Priority queues for database query processing

New techniques for tree-of-losers priority queues and for offset-value coding

Goetz Graefe[1]

Abstract: Interesting orderings let sort-based query processing out-perform hash-based algorithms, but only tree-of-losers priority queues and offset-value coding permit competing in all cases including large unsorted inputs with large or complex keys. As long as this competition persists, alternative algorithms with equivalent functionality will plague query execution, e.g., in software maintenance and in query plan scheduling, and mistaken algorithm choices will plague query optimization, e.g., for joins, intersection, and grouping.

After explaining tree-of-losers priority queues and offset-value coding, our work introduces necessary extensions for efficient run generation (in external merge sort) with variable-size records. The required changes in tree-of-losers priority queues support increasing and decreasing any key value at any time in logarithmic time, including incremental maintenance of offset-value codes, with the expected time for key value increases independent of the size of the priority queue. As all kinds of scheduling applications use priority queues, our contributions go beyond database query processing. A discussion of double-ended priority queues illustrates the concepts.

This may be the first time that tree-of-losers priority queues are extended to addressable priority queues and to non-monotone sequences of input keys; and that offset-value coding is extended to non-monotone sequences of input keys. The proposed solutions and the included code snippets are simple, small, and fast, in contrast to the time and effort spent on bringing them to this state.

Keywords: sorting; grouping; merge join; interesting orderings; tree of losers; offset-value coding.

1 Introduction

In many classic algorithms for database query execution [BE77, Ep79], from in-stream duplicate removal, grouping, and aggregation (for sorted streams) to in-sort grouping, merge join, and index nested-loops join, each algorithm's core is either a database index or a sort operation. Hash-based query execution algorithms, e.g., [Br84, DG85, De84, KTM83, NKT88], seem to have changed that, but other than computing hash values from column values, hash join and hash aggregation have at their core an index, i.e., a hash table, and a sort, i.e., internal and external distribution sort [IS56] on hash values rather than column values. Thus, sorting techniques are crucial for efficient database query processing, e.g., distribution sort, quicksort [Ho62], merge sort [Fr56], and priority queues.

Tree-of-losers priority queues [Go63, Kn98] are more efficient than the traditional and better known tree-of-winners priority queues because the former use only leaf-to-root

[1] Google; Madison, Wisconsin, USA GoetzG@Google.com

doi:10.18420/BTW2023-01

passes with $log_2 N$ comparisons, whereas the latter also need root-to-leaf passes with up to $2 \times log_2 N$ comparisons. Therefore, tree-of-losers priority queues can guarantee internal and external sorting with counts of comparisons practically equal to the provable lower bound[2]. Just as importantly, tree-of-losers priority queues work with offset-value coding [Co77], which minimizes the effort per row comparison by combining prefix truncation and order-preserving surrogate keys. Together, these techniques guarantee at most $N \times K$ column value comparisons when sorting N rows with K key columns; the remainder of the required $log_2(N!)$ row comparisons are compiled-in single-instruction integer comparisons and thus similar to hash values in hash-based query processing.

Tree-of-losers priority queues seem to require monotone (ever-increasing) sequences of keys, making tree-of-losers priority queues perfect for merging sorted runs. They can be adapted [Go63] to internal sorting, run generation in read-sort-write cycles, and run generation with continuous replacement selection for fixed-size data records. In continuous replacement selection for variable-size data records, e.g., using best-fit or first-fit memory management [LG98], occupancy counts in a priority queue change frequently, which published methods for tree-of-losers priority queues cannot accommodate.

Tree-of-losers priority queues as used to-date, i.e., with complete leaf-to-root passes, can serve neither as non-monotone priority queues nor as addressable priority queues. On the other hand, only tree-of-losers priority queues can sort with comparison counts near the provable lower bound – neither quicksort nor tree-of-winners priority queues do. Moreover, only tree-of-losers priority queues can use offset-value coding for linear costs for column value comparisons – neither quicksort nor tree-of-winners priority queues do.

Initially motivated by variable-size records in replacement selection, our contributions are:

1. a small extension to leaf-to-root passes in tree-of-losers priority queues that elevates them to addressable priority queues;
2. a significant extension to leaf-to-root passes in tree-of-losers priority queues that supports non-monotone sequences of input keys, including early and late fences for invalid (not occupied) slots;
3. new techniques for offset-value coding in tree-of-losers priority queues with non-monotone sequences of input keys;
4. an implementation of double-ended priority queues that uses two priority queues in opposite sort order and that, after popping the top element from one priority queue, repairs the other one in constant expected time (independent of the size, capacity, or tree height of the priority queue); and

[2] Sorting is equivalent to finding a permutation: N distinct key values permit $N!$ permutations; at best, each key comparison disproves half of the remaining possibilities; determining the permutation of the input requires at least $log_2(N!) \approx N \times log_2(N/e)$ comparisons with Euler's number $e \approx 19/7$.

5. an implementation of run generation using offset-value coding and continuous replacement selection for variable-size records.[3]

The next section provides technical background and reviews related prior work. Section 3 introduces a small extension in leaf-to-root passes of tree-of-losers priority queues that elevates them to addressable priority queues, whereupon Section 4 introduces a significant extension to support non-monotone sequences of input keys. Section 5 employs these extensions for a double-ended priority queue using two priority queues, one ascending and one descending. Section 6 adds a third extension for efficient incremental maintenance of offset-value codes in spite of non-monotone sequences of input keys. Section 7 uses all three extensions for run generation with variable record counts in memory. The final section sums up and concludes with thoughts on adopting offset-value coding in a broader context, specifically in all sort-based algorithms for database query evaluation.

2 Background and related prior work

This section provides some technical background about priority queues, offset-value coding, and external merge sort.

2.1 Priority queues

A priority queue manages a set of pairs, each a priority and some associated information. The priority is the sort key within the pair. The associated information can be detailed information, a pointer, an array index, or something else. The purpose of priority queues is to provide the minimum (or maximum) key value very quickly (from the root of a tree) and to absorb additional key-value pairs efficiently. For simplicity, all data structures considered here for priority queues assume a balanced binary tree of height n and capacity 2^n or $2^n - 1$.

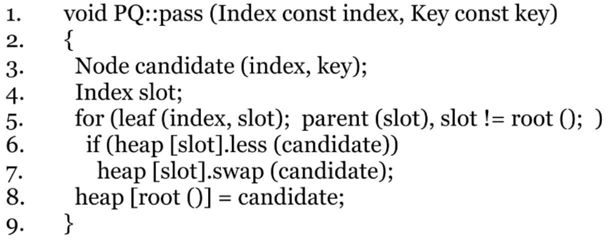
In a tree-of-winners priority queue [Kn98], the principal invariant is that a parent's key is lower than the key values in its two children (assuming an ascending ordering of key values). Popping (removing) the key-value pair with the minimum key value moves the right-most leaf entry to the root and then pushes it into the tree (with $2n$ comparisons in a worst-case root-to-leaf pass). A subsequent insertion requires only a leaf-to-root pass with n comparisons. Alternatively, if the removal leaves an invalid entry in the root (with logical

[3] Prior work [LG98] used a very early version of this technique, without explicit mention, without detail or explanation, and without offset-value coding. Early versions built a stack of move targets, which nominally cut moves to one third but on superscalar CPUs performs no better than swapping (see line 7 in Figure 1), separated partial and complete leaf-to-root traversals and their different looping conditions (see line 15 in Figure 3), gated the repair loop with an extra key comparison (see line 31 in Figure 4), supported incomplete binary trees (capacities other than 2^n), and organized the tree as b-tree of cache lines, with no benefit if the entire tournament tree easily fits into the L1 cache as recommended for sorting.

key value $-\infty$), pushing (inserting) a new key value simply replaces the invalid entry but then repairs the tree invariants, which usually requires two comparisons per tree level and may require all n tree levels, for $2n$ comparisons in a worst-case root-to-leaf pass.

2.1.1 Tree-of-losers priority queues

A tree-of-losers priority queue [Go63, Kn98], also known as a tournament tree, is a balanced binary tree. When mapped to an array, the tree's unary root is in array slot 0. It is efficient due to leaf-to-root passes with one comparison per tree level; root-to-leaf passes with two comparisons per tree level are not required. A pair of "pop" and "push" operations requires only a single leaf-to-root pass. The principal rules are that (i) two candidate keys compete at each node in the tree and (ii) after a comparison of two candidates, the loser remains in the node and the winner becomes a candidate in the next tree level. Thus, a new overall winner reaches the root node after n comparisons in a priority queue with 2^n entries. When merging, a fixed pair of runs competes at each leaf node. Run generation using read-sort-write cycles merges "sorted runs" of a single row each. Run generation by continuous replacement selection tries to extract longer sorted runs from the unsorted input.



```
1.   void PQ::pass (Index const index, Key const key)
2.   {
3.     Node candidate (index, key);
4.     Index slot;
5.     for (leaf (index, slot);  parent (slot), slot != root ();  )
6.       if (heap [slot].less (candidate))
7.         heap [slot].swap (candidate);
8.     heap [root ()] = candidate;
9.   }
```

Fig. 1: The traditional leaf-to-root pass

Figure 1 shows code extracted from a working prototype of a tree-of-losers priority queue. The "index" parameter is the information associated with the key value, e.g., a run identifier between 0 and $F - 1$ during a merge step with fan-in F. Line 3 creates a new tree node that will be swapped into the array "heap" that holds the priority queue. Line 4 defines an index for this array; line 5 initializes it, halves it to navigate from a child to its parent, and terminates the loop at the root. An equivalent to lines 4 and 5 is "for (Index slot = capacity + index; (slot /= 2) != 0;)", but the syntax in Figure 1 more readily enables the extensions required later. Another alternative is "for (Index slot = capacity/2 + index/2; slot != 0; slot /= 2)", which more directly shows skipping over the non-leaf nodes in the tree and assigning two index values to each tree leaf. Lines 6 and 7 in Figure 1 compare key values and ensure that the loser remains behind as the winner becomes a candidate at the parent. Line 8 saves the overall winner in the tree's root node.

With only leaf-to-root passes (and no root-to-leaf passes), run generation and merging with tree-of-losers priority queues guarantee near-optimal comparison counts. The count of comparisons in the best case, the worst case, and the expected case are all within rounding errors of the provable lower bound of $log_2(N!)$, i.e., better than the expected case of quicksort and far better than the worst case of quicksort. This includes the expected case for replacement selection, where one additional comparison per input row doubles the expected run size, cuts the run count in half, and saves one comparison per row in the merge process. With excellent expected and worst-case run-time complexity yet very limited implementation complexity, some hardware supports tree-of-losers priority queues. More specifically, the UPT "update tree" instruction of IBM's 370- and z-series mainframes [IB88, Iy05] implements essentially the logic of Figure 1.

In a tree-of-losers priority queue, each key value is paired with an index in the range 0 to $2^n - 1$ for a priority queue with tree height n. During a merge step, these indexes identify runs; during run generation, they identify rows in memory or more precisely slots in an array. These indexes determine where a leaf-to-root pass starts. At all times, each index value exists exactly once in the priority queue, possibly marked as an invalid entry by a fence with logical key value $-\infty$ or $+\infty$.

2.1.2 Addressable priority queues

In a tree-of-losers priority queue, an index value maps to a specific leaf and therefore to a specific leaf-to-root path, which can be used to identify, find, read, and perhaps modify or even delete any entry. For the same functionality, a tree-of-winners priority queue requires an additional data structure to find an entry in the tree. While this data structure permits finding an entry quickly, it must track every movement in the tree representing the priority queue, thus increasing the cost of any movement within the priority queue.

If a key value is modified or deleted, the priority queue and its invariants need repair. For tree-of-losers priority queues, Section 3 introduces efficient techniques without any additional data structure yet with efficient maintenance for key change and deletion.

2.1.3 Monotone priority queues

A tree-of-winners priority queue tolerates insertion of any low or high key value at any time. In contrast, a traditional tree-of-losers priority queue depends on ever-increasing key values. Section 4 introduces an efficient leaf-to-root pass that overcomes this limitation.

2.2 Offset-value coding

Offset-value coding [Co77] encodes one row's key value relative to another key that is earlier in the sort sequence. Offset-value codes are a by-product of comparisons, specifically in the loser of a comparison. The offset within a loser's new offset-value code is the position where the keys first differ, e.g., a column index, and the value is the loser's data value at that offset. Alternatively, the offset is the size of the shared prefix. For example, a duplicate key shares the entire key and thus has an offset equal to the key size.

	Column index				Descending offset-value codes			Ascending offset-value codes		
	0	1	2	3	offset	domain – value	OVC	arity – offset	value	OVC
Rows and their column values	5	4	7	3	0	95	95	4	5	405
	5	4	7	6	3	94	394	1	6	106
	5	4	8	5	2	92	292	2	8	208
	5	4	9	1	2	91	291	2	9	209
	5	4	9	1	4	100	500	0	-	0
	5	5	2	3	1	95	195	3	5	305
	5	5	6	7	2	94	294	2	6	206

Fig. 2: Offset-value codes in a sorted file or stream

Figure 2 illustrates descending and ascending offset-value codes in a stream of rows in ascending sort order on all columns. With four sort columns, the arity of the sort key is 4; the domain of each column is 1 to 99. For an ascending sort order, descending offset-value codes take the actual offset but the negative of the column value, whereas ascending offset-value codes take the negative offset but the actual column value. The first row has offset 0 by definition. Figure 2 ignores that small key domains permit encoding multiple key columns together. IBM's CFC "compare and form codeword" instruction [IB88, Iy05] supports offset-value coding for a descending sort order of normalized keys (order-preserving byte strings), blocks of bytes as values, and block counts as offsets.

With offsets and values combined as shown in Figure 2, a single integer instruction may decide a comparison. If two rows and their key values *A* and *B* are encoded relative to the same key *C* that is earlier in the sort sequence, and if the offsets of *A* and *B* differ, then the one with the higher offset is earlier in the sort sequence. Otherwise, if the two data values at the common offset differ, then these data values decide the comparison. Otherwise, additional data values in *A* and *B* must be compared.

In a tree-of-losers priority queue with offset-value coding, each tree node lost to the local winner and its local offset-value code is set relative to the local winner. Conversely, the local key value in a tree node other than a leaf was a winner in all nodes up from the leaf where it entered the tree, and all key values along that path are encoded relative to this local key value. The overall winner in the tree's root is no exception: along its entire leaf-to-root path, all key values lost to (and are encoded relative to) the overall winner.

Merging sorted runs repeatedly replaces the overall winner with its successor from the same merge input. With merge inputs' fixed assignment to leaf nodes in a tree-of-losers priority queue, a successor retraces the leaf-to-root path of the prior overall winner. As this successor and all keys on its leaf-to-root path are encoded relative to the prior overall winner, offset-value coding applies to all comparisons in a tree-of-losers priority queue.

Offset-value codes decide many comparisons in a tree-of-losers priority queue. Column value comparisons are required only if two rows have equal offset-value codes. They start after the offset and value encoded in these offset-value codes. After such a row comparison is decided, the loser's offset is incremented by the count of column value comparisons. With K sort columns, the sum of all offset increments is limited to K in each row; in an input with N rows, the sum of all increments and thus the count of all column value comparisons are limited to $N \times K$. Importantly, there is no $log(N)$ multiplier here. Thus, tree-of-losers priority queues and offset-value coding guarantee that the effort for column value comparisons is linear in the count of rows and in the count of sort columns, quite like the effort for computing hash values in hash-based query execution.

This limit on column value comparisons resonates with data-specific analysis of string sorting [Se10]: the total count of "=" comparisons of symbols (within strings) or of column values (within database rows) equals the opportunity for compression in a sorted dataset. Compression here may be prefix truncation in row storage (e.g., using offset-value coding as shown in Figure 2), run-length encoding of leading columns in column storage, or shared prefixes in a trie [Se10]. Conversions between these alternative formats do not require additional comparisons of symbols or column values [DG22].

Comparisons of offset-value codes are free if they are subsumed in other algorithm activities. In quicksort, for example, the inner-most loop not only compares key values but also looping indexes: when the loops from the left and the right meet, the partitioning step is complete. In priority queues, the inner-most loop compares key values only after testing whether there even are valid key values. This is needed because during queue construction, some entries have not yet been filled; after the end of some merge inputs, some queue entries no longer have valid keys; and during run generation by merging single-row runs, there is only queue build-up and tear-down.

During run generation by continuous replacement selection, the run identifier can prefix the user-defined key as an artificial leading key column. If so, early and late fence values may be modeled as initial and final runs with multiple early and late fence key values, e.g., one for each merge input [Gr06]. All of this can be folded into each row's offset-value code: if two rows have equal offset-value codes, they are both valid (neither early nor late fences), they go to the same output run, they differ from their shared base row (an earlier winner) at the same offset, they have the same value at that offset, and the next step must compare further columns. Thus, comparisons of offset-value codes are not overhead as they simply take the place of testing for fence keys during continuous replacement selection.

The design also reduces CPU cache faults. If a tree-of-losers priority queue requires 8 bytes per entry, an L1 cache can retain a priority queue (an array) of 512 or 1,024 entries. The data records may or may not fit in a lower-level cache but offset-value codes decide many comparisons without cache fault, many more than traditional pairs of key prefix and data pointer [Hu63, Ny95]. Mini-runs of this size remain in DRAM until merged (with fan-in 512 or 1,024) to form initial runs on temporary external storage [BL89, Fr56, Ny95].

2.3 External merge sort

Priority queues enable efficient external merge sort in multiple ways. The most obvious one is merging sorted runs. In fact, merging runs is a perfect application for traditional tree-of-losers priority queues with only complete leaf-to-root passes. A related application is forecasting [Fr56], i.e., predicting which merge input benefits most from an additional input buffer for asynchronous read-ahead. Forecasting tracks, for each merge input, the highest key value read so far and selects the lowest of these values. A third application of priority queues in external merge sort chooses which runs to merge next, e.g., the smallest existing runs [Hä77b] or the set with the most similar sizes. The former heuristic is widely used; the latter heuristic applies when the final input size is not yet known, i.e., when merging while still consuming unsorted input rows.

Another well-known application of tree-of-losers priority queues in external merge sort is run generation, whether in read-sort-write cycles or in continuous replacement selection [Go63]. In read-sort-write cycles, priority queues suffer from repeated build-up and tear-down such that quicksort is often preferred. In cache-optimized read-sort-write cycles, however, merging cache-sized runs in memory to form the initial run on external temporary storage [BL89, Fr56, Ny95] uses a priority queue; moreover, creating many cache-sized runs permits using a tree-of-losers priority queues very efficiently in a way similar to replacement selection. In traditional replacement selection using a single priority queue for all unsorted rows in memory, fixed-size rows enable runs twice the size of memory, but variable-size rows require the techniques introduced in Sections 4 and 6.

Finally, priority queues are useful in many scheduling decisions around external merge sort, e.g., which query to run next, where to grant more memory, where to pinch memory, etc.

3 Addressable priority queues

In a scheduling application or a simulation, if a future event is cancelled, an entry in the priority queue must be found and deleted, which requires an addressable priority queue. In a tree-of-losers priority queue, a deletion replaces a valid key value with a late fence.

In a tree-of-losers priority queue, the index is the handle by which to find and identify an entry. As each index maps to a specific leaf node, a search along one leaf-to-root path suffices. The search ends at the sought index value, with a 50% probability that the sought entry was a loser left behind in the leaf, a 25% probability for the parent node, etc. On average, just less than two nodes are inspected along a leaf-to-root path, for a constant expected time independent of the size or capacity of the priority queue.

If the new key value associated with an index is higher than the one in the priority queue, which is always true if the new key value is a late fence, the leaf-to-root pass can repair the tree-of-losers priority queue and its invariants. In fact, the required logic treats the sub-tree rooted at the sought index as the entire priority queue. The principal code change merely adds a termination condition to the core logic of tree-of-losers priority queues.

```
10.  void PQ::pass (Index const index, Key const key)
11.  {
12.    Node candidate (index, key);
13.    Index slot;
14.    for (leaf (index, slot);  parent (slot),
15.        slot != root ()  &&  heap [slot].index != index;  )
16.      if (heap [slot].less (candidate))
17.        heap [slot].swap (candidate);
18.    heap [slot] = candidate;
19.  }
```

Fig. 3: The modified leaf-to-root pass

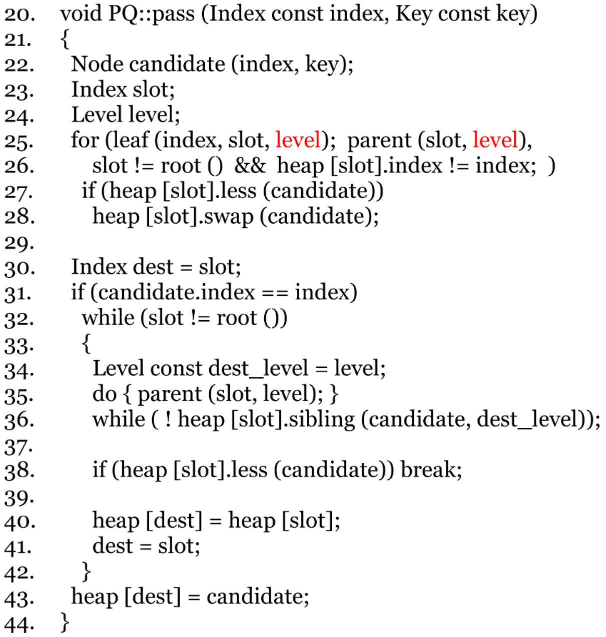
Figure 3 highlights the code changes relative to Figure 1, including saving the final candidate in the position of the replaced entry (line 18), not necessarily in the tree's root (line 8 in Figure 1). Note that swaps may occur along the path from leaf to replaced value (lines 16 and 17), that the new loop continuation condition (line 15) could replace rather than augment the original condition, and that line 18 could replace line 8 in Figure 1.

If the new key value associated with an index is lower than the one in the priority queue, the final position of the new key value is not between leaf and replaced entry but between replaced entry and root. Section 4 introduces the required new logic.

4 Non-monotone priority queues

If, in an ascending sort, a new key value is higher than the key value it replaces, the input key value cannot go further on its leaf-to-root pass than the position of the replaced key value. If, however, a new key value is lower than the key value that it replaces, including a valid key value replacing a late fence, its final position is on the path between the position of the replaced key value and the root. This requires an initial leaf-to-root pass that ends at the key value to be replaced. This first part equals the search discussed in Section 3, except that the lower key does not swap places with tree entries between the leaf and the replaced value. In fact, the absence of such swaps indicates that the new key value may be smaller than the key value that is about to be replaced in the priority queue.

The new key value may even be lower than some of the key values between the replaced value and the tree's root. If so, such a value might need to move backward on its leaf-to-root path in order to make room for the new, lower key value. Importantly, only one key value between replaced tree entry and root is a candidate for moving backward, namely the entry that formerly emerged as winner from the sub-tree rooted at the replaced key value. Thus, the first step is to locate this former winner between replaced value and root. The second step compares the new key value with this former winner; if the former winner wins again, then the new value simply overwrites the value to be replaced. Otherwise, the former winner moves backward, overwriting the replaced key value, and the steps repeat.



```
20. void PQ::pass (Index const index, Key const key)
21. {
22.   Node candidate (index, key);
23.   Index slot;
24.   Level level;
25.   for (leaf (index, slot, level);  parent (slot, level),
26.       slot != root ()  &&  heap [slot].index != index;  )
27.     if (heap [slot].less (candidate))
28.       heap [slot].swap (candidate);
29.
30.   Index dest = slot;
31.   if (candidate.index == index)
32.     while (slot != root ())
33.     {
34.       Level const dest_level = level;
35.       do { parent (slot, level); }
36.       while ( ! heap [slot].sibling (candidate, dest_level));
37.
38.       if (heap [slot].less (candidate)) break;
39.
40.       heap [dest] = heap [slot];
41.       dest = slot;
42.     }
43.   heap [dest] = candidate;
44. }
```

Fig. 4: The repair loop added

Figure 4 shows the repair loop (lines 30 to 43) that moves former winners backward on their leaf-to-root path. Instead of saving the final candidate directly (line 18 in Figure 3), the replaced key value becomes the initial destination for the candidate (line 30) and the candidate moves into the heap eventually (line 43). Line 31 ensures that the repair loop runs only if the new key value is small, i.e., after a search loop (lines 25 to 28) without a swap. The repair loop ends when it reaches the tree root (line 32) or when it finds a former winner with a lower key value (line 38). A nested loop searches for a former winner to move backward (lines 35 and 36). After a backward move (line 40), its source becomes the candidate's new destination (line 41). The tree level is tracked to test whether an ancestor was a former winner at the current destination and therefore could move backward to the current destination (lines 24, 25, and 34 to 36). Extended "leaf" and "parent" methods (lines 25 and 35) initialize the level to 0 and increment it by 1. Even with three loops in total (lines 25, 32, and 35), the complexity of the entire process is still strictly logarithmic like the height of the tree, with at most n invocations of the “less” comparison method (lines 27 and 38) in a tree-of-losers priority queue with capacity 2^n.

The conditions for the repair loop permit a few optional optimizations. First, the repair loop is not required if the old key value is an early fence or if the new key value is a late fence. Second, the four conditions for entering the loop could be reordered by probability and execution cost [Ha77a]. Third, the loop’s continuation test could move to the bottom.

```
45.    Index dest = slot;
46.    if (slot != root ()  &&  candidate.index == index  &&
47.       key != late_fence (index)  &&  heap [slot].key != early_fence (index))
48.     do
49.     {
50.       ...
51.     } while (slot != root ());
52.    heap [dest] = candidate;
```

Fig. 5: Alternative conditions for the repair loop

Figure 5 shows the relevant code fragments with these optional optimizations in lines 46 and 47. Lines 45 and 52 are lines 30 and 43 in Figure 4, respectively. Line 50 stands for lines 34 to 41 in Figure 4.

5 Double-ended priority queues

The example problem to be solved here is the following: a number of sellers frequently change their asking prices (for some goods or service); the lowest-price sellers often sell out and must withdraw their offer or raise their asking prices; and the highest-price sellers often withdraw their uncompetitive offers or drop their asking prices. Both buyers and sellers want the current lowest and highest asking prices readily available.

The solution employs an array of sellers with their current asking price if any. In addition, two priority queues, one ascending and one descending, track the lowest and highest asking prices. Together, they form a double-ended priority queue.

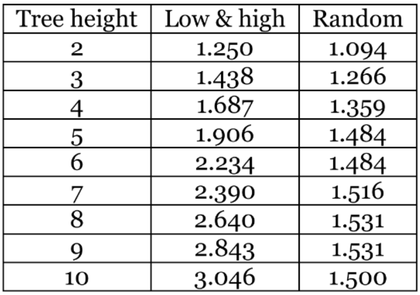
Both priority queues use the same indexes as the array, must be addressable to support deletion by index, and must be non-monotone to track all possible changes in asking prices, both increases and decreases.

Seeing current lowest and highest asking prices requires “top” methods. The lowest-price seller withdrawing requires a “pop” method in the ascending priority queue and a “delete” method in the descending priority queue. The highest-price seller withdrawing is just the opposite. Any other seller withdrawing requires a “delete” method in both priority queues. Any change in asking price requires an “update” method in both priority queues.

All of these methods invoke the “pass” method of Figure 4. The exception is “pop”: it might just replace a valid key value in the tree root with an early fence, but then subsequent “pop” or “top” invocations must invoke the “pass” method with the appropriate index and a late fence to propel a valid key value to the tree root. “Delete” also invokes “pass” with a late fence and “update” invokes “pass” with the new asking price.

Each invocation of the “pass” method requires time at most linear with the height and logarithmic with the capacity of the priority queue. The “delete” and “update” methods inspect just less than two tree nodes on average for constant expected time (independent of the size of the priority queue). The effort required in “delete” is likely less when invoked to match a “pop” in the other priority queue, but deleting the entry at a tree root forces a complete leaf-to-root pass in one of the priority queues. An optimization avoids or delays this full pass by placing an early fence in the root node.

The following experiments simulate up to 1,024 sellers (tree height 2-10) and $2^{25} \approx 33.5M$ changes in asking prices. They ran the code of Figures 4 and 5 on an Intel Celeron N3060 CPU (launched in 2016, 1.6 Ghz base frequency, 2.48 GHz burst frequency).



Tree height	Low & high	Random
2	1.250	1.094
3	1.438	1.266
4	1.687	1.359
5	1.906	1.484
6	2.234	1.484
7	2.390	1.516
8	2.640	1.531
9	2.843	1.531
10	3.046	1.500

Fig. 6: Maintenance effort for changing key value [CPU seconds]

Figure 6 shows the CPU times in seconds for two experiments. The first experiment (middle column) assumes that only lowest- and highest-price sellers change their asking prices. In other words, the double-ended priority queue runs “pop” and “delete” followed by an “insert” in both priority queues. For tree capacities of 4 and 1,024, i.e., tree heights of 2 and 10, the time difference is 1.796 seconds (3.046−1.250). Multiplying with the burst frequency (2.48 GHz) and dividing by the count of changes in asking prices (2^{25}) as well as the difference in tree heights (10−2) suggests that each tree level adds only about 17 CPU cycles to the effort of maintaining a double-ended priority queue constructed from two tree-of-losers priority queues.

The second experiment (right column) assumes that all sellers randomly change their asking prices. Recall that a randomly chosen seller (index in a priority queue) has remained in a tree leaf with a 50% probability, in a leaf’s parent with a 25% probability, etc., for an average search depth of just under two tree nodes. This is independent of the tree height, and indeed the elapsed times remain fairly steady. The differences observed are more likely a result of array sizes and CPU caches than of algorithm or code complexity.

6 Offset-value coding for non-monotone priority queues

Offset-value coding speeds up sorting with tree-of-losers priority queues. In fact, Section 2.2 shows how, in an external merge sort for N rows with K key columns, these techniques reduce worst-case counts of column value comparisons from $O(K \times NlogN)$ to $N \times K$. It has been unclear, however, whether scheduling and sorting applications with complex keys can benefit similarly from offset-value coding. Of course, no crisp benchmark and complexity metric exist for scheduling, but can offset-value coding reduce the comparison effort and can comparisons skip over column values already compared earlier?

The main difference to merging sorted runs is that predecessors and successors in a merge input are sorted and their offset-value codes can be known. In contrast, in scheduling applications, predecessor key values may not be known and successors may not have offset-value codes. In fact, as offset-value coding always is relative to a smaller key, a replacement value smaller than its predecessor cannot possibly have an offset-value code. A replacement key value without offset-value code relative to its predecessor (for the same index) requires full comparisons, i.e., starting at offset 0 within the key.

Fortunately, such a replacement key value still requires only a single leaf-to-root pass with a search loop (lines 25 to 28 in Figure 4) and a repair loop (lines 30 to 43 in Figure 4). Just as fortunately, full comparisons without the benefit of offset-value coding are required only until the search loop swaps a replacement key value into its correct location within the tree-of-losers structure. Thereafter, all comparisons benefit from offset-value coding. A replacement key value belongs into the leaf with 50% probability, into the leaf's parent with 25% probability, etc., for only about two full comparisons on average.

The first swap puts the new key value into its correct place within the tree-of-losers priority queue. After a swap, no repair loop is needed (see line 31 in Figure 4 and line 46 in Figure 5). A repair loop, if needed, may move some key values backward on their leaf-to-root paths (see Section 4 and Figure 4). In this case, offset-value codes along the leaf-to-root path must be adjusted such that losers are always encoded relative to the correct winner.

Figure 4 shows the traditional search loop (lines 25 to 28) and the new repair loop (lines 30 to 43). The traditional search can maintain offset-value codes in the traditional way: if column value comparisons are required, the loser's offset increases by their count. The comparison logic (line 27) can readily adjust the loser's offset-value code in this way.

Maintenance of offset-value codes in the repair loop seems expensive when a former winner moves backward on its leaf-to-root path, skipping backward over other tree nodes and key values (the ones skipped in lines 35 and 36 of Figure 4). When a new key value emerges as the new winner, existing skipped-over offset-value codes must be re-encoded relative to the new winner, incurring full row comparisons and thus column value comparisons.

Fortuitously, a recent theorem [GD22] on offset-value codes supplies a remedy: for three sorted key values $A < B < C$, the offset-value code of the third key value relative to the first one is the extreme of the other two offset-value codes, or $ovc(A,C) = max(ovc(A,B), ovc(B,C))$ for ascending offset-value codes[4]. When a former winner moves backward, the new winner is lower (earlier in the sort order) than the former winner, which in turn is lower than the skipped-over key values between the former winner's old and new locations along the leaf-to-root path. If the new winner is key value A in the theorem, the former winner is key value B, and each skipped-over key value is key value C, then the new offset-value codes for the skipped-over key values is simply the maximum of their existing offset-value codes relative to the former winner and the offset-value code of the former winner relative to new key value.

In order to adjust those offset-value codes, the repair loop of Figure 4 needs another nested loop. Its range matches the first nested loop (lines 35 and 36 in Figure 4). The new nested loop does not change the complexity of the process even if it is no longer strictly true that a tree-of-losers priority queue can absorb any new key in a single leaf-to-root pass.

```
inline void setMax (Key & x, Key const y) { if (x < y) x = y; }

void PQ::pass (Index const index, Key const key, bool full_comp)
{
  Node candidate (index, key);
  Index slot;
  Level level;
  for (leaf (index, slot, level);  parent (slot, level),
      slot != root ()  &&  heap [slot].index != index;  )
    if (heap [slot].less (candidate, full_comp))
      heap [slot].swap (candidate), full_comp = false;

  Index dest = slot;
  if (candidate.index == index)
    while (slot != root ())
    {
      Level const dest_level = level;
      do { parent (slot, level); }
      while ( ! heap [slot].sibling (candidate, dest_level));

      if (heap [slot].less (candidate, full_comp)) break;

      heap [dest] = heap [slot];
      while (parent (dest), dest != slot)
        setMax (heap [dest].key, heap [slot].key);
    }
  heap [dest] = candidate;
}
```

Fig. 7: Repair of offset-value codes added

[4] This theorem implies Iyer's "unequal value theorem" [Iy05]: After offset-value codes decide a row comparison, the loser retains its offset-value code. Formally: $ovc(A,B) < ovc(A,C) \Rightarrow ovc(B,C) = ovc(A,C)$. Proof: $ovc(A,C) = max(ovc(A,B), ovc(B,C)) \wedge ovc(A,C) \neq ovc(A,B) \Rightarrow ovc(A,C) = ovc(B,C)$.

Figure 7 adds a parameter to the “pass” method (line 55) to indicate whether full comparisons ought to be used in the “less” method (lines 62 and 73). If set initially, it remains true until the new key value is in its correct place after the first swap (line 63).

Figure 7 also adds the new loop (lines 76 and 77). The variable “dest” is abused as a looping variable, yet the loop’s continuation condition (“dest != slot” in line 76) echoes the direct move that the loop replaces (“dest = slot” in line 41 of Figure 4). The “parent” method in line 76 is the same as in Figures 1 and 3. The loop body (line 77) applies the theorem to any skipped-over offset-value codes. The “key” field in each tree entry is the offset-value code relative to the local winner. Auxiliary method “setMax” is given in line 53.

Like the key comparison in the search loop (line 62), the “less” method in the repair loop (line 73) must set the loser’s new offset-value code if column value comparisons are required. If the candidate wins, then the key value in the heap is the loser and must be encoded relative to the candidate when it travels backward to the current destination (line 75). Otherwise, the candidate goes back to the current destination (line 79) and must be encoded relative to the (old and new) winner.

In a general scheduling application and its priority queues, many index values will be active and inactive at various times. In a merge step, e.g., in an external merge sort, some of the merge inputs might disconnect from the merge and reconnect later, e.g., during a key range that is not represented in one (or several) of the input runs. A tree-of-losers priority queue models such inactive index values using invalid keys or fences with effective key value $+\infty$. But how does offset-value coding deal with fence keys? What are the offset-value codes for a valid key value relative to an early fence and for a late fence relative to a valid key value?

The solution follows directly from the role of fences: artificial keys preceding the first row and succeeding the last row in a sorted run. If all column values in fences are $-\infty$ or $+\infty$, then the first difference with any valid key value is at offset 0 and the value at that offset is $+\infty$ in a late fence. With these definitions, all algorithms above, including calculation of offset-value codes, work not only with valid key values but also with fences.

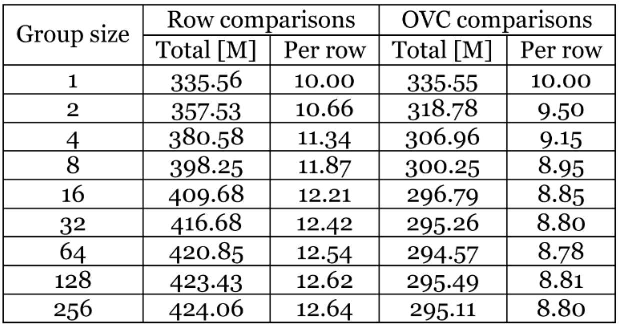
There is one important difference for “pop” and “delete” operations, however. Without offset-value coding, deletion of a valid key value in the root node can simply replace the key value with an early fence. With offset-value coding, offset-value codes in its leaf-to-root path require repair. This new leaf-to-root pass resets all offsets to 0 (because the winner is or would have been the new early fence in the root node); the value is the first column or $\pm\infty$ for fence keys. Alternatively, instead of a “pop” operation, a “top” operation is more desirable and is sufficient if the next operation is a “push” for the same index. For “delete” operations, the optimization above must be disabled in a priority queue with offset-value coding such that deletion always invokes a leaf-to-root pass, even when deleting the key value in the tree’s root.

7 Replacement selection for variable-size records

In the context of external merge sort and specifically run generation, multiple situations require the repair loop of Figure 4. First, in run generation for fixed-size records arriving and evicted in groups (e.g., pages) rather than one record at a time, the logic of Figure 3 can replace multiple entries in the tree by late fences until a group is complete. When new valid key values replace these late fences, the repair logic of Figure 4 is required because all valid key values sort lower than late fences. If the external merge sort uses offset-value coding to reduce column value comparisons to $N \times K$, it requires the new logic of Figure 7.

Second, run generation with variable-size records may require evicting multiple records from the in-memory workspace before it can absorb an arriving large record. In the opposite situation, when the sort logic evicts a large record from the workspace, multiple arriving small records might be inserted, replacing multiple late fences with valid key values.

Third, if a key range of non-trivial size occurs in only one of the merge inputs, there is no need to "merge" each key in this range. After observing successive overall winners from the same merge input, one might want to know the runner-up key and then scan or skip forward within the winning input. One way to determine the runner-up pushes the runner-up to the root by using a late fence to fake the winner's end-of-input. Re-introducing the former winner into the tree-of-losers priority queue after moving a key range directly to the merge output requires the new logic of Figure 7.



Group size	Row comparisons		OVC comparisons	
	Total [M]	Per row	Total [M]	Per row
1	335.56	10.00	335.55	10.00
2	357.53	10.66	318.78	9.50
4	380.58	11.34	306.96	9.15
8	398.25	11.87	300.25	8.95
16	409.68	12.21	296.79	8.85
32	416.68	12.42	295.26	8.80
64	420.85	12.54	294.57	8.78
128	423.43	12.62	295.49	8.81
256	424.06	12.64	295.11	8.80

Fig. 8: Priority queue comparison counts with various group sizes

Figure 8 shows counts of comparisons in a priority queue during run generation in an external merge sort. There are $2^{25} \approx 33.5M$ input rows; the priority queue holds $2^{10} = 1,024$ entries. The run generation logic deletes groups of rows (to fill an output page) and re-inserts the same count of rows (from an input page). The group or page size varies from 1 row, i.e., traditional replacement selection, to 256 rows. The center pair of columns in Figure 8 clearly shows that grouped removal and insertion into priority queues is somewhat less efficient than traditional replacement selection. Large group or page sizes require about 25% more comparisons.

The right pair of columns in Figure 8 shows how many comparisons benefit from offset-value coding and the new techniques of Section 6 and Figure 7. Their practical impact depends on column count, each column's count and distribution of distinct values, etc. This experiment aims to assess the new techniques independently of these parameters. It is obvious that in one-for-one replacement, practically all comparisons benefit from offset-value coding. For larger group or page sizes, the benefit remains substantial as about 70% of all comparisons might be decided by offset-value codes alone, meaning that in practical settings the new techniques saves substantially more than half of the effort for row comparisons.

8 Summary and conclusions

In summary, prior work has shown that

1. a tree-of-losers priority queue requires fewer comparisons than an equivalent traditional tree-of-winners priority queue;
2. internal and external merge sort with tree-of-losers priority queues comes very close to the provable lower bound of $log_2(N!)$ row comparisons for N rows;
3. offset-value coding reduces column or byte comparisons in large keys;
4. the count of column value comparisons in internal and external sorting with offset-value coding is bounded by $N \times K$ for sorting N rows with K key columns, i.e., linear in the count of rows and the count of key columns; and
5. the core logic for a traditional leaf-to-root pass in a tree-of-losers priority queue is small, simple, and efficient.

In addition, new work shows how

6. data-specific analysis of string sorting [Se10] applies equally to prefix sharing in a trie of strings, to offset-value coding in a sorted database table, and to the count of symbol comparisons or of column value comparisons in an internal or external merge sort using tree-of-losers priority queues and offset-value coding;
7. a small extension turns tree-of-losers priority queues into addressable priority queues, i.e., any key value can increase at any time and can be logically deleted at any time;
8. a larger extension turns tree-of-losers priority queues into non-monotone priority queues, i.e., new key values may be lower or higher than earlier key values;
9. a third extension enables efficient incremental maintenance of offset-value codes, even in cases of non-monotone sequences of input key values;
10. the first two extensions permit very efficient double-ended priority queues, among many other scheduling applications within and beyond database systems; and

11. all three extensions together enable new flexibility and efficiency in external merge sort for query processing, index creation and maintenance, database reorganization, and data processing frameworks such as MapReduce and its many successors.

In conclusion, the new techniques for tree-of-losers priority queues and offset-value coding complement recent innovations in database query processing. These innovations include passing offset-value codes from one query execution operation to the next, thus reducing most of their matching logic to comparisons of compiled-in integers rather than comparisons of large records with complex fields and expensive comparison logic [DG22, GD22]. For example, when a query like "... count (distinct...)... group by... " requires first duplicate removal and then grouping, these operations can share not only the sort but also offset-value codes such that the grouping logic never compares column values. For a second example, merge joins of sorted scans or streams can avoid many column value comparisons, perform their required row comparisons using offset-value codes just as efficiently as a hash join does using hash values, and save lots of CPU effort, memory, and overflow (compared to a hash join). In a variation of this example, complex-object assembly by multiple merge joins on the same column can use offset-value codes in all join operations if the join logic produces offset-value codes for its sorted output even if a join suppresses some input rows and duplicates others [GD22]. For a third example, if grouping on a foreign key can be "pushed down" to a join input, early aggregation and wide merging [DGN22] can speed up the sort and offset-value codes from the sort can speed up the merge join. On the other hand, if a grouping operation cannot be pushed down but the merge join produces offset-value codes for its output, then grouping on the join column(s) never needs to compare column values. More examples readily come to mind.

Put differently, wherever query planning can exploit interesting orderings in storage structures and intermediate query results [Se79], query execution can exploit offset-value codes [Co77]. Just as any industrial-grade state-of-the-art query optimizer exploits interesting orderings, query execution should exploit offset-value codes. Although both concepts originated in the 1970s, almost half a century ago, they were linked only recently by widening the scope of offset-value coding from merge sort to all sort-based query execution algorithms. Together, recent innovations for priority queues and for offset-value coding enable sort-based query processing to compete with hash-based query processing and to shine even for large unsorted inputs with large keys.

9 Acknowledgements

Jim Gray suggested exploring offset-value coding. Wey Guy and Thanh Do requested some of the explanations in this paper. Raimund Seidel pointed out bottom-up heapsort and data-specific algorithm analysis. Their help and contributions are gratefully acknowledged.

References

[BE77] Blasgen, Mike W.; Eswaran, Kapali P.: Storage and access in relational data bases. IBM Syst. J., 16(4):362–377, 1977.

[BL89] Baer, Jean-Loup; Lin, Yi-Bing: Improving Quicksort performance with a codewort data structure. IEEE Trans. Software Eng., 15(5):622–631, 1989.

[Br84] Bratbergsengen, Kjell: Hashing methods and relational algebra operations. In: VLDB. pp. 323–333, 1984.

[Co77] Conner, W. M.: Offset-value coding. In: IBM Technical Disclosure Bull. pp. 2832–37, 1977.

[De84] DeWitt, David J.; Katz, Randy H.; Olken, Frank; Shapiro, Leonard D.; Stonebraker, Michael; Wood, David A.: Implementation techniques for main memory database systems. In: ACM SIGMOD. pp. 1–8, 1984.

[DG85] DeWitt, David J.; Gerber, Robert H.: Multiprocessor hash-based join algorithms. In: VLDB. pp. 151–164, 1985.

[DG22] Do, Thanh; Graefe, Goetz: Robust and efficient sorting with offset-value coding. Accepted for publication in ACM TODS, March 2022.

[DGN22] Do, Thanh; Graefe, Goetz; Naughton, Jeff: Efficient sorting, duplicate removal, grouping, and aggregation. ACM TODS, 47(4), December 2022.

[Ep79] Epstein, Robert: Techniques for processing of aggregates in relational database systems. In: Univ. of California at Berkeley, UCB/ERL Memorandum M79/8. 1979.

[Fr56] Friend, Edward H.: Sorting on electronic computer systems. J. ACM, 3(3):134–168, 1956.

[GD22] Graefe, Goetz; Do, Thanh: Offset-value coding in database query processing. submitted for publication, September 2022.

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

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

[Ha77a] Hanani, Michael Z.: An optimal evaluation of Boolean expressions in an online query system. Commun. ACM, 20(5):344–347, 1977.

[Hä77b] Härder, Theo: A scan-driven sort facility for a relational database system. In: VLDB. pp. 236–244, 1977.

[Ho62] Hoare, C. A. R.: Quicksort. Comput. J., 5(1):10–15, 1962.

[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.

[IS56] Isaac, Earl J.; Singleton, Richard C.: Sorting by address calculation. J. ACM, 3(3):169–174, 1956.

[Iy05] Iyer, Bala R.: Hardware assisted sorting in IBM's DB2 DBMS. In: International Conference on Management of Data (COMAD). 2005.

[Kn98] Knuth, Donald Ervin: The art of computer programming, Volume III: sorting and searching, 2nd edition. Addison-Wesley, 1998.

[KTM83] Kitsuregawa, Masaru; Tanaka, Hidehiko; Moto-Oka, Tohru: Application of hash to data base machine and its architecture. New Gener. Comput., 1(1):63–74, 1983.

[LG98] Larson, Per-Ake; Graefe, Goetz: Memory management during run generation in external sorting. In: ACM SIGMOD. pp. 472–483, 1998.

[NKT88] Nakayama, Masaya; Kitsuregawa, Masaru; Takagi, Mikio: Hash-partitioned join method using dynamic destaging strategy. In: VLDB. pp. 468–478, 1988.

[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.

[Se79] Selinger, P. Griffiths; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G.: Access path selection in a relational database management system. In: ACM SIGMOD. p. 23–34, 1979.

[Se10] Seidel, Raimund: Data-specific analysis of string sorting. In: ACM-SIAM SODA. pp. 1278–1286, 2010.