CrocSort: Resource-Efficient, Skew-Resilient Parallel External Merge Sort

Riki Otaki*
University of Chicago
rotaki@uchicago.edu

Charles Benello*
University of Chicago
cmbenello@uchicago.edu

Fuheng Zhao
University of Utah
zhaofuheng1@gmail.com

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Goetz Graefe
Google
goetzg@google.com

ABSTRACT

Sorting is a core operator in large-scale data systems. As data increasingly exceeds main memory, external merge sort is essential, yet many implementations over-allocate memory and over-parallelize, decreasing efficiency. We present **CrocSort**, a byte-balanced parallel external merge sort with configurable memory and per-phase thread settings. Using analysis and experiments, we derive practical resource-configuration rules for selecting these settings from input size, memory budget, and thread cap. To balance parallel merge under skew, CrocSort reuses run sparse indexes for range partitioning to create a virtual total order over records. CrocSort also uses offset-value codes and related optimizations to reduce comparison work and, for prefix-redundant workloads, intermediate I/O volume. Across TPC-H and synthetic workloads on modern NVMe systems, CrocSort completes sorting at memory budgets where production systems abort, and the planner reduces unnecessary resource allocation compared to the greedy maximal approach across both tight- and ample-memory regimes.

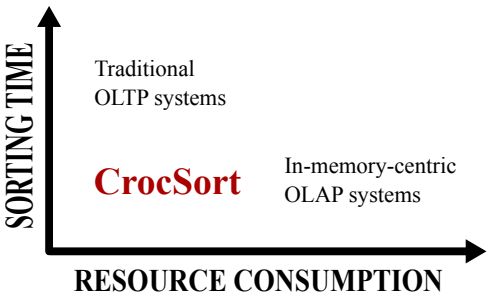


Figure 1: Resource–Performance Tradeoff in External Sorting. — Modern OLAP engines consume large amounts of memory and threads to sort larger-than-memory data, while traditional OLTP sorters conserve resources at the cost of longer execution times. CrocSort narrows this gap with configuration-first external sorting: it jointly plans memory and phase-specific parallelism to keep merge schedules shallow while sustaining high throughput.

PVLDB Reference Format:
Riki Otaki, Charles Benello, Fuheng Zhao, Aaron J. Elmore, and Goetz Graefe. CrocSort: Resource-Efficient, Skew-Resilient Parallel External Merge Sort. PVLDB, 19(11): 2978 - 2991, 2026.
doi:10.14778/3836663.3836667

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rotaki/ES.

1 INTRODUCTION

Sorting is a major cost in modern data systems, underlying data loading and index creation [22, 24], deduplication and aggregation [13, 27], top-k [8], and temporal joins [56, 75]. As data outgrows memory, external sorting becomes essential, and in shared cloud environments [47, 54] the incentive shifts toward algorithms that sustain performance under tight memory budgets.

External sorting has been extensively researched in terms of optimizing run generation [22, 41, 52], memory layout during run generation [44, 45], merge-tree scheduling for reduced I/O [21, 36, 42, 63, 79], overlapping I/O and compute [36, 63, 79], and comparison-acceleration techniques [12, 13, 22, 25]. Parallel run formation and merging have also been studied across shared-nothing designs, cache-conscious approaches, skew handling, and modern SSDs [2, 11, 14, 17, 32, 41, 49, 51, 52]. However, these systems typically optimize performance under fixed memory and number of threads, rather than choosing how much memory and parallelism a workload should use. In shared-memory thread-pool executors (including morsel-driven designs [46]), adding threads under a fixed memory budget reduces per-thread run memory, shortens runs, and can force deeper merge schedules; allocating excessive run-generation memory can also hurt once the single/few-step merge threshold is already reached. This creates a joint constraint that systems typically leave implicit.

We address this configuration problem with **CrocSort**, a budgeted parallel external sorter and a practical policy for configuring it. We present the design first, then derive memory/thread feasibility constraints and merge-I/O consequences, describe implementation details, and finally calibrate and validate the policy experimentally. The resulting guidance is to favor single/few-step merge plans, allocate more threads to run generation than merge when merge saturates earlier, and right-size run memory instead of

*These authors contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836667

maximizing run size. To keep merge balanced under skew, CrocSort reuses run sparse indexes for duplicate-safe Count-Balanced and Byte-Balanced range partitioning (Section 3), and combines replacement selection, tree-of-losers operators, budgeted memory reuse, and OVC to reduce comparison cost and intermediate I/O volume [13, 22, 25, 26, 28]. This paper makes the following contributions:

- **Bounded-memory parallel sort execution.** CrocSort integrates established techniques—replacement selection, a tree of losers, OVC, and sparse indexes—under a single explicit resource budget so the sort runs stably under tight memory constraints, holding memory near the configured limit across run generation and multiple merge steps while keeping comparison and I/O cost low on prefix-redundant workloads.
- **Sparse-index range partitioning for balanced merge.** We reuse sparse indexes built during run generation to support Key-Only, Count-Balanced, and Byte-Balanced partitioning; Byte-Balanced equalizes I/O across merge threads with split boundaries robust to both duplicate and size skew, keeping the merge well-balanced under skew without extra preprocessing.
- **Resource-aware planner with end-to-end validation.** We derive how memory and per-phase thread counts couple to determine merge depth and I/O cost. Combined with empirical phase asymmetries and a two-sided memory-scaling threshold, this yields a planner that cuts resource use without sacrificing sort throughput. CrocSort succeeds at memory budgets where DuckDB and ClickHouse OOM, and the planner uses fewer resources than a greedy maximal baseline in both tight- and ample-memory regimes.

2 BACKGROUND AND RELATED WORK

This section covers algorithmic and systems background for external sorting (run generation, merge, parallelization, and merge partitioning).

Foundations of External Sort. External sorting follows two broad paradigms: *merge-based* (generate sorted runs and merge them) and *distribution-based* (partition the key space, sort each partition, concatenate) [21, 22, 36, 37]. Distribution-based designs shift complexity to boundary selection and skew handling, so general-purpose systems (Unix `sort`, PostgreSQL, DuckDB) typically use merge-based external sort [41, 70, 72]. We focus on merge-based external sort, with emphasis on *configuration*: thread parallelism and memory budgets jointly determine run length and merge depth, which dominate end-to-end performance.

Run Generation Techniques. Run generation techniques trade off run length, CPU cost, and I/O efficiency [22]. Two common methods are *load-sort-store* and *replacement selection*. Load-sort-store reads a memory-sized chunk, sorts it in memory (e.g., quicksort or radix sort), and writes it out; representative optimizations include AlphaSort's cache-efficient key/pointer sorting [52] and modern sorting pipelines in DuckDB [40, 41]. Replacement selection uses a priority queue and offers two key advantages: it can overlap I/O and computation efficiently, and it adapts to input presortedness—producing runs approximately twice the available memory on random inputs and even longer runs on nearly sorted data [20, 22, 36, 44, 45]. These techniques are typically analyzed assuming a fixed effective memory budget per sorter; our focus is complementary: thread allocation changes effective per-thread memory and thus run length and merge depth.

Merge Techniques. Merging sorted runs is the second phase of external merge sort. Its cost is governed by merge degree (fan-in) and buffering, which determine merge-step count, total I/O volume, and per-record CPU overhead [22, 36, 40, 41, 45, 55, 63, 76, 79]. When one merge step cannot consume all runs, additional steps reread and rewrite data and often dominate end-to-end time. Multiway merging typically uses a priority queue or tournament tree; a tree of losers (a tournament-tree variant that stores the loser at each internal node) reduces comparisons per output element [18, 22, 25, 36]. Double buffering and prefetching/forecasting overlap I/O and computation [21, 36, 63, 79]. Our focus is instead configuration under fixed memory budgets.

Offset Value Coding. Offset-value coding (OVC) [3, 12, 13, 22, 25, 26] encodes a key comparison by a pair (o, v), where o is the first differing-byte offset from a reference key and v is the differing byte value. OVCs computed against the same reference key can be compared by (o, v), with a fallback key comparison only when the OVCs tie. In merge sort, OVCs can be produced during run formation and reused during multiway merge; a tree of losers updates these codes efficiently along the leaf-to-root path [25]. OVC is especially effective for normalized keys [22, 41], where compact OVC metadata can reduce comparison CPU work and full-key accesses, and it aligns with prefix truncation since o also reveals a shared-prefix length [7, 23, 24, 28]. We do not contribute to OVC itself—dedicated work studies its behavior across data types and key-length distributions [12, 25, 26, 28]. Our contribution is to integrate OVC under tight external-sort memory budgets and to evaluate how it interacts with multi-step merge pipelines.

Parallelizing External Merge Sort. Parallel external sorting is commonly categorized by how parallelism is applied in run generation and merging [69]. Run generation is typically parallelized by partitioning the input into chunks and sorting them independently [14, 52, 69]; modern engines such as DuckDB use morsel-driven scheduling so workers dynamically fetch input morsels [40, 41, 46]. Parallel merging is commonly realized in two forms: hierarchical parallel merge and range-partitioned (key-range) merge. Hierarchical parallel merge assigns independent merge subtasks to threads, forming a tree of merge tasks that can proceed concurrently; PostgreSQL and ClickHouse exemplify this approach with worker-local sorting followed by a global merge [65, 70]. Range-partitioned merge assigns disjoint key ranges to threads and therefore depends on balanced split points. For two-way merges, MergePath can compute exact balanced cuts [40, 53]; for multiway merges, systems usually rely on sampling-based approximations [1, 11, 32, 49]. DuckDB previously used parallel 2-way range-partitioned merge (MergePath) and recently moved to k-way range-partitioned merge as of version 1.4 [39–41]. Hierarchical merge spans multiple merge-tree depths, materializing intermediate runs at each level and requiring global-merge coordination, whereas range-partitioned threads merge their key range in a single level, independently. Hierarchical merge can use a smaller per-merge fan-in and larger per-stream buffers; buffer sizing, double buffering, and forecasting are future

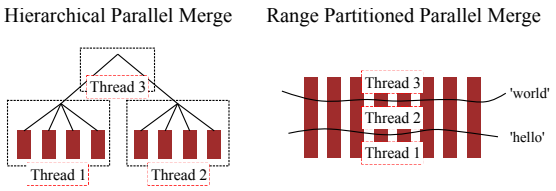


Figure 2: Parallel Merge Strategies. — Illustration of the two common parallel merge paradigms: hierarchical parallel merge (left), which assigns independent merge subtasks to threads, and range-partitioned merge (right), which assigns disjoint key ranges to threads.

work. These categories describe algorithmic structure but offer limited guidance on jointly choosing threads and memory allocation across phases under a fixed budget, which our work addresses.

Range Partitioning for Parallel Merge. Range partitioning is a common strategy for parallel multi-way merge, and prior work uses compact distribution summaries for boundary selection, including histograms, quantile sketches, and sampling [11, 16, 29, 35, 67, 77, 78]. Accurate boundary selection is critical: skewed partitions leave some workers with disproportionately large ranges, creating stragglers that bottleneck overall throughput. In-memory studies also show that partitioning strategy can strongly affect sorting throughput [57]. Many such methods primarily target equal tuple counts. Under heavy duplicates, value-based splitters can become unsplittable, so skew-robust systems partition by output rank (merge-path style) and/or use deterministic tie-breakers to split equal-key groups while preserving global order [40, 53]. A related limitation appears in LSM compaction: RocksDB subcompaction uses sampled anchor keys, but user-key atomicity and deduplicated boundary keys can still prevent splitting a heavy duplicate key [15, 33, 61]. Prior work on size skew emphasizes weighted/byte-aware partitioning and exact or bounded splitter selection [6, 59, 62, 73]. These limitations motivate duplicate-safe and byte-aware boundary selection. CrocSort's sparse run indexes, built during run generation, make such selection available as a natural byproduct (Section 3).

Learned, Distributed, and Cloud-Native Sort. Several directions are complementary rather than competing. Learned sorting predicts each key's final position to replace comparison-based ordering [38], and in external-sort contexts learned models have been applied to generating partition split points [37]; our approach instead uses deterministic byte-balanced sparse-index splits, which achieve strong balance in our evaluated workloads. Distributed-memory external sort [34, 59, 60, 64] composes per-node external sorters with cross-node shuffle, and our configuration-first argument applies per shared-nothing worker; byte-balanced sparse-index partitioning could extend to cross-node range partitioning. Cloud-native sort over object storage [48, 58, 66, 74] operates under a latency-bound, per-request billing cost model that warrants its own study. This work is scoped to single-node, disk-to-disk external sort on NVMe; we do not extend the experimental study to these settings.

Storage-Specialized External Sort. NVSort [51], an external sort optimized for NVMe SSDs, pairs sampling-based pre-sort range partitioning with rw-lock write arbitration and adaptive block sizing, but does not model multi-step merge cost. CrocSort instead schedules merge depth under the memory budget and uses replacement selection so reads and writes interleave continuously; the rw-lock is not a natural fit because it assumes a clean read/write phase boundary that replacement selection dissolves. Adaptive block sizing could be combined with CrocSort, but it optimizes a different goal: improving the sort pipeline with additional memory, whereas we study memory/thread tradeoffs under explicit budgets. We include adaptive block sizing in the NVSort baseline but not in CrocSort. MONTRES and MONTRES-NVM [5, 43] overlap merging with run formation to reduce SSD/NVM I/O traffic, addressing a different question than ours: we study how to operate under tight resource budgets rather than how to exploit additional memory for phase overlap.

3 CROCSORT DESIGN

To address the configuration gap identified in Section 2, we implement CrocSort, a disk-to-disk external sorting engine in Rust for datasets that exceed available memory. CrocSort's goal is to use resources only as needed while maintaining throughput. Sections 4 and 6 show how to configure these resources; this section describes the execution pipeline.

Overview. CrocSort currently supports row-format inputs that can be scanned in parallel and writes sorted data in a custom binary format that stores sort keys and payloads sequentially with their lengths for efficient downstream reads. The engine follows the structure of classical parallel external merge sort [68] with configurable memory and phase-specific thread settings. Figure 3 presents this overview. The algorithm proceeds in three phases: (1) *parallel run generation*, (2) *boundary selection*, and (3) *parallel range-partitioned merge*. The following subsections describe execution under a given configuration, whether selected by our policy or provided externally.

Parallel Run Generation. Given T threads, we first partition the input file into T disjoint segments by file size. For variable-length rows, each segment boundary is aligned to a record boundary using row-start positions when available (or a local scan to the next row delimiter, e.g., a newline in CSV). Each thread is assigned one segment and processes it independently. Each thread maintains a tournament tree (a tree of losers [18]) and fills it by reading rows from its assigned segment. We use replacement selection [20] to form runs: once the tree is full, the thread starts emitting records from the tree top into the current run. A new record from the input is pushed into the tree of losers if it is greater than the previously emitted record in the sort order. Otherwise, it is buffered to build the next run. This process repeats until all assigned input pages are processed.

Each run keeps periodic key/offset samples from the replacement-selection output stream while the run is written to disk, forming a sparse index that enables fast range scans over the run during the merge phase. We reserve up to 5 % of total memory for sparse indexes as a bounded metadata overhead and derive sampling strides from that budget.[1] Figure 3 (left) illustrates the process.

[1]An ablation over 1%–10% shows boundary quality is insensitive to this parameter while merge-side I/O degrades below 5%.

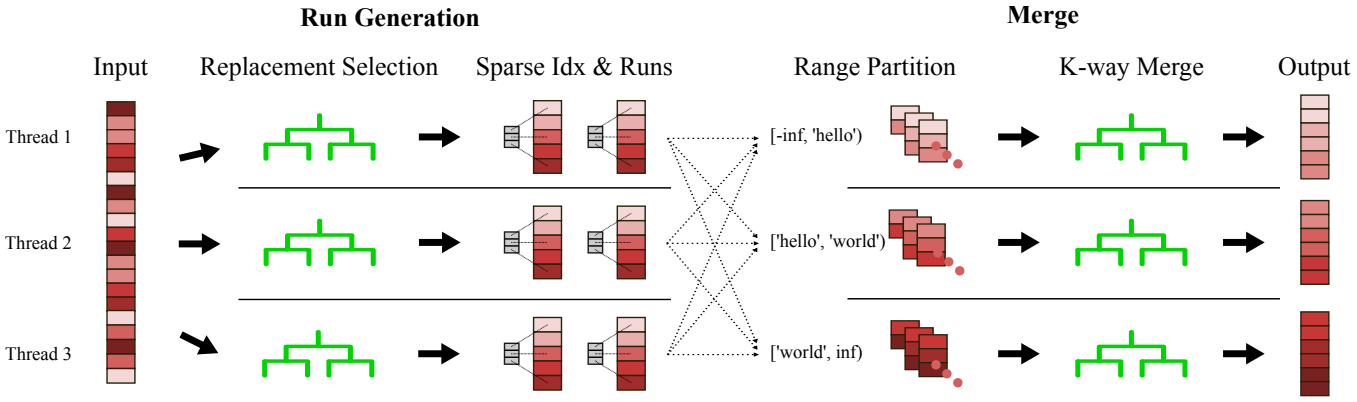


Figure 3: CrocSort External Sorting Pipeline. — Left: run generation. Middle: sparse-index-based range partitioning. Right: range-partitioned merge.

Sparse-Index Range Partitioning. We use range-partitioned merge to avoid the intermediate-run I/O, extra comparisons, and global coordination of hierarchical merge. Range partitioning instead partitions the output key space into disjoint ranges so that threads can independently merge the same set of runs without fine-grained synchronization. The critical challenge is selecting partition boundaries that balance workload across threads. Naive value-based (key-only) splitting cannot distinguish between duplicate keys, and equal tuple counts can still produce severe I/O imbalance under size skew. We address this by reusing sparse indexes already constructed during run generation. Section 5 presents the boundary-selection algorithm in detail. The key insight is that the byte offset already stored in each sparse entry serves a dual role: as a tie-breaker for splitting duplicate keys and as a direct I/O signal for byte-balanced partitioning. The sparse indexes are shared and accessible by all threads, so each thread independently derives its own $[low, high)$ boundary points. Figure 3 (middle) illustrates the process.

Parallel Range-Partitioned Merge. Each thread merges the same set of runs for its assigned key range. Each run is exposed via a range iterator that holds one read page, replaces it with the next page when consumed, and yields only tuples inside the assigned range. The range scan starts from the position corresponding to the lower bound by using the sparse index created during run generation. Each thread maintains a tree of losers over active iterators to perform the merge. Each thread also maintains a one-page output buffer for writing merged results. If the number of active iterators exceeds the maximum fan-in supported by the memory budget, we perform multi-step merging by iteratively merging the shortest runs in the run pool [36, 45]. CrocSort is an unstable sort because shortest-first (greedy) merging does not preserve input order. Figure 3 (right) illustrates the process. We next model the resource constraints that govern this pipeline.

4 ANALYTICAL MODEL FOR CROCSORT

This section derives an analytical model for the design in Section 3. Since data exceeds memory in external sorting, I/O dominates performance, making I/O cost the central focus under a fixed memory budget. Prior models mainly assume single-threaded execution [21, 45, 68]; our key contribution is capturing how phase-specific thread counts interact with memory. Under a fixed budget, more run-generation threads means less memory per thread, shorter runs, and more runs; more merge threads means higher buffer demand. We analyze this coupling among memory (M), input size (D), page size (P), and thread counts ($T_{\text{gen}}, T_{\text{merge}} \le T_{\max}$), identifying when single-step merging is feasible, how deeper schedules increase I/O, and when reducing parallelism is preferable. Table 1 summarizes the notation.

Table 1: Notation and Parameters

Symbol	Description	Unit
D	Input size	bytes
M	Total memory budget	bytes
ρ	Usable memory ratio ($M_{\text{eff}} = \rho M$)	unitless
E	Replacement-selection run-expansion factor	unitless
T_{gen}	Threads for run generation	count
T_{merge}	Threads for merge	count
$T_{\max}$	Hardware threads available	count
P	I/O page size (per input buffer)	bytes
RS	Average run size	bytes
N	Number of runs ($N \approx \lceil D/RS \rceil$)	count
K	Number of merge operations (steps)	count
F	Merge fan-in	count

Run Generation. We define an effective budget $M_{\text{eff}} = \rho M$ (ρ=0.8 in our implementation) to capture fixed per-thread overheads (32-block I/O buffers, tournament-tree metadata). Replacement selection produces an average run of size RS using roughly RS/E memory per thread, where E is the replacement-selection expansion factor (E=2 in the classical random-key estimate [36]; E=1 for load-sort-store sorters). With T_{gen} independent run generators, respecting the memory budget requires

$$\frac{RS}{E} \cdot T_{\text{gen}} \le \rho M \quad \Longrightarrow \quad RS \le \frac{E \rho M}{T_{\text{gen}}}. \tag{1}$$

We ignore output buffering for clarity.

Range-Partitioned Merge. Range partitioning assigns each merge thread a key range and has it merge that range from all runs. Each merge thread keeps one input buffer (one page) per run, so the

merge fits in memory in a *single-step* if

$$N \cdot T_{\text{merge}} \cdot P \le \rho M. \tag{2}$$

Using $N \approx D/RS$, we obtain a lower bound on run size:

$$\frac{D}{RS} \cdot T_{\text{merge}} \cdot P \le \rho M \quad \Longrightarrow \quad RS \ge \frac{D\, T_{\text{merge}}\, P}{\rho M}. \tag{3}$$

We ignore output buffers during merge; for $N \gg 1$, the output footprint is $O(T_{\text{merge}} P)$. This is negligible compared to the input-buffer footprint $O(N \cdot T_{\text{merge}} \cdot P)$. Using multiple buffers per run (e.g., double buffering) tightens Eq. 2.

Single-step Feasibility Window. We call a configuration *single-step feasible* if, under the fixed memory budget and phase-specific thread allocation, the sort can complete merge in one step without exceeding the budget. This is a resource-feasibility notion, not a performance guarantee. Combining Eq. 1 and Eq. 3, a single-step merge is possible iff there exists an RS satisfying

$$\frac{D\, T_{\text{merge}}\, P}{\rho M} \;\le\; RS \;\le\; \frac{E\rho M}{T_{\text{gen}}}. \tag{4}$$

Equivalently, the single-step window is non-empty iff

$$T_{\text{gen}} \cdot T_{\text{merge}} \;\le\; \frac{E\rho^2 M^2}{D\, P}. \tag{5}$$

Eq. 5 exposes the key planning rule: *single-step feasibility depends on the product of the phase parallelisms.* Intuitively, increasing T_{gen} reduces per-thread run-generation memory, which shortens runs and increases the run count N; this increases the merge-buffer footprint, so maintaining a single-step merge requires reducing T_{merge} (and vice versa).

Baseline: Symmetric Parallelism. Many external sorts tie the two phases to the same parallelism, $T_{\text{gen}} = T_{\text{merge}} = T$. Substituting into Eq. 5 yields the familiar single-step boundary

$$T \le T^* := \frac{\rho M \sqrt{E}}{\sqrt{D\, P}}. \tag{6}$$

When $T > T^*$, the feasible window in Eq. 4 is empty under symmetric parallelism: more threads shrink the per-thread run memory (smaller RS), increasing the run count N and therefore the merge-buffer footprint. Because external sorting is typically I/O-dominated, this boundary matters: crossing it can push the sort out of the single-step regime and introduce extra merge I/O. Figure 4 makes this explicit by showing (a) where the single-step region disappears under symmetric parallelism, (b) the resulting I/O increase, and (c) the induced tradeoff boundary between T_{gen} and T_{merge} once single/few-step merging is the goal.

When Baseline Exceeds T^*: Multi-step Merging. If $T_{\max} > T^*$ and we insist on symmetric parallelism ($T_{\text{gen}} = T_{\text{merge}} = T_{\max}$), then a single-step merge is infeasible and the merge must be performed in multiple steps. Here, a *step* means one merge operation that consumes a subset of runs and produces one output run. Let K denote the number of such operations. This is *not* a full-data merge pass: under Huffman merging [31, 36, 45, 50], different bytes may traverse different numbers of merge operations, which we capture later via L_{avg}. Under the Huffman objective, we iteratively merge the shortest runs first [36, 45]. For tractable planning, we use a coarse average approximation: we treat these K operations as evenly partitioning N runs, so each operation merges about N/K runs on average. This reduces the per-operation number of runs that must be buffered and relaxes the merge-memory lower bound by a factor of K:

$$RS \ge \frac{D\, T_{\text{merge}}\, P}{K\, \rho M}. \tag{7}$$

This expands the feasible window in Eq. 4 to

$$\frac{D\, T_{\text{merge}}\, P}{K\, \rho M} \;\le\; RS \;\le\; \frac{E\rho M}{T_{\text{gen}}}. \tag{8}$$

The multi-step feasibility condition is

$$T_{\text{gen}} \cdot T_{\text{merge}} \;\le\; \frac{EK\, \rho^2 M^2}{D\, P}. \tag{9}$$

In the symmetric case ($T_{\text{gen}} = T_{\text{merge}} = T$), this becomes

$$T \;\le\; \frac{\rho M \sqrt{EK}}{\sqrt{D\, P}}. \tag{10}$$

Figure 4a shows that larger K permits more symmetric parallelism at the cost of more merge I/O.

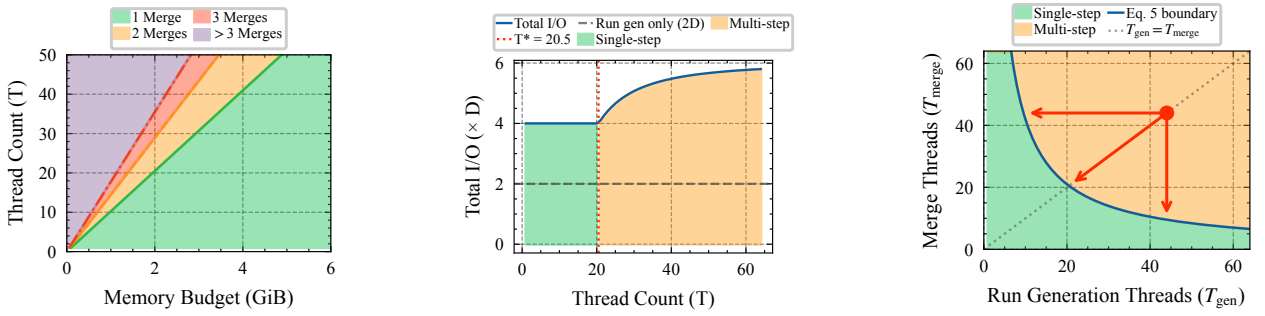
I/O Cost of Multi-step Merging. Assuming pre-formatted binary input, run generation reads and writes D bytes. Thus, the I/O cost of run generation is $2D$. For merging, the I/O cost depends on the merge schedule. We use the Huffman-merge schedule, which minimizes bytes moved by greedily merging the F shortest runs at each merge operation (after padding with zero-size dummies only to satisfy the F-ary tree shape).[2] This produces a graceful, continuous growth in I/O as the number of runs increases, rather than discrete step jumps each time an additional merge pass is required. Intuitively, bytes placed higher in the merge tree are rewritten fewer times than bytes placed deeper in the tree. Let L_{avg} denote the average number of merge operations traversed by real data bytes under this schedule. Each traversal incurs one read and one write, so merge I/O is $2D\, L_{\text{avg}}$. Including run generation ($2D$), the total sort I/O is $2D(1 + L_{\text{avg}})$.[3] Thus, larger L_{avg} directly captures the I/O penalty of multi-step merging relative to single-step merging. Figure 4b shows total I/O cost as a function of thread count.

Avoiding Multi-step: Decoupling and Reducing Parallelism. Multi-step merging is not the only way to execute the sort. When the baseline violates Eq. 6, we can instead decouple the phase parallelism and choose $(T_{\text{gen}}, T_{\text{merge}})$ to satisfy the general single-step condition in Eq. 5. Natural choices include reducing run-generation parallelism or reducing merge parallelism; more generally, any $(T_{\text{gen}}, T_{\text{merge}})$ below the hyperbolic boundary in Eq. 5 recovers a non-empty run-size window (Eq. 4) and permits a single merge step. Figure 4c illustrates this trade-off and the feasible single-step region.

These constraints define the feasible single/few-step region and sharply narrow the configuration space, but they do not by themselves determine a universal throughput optimum. We therefore

[2] Dummies shape the merge tree but contribute no I/O.

[3] Equal-run closed form used in our plots: assume N real runs of size RS (total $D = NRS$), let $d \equiv (1 - N) \bmod (F - 1)$, $N' = N + d$, and $h = \lceil \log_F N' \rceil$. In the optimal F-ary Huffman tree for equal weights, real leaves occupy depths $h - 1$ and h. If t leaves are at depth $h - 1$, then $N' = t + F(F^{h-1} - t) = F^h - (F - 1)t$, so $t = (F^h - N')/(F - 1)$. Thus $L_{\text{avg}} = \frac{t(h-1)+(N-t)h}{N} = h - \frac{F^h - N'}{(F-1)N}$, Merge I/O = $2DL_{\text{avg}}$, and Total I/O = $2D(1 + L_{\text{avg}})$. For single-step merge, $L_{\text{avg}} = 1$ and merge I/O is $2D$.



(a) Single-Step Feasibility Boundary **(b) I/O Penalty Beyond Boundary** **(c) Thread-Allocation Tradeoff Plane**

Figure 4: Symmetric Parallelism (T_{gen}=T_{merge}=T) and Its Limits. — (a) Single-step feasibility under symmetric parallelism. (b) I/O penalty once $T > T^*$ forces multi-step merging. (c) Thread tradeoff from Eq. 5 in the (T_{gen}, T_{merge}) plane; from (T_{max}, T_{max}), moving left/down reduces one phase and moving diagonally reduces both to recover (or approach) single-step feasibility. All panels use D=200 GiB, M=2 GiB, ρ=0.8, P=64 KiB, and E=2.

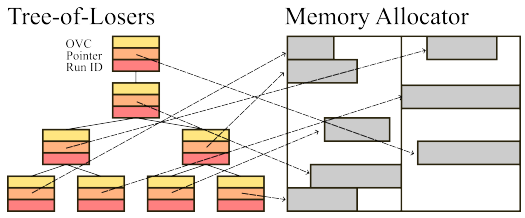


Figure 5: Memory Budget Allocation During Run Generation. — Illustration of the run-generation budget split used in our implementation.

use the empirical characterization in Sections 6.1–6.5 to derive and calibrate the resource-configuration guidelines summarized in Section 6.6.

5 IMPLEMENTATION

Building on the design (Section 3) and analytical observations (Section 4), this section details the concrete data structures and algorithms used in CrocSort. Our implementation focuses on three key aspects: (1) staying within a tight memory budget throughout run generation and merging, (2) fast and accurate boundary selection using sparse indexes to enable balanced parallel merging, and (3) cheap key comparison and I/O reduction via offset-value coding (OVC).

Memory Management during Run Generation. Run generation must divide memory between record storage, tree metadata, and sparse-index samples. After the 5 % sparse-index reservation, the remaining 95 % of the total memory budget is devoted to run generation (Figure 5). We implement a custom memory allocator with a best-fit strategy to manage records during run generation, inspired by previous work on memory management for external sorting [44, 45]. The allocator organizes memory into a pool of fixed-size *extents* (64 KiB each). A free-list array, indexed by block-size class (in multiples of 32 B), tracks available blocks within the pool. When an allocation request arrives, the allocator selects the smallest free block that satisfies the request; if none exists, a new extent is added until the memory budget is exhausted. When a block is freed it is coalesced with its neighbors to reduce fragmentation.

During run generation each thread allocates payload buffers for records from the custom allocator. The tree of losers, however, is allocated separately on the heap: its nodes hold only an allocator handle and a small amount of metadata. Concretely, each node is 12 B in the OVC variant (4 B OVC state + 4 B allocator handle + 4 B index) and 8 B in the plain variant (4 B allocator handle + 4 B index). To keep run-generation memory stable, we split the per-thread budget between the replacement-selection tree and record storage. Let X denote the per-thread run-generation budget, n the number of records resident in the tree, R the average record-storage bytes per record, and S the tree-node size. Because tree capacity is rounded to the next power of two in our implementation, the tree footprint can become significant for small records; using the conservative bound $\text{next\_pow2}(n) \le 2n$ yields the allocator budget target. The basic per-thread budget constraint is

$$\text{next\_pow2}(n)\, S + nR \le X, \tag{11}$$

which gives

$$L^* = nR \le \frac{X \cdot R}{R + 2S}. \tag{12}$$

Because R is unknown initially, the allocator is initially restricted to 50 % of X (a conservative cap for small records) and then raised once to L^* using the measured average record size. Table 2 shows the resulting tree/allocator split for representative block sizes. Calibration is one-shot: L^* is set after the first record-size measurement, which has already seen ~50% of X worth of records and is representative.

Variable Length Records and Tree Slot Reuse. Slot reuse in the tree of losers is irregular: a large incoming record can force multiple small records to be emitted, while a small record can fit without opening new space [23]. We track reusable slots with a bitset-backed free-slot map. The 64 KiB extent with best-fit allocation follows Larson and Graefe [45], who report ~85% utilization at typical sort-workspace sizes and a bounded best-fit search depth in this regime. When an incoming record fits no free block, CrocSort force-evicts a resident record to the next run. Records exceeding one extent are out of scope for the in-page allocator; a natural fit is

Table 2: Tree/Allocator Memory Split as a Function of Block Size R, Derived from Eq. (12). OVC nodes are 12 B ($2S$=24 bytes); plain nodes are 8 B ($2S$=16 bytes).

	OVC ($2S$=24 bytes)		Plain ($2S$=16 bytes)	
R (bytes)	Tree	Allocator	Tree	Allocator
32	42.9%	57.1%	33.3%	66.7%
64	27.3%	72.7%	20.0%	80.0%
128	15.8%	84.2%	11.1%	88.9%
256	8.6%	91.4%	5.9%	94.1%
512	4.5%	95.5%	3.0%	97.0%
1024	2.3%	97.7%	1.5%	98.5%
4096	0.6%	99.4%	0.4%	99.6%

key–value separation [4, 30], which we expect to compose with our partitioning since boundaries are on fixed-size keys.

Run File Layout. CrocSort packs many logical runs into a smaller number of physical files: each thread writes to one file at a time, and runs record their starting byte offset within the container. The physical-file count is bounded by $T_{\text{gen}} + K \cdot T_{\text{merge}}$, independent of the run count N, which avoids file-count growth as input size increases. A shared FD per file uses `pread`/`pwrite` so it is never seeked explicitly. Compared to a one-run-per-file layout, this can introduce non-sequential access patterns within a container file; NVMe devices generally tolerate such patterns well, and a quantitative comparison of the two layouts is left to future work.

Why Sparse Indexes Are Sufficient for Range Partitioning. Each run is written as a sorted stream, so periodic sampling is optimal for capturing its key distribution [9]—unlike general-purpose sketches [35, 78] built for arbitrarily ordered inputs. Across runs, we define a virtual total order by comparing entries lexicographically by (key, run_id, run_offset), with no physical merge (of sparse indexes) required. The sparse indexes are compact ($< 5\%$ of the memory budget in our implementation) and require no extra preprocessing. Sparse indexes solve two distinct partitioning problems with the same metadata. General key sketches summarize value distributions, but duplicate-heavy keys remain unsplittable without a deterministic per-record tie-breaker; the virtual order (key, run_id, run_offset) supplies that tie-breaker. Likewise, count-oriented summaries can balance tuple counts but not byte volume when payload sizes vary; run_offset encodes cumulative byte position, enabling byte-balanced boundaries under record-size skew.

Partitioning Policy Variants. On the same sparse-index mechanism, we support three boundary-selection policies: **Key-Only** (partition by key value only, no duplicate splitting), **Count-Balanced** (use the virtual total order to split duplicates and target equal tuple counts), and **Byte-Balanced** (use the same order but target equal byte volume). The chosen policy also determines how the sparse index is sampled during run generation. Key-Only and Count-Balanced use count-based sampling: a fixed tuple-count stride is used as the sampling interval. Byte-Balanced uses byte-based sampling: a fixed byte-volume stride is used instead. Both modes use a running estimate of average tuple size to convert the sparse-index memory budget into a stride, and update the next sampling target as `current_position + stride` after each sampled entry, keeping memory near budget while aligning samples with the boundary objective. Intervals are adjusted dynamically so that sparse-index memory usage tracks the configured budget with only small deviation.

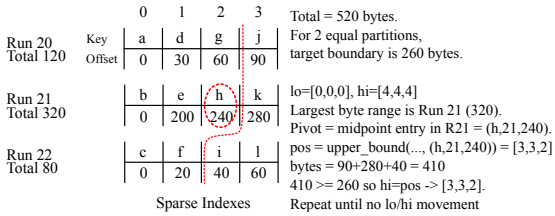


Figure 6: Sparse Index Boundary Selection. — Illustration of the multi-index binary search used to find partition split points across N sparse indexes under the Byte-Balanced policy. Each iteration selects a pivot from the run with the largest remaining index range and evaluates cumulative byte volume to converge on a target boundary.

Boundary Selection Algorithm. Figure 6 illustrates the boundary selection process under the Byte-Balanced policy. Before merging begins, each thread accesses the sparse indexes from the N runs to be merged. We compute the total objective value to be balanced (tuple count for Key-Only and Count-Balanced or byte volume for Byte-Balanced) across all runs. Each merge thread t then independently computes its boundaries to target its share of that objective, e.g., $[t \cdot O/T, (t+1) \cdot O/T)$ for objective total O. To find the sparse-index entry corresponding to a target objective value b_{target}, we perform a multi-index binary search directly on the N sparse indexes:

(1) **Pivot selection:** Maintain a current index range in each run's sparse index and take its midpoint as a pivot candidate. At each iteration, choose as the pivot the candidate from the run with the largest remaining range.
(2) **Objective evaluation:** For the chosen pivot, perform an upper-bound search in each of the N sparse indexes to determine how many tuples/bytes lie before that pivot under the virtual total order over key, run ID, and offset. Summing these N results yields the cumulative objective value. If it is less than b_{target}, search higher; otherwise search lower by updating the per-run ranges accordingly.

The search terminates after I pivot iterations, where $I = O(\log S)$ and $S = \sum_{i=1}^{N} n_i$ is the total number of sparse-index entries across runs. At that point only the boundary-adjacent entries from each run remain candidates. We evaluate this candidate set in sorted order and select the first entry whose cumulative objective coverage reaches or exceeds b_{target}. This boundary becomes the partition split point.

Boundary selection is performed independently per thread without coordination, so the boundary-search phase adds no inter-thread synchronization. Each boundary costs $O(N \log n \log S)$ per thread plus an $O(N)$ final candidate scan,[4] where n is the size of the largest sparse index; in practice this cost is negligible compared to

[4]Each pivot iteration costs $O(N \log n)$ because we perform N upper-bound searches (one per run), each taking $O(\log n)$ over that run's sparse index. With $I = O(\log S)$ pivot iterations, each boundary costs $O(I\,N \log n) = O(N \log n \log S)$, followed by an $O(N)$ final candidate scan.

merge I/O. Since boundaries are derived from sparse indexes, selection is approximate but bounded: with interval Δ (tuples or bytes), each run contributes at most Δ error, so the total objective error per boundary is at most $N\Delta$, with Δ set by the memory limit. Compared with schemes that globally materialize and sort anchor keys before selecting split points (e.g., RocksDB subcompaction [61]), our method probes existing sparse indexes directly and avoids a separate materialization stage, which reduces memory overhead while still providing accurate boundary selection.

Sparse Index Page Recycling. Sparse-index entries use fixed-size 4 KiB pages; during merge, the input sparse-index pages are needed only for boundary computation and initial seek, after which we recycle them through a shared pool to seed output sparse indexes (with one barrier per merge step). If a thread exhausts its seeded pages, it falls back to heap allocation.

Offset Value Codes. CrocSort uses a 32-bit OVC format: the first 16 bits encode the index of the first differing byte, and the remaining 16 bits store a two-byte value from that position onward. When runs are written to disk, records are prefix-truncated using the offset stored in the OVC: bytes before the first differing position are elided, reducing run sizes. The sparse index stores full keys (not truncated keys), so OVC reconstruction can start from any sparse-index entry; merge outputs are also OVC-encoded while sparse-index-based boundary selection remains unchanged. We also support a non-OVC format for ablation experiments.

Scope and Integration. CrocSort is a standalone disk-to-disk sorter; embedding it as a backend for sort-merge join, aggregation, or ORDER BY would require a streaming tuple interface and a budget contract with the host engine's memory manager, which we leave as future work. The configuration-first model still applies once budgets are fixed. Re-planning when host-engine cardinality estimates are stale or cold is also future work, as is adaptation to changing budgets [10, 54, 55].

6 EVALUATION

All experiments were conducted on a server with dual-socket Intel Xeon Silver 4116 CPUs (12 cores per socket, 2.10 GHz, hyperthreading enabled) and 192 GB of RAM. The external storage is an Intel 760p NVMe SSD (340 kIOPS random reads, 275 kIOPS random writes, 4 KiB blocks). We use up to T=44 worker threads (out of 48 logical CPUs) to leave headroom for OS tasks. CrocSort supports both Direct I/O and buffered I/O; experiments use Direct I/O unless noted otherwise.

This section has two parts. The first (Items 1–5) presents CrocSort-only ablation studies on parallelism scaling, memory budget effects, OVC benefit, and partitioning policy; the observations are synthesized into practical configuration guidelines in Section 6.6. The second (Item 6) applies those guidelines and validates them in a cross-system end-to-end comparison under explicit memory limits and thread caps.

(1) **Thread Allocation and Merge-Step Tradeoffs** (Section 6.1). At a fixed, tight memory budget, across the thread-allocation grid in Figure 4c, how should threads be allocated between run generation and merge? How do increases in merge-step count affect performance and efficiency?

Table 3: Dataset Specifications

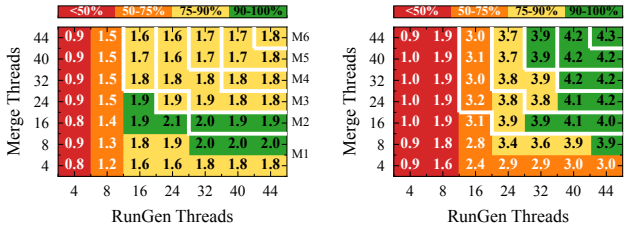
Dataset	Summary
GenSort [19]	**Size:** 200 GiB (≈ 2.15 B rows); **Key:** 10 B; **Payload:** 90 B Uniform random keys; Fixed-length records.
TPC-H [71] `lineitem`	**Size:** SF500, 200 GiB (≈ 3 B rows); **Avg. Key:** 49 B; **Avg. Payload:** 13 B High prefix redundancy keys; Variable-length records. **Key cols (card.):** `l_returnflag` (3), `l_linestatus` (2), `l_shipinstruct` (4), `l_shipmode` (7), `l_comment` (uniq.) **Payload cols:** `l_orderkey`, `l_linenumber`.
FreqKey	**Size:** 200 GiB (≈ 407 M rows); **Key:** 8 B; **Payload:** 512 B 50% heavy-hitter keys; Fixed-length records.
HeavyKey	**Size:** 200 GiB (≈ 63 M rows); **Key:** 8 B; **Payload:** 32 KiB / 128 B 10% heavy-hitter keys; Payload size skew (256×).

(2) **Memory Scaling** (Section 6.2). In a high-thread setting, how does end-to-end performance vary with memory budget, and where do returns diminish (or reverse)?
(3) **Offset-Value Coding** (Section 6.3). How does OVC affect end-to-end sorting time, especially when merge depth increases?
(4) **Partitioning Method** (Section 6.4). How do Key-Only, Count-Balanced, and Byte-Balanced range partitioning compare in terms of load balance and merge time under skew?
(5) **Budget and I/O Checks** (Section 6.5). Do measured memory usage and I/O volumes stay near configured budgets and qualitatively track the analytical expectations?
(6) **Cross-System Comparison and Planner Validation** (Section 6.7). Under explicit memory limits and a fixed-thread cap, how does CrocSort compare to existing systems in end-to-end sort time and robustness under tight memory? How does the planner-selected configuration perform relative to the symmetric threads and maximal memory configuration?

Each plotted point reports the mean of three completed runs. Table 3 summarizes datasets. We use ≈ 200 GiB GenSort and TPC-H SF500 `lineitem` for all experiments except the partitioning stress tests (Section 6.4), which use synthetic FreqKey and HeavyKey datasets. Unless noted otherwise, CrocSort ablations use M=2 GiB, Direct I/O, OVC, and Byte-Balanced partitioning; subsections note thread choices and other deviations. Except for the cross-system comparison in Section 6.7, CrocSort writes the final merged output to disk so timings reflect disk-to-disk sorting.

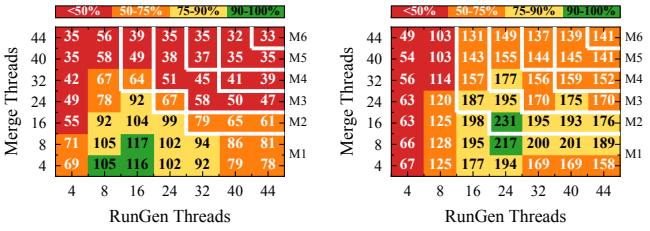
6.1 Thread Allocation and Merge-Step Tradeoffs

Under the default ablation memory budget (M=2 GiB), we independently vary T_{gen} and T_{merge} to produce throughput and throughput-per-CPU-second heatmaps (Figures 7 and 8), where throughput per CPU-second is wall-clock throughput divided by aggregate thread-seconds. Along the diagonal ($T_{\text{gen}} = T_{\text{merge}}$), GenSort throughput peaks at $T \approx 16$ and then degrades as shrinking per-thread buffers force multi-step merging, amplifying I/O and saturating SSD bandwidth. End-to-end throughput degrades gradually beyond this point, but throughput per CPU-second drops sharply once the schedule exceeds roughly three merge steps. `lineitem` tolerates higher thread counts more gracefully due to OVC's I/O reduction; Sections 6.3 and 6.5 detail how OVC reduces I/O for prefix-redundant keys, which in turn reduces multi-step merge penalties.



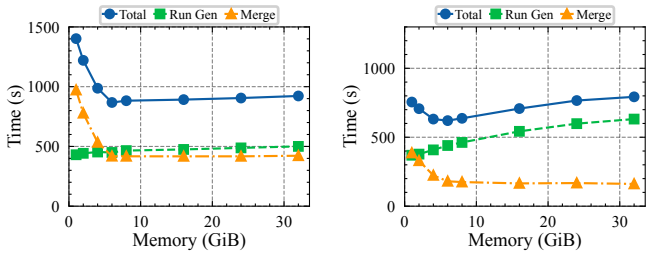
(a) GenSort (Synthetic) (b) `lineitem` (TPC-H SF500)

Figure 7: Throughput (M entries/sec) vs. Thread Allocation (T_{gen}×T_{merge}). — Color tiers show normalized throughput relative to the best configuration. The boundary marks transitions between observed K-step regions in the heatmap, and the right-side annotations indicate the corresponding merge-step counts.



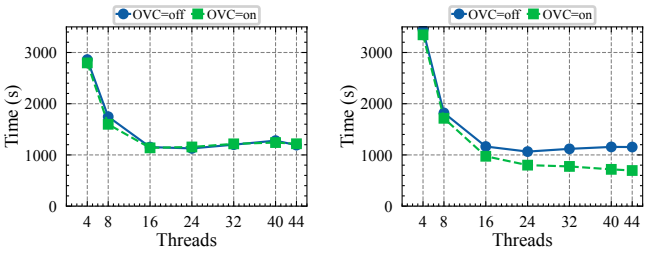
(a) GenSort (Synthetic) (b) `lineitem` (TPC-H SF500)

Figure 8: Throughput (K entries/sec) per CPU-second vs. Thread Allocation (T_{gen}×T_{merge}). — Color tiers show the normalized throughput-per-CPU-second score (throughput / aggregate CPU time) relative to the best configuration. The same boundary and right-side merge-step annotations are shown as in Figure 7.



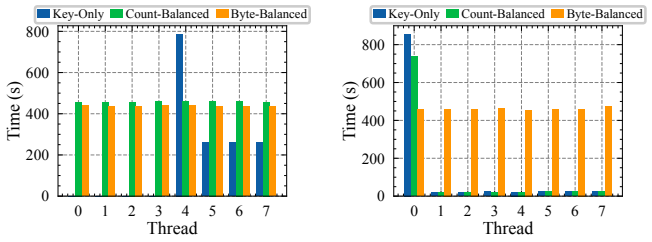
(a) GenSort (Synthetic) (b) `lineitem` (TPC-H SF500)

Figure 9: Memory-Scaling Threshold in CrocSort. — Sorting time vs. memory on ≈ 200 GiB input with fixed T=44; memory varies from 1 GiB to 32 GiB.



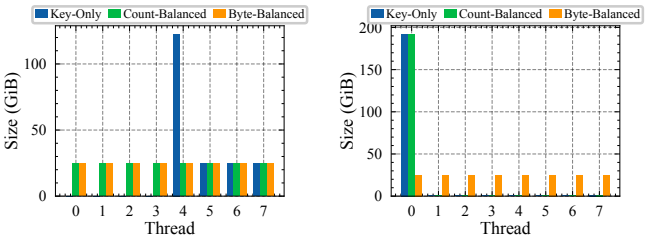
(a) GenSort (Synthetic) (b) `lineitem` (TPC-H SF500)

Figure 10: Impact of OVC on Total Sorting Time. — OVC vs. non-OVC on GenSort and `lineitem`.



(a) FreqKey (Synthetic) (b) HeavyKey (Synthetic)

Figure 11: Per-Thread Merge Time with 8 Merge Threads. — Comparison of partitioning policies on FreqKey and HeavyKey.



(a) FreqKey (Synthetic) (b) HeavyKey (Synthetic)

Figure 12: Per-Thread Partition Size (bytes) with 8 Merge Threads. — Comparison of partitioning policies on FreqKey and HeavyKey.

Decoupling T_{gen} and T_{merge} reveals that the high-throughput region is consistently larger when $T_{gen} > T_{merge}$, reflecting the phase asymmetry: run generation is comparatively CPU-intensive while merging is I/O-bound. For GenSort, adding merge threads beyond $T_{merge} \approx 16$ degrades efficiency; for `lineitem`, the optimal zone is broader. The practical takeaway is to provision threads where run generation still scales, keep merge parallelism below storage saturation, and target a single/few-step merge schedule.

6.2 Memory Scaling

We next study how end-to-end performance evolves under a high thread count as memory scales from tight to ample. We fix T=44 and sweep the memory budget from 1 GiB to 32 GiB; Figure 9 shows the resulting curves for both datasets and how the bottleneck shifts between merge and run generation as memory increases.

Run Generation vs. Merge Breakdown. At low memory, per-thread buffers shrink, the number of runs increases, and the merge schedule becomes deeper. Merge cost dominates because the sort performs more merge steps and moves more total bytes. As memory

increases from that regime, merge cost drops quickly because the number of runs decreases and the plan approaches a single/few-step merge. Beyond this *threshold*—the point where adding memory no longer reduces merge cost but begins to increase run-generation comparison work—run generation becomes the bottleneck. Merge keeps improving only slowly, but run-generation comparison work increases because the run-generation tree of losers operates over *records* in larger runs, while the merge tree of losers operates over *pages*. Thus, beyond the threshold, extra memory can increase run-generation cost more than it reduces merge cost. The effect is stronger on `lineitem`, where longer, prefix-heavy keys make run-generation comparisons more expensive [52]. Variable-length records may also increase run-generation allocator overhead: irregular slot reuse in the tree of losers can make it harder to place the next record as runs grow. This memory-scaling sweep is measured without merge prefetching/forecasting; those mechanisms may shift the crossover. These results suggest a practical memory-scaling rule: avoid over-allocating memory once the target single/few-step merge regime is reached.

6.3 OVC Impact on End-to-End Sorting

To isolate OVC's impact, we run the symmetric configuration ($T_{\text{gen}} = T_{\text{merge}} = T$) with OVC disabled, using standard replacement selection and comparisons in the tree of losers (Figure 10). OVC has little impact on GenSort, where key-prefix redundancy is low. For `lineitem`, OVC yields a substantial reduction in sorting time, especially at high thread counts: more threads mean shorter runs and more merge steps, raising merge cost—and OVC reduces that cost by compressing intermediate I/O and accelerating comparisons. The I/O reduction for prefix-redundant workloads is quantified in Section 6.5. OVC therefore mitigates multi-step merge penalties on prefix-redundant workloads, especially under tight memory and high thread counts.

6.4 Partitioning Methods

Earlier CrocSort experiments used Byte-Balanced partitioning by default. Here we compare the three boundary-selection policies from Section 3 and focus on their empirical impact on load balance and merge time. We evaluate on the FreqKey and HeavyKey datasets with 8 merge threads (Figures 11 and 12), each designed to isolate specific failure modes. On FreqKey (duplicate skew only, uniform 512 B records), Key-Only concentrates the heavy-hitter key (50% of rows) onto a single thread, causing it to process $\approx 5\times$ the load of other threads. Count-Balanced and Byte-Balanced both split the heavy-hitter and achieve balanced partitions; because all records are the same size, both behave identically.

On HeavyKey (duplicate + size skew), Key-Only again concentrates the heavy-hitter on one thread; Count-Balanced equalizes counts but not bytes (32 KiB heavy payloads vs. 128 B normal); Byte-Balanced produces the most even byte distribution and merge times in this stress test. Additional stress checks with multimodal keys and extreme duplicate skew (> 99.9% identical keys) showed the same qualitative balance. Merge time is not strictly proportional to I/O volume because stragglers running after others finish enjoy higher per-thread bandwidth—an effect that diminishes at lower thread counts where each thread already has a larger share. The takeaway: Byte-Balanced is the safest default under variable-size records or high merge parallelism.

6.5 Budget and I/O Validation

Memory Budget Adherence. Figure 13 shows the memory consumption over time for GenSort and `lineitem` under a high-thread stress setting (M=2 GiB, T_{gen}=T_{merge}=44). Memory is measured by polling the process's resident set size (RSS) at one-second intervals throughout execution. Despite the high thread count—which maximizes the number of merge steps—the system keeps memory close to the 2 GiB budget throughout both phases, with only minor overshoot. During run generation, memory ramps up quickly and stays near the budget ceiling; at the merge transition, run-generation buffers are reclaimed and reassigned to merge I/O buffers, and usage remains stable across all merge steps (M1–M6).

I/O Volume Breakdown. Figure 14 shows the I/O volume breakdown across run generation and merge phases for both datasets in the symmetric-thread setup ($T_{\text{gen}} = T_{\text{merge}} = T$) at the default ablation memory budget (M=2 GiB). Each stacked bar decomposes the total bytes transferred into reads and writes for run generation and each merge step. For GenSort, measured I/O closely follows the trend predicted by our Huffman-merge analysis: beyond ≈ 16 threads, shrinking per-thread buffers create more runs, which raises the average merge level L_{avg} and therefore total bytes moved. `lineitem` deviates more because OVC compresses intermediate bytes: merge steps still increase, but each step moves less physical data (Section 6.3).

6.6 Empirical Synthesis and Guidelines

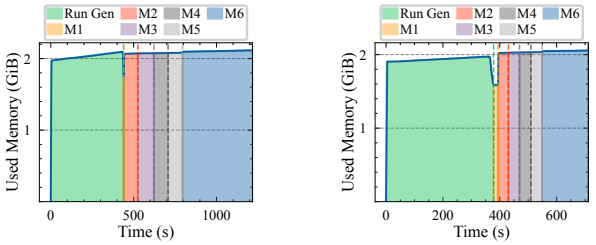
Synthesizing the CrocSort-only characterization in Sections 6.1–6.5, we summarize four empirical observations and combine them with the analytical feasibility constraints to formulate practical resource-configuration rules for CrocSort.

Key Takeaways. We distill four observations:

(1) *Favor $T_{gen} > T_{merge}$ once merge saturates, and target a single/few-step merge regime.* The heatmaps (Figures 7–8) show that the best throughput and throughput-per-CPU-second regions cluster in this regime. On GenSort, two-step merges often remain competitive, but throughput per CPU-second drops sharply beyond about three merge steps; on `lineitem`, prefix redundancy and OVC yield more graceful degradation, so deeper schedules can remain acceptable. Once merge saturates, symmetric allocation ($T_{\text{gen}} = T_{\text{merge}}$) tends to overspend merge threads with little end-to-end benefit.

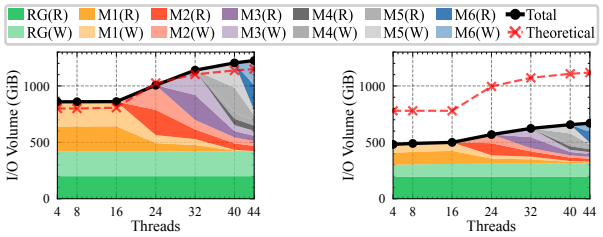
(2) *Memory budget has a two-sided threshold.* Figure 9 shows that below this threshold, too few per-thread buffers force deep merge schedules with high I/O, while above it, run-generation comparison cost rises as trees operate over larger runs. The practical target is a moderate budget near this threshold that keeps the sort in a single or few merge steps.

(3) *OVC acts as a robustness cushion for multi-step merges.* By truncating shared key prefixes, OVC compresses intermediate run data and speeds up comparisons. In our experiments, the performance gap between OVC and non-OVC widens as merge depth increases, meaning OVC helps absorb the cost of deeper



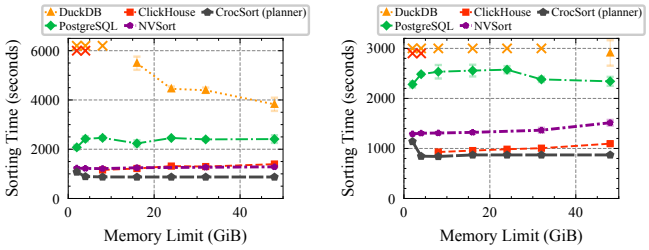
(a) GenSort (Synthetic) (b) `lineitem` (TPC-H SF500)

Figure 13: Memory Consumption Over Time. — Memory usage during external sort with 2 GiB budget and 44 threads. Shaded regions indicate sorting phases: run generation followed by merge steps M1–M6 (individual merge operations in the Huffman schedule, not full-data merge passes).



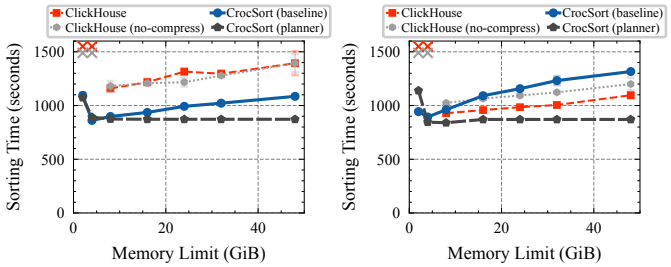
(a) GenSort (Synthetic) (b) `lineitem` (TPC-H SF500)

Figure 14: I/O Volume Breakdown. — Total bytes read and written during run generation and each merge step. $M = 2$ GiB, $T_{\text{gen}} = T_{\text{merge}} = T$.



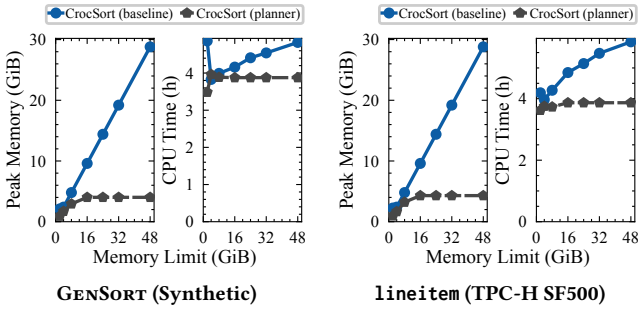
GenSort (Synthetic) `lineitem` (TPC-H SF500)

Figure 15: End-to-End Sort Time Across Systems. — ≈ 200 GiB inputs, fixed 16-thread configuration, varying memory limits. X denotes OOM runs.



GenSort (Synthetic) `lineitem` (TPC-H SF500)

Figure 16: CrocSort: Planner vs. Baseline. — Sort time for CrocSort (planner) versus CrocSort (baseline: symmetric T_{gen}=T_{merge}=16 with full memory allocation), with ClickHouse references. ≈ 200 GiB inputs, varying memory limits.



GenSort (Synthetic) `lineitem` (TPC-H SF500)

Figure 17: CrocSort: Planner vs. Baseline (Resource Usage). — Peak Memory (GiB) and CPU usage (hours) under varying memory budgets. ≈ 200 GiB inputs, T_{max}=16.

schedules. For random-key workloads (GenSort), prefix sharing is rare and OVC provides negligible benefit (Figure 10).

(4) *Byte-balanced partitioning improves load balance under skew.* Count-balanced partitioning equalizes tuple counts but ignores payload-size variation, leaving threads with heavy-payload keys processing disproportionate I/O (Figures 11 and 12). Byte balancing substantially improves partition-size balance and reduces merge-time stragglers.

Practical Resource Configuration. Combining the analysis and measurements above, we derive practical configuration rules for high performance under constrained resources. Finding the globally optimal $(T_{\text{gen}}, T_{\text{merge}}, M)$ configuration is an open problem: it depends on workload-specific I/O behavior, data properties, and hardware characteristics that are difficult to model exactly. Rather than searching for a global optimum, we aim to avoid configurations that overspend memory or merge threads without improving throughput relative to the baseline of symmetric threads and maximal memory. We keep OVC and Byte-Balanced partitioning as defaults, as they consistently reduce I/O and improve efficiency across workloads. Given parameters $(D, M, T_{\text{max}}, P, \rho, E)$, the goal is to stay in a single/few-step merge regime (K=1 preferred) without over-provisioning resources. We check feasibility via Eq. 6 and apply the following rules by regime:

- *Memory-bound* ($T_{\text{max}}^2 > \frac{E\rho^2 M^2}{DP}$, Eq. 6): using all threads symmetrically is not single-step feasible. Reduce T_{merge} first (merge is I/O-bound; fewer merge threads reduces buffer demand) until Eq. 5 is satisfied, then set T_{gen} to the largest value that remains

feasible. This avoids wasted CPU from over-parallelized multi-step schedules.

- *Thread-bound* ($T_{\max}^2 \leq \frac{E\rho^2 M^2}{DP}$, Eq. 6): the symmetric setting T_{gen}=T_{merge}=$T_{\max}$ is already single-step feasible. Start there, and cap T_{merge} if storage saturation is observed. Per-thread run-generation buffers should be kept in a moderate range: even if single-step merge is theoretically feasible, buffers below ≈40 MB produce tiny runs that cause I/O fragmentation and degrade sparse-index quality in practice—so we enforce this as a lower bound. Too large, and tree comparison cost during run generation rises (Section 6.2).

We use E=1 for conservative planning. If K=1 is infeasible at the chosen thread counts, increase K to the smallest value that restores feasibility. Note that these rules assume no prior knowledge of the data beyond its size—in particular, they do not account for data properties such as prefix redundancy. For workloads like `lineitem`, OVC's robustness cushion may allow deeper merge schedules to remain practical even when the analytical model suggests otherwise.

6.7 Cross-System Comparison and Planner Validation

Setup. We compare CrocSort against PostgreSQL 18.1, DuckDB 1.4.4, ClickHouse 25.11.8, and an NVMe-specialized NVSort baseline [51] re-implemented inside CrocSort (pre-sort sampling-based range partitioning, rw-lock write arbitration, adaptive block sizing) since the original code is unavailable, holding the storage stack constant. Each run executes under cgroup v2 hard memory limits (`memory.high`/`max` at the budget, swap disabled). To reduce OOM risk we also apply a 60% engine-level soft limit (DuckDB `memory_limit`, ClickHouse `max_bytes_before_external_sort`, PostgreSQL split across workers and leader). Threads are fixed to 16 (PostgreSQL: 15 workers plus leader); CrocSort's planner selects (T_{gen}, T_{merge}, M) within the same caps ($T_{\max}$=16, memory budget equal to the 60% soft limit). Datasets are preloaded as native tables with binary `sort_key`/`payload` columns; all systems run `SELECT sort_key, payload FROM t ORDER BY sort_key` with output suppressed (`EXPLAIN ANALYZE` for PostgreSQL/DuckDB, `FORMAT Null` for ClickHouse). Runs that OOM are shown as `X`. CrocSort uses Direct I/O; buffered I/O is up to 10% faster but omitted, making the comparison slightly conservative since other engines use buffered I/O.

Figure 16 isolates the effect of the planner by comparing two configurations under the same caps ($T_{\max}$=16, memory budget equal to the 60% soft limit). *CrocSort Baseline* is the greedy starting point: it sets T_{gen}=T_{merge}=16 and allocates the entire soft-limit budget to the run-generation working set. *CrocSort Planner* applies the rules from Section 6.6 within those same caps. The gap between the two lines therefore reflects only planner-driven configuration choices, with all other factors held constant. ClickHouse is shown with and without base-table compression; on `lineitem`, default ClickHouse compression reduces table size to ≈ 60 GiB, while GenSort is unaffected due to high entropy and remains at ≈ 200 GiB. We include the no-compression variant to separate sorting behavior from base-table size reduction and make runtime differences easier to interpret.

Observed Results. Figure 15 reports absolute sort time. Under the cgroup-constrained, OOM-mitigated settings, CrocSort is comparatively robust in the low-memory regime, completing runs at budgets where DuckDB and ClickHouse OOM. PostgreSQL is more robust to OOM but much slower (about 2× slower than CrocSort). Across completed runs in these settings, planner-configured CrocSort is competitive on both datasets. NVSort completes across budgets but is slower than planner-configured CrocSort. The gap is mainly in run formation: NVSort's rw-lock serializes phase-aligned writes, whereas CrocSort's replacement selection interleaves reads and writes continuously across workers. NVSort also lacks duplicate-key tie-breakers and does not plan for multi-step merge cost under tight memory.

In Figure 16, the no-compression ClickHouse is about 100 seconds slower than default ClickHouse on `lineitem` because the table is larger without compression, but performs similarly on GenSort. Two regimes appear. At tighter budgets, the planner reduces merge threads to avoid over-parallelized merge work. This keeps GenSort in a single-step regime and improves both wall-clock time and CPU-seconds; on `lineitem`, where OVC already reduces merge I/O, it primarily improves CPU efficiency and can trade a slight wall-clock regression for lower CPU-seconds. At larger memory budgets, once the single-step regime is reached, the planner right-sizes run-generation memory (while respecting the 40 MB per-thread minimum) to avoid degradation from over-allocation.

Figure 17 shows peak memory and total CPU time for the same two configurations. Under tight memory, the planner consistently uses fewer CPU resources by reducing merge-thread parallelism; under ample memory, it right-sizes the run-generation working set below the full budget, reducing peak memory consumption.

Implications for the Policy and Future Work. These results support the planner's two main rules: under tight memory, reduce merge-thread parallelism to maintain single/few-step schedules and avoid unnecessary CPU work; once the single-step threshold is reached, right-size the run-generation working set near that threshold rather than consuming the full budget. As future work, we plan to explicitly model dataset characteristics and hardware characteristics to further improve these policy decisions.

7 CONCLUSION

CrocSort combines a budgeted external-sort design with sparse-index-based range partitioning and an analytical/empirical policy for choosing memory and phase-specific parallelism. The main rule is to target single/few-step merge plans, favor run-generation threads once merge saturates, and right-size memory rather than maximize it. Experiments show stable memory adherence, byte-balanced sparse-index merging under skew, OVC benefits on prefix-redundant data, and greater robustness than the evaluated production systems under tight budgets.

ACKNOWLEDGMENTS

This work was supported in part by NSF Award IIS-2048088 and a Google Data Analytics and Insights (DANI) Award. Riki Otaki is supported in part by a MongoDB PhD Fellowship. We thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. 1997. A One-Pass Algorithm for Accurately Estimating Quantiles for Disk-Resident Data. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. 346–355.

[2] Nancy Amato, Ravishankar Iyer, Sharad Sundaresan, and Yan Wu. 1998. *A Comparison of Parallel Sorting Algorithms on Different Architectures*. Technical Report. Texas A & M University.

[3] Jean-Loup Baer and Yi-Bing Lin. 1989. Improving Quicksort Performance with a Codeword Data Structure. *IEEE Transactions on Software Engineering* 15, 5 (1989), 622–631.

[4] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2023. WiscSort: External Sorting for Byte-Addressable Storage. *Proc. VLDB Endow.* 16, 9 (May 2023), 2103–2116.

[5] Mohammed Bey Ahmed Khernache, Arezki Laga, and Jalil Boukhobza. 2018. MONTRES-NVM: An External Sorting Algorithm for Hybrid Memory. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 49–54.

[6] Timo Bingmann, Peter Sanders, and Matthias Schimek. 2020. Communication-Efficient String Sorting. arXiv:2001.08516 [cs.DC] https://arxiv.org/abs/2001.08516

[7] Mark Callaghan. 2026. Small Datum: Common Prefix Skipping, Adaptive Sort. https://smalldatum.blogspot.com/2026/01/common-prefix-skipping-adaptive-sort.html

[8] Yannis Chronis, Thanh Do, Goetz Graefe, and Keith Peters. 2020. External Merge Sort for Top-k Queries: Eager Input Filtering Guided by Histograms. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2423–2437.

[9] Graham Cormode and Pavel Veselý. 2020. A Tight Lower Bound for Comparison-Based Quantile Summaries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 81–93.

[10] Diane L. Davison and Goetz Graefe. 1994. Memory-Contention Responsive Hash Joins. In *VLDB*. 379–390.

[11] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. 280–291.

[12] Thanh Do and Goetz Graefe. 2023. Robust and Efficient Sorting with Offset-Value Coding. *ACM Transactions on Database Systems* 48, 1, Article 2 (March 2023).

[13] Thanh Do, Goetz Graefe, and Jeffrey Naughton. 2023. Efficient Sorting, Duplicate Removal, Grouping, and Aggregation. *ACM Transactions on Database Systems* 47, 4, Article 16 (Jan. 2023).

[14] Bin Dong, Surendra Byna, and Kesheng Wu. 2016. SDS-sort: Scalable Dynamic Skew-Aware Parallel Sorting. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 57–68.

[15] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Transactions on Storage* 17, 4, Article 26 (Oct. 2021).

[16] Pavlos S. Efraimidis and Paul G. Spirakis. 2006. Weighted Random Sampling with a Reservoir. *Inform. Process. Lett.* 97, 5 (2006), 181–185.

[17] Dina Bitton Friedland. 1981. *Design, Analysis, and Implementation of Parallel External Sorting Algorithms*. Ph. D. Dissertation. The University of Wisconsin - Madison.

[18] Edward H. Friend. 1956. Sorting on Electronic Computer Systems. *Journal of The Acm* 3, 3 (July 1956), 134–168.

[19] Gensort [n. d.]. Sort Benchmark Data Generator and Output Validator. https://www.ordinal.com/gensort.html

[20] Martin A. Goetz. 1963. Internal and Tape Sorting Using the Replacement-Selection Technique. *Communications of The Acm* 6, 5 (May 1963), 201–206.

[21] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *Acm Computing Surveys* 25, 2 (June 1993), 73–169.

[22] Goetz Graefe. 2006. Implementing Sorting in Database Systems. *ACM Comput. Surv.* 38, 3 (Sept. 2006), 10–es.

[23] Goetz Graefe. 2023. 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*, Vol. P-331. 27–46.

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

[25] Goetz Graefe and Thanh Do. 2022. Offset-Value Coding in Database Query Processing. In *International Conference on Extending Database Technology*.

[26] Goetz Graefe, Marius Kuhrt, and Bernhard Seeger. 2025. Modifying an Existing Sort Order with Offset-Value Codes. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*. 246–254.

[27] G. Graefe, A. Linville, and L. D. Shapiro. 1994. Sort vs. Hash Revisited. *IEEE Trans. on Knowl. and Data Eng.* 6, 6 (Dec. 1994), 934–944.

[28] Goetz Graefe, Soumya Sistla, and Deepa Shree Chickballapur Venkatachalapathi. 2025. Offset-Value Coding in Database Query Execution: A Technology Survey and a Measurement Study with New Conclusions. *ACM Transactions on Database Systems* (Nov. 2025).

[29] Michael Greenwald and Sanjeev Khanna. 2001. Space-Efficient Online Computation of Quantile Summaries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 58–66.

[30] George U. Hubbard. 1963. Some Characteristics of Sorting Computing Systems Using Random Access Storage Devices. *Communications of The Acm* 6, 5 (May 1963), 248–255.

[31] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[32] B. Iyer and D.M. Dias. 1990. System Issues in Parallel Sorting for Database Systems. In *[1990] Proceedings. Sixth International Conference on Data Engineering*. 246–255.

[33] Jin Jiang, Dongsheng He, Yu Hu, Dong Liu, Chenfan Xiao, Hongxiao Bi, Yusong Zhang, Chaoqu Jiang, and Zhijun Fu. 2024. CompassDB: Pioneering High-Performance Key-Value Store with Perfect Hash. arXiv:2406.18099 [cs.DB] https://arxiv.org/abs/2406.18099

[34] Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R. Nutter, and Jeremy D. Schaub. 2016. Tencent Sort. https://sortbenchmark.org/TencentSort2016.pdf

[35] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 71–78.

[36] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*.

[37] Ani Kristo and Tim Kraska. 2023. Parallel External Sorting of ASCII Records Using Learned Models. arXiv:2305.05671 [cs.DB] https://arxiv.org/abs/2305.05671

[38] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1001–1016.

[39] Laurens Kuiper. 2021. Fastest Table Sort in the West – Redesigning DuckDB's Sort. https://duckdb.org/2021/08/27/external-sorting.html

[40] Laurens Kuiper. 2025. Redesigning DuckDB's Sort, Again. https://duckdb.org/2025/09/24/sorting-again.html

[41] Laurens Kuiper and Hannes Mühleisen. 2023. These Rows Are Made for Sorting and That's Just What We'll Do. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2050–2062.

[42] Sai Choi Kwan and Jean-Loup Baer. 1985. The I/O Performance of Multiway Mergesort and Tag Sort. *IEEE Trans. Comput.* C-34, 4 (1985), 383–387.

[43] Arezki Laga, Jalil Boukhobza, Frank Singhoff, and Michel Koskas. 2017. MONTRES : Merge on-the-Run External Sorting Algorithm for Large Data Volumes on SSD Based Storage Systems. *IEEE Trans. Comput.* 66, 10 (2017), 1689–1702.

[44] P.A. Larson. 2003. External Sorting: Run Formation Revisited. *IEEE Transactions on Knowledge and Data Engineering* 15, 4 (2003), 961–972.

[45] Per-Åke Larson and Goetz Graefe. 1998. Memory Management during Run Generation in External Sorting. In *SIGMOD*. 472–483.

[46] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*. 743–754.

[47] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (May 2021), 1606–1612.

[48] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, SangBin Cho, Eric Liang, and Ion Stoica. 2023. ExoshuffleCloudSort. https://arxiv.org/abs/2301.03734

[49] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. 1998. Approximate Medians and Other Quantiles in One Pass and with Limited Memory. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. 426–435.

[50] J. Ian Munro and Sebastian Wild. 2018. Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, Vol. 112. 63:1–63:16.

[51] Kihyeon Myung, Sunggon Kim, Heon Young Yeom, and Jiwoong Park. 2021. Efficient and Scalable External Sort Framework for Nvme SSD. *IEEE Trans. Comput.* 70, 12 (Dec. 2021), 2211–2217.

[52] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1995. AlphaSort: A Cache-Sensitive Parallel External Sort. *The VLDB Journal* 4, 4 (Oct. 1995), 603–628.

[53] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. 2012. Merge Path - Parallel Merging Made Simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 1611–1618.

[54] Riki Otaki, Jun Hyuk Chang, Charles Benello, Aaron J. Elmore, and Goetz Graefe. 2025. Resource-Adaptive Query Execution with Paged Memory Management. In *CIDR*.

[55] HweeHwa Pang, Michael J. Carey, and Miron Livny. 1993. Memory-Adaptive External Sorting. In *VLDB*. 618–629.

[56] Danila Piatov, Sven Helmer, Anton Dignös, and Fabio Persia. 2021. Cache-Efficient Sweeping-Based Interval Joins for Extended Allen Relation Predicates. *The VLDB Journal* 30, 3 (May 2021), 379–402.

[57] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 755–766.

[58] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. 193–206.

[59] Mirko Rahn, Peter Sanders, and Johannes Singler. 2010. Scalable Distributed-Memory External Sorting. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 685–688.

[60] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. 2013. TritonSort: A Balanced and Energy-Efficient Large-Scale Sorting System. *ACM Trans. Comput. Syst.* 31, 1 (Feb. 2013).

[61] RocksDB [n. d.]. Subcompaction. https://github.com/facebook/rocksdb/wiki/Subcompaction

[62] Kenneth A. Ross and John Cieslewicz. 2009. Optimal Splitters for Database Partitioning with Size Bounds. In *Proceedings of the 12th International Conference on Database Theory*. 98–110.

[63] Betty Salzberg. 1989. Merging Sorted Runs Using Large Main Memory. *Acta Informatica* 27, 3 (Jan. 1989), 195–215.

[64] Betty Salzberg, Alex Tsukerman, Jim Gray, Michael Stuewart, Susan Uren, and Bonnie Vaughan. 1990. FastSort: A Distributed Single-Input Single-Output External Sort. In *SIGMOD*. 94–101.

[65] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3731–3744.

[66] Mehul A Shah and Chris Nyberg. 2014. CloudSort: A TCO Sort Benchmark. https://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf

[67] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. 2004. Medians and beyond: New Aggregation Techniques for Sensor Networks. In *ACM International Conference on Embedded Networked Sensor Systems*.

[68] A. Silberschatz, H.F. Korth, and S. Sudarshan. 2020. *Database System Concepts*.

[69] D. Taniar and Wenny Rahayu. 2000. Sorting in Parallel Database Systems. In *HPC Asia*. 830–835 vol.2.

[70] The PostgreSQL Global Development Group. 2024. PostgreSQL. https://www.postgresql.org

[71] TPCH 1999. TPC-H Homepage. https://www.tpc.org/tpch/

[72] linux unixsort. [n. d.]. Sort(1) - Linux Manual Page. https://man7.org/linux/man-pages/man1/sort.1.html

[73] Milan Vojnovic, Fei Xu, and Jingren Zhou. 2012. *Sampling Based Range Partition Methods for Big Data Analytics*. Technical Report. Microsoft.

[74] Qian Wang, Rong Gu, Yihua Huang, Reynold Xin, Wei Wu, Jun Song, and Junluan Xia. 2016. NADSort. https://sortbenchmark.org/NADSort2016.pdf

[75] Richard Wesley. 2023. DuckDB's AsOf Joins: Fuzzy Temporal Lookups. https://duckdb.org/2023/09/15/asof-joins-fuzzy-temporal-lookups.html

[76] Weiye Zhang and Per-Åke Larson. 1997. Dynamic Memory Adjustment for External Mergesort. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. 376–385.

[77] Fuheng Zhao, Divyakant Agrawal, A. El Abbadi, and Ahmed Metwally. 2021. SpaceSaving± an Optimal Algorithm for Frequency Estimation and Frequent Items in the Bounded Deletion Model. *Proc. VLDB Endow.* 15 (2021), 1215–1227.

[78] Fuheng Zhao, Sujaya Maiyya, Ryan Weiner, Divy Agrawal, and Amr El Abbadi. 2021. KLL±: Approximate Quantile Sketches over Dynamic Datasets. *Proc. VLDB Endow.* 14, 7 (2021), 1215–1227.

[79] LuoQuan Zheng and P.-A. Larson. 1996. Speeding up External Mergesort. *IEEE Transactions on Knowledge and Data Engineering* 8, 2 (1996), 322–332.