Adaptive sorting for large keys, strings, and database rows

Marius Kuhrt [1], Bernhard Seeger [1], Sebastian Wild [2], and Goetz Graefe [3]

Abstract: As sorting a database table may require expensive comparisons, e.g., due to column count or column types such as long or international strings, optimizing the count and cost of comparisons is important. Adaptive sorting avoids comparisons by exploiting pre-existing order in the input. If N keys happen to be sorted or reverse-sorted, $N-1$ comparisons suffice, in contrast to $\log_2(N!)$ comparisons in the expected and worst cases. Ideally, adaptive sorting ensures graceful degradation from the best case to the expected case, e.g., by merging sorted runs pre-existing in the input.

Adaptive sorting has proven successful for integer keys but not for large keys, e.g., when comparing symbols or characters in text strings, bytes or words in binary strings, or column values in database rows. On the other hand, using longest common prefixes or offset-value codes, sorting N strings with K characters, bytes, or columns can be limited to $N \times K$ comparisons. If those comparisons are the dominant cost of sorting, e.g., due to interpreted execution of predicates etc., the cost of sorting is linear in the input size and, in fact, equal to the cost of verifying a claimed and correct sort order.

By leveraging and combining a variety of techniques, some old and proven in practice, some new yet equally sound, we introduce sorting techniques that are efficient, scalable, and adaptive; that degrade gracefully from $N-1$ to $\log_2(N!)$ comparisons of strings or of database rows; and that guarantee at most $1.042N \times K$ comparisons of bytes or of column values in the worst case. We offer algorithm analyses and experimental results to test our hypotheses and support our claims.

1 Introduction

The need for efficient sorting is not new. For example, early work on database query optimization [Se79] and its focus on “interesting orderings” are still relevant, in particular in the context of B-tree indexes and log-structured merge-forests [Ja97, O’96]. An earlier example is the hardware sorter for census records on punch cards [Ho89]. More recent work includes order-based plans for computing data cubes [Gr97] using carefully planned roll-up operations, e.g., in [Ag96].

Reducing sort effort by exploiting pre-existing order is also not new. For example, Friend credits that the “von Neumann approach (named after its originator, John von Neumann) takes advantage of existing sequences in the data” [Fr56, Kn70, GvN48]. Burge states that “the optimum strategy is shown to depend on the order already existing in the data” [Bu58]. Other early work includes stringsort [Bo63]: “The procedure takes advantage of naturally

[1] University of Marburg, Marburg, Germany,
kuhrt@informatik.uni-marburg.de, https://orcid.org/0009-0000-3926-6218;
seeger@informatik.uni-marburg.de, https://orcid.org/0000-0002-9362-153X

[2] University of Marburg, Marburg, Germany and University of Liverpool, Liverpool, UK,
wild@informatik.uni-marburg.de, https://orcid.org/0000-0002-6061-9177

[3] Google, Madison, Wisconsin, USA, goetzg@google.com, https://orcid.org/0000-0003-0194-6466

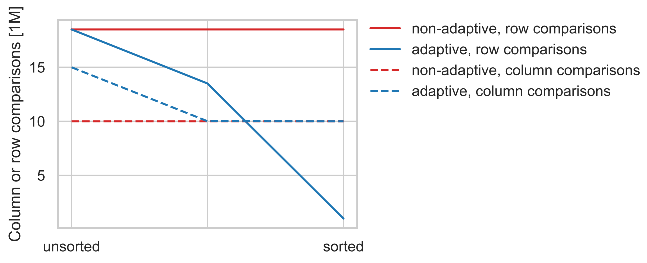


Fig. 1: Worst-case counts of row and column value comparisons in adaptive and non-adaptive mergesort (original expectation).

occurring ascending and descending order in the original data." Recent work includes adaptive sorting for integers [MW18, Au18, BK19, Ju20, GJK22] as well as modifying a sort order by exploiting order known to exist in the input and by reusing sort and comparison effort cached in offset-value codes within the input [Co77, GKS25].

In contrast to this recent work, we focus here on *adaptive* sorting algorithms that detect and exploit any *incidental* order found in the input – with no assumptions and without tailoring algorithms to specific contexts or use cases. In contrast to earlier adaptive sorting algorithms such as Timsort and Powersort [MW18], we focus on large keys such as text strings, binary strings, and database rows. In contrast to earlier work on sorting text strings [BSS20] and on hardware support for sorting binary strings [Iy05], we focus on adaptive sorting. In agreement with recent work on sorting in databases [Gr06, KRM21], we focus on database tables in row format rather than column format. Considering long-deployed and proven sorting hardware [Iy05, IB88] and recent advances in sort-based query processing [GD23], we focus on efficiently sorting strings by caching comparison effort in offset-value codes [Co77, DG23], which generalize sorting and merging with longest common prefixes [BSS20, NK08].

With these contrasts to earlier work, our new contributions are:

C1. efficient adaptive sorting for large and very large keys;

C2. with tournament trees, graceful degradation from $N - 1$ to $\log_2(N!)$ comparisons of N strings or database rows;

C3. with offset-value coding, linear cost for character, byte, or column value comparisons, specifically $N \times K$ without adaptive techniques (for key size K) and $1.042N \times K$ in the worst and most unfortunate case for adaptive techniques;

C4. initiation of offset-value codes during the search for pre-existing sorted runs;

C5. offset-value codes as side effect of insertion sort (e.g., within Powersort);

C6. algorithm analyses for adaptive sorting of large keys; and

C7. experimental validation of claims and hypotheses.

Figure 1 illustrates our initial expectations (hypotheses, not measurements) for the worst-case comparison counts of our recommended adaptive and non-adaptive sorting algorithms. These counts of row comparisons and of column value comparisons apply to both internal and external mergesort. All cases sort $N = 10^6$ rows with $K = 10$ columns. For simplicity of this initial analysis, the last column decides all comparisons. All variants use offset-value coding. Along the x-axis, Figure 1 shows three cases: unsorted inputs on the left, fully sorted inputs on the right, and 10^3 runs of 10^3 keys as an intermediate "half-sorted" case. A non-adaptive sort algorithm is oblivious to the order and thus requires the same comparison counts for presorted and unsorted inputs, hence the constant (red) lines. The adaptive sorting algorithm (blue) exploits presorted inputs. Its count of *row* comparisons is nearly optimal for unsorted inputs ($\log_2(N!) \approx 18.5M$) and for fully presorted input ($N - 1 \approx 1M$); and it degrades gracefully across the entire range. Its count of *column* value comparisons is $N \times K \approx 10M$ in good cases and $1.5\times$ as much in its worst case.[4] The worst case is an input with many runs and thus many run boundaries: the column value comparisons within runs can be cached by offset-value codes and hence are never repeated; the column value comparisons at run boundaries cannot be cached in this way and thus *add* to the total count of column value comparisons.

To summarize our hypotheses and expectations, we expect that

1. internal and external mergesort invoke row and column value comparisons near the provable lower bounds in both non-adaptive and adaptive variants;
2. adaptive mergesort exploits pre-existing sorted runs and degrades gracefully;
3. even adaptive mergesort requires linear effort for column value comparisons; and
4. its non-cacheable column-value comparisons never exceed half their required count.

It was thus our initial hope that the benefits of adaptive sorting and offset-value coding are orthogonal and that, ideally, they could be enjoyed simultaneously without prohibitive overhead. Can we have the cake and eat it, too?

The following sections review related prior work including required background information, then introduce and analyze techniques for adaptive sorting of large keys, list hypotheses and 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 mergesort, quicksort, and priority queues [Fr56, Go63, Ho61, Kn73]. Segmented sorting, merging pre-existing runs, and their combination are well known [Gr06].

[4] The factor 1.5 from our initial expectation is never actually required in our algorithms; as explained below, a standard optimization limits the actual worst case to the stated $1.042N \times K$.

2.1 Adaptive sorting for integers

Adaptive sorting refers to the capability of a sorting algorithm to profit (i.e., becoming faster) from existing (partial) order in the input; in the context of this work, existing order will mean the existence of nontrivial contiguous sorted *runs* in the input. Algorithm-theory work in this area assumes a list of atomic (black-box) objects that can only be inspected by pairwise comparisons (with outcomes "<", "=", or ">") [ECW92]. In this *comparison-based model* of sorting, information-theoretic considerations yield lower bounds on the number of (black-box) comparisons required by any valid comparison-based sorting algorithm; for example, the *run-length entropy* $\mathcal{H} = \frac{\ell_1}{N}\log_2(\frac{N}{\ell_1}) + \cdots + \frac{\ell_r}{N}\log_2(\frac{N}{\ell_r})$ for inputs with (known) runs of lengths $\ell_1, \ldots, \ell_r$, respectively, captures the sorting complexity [BN13, Thm. 2]: for sorting such an input, $N\mathcal{H} \pm O(N)$ comparisons are necessary and sufficient. For an intuition about $\mathcal{H}$, note that $N\mathcal{H}$ is an asymptotic approximation of $\log_2(N!) - \log_2(\ell_1!) - \cdots - \log_2(\ell_r!)$, i.e., what we save over resp. subtract from the cost of sorting N objects in general is precisely the cost of producing the existing runs by sorting.

Approaching this lower bound (up to linear extra cost) on the number of comparisons with otherwise very low overhead over a non-adaptive sorting method has lead to Timsort and its more recent refinement, Powersort [MW18], which has become the de-facto standard for internal stable sorting: It is used (in slight variations) for Python's built-in list sorting method (in both the CPython reference implementation and PyPy), and in the Java runtime library, the Android Java runtime, in the Chrome V8 JavaScript engine, as well as in the standard libraries of Rust, Swift, Apache Spark, Octave, and the NCBI C++ Toolkit.

Timsort is a variant of binary mergesort that maintains a stack of runs yet to be merged and alternates between finding the next run and merging adjacent runs near the top of the run stack. Additionally, Timsort enforces a minimal run length by "filling up" short runs using insertion sort up to a chosen minimal run length. Merge steps combine two adjacent runs into one run using a "galloping strategy": exponential search is used to find the prefix of one run that precedes the minimum in the other run using fewer comparisons than in a textbook linear merge. When either comparisons are very expensive, e.g., because they require interpreted client code, or when many duplicates are expected [GJK22], this is a prudent choice; otherwise, e.g., for integer sorting, where comparisons are cheap, galloping typically *increases* the overall elapsed time [MW18]. Hence, several of the software frameworks listed above replaced galloping with linear merge. We will also follow this choice, since the galloping strategy to-date forgoes the benefits offered by offset-value codes or longest-common-prefix values for sorting strings or database rows.

Powersort improves Timsort in terms of the *merge policy*, i.e., the rule that decides which runs on the run stack are merged before proceeding. Timsort's original policy used a suboptimal heuristic based solely on the lengths of runs; Powersort replaces this with a rule simulating Mehlhorn's algorithm for computing nearly optimal binary search trees [Me77, MW18] with low overhead, thereby achieving optimal adaptivity up to an additive linear term.

Timsort was originally designed for internal sorting and could only merge two runs at a time; Powersort has recently been extended to realize the benefits and the adaptivity of multi-way merging [Ge23]. Thus, Powersort is also applicable to external sorting.

2.2 Modifying an existing sort order

Whereas adaptive sorting detects and exploits *incidental* pre-existing order, modifying an existing sort order can rely on ordering and, ideally, offset-value codes for the input. In fact, recent work [GKS25] introduced techniques to adjusting offset-value codes for the input in order to create offset-value codes for the output and for speeding up the merge steps required to modify a sort order.

For example, when a table is sorted on A, B, C, D but is needed sorted on A, C, B, D, the values of A lead to segmented sorting (one small sort per distinct value of A), distinct values of B define runs to be merged, values of C define sorted input into merge steps, and values of D can be treated as duplicates. A–D can be columns, lists of columns (e.g., database rows), lists of characters (i.e., text strings), or lists of bytes (e.g., in normalized keys). Offsets (within offset-value codes for the input) permit classifying input rows without any column value comparisons. Adjusting offsets (subtracting the size of B or adding the size of C) permits creating offset-value codes for the output and its sort order also without column value comparisons. Thus, modifying an existing sort order may be accomplished without any column value comparisons at all (unless C is a list).

While previous work used tailor-made approaches to address a known re-ordering scenario like the above example by specifically exploiting knowledge about existing and desired sort order, our ambition in this work is similar performance with a *single* sorting method that simultaneously addresses all possible re-ordering tasks.

2.3 Managing rows, strings, keys, and indirection

Discussions on sorting keys and values, where both keys and values may be sizable strings, always touch upon indirection. This is an old issue with multiple known techniques and approaches [Fr56, Hu63]. For example, Nyberg et al. [Ny95] invoke quicksort [Ho61] on an array of pairs, each comprising a pointer to a record elsewhere in memory and a binary fixed-size order-preserving key prefix, also known as “poor man’s normalized key.” Similarly, the string representation in Umbra [NF20] stores a 4-byte prefix next to the pointer. We ignore this issue and the related issues of CPU cache traffic. Instead, the principal cost metric in our analyses and experiments is the count of comparisons of rows and column values (or of strings and characters); see Section 4 for more details.

2.4 Offset-value coding[5]

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. Transposition from column format with run-length encoding to row format with prefix truncation is fast as it requires no column value comparisons. For text strings and string sorting, prefixes and prefix truncation are discussed as longest common prefix (LCP) [NK08] and work very well with tournament trees (tree-of-losers priority queues) [BSS20]. A refinement of prefix truncation, offset-value coding 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 not only with lists of column values, i.e., database rows, but also with 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 decide a comparison. Such a comparison is much faster than comparing rows column-by-column. When two rows' offset-value codes are equal, column-by-column comparisons can resume after the shared prefix and a modified offset-value code in the loser caches the additional comparison effort. Within text strings, the required logic is like `strcmp()` with starting and ending offsets; within binary strings such as normalized keys, it is like `memcmp()` with starting and ending offsets. In run generation and merging using tournament trees, offset-value codes often decide many or most row comparisons.

	Col. index				Prefix truncation					Descending offset-value codes			Ascending offset-value codes		
	0	1	2	3	prefix size	0	1	2	3	offset	domain – value	OVC	arity – offset	value	OVC
Rows and their column values	5	4	7	1	0	5	4	7	1	0	95	95	4	5	405
	5	4	7	2	3				2	3	98	398	1	2	102
	5	6	2	6	1		6	2	6	1	94	194	3	6	306
	5	6	2	6	4					4	-	500	0	-	0
	5	6	3	4	2			3	4	2	97	297	2	3	203
	5	8	2	3	1		8	2	3	1	92	192	3	8	308
	5	8	4	7	2			4	7	2	96	296	2	4	204

Fig. 2: Derivation of prefix truncation and descending/ascending offset-value codes for a table sorted on 4 keys.

Figure 2 shows, on the left, rows in ascending order on all four columns. The remainder of the diagram illustrates the derivation of compression by prefix truncation and of both descending and ascending offset-value codes. The prefix size counts leading column values equal to the preceding row. 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

[5] The sections on offset-value coding and tournament trees are adapted from [GKS25]; we include them here for a self-contained presentation.

column is 100. Descending offset-value codes take the actual offset but 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 shared 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 has extended offset-value coding from mergesort to merge join, duplicate removal, and in fact most sort-based query execution [GD23]. Offset-value coding within external mergesort can speed up the in-sort logic for "distinct", "group by", "pivot", "limit", and "offset" queries, the in-stream logic for the same unary operations, as well as merge join and other binary operations, even for operators and pipelines with interesting orderings.

In many ways, offset-value codes serve in sorting and in sort-based query execution the roles of hash values in hash-based query execution [Gr23]. 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 only assert that two rows (or their keys) are different, offset-value codes can also assert that their equality or their sort order.

2.5 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 [Go63] for internal sorting, for run generation by replacement selection, and for merging sorted runs. They reduce the count of row comparisons to nearly the theoretical lower bound, i.e., $\log_2(N!) \approx N \times \log_2(N/e)$ for N rows and $e = 2.718... \approx 19/7$.

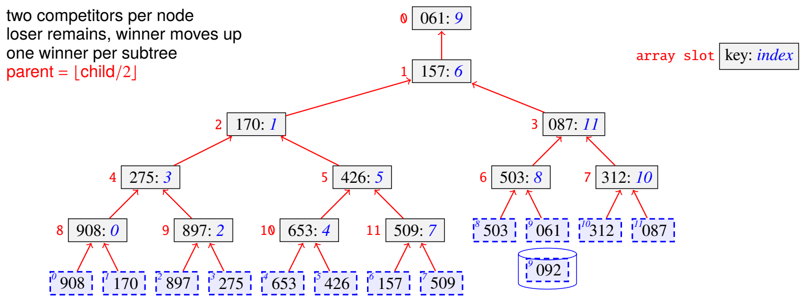


Fig. 3: A tournament tree with 12 merge inputs. The example node in the top-right corner explains the meaning of the numbers.

Figure 3 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. 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 it met and lost to the overall winner.

Tournament trees work very well with offset-value codes. Together, they 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 [CS77, Se10]. 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" [IB88, Iy05].

2.6 Other related prior work

The external mergesort of System R includes a simple yet effective adaptive technique [Hä77]. During run generation, it retains the last (highest) key value of four recent runs. Before creating a new run from sorted memory contents, it attempts to append all keys in memory to a recent run. This idea can be adapted to cache-size runs within memory.

Merge planning in external mergesort [Gr06] ensures that all merge steps employ the maximal fan-in, with the initial merge step the only exception, and that each merge step consolidates the smallest remaining runs. This idea may be applied to disjoint key ranges. In many cases, it renders the prior idea obsolete. For merge steps invoked prior to end-of-input, e.g., in log-structured merge-forests [O'96] and similar structures, the planning goal is to merge the most similar row counts in entire runs or within a key range.

3 Techniques for adaptive string sorting

This section contains the techniques used to modify Powersort to use offset-value coding, namely run detection, binary merging and insertion sort. They are, of course, applicable to other merge-based sorts as well.

3.1 Detecting ascending/descending runs

During run detection, the input to be sorted is scanned for presorted sub-sequences in either ascending or descending order. Many of the character comparisons performed can be cached and re-used in the merge process using offset-value coding. The direction (i.e. ascending or descending) of a run is decided by comparing the first two keys. Weakly increasing runs are considered ascending while strictly decreasing runs are considered descending. This is necessary to maintain stability of the sort as decreasing runs are simply reversed before merging. While scanning for a run, the comparison between the previous and a new key naturally provides the length of the common prefix (i.e. the offset), allowing the offset-value code to be calculated without extra effort. If the run currently being scanned is ascending, the first key is assigned an offset-value code with offset 0, while additional keys are assigned offset-value codes calculated during the comparisons, and have, therefore, the the immediate predecessor as a base. If the run is descending, offset-value codes produced during the comparison are assigned to the previous key, and the last key (i.e. the first key, after reversal) is assigned the offset-value code with offset 0. Afterwards, the descending run is reversed into ascending order. Only the detection of a run boundary incurs character comparisons that are not cached in offset-value codes and can not be reused in the merge logic.

3.2 Merging with offset-value codes

Merging with offset-value codes is, of course, well known and documented [Gr23, Iy05], but we include the algorithms for binary merging for completeness.

Algorithm 1 demonstrates the comparison logic for two strings s_1 and s_2 of length l with ascending offset-value codes with respect to the same base row. It returns a number <, = or > 0 if s_1 is <, = or > than s_2, respectively, and sets the offset-value code of the larger string with the smaller one as a base. Lines 1 and 2 contain the favorable case where the offset-value codes differ and the string comparison is decided by a single integer comparison. If both offset-value codes are 0, indicating duplicates with the base for ascending offset-value codes, the comparison returns 0 (lines 4 and 5). Otherwise, the strings are compared byte-wise, beginning at the position past the common prefix (line 7). If a difference is found, the offset-value code of the larger string is calculated with the current iteration index i as offset and the byte of that same string at position i (lines 11 and 14). If no difference is found and the comparison loop finishes, the offset-value code for the second string s_2 is set to 0, indicating a duplicate of s_1. In all cases, when the function returns, the larger string maintains an offset-value code with respect to the smaller string, while the offset-value code of the smaller string remains unchanged.

For zero-terminated strings, possibly of different lengths, the while loop (line 8) is replaced with an endless loop and a check is introduced after line 17. If this check is reached, the currently compared strings coincide and the loop is broken if either of them is a zero-byte,

```
    Data: $s_1, s_2$ strings of length $l$ with offset-value codes relative to the same base row
    Result: $n \in \mathbb{Z}$ s.t. $n \leq 0 \Leftrightarrow s_1 \leq s_2$; larger string has OVC w.r.t the smaller one
 1  if $s_1.ovc \neq s_2.ovc$ then
 2      return $s_1.ovc - s_2.ovc$;
 3  else
 4      if $s_1.ovc = 0$ then
 5          return 0;                       /* both strings are duplicates of the base */
 6      end
 7      $i \leftarrow offset(s_1.ovc) + 1$;
 8      while $i < l$ do
 9          if $s_1[i] \neq s_2[i]$ then
10              if $s_1[i] < s_2[i]$ then
11                  $s_2.ovc \leftarrow (i, s_2[i])$;       /* new OVC relative to the winner */
12                  return $-1$;
13              else
14                  $s_1.ovc \leftarrow (i, s_1[i])$;       /* new OVC relative to the winner */
15                  return 1;
16              end
17          end
18          $i \leftarrow i + 1$;
19      end
20      $s_2.ovc \leftarrow 0$;                               /* new duplicate found */
21      return 0;
22  end
```

Algorithm 1: CmpOVC for ascending offset-value codes

indicating a duplicate. The remaining comparison logic (lines 9 to 17) is left unchanged because the comparison of a character with a zero-byte results in the correct outcome.

```
    Data: $S_1, S_2$ sorted sequences of rows with offset-value codes
    Result: $S_0$ merged sorted sequence of rows with offset-value codes
 1  $i_1 \leftarrow 0$;
 2  $i_2 \leftarrow 0$;
 3  for $i \leftarrow 0$ to $|S_1| + |S_2| - 1$ do
 4      if $CmpOVC(S_1[i_1], S_2[i_2]) \leq 0$ then
 5          $S_0[i] \leftarrow S_1[i_1]$;
 6          $i_1 \leftarrow i_1 + 1$;
 7      else
 8          $S_0[i] \leftarrow S_2[i_2]$;
 9          $i_2 \leftarrow i_2 + 1$;
10      end
11  end
```

Algorithm 2: MergeOVC

Algorithm 2 shows the binary-merge logic using offset-value codes. S_1 and S_2 are sorted sequences of strings with offset value codes. For simplicity it is assumed that the sequences end with sentinel values that compare higher than any string. The logic compares the next keys in each sequence using CmpOVC and appends the smaller string to the output sequence

S_0. This process works correctly because the two keys compared in line 4 always hold offset-value codes with respect to the last string in the output sequence. In the very first iteration, the base string for the two smallest strings in the inputs is the (imaginary) smallest string, the offsets for the first two strings are 0. Afterwards, this invariant is maintained by CmpOVC. Note that, in case of duplicates, the logic chooses the key from the first sequence. This is in line with the fact that, in this case, CmpOVC sets the offset-value code of the second argument as a duplicate with respect to the first argument.

3.3 Insertion sort with offset-value codes

Timsort and Powersort perform insertion sort to extend short pre-existing runs before they are included in the merge process. There are several ways to find the correct position to insert a new key into the sorted sequence, some of which are unsuited for sorting strings with offset-value codes because they perform comparisons that cannot be cached and reused.

If binary search is used, existing offset-value codes in the sorted sequence are of no value, as the new string does not have an offset-value code with any other string as a base and full string comparisons must be performed. In this process, only two comparisons can be reused by caching them in offset-value codes: the comparisons with the very next smaller string and the next larger string allow for the computation of offset-value codes for the new row, and the next larger row, respectively.

Similarly, when probing linearly from the high end of the sorted sequence, only the last two comparisons are cached by encoding them in the offset-value codes for the new, and the next larger string.

If, however, the sorted sequence is probed from the low end, offset-value codes can be used effectively to find the correct position and cache every character comparison employed in the process. This search essentially is a binary merge of the sorted sequence with the new key, i.e., the new key is both the first and the last key in a singleton run. The offset-value code of the new string is initialized with offset 0 and it is then compared against the sorted sequence using the logic of Algorithm 1.

3.4 Summary of techniques

In summary, the new techniques allow for adaptive sorting while fully utilizing offset-value coding. Run detection with offset-value coding caches most of the performed comparisons so that they can be utilized in the merge process while insertion sort can be used to efficiently extend short runs, caching all comparisons in the process.

4 Algorithm analyses

In this section, we define and analyze abstract cost measures for adaptive string sorting, i.e., operations whose total cost serves as a simple model to predict elapsed-time cost of our implementations. We will express some results in terms of properties of the input (specifically its run-length entropy $\mathcal{H}$ and prefix-truncation potential $\mathcal{P}$, both defined below); in Section 5, we compare these on specific benchmark datasets. We assume that we sort an array of strings $A[0..N)$ where each $A[i]$ is a string of length K over the alphabet $\Sigma = [0..\sigma) = \{0, \ldots, \sigma - 1\}$. We will assume $\sigma = 256$, so characters are bytes.

A *string comparison* happens whenever an algorithm asks for two strings x and y, which of the following three cases occurs: $x < y$, i.e., x precedes y in lexicographic order, $x = y$, or $x > y$. To implement a string comparison, our algorithms use *character comparisons*, i.e., comparisons of individual letters $x[j]$ and $y[j]$ in two strings x and y, again with the outcomes "<", "=", ">". A single string comparison between x and y can cost anything between 1 and K character comparisons since we have to find the longest common prefix of x and y, i.e., the $LCP(x, y)$ initial characters that x and y share, to decide the comparison. By remembering longest-common-prefix values or (even more so) using offset-value codes, some character comparison normally executed, but known not to decide a string comparison from past comparisons or cached values are now skipped. We do not count such implicit character comparisons (since they need no computation whatsoever).

Significant contributors to the cost of sorting thus are the number of character comparisons and the number of longest-common-prefix value resp. *offset-value-code comparisons*. Note that every string comparison triggers exactly one offset-value-code comparison in all our offset-value-code-based algorithms, so their numbers are equal.

4.1 String comparisons

The number of string comparisons exactly matches the comparisons in the theoretical comparison model of prior work on adaptive sorting. The corresponding analyses of Powersort [MW18, Thm. 5] hence apply verbatim for string comparisons: for any input with runs of respective lengths $\ell_1, \ldots, \ell_r$, our Powersort implementation never uses more than $\mathcal{H}N + O(N)$ string/offset-value-code comparisons, where $\mathcal{H} = \sum_{i=1}^{r} \frac{\ell_i}{N} \log_2(\frac{N}{\ell_i}) \le \log_2 r$ is the "run-length entropy" of the input. The constant in $O(N)$ depends on how the minimal-run-length requirement is implemented. Improving this bound will be subject to future work.

4.2 Character comparisons

Clearly, an algorithm using c string comparisons uses never more than cK character comparisons, so this also bounds the number of character comparisons. Orthogonal to that, when using longest-common-prefix or offset-value-code information in mergesort as described in Section 3, any character comparison done during merging that has outcome “=” will increase an LCP/offset by one. Since these values are never decreased and, in the final sorted list A, correspond to the dataset’s *prefix truncation potential* $\mathcal{P} = \sum_{i=1}^{N-1} LCP(A[i], A[i+1])$, i.e., the length of longest common prefixes of adjacent strings in sorted order, the number of “=”-outcome character comparisons during mergesort based on longest common prefixes is bounded by $\mathcal{P}$. On the other hand, any sorting algorithm has to discover all common prefixes to certify the correctness of the produced order. The total number of “=”-outcome character comparisons hence is at least $\mathcal{P}$. When using offset-value codes, some of these character comparisons might be decided implicitly via a single offset-value-code comparison. Any character comparison with outcome “<” or “>” decides a string comparison, so their number is bounded by the number of string comparisons and the string sizes.

This leaves “=”-outcome character comparisons that are not part of merging. In a classic (non-adaptive) mergesort, all comparisons are part of merging. In our adaptive methods, most comparisons during run detection – namely those that extend the detected run – are likewise advancing the longest common prefix or offset and thus behave just like comparisons during a merge. Only the string comparisons that detect the *end of a run* can trigger non-cached “=”-character comparisons with the first key of the subsequent run. These two keys are not part of a single run, hence the character comparisons are not cached/reused; they are thus in addition to $\mathcal{P}$. The number of run ends is $r - 1$, where r is the number of pre-existing runs; each contributes at most K non-cached character comparisons. Hence at most $(r-1)K$ character comparisons are done that cannot be reused.

4.3 Number of runs

In the worst case, $r = N/2$, which yields the $1.5NK$ worst case of our initial expectation. A (nontrivial) analysis shows that the average case for a uniform random permutation is not far from this: the expected number of runs is αN for $\alpha = 1 + 2\tan(1) - 2/\cos(1) \approx 0.4132$ [he]. However, the more sorted the input, the smaller r; as the input gets more sorted, the number of non-cached character comparisons during run detection approaches 0.

As is standard in adaptive sorting methods, our Powersort variants actually enforce a minimal run length (we use 24) by “filling up” any occurring runs that are shorter than this minimal length. Conventionally, insertion sort is used for this purpose. As described in Section 3, insertion sort behaves like a sequence of (degenerate) merge steps and, in particular, all character comparisons apart from those ending the *newly created* run are now cached. For counting extraneous “=”-comparisons, r is effectively bounded by $N/24$, for a total of $\mathcal{P}N + (\frac{N}{24} - 1)(K - 1) \leq 1.042NK$ “=”-comparisons.

4.4 Summary of algorithm analysis

The number of string comparisons is substantially reduced using adaptive sorting, whereas the character comparisons are mostly predetermined and optimal for any merging-based method that caches longest-common-prefix values – except for some character comparisons during run detection. While our initial expectation was that in the worst case, the extraneous character comparisons could reach up to 50% of cost, this fraction is reduced to a negligible 4.2% by maintaining offset-value codes when extending runs to a minimal length of 24. From the perspective of abstract cost measures, Powersort with offset-value codes indeed realizes the combined saving of both techniques, at (almost) no detriment! In the next section, we investigate whether this theoretical analysis is also predictive of practical performance.

5 Performance claims and measurements

The following experiments try to refute or support specific hypotheses. First, the section introduces the hypotheses and discusses the ones that do not require an experimental evaluation. For the other ones, their implementation aspects are detailed, and the datasets are introduced. Finally, the results of the experiments are discussed in the context of the hypotheses to be examined.

5.1 Hypotheses and claims

The following claims and hypotheses delineate the value of advanced sorting, particularly adaptive sorting, and the impact of offset-value codes.

- H1. Adding offset-value coding to state-of-the-art adaptive sorting methods (i.e., Powersort) can yield significant performance improvements.
- H2. The improvements over algorithms without offset-value codes can be explained by the high number of comparisons that are decided by offset-value codes and the decreased number of character or column comparisons.
- H3. The costs (space and elapsed time) of initializing offset-value codes during the initial scan is negligible.
- H4. Encoding more characters or columns in the offset-value codes leads to performance improvements as offset-value codes decide more sort comparisons.

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

Hypothesis 1 will be tested in a variety of situations (Section 5).

For Hypothesis 2, we note that caching previous character comparisons and preventing expensive string comparisons are the two ways offset-value coding effects the sort logic. An experiment will be conducted to give more insight.

Regarding Hypothesis 3, we observe that twice as much space for the array and buffer is required, as the size of the individual keys is now 16 bytes instead of 8. However, the strings occupy most of the space that is not affected by offset-value codes. With respect to elapsed time, there is very little overhead for producing offset-value codes during the initial scan of the input. The reason is that pairs of strings must be compared byte-wise anyway until a difference is detected. The index of the first two different bytes is precisely the offset.

Hypothesis 4 might seem obvious because encoding more data in offset-value codes naturally improves their potency as surrogate keys. However, creating offset-value codes with additional value bytes is more expensive on little-endian machines as bytes have to be reversed before encoding. An experiment will show whether this hypothesis holds.

5.2 Implementation and system context

All the algorithms are implemented in C++17. The implementation of Powersort supplemented in [Ge23] has been extended to use offset-value codes and is used in the experiments. All variants of Powersort utilize the default configuration; in particular, short pre-existing runs are extended to 24 keys with insertion sort. In addition, some of our experiments examined two sorting implementations from the standard library: non-stable `std::sort` based on quicksort and the merge-based `std::stable_sort`.

Strings of characters are implemented as pointers to zero-terminated arrays of characters in memory, while byte strings of fixed length are not terminated and can contain zero-bytes. Zero-terminated strings without offset-value codes are compared in a simple loop via bytewise comparisons, whereas comparisons of byte strings of known length utilize the very fast `memcmp`. To provide meaningful comparisons, we use `glibc`'s portable implementation of `memcmp` without architecture-specific optimizations[6].

Offset-value codes, if used, are stored next to the corresponding pointer in memory. Thus, the size of the keys being sorted on a 64-bit machine is 16 bytes instead of 8.

All experiments were performed on a workstation (AMD Ryzen 7 5800X @ 3.8GHz, 2x16GB DDR-4 @ 3200MHz, Linux 6.10.6), using `g++` 14.2.1 with optimization flags `-O3 -Ofast -march=native -mtune=native`.

[6] We note that `memcmp` with architecture-specific optimizations such as AVX improved elapsed times by a factor of up to 2 for bytestrings longer than 16 on the used system. We reserve it for future work to devise similarly optimized implementations of offset-value codes.

5.3 Datasets

Our experiments utilized four different datasets with different characteristics. Two of them are derived from a TPC benchmark, while the others represent real-world data sets. Table 1 summarizes the characteristics of the datasets.

Tab. 1: Characteristics of the datasets used: the number of items (N), the total size of the data, the average length of each string in bytes, and the percentage of duplicates in the dataset. The last column displays the compression opportunity using prefix truncation on the sorted dataset and gives an idea about how deep the strings are inspected during sorting.

Dataset	N	size	avg. length	duplicates	prefix truncation potential
TPC(SF=1)	1.4M	22MiB	16	0%	87.6%
TPC(SF=10)	14.4M	219MiB	16	0%	86.8%
URLs	122.3M	7.6GiB	64.8	7.9%	62.6%
LCC	166.8M	109MiB	5.8	90.8%	96.8%

The TPC(SF=n) datasets were created using the TPC-DS [Pö02] data generator with scale factor n. The two primary key columns, `cs_item_sk` and `cs_order_number`, of the `catalog_sales` table were turned into normalized keys by reversing the bytes of each 8-byte integer and concatenating them into binary strings of length 16.

The URLs dataset was produced from the Common Crawl[7] June 2024 Crawl Archive. It contains 122.3 million URLs with the German `.de` top-level domain that were extracted from the URL index files within the archive. The strings are of the form `de,domain,subdomain,subsubdomain,...)/<path>` and have varying lengths of up to a maximum of 2042 characters with an average of 64.8.

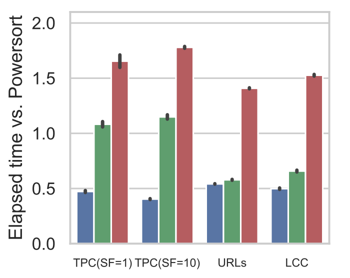
The LCC dataset was built from the Leipzig Corpora Collection [GEQ12] `deu_news_2023_1M` corpora. The corpora contains 1 million lines scraped from German news articles, which were then split into 166.8 million words. Notably, this dataset contains over 90% duplicates.

5.4 Experiments and measurements

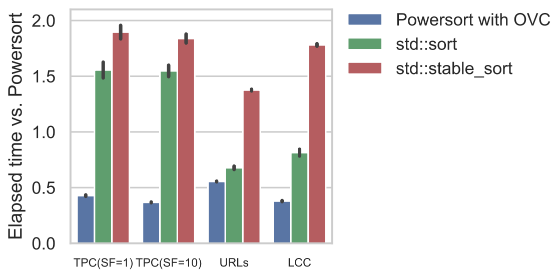
All diagrams in this section show averages and standard deviations of 100 repetitions when sorting the the TCP(SF=1) dataset, 20 repetitions for TPC(SF=10) and 5 repetitions for URLs and LCC.

We tested Hypothesis 1 by reporting the elapsed time relative to the one of standard Powersort for our four datasets. Figure 4a shows the relative elapsed time on randomly shuffled data. Powersort with offset-value coding sorts TPC(SF=1), TPC(SF=10), URLs, and LCC over 52%, 57%, 47% and 50% faster than standard Powersort, respectively. `std::sort`

[7] https://commoncrawl.org



(a) Elapsed time for sorting different, completely shuffled datasets.

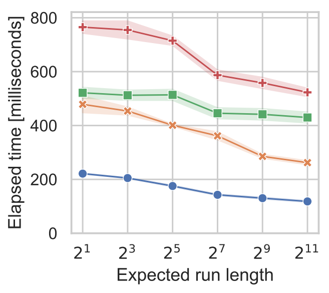


(b) Elapsed time for sorting different datasets, presorted to approx. $\sqrt{N}$ runs of expected length $\sqrt{N}$.

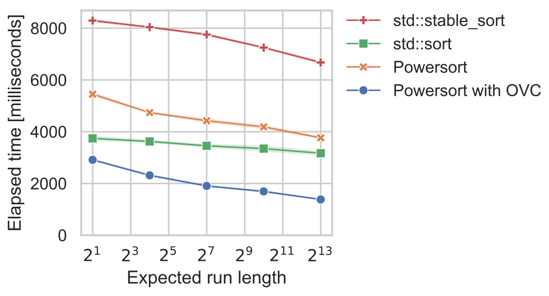
Fig. 4: Running time of our algorithms and C++ library sorts, relative to standard Powersort. Note that std::sort is not a stable sort.

performs well on these random data sets being superior to standard Powersort in all datasets but TPC(SF=10). `std::stable_sort` is consistently slower than Powersort by 40-80%. Figure 4b displays the results for "halfway sorted" data. The datasets is now arranged in (expected) $\sqrt{N}$ sorted runs with $\sqrt{N}$ expected keys each, making $N\mathcal{H} \approx \frac{1}{2}\log_2(N!)$. The speedup for using offset-value codes in Powersort increases slightly for TPC(SF=1), TPC(SF=10) and remains stable for URLs. For LCC, the improvement in elapsed time increases to 60% compared to the experiment with random input. `std::sort` performs slower on the two TPC datasets but is still superior to standard Powersort on URLs and LCC; `std::stable_sort` remains consistently slowest.

Figure 5a shows the elapsed time as a function of presortedness measured by the expected length of existing runs for four methods sorting TPC(SF=1); Figure 5b shows the same for LCC. In all remaining plots, the *x*-axis shows presortedness measured by the expected length of runs; larger values mean more presorted inputs.



(a) Elapsed time of sorting TPC(SF=1) with initially sorted runs of varying length.

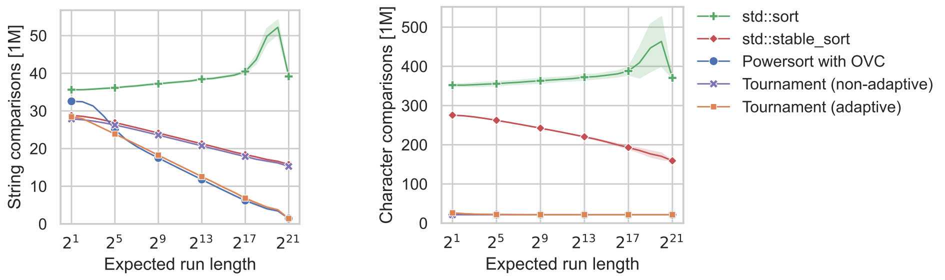


(b) Elapsed time of sorting LCC with initially sorted runs of varying length.

Fig. 5: The elapsed times of sorting TPC(SF=1) and LCC with varying degrees of presortedness.

To demonstrate that offset-value codes and adaptive sorting work well together also outside Powersort, we compared two further algorithms that can make use of offset-value coding: *Tournament sort (non-adaptive)* uses a tournament tree with a sufficiently high fan-in to merge the entire input with only one merge operation. The non-adaptive version initially creates runs of length 2 by comparing the input in pairs, setting offset-value codes, and swapping them, if necessary. Because it avoids checking run boundaries, all character comparisons are cached in offset-value codes. *Tournament sort (adaptive)* also performs a single merge. In contrast to the non-adaptive version, it scans for ascending and descending runs just like Powersort, but without applying insertion sort to extend runs. We also include the non-adaptive C++ library sort algorithms `std::sort` and `std::stable_sort`.

Figure 6a shows the number of string comparisons performed by the different algorithms. `std::sort` does not benefit from sorted inputs; the number of comparisons shows a sharp spike in the almost-sorted case (likely an artifact of having exactly 2 runs). The non-adaptive `std::stable_sort` and tournament sort require approximately the same number of string comparisons, decreasing from 34M to 16M. Powersort and adaptive tournament sort require approximately the same number of comparisons – as long as sorted runs in the input exceed the threshold for insertion sort that Tournament sort (adaptive) does not employ. Figure 6b shows the number of character comparisons. The sort routines from the C++ library (unsurprisingly) require orders of magnitude more comparisons than the three algorithms with offset-value codes; all of them are close to the minimum at 21.5M comparisons. Though barely visible in the plots, Tournament sort (adaptive) requires an additional 4M character comparisons during run-detection in the unsorted case; Powersort reduces this number to 420K by extending the runs with insertion sort.



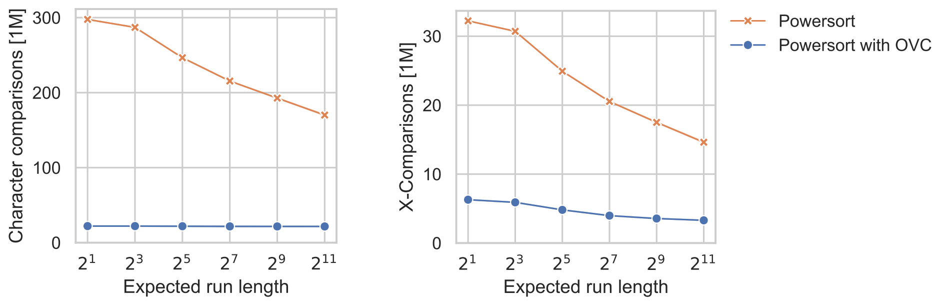
(a) Number of string comparisons for sorting TPC(SF=1).

(b) Number of character comparisons for sorting TPC(SF=1).

Fig. 6: The number of comparisons of Powersort and multiway merging using tournament trees.

We tested Hypothesis 2 by sorting TPC(SF=1) with varying presortedness and counting the number of character comparisons and the number of “X-comparisons” (expensive comparisons), i.e., string comparisons that were *not* decided by offset-value codes. Figure 7a shows that offset-value coding vastly reduces the total number of character comparisons. The effects of boundary detection on the number of comparisons are negligible as runs are extended to length 24 with insertion sort, resulting in $N/24 \approx 60K$ runs and about $0.42M$

non-cached comparisons. Figure 7b shows the number of X-comparisons. For standard Powersort, every string comparison is expensive so the curve simply displays the number of string comparisons.

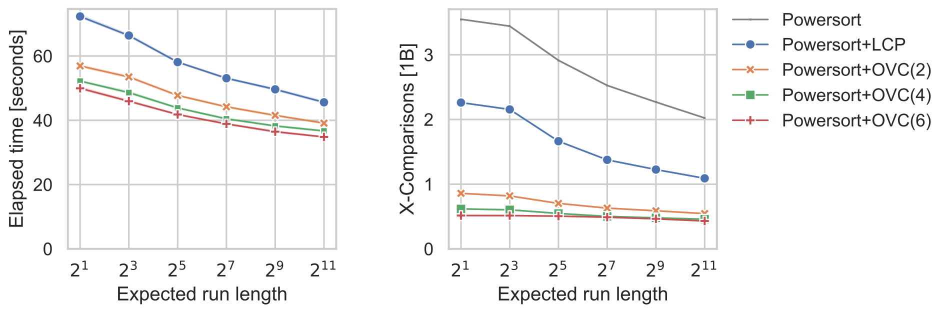


(a) Number of character comparisons.

(b) Number of X-comparisons (string comparisons not decided by offset-value codes).

Fig. 7: The effect of offset-value codes on the number of comparisons for sorting TPC(SF=1).

For testing Hypothesis 4, we sorted URLs with Powersort and different offset-value codes where OVC(i) denotes that i bytes (of 8) are used for the value. Note that longest-common-prefix (LCP) values correspond to OVC(0). Figure 8a and Figure 8b show the elapsed time and the number of comparisons not decided by offset-value codes, respectively. Figure 8a shows a decrease in elapsed time as more bytes are encoded in the offset-value code. Note that the decrease of LCP (corresponding to OVC(0)) to OVC(2) is the largest. The effects diminish as more bytes are encoded. These differences in elapsed time can be explained by considering the number of comparisons decided by the different offset-value codes. 8b shows that LCPs only avoid about a third of expensive string comparisons in the unsorted case and about half in the more sorted cases, whereas offset-value codes approximately *double* these savings to two-thirds in the unsorted case, and to three quarters in the more sorted cases.



(a) Elapsed time of sorting URLs using OVC(i), $i = 0, 2, 4, 6$.

(b) Number of string comparisons not decided by OVCs sorting URLs.

Fig. 8: Impact of value size in OVCs.

5.5 Summary of the experimental evaluation

The experiments of Section 5.4 confirm our hypotheses and claims of Section 5.1. We have demonstrated how adaptive sorting methods in combination with offset-value codes reduce the number of character comparisons and expensive string comparisons. We have shown that encoding more bytes in offset-value codes improves run times even further, in particular in comparison with the more widely known longest-common-prefix values.

6 Summary and conclusions

In summary, our work introduces

1. efficient adaptive sorting for large keys;
2. $\log_2(N!)$ string or row comparisons in the expected and worst cases –very robust performance guarantees with tournament trees carefully implemented and used;
3. $N - 1$ string or row comparisons in the best case – very effective discovery of fully sorted inputs;
4. graceful degradation from best case to expected case;
5. even with discovery of incidental pre-existing sorted runs, whether or not particularly successful, effort for character or column value comparisons remains linear with the total size of all keys; and
6. full benefit of traditional *adaptive sorting* for reducing the count of *row comparisons* plus full benefit of *offset-value coding* for reducing the count of *column value comparisons* and thus the cost of row comparisons.

In conclusion, the paper reveals an unexpectedly effective synthesis of techniques for designing adaptive sorting methods with a high impact on sorting and sort-based algorithms in database systems. While various theoretical and practical results convincingly demonstrated that adaptive sorting methods can optimally exploit whatever extent of presorted runs exists in the input for small objects of fixed size, our novel synthesized approach makes adaptive algorithms applicable to sorting large objects, e.g., an arbitrary sequence of key columns represented as strings of characters or bytes. In particular, this paper shows that carefully engineered variants of adaptive mergesort (Powersort) combined with offset-value codes are highly beneficial and outperform state-of-the-art (adaptive) sorting methods for various datasets regarding the number of character comparisons, string comparisons, and execution time. The performance advantages of the (new) adaptive methods combine with the benefits of offset-value coding for large objects, at a practically insignificant expense of 4.2% additional character comparisons in the worst case.

Acknowledgments: This work was partially supported by DFG grant SE 553/9-1 and EPSRC grant EP/X039447/1.

Bibliography

[Ag96] Agarwal, Sameet; Agrawal, Rakesh; Deshpande, Prasad; Gupta, Ashish; Naughton, Jeffrey F.; Ramakrishnan, Raghu; Sarawagi, Sunita: On the Computation of Multidimensional Aggregates. In: VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India. Morgan Kaufmann, pp. 506–521, 1996.

[Au18] Auger, Nicolas; Jugé, Vincent; Nicaud, Cyril; Pivoteau, Carine: On the Worst-Case Complexity of TimSort. In (Yossi Azar, Hannah Bast; Herman, Grzegorz, eds): 26th Annual European Symposium on Algorithms (ESA 2018). Leibniz International Proceedings in Informatics (LIPIcs), 2018.

[BK19] Buss, Sam; Knop, Alexander: Strategies for Stable Merge Sorting. In: Symposium on Discrete Algorithms (SODA 2019), pp. 1272–1290. SIAM, jan 2019.

[BN13] Barbay, Jérémy; Navarro, Gonzalo: On compressing permutations and adaptive sorting. Theor. Comput. Sci., 513:109–123, 2013.

[Bo63] Boothroyd, J.: Algorithm 207: Stringsort. Comm. of the ACM, 6(10):615, 1963.

[BSS20] Bingmann, Timo; Sanders, Peter; Schimek, Matthias: Communication-Efficient String Sorting. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020. IEEE, pp. 137–147, 2020.

[Bu58] Burge, William H.: Sorting, Trees, and Measures of Order. Inf. Control., 1(3):181–197, 1958.

[Co77] Conner, W. M.: Offset-value coding. IBM Technical Disclosure Bulletin, p. 2832–2837, 1977.

[CS77] Comer, Douglas; Sethi, Ravi: The Complexity of Trie Index Construction. J. ACM, 24(3):428–440, 1977.

[DG23] Do, Thanh; Graefe, Goetz: Robust and Efficient Sorting with Offset-value Coding. ACM Trans. Database Syst., 48(1):2:1–2:23, 2023.

[ECW92] Estivill-Castro, Vladmir; Wood, Derick: A survey of adaptive sorting algorithms. ACM Computing Surveys, 24(4):441–476, December 1992.

[Fr56] Friend, Edward H.: Sorting on Electronic Computer Systems. J. ACM, 3(3):134–168, 1956.

[GD23] Graefe, Goetz; Do, Thanh: 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, pp. 464–470, 2023.

[Ge23] Gelling, William Cawley; Nebel, Markus E.; Smith, Benjamin; Wild, Sebastian: Multiway Powersort. In: Symposium on Algorithm Engineering and Experiments (ALENEX). pp. 190–200, 2023.

[GEQ12] Goldhahn, Dirk; Eckart, Thomas; Quasthoff, Uwe: Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages. In (Calzolari, Nicoletta; Choukri, Khalid; Declerck, Thierry; Dogan, Mehmet Ugur; Maegaard, Bente; Mariani, Joseph; Odijk, Jan; Piperidis, Stelios, eds): Proceedings of the Eighth International Conference on Language Resources and Evaluation, LREC 2012, Istanbul, Turkey, May 23-25, 2012. European Language Resources Association (ELRA), pp. 759–765, 2012.

[GJK22] Ghasemi, Elahe; Jugé, Vincent; Khalighinejad, Ghazal: Galloping in Fast-Growth Natural Merge Sorts. In (Bojańczyk, Mikołaj; Merelli, Emanuela; Woodruff, David P., eds): 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022). volume 229 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 68:1–68:19, 2022.

[GKS25] Graefe, Goetz; Kuhrt, Marius; Seeger, Bernhard: Modifying an existing sort order with offset-value codes. EDBT conf, Barcelona, 2025.

[Go63] Goetz, Martin A.: Internal and tape sorting using the replacement-selection technique. Commun. ACM, 6(5):201–206, 1963.

[Gr97] Gray, Jim; Chaudhuri, Surajit; Bosworth, Adam; Layman, Andrew; Reichart, Don; Venkatrao, Murali; Pellow, Frank; Pirahesh, Hamid: Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. Data Min. Knowl. Discov., 1(1):29–53, 1997.

[Gr06] Graefe, Goetz: Implementing sorting in database systems. ACM Comput. Surv., 38(3):10, 2006.

[Gr23] Graefe, Goetz: Priority queues for database query processing. In: Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme"(DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings. volume P-331 of LNI. Gesellschaft für Informatik e.V., pp. 27–46, 2023.

[GvN48] Goldstine, Herman H.; von Neumann, John: 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, 15 April 1948.

[Hä77] Härder, Theo: A Scan-Driven Sort Facility for a Relational Database System. In: VLDB. pp. 236–244, 1977.

[he] (https://math.stackexchange.com/users/177399/mike earnest), Mike Earnest: Probability of partially sortedness of a perfectly shuffled array of unique numbers. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/4596036 (version: 2022-12-10).

[Ho89] Hollerith, Herman: Art of compiling statistics. US Patent, 395,781, Jan 8, 1889.

[Ho61] Hoare, C. A. R.: Algorithm 64: Quicksort. Commun. ACM, 4(7):321, 1961.

[Hu63] Hubbard, George U.: Some characteristics of sorting computing systems using random access storage devices. Commun. ACM, 6(5):248–255, 1963.

[IB88] IBM: Enterprise System Architecture/370, Principles of Operation. IBM publication SA22-7200-0, 1988.

[Iy05] Iyer, Balakrishna R.: Hardware Assisted Sorting in IBM's DB2 DBMS. In: Advances in Data Management 2005, Proceedings of the 12th International Conference on Management of Data, COMAD 2005b, December 20-22, 2005, Hyderabad, India. Computer Society of India, 2005.

[Ja97] Jagadish, H. V.; Narayan, P. P. S.; Seshadri, S.; Sudarshan, S.; Kanneganti, Rama: 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, pp. 16–25, 1997.

[Ju20] Jugé, Vincent: Adaptive Shivers Sort: An Alternative Sorting Algorithm. In: Symposium on Discrete Algorithms (SODA 2020), pp. 1639–1654. SIAM, jan 2020.

[Kn70] Knuth, Donald E.: Von Neumann's First Computer Program. ACM Comput. Surv., 2(4):247–260, 1970.

[Kn73] Knuth, Donald E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley, 1973.

[KRM21] Kuiper, Laurens; Raasveldt, Mark; Mühleisen, Hannes: Efficient External Sorting in DuckDB. In: British International Conference on Databases 2021, London, United Kingdom, March 28, 2022. volume 3163 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 40–45, 2021.

[Me77] Mehlhorn, Kurt: A Best Possible Bound for The Weighted Path Length of Binary Search Trees. SIAM Journal on Computing, 6(2):235–239, June 1977.

[MW18] Munro, J. Ian; Wild, Sebastian: Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs. In (Azar, Yossi; Bast, Hannah; Herman, Grzegorz, eds): 26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland. volume 112 of LIPIcs, pp. 63:1–63:16, 2018.

[NF20] Neumann, Thomas; Freitag, Michael J.: Umbra: A Disk-Based System with In-Memory Performance. In: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. www.cidrdb.org, 2020.

[NK08] Ng, Waihong; Kakehi, Katsuhiko: Merging string sequences by longest common prefix. In: IPSJ Digital Courier. volume 4, pp. 69–78, 2008.

[Ny95] Nyberg, Chris; Barclay, Tom; Cvetanovic, Zarka; Gray, Jim; Lomet, David B.: AlphaSort: A Cache-Sensitive Parallel External Sort. VLDB J., 4(4):603–627, 1995.

[O'96] O'Neil, Patrick E.; Cheng, Edward; Gawlick, Dieter; O'Neil, Elizabeth J.: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica, 33(4):351–385, 1996.

[Pö02] Pöss, Meikel; Smith, Bryan; Kollár, Lubor; Larson, Per-Åke: TPC-DS, taking decision support benchmarking to the next level. In (Franklin, Michael J.; Moon, Bongki; Ailamaki, Anastassia, eds): Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002. ACM, pp. 582–587, 2002.

[Se79] Selinger, Patricia G.; Astrahan, Morton M.; Chamberlin, Donald D.; Lorie, Raymond A.; Price, Thomas G.: Access Path Selection in a Relational Database Management System. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1. ACM, pp. 23–34, 1979.

[Se10] Seidel, Raimund: Data-Specific Analysis of String Sorting. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010. SIAM, pp. 1278–1286, 2010.