Dissecting the Duck's Innards

③

Managing Memory + Grouped Aggregation

Winter 2025/26

Torsten Grust
Universität Tübingen, Germany

1 ┆ Memory: Fast, But Tiny vs. Slow, But Large

- There are enormous **latency gaps** between accesses to the (L2) CPU cache, RAM, and secondary storage (SSD/HDD):

Memory	Actual Latency	Human Scale	Typical Size
CPU L2 cache	2.8 ns	1 s	$\frac{1}{4}$–16 MB
RAM	≈ 100 ns	36 s	4–256 GB
SSD	50–150 µs	5–15 h	$\frac{1}{2}$–4 TB
HDD	1-10 ms	4-40 days	1–16 TB

- Fact: faster memory is significantly smaller. We will *not* be able to build cache-only DBMSs. Thus:
  - Keep **persistent database data** on secondary memory (disk).
  - During **query processing**, try to keep all of the currently relevant (or: “hot”) data in RAM.
  - General: attempt to process all queries at the upper levels of the memory hierarchy ( ).

Memory Management in DuckDB

Most DuckDB internals are optimized for **in-memory operation.** By default, 80% of the host RAM[1] is used to process data and queries.

Memory is a precious resource. DuckDB is built to use all RAM available but gradually moves to disk-based processing if required (*out-of-core processing*):

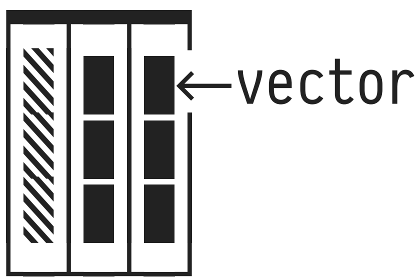
- If queries permit, **stream small data chunks** (*vectors*) through the system. Avoid to materialze entire data sets in memory.
- Try to hold **intermediate data structures** (e.g., hash tables) in memory. **Spill to disk** if a query produces huge intermediates.
- In the remaining memory space (if any), cache disk-resident table data to speed up future accesses.
  DB lingo: **buffering.**

#009

[1] Compare these 80% to PostgreSQL's default memory buffer size of 128 MB (shared_buffers). DuckDB's configuration parameter memory_limit controls the maximum RAM size available to the DBMS.

Streaming Query Execution (Pipelining)

Many SQL queries admit **streaming execution** in which **vectors** of only 2048 elements each flow from data source (tables) to result:



1 Route vectors (█,█) through query plan

(▨: irrelevant col)

2 Memory requirement is █+█+δ per thread at any instant
(δ: intermediate)

- Simple filter (WHERE) + project (SELECT) operations.
- Ungrouped aggregations or grouped aggregations (GROUP BY) for which the number of groups is small.
- Returning *N* rows only (where *N* is small, LIMIT *N*).
- Reading data from one file and writing to another (e.g., reading from CSV and writing to Parquet, COPY).

Such queries can be evaluated over larger-than-memory tables even under very constraining memory_limit settings.

#010

Spilling

Intermediate data structures larger than available RAM are common when processing complex SQL queries:

- Aggregation in which the grouping criteria create *many* groups (`GROUP BY`: hash table).
- Counting the *distinct* values in a column *C* with many distinct values (`count(distinct C)`: hash table).
- Joining two tables that *both* are larger than memory (join builds hash table for smaller table).
- Sorting larger-than-memory input data (`ORDER BY`).
- Evaluating a complex window function over larger-than-memory input (`… OVER …<frame>` with a sweeping frame).

DuckDB temporarily uses **out-of-core** memory on disk to hold (parts of) large intermediates (DB lingo: **spilling**).

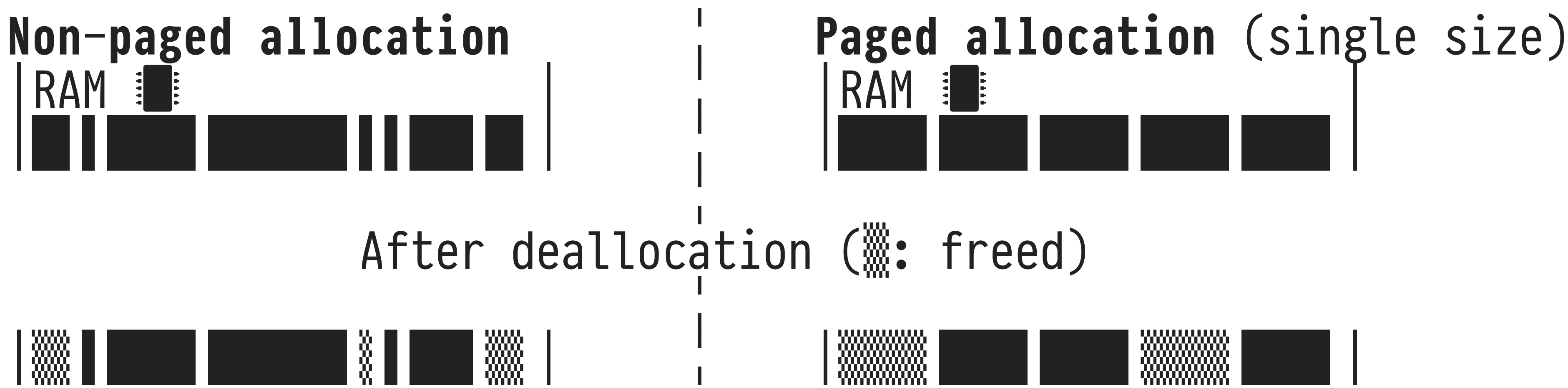
#011

DuckDB: Unified Memory Management

Use the *same memory area* of maximum size memory_limit to perform

1. **non-paged allocations:** fragmented (typically small) bits of data, e.g., as used in pointer-based data structures, and
2. **paged allocations:** blocks of contiguous data of various sizes, typically holding (parts of) tabular data structures. Typical block sizes range from 32KB to 256KB (default).

- DuckDB prefers paged allocation to avoid memory fragmentation:



Column-Wise Buffering of Base Table Data (→)

As long as memory usage permits, DuckDB **buffers** data read from persistent database file *.db file in RAM:

- File I/O is performed in 256KB blocks (as opposed to byte-by-byte) which matches the memory manager's default page size.

- Only buffer table columns relevant for query processing (recall Projections: … in operator SEQ_SCAN).

- If a page is no longer needed, add it to a LRU queue (DB lingo: **unpin**) as a candidate for replacement (DB lingo: **eviction**).
  - Buffered pages can be useful across queries: only evict once memory indeed is tight and needs to be reclaimed.
  - Buffered pages cache on-disk data (that could be re-read): *no need* to write evicted pages back to disk (↛).

Unified Memory Management in DuckDB

DuckDB flexibly assigns the available primary memory to all three kinds of allocations and thus is able to adapt to workloads/query specifics:

	Non-Paged Intermediate	Paged Intermediate	Buffered Base Table Data
Size	flexible	typical: 32…256KB (max: 2^{62} bytes)	256KB
Lifetime	query	query or session	across queries (database session)
Spilling	👎	👍	👎

Unified Memory Management in DuckDB

- In contrast to many DBMSs that require a pre-configuration[2] of buffer size (# of pages) or memory size for intermediates.

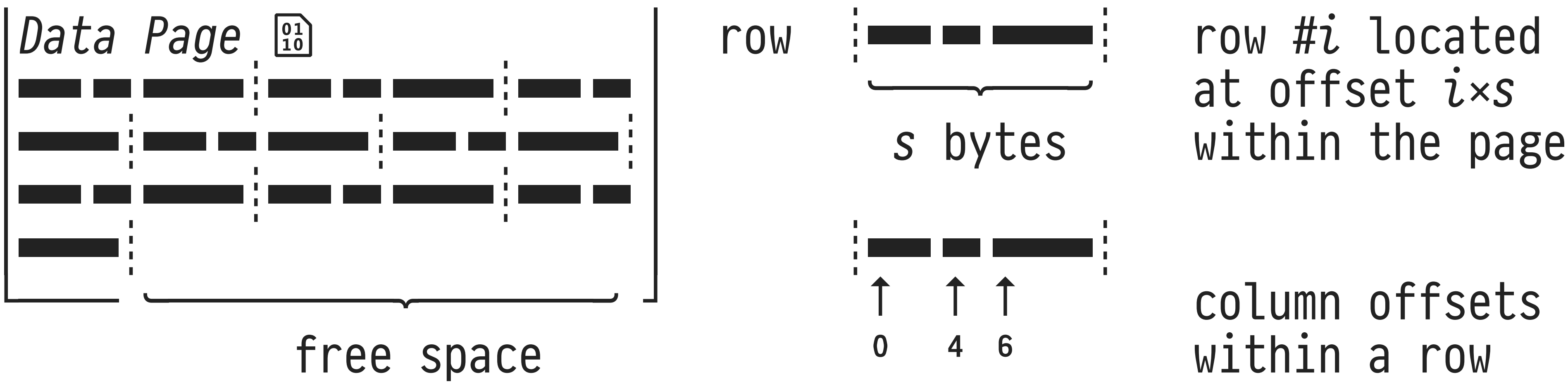
[2] Since DuckDB shares the address space with its host process, it is important that memory consumption is low when the DBMS is idle. DuckDB thus does use not a fixed-size pre-configured buffer.

#012

2 ┆ Layout of Paged Intermediate Data Structures

- DuckDB uses **column-based** storage for base tables, but relies on a **row-major layout** for paged intermediate data:
  - Co-locating the columns of a row helps row comparison during sorting, hashing, or joins (column access locality).

- **Fixed-size rows** with **columns of known width** admit efficient row/column access via offset calucations:

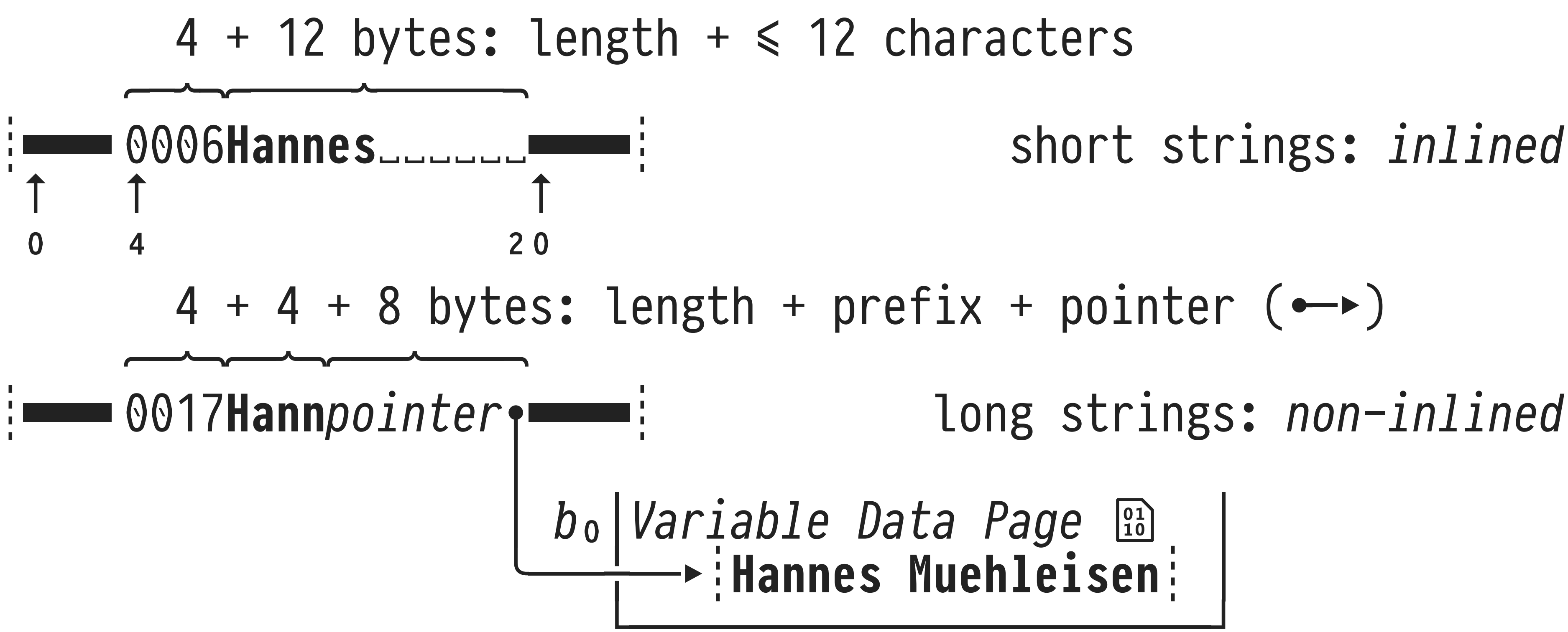


  - Row size s and column offsets are known at query plan generation time: store these only once per *Data Page*.

Variable-Sized Columns in Fixed-Sized Rows?

How to cope with **variable-sized columns** (of types `int[]` or `text`)?

- DuckDB: Store variable-sized data *outside* of fixed-sized rows on **separate pages.** Use pointers to refer to variable data:[3]

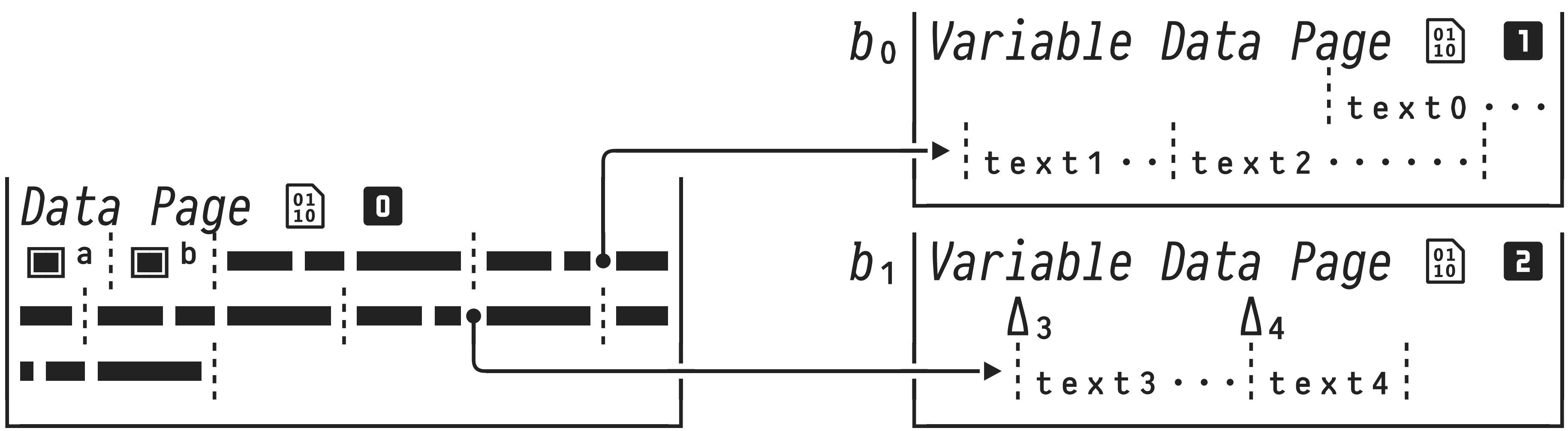


[3] This particular layout for string data is also known as "*German Strings*" since it has been popularized by the tabular DBMS *Umbra* developed at TU Munich (TUM). Now also used by *Apache Arrow* or *Polars*, for example.

Spilling Pages that Hold Pointer Targets

When a *Variable Data Page* is spilled and later loaded back, its base address b_0 may change and pointers ⟶ become invalid. 👎

1. A string on a *Var Page* with base address b lives at $b+\Delta$.
2. Store b as meta data ▣ at the start of the *Data Page*.
3. If the *Var Page* is loaded back at base address B, on the next dereference (lazily) replace pointers to $b+\Delta$ by $B+\Delta$.



▣ a: ⟨data page 0, row offset 0, var page 1 @ b_0⟩ } meta
▣ b: ⟨data page 0, row offset 3, var page 2 @ b_1⟩ } data

3 ¦ Hash-Based Grouped Aggregation

Grouping and aggregation are core operations in OLAP workloads:

```
SELECT l_orderkey, count(*)
FROM   lineitem                -- TPC-H (sf = 100): 600,000,000 rows
GROUP BY l_orderkey            -- creates 1,500,000 groups
```

- DuckDB implements grouped aggregation in terms of plan operator HASH_GROUP_BY, i.e., through **hashing:**[4]
  1. Row with l_orderkey value o hashes to key k.
     Update entry hash_table[k]: (o,$count$) → (o,$count$+1).
  2. Row with l_orderkey value $o' \neq o$ also hashes to key k.[5]
     Collides with entry for o at hash_table[k]. DuckDB uses linear probing to find entry (o',$count$)—hopefully nearby.

[5] If DuckDB statically knows that no collisions will occur (e.g., when the grouping key has few distinct values like lineitem.l_returnflag), the efficient alternative operator PERFECT_HASH_GROUP_BY is used.

[4] Other tabular DBMSs also implement grouped aggregation through sorting (here: on column l_orderkey). Profitable if the query also contains an ORDER BY l_orderkey clause (DB lingo: *interesting order*).

DuckDB: Robust External Grouped Aggregation

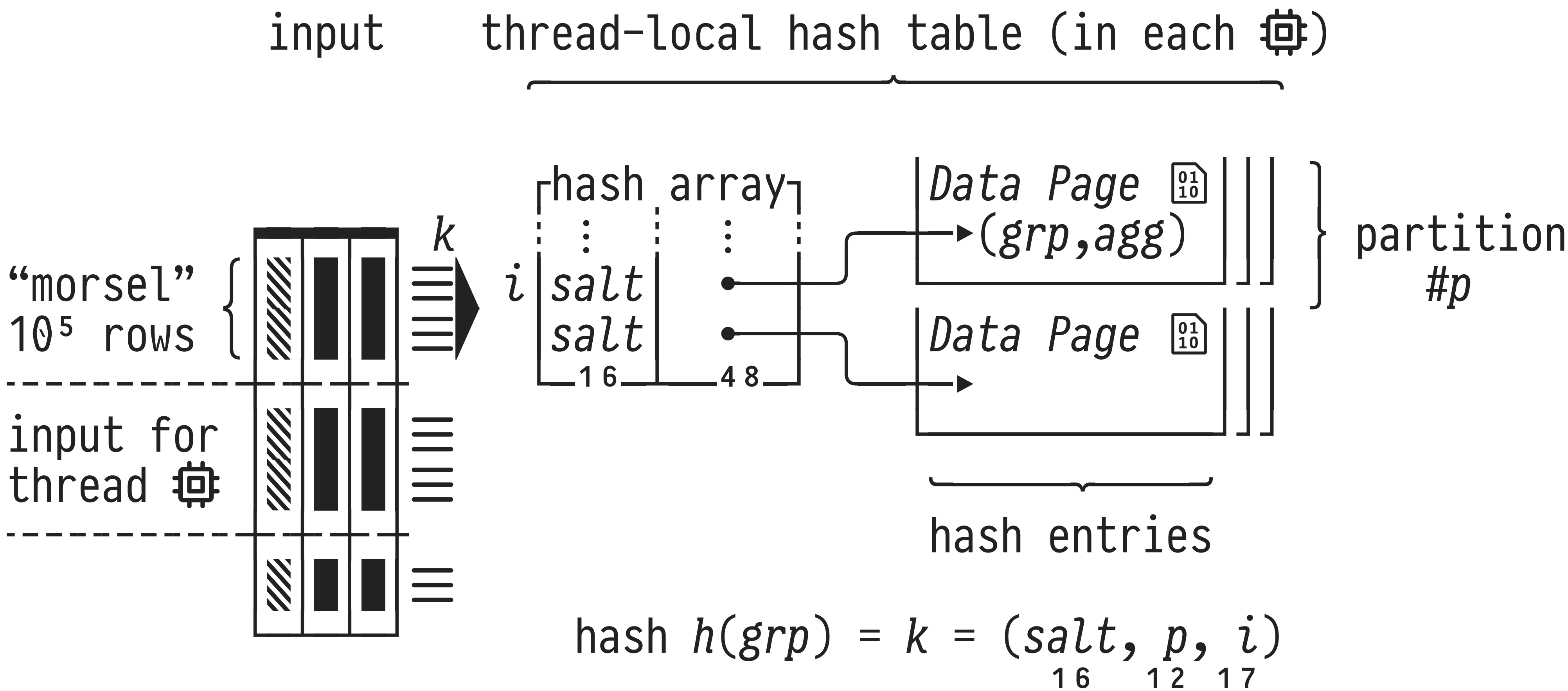
Designing the implementation of HASH_GROUP_BY:

- Aim to use all cores 🖥🖥🖥 of modern **multi-threaded** CPUs.
- If we create **many groups,** the hash table entries may exceed available memory and thus need to be **spilled** to disk.
  - 💡 Maintain hash table entries as a *paged intermediate data structure*, let the memory manager handle the spilling.

DuckDB performs external grouped aggregation in two phases:

- Phase ❶: Thread-local pre-aggregation (one 🖥 per morsel).
- Phase ❷: Partition-wise aggregation (one 🖥 per partition).

External Aggregation (Phase ❶): Thread-Local Aggregation



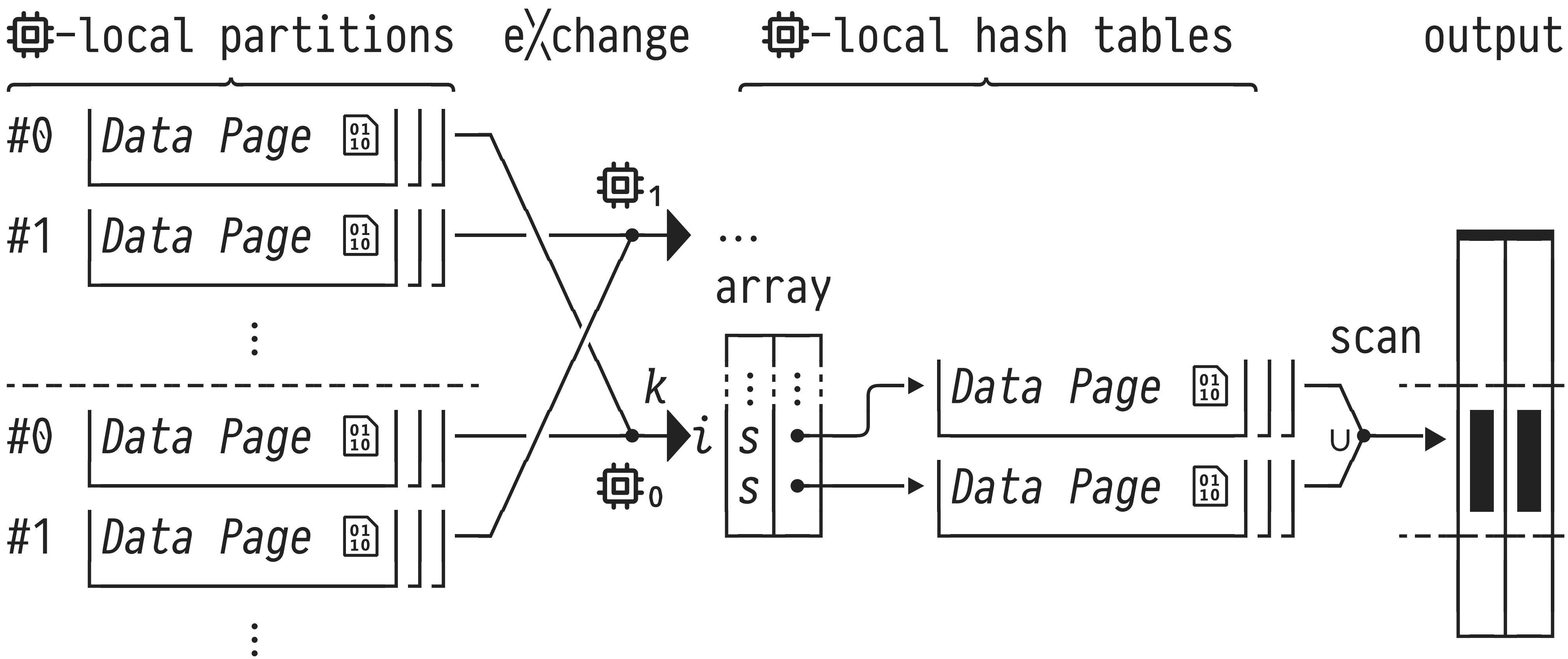
- Relevant columns of input table (grouping keys *grp* + arguments of aggregate *agg*) are split into **morsels** of ≈100,000 rows each.
- Each thread ⚙ reads a morsel. Typical: more morsels than ⚙s.

External Aggregation (Phase ❶): Thread-Local Aggregation

```
In each thread :
└ For each row in morsel:
   ├ Compute hash key k = h(grp). Split the bits of k:
   │  ├ Lower    17 bits i:    index in hash array (131,072 slots)
   │  ├ Middle ⩽ 12 bits p:    partition # for hash entry
   │  └ Upper    16 bits salt: used to optimize collision handling
   └ Hash array slot i occupied?
      ├y─ Are salt values equal?
      │   ┃y┬ Follow •─▸ to entry (grp′,agg) on data page .
      │   ┃ └ Is grp′ = grp?
      │   ┃     ├y─ No collision. Update value of agg. Done.
      │   ┃     └n─ Collision. Linear probing to find proper slot.
      │   ┗n─ Collision. Linear probing.
      └n┬ Add entry e = (grp,agg₀) to data page  in partition #p.
        └ Place (salt,•─▸ to e) in array slot i.
```

- When hash slots are $\frac{2}{3}$ full and collisions become frequent:
  1. Unpin data pages —the memory manager may evict these.
  2. Empty hash array, continue (reset OK because of Phase ❷).

External Aggregation (Phase ❷): Partition-Wise Aggregation



- Run one thread per partition #. Choose bits for *p* such that the hash table for a fully aggregated partition fits memory (create more but smaller partitions).
- When a ends, immediately scan its output data pages. These become a morsel to be fed to the downstream plan.