

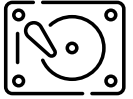
DB 2

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
```

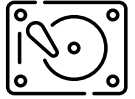
```
❷ SELECT DISTINCT i.c  
   FROM indexed AS i
```

```
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.

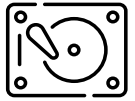
Sorting Takes Time



- 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 introduces 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

2 : External Merge Sort



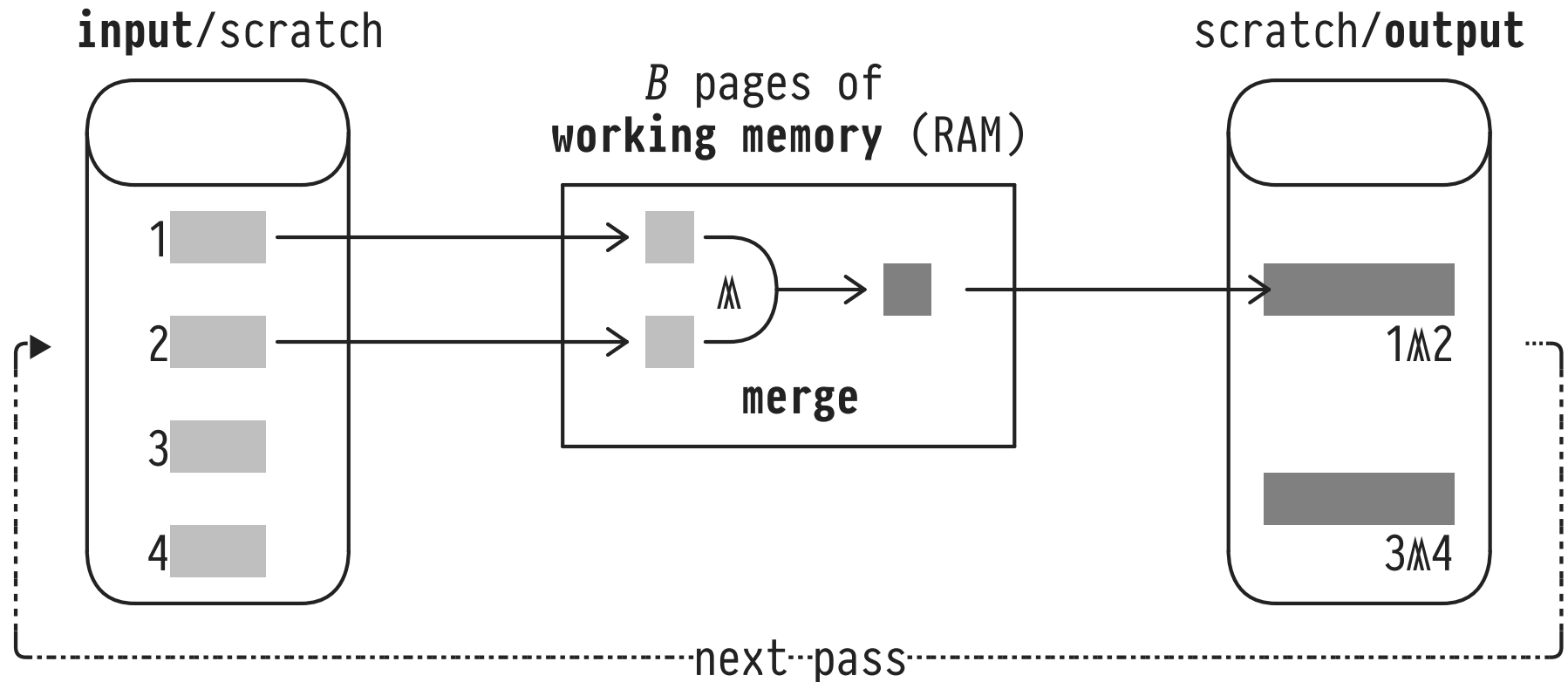
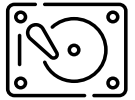
Now assume the following typical scenario:



- 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): $\geq 2 \times N$ blocks.

External Merge Sort can sort heap files of any size as long as $B \geq 3$ 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 ($B = 3$)

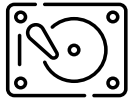


 input run \setminus sorted
 merged run \int
 \bowtie $(B-1)$ -way merge

$$T = \text{input run} \cup \dots \cup \text{input run} = \text{merged run} \cup \dots \cup \text{merged run}$$

$$(B-1) \times |\text{input run}| = |\text{merged run}|$$

External Merge Sort



ExternalMergeSort(T, B):

$N \leftarrow \text{\#pages of } T$;

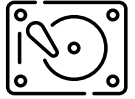
$R \leftarrow \lceil N/B \rceil$; } R : current number of runs

split input T into R partitions p_i of B pages; }
for each $i \in 1 \dots R$ } pass 0
| run $r_i \leftarrow$ in-memory sort of p_i ;

while $R > 1$
| $R \leftarrow \lceil R / (B-1) \rceil$;
| **for each** $i \in 1 \dots R$ } passes 1, 2, ...
| | \mathbb{M} : merge next $B-1$ runs into one run;

return single sorted run;

External Merge Sort: Passes and I/O Operations

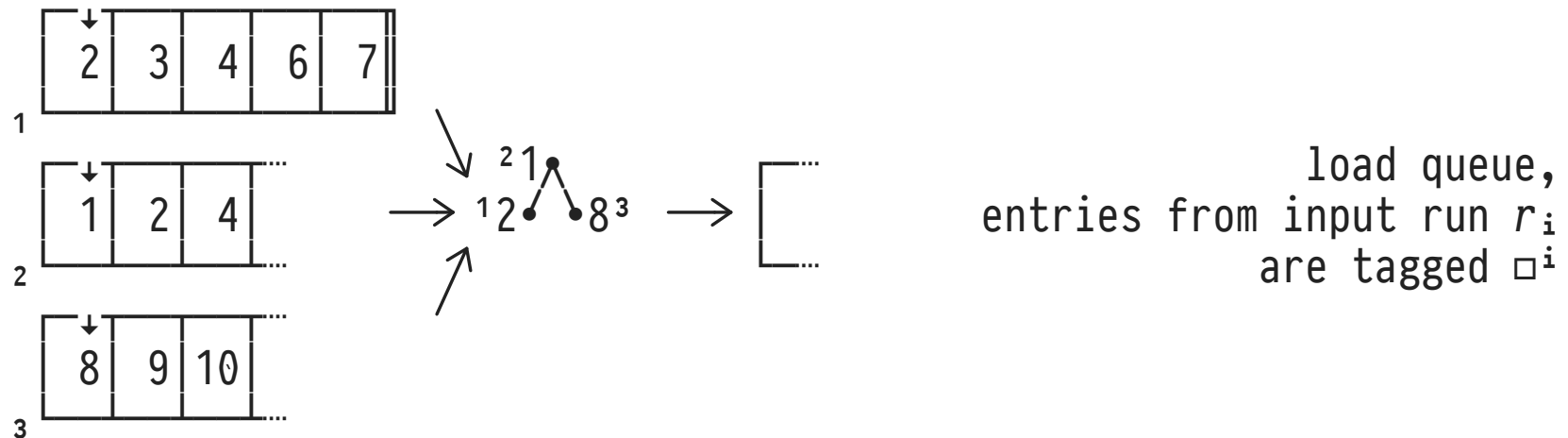
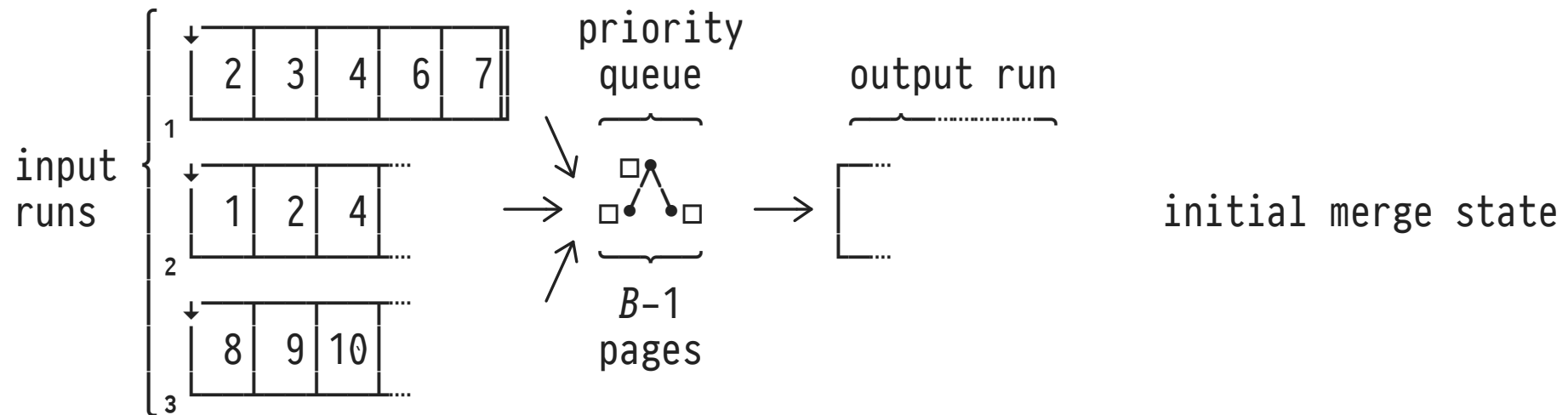
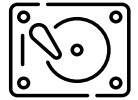


pass	input: #runs	input: run size	output: #runs	output: run size
1	$\lceil N/B \rceil$	B	$\lceil N/B \rceil / (B-1)$	$B \times (B-1)$
2	$\lceil N/B \rceil / (B-1)$	$B \times (B-1)$	$\lceil N/B \rceil / (B-1)^2$	$B \times (B-1)^2$
3	$\lceil N/B \rceil / (B-1)^2$	$B \times (B-1)^2$	$\lceil N/B \rceil /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

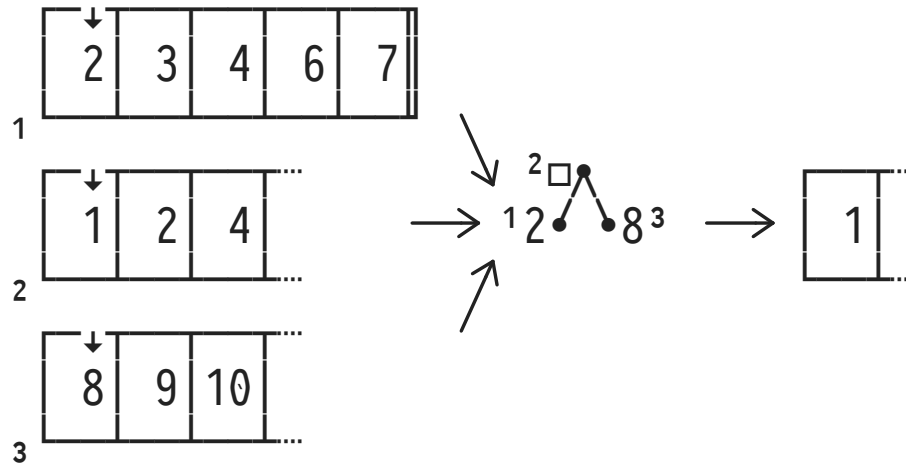
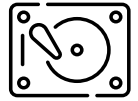
- In each pass:
 $N = \text{input} (\#runs \times \text{run size}) = \text{output} (\#runs \times \text{run size})$.
 - Each pass performs $2 \times N$ I/O operations.
- Passes required by External Merge Sort with B buffers:

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

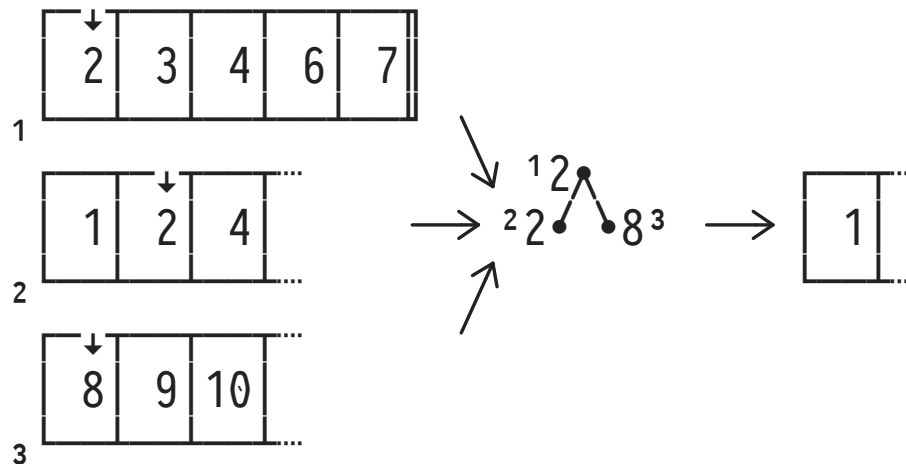
3 : $(B-1)$ -Way Merge (Passes 1,2,...)



$(B-1)$ -Way Merge (Passes 1,2,...)

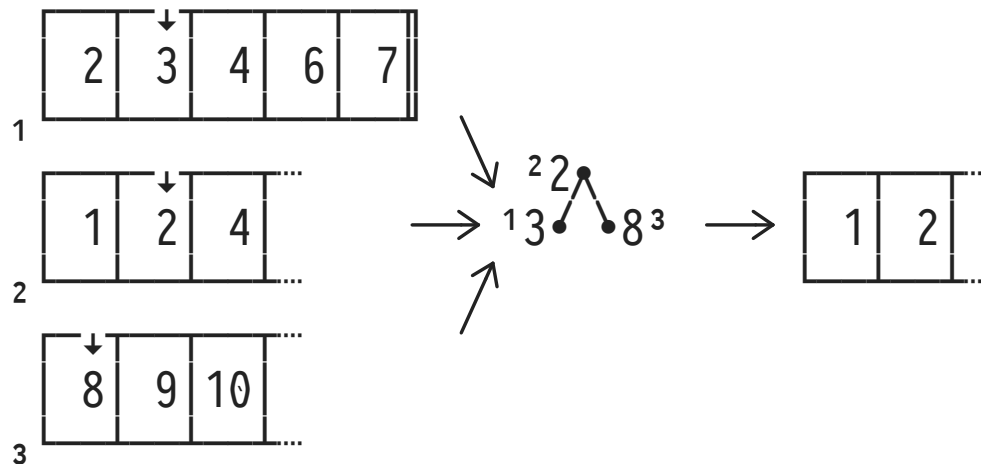
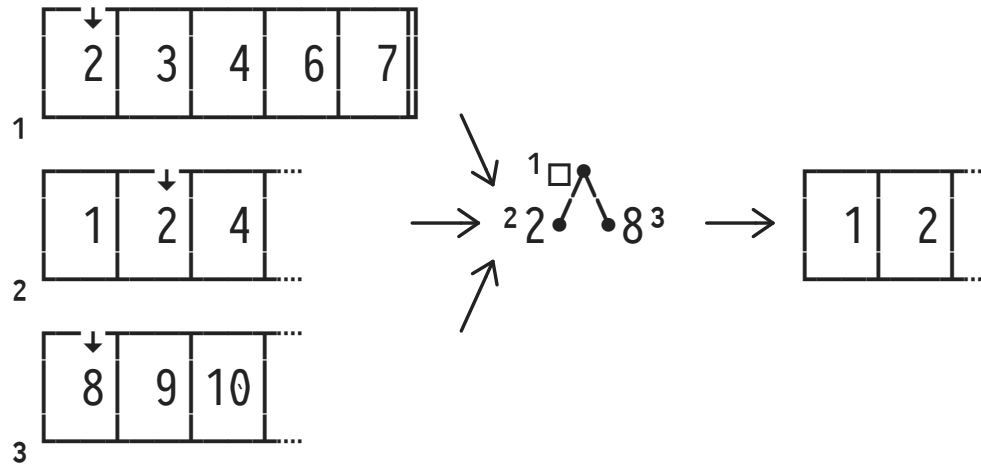
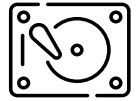


queue head \rightarrow output run



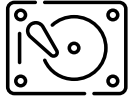
refill queue from input run

(B-1)-Way Merge (Passes 1,2,...)



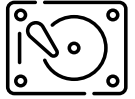
⋮

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

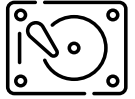
Disk seeking will need 0.1 minutes, while 0.2 minutes is spent on I/O itself.

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



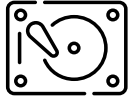
- The *initial number of runs created in pass 0* influence overall sort performance:

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

runs created in pass 0

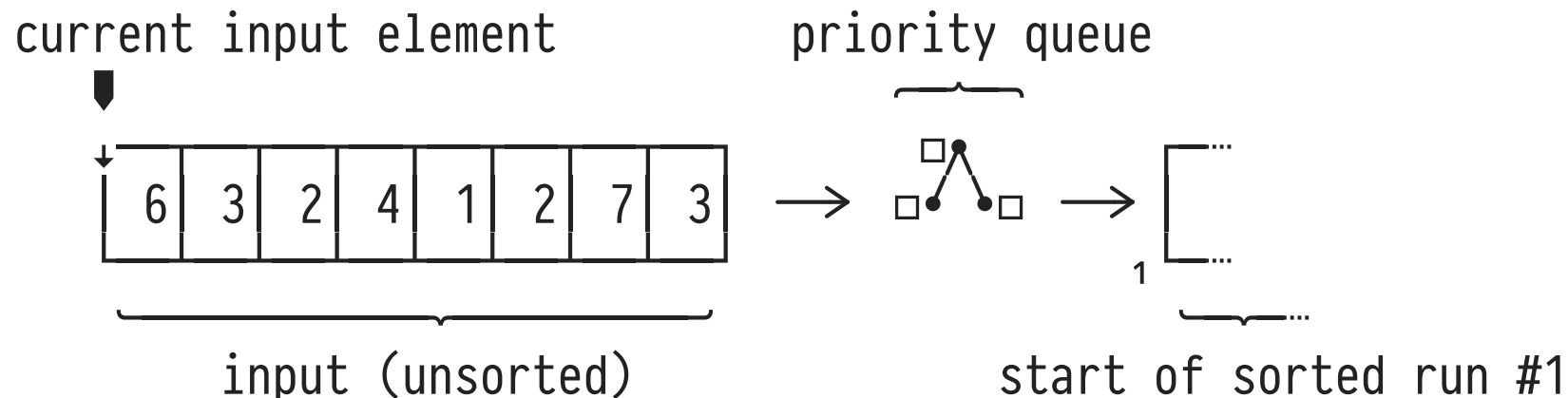
- **Q:** Given only B buffers, can we create sorted runs *longer than B pages*?
 - **A:** Yes! In pass 0, use **Replacement Sort** (instead of QuickSort, for example).

Replacement Sort

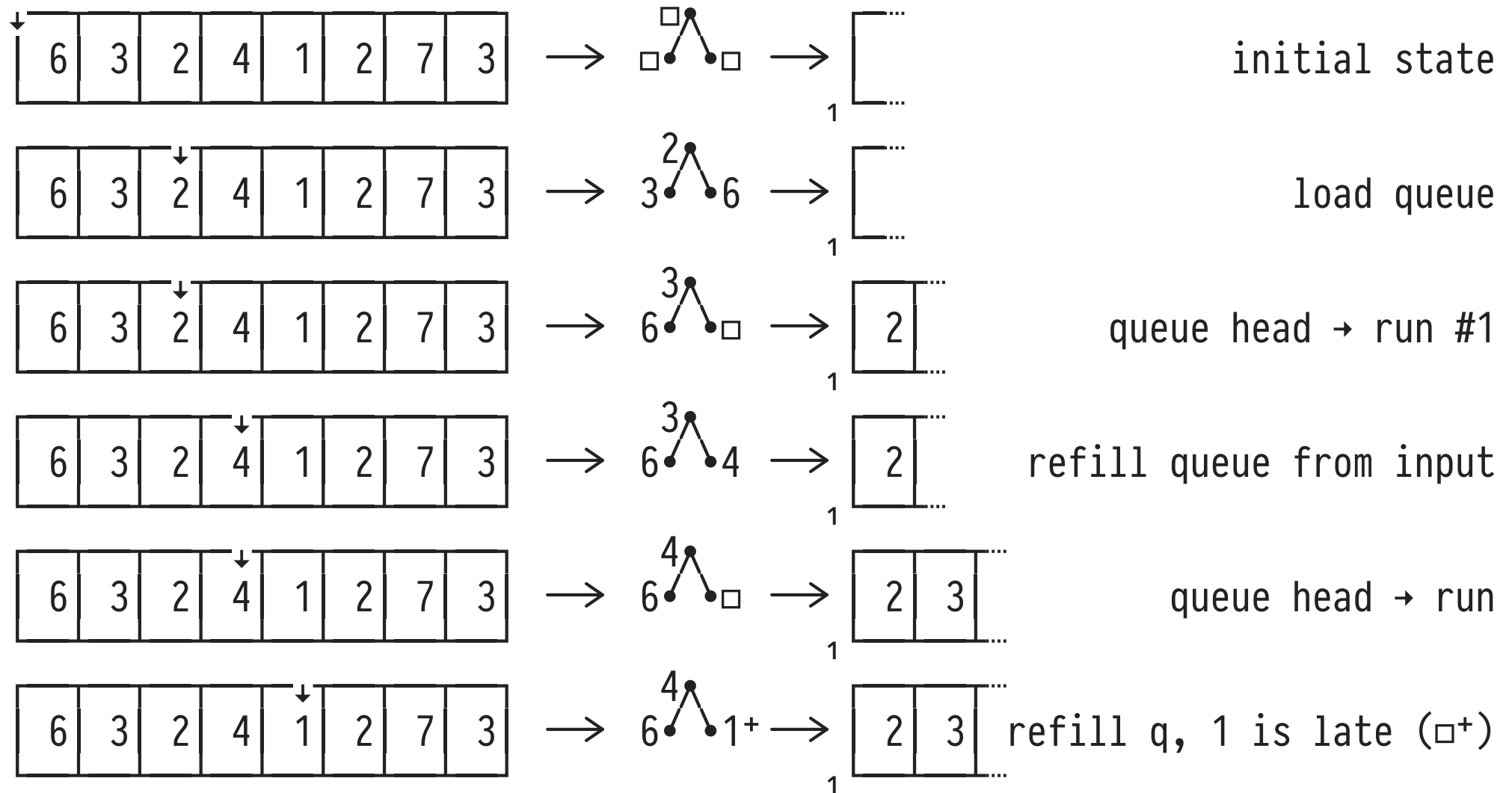
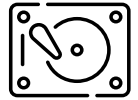


Again, use $B-1$ buffer pages to set up a **priority queue**:

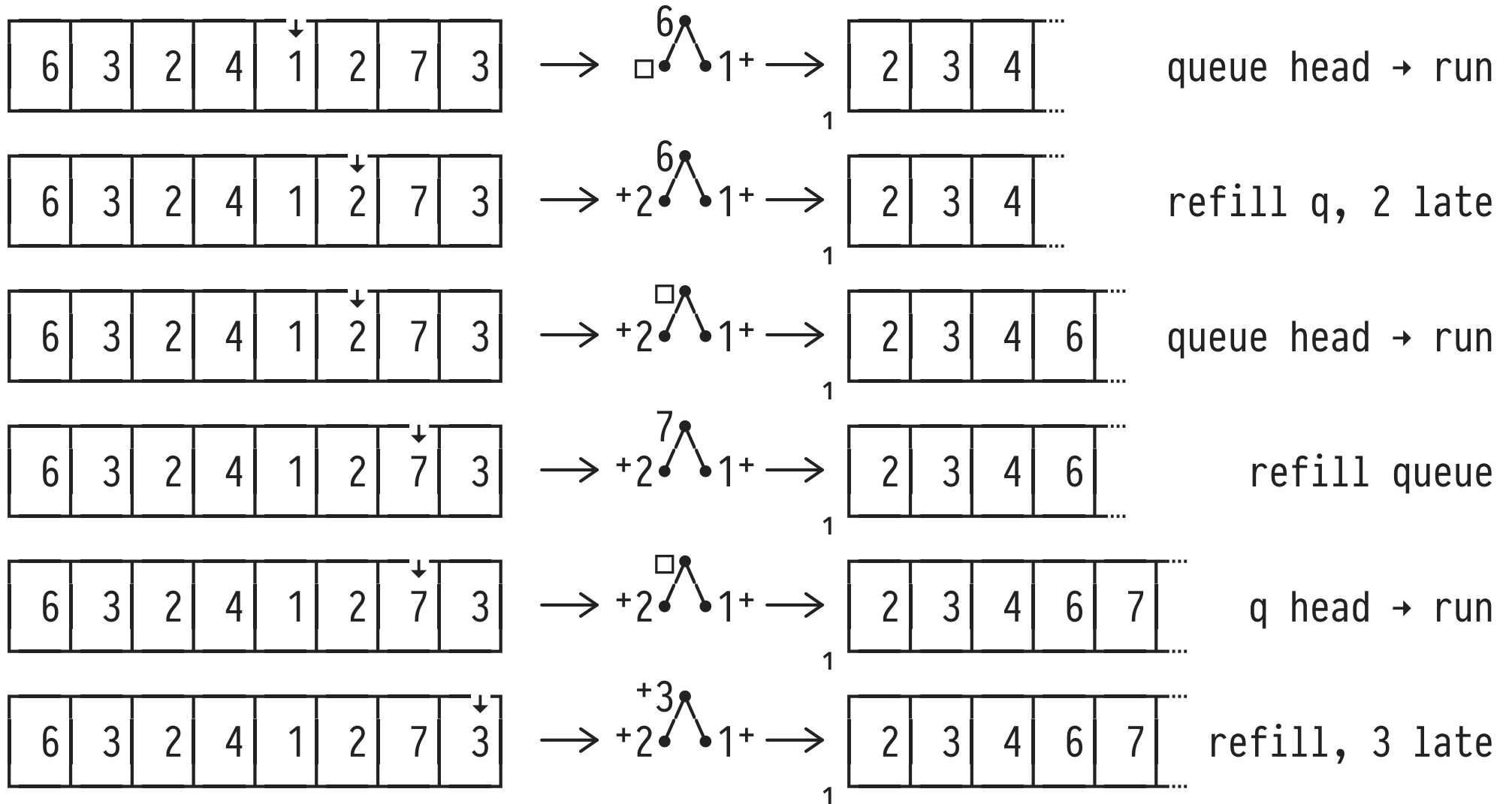
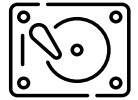
1. Elements arriving too late for inclusion in current run are marked (\square^+) and receive lower priority.
2. When all elements in queue are marked, close the current run, unmark all elements, open a new run.



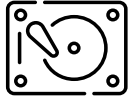
Replacement Sort ($B = 4$)



Replacement Sort ($B = 4$)

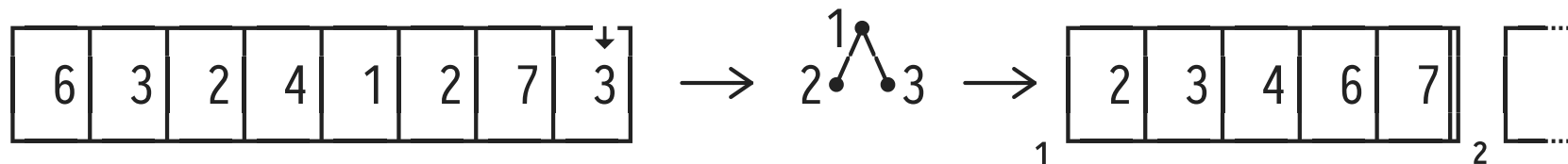


Replacement Sort ($B = 4$)



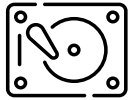
All entries in queue are late (\square^+):

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



- Replacement Sort produces runs of length $\approx 2 \times (B-1) > B$ (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).

5 | Q₁₀: Grouping

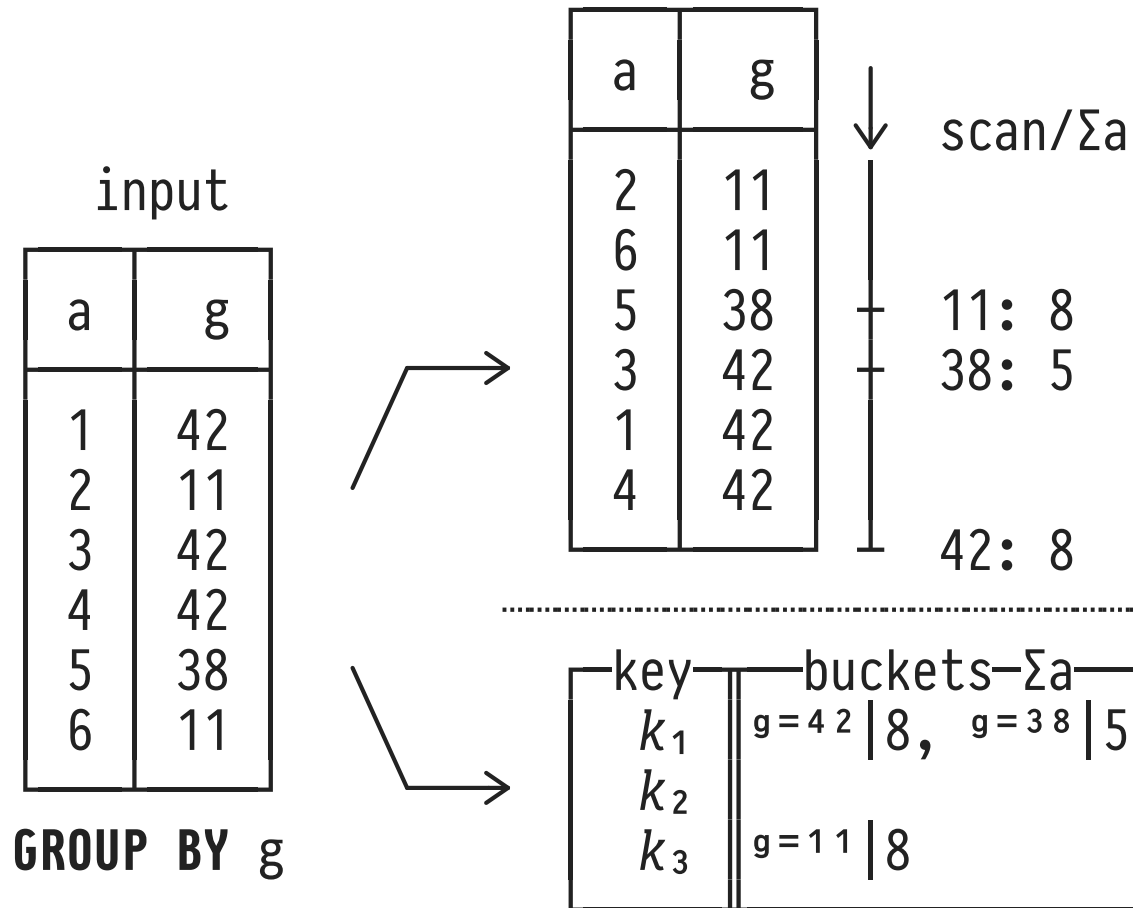
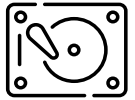


Grouping coarsens the granularity of data processing
(individual rows ↘ groups of rows):

```
2 SELECT g.g, SUM(g.a) AS s -- out: 104 groups (aggregates)
   FROM grouped AS g      -- in: 106 rows
1 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



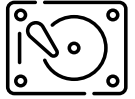
- scan sorted table for group boundaries
- aggregate while scanning

Sorting

Hashing

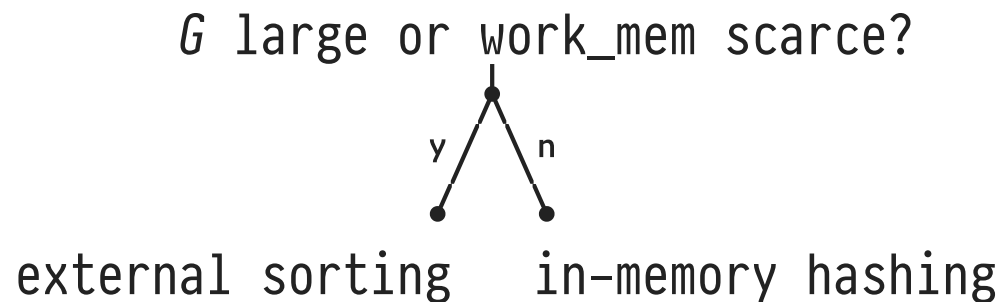
- hash buckets hold grouping criterion and aggregate value

Grouping: Sorting vs. Hashing



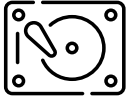
PostgreSQL plans for sorting vs. hashing based on

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` $\Rightarrow G \leq 2$ not understood by PostgreSQL).
 - \Rightarrow Overestimate G conservatively, use sorting.

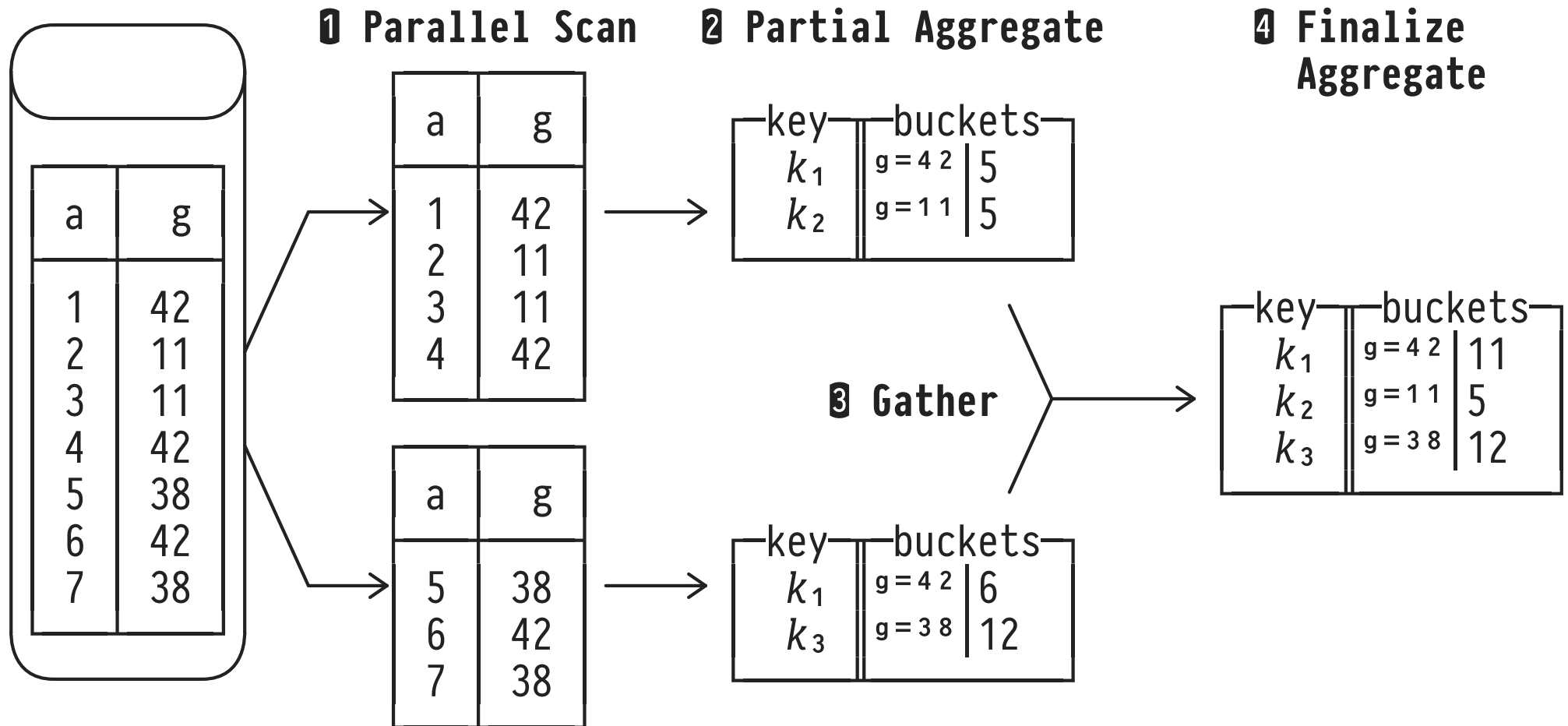
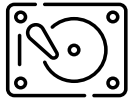
6 | Parallel Grouping and Aggregation



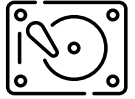
Grouping and aggregation are query operations that are straightforward to **parallelize**:

- 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))



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 ...)

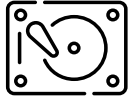
Workers Planned: 2 ◀ *//ism degree: 3 (2 worker + 1 leader)*

-> **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



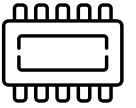
- Parallel evaluation of aggregate *AGG* depends on the **distributivity** over \sqcup (bag union):

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

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

$$\underbrace{SUM(X \sqcup Y)}_{\text{distribute work}} = SUM(\underbrace{\{SUM(X)\}}_{\text{partial aggregates}} \sqcup \underbrace{\{SUM(Y)\}}_{\text{partial aggregates}}) = SUM(X) \overset{\uparrow}{+} SUM(Y) \quad \text{finalize}$$

7 | Q₉: Sorting in MonetDB

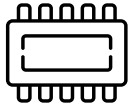


```
CREATE TABLE sorted (a text, s int);  
:  
SELECT s.a, s.s  
FROM   sorted AS s  
ORDER BY s.s [, s.a] -- single- or multi-column criteria
```

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.

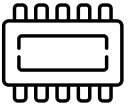
Recall: Order Indexes (**ORDER BY s.s**)



a			s			oidx ^s			a ^{ord(s)} s ^{ord(s)}		
head	tail	tail	head	tail	tail	head	tail	tail	head	tail	tail
0@0	a	40	0@0	1@0		0@0	b	0	0@0	b	0
1@0	b	0	1@0	7@0		1@0	h	10	1@0	h	10
2@0	c	50	2@0	8@0		2@0	i	10	2@0	i	10
3@0	d	30	3@0	5@0		3@0	f	10	3@0	f	10
4@0	e	50	4@0	9@0		4@0	j	20	4@0	j	20
5@0	f	10	5@0	3@0		5@0	d	30	5@0	d	30
6@0	g	50	6@0	0@0		6@0	a	40	6@0	a	40
7@0	h	10	7@0	2@0		7@0	c	50	7@0	c	50
8@0	i	10	8@0	6@0		8@0	g	50	8@0	g	50
9@0	j	20	9@0	4@0		9@0	e	50	9@0	e	50

...algebra.
projection(oidx^s, ·)

Order Indexes on the Fly: **algebra.sort**



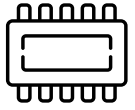
EXPLAIN

```
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);
(sord(s), oidxs, gidxs)      desc ↘ ↙ stable
                               := algebra.sort(s, false, false);
a0      :bat[:str] := sql.bind(sql, "sys", "sorted", "a", ...);
a       :bat[:str] := algebra.projection(sorted, a0);
aord(s) :bat[:str] := algebra.projectionpath(oidxs, sorted, a0);

io.print(aord(s), sord(s));
```

Persistent Order Indexes

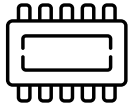


If sorting is central to the query workload, create a **persistent order index** that is immediately applicable:

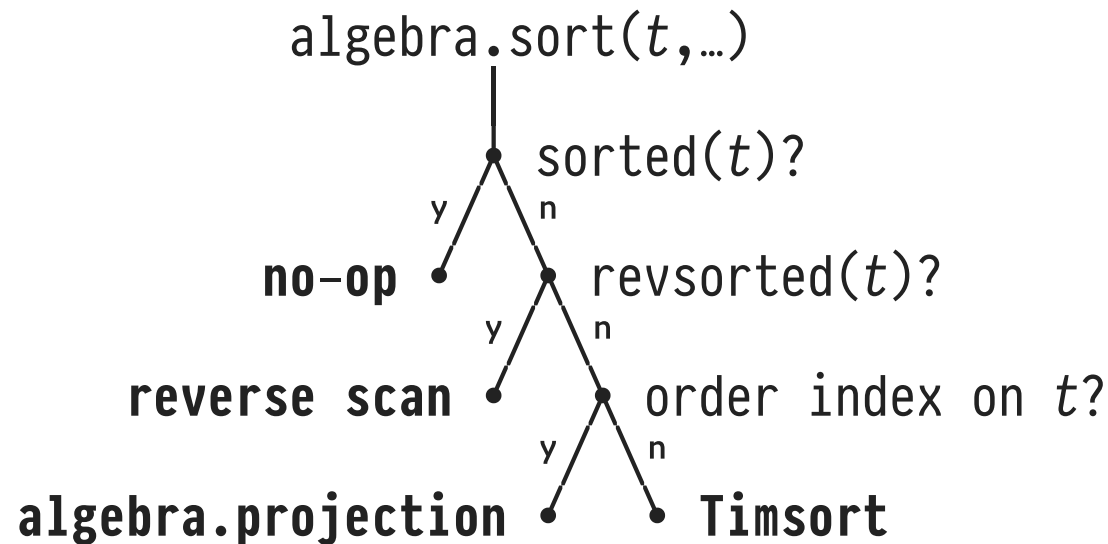
```
ALTER TABLE sorted SET READ ONLY;      -- ⚠
```

```
CREATE ORDERED INDEX oidxs 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.

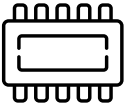


- `algebra.sort` aims to avoid actual sorting effort based on properties of BAT `t` and the presence of order indexes:



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

8 | Multi-Criteria ORDER BY

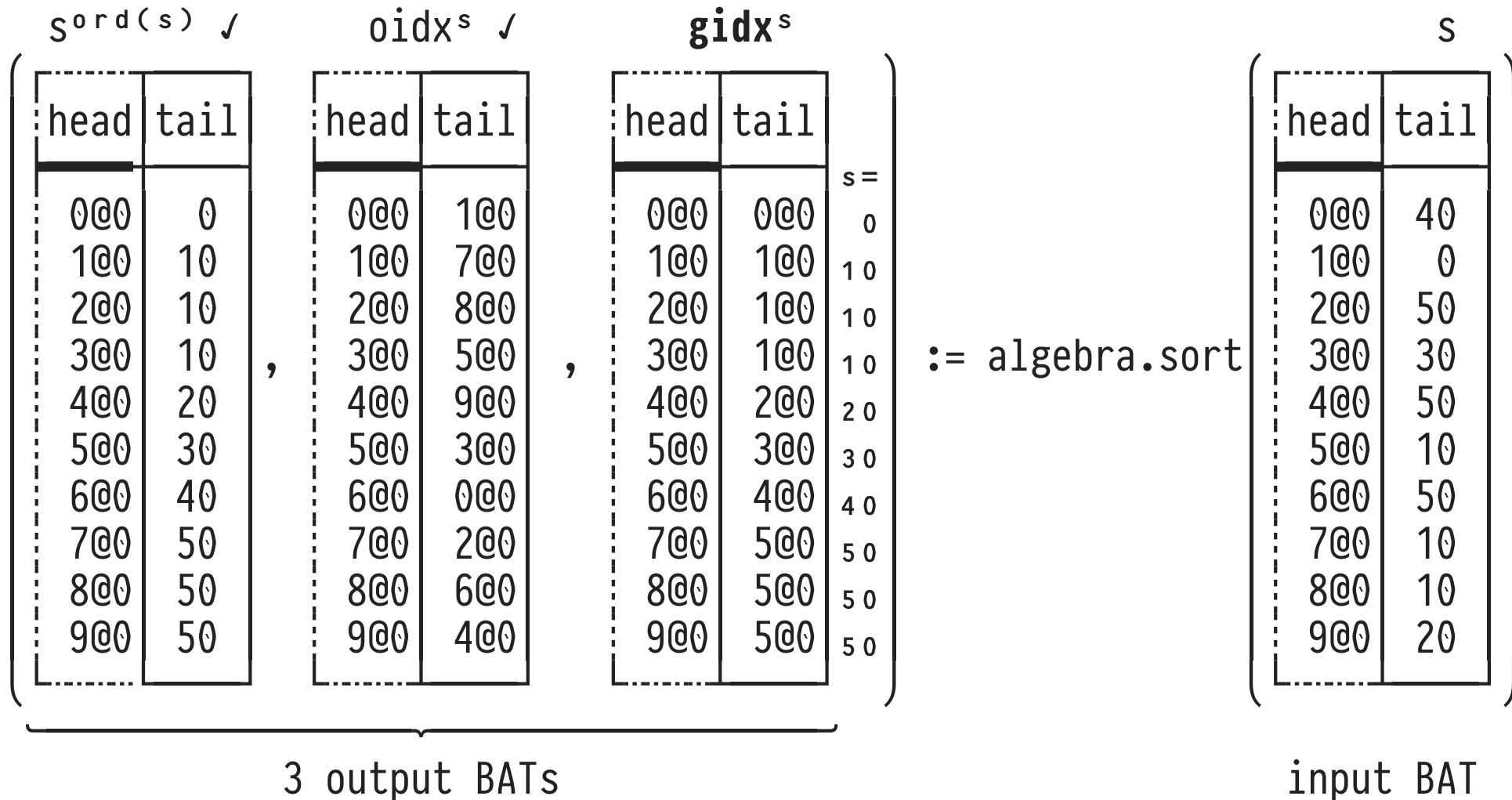
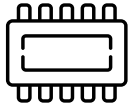


Multi-column ordering criteria require special treatment:
`algebra.sort(s)` only receives single criterion `s`.

```
SELECT s.a, s.s
FROM   sorted AS s
ORDER BY s.s, s.a --  $s_1 < s_2 \Leftrightarrow s_1.s < s_2.s \vee$ 
                  --  $(s_1.s = s_2.s \wedge s_1.a < s_2.a)$ 
```

- 💡 Let `algebra.sort(s)` return *three* result BATs:
 1. `sord(s)` (the ordered input `s`) ✓
 2. `oidxs` (order index) ✓
 3. `gidxs` (groups rows that agree on criterion `s`).

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



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

