Dissecting the Duck's Innards

④

Sorting Large Tables

Winter 2025/26

Torsten Grust
Universität Tübingen, Germany

1 ¦ Sorting is a Core Operation

The tabular model is based on unordered bags of rows, but the **sorting** of (intermediate) results of SQL queries is material:

1. **ORDER BY** clause,[1]
2. ordered aggregates (list(e_1 **ORDER BY** e_2)),
3. window functions (row_number() **OVER** (**ORDER BY** e)),
4. joins over time series data (**ASOF JOIN**), or
5. to ensure deterministic **LIMIT** and **OFFSET** behavior.

```
SELECT *                -- wide payload (here: reorder all columns)
FROM   lineitem
ORDER BY l_shipdate DESC NULLS FIRST  ,  l_quantity;
--                  ⏟    ⏟             ↑
--         defaults: ASC  NULLS LAST   lexicographic ordering
```

[1] In DuckDB, can adapt ordering defaults via SET default_order = {ASC,DESC} and SET default_null_order = {NULLS_FIRST,NULLS_LAST,NULLS_FIRST_ON_ASC_LAST_ON_DESC,NULLS_LAST_ON_ASC_FIRST_ON_DESC}.

Sorting (Large) Tables in DuckDB

To this day,[2] DuckDB's internal sorting routines are on the workbench 🛠 and still undergo active development.

Key aspects:

- **Efficient row comparison** that respects column types, directions (↑↓), handles NULLs, allows wide keys (lexicographic ordering).
- Adaptation to **pre-sorted input data.**
- Sorting strategy can process **tables larger than main memory,** spilling to disk 🖴 if required.
    - Effectively, sorting needs to materialize its entire input table (last input row could be the first in sort order).
- Balanced **use of all available CPU cores** during sorting.

[2] The last major rewrite of DuckDB's sorting code has shipped with version 1.4.0 in September 2025.

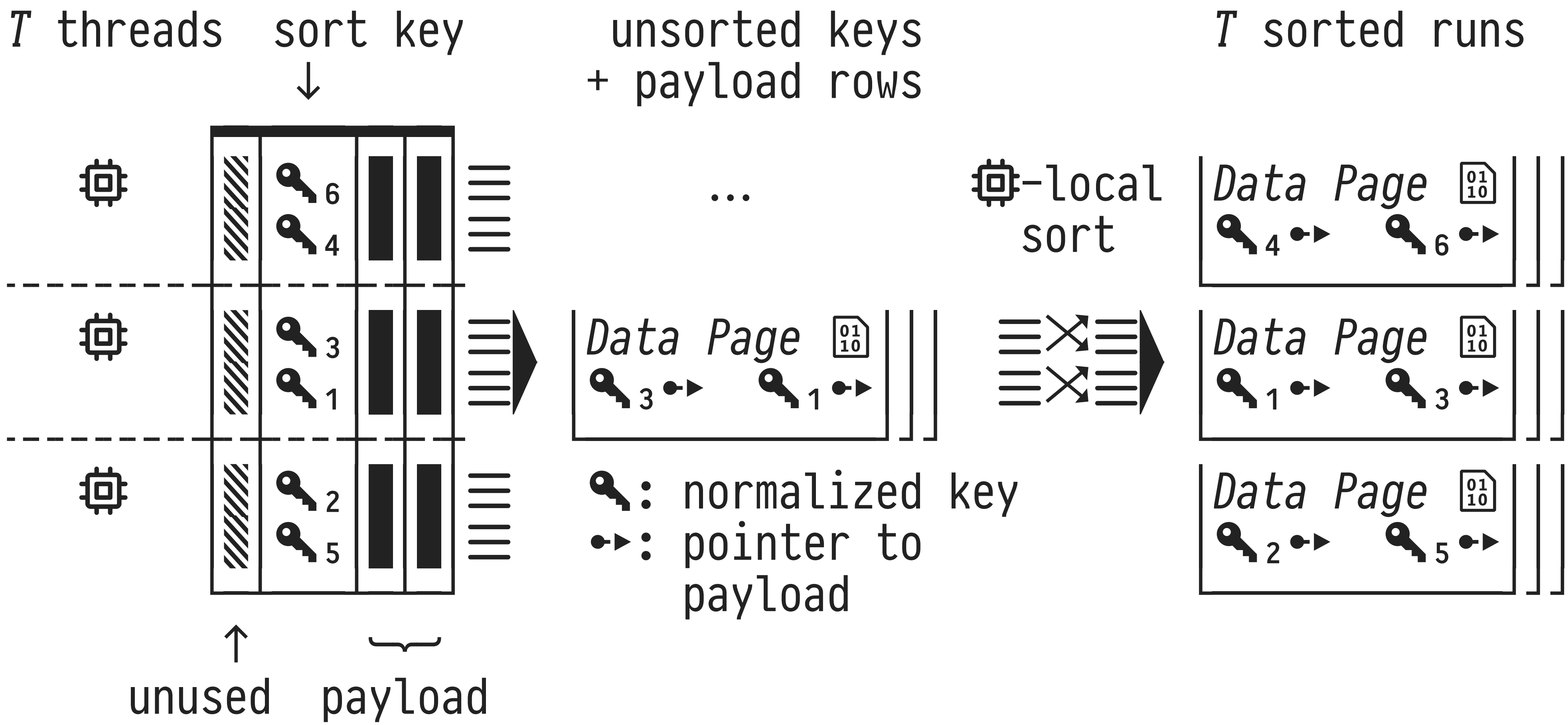
2 ┆ DuckDB: Two-Phase Merge Sort

DuckDB implements a variant of a **two-phase merge sort** strategy that utilizes all available CPU cores (say T) in both phases:

- **Phase ❶ (thread-local sorting):**
  - Each : read $1/T$ fraction of input, gather relevant columns to form rows on data pages, **normalize sort keys**,
  - then generate a **sorted run** of keys.

- **Phase ❷ (T-way merge of the sorted runs):**
  - Precompute boundaries of T non-overlapping **segments** in each run that can be **merged independently** by the s.
  - Each immediately outputs its merged run segments for consumption by the downstream query plan.

Phase ❶: Thread-Local Sorting



Phase ❶: 🔧 Key Normalization

- Sorting compares keys to decide row order ⤮. These comparison are frequent, but can be complex and costly:

```
⋮
ORDER BY c₁ ASC NULLS LAST, c₂ DESC NULLS FIRST
```

  - Dispatch on type of c_i to select proper < operator, respect sort order (↑↓), implement lexciographic ordering (column c_1 dominates), proper NULL treatment.

- 💡 **Key normalization:** Map[3] values in columns c_i to a byte sequence such that a *single < comparison* on two sequences can decide row order. #014

[3] DuckDB's key normalization is available at the user level in terms of scalar SQL function create_sort_key().

Phase ❶: Key Normalization for Fixed-Size Keys

CPUs compare fixed-size types (e.g., 32/64-bit ints) faster than variable-length byte sequences.

- 💡 If sequence length is known to be n bytes, store it in groups of $\lceil n/8 \rceil$ 64-bit ints. Directly compare these integers:[4]

```
struct FixedSortKey {
  uint64_t       part0;   // assume 8 < n ⩽ 16: group size = 2
  uint64_t       part1;
  sort_key_ptr_t payload; // (only if query has payload)
};

bool LessThan(const FixedSortKey &k1, const FixedSortKey &k2) {
  return k1.part0 < k2.part0 ||
         (k1.part0 == k2.part0 && k1.part1 < k2.part1);
}
```

[4] If SQL queries have no payload (SELECT c_1,c_2 FROM … ORDER BY c_1,c_2), DuckDB only keeps the normalized keys in fields part0, …: normalization can be inverted before output is generated.

Phase ❶: Sorting Algorithms

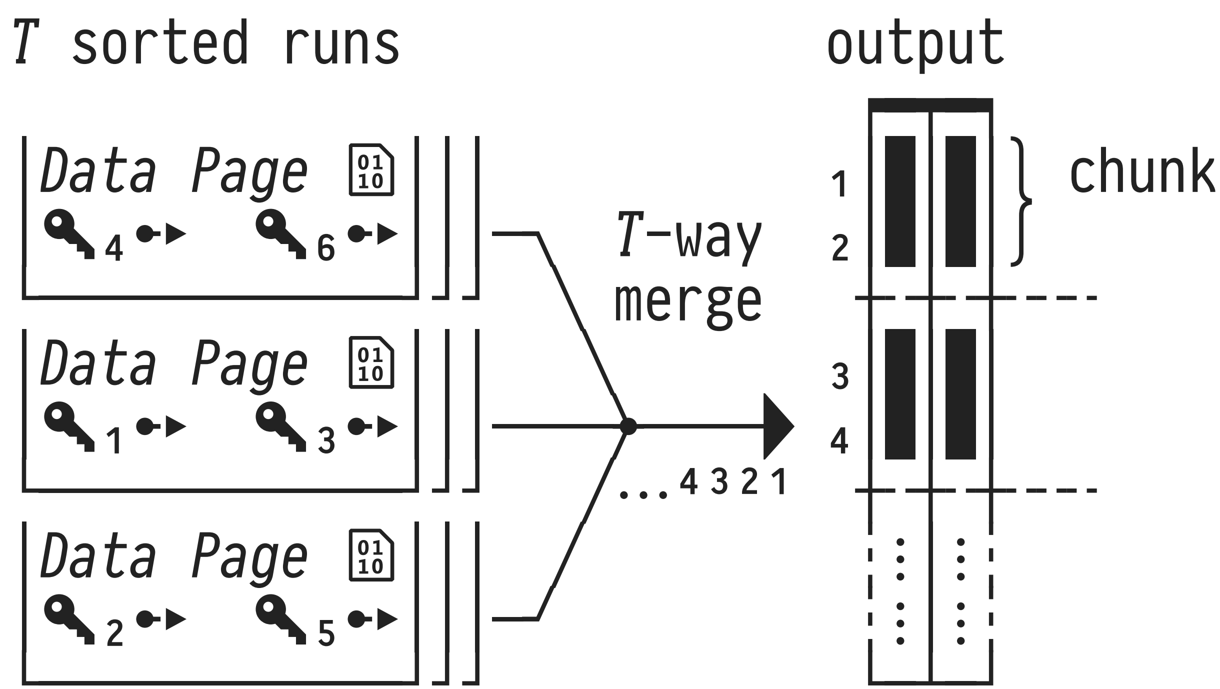
Typed keys (that can be compared via `<`) fit off-the-shelf sort implementations that assume the C++ `std::iterator` interface.

- DuckDB implements `std::iterator` for chains of 256KB data pages: → → ⋯ →, maps element indices to (page#, offset) pairs.
  - (A single 256KB data page would only be able to hold about 10,000 24-byte `FixedSortKey` structs. DuckDB aims to generate runs much longer than that.)

- DuckDB thus is able to adapt and combine three existing sorting algorithms:
  1. *Vergesort* (detects runs of already sorted data).
  2. *Ska Sort* (radix sort on the first 64 bits of the key).
  3. *Pattern-defeating QuickSort* (fallback, if the first 64 bits do not already order the data).

Phase ❷: *T*-Way Merge

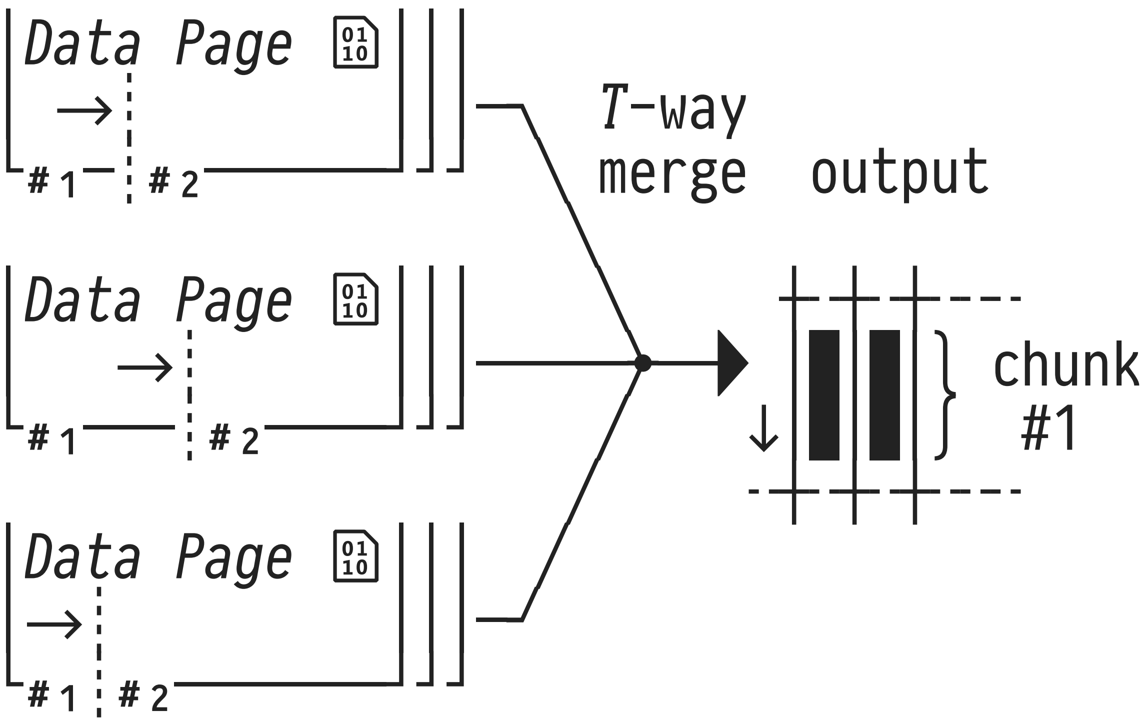
- Consume the *T* sorted runs produced by Phase ❶:
  - **Merge** pairs (🔑$_k$, •▸) using $<$ on key 🔑$_k$,
  - dereference pointers •▸ to access payload,
  - emit sorted output columns chunk-by-chunk:



- How can all CPU cores contribute equally during this merge?

Phase ❷: Parallelizing T-Way Merge

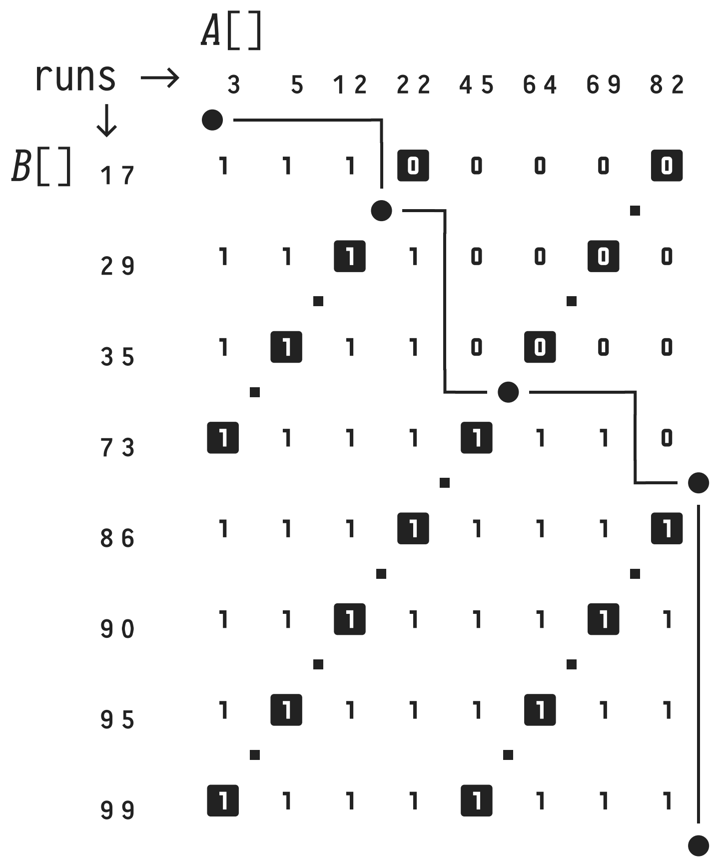
T sorted runs



- Runs contribute rows in **segment** #1 (up to ⋮) to output chunk #1.
- Run segments #*i* hold the rows for chunk #*i*. No overlap between segments.
- 💡 Precompute segment boundaries ⋮. Threads work on one segment.

- Once run segments #*i* are merged, the thread *immediately* emits chunk #*i*. The downstream plan can assemble the sorted output based on indices #*i* (DuckDB lingo: **batch indices**).

Phase ❷: Precomputing Segment Boundaries (*Merge Path*)



- Merge matrix $M[i,j]$: **1** if $A[i] < B[j]$, **0** otherwise.
- Merge path marks the **1**/**0** boundary.
- The *n*th point of the merge path lies on the *n*th cross diagonal ▪▪▪ of M.
- Partitioning the path into k segments of equal size can be done by considering k−1 evenly-spaced diagonals.
- Point -●- marks the only transition **1** → **0** on the diagonals. Can be found by binary seach on A and B. (Merge matrix and merge path *need not* be constructed.)
- Points -●- define segments in A and B (┆ ≡ boundaries):

A:	3 5 12	22 45	64 69 82	
B:	17	29 35	73	86 90 95 99
	#1	#2	#3	#4

- DuckDB generalizes the *Merge Path* idea from two runs to T runs. (Profiling shows that boundary computation uses ⩽ 2% of the overall execution time.)