Modifying an existing sort order with offset-value codes

Goetz Graefe
goetzg@google.com
Google
Madison, Wisconsin, USA

Marius Kuhrt
kuhrt@mathematik.uni-marburg.de
University of Marburg
Marburg, Germany

Bernhard Seeger
seeger@mathematik.uni-marburg.de
University of Marburg
Marburg, Germany

ABSTRACT

Sorting in databases can exploit an existing sort order if it is related to the desired sort order. In some cases, an external merge sort can become multiple internal sorts; in other cases, sorting can become merging, i.e., merge sort without run generation, e.g., without an initial quicksort phase. If the input includes offset-value codes, they readily map to offset-value codes for the output, enabling efficient comparisons in merge steps and in subsequent query operations, e.g., a merge join.

There are many cases in which the existing sort order and existing offset-value codes can help creating a desired sort order. While today's database systems implement only the simplest cases, i.e., changing the sort order from A, B to A or from A to A, B, interesting orderings and offset-value codes permit substantial savings in more complex cases such as changing an existing sort order of A, B, C, D to A, C, B, D. Our research introduces the required techniques and measures achievable savings.

1 INTRODUCTION

Interesting orderings have been central in database query optimization for decades [26]. We believe, based on our product experience and experiments reported below, that the concept and industrial use of interesting orderings deserve more generality than is usually considered or implemented.

If sort order is important yet expensive to create, then caching and reusing prior sort effort is very worthwhile, e.g., effort spent on comparing strings. Offset-value coding [3] is a caching technique that is about as old as interesting orderings [26] but until recently has been used only for sorting and merging. In fact, offset-value coding is useful not only in external merge sort for unsorted inputs [16] but also in merge join and in all other sort-based query execution algorithms [13]; it now turns out that it is useful even for modifying an existing sort order. Table 1 lists cases of going from one sort order to another; the discussion below considers the required data movement (e.g., merging), the required data comparisons (of rows and of columns), and the savings enabled by novel use of offset-value codes present in the input. The proposed techniques eliminate about half of the row comparisons and often all column value comparisons.

The ability to modify a sort order very efficiently has substantial practical implications, even beyond query plan execution and query optimization. Take, for example, any many-to-many relationship such as enrollments of students in courses. In order to enable efficient access to student transcripts, i.e., a merge join of students and enrollments, one copy of the enrollment table must be sorted and indexed on student identifier. In order to enable efficient access to class rosters, i.e., a merge join of courses and enrollments, another copy of the enrollment table must be sorted and indexed on course number. Thus, two copies of the enrollment table must be created and maintained in the database. If, however, an index ordered on (course number, student identifier) can be scanned efficiently ordered on (student identifier, course number), then a single copy of the enrollment table suffices and can serve both transcripts and rosters, i.e., merge joins with either students or courses.

Table 1: Sort keys for which an existing sort order and existing offset-value codes can help creating a desired sort order and new offset-value codes. Each letter can be a column, a list of columns, i.e., a database row, a list of characters, i.e., a text string, or a list of bytes, e.g., a normalized key.

Case	Existing	Desired
0	A, B	A
1	A	A, B
2	A, B	B
3	A, B	B, A
4	A, B, C	A, C
5	A, B, C	A, C, B
6	A, B, C	B, A, C
7	A, B, C, D	A, C, B, D

Of course, this alone does not solve the problem of joining all three tables, but the techniques below also enable efficiently sorting an initial join result sorted on (course number, student identifier) into an input for the second join, i.e., sorted on (student identifier, course number). In terms of Table 1, both the scan for a two-table join and the intermediate sort for the three-table join are examples of case 3. In fact, any many-to-many relationship induces an example of cases 3 or 6. The reader may apply any discussion of case 3 to enrollments of students in courses. [1]

The following sections review related prior work including required background information, then introduce techniques for modifying a pre-existing sort order in ways that exploit old offset-value codes and create new ones, list hypotheses or claims about performance and scalability, report on experiments testing these hypotheses, and finally sum up and conclude.

2 RELATED PRIOR WORK

Any work on advanced sorting techniques owes much to pioneering work on internal and external merge sort, quicksort, and priority queues [5, 6, 14, 19]. Segmented sorting, merging pre-existing runs, and their combination are well known [10]. Friend credits that the "von Neumann approach (named after its originator, John von Neumann) takes advantage of existing sequences in the data" [5, 7, 18].

[1]For small joins, i.e., a lookup join from a student to her enrollment or a lookup join from a course to its enrollment, a traditional design requires the same two indexes on the enrollment table. In contrast, the MDAM design [20] can search fairly efficiently even with only a single index. While the MDAM design focuses on search techniques, our new techniques focus on merge techniques.

	Column index			
	0	1	2	3
Rows and their column values	5	4	7	1
	5	4	7	2
	5	6	2	6
	5	6	2	6
	5	6	3	4
	5	8	2	3
	5	8	4	7

Columns w/ run-length encoding			
0	1	2	3
5	4	7	1
			2
	6	2	6
		3	4
	8	2	3
		4	7

Rows with prefix truncation				
prefix size	0	1	2	3
0	5	4	7	1
3				2
1		6	2	6
4				
2			3	4
1		8	2	3
2			4	7

Descending offset-value codes			Ascending offset-value codes		
offset	domain – value	OVC	arity – offset	value	OVC
0	95	95	4	5	405
3	98	398	1	2	102
1	94	194	3	6	306
4	-	500	0	-	0
2	97	297	2	3	203
1	92	192	3	8	308
2	96	296	2	4	204

Figure 1: Derivation of prefix truncation and descending/ascending offset-value codes for a table sorted on 4 keys.

2.1 Offset-value codes

Within a sorted table in columnar format, run-length encoding suppresses column values that equal the same column in the preceding row. In row format, prefix truncation suppresses leading sort columns that equal the same column in the preceding row. Transposing between column format with run-length encoding and row format with prefix truncation is fast as it requires no column value comparisons.

For text strings and string sorting, prefix truncation is discussed as longest common prefix (LCP) [21] and works well with tournament trees [1], discussed in Section 2.2. Offset-value coding is a refinement of prefix truncation: it combines the prefix size and the following column value into a single fixed-size integer and surrogate key. Prefix truncation and offset-value coding work with lists of column values, i.e., database rows, lists of characters, i.e., text strings, and lists of bytes, e.g., normalized keys.

Importantly, offset-value codes are order-preserving. If two rows are encoded relative to the same base row, comparing the two rows' offset-value codes can quickly decide a comparison. If two rows' offset-value codes are equal, column-by-column comparisons can resume after the shared prefix. The new comparison effort is cached in a new offset-value code in the loser. Within text strings, the logic is like *strcmp*() with starting and ending offsets; within binary strings and normalized keys, it is like *memcmp*() with starting and ending offsets. In run generation and merging using tournament trees, offset-value codes decide most row comparisons.

Figure 1 shows, on the left, rows in ascending order on all four columns. The second block shows the same table in columnar format compressed with run-length encoding of leading sort columns. The third block shows rows compressed with prefix truncation – note that precisely the same values are suppressed. The remainder of Figure 1 illustrates the derivation of both descending and ascending offset-value codes. In the calculation of offset-value codes, the arity of the sort key is 4 due to four sort columns; and the example assumes that the domain size of each column is 100. Descending offset-value codes take the offset and the negative of the column value. In a comparison of two rows with offset-value codes relative to the same base key, the higher offset-value code is the winner, e.g., a duplicate of the base key with code value 500. Ascending offset-value codes take the negative offset but the actual column value. In a comparison, the lower offset-value code is the winner, e.g., a duplicate row with code value 0.

Recent research [13] has extended offset-value coding from merge sort to merge join, duplicate removal, and in fact most sort-based query execution. Offset-value codes can speed up the in-sort logic for "distinct", "group by", "pivot", "limit", and "top" queries, the in-stream logic for the same operations over sorted data, as well as merge join and other binary operations. For operators in pipelines with interesting orderings, simple yet fast computations map offset-value codes for a sorted input to offset-value codes for a sorted output.

In many ways, offset-value codes serve in sorting and in sort-based query execution the role of hash values in hash-based query execution [11]. Both offset-value codes and hash values are fixed-size fixed-type surrogate keys, e.g., 8-byte unsigned integers. Their comparisons are compiled into query execution algorithms and thus very fast. However, whereas hash values can assert only that two rows (or their keys) are different, offset-value codes can also assert their equality or their sort order.

2.2 Tournament trees

Tournament trees, also known as tree-of-losers priority queues and related to elimination rounds in sports competitions, have been used for decades for internal sorting, for run generation by replacement selection, and for merging sorted runs [6]. They reduce the count of row comparisons to nearly the provable lower bound, i.e., $\log_2(N!) \approx N \times \log_2(N/e)$ for N rows and $e = 2.718... \approx 19/7$. Among all algorithms and data structures for sorting and priority queues, tournament trees seem the best match for offset-value codes.

Figure 2 shows a tournament tree immediately after initialization with the lowest key value in each of 12 merge inputs. Dashed boxes along the bottom represent the current keys from the runs to be merged; solid boxes are nodes in the tournament tree. The example node in the top-right corner explains what all the numbers mean. An index is here the run identifier within the merge logic, values 0-11. The root of this tournament tree is in array slot 0. It holds the overall smallest key value, 61, which came from merge input 9. The next key value in merge input 9 is also shown; this key value will start the next leaf-to-root pass. Key value 157 is above key value 87 because 157 emerged as the winner from the left subtree and 87 is only the runner-up in the right subtree. The overall runner-up key always is somewhere along the overall winner's leaf-to-root path, because it would have "won" all comparisons and reached the root node if it had not "met" the overall winner, necessarily along the overall winner's leaf-to-root path. The runner-up can be anywhere along the winner's leaf-to-root path, even in a leaf node if that is where the runner-up met the overall winner.

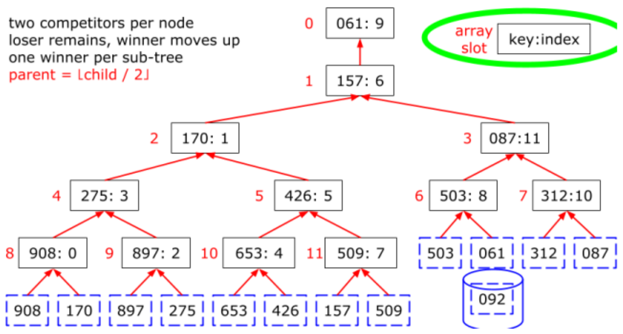

Figure 2: A tournament tree with 12 merge inputs.

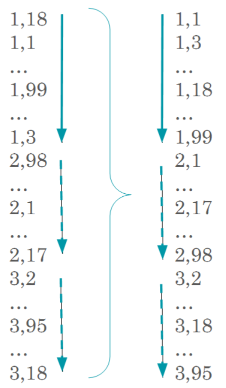

Figure 3: Segmented sorting from key A to key A, B.

Together, tournament trees and offset-value codes reduce column value comparisons (or character comparisons when sorting strings) to a linear count, i.e., $\leq N \times K$ for N rows with K key columns. The count of required column value comparisons equals the sum $x + y$ where x is the count of "=" comparisons in verifying a claimed (and correct) sort order and y is the count of rows minus one, which is also the count of "<" comparisons in verifying a sort order. In a sorted table, x also equals the compression opportunity by prefix truncation (within rows), by run-length encoding (within leading sort columns), and by tries [2, 25]. The core algorithms of tournament trees and offset-value coding are so simple and concise that they are captured in the mainframe instructions UPT "update tree" and CFC "compare and form codeword" [15, 16].

2.3 Other related prior work

Less directly connected to the new work, even if they have inspired some of our thinking, are MDAM [20], which interprets compound (multi-column) b-trees as multi-dimensional indexes and supports a large variety of query boxes, sorting from UB-trees [29], which produces output sorted in one dimension from a multi-dimensional index based on a b-tree and a space-filling curve, and further related work such as [23, 24, 28]. An earlier survey [8] ignores the problems and opportunities in modifying an existing sort order. [2]

3 MODIFYING A SORT ORDER

Table 1 lists prototypical cases. Among those, case 0 is trivial and often the only case implemented in database query optimization and query execution. What appears as column names in the table could be a list of columns or expressions, presumably with such lists reduced based on functional dependencies [27]. Those may originate from database integrity constraints or from prior operations within a query, e.g., a "group by" creating an intermediate result with a new primary key.

The following techniques – segmented execution, merging pre-existing runs, and their combination – have been known in the past. The new contributions here are integration of offset-value coding, modification of pre-existing offset-value codes, and conclusions about new or expanded techniques in query execution, in query optimization, and in physical database design. Importantly, modifying a pre-existing offset-value code means reusing previously cached comparison effort in order to avoid repeating this effort.

[2]There might be further related prior work on changing the representation of sparse multi-dimensional matrices, specifically modifying the order of dimensions.

3.1 Segmented sorting

Case 1 in Table 1 turns the sort key in the input into a prefix of the sort key in the output. In terms of sorting, this prefix permits segmenting the input by distinct values of this prefix and then sorting each segment on the remaining desired sort keys, treating the rows within each input segment as unsorted. In other words, for each distinct value of A, sort on B. Based on the terms major sort key and minor sort key, this is called a minor sort. The minor sort is invoked as many times as there are distinct value of A; the cost of each minor sort depends on the count of B values.

Figure 3 illustrates the technique. On the left, pairs of numbers are sorted only on the first column. A scan and sort per segment produces a final output sorted on both columns, shown on the right. Using solid arrows for completed work and dashed arrows for pending work, the diagram attempts to capture the state after the first segment has been read, sorted, and written; the second segment has been read but not yet written; and the third segment has yet to be read.

In terms of efficiency, the count of row comparisons that can be saved depends on the count and size distribution of segments. It is possible to save half of the comparisons, e.g., if the count of distinct values in A is a square root of the count of rows. Detection of segment boundaries can use extrapolation search rather than comparing each row with its predecessor. More importantly, segmented sorting can save merge levels. It achieves the greatest advantage when each segment requires only an internal sort whereas sorting the entire input requires an external sort.

Offset-value coding can help in two ways in case 1 of Table 1. First, it can detect segment boundaries by offsets smaller than the size of A (i.e., 1 if A is a column, otherwise the list size). Second, when sorting on B, offset-value codes can bound the count of column value comparisons to the size of B (instead of the sum of sizes of A and B) times the row count (minus the segment count).

3.2 Merging pre-existing runs

Case 2 in Table 1 forms the sort key in the output from a suffix of the sort key in the input. In terms of sorting, the input consists of runs sorted on the desired key, one for each distinct value of the prefix of the input sort key. Thus, an external merge sort can skip run generation. Distinct values of the prefix define runs. If the input has too many distinct values in its sort prefix, graceful degradation from a single merge step to multiple merge steps may be required.

Figure 4 shows a b-tree (represented by a triangle with the root node on the left) with pairs of numbers as its key values.

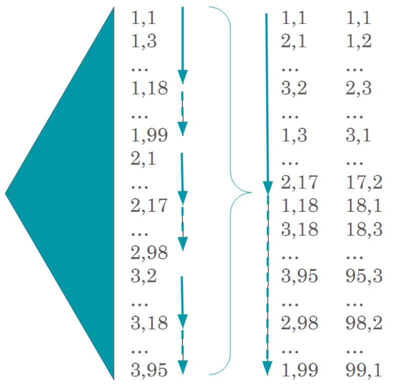

Figure 4: Merging pre-existing runs from key A, B to key B, A.

Three scans have reached value 17 in the second column and will eventually proceed to 99. The output of the three scans is merged and written sorted on the second column; the right-most set of pairs merely inverts the column order to make the sort order more visible. Note that the two instances of value 18 in the second column are sorted on the first column, i.e., the merge logic implements a stable sort.

In terms of efficiency, the savings from skipping run generation can be substantial, since a traditional external merge sort invokes most of its row comparisons during run generation (assuming more rows per run than runs).

Offset-value coding can help in two ways. First, it can detect run boundaries. Second, the offset-value codes within the pre-existing runs can be reused after the offset component has been decremented by the size of the prefix (i.e., 1 if A is a column, otherwise the list size). This manipulation of offset-value codes is novel and unique to changing the position of a column within a sort key, e.g., making the suffix of the input sort key into the prefix of the output sort key. After offset-value codes have been adapted in this way, the merge logic can use them as in a traditional merge sort. Merging may compare additional column values and set new offset-value codes, e.g., when finding duplicate values in separate runs.

Case 3 in Table 1 is similar except for stable sorting (merging). For two equal values of B, the row earlier in the input is also earlier in the output. This is easily added to the merge logic.

3.3 Combined cases

Case 4 in Table 1 combines the conditions of cases 1 and 2; case 5 similarly combines cases 1 and 3. Case 6 extends case 4 with an additional column that makes stable sorting more explicit; case 7 similarly extends case 5. As the techniques required for case 4 readily extend to cases 5-7, case 4 is the focus here.

In terms of sorting, the sort keys of input and output share a prefix, which suggests segmented sorting. Within each segment, the input consists of runs pre-sorted on the desired sort suffix, one for each distinct value of the infix of the input sort key.

In terms of efficiency, segmented sorting may save merge levels (as in case 1) and reusing pre-existing runs saves run generation (as in case 2) – thus, the efficiency gains of cases 1 and 2 combine.

Offset-value coding can help in multiple ways: detecting segment boundaries, detecting run boundaries within segments, and providing offset-value codes for the merge phase. Offset-value codes for the output derive from offset-value codes in the input by decrementing offsets by the size of the infix. In other words, much of the effort for column value comparisons cached in offset-value codes of the input is reused and saves column value comparisons when merging and producing rows in their new sort order.

row#	A	B	C	offset	value
1	1	1	1	0	1
2	2	1	1	0	2
3	2	1	3	2	3
4	2	2	1	1	2
5	2	2	2	2	2
6	2	3	4	1	3
7	2	3	4	3	–
8	2	3	5	2	5
9	3	1	1	0	3
…					

Figure 5: An input table sorted on columns A, B, C, shown with row#s, offsets and values.

Returning to the example of the introduction, i.e., enrollments of students in courses, consider a university with multiple campuses such that both student identifiers and course numbers are meaningful only within the context of a specific campus. In that case, all indexes, join predicates, and required sort orders, e.g., for merge joins, have "campus code" as a new prefix. Instead of modifying a sort order from (course number, student identifier) to (student identifier, course number), which the introduction used as an example of case 3 within Table 1, the new problem is to modify a sort order from (campus code, course number, student identifier) to (campus code, student identifier, course number), which is an example of case 5. Incidentally, if students can repeat a course, e.g., after a first attempt with a poor grade, the keys of input and output need a suffix "semester", which creates an example of case 7.

While the comparison effort for integer columns as in this example is moderate and perhaps not crucial to avoid, the reader is encouraged to imagine lists of columns, of bytes, or of symbols (strings). In fact, even student identifiers and course numbers are often strings rather than integers.

3.4 A concrete example

Figures 5-9 illustrate modifying a sort order from A, B, C to A, C, B using a combination of segmented sorting, merging pre-existing runs, and offset-value codes. The reader is encouraged to think of A, B, and C as integers, as lists of columns, or as strings (lists of symbols or bytes, even variable-sized). This example shows how innovative use of old offset-value codes (for the pre-existing sort order) permits creating a new sort order and setting new offset-value codes, all without any column value comparisons.

Figure 5 shows a table sorted on columns A, B, C plus row numbers as well as offsets and values that should be combined into offset-value codes. Of course, only columns A, B, and C need to be stored. In fact, compression can truncate in each row a prefix with a size equal to the offset.

While modifying the sort key from A, B, C to A, C, B, the input rows with $A = 2$ form a segment. The first row in each segment (and no other row) has $offset = 0$ or, if A is a list, $offset < |A|$, i.e., the size of list A.

Figure 6 shows pre-existing runs identified by distinct values of B within the segment with $A = 2$. The first row in each run

row#	A	B	C	offset	value	classification of rows
2	2	1	1	0	2	first row in segment
3	2	1	3	2	3	other row
4	2	2	1	1	2	first row in run
5	2	2	2	2	2	other row
6	2	3	4	1	3	first row in run
7	2	3	4	3	–	duplicate row
8	2	3	5	2	5	other row

Figure 6: Pre-existing runs in the segment defined by $A = 2$ **with rows classified based on the old offsets.**

row#	A	C	B	offset	value	classification of rows
2	2	1	1	**1**	**1**	first row in segment
3	2	3	1	1	3	other row
4	2	1	2	**1**	**1**	first row in run
5	2	2	2	1	2	other row
6	2	4	3	**1**	**4**	first row in run
7	2	4	3	*3*	–	duplicate row
8	2	5	3	1	5	other row

Figure 7: Pre-existing runs prepared for merging on C, B **within the segment defined by** $A = 2$**.**

(after a segment's first run) has $offset = 1$ or $|A| \leq offset < |AB| = |A| + |B|$. These runs are pre-sorted on C and can be merged in order to sort the entire segment on C, B. The last column of Figure 6 contains the classification derived from those offset-value codes. Importantly, classifying rows relies entirely on offset-value codes; there is no need to inspect or compare column values.

Figure 7 shows the pre-existing runs prepared for merging on C, B. This preparation is applied while scanning these rows; there is no need to store the prepared rows, not even temporarily. Compared to the prior figures, the column sequence is adjusted for the new sort order. This is done for illustration purposes only; it is not required in memory or on storage.

Duplicate rows, identified by offsets equal to the key size or $offset = |ABC|$, retain their offset-value codes (shown in italic font in Figure 7 and indicated as classification). The first row in each run has its offset-value code saved and overwritten (shown in bold font in the figure). The old offset-value code reflects the difference to the values of A or of B in the row preceding the run. The new offset is 1 or $|A|$; the new value must be extracted from key column C. Other rows within each run, i.e., old $offset = 2$ or $|AB| \leq offset < |ABC|$, retain the value part of their offset-value code but the offset part is decremented by 1 or by $|B|$ (shown underlined in Figure 7).

While the small example in Figure 7 shows only three "other" rows (one in each run), such rows would be the bulk of the merge input in more realistic examples, e.g., tens or even thousands of rows in each run. After their simple and efficient adjustment for the new sort order, their new offset-value codes ensure practically effortless comparisons in the merge logic. This is crucial for efficiently modifying a sort order.

old row#	A	C	B	offset	value
2	2	1	1	1	1
4	2	1	2	2	2
5	2	2	2	1	2
3	2	3	1	1	3
6	2	4	3	1	4
7	2	4	3	3	–
8	2	5	3	1	5

Figure 8: Output of merging on C, B **within segment** $A = 2$**.**

Figure 8 shows the output of merging the runs in Figure 7. The merge logic discovers that $C = 1$ occurs in two of the merge inputs and adjusts offsets and values accordingly. Even in this case, there is no need to compare values of B because values of B are necessarily constant within each merge input run, different between runs, and sorted from run to run. If B is a list or $|B| > 1$, the required new offset-value code for this difference in B can be derived efficiently, using a proven theorem and its corollary [3, 13], from the old offset-value codes saved from the first row of each run. The offsets within these offset-value codes must be incremented by 1 or by $|C|$ to reflect the position of column B within the new sort key. In contrast to column B, values of C may need comparisons if an offset-value code cannot capture an entire value, e.g., if C is a list or $|C| > 1$.

Duplicate keys bypass the merge logic and immediately follow their preceding key from the merge input to the merge output. The merge logic cannot discover new duplicate keys because runs differ in their value of B. However, when modifying a sort order from A, B, C to A, C, the merge logic may discover new duplicate keys. As part of discovering new duplicate keys, the merge logic using a tree-of-losers priority queue and the comparison logic using offset-value coding assign offsets equal to the key size.

Figure 9, the final figure in this sequence, shows the merge output for the segment with $A = 2$ between an earlier segment and a later one. The only difference to Figure 8 is the adjusted offset and value in the first row of each segment (shown in bold font). The new offset-value code for a segment's first output row is the value saved from the segment's first input row, i.e., from the first row of the first run within the segment. In the example, it is the offset-value code that indicates $A = 2$ following an earlier value in column A.

This example does not require any column value comparisons for A or B, whether those are columns or lists of columns, characters, or bytes. The example requires column value comparisons for C only if C is a list, i.e., $|C| > 1$, and if the merge logic finds values of C in different runs with equal offset-value codes. If such a column value comparison is required, it takes advantage of the given offset, assigns a new offset-value code to the loser, and indicates a new opportunity for compression by prefix truncation in the output.

This example of case 5 in Table 1 can easily be modified into one for case 3 – it merely requires assuming a constant (single) value for column A and thus a single segment. It can also be modified into an example for case 7 – multiple values with distinct values in column D would be treated like duplicates in the figures above, i.e., the condition would change from $offset = |ABC|$ to $offset \geq |ABC|$. Thus, Figures 5-9 also illustrate, in a compact way, all the examples of enrollments of students in courses, with and without the prefix "campus code" or the suffix "semester".

new row#	A	B	C	offset	value
...	1	...	...	...	...
2	2	1	1	**0**	**2**
3	2	1	2	2	2
4	2	2	2	1	2
5	2	3	1	1	3
6	2	4	3	1	4
7	2	4	3	3	–
8	2	5	3	1	5
...	3	...	...	**0**	**3**

Figure 9: Final merge output of the segment with $A = 2$ with adjusted offsets and values.

3.5 Algorithm

Even if the example answers most questions, a more algorithmic description of the technique may further clarify.

The first step (during compile-time) compares the existing sort order and the desired sort order. If there is a shared prefix, including ascending vs descending sort as well as ordering for international strings, then this prefix defines segments for segmented sorting. If, in the desired sort order, the next sort key(s) appear in a later position within the existing sort order, this key (these keys) define the merge order. The intervening sort key(s) in the existing sort order define run identifiers. A cost-based decision based on run sizes and run counts (i.e., counts of distinct values) determines whether to exploit the pre-existing sort order in the way described above. A generalization of this analysis could consider opportunities using backward scans of sorted runs.

The next step (during run-time) ensures that the pre-sorted input is available for scanning and merging. This may materialize the input in memory or on storage, either entirely or one segment at a time. The steps so far are not new or unique to the present work.

The third step identifies and merges runs as shown in Section 3.4. In a scan of the pre-sorted input, end-of-input or a segment boundary (identified by the row's offset-value code for the old sort order) flushes the prior segment. Each run boundary (also identified by old offset-value codes) saves the old offset-value code and then adds a run to the merge logic, i.e., one entry into the priority queue. Flushing a prior segment means executing the merge step. The required merge logic is well-known from external merge sort, including offset-value codes deciding many or most row comparisons.

There are, however, a few differences due to offset-value codes. The first difference is that duplicate key values are identified by old (pre-existing) offset-value codes to let these keys bypass the tournament tree merging in the new sort order. An important difference is that the scans feeding the merge logic adjust offset-value codes, e.g., decrementing offsets for key values going into the merge logic (called "other" key values in Section 3.4). A further difference is the calculation of new offset-value codes for newly detected duplicates in the merging keys: the offset-value codes saved from the pre-existing sort order determine new offset-value codes without comparison of column values that define runs within the input.

3.6 Summary of techniques

In summary, a pair of simple techniques – segmented sorting and merging pre-existing runs – can speed up sorting in database query processing and probably in many other environments and contexts. Their complexity is very limited yet their benefits can be substantial. If comparison effort is cached in offset-value codes and if offset-value codes are reused with adjusted offsets as introduced above, this benefit is greatly amplified, as the following hypotheses claim and the subsequent measurements show.

4 CLAIMS AND MEASUREMENTS

Our experiments aim to refute or support specific hypotheses.

4.1 Hypotheses and claims

The following claims and hypotheses delineate the value of advanced sorting, in particular segmented sorting and merging pre-existing runs, and of offset-value codes in those contexts.

(1) Segmented sorting can save a merge level, even turning external merge sort into internal sorting.
(2) Segmented sorting benefits from using offset-value codes in the input for detecting segment boundaries and from extending those offset-value codes while sorting a segment.
(3) External merge sort invokes most of its row and column comparisons during run generation, i.e., the input phase.
(4) Merging runs pre-existing in a sorted input saves many or most comparisons.
(5) Reusing and adapting offset-value codes from the input speeds up merging pre-existing runs.
(6) Run-length encoding in sorted column stores enables efficient detection of segment boundaries, efficient transposition to row-by-row prefix truncation and offset-value coding, and efficient merging of pre-existing runs.
(7) Merging runs pre-existing in a storage structure (such as a b-tree or a column store) also saves I/O (compared to an external merge sort).
(8) Merging pre-existing runs also applies to other storage structures such as log-structured merge-forests, stepped merge-forests, and partitioned b-trees [9, 12, 17, 22].
(9) The benefits of segmented sorting and of merging pre-existing runs are cumulative.
(10) Interesting orderings in database query optimization should be expanded beyond using an existing sort order – they should also exploit techniques for modifying an existing sort order, and they should do so in traditional row stores and their indexes as well as in column stores and their query execution strategies.

4.2 Thought experiments

Most of the hypotheses above do not require detailed experimental support; therefore, we support them here with arguments rather than measurements.

For Hypothesis 1, consider a large input that requires external merge sort, possibly even multiple merge steps and levels. If segments are smaller than the available memory, internal sorting one segment at a time suffices. This is an example of saving a merge level, possibly even multiple merge levels.

For Hypothesis 2, consider an input with segments defined by k leading sort columns. Row-by-row comparisons are faster if they can rely on offset-value codes rather than comparing k column values. The second half of the hypothesis depends on how comparisons within a segment are implemented: if they skip over the k columns defining segments, and if a single column defines the sort order within a segment, then extending pre-existing offset-value codes has limited value; if they always compare these k columns first, offset-value codes save sorting effort.

The validity of Hypothesis 3 depends on the number M of rows per initial external run and the number W of initial external runs. In most practical cases, the former far exceeds the latter ($M \gg W$) with more comparisons per row during run generation than across all merge steps ($\log_2(M/e) > \log_2(W)$).

Hypothesis 4 follows directly from Hypothesis 3 as merging pre-existing runs skips run generation.

Hypothesis 5 deserves experimental validation.

Detailed discussions in earlier work [4] support Hypothesis 6.

Hypothesis 7 complements Hypothesis 4. An input as assumed here saves run generation in its entirety, not only the effort for key value comparisons but also the effort for saving initial external runs. If the sort input comes from a data stream, e.g., as an output of earlier query execution operations, then the input needs to be written with no I/O savings compared to run generation.

In Hypothesis 8, if the input is partitioned on the sort key, Hypotheses 4 and 7 apply. Otherwise, each partition may hold key values from the entire key domain. This is the common situation in log-structured merge-forests and similar storage structures. Suppose segments within partitions are aligned (same segment boundaries among all partitions, e.g., distinct key values in leading key columns). In that case, segments (and their key ranges) can be sorted one at a time, which requires sorting within segments and merging across partitions. Taking advantage of unaligned segment boundaries requires complex bookkeeping with rare applications and overall uncertain benefits.

Hypothesis 9 deserves experimental validation.

Hypothesis 10 is really a conclusion from all prior hypotheses: these run-time benefits are available to database users only if compile-time query optimization recognizes opportunities to exploit them and assembles query execution plans accordingly.

4.3 Implementation and system context

The prototype query execution engine [3] used in the following experiments is implemented in C++17. All operators in this engine are pull-based, resulting in simple and clean interfaces. Each row consists of its column values and a special (non-columnar) field holding the offset-value code. To accommodate all experiments, the number of columns is set to 32, where each column is an 8-byte integer. All sort-based operators use a tree-of-losers priority queue for run generation and for merging. Each tree entry within this queue holds an integral 8-byte key and a reference to the corresponding row or run. This key encodes the offset-value code of the row along with additional information. For example, high fence keys are introduced to indicate that a run is exhausted. As these fences are higher than any valid key, they reach the head of the queue only when the merge process is complete.

For the following experiments all sort operators in this engine have been adjusted to eliminate the need to buffer rows from the input. Instead, pointers to input rows are assumed to stay valid and only these pointers are sorted and passed to the consumer. In particular, the entire table is kept in memory and no rows are spilled to disk in any operation. Each bar displays the average run time of 10 executions on a workstation (AMD Ryzen 7 2700X, 2x16GB DDR-4 3200, Linux 6.7.8, GCC 13.2.1) and a (very small) bar for the 95% confidence interval.

[3]Code available at https://github.com/kmarius/offset-value-coding/tree/edbt

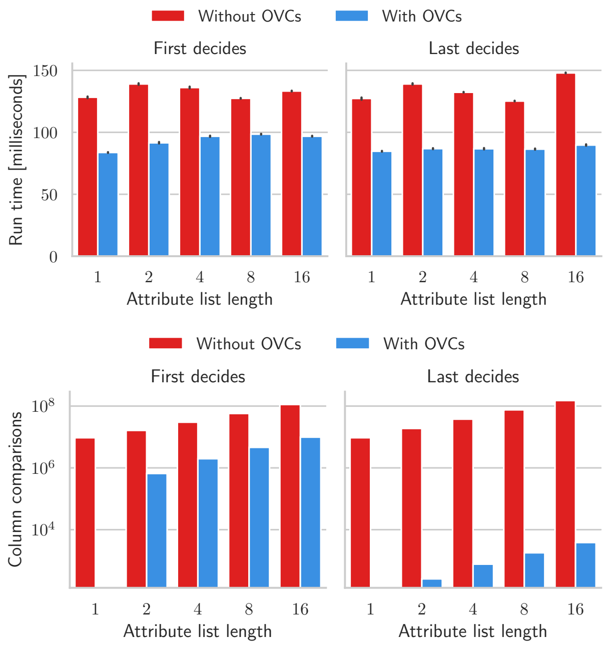

Figure 10: Run time and column comparisons of changing an existing sort order from A, B to B, A with column lists of varying lengths and 2^{20} rows.

4.4 Experiments and measurements

We tested Hypothesis 5 by sorting $2^{20} \approx 10^6$ rows initially sorted on A, B but desired sorted on B, A. This experiment mirrors the example with course numbers and student identifiers. For the example of Section 3.4, the experiment represents the work within one segment.

In this experiment, A and B are lists of columns. Their sizes vary by powers of 2 from 1 (a single column) to 16 (a large list uncommon in “group by” and “join” operations but common in “distinct” and “intersect” operations). Two versions of this experiment use key values such that row comparisons are decided by the first or the last column, respectively. To that end, most columns of each row are filled with zeroes and only the first (last) column in each list contains a different value (not necessarily unique). Experiments with medium-size lists, e.g., 4 columns, are representative also for longer lists, e.g., 8 columns, with comparisons decided somewhere between the first and last columns. A sort operation without offset-value codes must compare the columns in list A of each row with its preceding row to detect pre-existing runs. The sort logic then uses the columns in list B to sort the data. No additional comparisons of A are needed: in case of duplicate values of B, the row from the run with the lower index sorts lower. Hypothesis 5 is supported if offset-value coding reduces comparison efforts and run-times.

The top diagram in Figure 10 shows the run times. In general, longer column lists result in more expensive row comparisons and increased run times. The fact that the operation without offset-value codes shows only a vague upward trend can only be attributed to cache effects. The benefits of utilizing offset-value codes range from 20% to 35%, where the difference is smaller when the first column of each list decides the comparisons. In this case, the traditional sort method detects differences in rows quickly, but duplicates require comparing the complete list. If the last column decides a comparison, the full list has to be compared in all cases. Conversely, when using offset-value codes, the sort operation on the data where the last column decides the comparison is slightly faster than on the data where the first column decides. Examining counts of column comparisons provides more insights into these results.

The bottom diagram in Figure 10 shows counts of column comparisons (i.e., the actual comparisons of column values, not counting comparisons of offset-value codes). Note the logarithmic scale. With an column list length of 1, the first column is also the last column of each list and the two versions of the experiment coincide. In this case, existing offset-value codes in the input have already captured all information so that no additional comparisons are needed during merging the existing runs.

In the bottom left diagram, the first column decides all comparisons. The sort operation without offset-value codes performs $9.5 \cdot 10^6$ comparisons when each list consists of only one column. This count grows to $112 \cdot 10^6$ for an column list of length 16. In contrast, the operation with offset-value codes performs approximately $0.66 \cdot 10^6$ to $9.9 \cdot 10^6$ comparisons for column lists of sizes 2 to 16, respectively. Column comparisons are needed when the merge logic finds two rows from different runs with offset $|A|$ and the same first column in list B. In this case, the remaining columns in B have to be compared.

In the bottom right diagram, the last column in each list decides comparisons. Sorting without offset-value codes performs from $9.5 \cdot 10^6$ to $151 \cdot 10^6$ comparisons. With the benefit of offset-value codes, this count varies from 0 when each list is a single column to just under $4,000$ when each list holds 16 columns. This might seem surprising but the explanation is actually simple. All comparisons performed during a merge with offset-value codes result from comparisons within the column list B. Comparisons of A are completely avoided by comparing run numbers. Most rows (other than the first in each run) have an offset of $|AB| - 1$ or $|AB|$ if they are different from their preceding row or a duplicate, respectively. In both cases, offset-value codes contain the full information and the merge logic invokes no further comparisons. Only the first row in each run has a smaller offset and once it participates in a comparison that is not decided by offset-value codes, its offset-value code is adjusted (with offset $|AB| - 1$ or $|B|$) and no further column comparisons are needed to distinguish it from other rows.

Taken together, the measurements show the benefits of (re-)using offset-value codes and support Hypothesis 5.

We tested Hypothesis 9 with an experiment similar to that of Hypothesis 5: an input sorted on A, B, C but needed sorted on A, C, B. A, B, and C are lists of 8 columns; all comparisons are decided by the last column. The counts of segments, i.e., of distinct values in A, vary by powers of 4 from 2 (a single segment boundary) to 2^{19} (segments of two rows). Data is generated in such a way that, as the size of a segment shrinks by a factor of 4, both the number of runs (i.e., distinct values of B within each segment) and the size of each run are halved. Data is sorted on A, C, B with three different methods, all using existing offset-value codes in the input. The first method segments the data by distinct values of A. Each segment is then sorted directly using a tournament tree, disregarding the existence of pre-sorted runs. Every row has its offset changed to $|A|$ with a value extracted from C, corresponding to runs of size 1. The second method does not segment the data, but instead generates runs with distinct pairs A, B and merges them in one process. The first row of each run has its offset changed to 0 and value extracted from A. Every

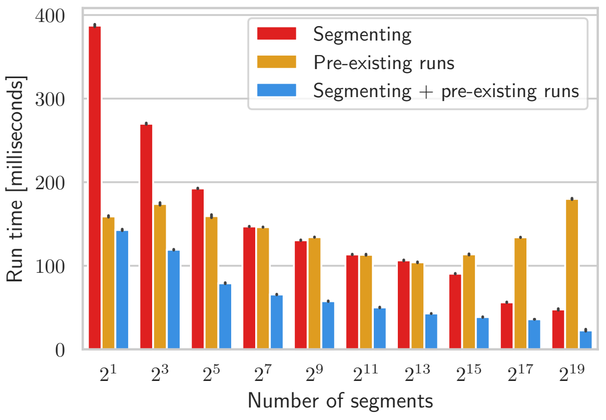

Figure 11: Run time of changing an existing sort order from A, B, C to A, C, B with 2^{20} rows and varying numbers of segments.

other row within the runs is treated in the same way as described in Section 3.4 and Figure 7. The third method combines these techniques by segmenting the input and merging runs within each segment.

Figure 11 shows run times of these methods. Sorting segments directly without utilizing existing runs is the slowest method for large segments (2 to 2^7 segments, i.e., 2^{19} to 2^{13} rows per segment). The run time of this method decreases with the size of the segments as it takes less effort to sort several small segments than fewer large ones. The second method segments data by AB and runs twice as fast as the first method for 2 segments. As the number of segments increases, the run time of this method decreases but eventually increases again. This increase is due to the fact that merging many short runs takes more comparisons than merging fewer long ones. The third method combines both techniques and consistently has the best run-times, supporting Hypothesis 9 about the cumulative benefits of segmented sorting and merging pre-existing runs.

4.5 Summary of the experimental evaluation

Overall, the arguments of Section 4.2 and the experiments of Section 4.4 support the hypotheses and claims of Section 4.1. While perhaps some of the experimental results were predictable qualitatively, the quantity of possible benefits should be surprising. If the extent of these benefits were already known and understood, the implementations of sorting in query execution as well as the definition of interesting orderings in query optimization would already be much more general than they are in today's database management systems.

5 SUMMARY AND CONCLUSIONS

In summary, it is wasteful to treat and sort an input as unsorted instead of exploiting an existing sort order. By exploiting old offset-value codes using the techniques introduced here, it is often possible to create a new sort order, to derive new offset-value codes, to compress by prefix truncation, and to transform into columnar format with compression by run-length encoding – all entirely without column value comparisons. Our claims, hypotheses, and experiments show that substantial savings can be realized.

Those savings, however, require extensions to both traditional query execution algorithms and traditional query optimization heuristics. Query execution needs segmented sorting or, more generally, segmented query execution; and external merge sort must skip the run generation phase when suitable runs are readily available in the input. In query optimization, the concept and the enforcement of interesting orderings must be generalized beyond the basic support in today's database query optimizers.

These capabilities in query optimization and query execution also affect physical database design. The example with enrollments of students in courses shows how any many-to-many relationship can support efficient join queries with fewer copies and fewer indexes if case 3 in Table 1 is supported by merging runs pre-existing in the sorted input; a subsequent discussion (using a multi-campus university) extends this argument to case 5 and a side note extends it to case 7 in Table 1.

Offset-value codes in a sorted input reflect the prior sort order. Scans of b-trees with prefix truncation can readily supply offset-value codes, as can scans of sorted tables in columnar databases with run-length encoding. A new but very simple adaptation adjusts offsets and thus permits reusing the input's offset-value codes when modifying an existing sort order. The final offset-value codes of the new sort order support, without further column value comparisons, compressing the sorted output by prefix truncation in rows or by run-length encoding in columns.

In conclusion, it has long been known, even if not widely, that offset-value coding substantially speeds up internal and external merge sort; that offset-value coding and tree-of-losers priority queues (tournament trees) complement each other very well; and that their core logic and inner-most loops are so simple and concise that they have been captured in a pair of mainframe instructions. Recent research has expanded offset-value codes to sort-based query execution algorithms including merge join, set operations such as intersection, duplicate removal, grouping, and pivoting. The essential insight is that offset-value codes cache comparison efforts; these recent techniques extend the cache effects from merge sort to sort-based query execution.

The work reported here extends offset-value coding and its caching effects beyond all earlier work. While the new techniques also apply to intermediate query results, their greatest impact may be in scanning and merging from sorted storage structures, e.g., ordered column stores and b-tree indexes.

ACKNOWLEDGMENTS

We thank Xiaoyu Wang and Herald Kllapi for their detailed and encouraging comments. The German Research Foundation (DFG) supported parts of this work with grant SE 553/9-1.

REFERENCES

[1] Timo Bingmann, Peter Sanders, and Matthias Schimek. 2020. Communication-Efficient String Sorting. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 137–147.

[2] Douglas Comer and Ravi Sethi. 1977. The Complexity of Trie Index Construction. *J. ACM* 24, 3 (1977), 428–440.

[3] W. M. Conner. 1977. Offset-value coding. *IBM Technical Disclosure Bulletin* (1977), 2832–2837.

[4] Thanh Do, Goetz Graefe, and Jeffrey F. Naughton. 2022. Efficient Sorting, Duplicate Removal, Grouping, and Aggregation. *ACM Trans. Database Syst.* 47, 4 (2022), 16:1–16:35.

[5] Edward H. Friend. 1956. Sorting on Electronic Computer Systems. *J. ACM* 3, 3 (1956), 134–168.

[6] Martin A. Goetz. 1963. Internal and tape sorting using the replacement-selection technique. *Commun. ACM* 6, 5 (1963), 201–206.

[7] Herman H. Goldstine and John von Neumann. 15 April 1948. *Planning and coding of problems for an electronic computing instrument. Part II, Volume II, Section 11: Coding of some combinatorial (sorting) problems.* Technical Report. The Institute for Advanced Study, Princeton NJ.

[8] Goetz Graefe. 2006. Implementing sorting in database systems. *ACM Comput. Surv.* 38, 3 (2006), 10.

[9] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (2011), 203–402.

[10] Goetz Graefe. 2011. Sorting in a Memory Hierarchy with Flash Memory. *Datenbank-Spektrum* 11, 2 (2011), 83–90.

[11] Goetz Graefe. 2023. Priority queues for database query processing. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), Dresden, Germany (LNI)*, Vol. P-331. Gesellschaft für Informatik e.V., 27–46.

[12] Goetz Graefe. 2024. More Modern B-Tree Techniques. *Found. Trends Databases* 13, 3 (2024), 169–249.

[13] Goetz Graefe and Thanh Do. 2023. Offset-value coding in database query processing. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023.* OpenProceedings.org, 464–470.

[14] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (1961), 321.

[15] IBM. 1988. *Enterprise System Architecture/370, Principles of Operation.* IBM publication SA22-7200-0.

[16] Balakrishna R. Iyer. 2005. Hardware Assisted Sorting in IBM's DB2 DBMS. In *Advances in Data Management 2005, 12th International Conference on Management of Data, COMAD 2005b, December 20-22, 2005, Hyderabad, India.* Computer Society of India.

[17] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece.* Morgan Kaufmann, 16–25.

[18] Donald E. Knuth. 1970. Von Neumann's First Computer Program. *ACM Comput. Surv.* 2, 4 (1970), 247–260.

[19] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley.

[20] Harry Leslie, Rohit Jain, Dave Birdsall, and Hedieh Yaghmai. 1995. Efficient Search of Multi-Dimensional B-Trees. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.* Morgan Kaufmann, 710–719.

[21] Waihong Ng and Katsuhiko Kakehi. 2008. Merging string sequences by longest common prefix. In *IPSJ Digital Courier*, Vol. 4. 69–78.

[22] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[23] Jack A. Orenstein and T. H. Merrett. 1984. A Class of Data Structures for Associative Searching. In *Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Waterloo, Ontario, Canada.* ACM, 181–190.

[24] Peter Scheuermann and Aris M. Ouksel. 1982. Multidimensional B-trees for associative searching in database systems. *Inf. Syst.* 7, 2 (1982), 123–137.

[25] Raimund Seidel. 2010. Data-Specific Analysis of String Sorting. In *Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010.* SIAM, 1278–1286.

[26] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1.* ACM, 23–34.

[27] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. 1996. Fundamental Techniques for Order Optimization. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings (Lecture Notes in Computer Science)*, Vol. 1057. Springer, 625–628.

[28] Hermann Tropf and Helmut Herzog. 1981. Multimensional Range Search in Dynamically Balanced Trees. *Angew. Inform.* 23, 2 (1981), 71–77.

[29] Martin Zirkel, Volker Markl, and Rudolf Bayer. 2001. Exploitation of Pre-sortedness for Sorting in Query Processing: The TempTris-Algorithm for UB-Trees. In *International Database Engineering & Applications Symposium, IDEAS '01, July 16-18, 2001, Grenoble, France, Proceedings.* IEEE Computer Society, 155–166.