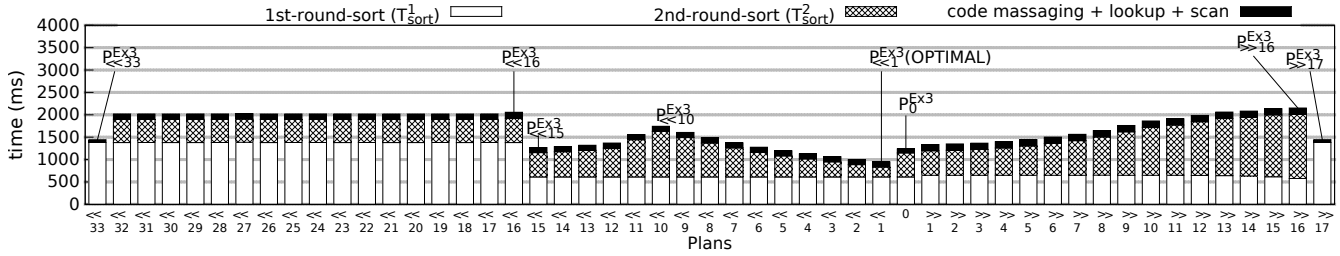
Code massaging is not as simple as always stitching all input columns together. One obvious example is that we cannot stitch all input columns together when their total code width W exceeds 64 bits, the maximum bank width of AVX2. Another example is as follows.

[Example Ex2] Figure 3b shows the performance results of a multi-column sorting on a 15-bit column and a 31-bit column. We see that a reckless stitch of the two columns as a 46-bit column would result in a degraded performance. Originally, the two columns could be sorted by one 16-bit SIMD-sort and one 32-bit SIMD-sort (Plan $P_0^{Ex2} = \{R_1 : 15/[16], R_2 : 31/[32]\}$). Stitching them together (Plan $P_{\ll 31}^{Ex2} = \{R_1 : 46/[64]\}$) would require one 64-bit SIMD-sort instead. 64-bit SIMD-sort has a relatively weaker data parallelism. That outweighs the benefit of eliminating one round of sorting, explaining why P_0^{Ex2} is better than $P_{\ll 31}^{Ex2}$ for this specific instance. □

Code massaging actually involves a huge space of plan choices in addition to the original column-at-a-time plan and the simple "stitch-all" plan. In the following, we illustrate that with two more examples.

[Example Ex3] This example is a multi-column sorting on a 17-bit column and a 33-bit column. Figure 4a shows the performance of all possible massage plans for that. The plans $P_{\ll 33}^{Ex3}$ and $P_{\gg 17}^{Ex3}$ at the two extremes are referring to the same "stitch-all" plan: $P_{\ll 33}^{Ex3}$

[3] If $w < 13$, then 2^w distinct values are generated instead.



(a) Time breakdown of various code massage plans

Shifted Bits	$P^{Ex3}_{\ll 32} \sim P^{Ex3}_{\ll 16}$	$P^{Ex3}_{\ll 15}$	$P^{Ex3}_{\ll 13}$	$P^{Ex3}_{\ll 11}$	$P^{Ex3}_{\ll 10}$	$P^{Ex3}_{\ll 2}$	$P^{Ex3}_{\ll 1}$	P^{Ex3}_{0}	$P^{Ex3}_{\gg 1}$	$P^{Ex3}_{\gg 10}$	$P^{Ex3}_{\gg 16}$
num. SIMD-sort (N_{sort})	2.0e06	2.1e06	2.9e06	4.4e06	4.6e06	**31756**	**15878**	8192	7235	64	2
num. groups (N_{group})	1.45e07	1.43e07	1.31e07	0.96e07	0.67e07	**31756**	**15878**	8192	7235	64	2
avg. group size ($\overline{N^i_{code}}$)	1.15	1.17	1.27	1.73	2.47	**528**	**1056**	2048	2318	0.26e6	0.84e6

(b) Factors in first round sorting that influence the time cost of second round sorting T^2_{sort}

Figure 4: [Example Ex3] **`ORDER BY` 17-bit-column, 33-bit-column**

shows the result of shifting all bits of the 33-bit column to the left and forming a single 50-bit super column while $P^{Ex3}_{\gg 17}$ shows the result of shifting all bits of the 17-bit column to the right and forming the same 50-bit super column. Both plans require only one round of 64-bit SIMD-sort. From Figure 4a, we observe that $P^{Ex3}_{\ll 33}$ and $P^{Ex3}_{\gg 17}$ are slightly inferior to the original column-at-a-time plan P^{Ex3}_0 for the same reason that we explained in Example Ex2.

Figure 4a indeed contains observations that are more intriguing. First, we see that by left-shifting one bit from the original plan P^{Ex3}_0={R_1: 17/[32], R_2: 33/[64]}, the optimal plan $P^{Ex3}_{\ll 1}$={R_1: 18/[32], R_2: 32/[32]} is found. $P^{Ex3}_{\ll 1}$ is superior in terms of using 32-bit SIMD-sort in both rounds when compared with P^{Ex3}_0.

Second, as we further left-shift the bits from $P^{Ex3}_{\ll 1}$, we observe a time hill with plan $P^{Ex3}_{\ll 10}$={R_1: 27/[32], R_2: 23/[32]} as the peak, plan $P^{Ex3}_{\ll 15}$={R_1: 32/[32], R_2: 18/[32]} as the left tail, and the optimal plan $P^{Ex3}_{\ll 1}$ as the right tail. We explain that starting from the right tail, i.e., $P^{Ex3}_{\ll 1}$. Referring to Figure 4b, as we left-shift one bit from $P^{Ex3}_{\ll 1}$ and result in plan $P^{Ex3}_{\ll 2}$={R_1: 19/[32], R_2: 31/[32]}, that would:

(a) *increase the number of groups* N_{group} formed by the tied values in the first round from 15878 to 31756, since the width of the first column has increased by one and *more distinct values* are found in that column;

(b) *increase the number of SIMD-sort invocations* N_{sort} in the second round from 15878 to 31756, because each group formed in the first round would invoke one SIMD-sort;

(c) *decrease the number of codes for sorting* N^i_{code} *in each group* i in the second round from 1056 to 528, because now there are more groups.

Therefore, the time cost T^k_{sort} of the k-th round of sorting can be captured by:

$$T^k_{sort} = \sum_{i=1}^{N_{sort}} T_{sort}(N^i_{code}, b) \quad (1)$$

where $T_{sort}(N^i_{code}, b)$ is the time spent by a b-bit SIMD-sort to sort N^i_{code} codes. Assume the implementation of merge-sort with sorting-network kernel [5] is used, then:

$$T_{sort}(N^i_{code}, b) = \mathcal{C}_{overhead} + T_{mergesort}(N^i_{code}, b) \quad (2)$$

where $\mathcal{C}_{overhead}$ is the overhead of memory allocation and initialization of a merge-sort function call, and $T_{mergesort}(N^i_{code}, b)$ is the cost of running a SIMD-enabled merge-sort.

So, following the discussion above, $P^{Ex3}_{\ll 2}$ has a *larger* N_{sort} but a *smaller* N^i_{code} when compared with $P^{Ex3}_{\ll 1}$. That means $P^{Ex3}_{\ll 2}$ has to carry out *more SIMD-sort* in round 2 but each SIMD-sort in that round *has a smaller input size*. With the existence of $\mathcal{C}_{overhead}$ per SIMD-sort, the increase of N_{sort} would increase the overall overhead in T^2_{sort}, and that outweighs the decrease of per-merge-sort running time caused by the decrease of N^i_{code}. That explains the uphill from $P^{Ex3}_{\ll 1}$ to the peak $P^{Ex3}_{\ll 10}$.

The downhill from the peak $P^{Ex3}_{\ll 10}$ to the left-tail $P^{Ex3}_{\ll 15}$ can be explained similarly, but in a different angle. Referring to Figure 4b, we observe a *decrease* in the number of SIMD-sort (N_{sort}) in round 2 starting from $P^{Ex3}_{\ll 11}$. That is because as we continue to shift more bits to the left, more groups are generated in the first round, and each group has fewer tuples. Eventually, some groups would contain just one tuple. Notice that *groups with one tuple do not require sorting*. So, when we proceed from $P^{Ex3}_{\ll 10}$ to $P^{Ex3}_{\ll 15}$, while N_{group} would increase, more singleton groups are generated, causing fewer SIMD-sorts are called (i.e., N_{sort} would decrease). That explains the downhill from the peak $P^{Ex3}_{\ll 10}$ to $P^{Ex3}_{\ll 15}$. In Figure 4a, T^1_{sort}, the time cost of the first round of sorting, remain steady throughout the whole hill because all plans between the left-tail $P^{Ex3}_{\ll 15}$ and right-tail $P^{Ex3}_{\ll 1}$ use the same 32-bit SIMD-sort to sort the same amount $N = 2^{24}$ of codes in the first column.

Let us move forward to observe the consequence of further left-shift one more bit from the left-tail $P^{Ex3}_{\ll 15}$ ={R_1: 32/[32], R_2: 18/[32]} of the hill to $P^{Ex3}_{\ll 16}$={R_1: 33/[64], R_2: 17/[32]}. There, we observe an increase in cost. That is because $P^{Ex3}_{\ll 16}$ has to use a 64-bit SIMD-sort in place of a 32-bit SIMD-sort to sort a 33-bit column in the first round. From $P^{Ex3}_{\ll 16}$ to $P^{Ex3}_{\ll 17}$={R_1: 34/[64], R_2: 16/[16]} we observe a minor decrease in cost. That is due to the switch of using a 16-bit SIMD-sort in round 2.[4] The relatively stable performance between $P^{Ex3}_{\ll 17}$ and $P^{Ex3}_{\ll 32}$={R_1: 59/[64], R_2:

[4]In current generation of SIMD, i.e., AVX2, we find that 16-bit SIMD-merge-sort only outperforms its 32-bit counterpart slightly. That is because some SIMD functions available for 32-/64-bit banks are not available for 8-/16-bit banks. So, they have to be simulated with more primitive instructions. That is also the reason why we do not use 8-bit bank SIMD-merge-sort in this paper.

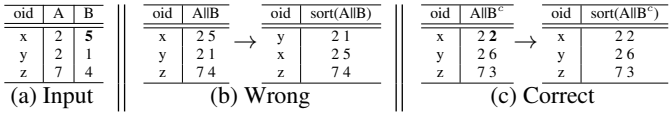


Figure 5: Code massaging `ORDER BY` **with attributes in different orders: (a) input and result of** `ORDER BY A ASC, B DESC`**; (b) without complement (wrong); (c) complement before stitch (correct)**

1/[16]} is due to the consistent use of 64-bit SIMD-sort in round 1 and 16-bit SIMD-sort in round 2.

Let us wrap up this long example by explaining the increasing cost from P_0^{Ex3} to $P_{\gg 16}^{\text{Ex3}}$, which involves *right*-shifting bits from the 17-bit first column to the 33-bit second column. We observe an increasing cost trend there because of the decrease of the number of groups N_{group} and thus the increase of average group size $\overline{N}_{code}^{i}$. When $\overline{N}_{code}^{i}$ reaches a certain value, the increase in $\overline{N}_{code}^{i}$ manifests the cost $T_{mergesort}(\overline{N}_{code}^{i}, b)$, which has $\Theta(\overline{N}_{code}^{i} \log \overline{N}_{code}^{i})$ complexity, dominates $\mathcal{C}_{overhead}$. That explains the increasing trend from P_0^{Ex3} to $P_{\gg 16}^{\text{Ex3}}$. □

[Example Ex4] This last example is a multi-column sorting on two 48-bit columns, which contains another important observation. The original column-at-a-time plan P_0^{Ex4}={R_1: 48/[64], R_2: 48/[64]} requires two rounds of sorting. Figure 3c shows the performance of P_0^{Ex4} and a code massaged plan, $P_{32\times 3}^{\text{Ex4}}$={$R_1$: 32/[32], R_2: 32/[32], R_3: 32/[32]}. It shows that the time to sort two 48-bit columns can be reduced by *increasing the number of rounds*! In this case, the performance improvement is brought by the better utilization of SIMD parallelism in three rounds, compared with a waste of 64–48=16 bits in two rounds of sorting. That implies code massaging should also consider increasing the number of sorting rounds. □

We close this section by presenting four more aspects about code massaging.

First, unlike `ORDER BY`, multi-column sorting incurred by `GROUP BY` and `PARTITION BY` requires no specific sorting sequence among the columns. That is, while the result of `ORDER BY A, B` is different from the result of `ORDER BY B, A`, the results between `GROUP BY A, B` and `GROUP BY B, A` are the same. `PARTITION BY` exhibits the same property as with `GROUP BY` in this aspect. Therefore, the plan space of multi-column sorting incurred by `GROUP BY` and `PARTITION BY` could be $m!$ times larger than the plan space incurred by `ORDER BY` (where m is the number of attributes involved in the clause).

Second, it is legitimate to massage the bits across *non-adjacent* columns for `GROUP BY` and `PARTITION BY` clauses. For example, consider a query with `GROUP BY A, B, C` and a tuple $\langle a_1||a_2, b, c_1||c_2\rangle$. The end results would remain correct even though the tuple is massaged as $\langle a_1||c_1||a_2, b, c_2\rangle$. Nevertheless, we do not consider this option in this paper to avoid space explosion.

Third, code massaging needs an extra step if the attributes in an `ORDER BY` clause specify different sorting orders. Consider a query with `ORDER BY A ASC, B DESC` that ensures all tuples are sorted in ascending order according to column A but sorted in descending order according to column B for tied values. Code massaging has to carry out a complement on column B before it stitches them together. Figure 5 illustrates that with an example. The input data is listed in Figure 5a. Coincidentally, the result of `ORDER BY A ASC, B DESC` is identical to its input. Prior to stitching the two columns together, code massaging has to carry out a complement operation on column B first (Figure 5c)[5]. Otherwise, an incorrect result would be obtained (Figure 5b).

Lastly, we note that the code massaging process is very lightweight because the column access pattern during code massaging is *highly sequential* and *branchless*, allowing efficient superscalar execution and cache line pre-fetching. We also note that code massaging can easily support multi-threading — we can partition the input columns evenly and each thread massages partitions from every column independently.

4. COST MODEL

In this section, we present a cost model that estimates the execution time of an instance of multi-column sorting problem under a specific code massage plan P. The cost model takes as inputs a code massage plan P and basic statistics about the data such as the number of tuples (i.e., number of codes per column) N, the column width w_i, and the value distribution of a column (e.g., a histogram). The cost model could be used by any cost-based plan search algorithm in quest of a low-cost code massaging plan.

Cost modeling in main-memory databases is still an active topic. One popular direction is to develop cache-aware cost models (e.g., [32, 15]) that estimate the number of cache misses of a specific database operation. Cache misses are highly correlated with the total CPU cycles in database access methods because the primary action of these access methods is to retrieve values from memory [15]. Multi-column sorting, however, includes not only data access but also computational demanding sorting operations. Recent work shows that a *calibrated* cost model [16, 40] can provide good estimates of query execution time. Therefore, we provide a cache-aware cost model that captures both computational and data access costs, with values of various architectural-dependent parameters (e.g., the latency of a cache miss) calibrated based on running controlled experiments on the column-store hardware platform.

Given a code massage plan P and basic statistics about the data, the cost model aims to compute, T_{mcs}, the CPU time to execute multi-column sorting according to P. Referring to Section 3 and Figure 2, T_{mcs} is composed of four major subcosts:

(a) **Lookup** (T_{lookup}): Time spent on reordering a column based on another column's sorting order (e.g., Step ②a in Figure 2a);

(b) **Code massaging** ($T_{massage}$): Time spent on massaging the input columns to form new columns (e.g., Step ① in Figure 2b);

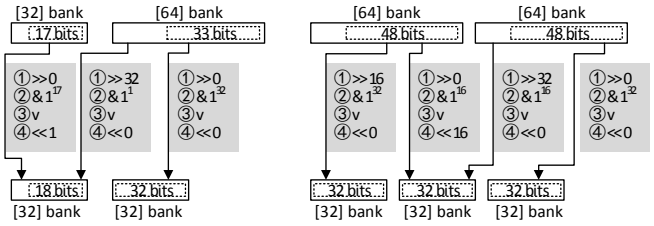
(c) **SIMD-sort** (T_{sort}): Time spent on sorting N codes by SIMD-sort (e.g., Step ② in Figure 2b);

(d) **Scan** (T_{scan}): Time spent on scanning a column to form groups based on the tied values (e.g., Step ②b in Figure 2a);

In the following, we elaborate on each subcost:

[Estimating T_{lookup}] T_{lookup} gives the cost of using the lookup operator to reorder a w-bit column C with N codes. This lookup process is essentially issuing N random accesses to the column. Let $size(w)$ be $2^{\lceil log_2 \lceil w/8 \rceil \rceil}$, which gives the size (in bytes) of the smallest data type that can hold a code in C, e.g., $size(15) = 2$ (`int16`) and $size(17) = 4$ (`int32`). Then, $N \cdot size(w)$ is the memory footprint of column C. Further, let M_{LLC} be the size (in bytes) of the last level cache, which is given by the hardware specification. So, we model the *cache hit ratio* as $\frac{M_{LLC}}{N \cdot size(w)}$. Let $\mathcal{C}_{cache}$ be the access latency of a data item in cache (usually in tens

[5]Example of complement: let $x.B = 5 = (101)_2$, so complement of $x.B$ is: $x.B^c = (101)_2^c = (010)_2 = 2$.



(a) $P^{\mathsf{Ex3}}_{\ll 1}$ (b) $P^{\mathsf{Ex4}}_{32\times 3}$

Figure 6: Code Massaging Implementation

of cycles) and $\mathcal{C}_{mem}$ is the access latency of a data item in memory (usually in hundreds of cycles), we can therefore model T_{lookup} as:

$$T_{lookup} = N \cdot \big(\mathcal{C}_{cache} \cdot \frac{M_{LLC}}{N \cdot size(w)} + \mathcal{C}_{mem} \cdot (1 - \frac{M_{LLC}}{N \cdot size(w)})\big) \quad (3)$$

which is essentially the number of cycles of random accessing N items with a cache hit ratio of $\frac{M_{LLC}}{N \cdot size(w)}$.

To obtain the values of $\mathcal{C}_{cache}$ and $\mathcal{C}_{mem}$, we use a linear system based calibration method similar to [40]. In this method, we calibrate the values of $\mathcal{C}_{cache}$ and $\mathcal{C}_{mem}$ simultaneously instead of micro-benchmark them individually. This approach has the advantage of calibrating parameters that are correlated. Otherwise, it would be difficult to micro-benchmark the cost of memory access without cache access.

The calibration process is as follows. We generated an array of N_{cal} w-bit integers and an array of random permuted oid's. Then we carried out the lookup procedure on the array based on the oid array and measured the execution time $T_{measured}$. After that, we plugged $T_{measured}$, N_{cal}, w along with M_{LLC} (obtained from the hardware specification) into T, N, w and M_{LLC} of Equation 3, respectively, to form an instantiation of the equation containing two unknowns: $\mathcal{C}_{cache}$ and $\mathcal{C}_{mem}$. Since there are two unknowns, we need two instantiations of the equation in order to solve them. So, we used two different N_{cal} values and generated two instantiations of Equation 3 (with two different $T_{measured}$), and then solved $\mathcal{C}_{cache}$ and $\mathcal{C}_{mem}$ as a linear system. By default, we used two N_{cal} values that lead to a cache hit ratio (i.e., $\frac{M_{LLC}}{N \cdot size(w)}$), of 0.1 and 0.9, respectively. Alternatively, we can use more than two N_{cal} values to generate more than two instantiations of Equation 3 and use statistical methods (e.g., linear regression) to obtain the best fit values.

[Estimating $T_{massage}$] $T_{massage}$ gives the cost of massaging the input columns according to a specific code massage plan P. The code massaging process, in a specific sense, consists of multiple invocations of a *four-instruction program (FIP)*: (1) shift ($\ll$ / $\gg$), (2) mask, (3) bitwise-OR ($\vee$), and (4) shift ($\ll$ / $\gg$).[6]

Figure 6 illustrates the four-instruction program (FIP) in the code massaging process for plan $P^{\mathsf{Ex3}}_{\ll 1}$={R_1: 18/[32], R_2: 32/[32]} in example Ex3 and the code massage plan $P^{\mathsf{Ex4}}_{32\times 3}$={$R_1$: 32/[32], R_2: 32/[32], R_3: 32/[32]} in example Ex4. We see that the former goes through that FIP *three* times whereas the latter goes through that FIP *four* times. Let $\mathcal{C}_{massage}$ be the number of CPU cycles of executing an FIP. Then, we can model $T_{massage}$ as:

$$T_{massage} = \mathcal{I}_{FIP} \cdot \mathcal{C}_{massage} \cdot N \quad (4)$$

where $\mathcal{I}_{FIP}$ is the number of invocations of the FIP. $\mathcal{I}_{FIP} = 3$ for $P^{\mathsf{Ex3}}_{\ll 1}$ and $\mathcal{I}_{FIP} = 4$ for $P^{\mathsf{Ex4}}_{32\times 3}$. The value of $\mathcal{I}_{FIP}$ is the number of elements in the union of two prefix sum sequences:

$$\mathcal{I}_{FIP} = |\{s_1, s_2, ...\} \cup \{s'_1, s'_2, ...\}|$$

where $\{s_1, s_2, ...\}$ is the prefix sum of the widths w_1, w_2, ... of the input columns involved, and $\{s'_1, s'_2, ...\}$ is the prefix sum of the widths w'_1, w'_2, ... of the columns after massaging. For plan $P^{\mathsf{Ex3}}_{\ll 1}$={R_1: 18/[32], R_2: 32/[32]} that sorts a 17-bit column and a 33-bit column in example Ex3:

$$\mathcal{I}_{FIP} = |\{17, 50\} \cup \{18, 50\}| = |\{17, 18, 50\}| = 3$$

To obtain the value of $\mathcal{C}_{massage}$, we carried out calibration based on generating N_{cal} random codes using the massage plans in Examples Ex1 to Ex4[7]. We measured the average time of executing code massaging according to selected code massage plan and divide that by $N_{cal} \cdot \mathcal{I}_{FIP}$ to get an estimation of $\mathcal{C}_{massage}$.

[Estimating T_{sort}] T_{sort}, or more specifically, $T_{sort}(N, b)$, is the time spent by a b-bit SIMD-sort to sort N codes, captured by Equation 2. The constant $\mathcal{C}_{overhead}$ in Equation 2 is the number of CPU cycles spent on initial function call and memory allocation, and how to calibrate this value would be discussed after we have introduced the other constants. The cost $T_{mergesort}(N, b)$ is specific to the implementation. Assume the implementation of merge-sort with sorting-network kernel [5] is used, then $T_{mergesort}(N, b)$ can be represented as:

$$\begin{aligned} &T_{mergesort}(N, b) = \\ &\quad T_{in\text{-}register}(N,b) + T_{in\text{-}cache}(N, b) + T_{out\text{-}of\text{-}cache}(N, b) \end{aligned} \quad (5)$$

Specifically, the merge-sort implementation in [5] is composed of three phases:

(a) **In-register sorting** ($T_{in\text{-}register}$): Using SIMD instructions with bank size b and S-bit registers, in-register sorting reads in $(S/b)^2$ codes each time, executes S/b sorting networks in parallel, and generates as outputs S/b sorted runs. Each sorted run contains S/b values, and each value is $b/8$ bytes. We call each sorted run as an *in-register sorted run* because it has a size of $S/8$ in bytes, which fits into a (SIMD) CPU register. So, if we denote $\mathcal{C}^b_{sort\text{-}network}$ as the number of cycles of running in-register sort per code under bank size b, then

$$T_{in\text{-}register}(N, b) = \mathcal{C}^b_{sort\text{-}network} \cdot N \quad (6)$$

(b) **In-cache merging** ($T_{in\text{-}cache}$): This phase recursively merges in-register sorted runs from the first phase to form larger sorted runs that fit into half L2 cache. The merging is carried out by bitonic merge networks, which can also be parallelized by SIMD. Let M_{L2} be the size (in bytes) of the L2 cache. Each *in-cache merged run* is therefore $0.5M_{L2}$ in bytes and holds $0.5M_{L2}/(b/8)$ values. As each in-register sorted run holds S/b values, each in-cache merged run is therefore merged from $\frac{0.5M_{L2}/(b/8)}{(S/b)}$ in-register sorted runs in $\lceil \log_2(\frac{0.5M_{L2}/(b/8)}{(S/b)}) \rceil$ passes. So, if we denote $\mathcal{C}^b_{in\text{-}cache\text{-}merge}$ as the number of cycles spent on merging $\frac{0.5M_{L2}/(b/8)}{(S/b)}$ in-register sorted runs divided by the number of values being merged ($0.5M_{L2}/(b/8)$), then we have:

$$T_{in\text{-}cache}(N, b) = \mathcal{C}^b_{in\text{-}cache\text{-}merge} \cdot N \quad (7)$$

[6]For `ORDER BY` clause with different sorting orders, the program needs to do complement as well.

[7]For Example Ex3, the optimal plan $P^{\mathsf{Ex3}}_{\ll 1}$ is used for calibration.

(c) **Out-of-cache merging** ($T_{out\text{-}of\text{-}cache}$): This final phase merges the in-cache merged runs from the second phase recursively until a single sorted run is produced. In order to reduce cache misses, instead of a binary merge, a merge tree with a larger fanout F is used. Since the second phase will generate $\frac{Nb/8}{0.5M_{L2}}$ in-cache merged runs, merging them requires $\lceil \log_F \frac{Nb/8}{0.5M_{L2}} \rceil$ passes. So, if we denote $\mathcal{C}^b_{out\text{-}of\text{-}cache\text{-}merge}$ as the number of cycles per tuple of merging F in-cache merged runs, we have:

$$T_{out\text{-}of\text{-}cache}(N,b) = \mathcal{C}^b_{out\text{-}of\text{-}cache\text{-}merge} \cdot N \cdot \lceil \log_F \frac{Nb/8}{0.5M_{L2}} \rceil \quad (8)$$

Calibrating the constants We calibrate $\mathcal{C}_{overhead}$, $\mathcal{C}^b_{sort\text{-}network}$, $\mathcal{C}^b_{in\text{-}cache\text{-}merge}$ and $\mathcal{C}^b_{out\text{-}of\text{-}cache\text{-}merge}$ in a batch by treating them as four unknowns to a linear system. By putting Equations 1, 2, 5, 6, 7, and 8 together, we obtain a linear equation for T^k_{sort} containing the above four unknowns. We generated an array of N_{cal} b-bit random integers and an array of N_{cal} entries to hold the group information. After that we executed the sorting algorithm multiple times based on data with different N_{group} values (1, 2, 4, ..., N_{cal}) to obtain different T^k_{sort} values. Then we can solve the linear system with the four constants as the unknowns. By default, we set N_{cal} such that $N_{cal} \cdot b/8$ is 100X in size of the last level cache and uniformly distributed the integers to N_{group} groups. Finally, the above steps are repeated for b equals to 16, 32 and 64, respectively.

[Estimating T_{scan}] T_{scan} gives the cost of a sequential scan of a sorted column to extract the grouping information. Thus,

$$T_{scan} = \mathcal{C}_{scan} \cdot N \quad (9)$$

where $\mathcal{C}_{scan}$ is the average cost of sequentially scanning N values and filling in the grouping information to another array. Since modern processors are very efficient in hiding *sequential* memory access latency through prefetching, here we do not distinguish between cache hit and cache miss as we do for estimating T_{lookup} (which incurs *random* memory access).

To obtain the value of $\mathcal{C}_{scan}$, we generated an array of N_{cal} sorted integers and pre-allocated another array with N_{cal} entries to hold the grouping information. Then we executed the scan procedure to scan the array and fill the grouping information array and measured the average cycles per row. By default, we set N_{cal} such that the data size is 100X in size of the last level cache.

5. PLAN SEARCH

We begin this section with a quantification of the size of the whole code massage plan space $\mathcal{P}$. Let $W = \sum_1^m w_i$ be sum of the column widths $w_1, w_2, \ldots, w_m$ of m input columns involved. The code massage plan space $\mathcal{P}$ is equivalent to the integer compositions problem [19], i.e., enumeration of all ways of writing W as the sum of a sequence of (strictly) positive integers. Hence, $|\mathcal{P}| = 2^{W-1}$. For multi-column sorting problems incurred by `GROUP BY` and `PARTITION BY`, the sorting order among the columns does not matter. So, their plan spaces are $m!$ times larger, although m is usually a small number in real workload (e.g., the largest m in TPC-H is 7).

In the following, we present our round-based greedy plan search algorithm. By round-based we mean the algorithm searches for the optimal plan from candidate plans that are restricted to have only one round of sorting, then from candidate plans that are restricted to have only two rounds, and so on. By greedy, we mean, given a restricted plan space (e.g., when considering plans only with three rounds), the algorithm greedily assigns a certain column width to the first round, then to the second round, and so on.

Bounding the number of rounds Normally, our plan search algorithm should consider candidate plans up to W rounds of sorting for not missing the optimal plan (remember Example Ex4?). So, consider a multi-column sorting instance, whose original plan $P_0 = \{R_1 : 17/[32], R_2 : 30/[32], R_3 : 12/[16]\}$ requires three rounds of sorting, the plan $P_{1\times 59} = \{R_1 : 1/[16], R_2 : 1/[16], \ldots, R_{59} : 1/[16]\}$ that requires $W = 17 + 30 + 12 = 59$ rounds is actually viable. In the following, we show that plans with more than $\lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$ rounds (where b_{min} is minimum bank size used by the available SIMD-sort implementations) are dominated by plans with fewer than or equal to $\lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$ rounds in cost. Therefore, for the example above, our plan search algorithm would only consider plans that require at most $\lfloor \frac{2\cdot(59-1)}{16} \rfloor + 1 = 8$ rounds of sorting, when we have 16/32/64-bit SIMD-sort implementations available.[4]

LEMMA 2. *Plans with more than $\lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$ rounds are dominated by plans with fewer than or equal to $\lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$ rounds in cost.*

PROOF. Consider a candidate/original plan P with K rounds (where $K \leq W$). If we stitch two columns C_i and C_j in P, we get a new plan P' with $K-1$ rounds and a column C' of size $w' = w_i + w_j$. When (a) C_i and C_j are adjacent, i.e., $C_j = C_{i+1}$ and (b) $w' \leq b_i$, where b_i is the minimum bank size that is enough to hold C_i, we can conclude that P' is better than P in cost [8]. As an example, consider plan $P_{1\times 59}$ above, whose $K = 59$. If we stitch columns C_1 and C_2 to be column C', we get a new plan P' with K–1=58 rounds. Note that as $w' = 2 \leq b_1 = 16$, so P' would use the same b_1-bit SIMD-sort to sort C' but with one round fewer than $P_{1\times 59}$. Therefore, P' must be better than $P_{1\times 59}$ in cost. In other words, for each candidate plan P with K rounds, if we are able to find two columns C_i and C_j that satisfy (a) and (b), that implies all plans with K rounds can be dominated by some plans with $K-1$ rounds. Hence, all candidate plans with K rounds can be pruned.

In contrast, consider $P_0 = \{R_1 : 17/[32], R_2 : 30/[32], R_3 : 12/[16]\}$ above, whose $K = 3$. If we stitch columns C_1 and C_2 to be column C', we get a new plan P' with $K - 1 = 2$ rounds. However, as $w' = 17 + 30 = 47 \not\leq b_1 = 32$, column C' has to be sorted using a 64-bit SIMD-sort, which has a lower degree of parallelism. Likewise when we try to merge C_2 and C_3. Therefore, P' may not be better than P_0 in this case. In other words, plans with $K = 3$ rounds should be retained for the sake of not discarding any competitive plans.

Summing up the above, when considering plans with K rounds, denoted as $\mathcal{P}^K$, if we stitch any two adjacent columns in a plan $P \in \mathcal{P}^K$ (condition (a) holds) and the resulting plan P' may not be better than P, then we shall retain plans with K rounds in the search space for costing. The only reason that the resulting plan P' may not be better than P when condition (a) holds is: *condition (b) is violated*. So, that gives us:

$$\begin{cases} w_1 + w_2 & > b_1 \\ w_2 + w_3 & > b_2 \\ \ldots & \ldots \\ w_{K-1} + w_K & > b_{K-1} \end{cases}$$
$$\implies 2\sum_{i=1}^{K} w_i - w_1 - w_K > \sum_{i=1}^{K-1} b_i$$

[8] A formal proof of this is given as Property 1 in Appendix A.

$$
\begin{aligned}
2W - w_1 - w_K &> \sum_{i=1}^{K-1} b_i \\
2W - 2 \geq 2W - w_1 - w_K &> \sum_{i=1}^{K-1} b_i \quad \because w_1 \geq 1, w_K \geq 1 \\
2W - 2 &\geq \sum_{i=1}^{K-1} b_i \\
2W - 2 > \sum_{i=1}^{K-1} b_i &\geq \sum_{i=1}^{K-1} b_{min} \quad \because \forall_i b_i \geq b_{min} \\
2W - 2 &> \sum_{i=1}^{K-1} b_{min} = (K-1)b_{min} \\
K &< \frac{2(W-1)}{b_{min}} + 1 \\
K &\leq \lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1 \qquad (10)
\end{aligned}
$$

Tracing backward, when $K \leq \lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$, condition (b) is violated, then we should retain plans with that value of K for the sake of not discarding any competitive plans. In other words, our round-based plan search algorithm can skip plans with more than $\lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$ rounds. □

Round-based enumeration and greedy costing Our round-base enumeration strategy is to first enumerate and cost candidate plans with $k = 1$ round, then candidate plans with $k = 2$ rounds, and so forth. The last set of candidate plans to be considered are those with $\lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$ rounds.

When enumerating candidate plans that are restricted to only k round(s) of sorting, it first partitions the subspace according to various bank size combinations. Then, for each subspace under a specific bank size combination, it assigns a_j-bits to the j-th bank based on minimizing T_{sort}^{j+1}, the time cost of the $(j+1)$-th round of sorting, starting from j equals 1, till j equals $k-1$; remaining bits are assigned to k-th round.

For example, consider $P_0 = \{R_1 : 17/[32], R_2 : 30/[32], R_3 : 12/[16]\}$. Our plan enumeration strategy would first enumerate candidate plans that are restricted to only one round of sorting, then to only two rounds, and finish until it has enumerated plans with $\lfloor \frac{2 \cdot (59-1)}{16} \rfloor + 1 = 8$ rounds. Specifically, when $k = 1$, only one valid bank size combination is available — the use of one 64-bit bank because $64 > W = 59 > 32$. Therefore, candidate plans are in the form of $\mathcal{P}^1 = \{R_1 : a_j/[64]\}$ and then the goal is to cost all valid plans in $\mathcal{P}^1$. In this example, the only valid plan is $\{R_1 : 59/[64]\}$, i.e., $a_j = 59$, so this plan would be costed. Next, our plan enumeration strategy would consider $k = 2$, i.e., plans with only two rounds of sortings. When $k = 2$, valid candidate plans $\mathcal{P}^2$ are in the form of:

$\{R_1 : a_1/[16], R_2 : a_2/[64]\}$ $\{R_1 : a_1/[32], R_2 : a_2/[32]\}$
$\{R_1 : a_1/[32], R_2 : a_2/[64]\}$

Note that candidate plans in the form of $\{R_1 : a_1/[16], R_2 : a_2/[16]\}$, $\{R_1 : a_1/[16], R_2 : a_2/[32]\}$, $\{R_1 : a_1/[32], R_2 : a_2/[16]\}$ are *invalid* because the two banks are insufficient to hold $W = 59$ bits. Candidate plans in the form of $\{R_1 : a_1/[64], R_2 : a_2/[16]\}$, $\{R_1 : a_1/[64], R_2 : a_2/[32]\}$, $\{R_1 : a_1/[64], R_2 : a_2/[64]\}$ are *pruned* for the same reason given by Property 1 of Lemma 2.

Given a valid bank size combination, our plan enumeration strategy would then further trim the search plan by considering only plans that are in compliance with the input instances. For example, consider plans in the form of $\{R_1 : a_1/[16], R_2 : a_2/[64]\}$, the plan $\{R_1 : 1/[16], R_2 : 63/[64]\}$ would not be enumerated because the two columns would be wider than $W = 59$ bits. So, for this example, the subspace consists of plans instantiated with $(a_1 = 1, a_2 = 58), (a_1 = 2, a_2 = 57), \ldots, (a_1 = 16, a_2 = 43)$. These 16 plans would be costed.

When $k \geq 3$, each subspace formed by a valid bank size combination alone has size $O(\binom{W-1}{k-1})$. To avoid plan explosion, we thus enumerate only the subspace with a chosen a_j value that gives the $(j+1)$-th round the lowest sorting cost T_{sort}^{j+1}.

Consider a specific subspace $\{R_1 : a_1/[16], R_2 : a_2/[16], R_3 : a_3/[16], R_4 : a_4/[16]\}$ when $k = 4$. The value of a_1, the number of bits to assigned to the $(j = 1)$-th bank, is determined based on minimizing T_{sort}^2, the time spent on the $(j+1)$-th round of sorting. Recall from Equation 1 that different a_1 values would lead to different N_{sort} and N_{code}^i values. So, among 16 different values for a_1 (from $1, 2, \ldots, 16$), the one that gives the lowest T_{sort}^2 would be greedily chosen as the value of a_1.

Assuming $a_1 = 5$ gives the lowest T_{sort}^2 , the enumeration would proceed to determine the value of a_2 recursively. Specifically, with $a_1 = 5$, the enumeration would focus on plans in the form of $\{R_1 : 5/[16], R_2 : a_2/[16], R_3 : a_3/[16], R_4 : a_4/[16]\}$ and choose a value for a_2 that minimizes T_{sort}^3, so on and so forth.

Finally, among plans that have been enumerated and costed, the one that minimizes T_{mcs} is chosen and returned.

Pseudocode and time-bound optimization Algorithm 1 depicts the pseudocode of the plan search algorithm. The pseudocode is self-explanatory and mainly summarizes what we have discussed

Algorithm 1 Plan Search Algorithm

Input: original column-at-a-time plan P_0
Input: cost model $T(\cdot)$
Input: input statistics (number of rows, etc.)
Input: time threshold ρ
Output: P^*: plan with the lowest cost

1: $W = \sum_i P_0.w_i$ ▷ Total num of bits
2: $P^* = P_0$ ▷ Initialize global optimal
3: Start stopwatch Ω
4: **for** $k = 1 \ldots \lfloor \frac{2(W-1)}{b_{min}} \rfloor + 1$ **do**
5: **for** each valid bank size combination $(b_1, \ldots, b_k) \in \mathcal{B}^k$ **do**
6: **if** $\Omega > \rho \cdot T_{mcs}(P^*)$ **then**
7: **return** P^* ▷ Time's up
8: $\hat{P} = \emptyset$ ▷ Initialize local optimal
9: **for** $i = 1 \ldots k-1$ **do**
10: **for** each possible a bits assigned to R_i **do**
11: Estimate T_{sort}^{i+1} given $\hat{P} \cup \{R_i : a/[b_i]\}$
12: **end for**
13: $\hat{a} = \text{argmin}_a \, T_{sort}^{i+1}$ given $\hat{P} \cup \{R_i : a/[b_i]\}$
14: $\hat{P} = \hat{P} \cup \{R_i : \hat{a}/[b_i]\}$ ▷ Greedy construction
15: **end for**
16: Assign the remaining bits to R_k
17: **if** $T_{mcs}(\hat{P}) < T_{mcs}(P^*)$ **then**
18: $P^* = \hat{P}$ ▷ Update global optimal
19: **end for**
20: **end for**
21: **if** handling `GROUP BY` or `PARTITION BY` **then**
22: Generate next permutation of P_0 then repeat Lines 4 – 20
23: **return** P^*

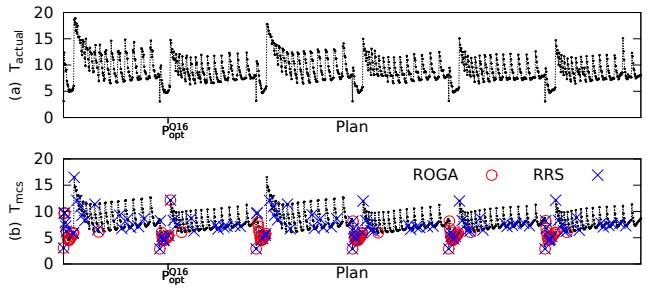


Figure 7: Q16 in TPC-H: (a) perfect cost model $\mathcal{A}_{16}$ based on exhaustively enumerating and executing all feasible plans; (b) our cost model. Both ROGA and RRS find the actual optimal plan P_{opt}^{Q16}.

Table 1: Cost Model and Plan Quality

Measure	TPC-H		TPC-H Skew		TPC-DS		Real Data	
	ROGA	RRS	ROGA	RRS	ROGA	RRS	ROGA	RRS
(a) $\overline{rank}$	4.8	110.8	5.2	72.7	6	47	8	43.2
(b) Best Rank	1	1	1	1	1	1	1	1
(c) Worst Rank	48	1206	58	576	53	223	36	169
Cost Model MRE	0.42		0.57		0.36		0.4	

above, so we do not give it a full walkthrough here. The only issue worth noticing is that we introduce a time threshold ρ (line 6) to the whole optimization process. One reason is that multi-column sorting, if accelerated by code massaging, could be executed *really* fast. Another reason is that the optimization would take a longer time when the total code width W is really very large. Therefore, we have to ensure that the searching process itself would not be a bottleneck. So, the algorithm starts a stopwatch at the beginning of the optimization process, and the algorithm terminates once the elapsed time Ω has exceeded a percentage ρ of the cost of the best plan so far. Nevertheless, since our algorithm is round-based (incrementing k one-by-one) and the optimal plans are usually those with small k values, such early stopping never discards any optimal plans in our experiments.

6. EXPERIMENTAL EVALUATION

The techniques used and proposed in this paper can be integrated into any main-memory column-store. The implementation can be done by (i) modifying the *storage manager* to support ByteSlice data layout [14]; (ii) adding four physical operators into the *query execution engine*: `ByteSlice-Scan` and `ByteSlice-Lookup` that respectively execute fast scans and lookups on ByteSlice data as proposed in [14]; `SIMD-Sort` that carries out (single-column) SIMD sorting as proposed in [5] with multi-threading support; and `Code-Massage` that carries out code massaging as depicted in Figure 6; (iii) modifying the *query optimizer* to include our cost model (Section 4), our plan search algorithm (Section 5), and WideTable query rewriting [31]. We note that changes to these components can be done in any typical relational column-stores. In this paper, we present results based on our own column-store prototype. Appendix B presents a reference implementation based on open-source MonetDB.

We run our experiments on two hardware platforms. The first is a server with a 2.60GHz Intel Xeon E5-2660 v3 10-core processor, and 32GB DDR3 memory. Each core has 32KB L1i cache, 32KB L1d cache and 256KB L2 unified cache. All cores share a 25MB L3 cache. The second is a PC with a 3.40GHz Intel i7-4770 quad-core processor, and 16GB DDR3 memory. Each core has 32KB L1i cache, 32KB L1d cache and 256KB L2 unified cache. All cores share an 8MB L3 cache. Both CPUs are based on Haswell microarchitecture which supports AVX2 instruction set. Due to space limit, we mainly present Xeon experimental results and but include i7 results when necessary.

All implementations are done in C++ and compiled using g++ 4.9 with optimization flag -O3. We use Intel Performance Counter Monitor [23] to collect the performance profiles. All experiments are run in a single process with a single thread by default. We discuss multi-thread experiments in Section 6.4.

We evaluate our techniques based on TPC-H, TPC-H skew [9] (skew factor $zipf = 1$), TPC-DS, and real data. There are 9 out of 22 TPC-H queries and 71 out of 99 TPC-DS queries require multi-column sorting. As TPC-H queries do not have `PARTITION BY` clauses, we complement that by selecting TPC-DS queries that include `PARTITION BY` clauses. Among 12 eligible TPC-DS queries, 8 of them include moving window, which is not yet supported by our prototype. So, we can only include 4 TPC-DS queries in the experiments. A scale factor of one is used by default. The real data is the Airline Origin and Destination Survey dataset released by Bureau of Transportation Statistics in US. The dataset is 4G in size and is publicly available at [1]. Table 4 and Table 5 in the Appendix list the table schema and the five queries that require multi-column sorting extracted from a real application on top of that dataset.

6.1 Quality of Plans

The effectiveness of code massaging depends on (i) the accuracy of the cost model and (ii) the quality of the plan search algorithm.

Recent works that focus on query latency prediction [2, 40] employ mean relative error (MRE) to measure *the accuracy of the cost model*. In this paper, we also report the MRE of our cost model for a query workload $\mathcal{Q}$. The last row of Table 1 shows that our cost model has MRE between 0.36 and 0.57 for all workloads, which is good enough even in the context of latency prediction [28, 40].

Define $P_{opt}^{Q_i}$ as the optimal plan for a query Q_i returned by a plan search algorithm based on our cost model. We report *the quality of a plan search algorithm for a workload* $\mathcal{Q}$ as

$$\overline{rank} = \frac{1}{|\mathcal{Q}|}\sum_{i=1}^{|\mathcal{Q}|} rank(P_{opt}^{Q_i}, \mathcal{A}_i)$$

where (i) $\mathcal{A}_i$ is the perfect cost model based on exhaustively enumerating all feasible plans for Q_i and measuring their actual running times (yes, it took us weeks to obtain $\mathcal{A}_i$'s); and (ii) $rank(P_{opt}^{Q_i}, \mathcal{A}_i)$ is the rank of $P_{opt}^{Q_i}$ with respect to $\mathcal{A}_i$. For example, if the plan $P_{opt}^{Q_i}$ chosen by a plan search algorithm is the second best plan based on $\mathcal{A}_i$, then $rank(P_{opt}^{Q_i}, \mathcal{A}_i) = 2$. In other words, if a plan search algorithm is always able to find the actual optimal plan for a given workload $\mathcal{Q}$ based on our cost model, then $\overline{rank} = 1$. So, while MRE focuses on the difference between the actual execution cost of a multi-column sorting instance T_{actual} and its estimated cost T_{mcs}, the metric $\overline{rank}$ focuses on the *reliability of the cost model and the plan search algorithm*.

Figure 7 shows an example of how $\overline{rank}$ complements MRE. Figure 7a plots the actual execution costs T_{actual} of all feasible plans for TPC-H query Q16, which has 3 attributes in its `GROUP BY` clause. Figure 7b plots the estimated costs T_{mcs} of all feasible plans based on our cost model. If we simply look at the MRE, our cost model reports an MRE of 0.42 for TPC-H workload (Table 1). However, we can see that our cost model indeed captures the

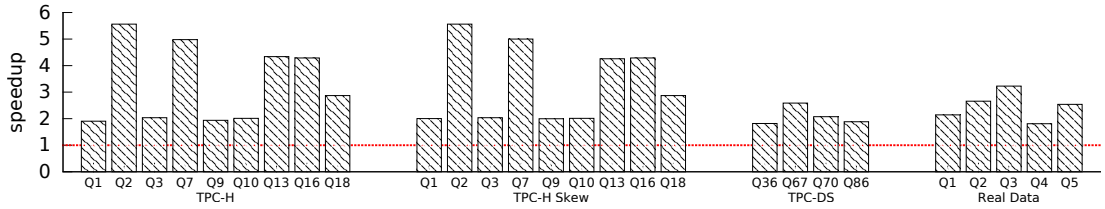


Figure 8: Speedup of Multi-column Sorting.

Table 2: Percentage of Time Used by ROGA in Multi-column Sorting

Workload	TPC-H									TPC-H Skew									TPC-DS				Real Data				
	Q1	Q2	Q3	Q7	Q9	Q10	Q13	Q16	Q18	Q1	Q2	Q3	Q7	Q9	Q10	Q13	Q16	Q18	Q36	Q67	Q70	Q86	Q1	Q2	Q3	Q4	Q5
%	0.002	0.09	0.07	0.04	0.06	0.1	0.07	0.01	0.1	0.004	0.08	0.08	0.05	0.05	0.1	0.06	0.02	0.1	0.09	0.1	0.05	0.01	0.08	0.02	0.01	0.09	0.07

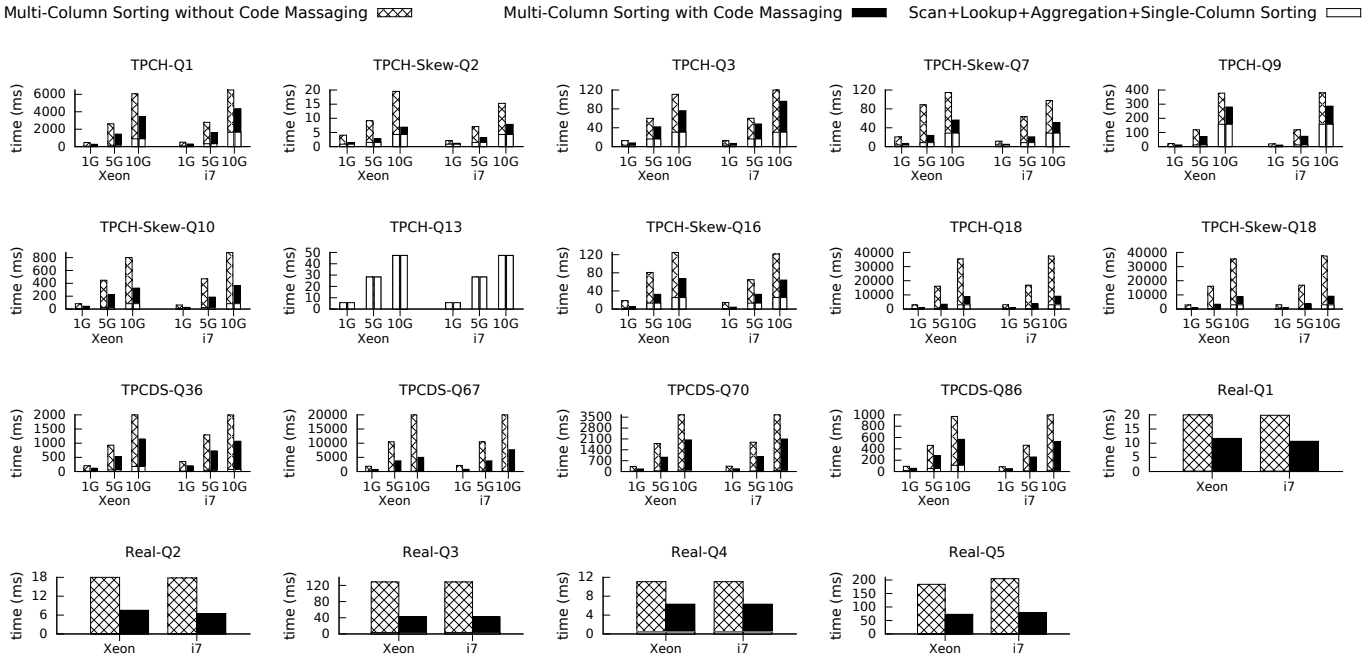


Figure 9: Query Execution Time (1G, 5G, and 10G data)

actual behavior very well. Figure 7b also shows all the plans (in circles ◯) enumerated by our round-based greedy algorithm (ROGA for short). For comparison, Figure 7b also shows all the plans (in crosses ×) enumerated by a recursive random search (RRS) algorithm [41] that searches the plan space based on our cost model. RRS is a powerful technique developed to solve black-box optimization problems. It first samples the subspace randomly to identify promising regions that contain the optimal setting with high probability. It then samples recursively in these regions which either move or shrink gradually to local-optimal settings based on the samples collected. RRS then restarts random sampling to find a more promising region to repeat the recursive search. RRS has been adopted as a plan search algorithm in a number of recent works (e.g., [17, 18]). For TPC-H query Q16, our cost model is actually reliable enough to allow both our plan search algorithm ROGA and RRS to find the actual optimal plan, i.e., $rank(P_{opt}^{Q_{16}}, \mathcal{A}_{16}) = 1$ for both ROGA and RRS, despite an MRE of 0.42 for the whole query workload is reported. In the experiments, the default time threshold ρ for ROGA is 0.1%, meaning ROGA would not spend more than 0.1% of the optimal plan (estimated) execution time to find the plan. For fairness, we stop RRS when ROGA stops.

Table 1 also reports the quality of the plan search algorithms in terms of rank. We see that ROGA consistently returns higher quality plans than RRS. For example, the $\overline{rank}$ of ROGA is 4.8 on TPC-H data (see row (a)), meaning the 5-th best actual plan is found for all involved TPC-H queries on average. In contrast, the $\overline{rank}$ of RRS is only 110.8 on TPC-H data. We notice that both ROGA and RRS can find out the actual optimal plan for some queries in the workloads (see row (b)). Worst cases happen when working on TPC-H skew data (see row (c)). Nevertheless, ROGA can find out the 58-th best actual plan. In contrast, RRS can only find the 576-th best actual plan in the same experiment.

6.2 Multi-Column Sorting Speedup

Figure 8 shows the speedup of multi-column sorting time by using code massaging for eligible TPC-H, TPC-DS, and real queries.

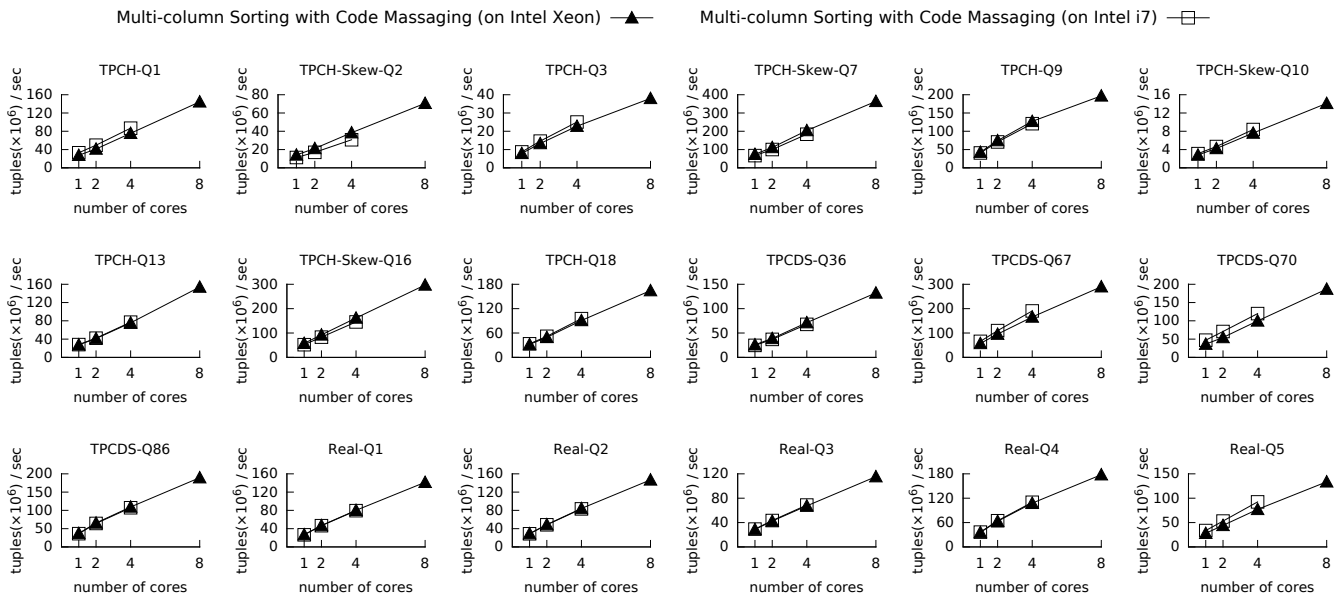


Figure 10: Throughput: Varying Number of Cores.

When code massaging is disabled, multi-column sorting is carried out using the column-at-a-time plan. When code massaging is enabled, multi-column sorting is carried out using the best plan among the column-at-a-time plan and a variety of enumerated code massage plans, returned by ROGA. From Figure 8, we observe that multi-column sorting with code massaging yields at least 1.8X speedup (real query Q4) and up to 5.5X speedup (TPC-H Q2). We also observe that the time used by ROGA to find a good code massage plan is negligible (see Table 2). Indeed, under $\rho = 0.1\%$, 22 out of all 27 queries (9 TPC-H queries on uniform and skewed data + 4 TPC-DS queries + 5 real queries) completed the whole plan search process before the deadline. In Appendix C, we present experiments of varying ρ values on different workloads and hardware platforms. We see that $\rho = 0.1\%$ is a sufficiently good threshold in general because it gives ROGA (usually more than) enough time to find a very high quality plan, without making itself as a bottleneck. In general, we recommend $\rho = 0.1\%$ based on our empirical results above. Finding ρ automatically is feasible and we discuss that in Appendix C.

6.3 Overall Query Speedup

Figure 9 shows the execution times of TPC-H, TPC-DS, and real queries on both Xeon and i7 CPUs. TPC-H and TPC-DS datasets have three scale factors 1, 5 and 10. In TPC-H, 9 queries Q1, Q2, Q3, Q7, Q9, Q10, Q13, Q16, Q18 require multi-column sorting. However, for space reasons, we only selectively present the results of Q1, Q3, Q9, Q13, Q18 on TPC-H *dbgen* data, Q2, Q7, Q10, Q16, Q18 on TPC-H skewed data, all 4 eligible queries on TPC-DS data, and all 5 eligible queries on real data.

From the experimental results, we observe that query execution with code massaging enables a query speedup of up to 4.7X (Q18; 5G; i7) on TPC-H, 4.7X (Q18; 10G; i7) on TPC-H skew, 4X (Q67; 10G; Xeon) on TPC-DS, and 3.2X (Q3; i7) on real data. TPC-H query Q13 is an exception. TPC-H Q13 has a `GROUP BY` clause on a single attribute `C_COUNT` and an `ORDER BY` clause on two attributes `CUSTDIST` and `C_COUNT`. During evaluation, the `GROUP BY` clause is first evaluated. So, the `ORDER BY` clause, which requires a multi-column sorting, is actually working on each individual group. In this case, the time percentage spent by multi-column sorting is actually insignificant (see Figure 1). Therefore, the 4.2X speedup brought by code massaging (see Figure 8 TPCH-Q13) is insignificant with respect to the whole query execution. Despite that, we observe from Figure 9 that code massaging consistently yields promising query speedup on data in different scales.

6.4 Varying Number of Cores

Figure 10 shows the throughput (in terms of the number of million tuples per second) of selected queries when code massaging is enabled with different number of threads. Each thread is pinned to a distinct physical core in the processor. Linear core/thread scalability is observed across all workloads and two different CPU models.

7. CONCLUSION

Recently, there is a resurgence of interest in main-memory analytic databases because of the large RAM capacity of modern servers (e.g., Intel Xeon E7 v2 servers can support 6TB of RAM) and the increasing demand for real-time analytic platforms. With the advent of recent fast scans and denormalization techniques, we observe that multi-column sorting would become a bottleneck for queries possessing multiple attributes in their `GROUP BY`, `ORDER BY`, or `PARTITION BY` clauses. Queries of that kind are not uncommon in real workloads. This paper therefore provides the first solution, namely, code massaging, to this emerging problem. Code massaging reduces the time of multi-column sorting by manipulating the bits across input columns so that the overall sorting time can be reduced through fewer sorting rounds and/or higher degree of SIMD data parallelism. Experiments on TPC and real workloads show that a main-memory column-store with code massaging can achieve up to 4.7X speedup in query execution. Our future work is to include radix-sort [11] into our study. The performance of in-memory radix-sort depends on the size (number of bits) of the radix, which is a parameter. Code massaging would allow a careful choice of the radix size when radix-sorting multiple columns, thereby improving the performance of multi-column sorting with a different flavor.

8. REFERENCES

[1] Airline Origin and Destination Survey Batabase. http://www.transtats.bts.gov/.

[2] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, 2012.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 2012.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Parallel Joins on Multi-Core. http://www.systems.ethz.ch/projects/paralleljoins.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 2014.

[6] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, et al. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 2012.

[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.

[8] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[9] S. Chaudhuri and V. Narasayya. Program for TPC-D data generation with skew, 2012.

[10] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 2008.

[11] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri. PARADIS: an efficient parallel algorithm for in-place radix sort. *PVLDB*, 2015.

[12] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 2010.

[13] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.*, 2012.

[14] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*, 2015.

[15] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. *PVLDB*, 2010.

[16] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst. (TODS)*, 2009.

[17] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 2011.

[18] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *ACM Symposium on Cloud Computing*, 2011.

[19] S. Heubach and T. Mansour. *Combinatorics of compositions and words*. CRC Press, 2009.

[20] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 2012.

[21] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, 2007.

[22] H. Inoue and K. Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment*, 2015.

[23] Intel. Intel Performance Counter Monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor/.

[24] Intel. Intel architecture instruction set extentions programming reference, 2013.

[25] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 2008.

[26] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2009.

[27] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 2011.

[28] J. Li, R. V. Nehme, and J. Naughton. Gslpi: A cost-based query progress indicator. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, 2012.

[29] Y. Li, C. Chasseur, and J. M. Patel. A padded encoding scheme to accelerate scans by leveraging skew. In *SIGMOD*, 2015.

[30] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, 2013.

[31] Y. Li and J. M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 2014.

[32] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *Knowledge and Data Engineering, IEEE Transactions on (TKDE)*, 2002.

[33] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *PVLDB*, 2009.

[34] Oracle Exalytics In-memory Machine. http://www.oracle.com/us/solutions/ent-performance-bi/business-intelligence/exalytics-bi-machine/overview/index.html.

[35] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, SIGMOD, 2015.

[36] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *SIGMOD*, 2014.

[37] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious simd sort. In *SIGMOD*, 2010.

[38] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH*, 2008.

[39] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *PVLDB*, 2005.

[40] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigumus, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, 2013.

[41] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS*, 2003.

[42] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, 2012.

APPENDIX

A. STITCHING COLUMNS C_K AND C_{K+1} WITH $W_K + W_{K+1} \leq B_K$ YIELDS BETTER PLAN

PROPERTY 1. *Stitching columns C_k and C_{k+1} with $w_k + w_{k+1} \leq b_k$ yields better plan.*

PROOF. Without loss of generality, assume the plan P before stitching has n rounds. Then if we are allowed to stitch C_k and C_{k+1} of P as stated to form a new plan P', the plans would look like:

Table 3: Major notations in this paper

Notation	Meaning
P	A code massage plan
P_0	The original column-at-a-time plan (no massaging)
$R_i : w/[b]$	The i-th round of a plan, containing w bits of data and using a b-bit bank SIMD-sort
C_i	The i-th column of a plan/tuple
w_i	The number of bits of the C_i
b_i	The bank size used in the i-th round of a plan
m	The number of input columns to be sorted
T_{mcs}	The total cost of multi-column sorting based on our cost model
T^i_{sort}	The sorting subcost of the i-th round
C	Calibrated constant used in cost model
M_{LLC}, M_{L2}	The capacity (in bytes) of the last level cache (LLC) / L2 cache
N	Input size (number of rows) to a multi-column sort program
$\mathcal{A}_i$	The perfect cost model based on exhaustively enumerating all feasible plans for query Q_i and measuring their actual running times
$\mathcal{Q}$	A query workload

$$
\begin{aligned}
P = \{\quad & R_1 : w_1/[b_1], \cdots \\
& \mathbf{R_k} : \mathbf{w_k}/[\mathbf{b_k}], \qquad \mathbf{R_{k+1}} : \mathbf{w_{k+1}}/[\mathbf{b_{k+1}}], \\
& \cdots, R_n : w_n/[b_n]\} \\
P' = \{\quad & R_1 : w_1/[b_1], \cdots \\
& \mathbf{R_k} : (\mathbf{w_k} + \mathbf{w_{k+1}})/[\mathbf{b_k}], \quad \mathbf{R_{k+1}} : \emptyset, \\
& \cdots, R_n : w_n/[b_n]\}
\end{aligned}
$$

The resulted plan P' should have $\mathbf{n-1}$ rounds. For the sake of proof, we insert a dummy "empty round" ($\emptyset$) at the $(\mathbf{k+1})$-th round above to make both P and P' have the form of $\mathbf{n}$ rounds. It can be seen that any R_i for $i < k$ and $i > k+1$ of both plans are identical.

Referring to Figure 2 and Section 4, three types of subcosts are incurred for *each* round: T_{lookup}, T^k_{sort} and T_{scan}, while $T_{massage}$ is incurred upfront. Let's define $cost(P.R_i)$ be the sum of T_{lookup}, T^i_{sort} and T_{scan} in the i-th round. We can therefore represent the total cost T_{mcs} of plan P and plan P' as:

$$T_{mcs}(P) = T_{massage}(P) + \sum_{i=1}^{n} cost(P.R_i) \tag{11}$$

$$T_{mcs}(P') = T_{massage}(P') + \sum_{i=1}^{n} cost(P'.R_i) \tag{12}$$

Now we are going to prove $T_{mcs}(P) - T_{mcs}(P') > 0$.

We first perform a **round-wise** comparison based on our cost model, to show that $\sum_{i=1}^{n} cost(P.R_i) - \sum_{i=1}^{n} cost(P'.R_i) = cost(P.R_{k+1})$:

1. When $i < k$, $cost(P.R_i) = cost(P'.R_i)$
   Intuitively, both plans behave identically from the first round until the k-th round. To quantify that, let us compare $P.R_i$ and $P'.R_i$. Note that the following of both are the same when $i < k$: (a) input size; (b) SIMD-sort bank size; (c) group information. Referring to our cost model in Section 4, these parameters suffice to determine T_{lookup}, T^k_{sort} and T_{scan}. As a result, all subcosts of the i-th round remain the same and thus $cost(P.R_i) = cost(P'.R_i)$.
2. $cost(P.R_k) = cost(P'.R_k)$
   Note the three factors (a), (b) and (c) above still remain the same for the k-th round. It is straightforward to see both (a) input size and (b) bank size remain the same. For (c), note the important fact that the group information fed to the i-th round is determined by **the number of bits prior to the i-th round**. Hence, though the bits allocated to $P.R_k$ and $P'.R_k$ are different, they both have $\sum_{j=1}^{k-1} w_j$ bits prior to them, so their costs are equal based on our cost model.
3. $cost(P.R_{k+1}) - cost(P'.R_{k+1}) = cost(P.R_{k+1})$
   This is obvious as $cost(P'.R_{k+1}) = cost(\emptyset) = 0$.
4. When $i > k+1$, $cost(P.R_i) = cost(P'.R_i)$.
   The reasoning is akin to (2). Although the bits allocated to R_k and R_{k+1} are different, the number of bits **prior to** R_i for $i > k+1$ are $\sum_{j=1}^{i-1} w_j$ in both plans. So all factor of (a), (b) and (c) are the same for both $P.R_i$ and $P'.R_i$.

Summarizing above, we have:

$$
\begin{aligned}
& \sum_{i=1}^{n} cost(P.R_i) - \sum_{i=1}^{n} cost(P'.R_i) \\
&= \sum_{i=1}^{n} \big(cost(P.R_i) - cost(P'.R_i)\big) \\
&= cost(P.R_{k+1})
\end{aligned} \tag{13}
$$

Now let's consider $T_{massage}$. It is actually possible for P' to incur a higher $T_{massage}$ by introducing more invocations of the four-instruction program ($\mathcal{I}_{FIP}$, cf. Section 4). However, the increase of $\mathcal{I}_{FIP}$ is bounded by 1 (i.e., $\mathcal{I}_{FIP}(P') - \mathcal{I}_{FIP}(P) \leq 1$), as we can always form columns of P' by carrying out one more FIP that stitches C_i and C_{i+1} of P. So,

$$
\begin{aligned}
& T_{massage}(P') - T_{massage}(P) \\
&= \mathcal{I}_{FIP}(P') \cdot \mathcal{C}_{massage} \cdot N - \mathcal{I}_{FIP}(P) \cdot \mathcal{C}_{massage} \cdot N \\
&= \big(\mathcal{I}_{FIP}(P') - \mathcal{I}_{FIP}(P)\big) \cdot \mathcal{C}_{massage} \cdot N \\
&\leq \mathcal{C}_{massage} \cdot N
\end{aligned} \tag{14}
$$

Putting Equations 11, 12, 13, 14 together, we have:

$$T_{mcs}(P) - T_{mcs}(P') \geq cost(P.R_{k+1}) - \mathcal{C}_{massage} \cdot N \tag{15}$$

In reality, $\mathcal{C}_{massage} \cdot N$ is always at least one order of magnitude smaller than $cost(P.R_{k+1})$, because the former only includes the cost of carrying out one FIP for N codes, while the latter includes the costs of lookup, sorting and scanning N codes. So $T_{mcs}(P) - T_{mcs}(P') > 0$.

In summary, we can conclude P' is a better plan than P by stitching adjacent columns as stated. □

B. MONETDB REFERENCE IMPLEMENTATION

Figure 11 shows a reference system architecture if implementing the techniques mentioned in this paper into MonetDB (modified/added components are shaded). Specifically:

- We shall add a *ByteSlice storage manager* that implements ByteSlice [14] column-store layout.
- We shall add four BAT[9] algebra operators to the MonetDB execution engine (aka. Gorblin Database Kernel [20]), including `ByteSlice-Scan` and `ByteSlice-Lookup` that execute fast scans and lookups on ByteSlice data as proposed in

[9]Binary Association Table, the primitive data structure used in MonetDB's execution engine.

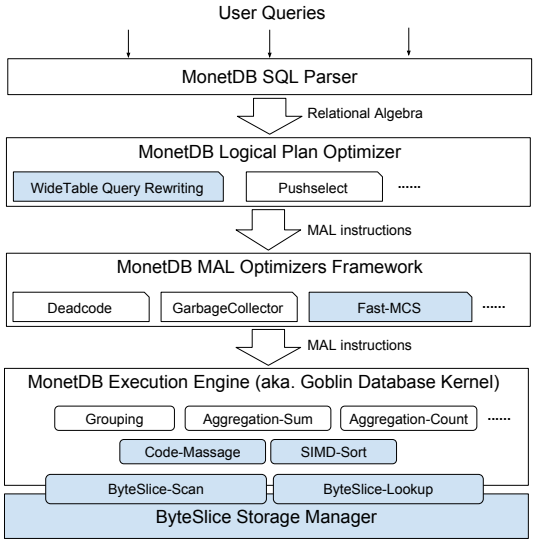


Figure 11: Implementing Code Massaging into MonetDB

[14], `SIMD-Sort` adapted from [4] that carries out (single-column) sorting with SIMD and multi-threading support, and `Code-Massage` that carries out code massaging process described in Figure 6. Other operators in MonetDB shall leave intact because operations other than scan and lookup manipulate intermediate data structures (BAT in this case) instead of base column storage [14].

- We shall add a `Fast-MCS` (fast multi-column sorting) module in the MonetDB MAL[10] optimizers framework. In MonetDB, an MAL plan is optimized by a series of modules in an *optimizer pipeline* [33]. Each module takes an MAL plan as input and transforms it into a more efficient one. For example, the `garbageCollector` module takes as input an MAL plan and injects calls to the garbage collector to free up space. Similarly, the `Fast-MCS` module shall (a) examine an input MAL plan and identifies the MAL instructions carrying out multi-column sorting, then (b) invoke the plan search algorithm (cf. Algorithm 1) to find an optimal massaging plan, finally (c) re-write the MAL instructions for multi-column sorting with code massaging. For example, a simplified set of MAL instructions to sort column `a` and column `b` with 16-bit bank would be:

```
(permuted_oid, group_info):=
          SIMD-Sort(a, 16, NULL)
permuted_b :=
          Lookup(b, permuted_oid)
(final_oid,final_group_info):=
          SIMD-Sort(permuted_b, 16, group_info)
```

Assume stitching these two columns as one and sorting it with 32-bit bank turns out to be the optimal plan in this case, the MAL plan re-written by `Fast-MCS` module would be:

```
super_column :=
          Code-Massage(a,b,'stitch')
(final_oid,final_group_info):=
          SIMD-Sort(super_column,32, NULL)
```

[10]MonetDB Assembly Language, the language to express a physical plan to be interpreted by the execution engine.

- We shall add a `WideTable Query Rewriting` module in the MonetDB logical plan optimizer. In MonetDB, the logical plan parsed from SQL is optimized using domain-specific rules. For example, `Pushselect` optimizes the logical plan by pushing down selections. So, the `WideTable Query Rewriting` module shall translate the logical plans into WideTable aware plans (i.e., to remove joins from the plans and use denormalized tables).

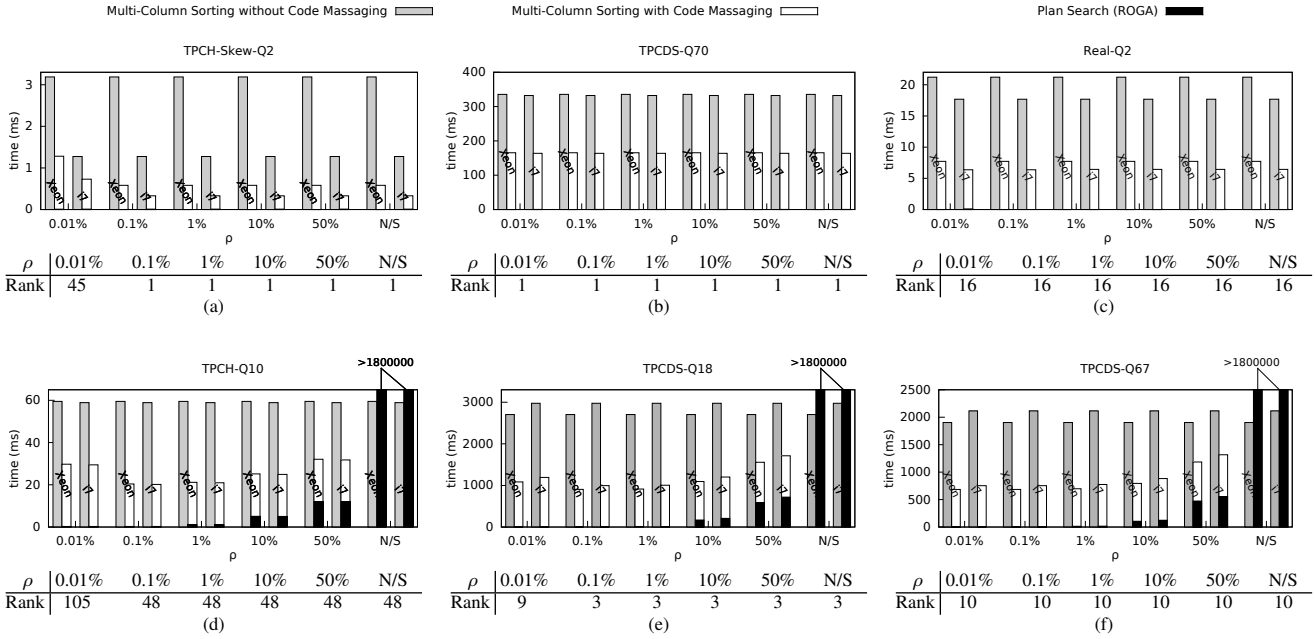
C. VALUE FOR TIME THRESHOLD ρ

Our experiments use $\rho = 0.1\%$ by default. Figures 12a, 12b and 12c show the time breakdown and the rank of resulted plan of one TPC-H, one TPC-DS, and one real query under various ρ on both Xeon and i7. We have the following findings. First, ROGA is so efficient that its running time is not observable in the figures. Second, we see that ROGA can complete even we do Not Set (N/S) any time threshold. Third, the effectiveness of ROGA is indeed not sensitive to ρ unless a really stringent value (e.g., 0.01%) is given. Indeed we found 22 such queries out of all 27 queries (9×2 TPC-H queries on uniform and skewed data + 4 TPC-DS queries + 5 real queries) that follow the findings above.

For the remaining 5 out of 27 queries, their total code width W is larger than 87, which may require up to 11 rounds of sorting (cf. Equation 10). Figures 12d, 12e, and 12f show the time breakdown of three of them. For those queries, we still see $\rho = 0.1\%$ is a good choice because that gives ROGA sufficient time to find a very competitive plan, without making itself as a bottleneck.

So, overall, we generally recommend $\rho = 0.1\%$. In general, we believe such an empirical approach is sufficient to determine the value of ρ. As a future work, we will study the following two approaches that can automate the process of finding ρ:

- **Offline calibration**: In this approach, the system uses a collection of sample queries and invokes the plan search algorithm (Algorithm 1) on each query with different ρ values. It uses a wide range of possible ρ values, from very stringent (e.g., 0.01%) to very loose (e.g.,10%, ρ higher than this is considered unacceptable anyway). For each query, we will record a list of estimated cost (T_{mcs}) with their corresponding ρ. We mark the "best" plan for each query as the one with the lowest estimated cost (T_{mcs}), which is usually obtained at the highest ρ. As observed in Figure 12, many queries find their best plans much earlier before using the largest ρ. So we find the smallest threshold ρ_0 that allows every query to reach its best plan. In other words, no query would obtain better plans with lower estimated cost if we increase ρ beyond ρ_0. Note that in this approach, the system only invokes the cost model to estimate the costs without actually executing the queries. So the process is fast and incurs very little overhead.
- **Online calibration**: In this approach, the optimizer assigns different ρ values to different queries online. Specifically, the plan search algorithm starts with setting $\rho = \rho_{low}$, where ρ_{low} is a low watermark. When time is up, we check whether the best plan found so far (P^*) has been updated. If yes, we anticipate further improvement is possible and extend the time limit by, say, $\rho \leftarrow \rho \times 2$. When the new time limit is reached, we check whether P^* has been updated again and increase ρ conditionally. To prevent ρ from growing infinitely, we cap it with a high watermark ρ_{high}. When ρ grows above ρ_{high}, we stop the plan search algorithm anyway. ρ_{low} and ρ_{high} can be set upfront as sufficiently small and large values respectively, for example, 0.01% and 10%.



ρ	0.01%	0.1%	1%	10%	50%	N/S
Rank	45	1	1	1	1	1

(a)

ρ	0.01%	0.1%	1%	10%	50%	N/S
Rank	1	1	1	1	1	1

(b)

ρ	0.01%	0.1%	1%	10%	50%	N/S
Rank	16	16	16	16	16	16

(c)

ρ	0.01%	0.1%	1%	10%	50%	N/S
Rank	105	48	48	48	48	48

(d)

ρ	0.01%	0.1%	1%	10%	50%	N/S
Rank	9	3	3	3	3	3

(e)

ρ	0.01%	0.1%	1%	10%	50%	N/S
Rank	10	10	10	10	10	10

(f)

Figure 12: Sorting time breakdown with various time threshold; N/S: do not set any time threshold

Table 4: Schema of the real dataset

Attribute name of relation `Ticket`	Description	Attribute name of relation `Market`	Description
ItinID	Itinerary ID	ItinID	Itinerary ID
Year	Year	MktID	Market ID
Quarter	Quarter (1-4)	Year	Year
OriginAirportID	Origin airport ID	Quarter	Quarter (1-4)
OriginCountry	Country of origin airport	OriginAirportID	Origin Airport ID
OriginStateName	State name of origin airport	DestAirportID	Destination Airport ID
RoundTrip	Round Trip Indicator (1=Yes)	OpCarrier	Operating carrier
DollarCred	Dollar Credibility Indicator	Passengers	Number of passengers
ParePerMile	Itinerary Fare Per Miles Flown in Dollars	MktFare	Market fare (yield per itinerary mile $\times$ miles flown)
RPCarrier	Reporting Carrier	MktDistance	Market distance (including ground transport)
Passengers	Number of passengers	MktDistanceGroup	Distance group, in 500 Mile intervals
Distance	Itinerary distance (including ground transport)	MktMilesFlown	Market Miles flown (Track Miles)
DistanceGroup	Distance group, in 500 Mile intervals	ItinGeoType	Itinerary geography type
ItinGeoType	Itinerary geography type (1=Non-contiguous Domestic)		

Table 5: Queries on the real dataset

Query ID	SQL	Semantic
Q1	`SELECT OriginAirport, DollarCred, FarePerMile FROM Ticket WHERE OriginStateName = 'Texas' ORDER BY DollarCred, FarePerMile`	Check the relationship between credibility and fare per mile in a given state.
Q2	`SELECT OriginAirportID, DistanceGroup, Passengers, RANK() OVER (PARTITION BY OriginAirportID, DistanceGroup ORDER BY Passengers) FROM Ticket WHERE ItinGeoType = 1`	For each origin airport and distance group, obtain number of passengers in order.
Q3	`SELECT RPCarrier, OriginState, RoundTrip, DistanceGroup, AVG(Passengers) FROM Ticket GROUP BY RPCarrier, OriginState ,RoundTrip, DistanceGroup`	For each carrier, calculate the average number of passengers per state, per trip type, and per distance group.
Q4	`SELECT OriginAirportID, DestAirportID, AVG(MktFare) FROM Market WHERE OpCarrier = 'B6' GROUP BY OriginAirportID, DestAirportID`	Retrieve the average fare between each pair of airport.
Q5	`SELECT OpCarrier, MktFare, RANK() OVER (PARTITION BY OpCarrier, ItinGeoType ORDER BY MktFare) FROM Market WHERE MktDistanceGroup = 1`	For each carrier and itinerary type, check its rank of market fare.