DB 2

11 – Sorting and Grouping

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 ¦ A Family of Q_9: The Ubiquitous Sort

Recall table indexed (with B$^+$Tree index indexed_a only):

❶
```
SELECT   i.*
FROM     indexed AS i
ORDER BY i.c
```

❷
```
SELECT DISTINCT i.c
FROM     indexed AS i
```

❸
```
SELECT   i.c, SUM(i.a) AS s
FROM     indexed AS i
GROUP BY i.c
```

❹
```
SELECT DISTINCT i1.a
FROM     indexed AS i1,
         indexed AS i2
WHERE    i1.a = i2.c :: int
```

All four queries are evaluated using the Sort plan operator.

Demonstrate the use of Sort in all plans for the four queries shown (and more):

```
-- table indexed and its sole primary key index on column a:

  \d indexed
                Table "public.indexed"
 ┌────────┬──────────────┬───────────┬──────────┬─────────┐
 │ Column │     Type     │ Collation │ Nullable │ Default │
 ├────────┼──────────────┼───────────┼──────────┼─────────┤
 │ a      │ integer      │           │ not null │         │
 │ b      │ text         │           │          │         │
 │ c      │ numeric(3,2) │           │          │         │
 └────────┴──────────────┴───────────┴──────────┴─────────┘
Indexes:
    "indexed_a" PRIMARY KEY, btree (a)

-- Query ❶: ORDER BY

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c;
 ┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
 │                                                        QUERY PLAN   blocking: high latency for first row                     │
 ├───────────────────────────────────────────────────────────────────▼──────────────────────────────────────────────────────────┤
 │ Sort  (cost=180530.84..183030.84 rows=1000000 width=41) (actual time=4193.436..4364.444 rows=1000000 loops=1)                │
 │ ☗ Output: a, b, c                                                                                                            │
 │   Sort Key: i.c ◀                                                                                                            │
 │   Sort Method: external sort  Disk: 50880kB ◀                                                                                │
 │   ->  Seq Scan on public.indexed i  (cost=0.00..19343.00 rows=1000000 width=41) (actual time=0.009..137.985 rows=1000000 loops=1) │
 │         Output: a, b, c                                                                                                      │
 │ Planning time: 0.059 ms                                                                                                      │
 │ Execution time: 4423.025 ms                                                                                                  │
 └──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘

-- Query ❷: DISTINCT

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT DISTINCT i.c
    FROM   indexed AS i;
 ┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
 │                                                          QUERY PLAN                                                          │
 ├──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┤
 │ Unique  (cost=132675.34..137675.34 rows=201 width=4) (actual time=4321.531..4716.960 rows=201 loops=1)                       │
 │   Output: c                                                                                                                  │
 │   ->  Sort  (cost=132675.34..135175.34 rows=1000000 width=4) (actual time=4321.530..4456.181 rows=1000000 loops=1)           │
 │       ☗ Output: c                                                                                                            │
```

```
        Sort Key: i.c ◀
        Sort Method: external sort  Disk: 14672kB
        ->  Seq Scan on public.indexed i  (cost=0.00..19343.00 rows=1000000 width=4) (actual time=0.010..231.416 rows=1000000 loops=1)
              Output: c
Planning time: 0.056 ms
Execution time: 4785.977 ms
```

```
-- Query ❸: GROUP BY

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT i.c, SUM(i.a) AS s
    FROM   indexed AS i
    GROUP BY i.c;
```

```
                                 QUERY PLAN

GroupAggregate  (cost=132675.34..140177.35 rows=201 width=12) (actual time=6890.495..7688.832 rows=201 loops=1)
  Output: c, sum(a)
  Group Key: i.c
  ->  Sort  (cost=132675.34..135175.34 rows=1000000 width=8) (actual time=6866.858..7092.946 rows=1000000 loops=1)
      ⬆ Output: c, a
        Sort Key: i.c ◀
        Sort Method: external sort  Disk: 21520kB
        ->  Seq Scan on public.indexed i  (cost=0.00..19343.00 rows=1000000 width=8) (actual time=0.018..400.581 rows=1000000 loops=1)
              Output: c, a
Planning time: 0.114 ms
Execution time: 7694.171 ms
```

```
-- Query ❹: merge join

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT DISTINCT i1.a
    FROM   indexed AS i1,
           indexed AS i2
    WHERE  i1.a = i2.c :: int;
```

```
                                 QUERY PLAN

Unique  (cost=132675.77..186155.77 rows=1000000 width=4) (actual time=1199.513..1435.166 rows=1 loops=1)
  Output: i1.a
  ->  Merge Join  (cost=132675.77..183655.77 rows=1000000 width=4) (actual time=1199.512..1407.527 rows=335165 loops=1)
      ⬆ Output: i1.a
      ➡ Merge Cond: (i1.a = ((i2.c)::integer))
        ->  Index Only Scan using indexed_a on indexed i1  (cost=0.42..25980.42 rows=1000000 width=4) (actual time=0.022..0.027 rows=2)
```

```
            ☗    Output: i1.a
sort for free    Heap Fetches: 0
        ->  Materialize  (cost=132675.34..137675.34 rows=1000000 width=4) (actual time=870.544..1207.765 rows=1000000 loops=1)
            ☗ Output: i2.c, ((i2.c)::integer)
           ⁂ ->  Sort  (cost=132675.34..135175.34 rows=1000000 width=4) (actual time=870.542..1030.140 rows=1000000 loops=1)
                 ☗  Output: i2.c, ((i2.c)::integer)
                    Sort Key: ((i2.c)::integer) ⬅ sort on join criterium
                    Sort Method: external sort  Disk: 21520kB
                    ->  Seq Scan on public.indexed i2  (cost=0.00..19343.00 rows=1000000 width=4) (actual time=0.012..415.605 rows=1000000)
                          Output: i2.c, (i2.c)::integer
Planning time: 0.287 ms
Execution time: 1443.858 ms
```

```
- Materialize: Merge Join needs to scan backward in join input

-- Query ❺ (not on slide): window aggregate

  EXPLAIN (VERBOSE, ANALYZE)
   SELECT i.c, SUM(i.a) OVER (ORDER BY i.c ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS w
   FROM   indexed AS i;
```

```
                                                                QUERY PLAN
WindowAgg  (cost=132675.34..150175.34 rows=1000000 width=12) (actual time=4330.367..5395.921 rows=1000000 loops=1)
  Output: c, sum(a) OVER (?)
  ->  Sort  (cost=132675.34..135175.34 rows=1000000 width=8) (actual time=4330.354..4473.724 rows=1000000 loops=1)
      ☗ Output: c, a
        Sort Key: i.c ⬅
        Sort Method: external sort  Disk: 21520kB
        ->  Seq Scan on public.indexed i  (cost=0.00..19343.00 rows=1000000 width=8) (actual time=0.010..242.257 rows=1000000 loops=1)
              Output: c, a
Planning time: 0.099 ms
Execution time: 5452.840 ms
```

Sorting Takes Time

- Operator Sort may be costly to evaluate and RDBMSs try to plan query execution without sorting if possible:
  - In queries ❶ to ❹ above, replace i.c (i2.c) by i.a and PostgreSQL will use Index Only Scans on a-ordered B^+Tree indexed_a instead of Sort.

- Sort is a **blocking operator** and intoduces plan latency:

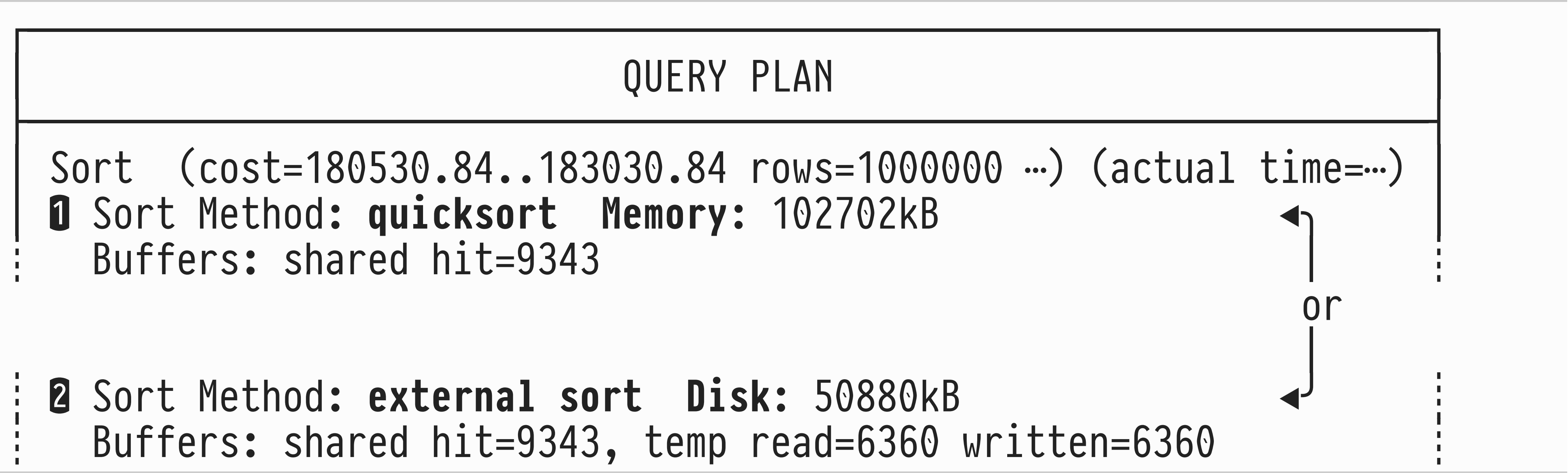
```
                         QUERY PLAN
 Sort (cost=180530.84..183030.84 rows=1000000 width=41)
  Output: a, b, c
```

Sorting Needs Space

Sorting may need (lots of) **temporary working memory:**

❶ Try to stay RAM-resident if possible,

❷ otherwise, resort to a **disk-based sorting algorithm:**



```
                              QUERY PLAN
 Sort  (cost=180530.84..183030.84 rows=1000000 …) (actual time=…)
 ❶ Sort Method: quicksort  Memory: 102702kB
   Buffers: shared hit=9343
                                                                or

 ❷ Sort Method: external sort  Disk: 50880kB
   Buffers: shared hit=9343, temp read=6360 written=6360
```

Demonstrate how PostgreSQL chooses sort implementations based on memory constraints/availability:

```
-- ❶ Evaluate query under tight memory constraints

  show work_mem;
```

work_mem
4MB

```
  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c;
```

QUERY PLAN

```
Sort  (cost=180530.84..183030.84 rows=1000000 width=41) (actual time=4443.492..4633.045 rows=1000000 loops=1)
  Output: a, b, c
  Sort Key: i.c      ⬇
  Sort Method: external sort  Disk: 50880kB ⬅
  Buffers: shared hit=9343, temp read=6360 written=6360 ⬅ 6360 × 8192 bytes = 50880 kB
  ->  Seq Scan on public.indexed i  (cost=0.00..19343.00 rows=1000000 width=41) (actual time=0.016..215.568 rows=1000000 loops=1)
        Output: a, b, c
        Buffers: shared hit=9343
Planning time: 0.111 ms
Execution time: 4713.534 ms ⬅
```

```
-- ❷ Re-valuate query with plenty of RAM-based temporaty working memory

  set work_mem = '1GB';

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT i.*
    FROM   indexed AS i
    ORDER BY i.c;
```

QUERY PLAN

```
Sort  (cost=119000.84..121500.84 rows=1000000 width=41) (actual time=472.288..546.005 rows=1000000 loops=1)
  Output: a, b, c
  Sort Key: i.c    ⬇
  Sort Method: quicksort  Memory: 102702kB ⬅ ~100MB of working memory used, no other queries running
  Buffers: shared hit=9343 ⬅ no additional buffer space needed for sorting
  ->  Seq Scan on public.indexed i  (cost=0.00..19343.00 rows=1000000 width=41) (actual time=0.018..115.513 rows=1000000 loops=1)
        Output: a, b, c
        Buffers: shared hit=9343
Planning time: 0.103 ms
Execution time: 630.707 ms ⬅ faster
```

```
set work_mem = default;
```

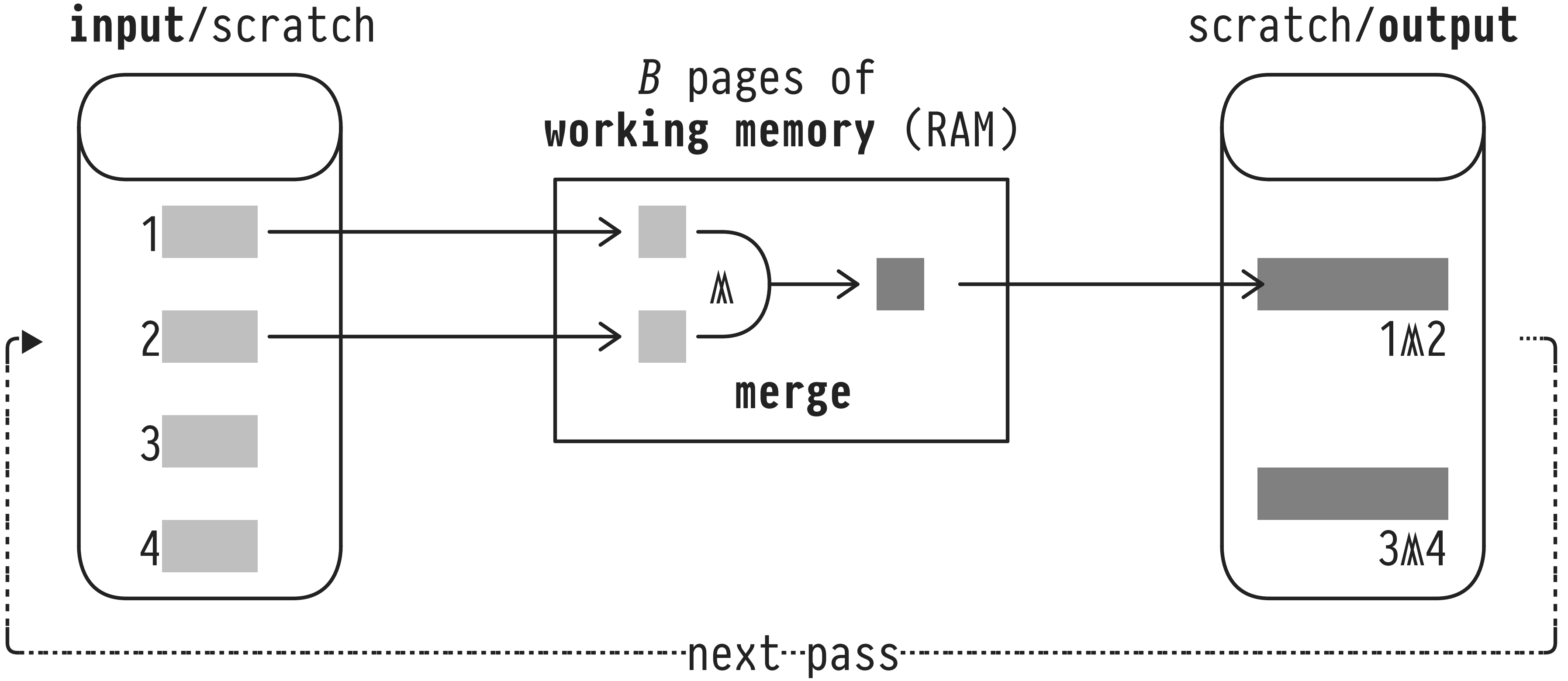
Now assume the following typical scenario:

- input heap file T to be sorted: N pages,
- size of temporary working memory (RAM): $B \ll N$ pages,
- size of secondary scratch memory (disk): $\geqslant 2 \times N$ blocks.

External Merge Sort can sort heap files of any size as long as $B \geqslant 3$ pages of working memory are available:

- reads unsorted input file, writes sorted output file,
- creates partially sorted sub-files (*runs*) on disk,
- multiple passes (the larger B, the fewer passes).

An External Merge Sort Pass (*B* = 3)



░ input run ⎫ sorted
▓ merged run ⎭
⩕ (*B*–1)-way merge

T = ░ ∪ ⋯ ∪ ░ = ▓ ∪ ⋯ ∪ ▓

(*B*–1) × |░| = |▓|

External Merge Sort

```
ExternalMergeSort($T$,$B$):
  $N$ ← #pages of $T$;
  $R$ ← $\lceil N/B \rceil$;                          } $R$: current number of runs

  split input $T$ into $R$ partitions $p_i$ of $B$ pages;  ⎫
  for each $i$ ∈ 1…$R$                                  ⎬ pass 0
   ⌊ run $r_i$ ← in-memory sort of $p_i$;                ⎭

  while $R$ > 1
   | $R$ ← $\lceil R$ / $(B-1) \rceil$;                  ⎫
   | for each $i$ ∈ 1…$R$                              ⎬ passes 1,2,…
   ⌊  ⌊ ⩕: merge next $B$-1 runs into one run;        ⎭

  return single sorted run;
```

- In each pass: if *R* is not perfectly divisible by *B*-1, the last merge ⩕ may merge less than *B*-1 runs.

External Merge Sort: Passes and I/O Operations

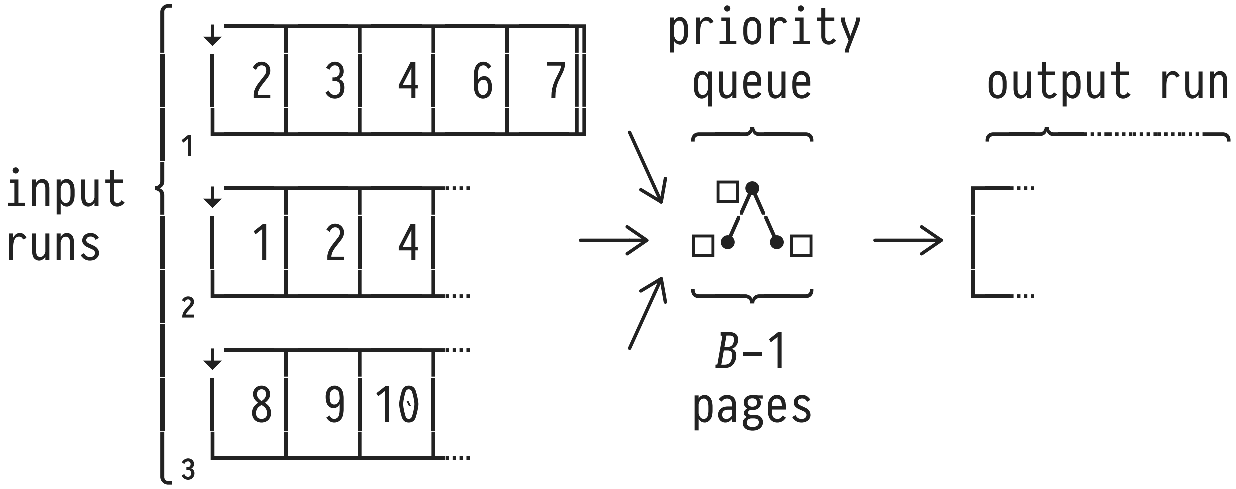
pass	input: #runs	input: run size	output: #runs	output: run size
1	$\lceil N/B \rceil$	B	$\lceil N/B \rceil$ / $(B-1)$	$B \times (B-1)$
2	$\lceil N/B \rceil$ / $(B-1)$	$B \times (B-1)$	$\lceil N/B \rceil$ / $(B-1)^2$	$B \times (B-1)^2$
3	$\lceil N/B \rceil$ / $(B-1)^2$	$B \times (B-1)^2$	$\lceil N/B \rceil$ / $(B-1)^3$	$B \times (B-1)^3$
⋮				
n	$\lceil N/B \rceil$ / $(B-1)^{n-1}$	$B \times (B-1)^{n-1}$	$\lceil N/B \rceil$ / $(B-1)^n$	$B \times (B-1)^n$

- In each pass:
  N = input (#runs × run size) = output (#runs × run size).
  - Each pass performs $2 \times N$ I/O operations.

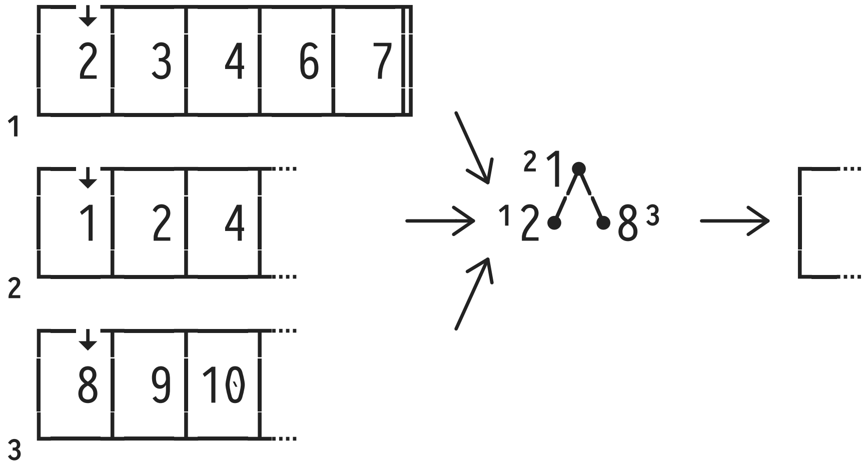
- Passes required by External Merge Sort with B buffers:

$$\underbrace{1}_{\text{pass 0}} + \underbrace{\lceil \log_{B-1} \lceil N/B \rceil \rceil}_{\text{merge passes}}$$

3 ¦ (B−1)-Way Merge (Passes 1,2,…)



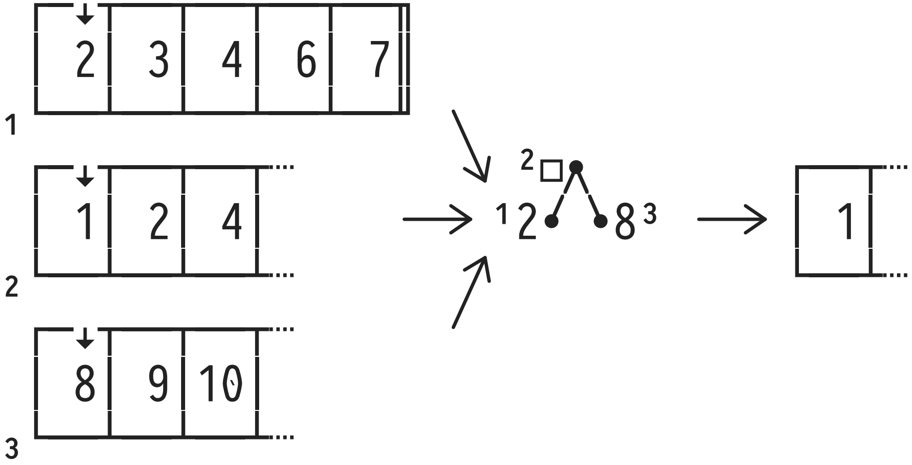
initial merge state



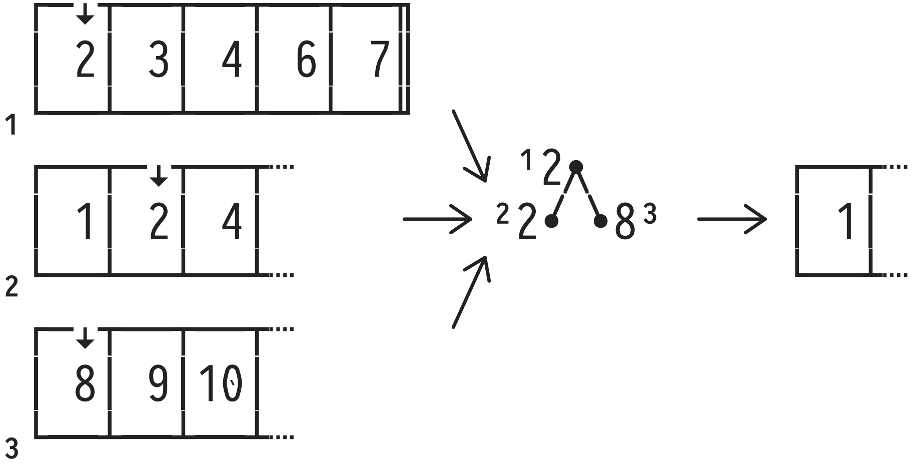
load queue,
entries from input run r_i
are tagged $□^i$

For simplicity, the above and the following diagrams assume that a block can only hold a single element (*e.g.*, element 2 in the first block of run #1).

(B−1)-Way Merge (Passes 1,2,…)

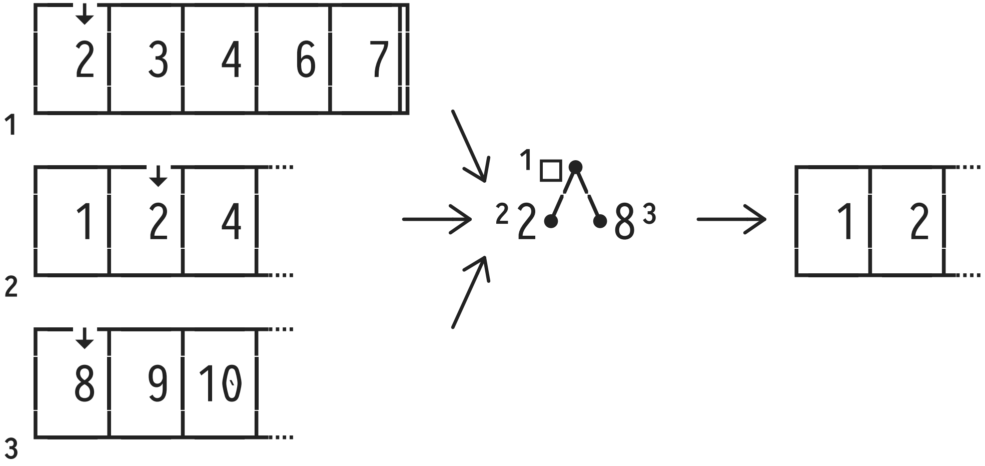


queue head → output run

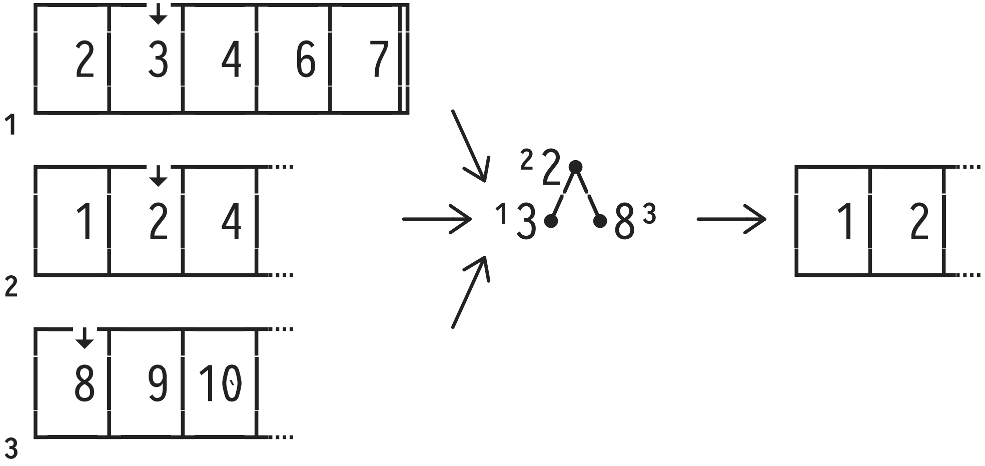


refill queue from input run

(B−1)-Way Merge (Passes 1,2,…)



queue head → output run



refill queue from input run

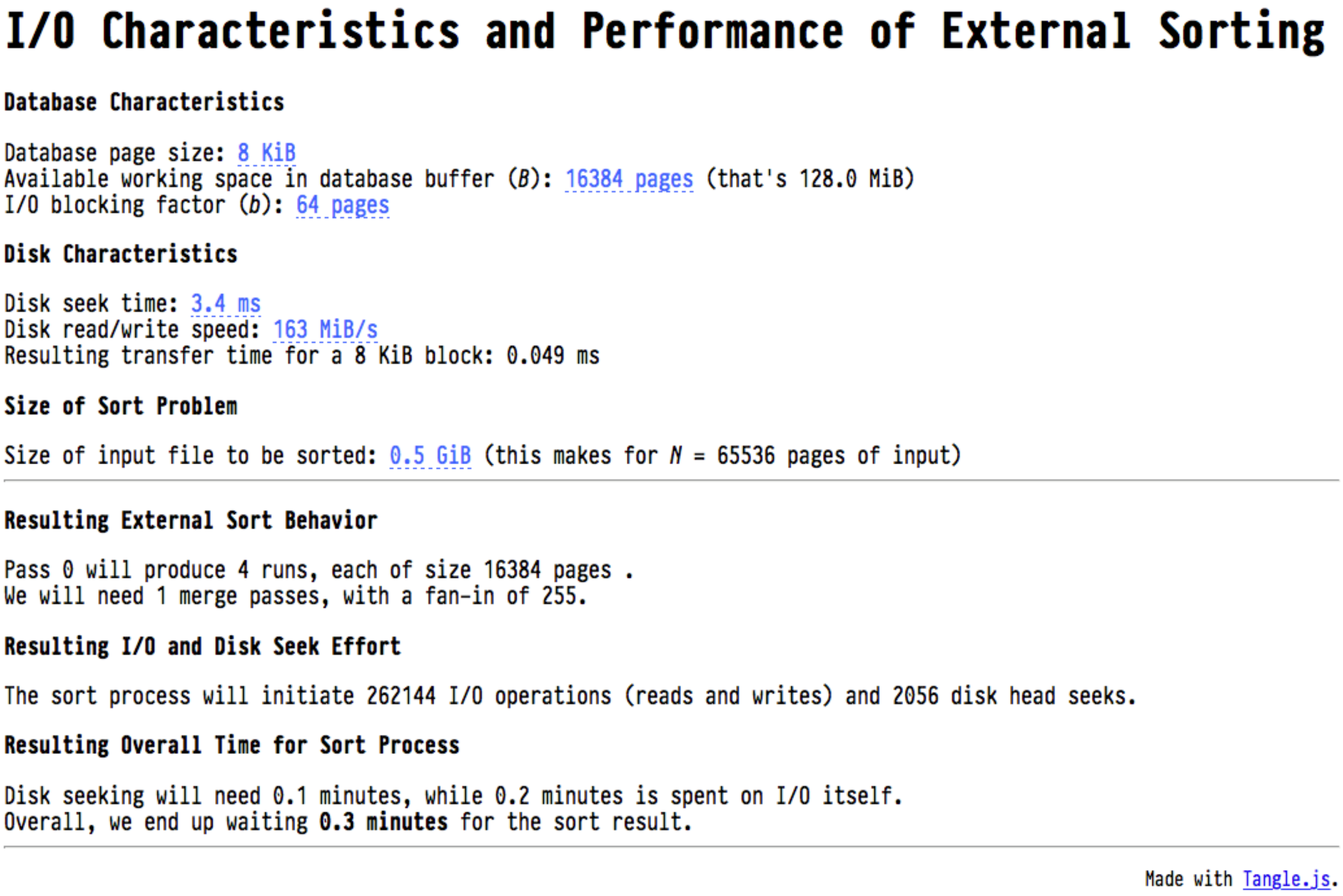
⋮

External Merge Sort: Access Patterns and Blocked I/O

- I/O access patterns in
  - pass 0: sequential read/write chunks of B pages, 👍
  - merge passes 1,…: random reads from the $B-1$ runs. 👎

- 💡 Perform **blocked I/O** in merge passes 1,2,…:
  - Seek once to read $b > 1$ pages at a time from each run. Reduces per-page I/O cost by a factor of $\approx b$.
  - Reduced fan-in: can only merge $\lfloor (B-1)/b \rfloor$ runs per pass.

External Merge Sort Parameters (Interactive)



I/O Characteristics and Performance of External Sorting

Database Characteristics

Database page size: 8 KiB
Available working space in database buffer (*B*): 16384 pages (that's 128.0 MiB)
I/O blocking factor (*b*): 64 pages

Disk Characteristics

Disk seek time: 3.4 ms
Disk read/write speed: 163 MiB/s
Resulting transfer time for a 8 KiB block: 0.049 ms

Size of Sort Problem

Size of input file to be sorted: 0.5 GiB (this makes for *N* = 65536 pages of input)

Resulting External Sort Behavior

Pass 0 will produce 4 runs, each of size 16384 pages .
We will need 1 merge passes, with a fan-in of 255.

Resulting I/O and Disk Seek Effort

The sort process will initiate 262144 I/O operations (reads and writes) and 2056 disk head seeks.

Resulting Overall Time for Sort Process

Disk seeking will need 0.1 minutes, while 0.2 minutes is spent on I/O itself.
Overall, we end up waiting **0.3 minutes** for the sort result.

Made with Tangle.js.

Interactive

4 ¦ Pass 0: Reducing the Number of Runs

- The *initial number of runs created in pass 0* influence overall sort performance:

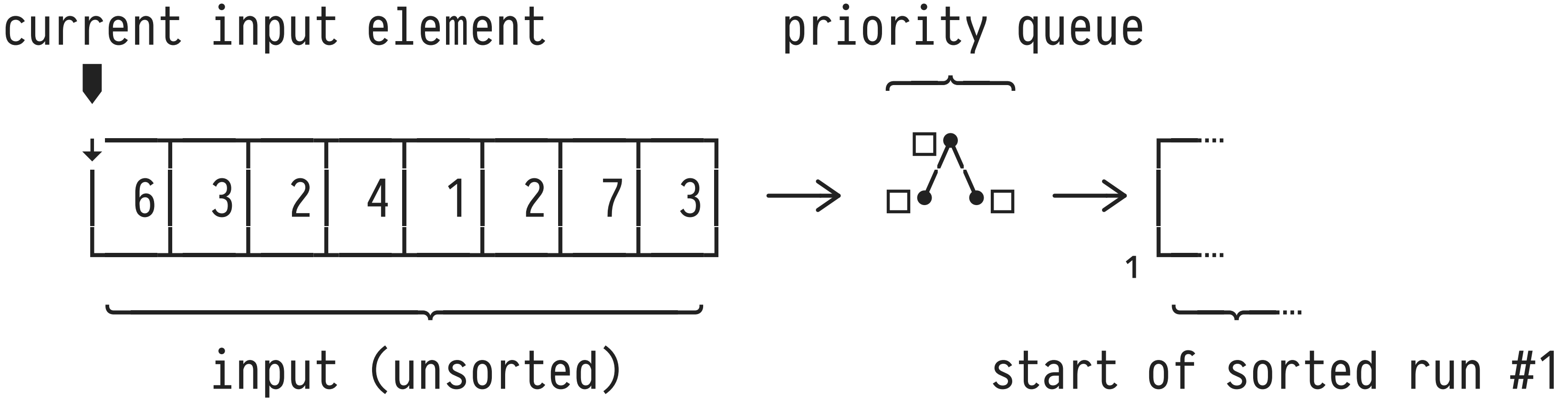
$$\text{\# I/O operations} = 2 \times N \times (1 + \lceil \log_{B-1} \underbrace{\lceil N/B \rceil}_{\text{\# runs created in pass 0}} \rceil)$$

- **Q:** Given only B buffers, can we create sorted runs *longer than* B pages?
    - **A:** Yes! In pass 0, use **Replacement Sort** (instead of QuickSort, for example).

Replacement Sort

Again, use *B*−1 buffer pages to set up a **priority queue:**

1. Elements arriving too late for inclusion in current run are marked (□$^{+}$) and receive lower priority.
2. When all elements in queue are marked, close the current run, unmark all elements, open a new run.



current input element

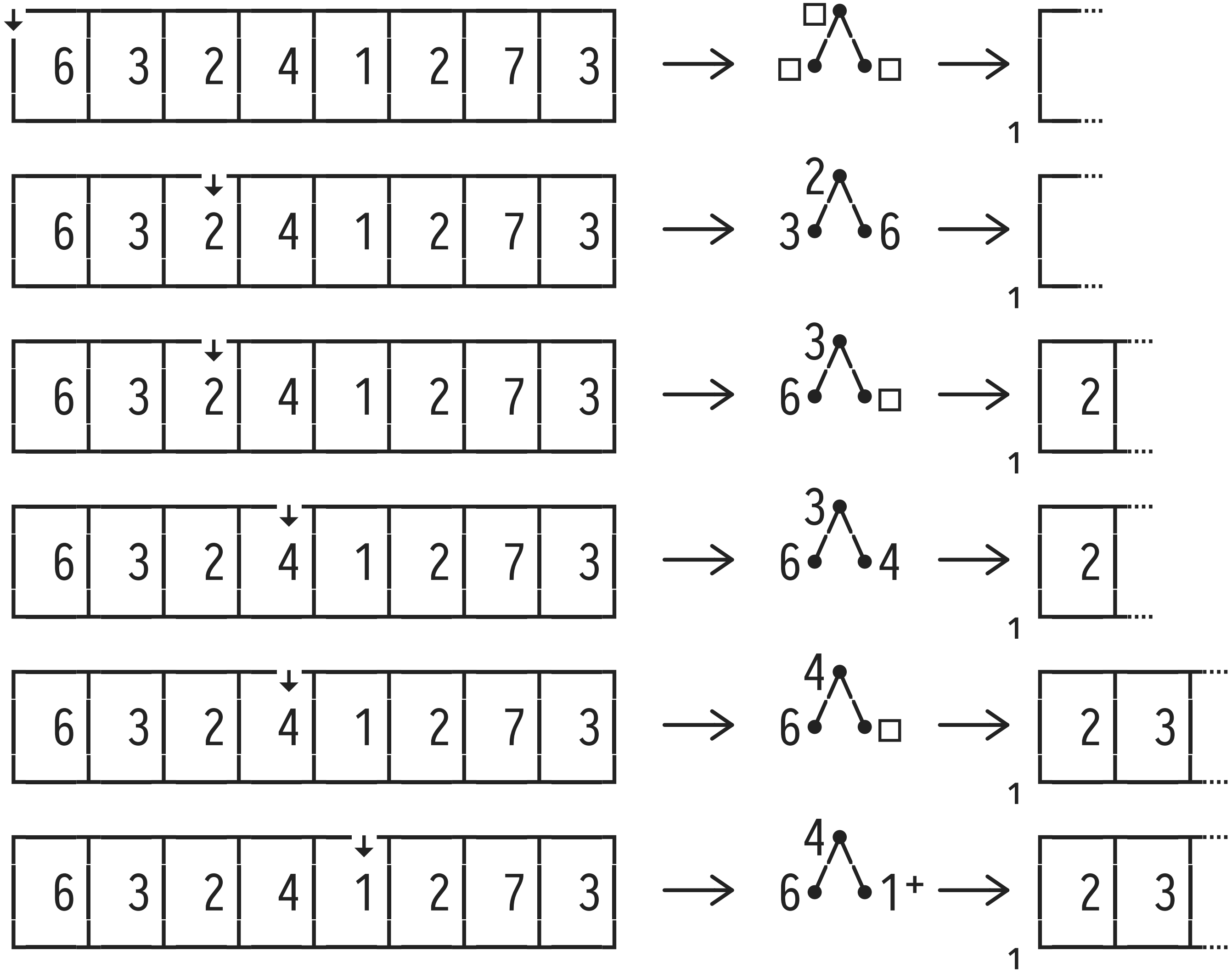
priority queue

6 3 2 4 1 2 7 3

input (unsorted)

start of sorted run #1

Replacement Sort (*B* = 4)



initial state

load queue

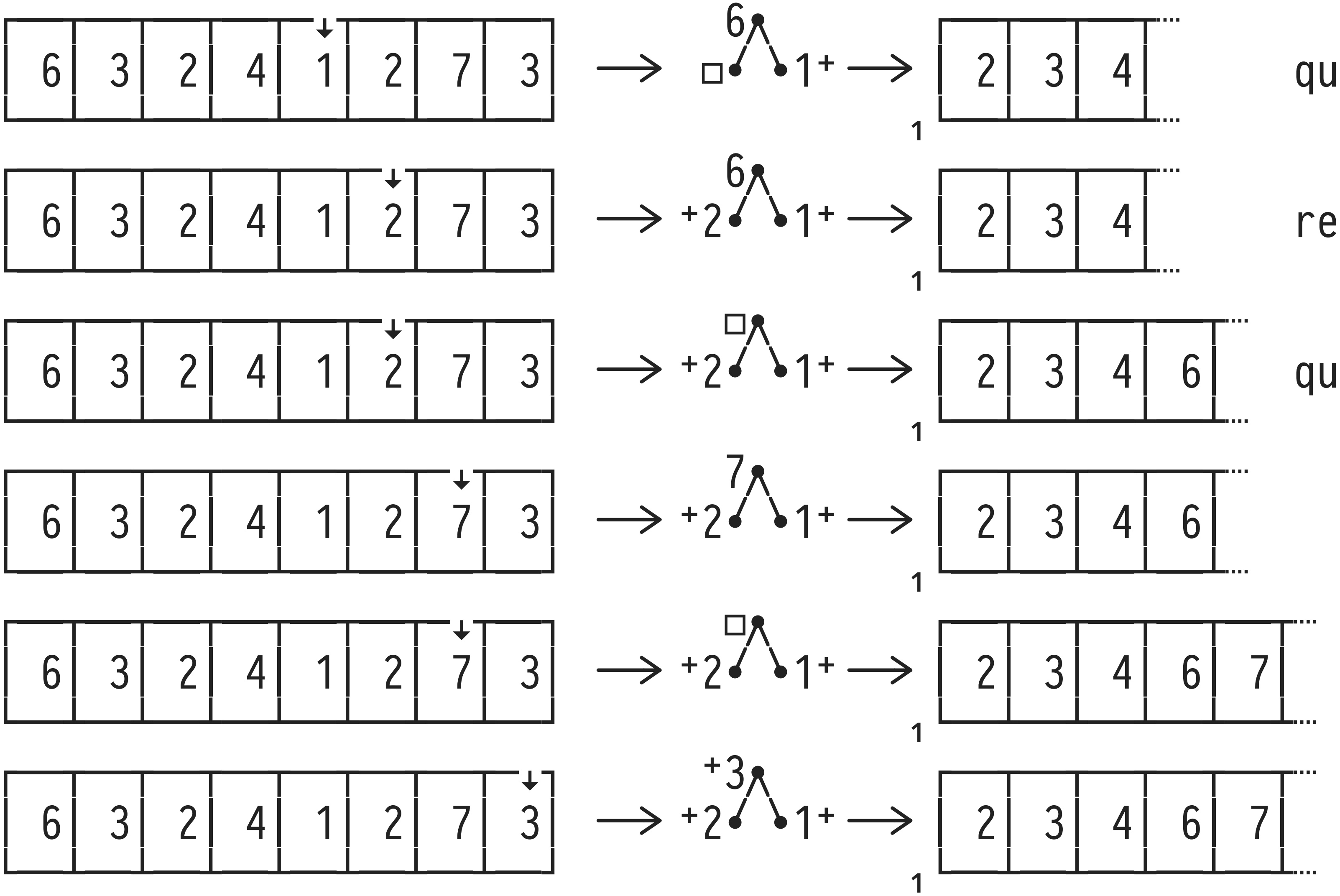
queue head → run #1

refill queue from input

queue head → run

refill q, 1 is late ($\square^{+}$)

Replacement Sort (B = 4)



queue head → run

refill q, 2 late

queue head → run

refill queue

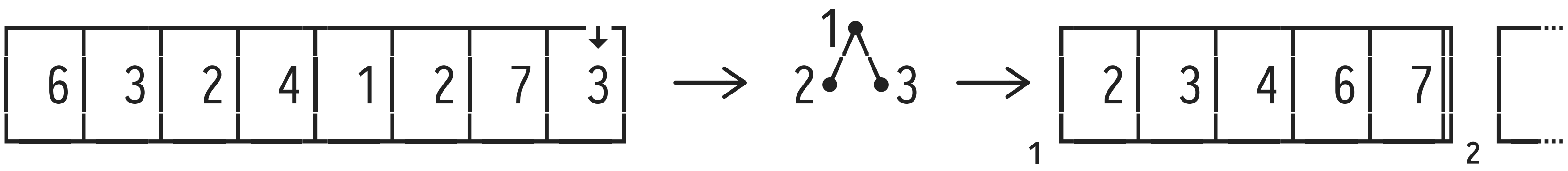
q head → run

refill, 3 late

Replacement Sort (B = 4)

All entries in queue are late ($\square^{+}$):

- Close current run #1, open new run #2.
- Reorder entries in queue, continue processing.



- Replacement Sort produces runs of length ≈ $2 \times (B-1) > B$ (see Knuth, TAoCP, volume 3, p. 254).
- Replacement Sort generates longer runs if input file is almost sorted (*e.g.*, consider a heap file that was once clustered but has received a few updates since then).

⚠️ This could work well as a homework assignment.

Replacement Sort with m memory slots ala Knuth:

1. The m slots are filled with records from the input to be sorted.
2. All slots are put into the **on** state.
3. Select the slot which has the smallest of all **on** slots.
4. Transfer the contents of the selected slot to the output (call its key Y).
5. Replace the contents of the selected slot by the next input record:
    - If new record key > Y, go to step 3.
    - If new record key = Y, go to step 4.
    - If new record key < Y, go to step 6.
6. Turn the selected key slot **off**.
    - If all slots are now **off**:
        - We have completed a sorted run.
        - Start a new run and go to step 2.
    - Else, go to step 3.

5 ¦ Q_{10}: Grouping

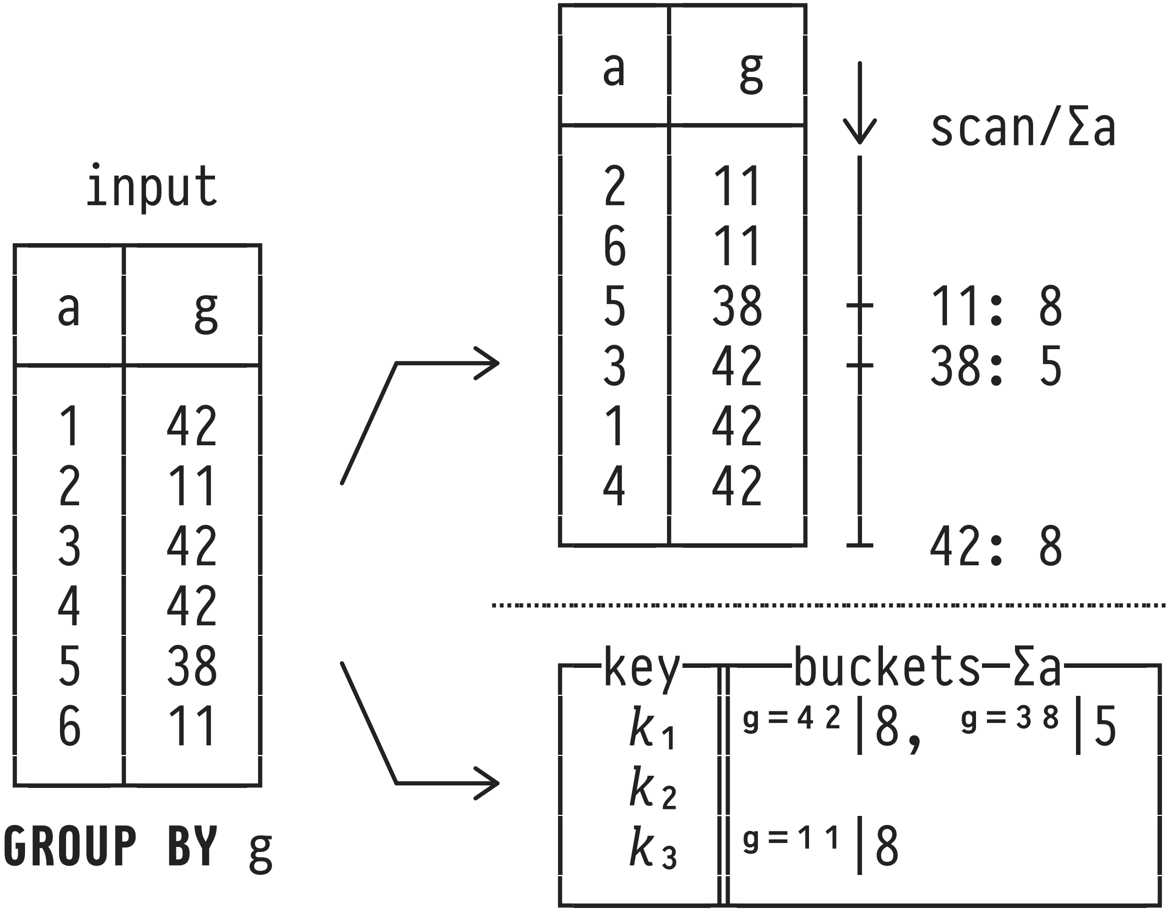
Grouping coarsens the granularity of data processing (individual rows ↘ groups of rows):

```
❷ SELECT g.c, SUM(g.g) AS s -- out: 10^4 groups (aggregates)
  FROM   grouped AS g       -- in:  10^6 rows
❶ GROUP BY g.g
```

❶ **Partition** table indexed by criterion g.g (all rows agreeing on g.g form one group),

❷ output group criterion and **aggregates** of the group's member rows (the group member rows themselves are never output).

Grouping: Sorting vs. Hashing



input

a	g
1	42
2	11
3	42
4	42
5	38
6	11

GROUP BY g

a	g
2	11
6	11
5	38
3	42
1	42
4	42

scan/Σa

11: 8
38: 5
42: 8

key	buckets—Σa
k_1	g=42|8, g=38|5
k_2	
k_3	g=11|8

- scan sorted table for group boundaries
- aggregate while scanning

Sorting

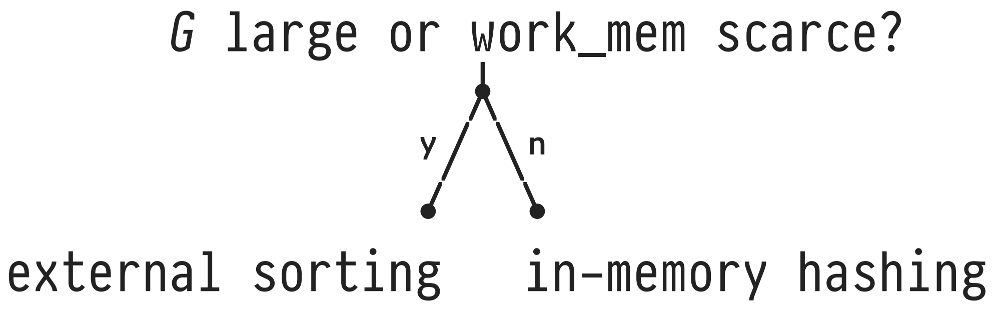
Hashing

- hash buckets hold grouping criterion and aggregate value

Grouping: Sorting vs. Hashing

PostgreSQL plans for sorting vs. hashing based on

1. the available working memory (work_mem) and
2. the estimated number G of resulting groups:



- Often, G is unknown or cannot be derived (*e.g.*, GROUP BY g.g % 2 ⇒ $G \leqslant 2$ not understood by PostgreSQL).
  - ⇒ Overestimate G conservatively, use sorting.

Demonstrate the switch from hashing to sorting when work_mem becomes scarce or when the estimated number of groups becomes (too) large:

```
-- ❶ Prepare table grouped, start off with default work_mem

  DROP TABLE IF EXISTS grouped;
  CREATE TABLE grouped (a int, g int);

  INSERT INTO grouped (a, g)
    SELECT i, i % 10000                      -- 10⁴ groups
    FROM   generate_series(1,1000000) AS i;  -- 10⁶ rows

  ANALYZE grouped;
  \d grouped
               Table "public.grouped"
```

Column	Type	Collation	Nullable	Default
a g	integer integer			

```
  show work_mem;
```

work_mem
4MB

```
-- ❷ Perform grouping with plenty of work_mem

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT g.g, SUM(g.a) AS s
    FROM   grouped AS g
    GROUP BY g.g;
```

```
                                         QUERY PLAN
      ▼
HashAggregate  (cost=19425.00..19524.74 rows=9974 width=12) (actual time=525.437..528.044 rows=10000 loops=1)
  Output: g, sum(a)                                    ▲
  Group Key: g.g                     proper estimate of G
  ->  Seq Scan on public.grouped g  (cost=0.00..14425.00 rows=1000000 width=8) (actual time=0.028..120.406 rows=1000000 loops=1)
        Output: a, g
Planning time: 0.179 ms
Execution time: 528.696 ms ◀ fast
```

```
-- ❸ Repeat grouping with scarce work_mem
```

```
set work_mem = '512kB';

EXPLAIN (VERBOSE, ANALYZE)
  SELECT g.g, SUM(g.a) AS s
  FROM   grouped AS g
  GROUP BY g.g;
```

```
                                                                    QUERY PLAN
──────▼────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
GroupAggregate  (cost=155106.34..162706.08 rows=9974 width=12) (actual time=1092.136..1520.714 rows=10000 loops=1)
  Output: g, sum(a)
  Group Key: g.g
  ->  Sort  (cost=155106.34..157606.34 rows=1000000 width=8) (actual time=1092.072..1299.080 rows=1000000 loops=1)
     ▲ Output: g, a
        Sort Key: g.g
        Sort Method: external merge  Disk: 17640kB
        ->  Seq Scan on public.grouped g  (cost=0.00..14425.00 rows=1000000 width=8) (actual time=0.029..206.028 rows=1000000 loops=1)
              Output: g, a
Planning time: 0.094 ms
Execution time: 1529.948 ms ◀ slow
```

```
-- ❹ Group count G is conservatively overestimated unless truly obvious for the system

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT g.g % 2, SUM(g.a) AS s
    FROM   grouped AS g
    GROUP BY g.g % 2;     -- ◀ will create two groups max, goes undeteced by PostgreSQL :-(
```

```
                                                                    QUERY PLAN
──────▼────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
GroupAggregate  (cost=157606.34..165231.02 rows=9974 width=12) (actual time=1093.451..1284.498 rows=2 loops=1)
  Output: ((g % 2)), sum(a)                               ▲                                                  ▲
  Group Key: ((g.g % 2))                      should be 2 :-(                                             aha!
  ->  Sort  (cost=157606.34..160106.34 rows=1000000 width=8) (actual time=897.882..1081.040 rows=1000000 loops=1)
        Output: ((g % 2)), a
        Sort Key: ((g.g % 2))
        Sort Method: external merge  Disk: 17640kB
        ->  Seq Scan on public.grouped g  (cost=0.00..16925.00 rows=1000000 width=8) (actual time=0.031..221.928 rows=1000000 loops=1)
              Output: (g % 2), a ◀ Seq Scan already performs arithmetics (g.g not needed in downstream plan)
Planning time: 0.108 ms
Execution time: 1293.528 ms
```

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT g.g % 2 = 0, SUM(g.a) AS s
  FROM   grouped AS g
  GROUP BY g.g % 2 = 0;  -- ⬅ creates a Boolean, this IS detected by PostgreSQL (|dom(bool)| = 2)
```

```
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
HashAggregate  (cost=24425.00..24425.03 rows=2 width=9) (actual time=605.891..605.892 rows=2 loops=1)
  Output: (((g % 2) = 0)), sum(a)           ⬆
  Group Key: ((g.g % 2) = 0)       proper group estimate for G
  ->  Seq Scan on public.grouped g  (cost=0.00..19425.00 rows=1000000 width=5) (actual time=0.062..257.098 rows=1000000 loops=1)
        Output: ((g % 2) = 0), a
Planning time: 0.157 ms
Execution time: 606.036 ms
```

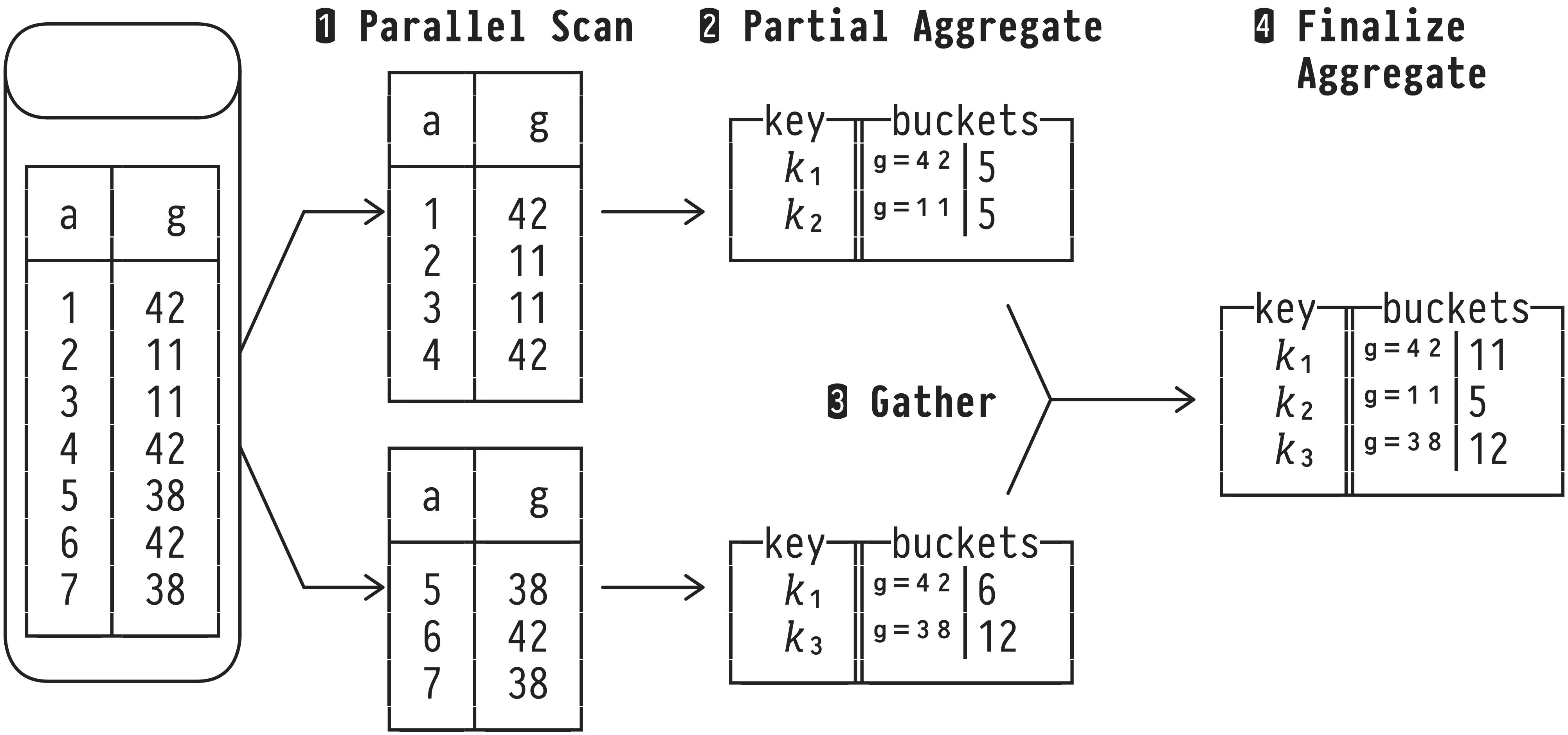
```
set work_mem = default;
```

6 ¦ Parallel Grouping and Aggregation

Grouping and aggregation are query operations that are straightforward to **parallelize:**

- Spawn **workers,** each of which execute in // (on dedicated CPU core). Constrain max number of workers to fit host.
- Try to **evenly distribute work** (*e.g.*, data volume) among workers.
- Assign a **leader** thread/process that coordinates workers and **gathers partial query results.**
- After gathering, **merge/finalize partial results** to produce a single complete query result.

Parallel Grouping (GROUP BY g — SUM(a))



a	g
1	42
2	11
3	11
4	42
5	38
6	42
7	38

❶ Parallel Scan

a	g
1	42
2	11
3	11
4	42

a	g
5	38
6	42
7	38

❷ Partial Aggregate

key	buckets	
k_1	g=42	5
k_2	g=11	5

key	buckets	
k_1	g=42	6
k_3	g=38	12

❸ Gather

❹ Finalize Aggregate

key	buckets	
k_1	g=42	11
k_2	g=11	5
k_3	g=38	12

Parallel Grouping for Q_{10}

```
EXPLAIN
  SELECT g.g, SUM(g.a) AS s
  FROM   grouped AS g
  GROUP BY g.g;
```

```
                              QUERY PLAN

Finalize HashAggregate (cost=13869.28..13969.02 …)
 Group Key: g
 -> Gather (cost=11675.00..13769.54 …)
     Workers Planned: 2 ⬅ ∥ism degree: 3 (2 worker + 1 leader)
     -> Partial HashAggregate (cost=10675.00..10774.74 …)
         Group Key: g
         -> Parallel Seq Scan on grouped g (cost=0.00..8591.67 …)
```

Partial Aggregation and Finalization

- Parallel evaluation of aggregate *AGG* depends on the **distributivity** over $\uplus$ (bag union):

$$AGG(X \uplus Y) = AGG(\{AGG(X)\} \uplus \{AGG(Y)\}).$$

- Many SQL aggregates (COUNT, SUM, MAX, MIN, AVG, bool_and, bool_or, …) exhibit this property:

$$\text{SUM}(\underbrace{X \uplus Y}_{\substack{\text{distribute}\\ \text{work}}}) = \text{SUM}(\{\underbrace{\text{SUM}(X)}_{\text{partial}}\} \uplus \{\underbrace{\text{SUM}(Y)}_{\text{aggregates}}\}) = \text{SUM}(X) \underset{\substack{\uparrow\\ \text{finalize}}}{+} \text{SUM}(Y)$$

Demonstrate the parallel grouping and aggregation for query Q_{10}. Works for distributive aggregate SUM/+, does *not* work for array_agg/||.

```
-- ❶ Enable generation of // plans (⚠ this is supposed to be disabled in the lecture)

  set max_parallel_workers = default;             -- = 8
  set max_parallel_workers_per_gather = default;  -- = 8

-- ❷ Parallel grouping for `SUM`

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT g.g, SUM(g.a) AS s       -- 10^4 groups
    FROM   grouped AS g             -- 10^6 rows
    GROUP BY g.g;
```

```
                                            QUERY PLAN
Finalize HashAggregate  (cost=13869.28..13969.02 rows=9974 width=12) (actual time=295.894..298.032 rows=10000 loops=1)
  Output: g, sum(a)                                                                                     ⬆
  Group Key: g.g                                                                               merge partial group aggregates
  ->  Gather  (cost=11675.00..13769.54 rows=19948 width=12) (actual time=270.706..282.993 rows=30000 loops=1)
        Output: g, (PARTIAL sum(a))
        Workers Planned: 2  ⬅ 2 × worker + 1 × leader (subplan below Gather is evaluated in //)
        Workers Launched: 2
        ->  Partial HashAggregate  (cost=10675.00..10774.74 rows=9974 width=12) (actual time=257.653..261.423 rows=10000 loops=3)
              Output: g, PARTIAL sum(a)                                                                                  ⬆
              Group Key: g.g                                                                                             ⋮
              Worker 0: actual time=251.271..255.311 rows=10000 loops=1    ⬅ each worker + leader contributes to all
              Worker 1: actual time=251.158..255.345 rows=10000 loops=1    ⬅ 10^4 groups
              ->  Parallel Seq Scan on grouped g  (cost=0.00..8591.67 rows=416667 width=8) (actual time=0.031..96.647 rows=333333 loops=3)
                    Output: g, a                                                                                                        ⬆
                    Worker 0: actual time=0.036..96.484 rows=326294 loops=1                                                  degree of //ism
                    Worker 1: actual time=0.041..95.847 rows=315948 loops=1
Planning time: 0.087 ms                                        ⬆
Execution time: 298.778 ms      size of data partition for workers + leader: 333333 + 326294 + 315948 ≈ 1000000
```

```
-- ❸ Check aggregates and their finalize operations (for type int)
--   (aggregates that can be used in parallel/partial mode [missing: array_agg, ...])

  SELECT a.aggfnoid, a.aggcombinefn, a.agginitval, t.typname
  FROM   pg_aggregate AS a, pg_type AS t
  WHERE  a.aggcombinefn <> 0 and a.aggkind = 'n'
  AND    a.aggtranstype = t.oid AND t.typname LIKE '%int_';
```

aggfnoid	aggcombinefn	agginitval	typname

```
│ pg_catalog.avg      │ int4_avg_combine │ {0,0}       │ _int8  │ (AVG, (sum₁,n₁) ⊕ (sum₂,n₂) = (sum₁+sum₂,n₁+n₂)), finalize: sum/n
│ pg_catalog.avg      │ int4_avg_combine │ {0,0}       │ _int8  │
│ pg_catalog.sum      │ int8pl           │ □ ◀         │ int8   │ (SUM, +)
│ pg_catalog.sum      │ int8pl           │ □           │ int8   │
│ pg_catalog.max      │ int8larger       │ □           │ int8   │ (MAX, >)
│ pg_catalog.max      │ int4larger       │ □           │ int4   │
│ pg_catalog.max      │ int2larger       │ □           │ int2   │
│ pg_catalog.min      │ int8smaller      │ □           │ int8   │ (MIN, <)
│ pg_catalog.min      │ int4smaller      │ □           │ int4   │
│ pg_catalog.min      │ int2smaller      │ □           │ int2   │
│ pg_catalog.count    │ int8pl           │ 0 ◀         │ int8   │ (COUNT, +)
│ pg_catalog.count    │ int8pl           │ 0           │ int8   │
│ regr_count          │ int8pl           │ 0           │ int8   │
│ pg_catalog.bit_and  │ int2and          │ □           │ int2   │
│ pg_catalog.bit_or   │ int2or           │ □           │ int2   │
│ pg_catalog.bit_and  │ int4and          │ □           │ int4   │
│ pg_catalog.bit_or   │ int4or           │ □           │ int4   │
│ pg_catalog.bit_and  │ int8and          │ □           │ int8   │
│ pg_catalog.bit_or   │ int8or           │ □           │ int8   │
└─────────────────────┴──────────────────┴─────────────┴────────┘

-- ❹ Plans with non-distributive aggregates cannot be ∥ized this easily,
--   example: array_agg/||
--   (⚠️ Future versions of PostgreSQL may add parallel array_agg)
--
--   array_agg({1,3,5,2,4,6} ORDER BY x)
--    ≠
--   array_agg({1,3,5} ORDER BY x) || array_agg({2,4,6} ORDER BY x)

  SELECT array_agg(x ORDER BY x) AS xs
  FROM   generate_series(1, 10) AS x;

┌──────────────────────┐
│          xs          │
├──────────────────────┤
│ {1,2,3,4,5,6,7,8,9,10} │
└──────────────────────┘


  SELECT (
    (SELECT array_agg(x ORDER BY x) AS xs
     FROM   generate_series(1, 10) AS x
     WHERE  x % 2 = 0)
      ||
    (SELECT array_agg(x ORDER BY x) AS xs
     FROM   generate_series(1, 10) AS x
     WHERE  NOT(x % 2 = 0))
  ) AS xs;

┌──────────────────────┐
│          xs          │
```

```
{2,4,6,8,10,1,3,5,7,9}
```

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT g.g, array_agg(g.a ORDER BY g.a) AS s   -- 10⁴ groups
  FROM   grouped AS g                            -- 10⁶ rows
  GROUP BY g.g;
```

```
                                                              QUERY PLAN
GroupAggregate  (cost=127757.34..135382.02 rows=9974 width=36) (actual time=1090.934..1872.477 rows=10000 loops=1)
  Output: g, array_agg(a ORDER BY a)
  Group Key: g.g
  ->  Sort  (cost=127757.34..130257.34 rows=1000000 width=8) (actual time=1090.836..1332.555 rows=1000000 loops=1)
        Output: g, a
        Sort Key: g.g
        Sort Method: external merge  Disk: 17696kB
        ->  Seq Scan on public.grouped g  (cost=0.00..14425.00 rows=1000000 width=8) (actual time=1.214..341.390 rows=1000000 loops=1)
            ▲ Output: g, a                                            ▲
Planning time: 0.077 ms                                       no work distribution
Execution time: 1880.262 ms
```

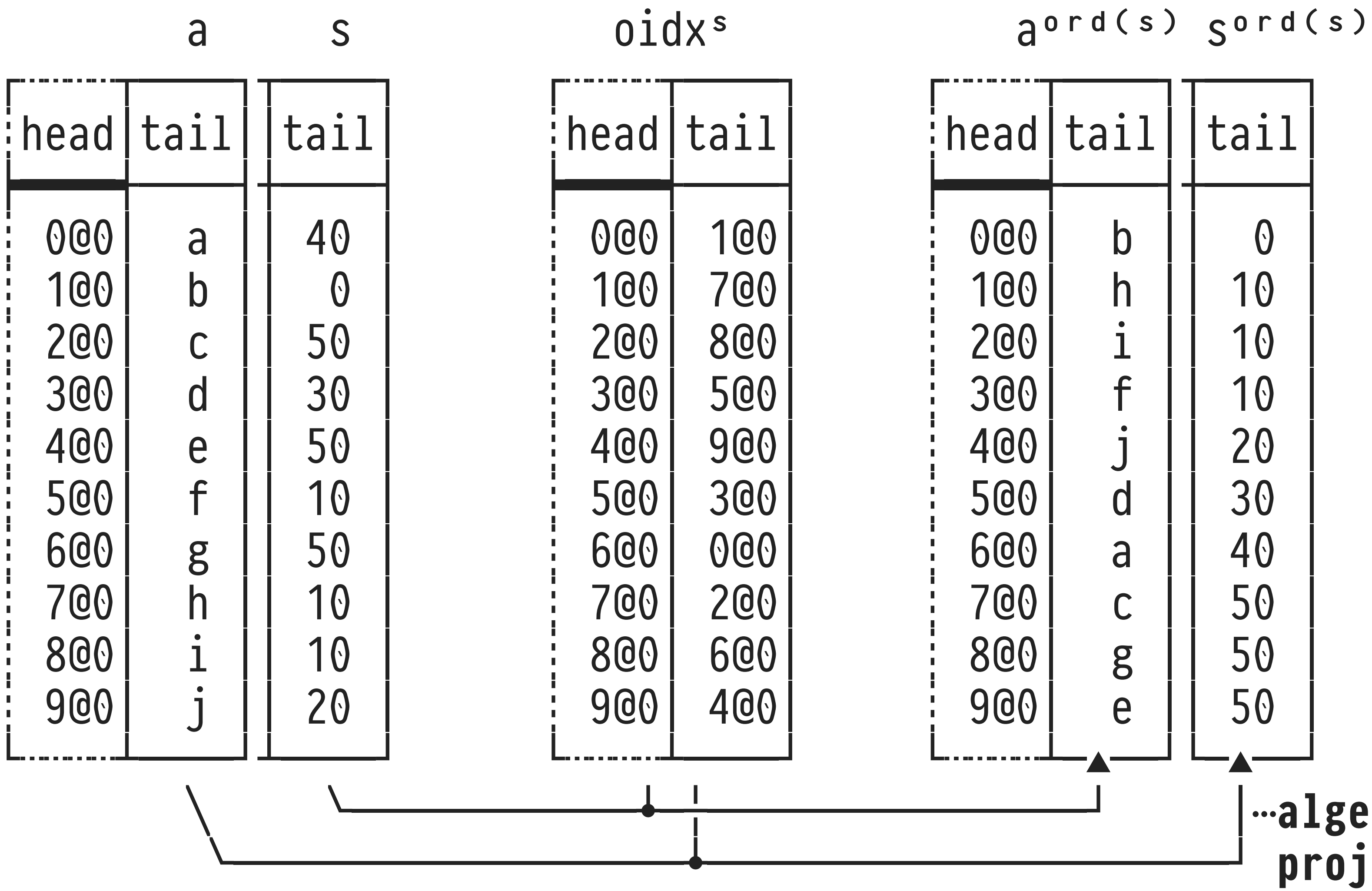
Q_9: Sorting in MonetDB

```
CREATE TABLE sorted (a text, s int);
  ⋮
SELECT s.a, s.s
FROM   sorted AS s
ORDER BY s.s [, s.a]   -- single- or multi-column criteria
```

MonetDB's BATs already provide **ordered row storage.** Some ORDER BY queries will thus be no-ops (recall tail properties sorted, revsorted).

Otherwise, use **order indexes**—either persistent or computed on the fly—to apply column re-ordering.

Recall: Order Indexes (ORDER BY s.s)



a

head	tail
0@0	a
1@0	b
2@0	c
3@0	d
4@0	e
5@0	f
6@0	g
7@0	h
8@0	i
9@0	j

s

tail
40
0
50
30
50
10
50
10
10
20

oidx^{s}

head	tail
0@0	1@0
1@0	7@0
2@0	8@0
3@0	5@0
4@0	9@0
5@0	3@0
6@0	0@0
7@0	2@0
8@0	6@0
9@0	4@0

a$^{ord(s)}$

head	tail
0@0	b
1@0	h
2@0	i
3@0	f
4@0	j
5@0	d
6@0	a
7@0	c
8@0	g
9@0	e

s$^{ord(s)}$

tail
0
10
10
10
20
30
40
50
50
50

…**algebra.**
projection(oidx^{s},·)

Order Indexes on the Fly: algebra.sort

```
EXPLAIN
  SELECT s.a, s.s
  FROM   sorted AS s
  ORDER BY s.s;

sorted :bat[:oid] := sql.tid(sql, "sys", "sorted");
s0     :bat[:int] := sql.bind(sql, "sys", "sorted", "s", …);
s      :bat[:int] := algebra.projection(sorted, s0);
(s^ord(s), oidx^s, gidx^s)                desc↓      ↓stable
                  := algebra.sort(s, false, false);
a0     :bat[:str] := sql.bind(sql, "sys", "sorted", "a", …);
a      :bat[:str] := algebra.projection(sorted, a0);
a^ord(s):bat[:str] := algebra.projectionpath(oidx^s, sorted, a0);

io.print(a^ord(s), s^ord(s));
```

Persistent Order Indexes

If sorting is central to the query workload, create a **persistent order index** that is immediately applicable:

```
ALTER TABLE sorted SET READ ONLY;      -- ⚠

CREATE ORDERED INDEX oidx^s ON sorted(s);
```

- Order indexes are **static** structures that are *not* dynamically maintained (as opposed to B$^+$Trees). If order index has been created…
  1. on the fly: throw away on table update,
  2. persistent: read-only table, no updates at all.

What does it mean to maintain an order index on table update?

1. Before update:

s

head	tail
0@0	40
1@0	0
2@0	50
3@0	30
4@0	50
5@0	10
6@0	50
7@0	10
8@0	10
9@0	20

oidx^s

head	tail
0@0	1@0
1@0	7@0
2@0	8@0
3@0	5@0
4@0	9@0
5@0	3@0
6@0	0@0
7@0	2@0
8@0	6@0
9@0	4@0

2. After update (update row 5@0 with 10 → 42), ◀ ≡ changed:

s'

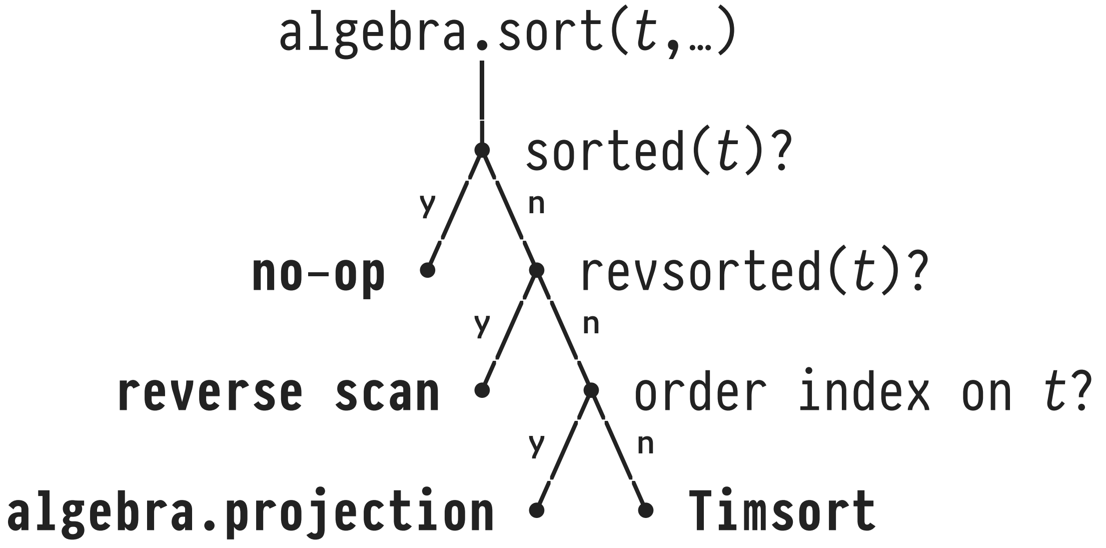
head	tail	
0@0	40	
1@0	0	
2@0	50	
3@0	30	
4@0	50	
5@0	42	◀
6@0	50	
7@0	10	
8@0	10	
9@0	20	

oidx^s'

head	tail	
0@0	1@0	
1@0	7@0	
2@0	8@0	
3@0	9@0	◀
4@0	3@0	◀
5@0	0@0	◀
6@0	5@0	◀
7@0	2@0	
8@0	6@0	
9@0	4@0	

Tactical Optimization for algebra.sort

- algebra.sort aims to avoid actual sorting effort based on properties of BAT *t* and the presence of order indexes:



- If all else fails, apply in-memory sort algorithm **Timsort** (1993; hybrid of merge/insertion sort, run-aware).

Demonstrate the tactical optimization in algebra.sort (sort table sorted without and with a persistent order index):

```
-- ❶ Start MonetDB server with `--algorithms` to observe tactical optimization decisions
$ pwd
/Users/grust/teach/SS18/DB2/course/MonetDB
$ mserver5 --dbpath=(pwd)/data/scratch --set monet_vault_key=(pwd)/data/scratch/.vaultkey  --algorithms

-- ❷ Prepare table sorted (as on slides), query with no order index

$ mclient -d scratch -l sql

  DROP TABLE IF EXISTS sorted;
  CREATE TABLE sorted (a text, s int);

  INSERT INTO sorted(a,s) VALUES
    ('a', 40),
    ('b',  0),
    ('c', 50),
    ('d', 30),
    ('e', 50),
    ('f', 10),
    ('g', 50),
    ('h', 10),
    ('i', 10),
    ('j', 20);

  EXPLAIN
    SELECT s.*
    FROM   sorted AS s
    ORDER BY s.s;

  [...]
  X_4 := sql.mvc();
  C_5:bat[:oid] := sql.tid(X_4, "sys", "sorted");
  X_18:bat[:int] := sql.bind(X_4, "sys", "sorted", "s", 0:int);
  X_24 := algebra.projection(C_5, X_18);
  (X_25, X_26, X_27) := algebra.sort(X_24, false, false); ⬅
  X_31 := algebra.projection(X_26, X_24);
  X_8:bat[:str] := sql.bind(X_4, "sys", "sorted", "a", 0:int);
  X_30:bat[:str] := algebra.projectionpath(X_26, C_5, X_8);
  [...]


  SELECT s.*
  FROM   sorted AS s
  ORDER BY s.s;
```

```
MonetDB server output:

  [...]
  #BATproject(l=tmp_1622,r=tmp_345)=tmp_41#10
  #BATmaterialize(335);
  #BATordered_rev: fixed norevsorted(1) for tmp_517#10 (1 usec)
  #BATgroup(b=tmp_517#10[int],s=NULL#0,g=NULL#0,e=NULL#0,h=NULL#0,subsorted=1): compare consecutive values
  [...]
  #BATidxsync: persisting orderidx 03/345.torderidx (2358 usec)
         ▲
     ⚠ MonetDB server may create order index on the fly and reuse it when
       query is repeated


-- ❸ Add persistent order index, repeat query

  ALTER TABLE sorted SET READ ONLY;
  CREATE ORDERED INDEX oidx_s ON sorted(s);

  -- EXPLAIN plan is unchanged: algebra.sort optimizes tactically

  SELECT s.*
  FROM   sorted AS s
  ORDER BY s.s;

  [...]
  #BATproject(l=tmp_3101#10-sorted-key,r=tmp_345#10[int])
  #BATproject(l=tmp_3101,r=tmp_345)=tmp_517#10
  #BATcheckorderidx: reusing persisted orderidx 229 ⬅
  #BATproject(l=tmp_1622#10-key,r=tmp_517#10[int])
  #BATproject(l=tmp_1622,r=tmp_517)=tmp_267#10 16us
  [...]
```

- Readable MAL plan:

```
sql.init();
sql := sql.mvc();

sorted :bat[:oid] := sql.tid(sql, "sys", "sorted");
s0     :bat[:int] := sql.bind(sql, "sys", "sorted", "s", 0:int);
s      :bat[:int] := algebra.projection(sorted, s0);

(s_ord_s, oidx_s, gidx_s) := algebra.sort(s, false, false);

a0     :bat[:str] := sql.bind(sql, "sys", "sorted", "a", 0:int);
a_ord_s:bat[:str] := algebra.projectionpath(oidx_s, sorted, a0);
```

Multi-Criteria ORDER BY

Multi-column ordering criteria require special treatment: algebra.sort(s) only receives single criterion s.

```
SELECT s.a, s.s
FROM   sorted AS s
ORDER BY s.s, s.a -- s₁ < s₂ ⇔ s₁.s < s₂.s ∨
                  --            (s₁.s = s₂.s ∧ s₁.a < s₂.a)
```

- 💡 Let algebra.sort(s) return *three* result BATs:
  1. $s^{ord(s)}$ (the ordered input s) ✓
  2. $oidx^{s}$ (order index) ✓
  3. $gidx^{s}$ (groups rows that agree on criterion s).

Multi-Criteria **ORDER BY:** Group Index **gidx**

$s^{ord(s)}$ ✓

head	tail
0@0	0
1@0	10
2@0	10
3@0	10
4@0	20
5@0	30
6@0	40
7@0	50
8@0	50
9@0	50

,

$oidx^s$ ✓

head	tail
0@0	1@0
1@0	7@0
2@0	8@0
3@0	5@0
4@0	9@0
5@0	3@0
6@0	0@0
7@0	2@0
8@0	6@0
9@0	4@0

,

$gidx^s$

head	tail	s =
0@0	0@0	0
1@0	1@0	10
2@0	1@0	10
3@0	1@0	10
4@0	2@0	20
5@0	3@0	30
6@0	4@0	40
7@0	5@0	50
8@0	5@0	50
9@0	5@0	50

3 output BATs

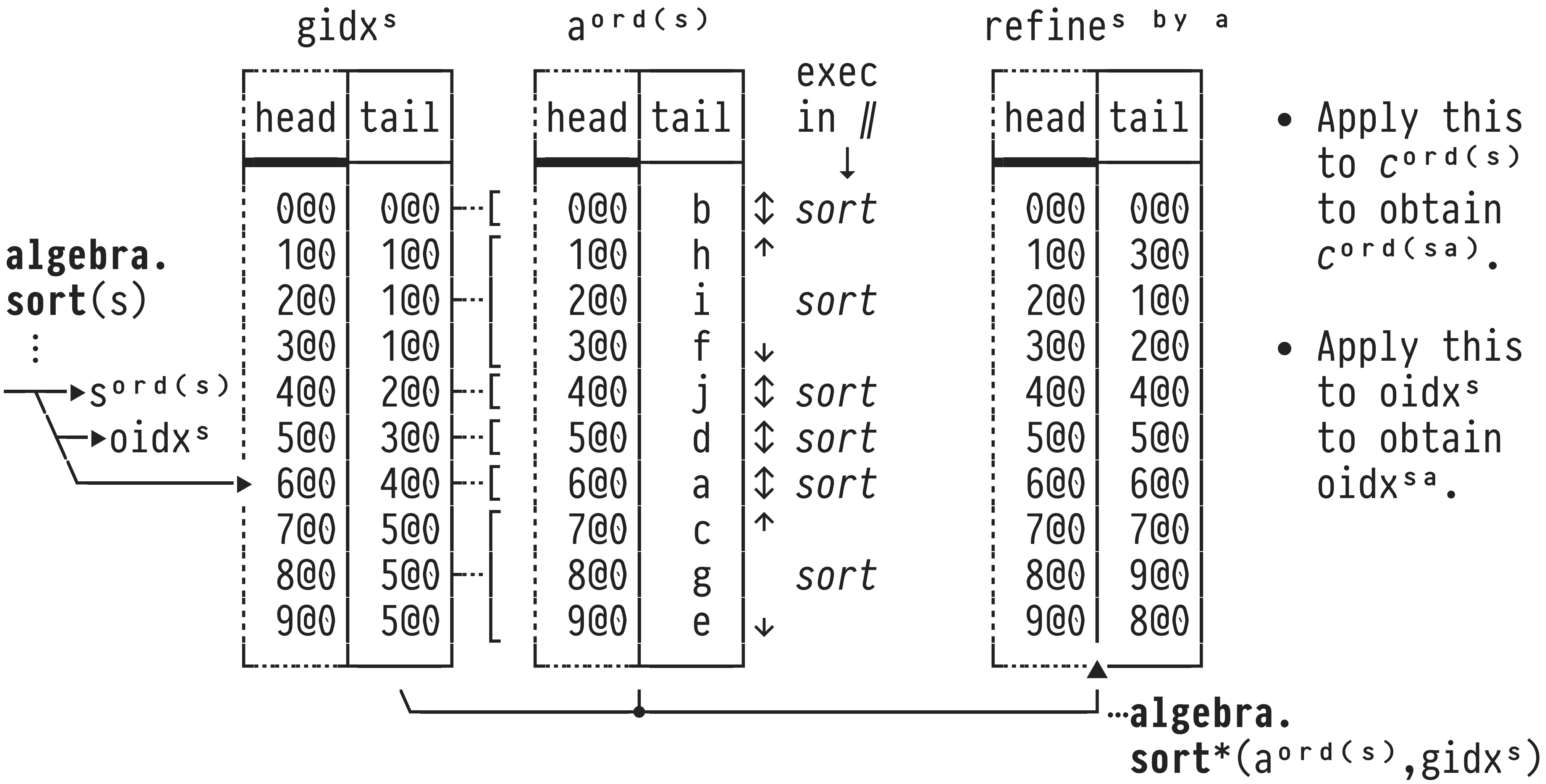
:= algebra.sort

s

head	tail
0@0	40
1@0	0
2@0	50
3@0	30
4@0	50
5@0	10
6@0	50
7@0	10
8@0	10
9@0	20

input BAT

Multi-Criteria ORDER BY s,a: Refine ORDER BY s by a



$gidx^{s}$

head	tail
0@0	0@0
1@0	1@0
2@0	1@0
3@0	1@0
4@0	2@0
5@0	3@0
6@0	4@0
7@0	5@0
8@0	5@0
9@0	5@0

$a^{ord(s)}$

head	tail
0@0	b
1@0	h
2@0	i
3@0	f
4@0	j
5@0	d
6@0	a
7@0	c
8@0	g
9@0	e

$refine^{s\ by\ a}$

head	tail
0@0	0@0
1@0	3@0
2@0	1@0
3@0	2@0
4@0	4@0
5@0	5@0
6@0	6@0
7@0	7@0
8@0	9@0
9@0	8@0

- Apply this to $c^{ord(s)}$ to obtain $c^{ord(sa)}$.
- Apply this to $oidx^{s}$ to obtain $oidx^{sa}$.

algebra.sort*: actual MAL code uses multi-argument algebra.sort:

$$(\underbrace{a^{ord(sa)}}, \underbrace{oidx^{sa}}, gidx^{sa}) := \text{algebra.sort}(\underbrace{a, oidx^{s}}, gidx^{s}, \text{false}, \text{false});$$

internally uses algebra.projection(a, oidx^{s}) to derive $a^{ord(s)}$

$oidx^{sa} \equiv$ algebra.projection(refine$^{s \text{ by } a}$, oidx^{s})

$a^{ord(sa)} \equiv$ algebra.projection(oidx^{sa}, a) = algebra.projectionpath(refine$^{s \text{ by } a}$, oidx^{s}, a)

$oidx^{sa}$

head	tail
0@0	1@0
1@0	5@0
2@0	7@0
3@0	8@0
4@0	9@0
5@0	3@0
6@0	0@0
7@0	2@0
8@0	4@0
9@0	6@0

- MAL plan for multi-criteria ORDER BY s,a:

```
EXPLAIN
  SELECT s.a, s.s
  FROM   sorted AS s
  ORDER BY s.s, s.a;

[...]
sql.init();
sql := sql.mvc();
sorted:bat[:oid] := sql.tid(sql, "sys", "sorted");
a0    :bat[:str] := sql.bind(sql, "sys", "sorted", "a", 0:int);
a     :bat[:str] := algebra.projection(sorted, a0);
# ❶ ... ORDER BY s.s
s0    :bat[:int] := sql.bind(sql, "sys", "sorted", "s", 0:int);
s     :bat[:int] := algebra.projection(sorted, s0);
(s_ord_s, oidx_s, gidx_s) := algebra.sort(s, false, false);
# ❷ refine ... ORDER BY s.s, s.a
(a_ord_sa, oidx_sa, gidx_sa) := algebra.sort(a, oidx_s, gidx_s, false, false); # ⬅
s_ord_sa:bat[:int] := algebra.projection(oidx_sa, s);
a_ord_sa:bat[:str] := algebra.projection(oidx_sa, a);

io.print(a_ord_sa, s_ord_sa);
[...]
```