

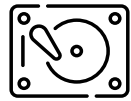
# DB 2

---

12 – Joins

Summer 2022

Torsten Grust  
Universität Tübingen, Germany



## 1 : $Q_{1,1}$ : One-to-Many Joins

---

**Join ( $\bowtie$ )** is a core operation in query processing: given two tables,<sup>1</sup> form all pairs of related rows.

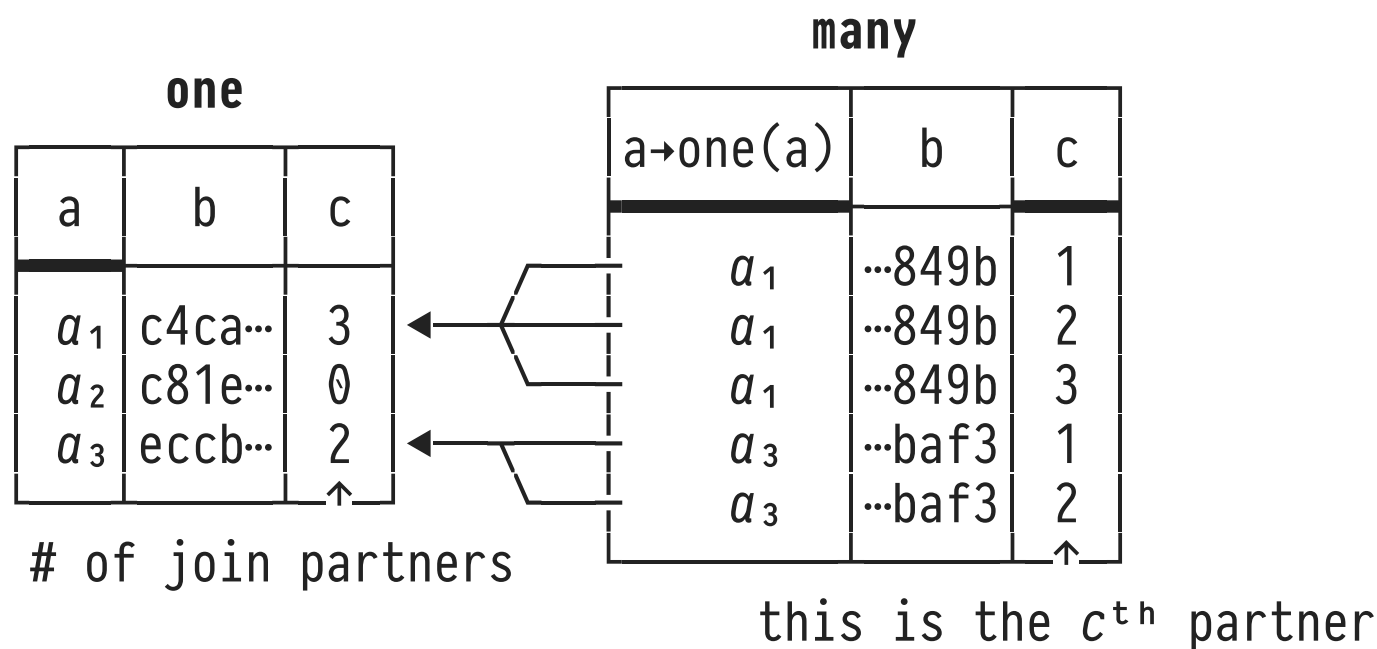
```
SELECT o.a, o.b AS b1, m.b AS b2, m.c
FROM   one  AS o,           -- one o may relate to many m:
       many AS m           -- [one]-(0,*)-<R>-(1,1)-[many]
WHERE  o.a = m.a
```

- A **one** row relates to  $0 \dots n$  rows of **many**: *1:n relationship*.
  - $\Rightarrow$  Join size  $\in \{0, \dots, n \times |\text{one}|\}$  rows. Largest possible result if  $n = |\text{many}|$  (Cartesian product).

<sup>1</sup> Note: the left and right tables may indeed be the *same* table. This is then coined a **self-join**.



# A Sample One-to-Many Relationship (Playground)



- Join predicates:
  - $\text{one.a} = \text{many.a}$  (index-supported)
  - $\text{md5}(\text{one.a}) = \text{one.b} || \text{many.b}$  ( $||$ : string concat)



## PostgreSQL: Join Algorithms

---

RDBMSs choose between several **join algorithms** based on

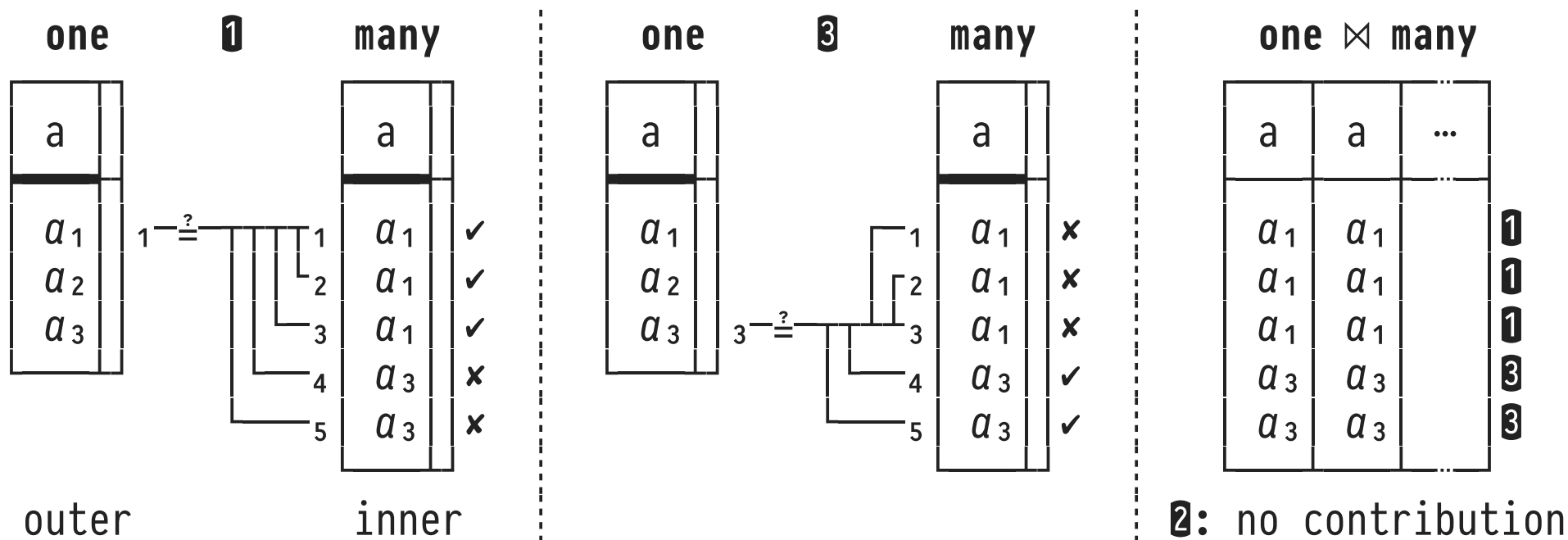
- the join **predicate type** (equi-join vs.  $\theta$ -join)
- the existence of **indexes** on the join predicate columns,
- the availability of **working memory**, or
- **interesting sort orders** of join inputs *and* output:

Join Algorithm	Characteristic
Nested Loop Join	processes any $\theta$ , can benefit from index support
Hash Join	fast equi-joins if plenty working memory available
Merge Join	requires sorted input, produces sorted output

PostgreSQL implements all three kinds of join algorithms.



## 2 : Nested Loop Join (NLJ, NL)



- Iterate ①...③ over rows of **outer table** (here: **one**) once.
  - For every outer row, iterate over **inner table**.
- Performs  $|outer| \times |inner|$  join predicate evaluations.



## Nested Loop Join (NL $\bowtie$ ) — “The Fallback”

```
NLJ(outer,inner, $\theta$ ):  
   $j = \phi$ ;  
  for  $o \in \textit{outer}$   
    for  $i \in \textit{inner}$   
      if  $o \theta i$   
        append  $\langle o, i \rangle$  to  $j$ ;  
  return  $j$ ;
```

- No restrictions regarding  $\theta \in \{=, <, \leq, <>, \dots\}$ . 👍
- No restrictions regarding sort order of *outer/inner*. 👍
- Preserves sort order of *outer*. 👍
- Indexes on *outer/inner* are ignored. 👎
- Benefits if *inner* can be iterated over quickly (e.g., materialized and/or fits into database buffer).



## Block Nested Loop Join (BNL⌘)

BlockNLJ(*outer*, *inner*,  $\theta$ ):

$j = \phi$ ;

**foreach** block (of size  $b_o$ )  $bo \in outer$

**foreach** block (of size  $b_i$ )  $bi \in inner$

**for**  $o \in bo$

**for**  $i \in bi$

**if**  $o \theta i$

                    append  $\langle o, i \rangle$  to  $j$ ;

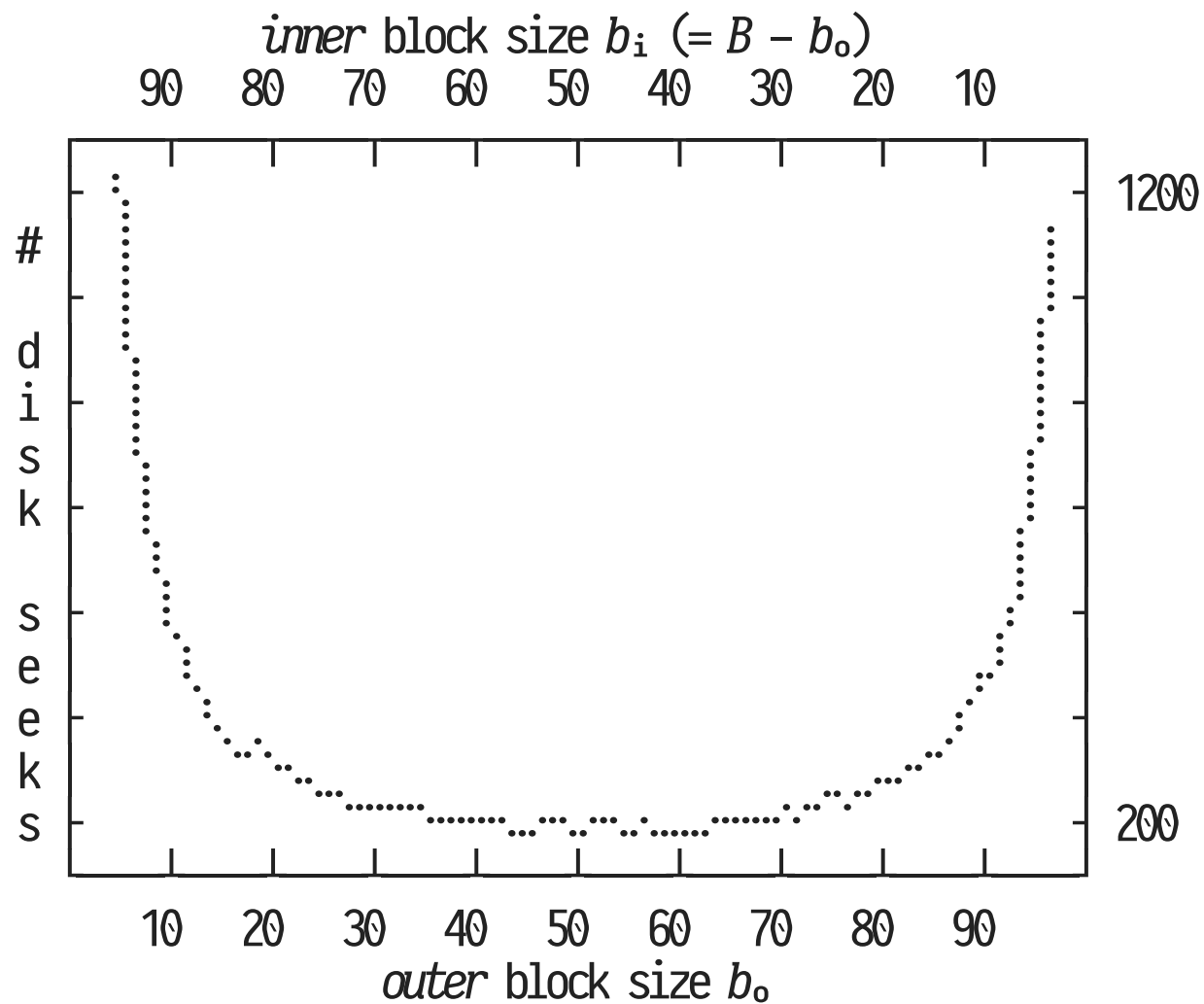
} entirely performed  
inside the buffer

**return**  $j$ ;

- Perform blocked I/O on *outer/inner*: less disk seeks. 👍
  - # seeks on *outer*:  $\lceil |outer|/b_o \rceil$ .
  - # seeks on *inner*:  $\lceil |outer|/b_o \rceil \times \lceil |inner|/b_i \rceil$ .



## Sharing a Buffer of Size $B = 100$ Slots







## NL⋈: Materialization of the Inner Input

---

The inner NL⋈ input is scanned [*outer*/*b<sub>o</sub>*] times (see PostgreSQL **EXPLAIN** plans: ... *loops=n* ...).

- 💡 Plan operator **Materialize**:
  1. **Evaluates its subplan once, saves rows** in working memory or temporary file (“tuple store”).
  2. Can scan tuple store more quickly than regular heap file pages (*e.g.*, no *xmin/xmax* checking).

```

                                QUERY PLAN
                                ───────────
      ∴
      -> Materialize (cost=...) (actual time=... loops=n)
            └-> Subplan (cost=...) (actual time=... loops=1)
                  └──────────┘

```



### 3 : Index Nested Loop Join (INL⋈)

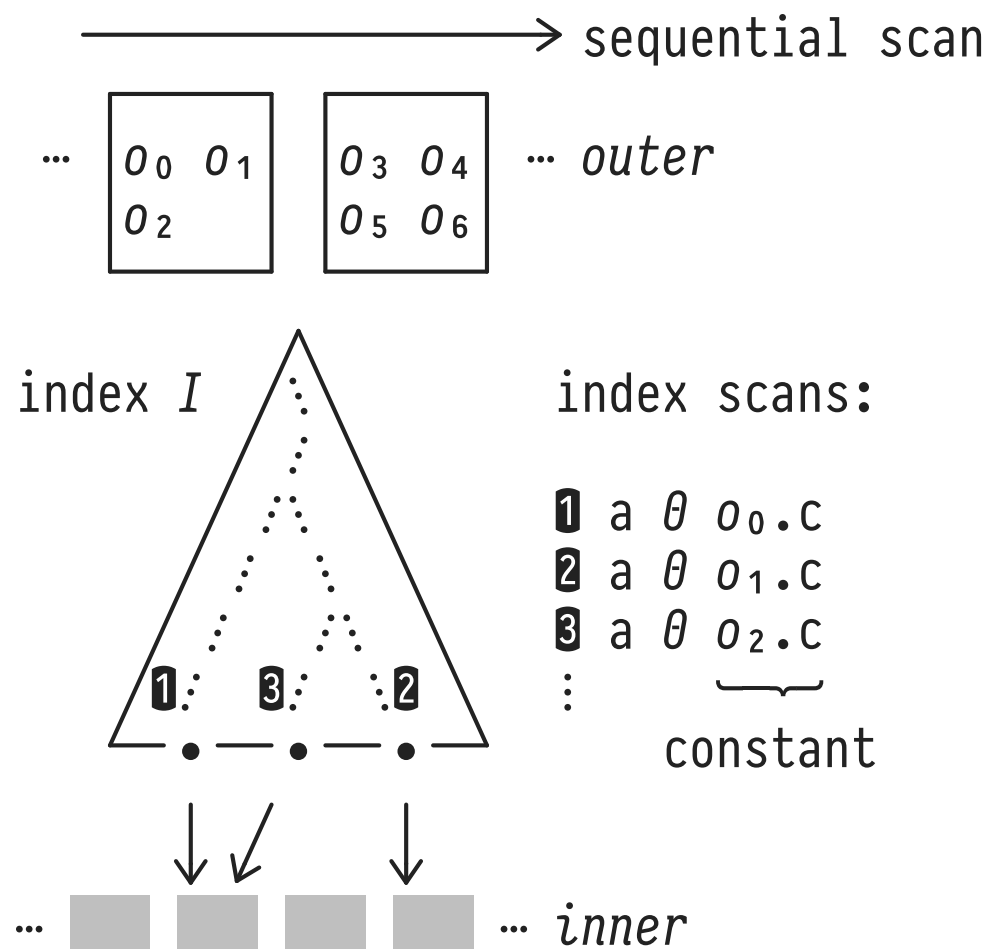
NL⋈ may be sped up considerably if the **|outer|** scans of *inner* can be turned into **|outer| index scans on inner**:

```
IndexNLJ(outer, inner,  $\theta$ ):
   $j = \phi$ ;
  for  $o \in outer$ 
    for  $i \in \text{Index[Only]Scan}(I, \underbrace{o \ \theta \ \square}_{\text{index condition}})$ 
      append  $\langle o, i \rangle$  to  $j$ ;
  return  $j$ ;
```

- *NB.* In each of the **|outer|** invocations of **IndexScan**, row *o* essentially is a constant.
  - Index *I* on *inner* must be able to support predicate  $\theta$ .
- The index scan only delivers actual partners for *o*. 👍



# Index Nested Loop Join (INL $\bowtie$ )



```
CREATE INDEX I ON many
      USING btree (a);
```

```
SELECT *
FROM   one  AS o, -- outer
       many AS m  -- inner
WHERE  m.a θ o.c;
```



## 4 : Merge Join

---

Join algorithm **Merge Join** supports equality join predicates (“equi-joins”) of the form  $c_1 = c_r$ :<sup>2</sup>

1. left input *must* be **sorted** by  $c_1$ , right input *must* be **sorted** by  $c_r$ ,
2. left input scanned once in order, right input scanned once but must support repeated *re-scanning* of rows,
3. the **join output is sorted** by  $c_1$  (and thus  $c_r$ ).

**NB.** Merge Join's guaranteed output order can provide a true benefit during later query processing stages.

<sup>2</sup> Generalizations to predicates  $c_1 \theta c_r$  with  $\theta \in \{<, \leq, \dots\}$  have been defined but are seldomly found implemented in actual RDBMSs.



## Merge Join Ingredients

---

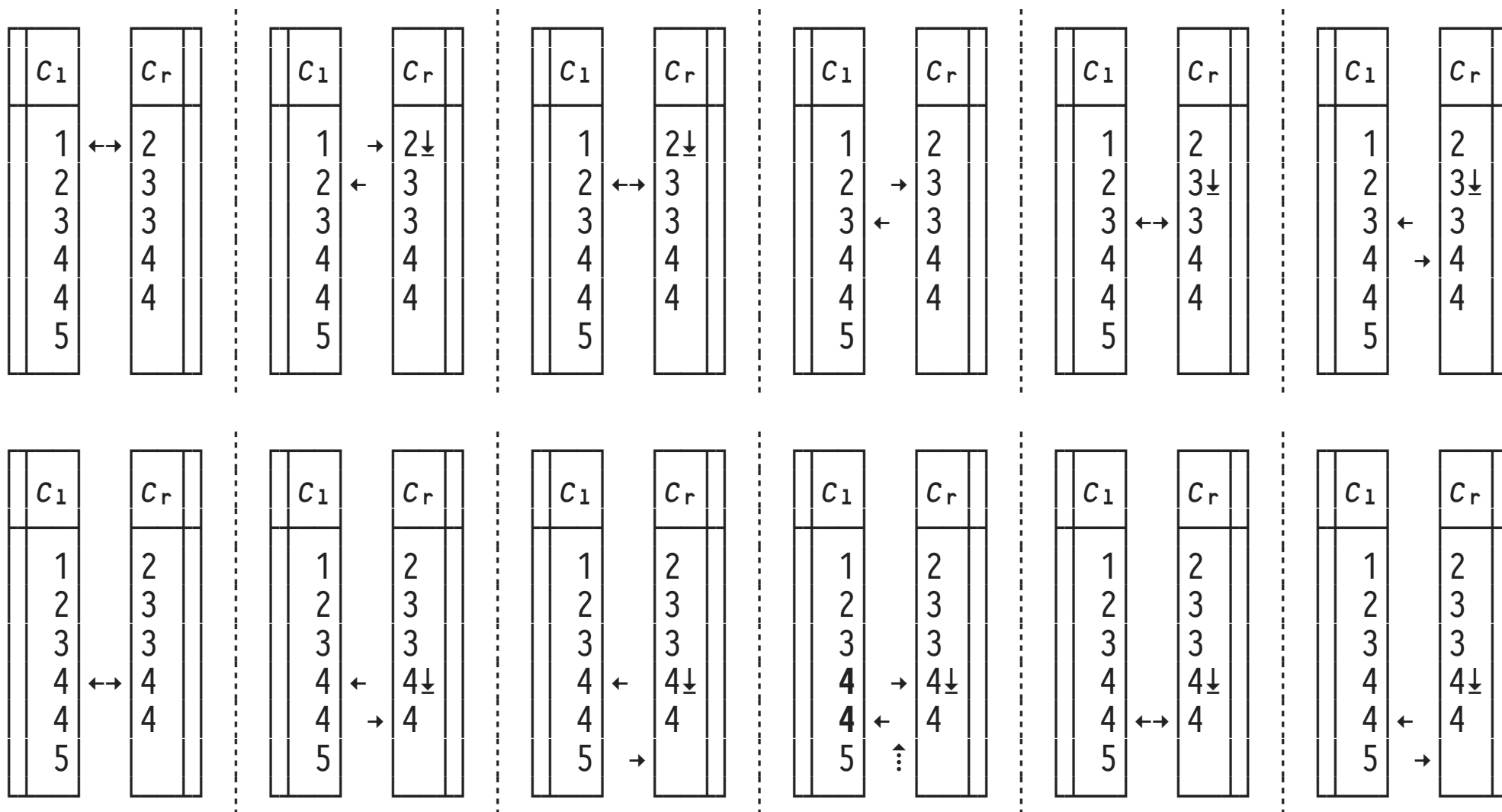
**Merge Join** performs synchronized forward ( $\equiv$  sorted) scans:<sup>3</sup>

- Maintain row pointers into left/right inputs ( $\leftarrow/\rightarrow$ ).
- Iterate:
  - Move row pointers forward in lock step:
    - If  $c_1 < c_r$ , advance  $\leftarrow$ . If  $c_1 > c_r$ , advance  $\rightarrow$ .
    - If  $c_1 = c_r$ , emit joined row.
  - If required, save current position ( $\downarrow$ ) of  $\rightarrow$  so that we can reset ( $\uparrow$ ) the scan of the right input back to  $\downarrow$ .
    - This resetting may lead to (limited) re-scanning of the right input.

<sup>3</sup> Arrow symbols  $\leftarrow$ ,  $\rightarrow$ ,  $\downarrow$ ,  $\uparrow$  refer to the illustration on the next slide. Only the join columns  $c_1$ ,  $c_r$  (of type `int`) are shown.



# Merge Join: Synchronized Scan Pointers





## Merge Join: Pseudo Code

```

MergeJoin(left, right,  $c_l$ ,  $c_r$ ):
   $j \leftarrow \emptyset$ ;
  while  $left \neq \perp \wedge right \neq \perp$  } reached end-of-table?
  | while  $left.c_l < right.c_r$  }
  | | advance left; } move scans forward
  | while  $left.c_l > right.c_r$  } in lock step
  | | advance right; }
  |  $\downarrow \leftarrow right$ ; } save current right pos
  | while  $left.c_l = \downarrow.c_r$  } scan repeating left group
  | |  $right \leftarrow \downarrow$ ; } reset right scan
  | | while  $left.c_l = right.c_r$ 
  | | | append  $\langle left, right \rangle$  to  $j$ ;
  | | | advance right;
  | | advance left;
  return  $j$ ;

```



## Merge Join: Sorted Inputs

---

**Merge Join** requires inputs sorted on  $c_1/c_r$ . Options:

1. Introduce **explicit Sort** plan operator below **Merge Join**.
2. Input is **Index Scan** with key column prefix  $c_1/c_r$ .<sup>4</sup>
3. Input table is (perfectly) **clustered** on  $c_1/c_r$ .
4. **Subplan** below **Merge Join** delivers rows in  $c_1/c_r$  **order**.

```

                                QUERY PLAN
Merge Join (cost=...) (actual time=... loops=n)
  -> 「 Subplan left (cost=...) (actual time=... loops=1)  」
      「
  -> 「 Subplan right (cost=...) (actual time=... loops=1)  」
      「
  」

```

<sup>4</sup> Q: Will **Bitmap Index/Heap Scan** also fit the bill here?







## Interesting Orders

---

If a subplan *delivers* rows in a well-defined **interesting order**, the *downstream* query plan may

- save an explicit **Sort** operator—*e.g.*, to implement **ORDER BY** or **GROUP BY**—that now becomes obsolete,
- employ order-dependent operators at no extra cost.

May reduce overall plan cost, even if the subplan itself does not benefit: sorting effort will only pay off later.

- **Nested Loop Join** and **Merge Join** can deliver rows in such interesting orders.



## Merge Join: Low Memory Requirements

---

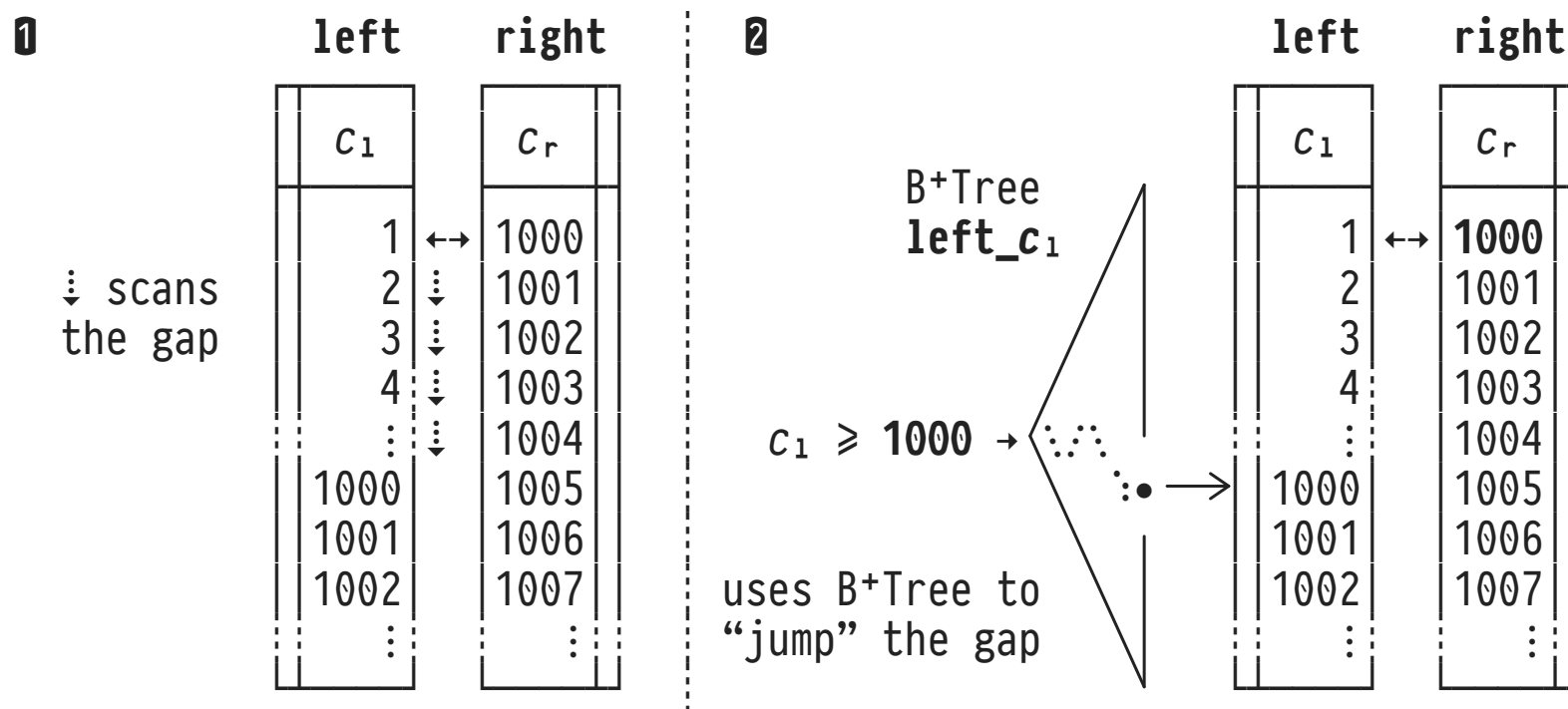
**Hash Join** (see below) is the go-to equi-join algorithm in modern RDBMSs including PostgreSQL. If **memory is tight**, however, **Merge Join** may be superior:

- If inputs are sorted, the actual *merging* requires as few as 3 buffer pages ( $2 \times \text{input} + 1 \times \text{output}$ ).
  - Requirement: *right* needs no re-scanning, e.g., if *left.c<sub>1</sub>* is unique.
  - See **Merge Join** plan property: **Inner Unique: true**.
  - Algorithm **MergeJoinUnique(left, right, c<sub>1</sub>, c<sub>r</sub>)** requires no management of  $\downarrow$  at all. **Q**: Simplified code?



## Challenges for Merge Join

- Large groups of repeating values in *right* input (*i.e.*, positions of  $\downarrow$  and  $\rightarrow$  diverge). Q: Worst case?
- Large *left.c<sub>1</sub>*  $\leftrightarrow$  *right.c<sub>r</sub>* gaps. Consider ①:





## 5 : Hash Join

---

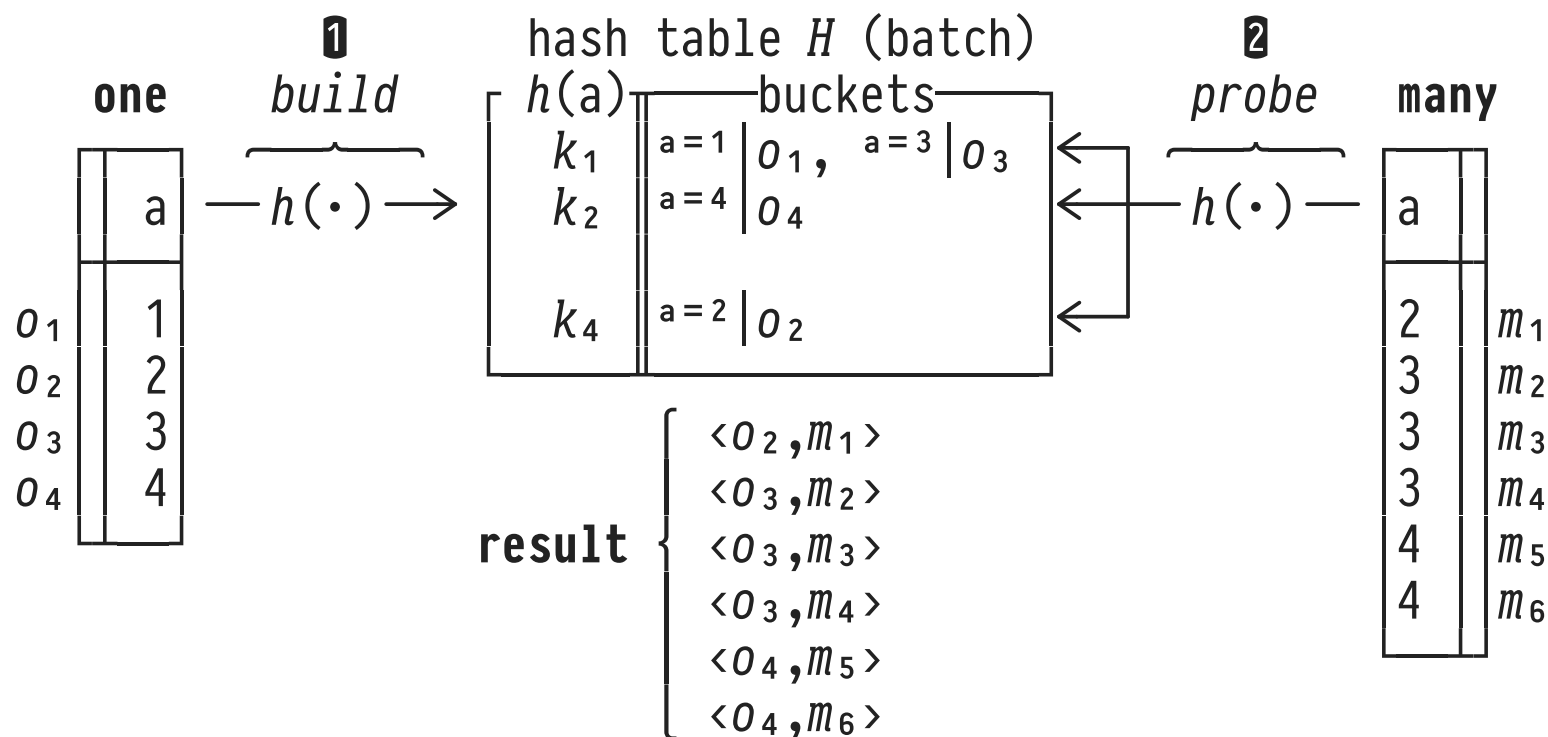
Equi-joins—*e.g.*, foreign-key joins—are arguably the most prominent kinds of relational join. Merge Join relies on *sorting* while **Hash Join** uses *hashing* to perform equi-joins:

1. **Build:** Read and hash the rows of one input table to populate a **hash table**  $H$ . Requires memory to store  $H$ .
2. **Probe:** Iterate over and hash rows of other input table. Find potential join partner rows in hash bucket of  $H$ .
  - If  $|H| > \text{working memory}$ , partition the build/probe tables, iterate phases (**Hybrid Hash Join**).
  - Hash Join does not require input order and does not guarantee output order.

# Hash Join: ... FROM one AS o, many AS m WHERE o.a = m.a



- **Build + Probe:** Apply hash function  $h(\cdot)$  to columns  $a$ .
- **Probe:** Evaluate join predicate  $o.a = m.a$  for entries in hash bucket with key  $k_i = h(m.a)$  only.





## Hash Join: Pseudo Code

---

```

HashJoin(build, probe,  $c_1$ ,  $c_r$ ):
     $j \leftarrow \phi$ ;
     $H \leftarrow []$ ;                                } empty hash table

    for  $b \in build$                                 } 1 build
    | insert  $b$  into bucket  $H[h(b.c_1)]$ ;           } phase

    for  $p \in probe$                                 }
    | for  $b \in H[h(p.c_r)]$                          } 2 probe
    | | if  $b.c_1 = p.c_r$                              } phase
    | | | append  $\langle b, p \rangle$  to  $j$ ;

    return  $j$ ;
  
```



## Hash Join: Execution Plan

### QUERY PLAN

Hash Join (cost=...) (actual time=... loops=...)

Hash Cond: (... = ...)

-> 「 Subplan probe (cost=...) (actual time=... loops=1) ㄱ

└─> Hash (cost=...) (actual time=... loops=1)

└─> 「 Subplan build (cost=...) (actual time=... loops=1) ㄱ

- Use smaller join input for *build* phase (reduces  $|H|$ ).
- ⚠ Indexes on *build* and *probe* inputs remain unused, even if defined on join predicate columns.

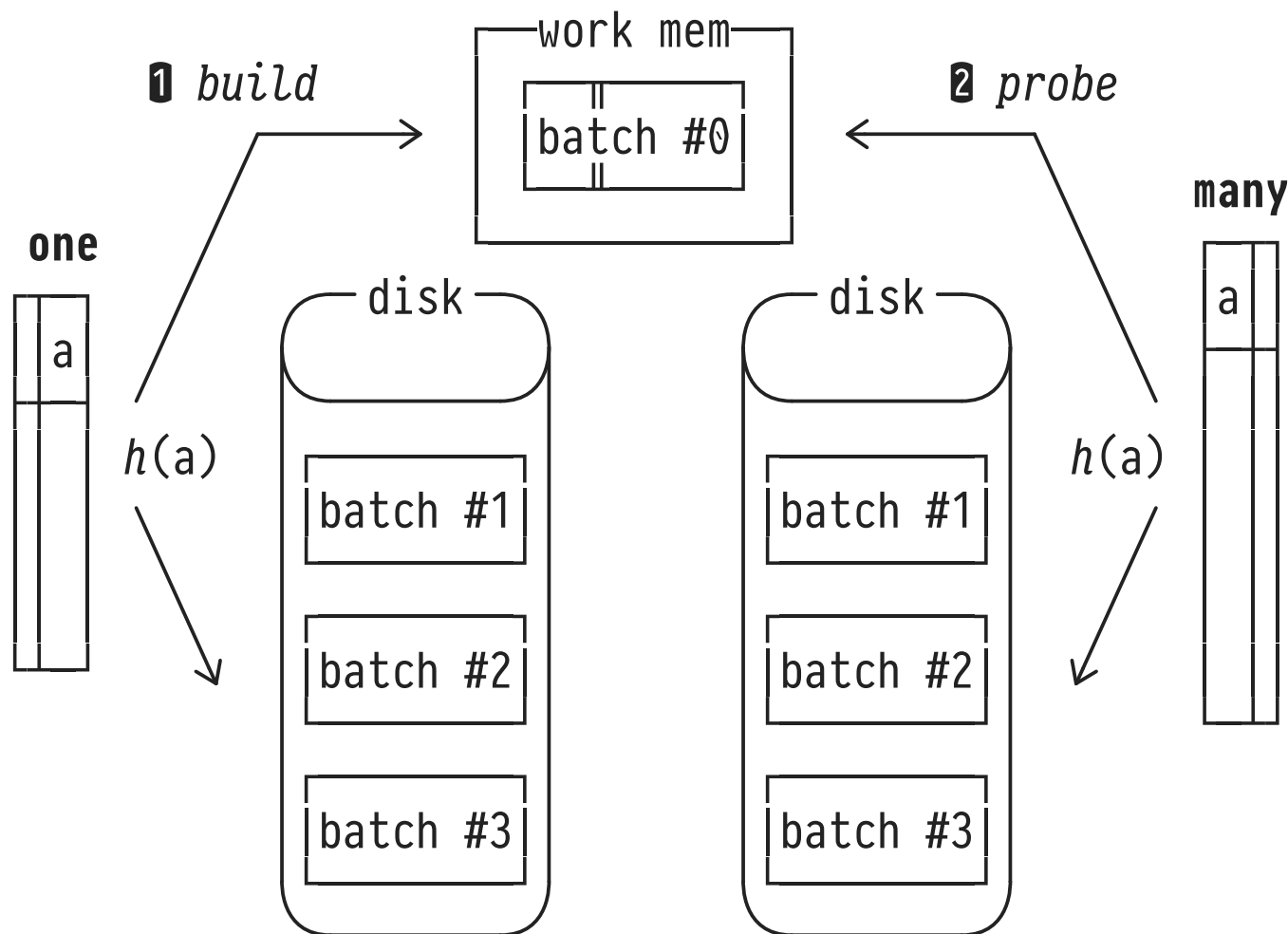




# Multiple Rounds: (Parallel) Hybrid Hash Join

- Input in round 0: tables **one** and **many**.
- Input in round  $i \geq 1$ : batches  $\#i$  read from temp files.
- Prepare  $2^n$  batches, first  $n$  bits of  $h(a)$  determine batch #:

batches #0: 0 0...  
 #1: 0 1...  
 #2: 1 0...  
 #3: 1 1...



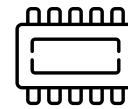


## Parallel Hybrid Hash Join (With Skew)

---

- If working memory cannot hold entire hash table  $H$ , use hash key  $h(\cdot)$  to split *build* input into  $2^n$  batches.
  - *Probe* input hashed into batch #0 is joined as usual (**round 0**).
  - All other batches processed in  **$2^n - 1$  rounds** (in //).
- 💡 Allocate additional **skew batch** in working memory:

Place row  $t$  in  $\left\{ \begin{array}{l} \text{skew batch, if } t.a \text{ among most common} \\ \text{a-values in } \textit{probe} \text{ input,} \\ \text{batch } \#i, \text{ based on } h(t.a), \text{ otherwise.} \end{array} \right.$



## 6 : $Q_{1,1}$ : Equi-Joins in MonetDB

---

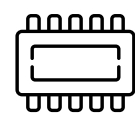


```
SELECT o.b AS b1, m.b AS b2
FROM   one AS o,
       many AS m
WHERE  o.a = m.a
```

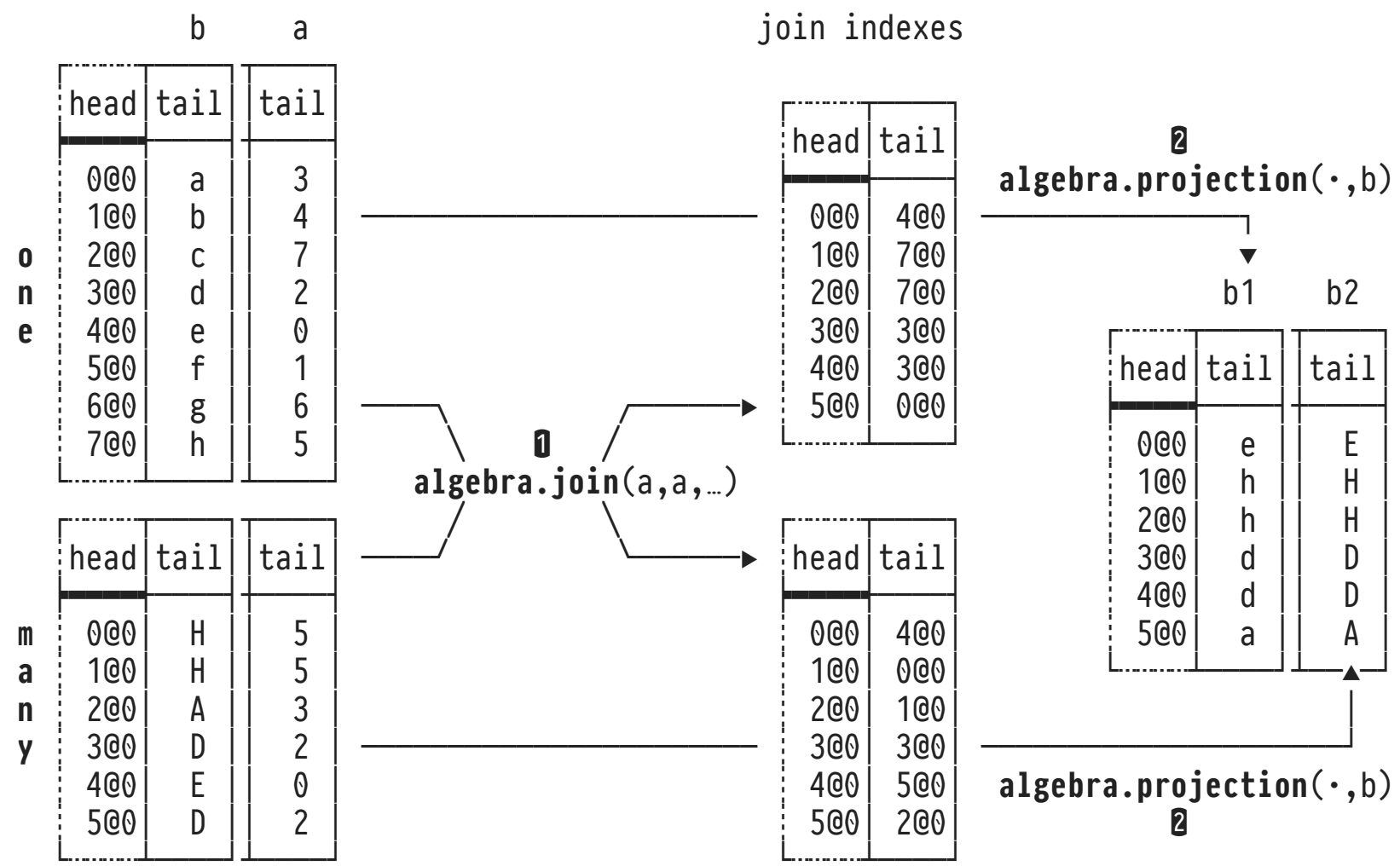
Since database instances reside on hosts with plenty of RAM, **Hash Join** is the go-to join method for MMDBMSs.

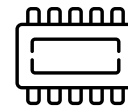
In MonetDB, a join computes **join index** BATs<sup>5</sup> to identify rows in **one**, **many** that find a join partner.

<sup>5</sup> Much like filtering is implemented in terms of **selection vectors**.



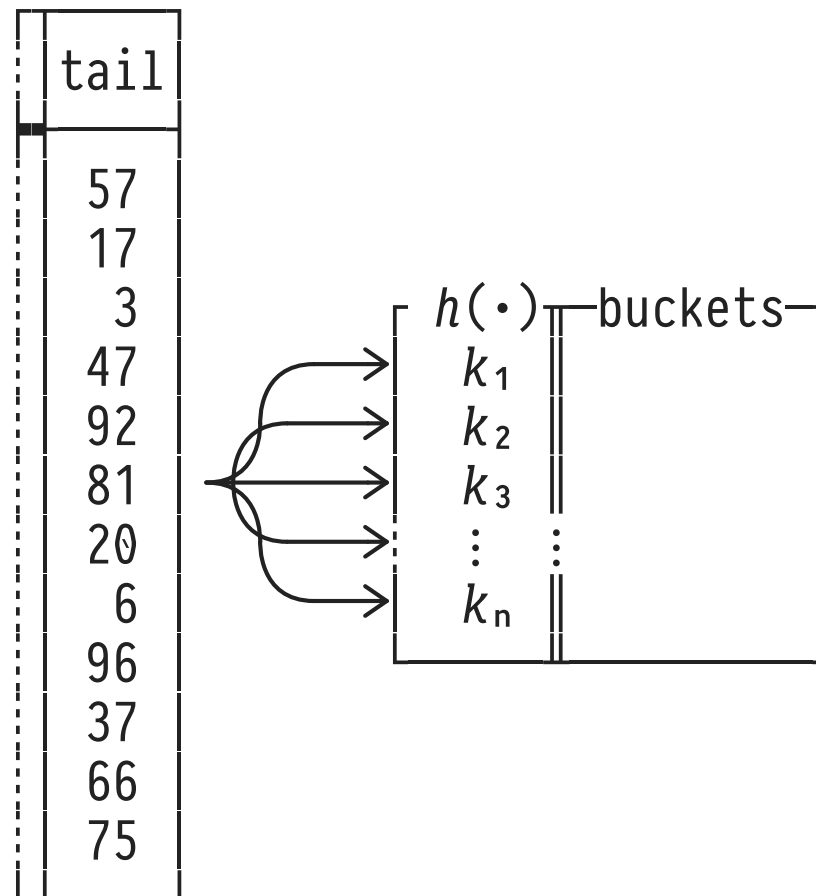
# Equi-Joins and Join Indexes in MonetDB



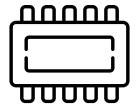


## Partitioning BATs Into (Too) Many Buckets

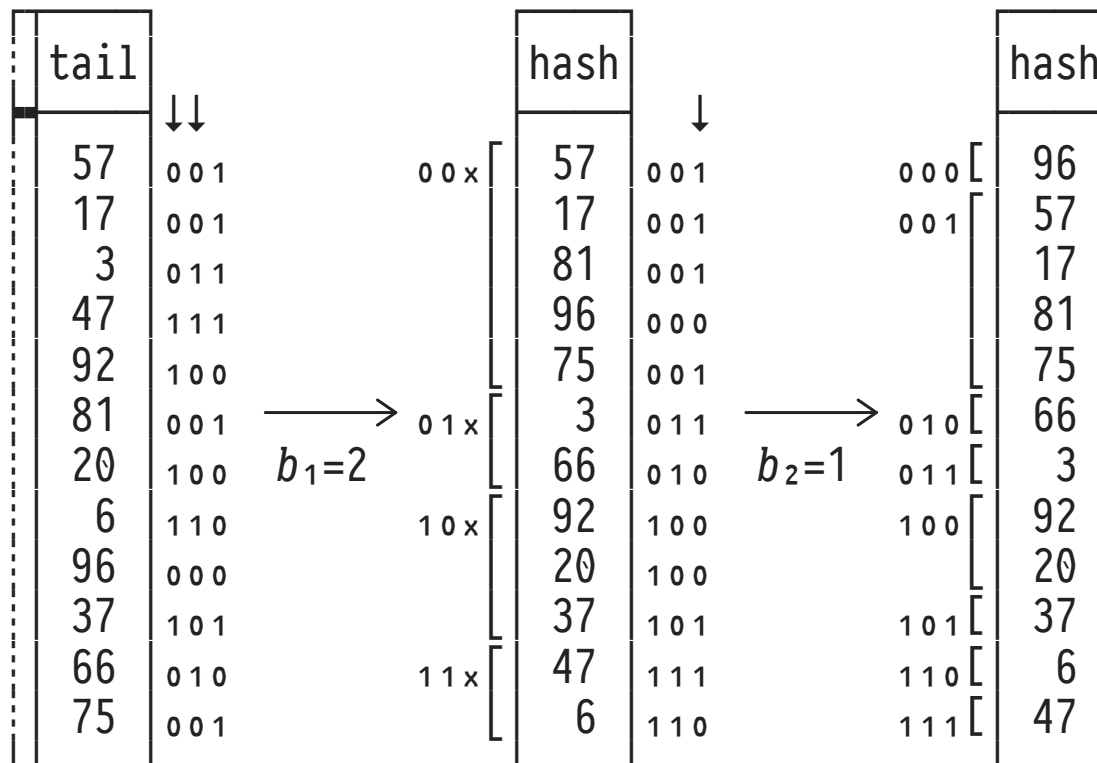
- To prepare Hash Join, use  $h(\cdot)$  to distribute rows into hash buckets.
- Requires random writes into  $n$  different memory locations.
- If  $n$  is (too) large:
  - Cache thrashing (# of cache lines exceeded). 🙄
  - TLB<sup>6</sup> misses. 🙄
- 💡 Reduce number of buckets considered at any one time.



<sup>6</sup> The CPU's *Translation Lookaside Buffer* stores recent translations from virtual into physical memory locations.



# Radix-Clustering



- To distribute by  $B$  bits in  $p$  passes:

- Define  $b_i$  such that

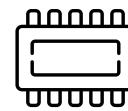
$$B = \sum_{i=1}^p b_i$$

- In pass  $i$ , distribute by  $b_i$  bits of the hash.

- # of buckets created:

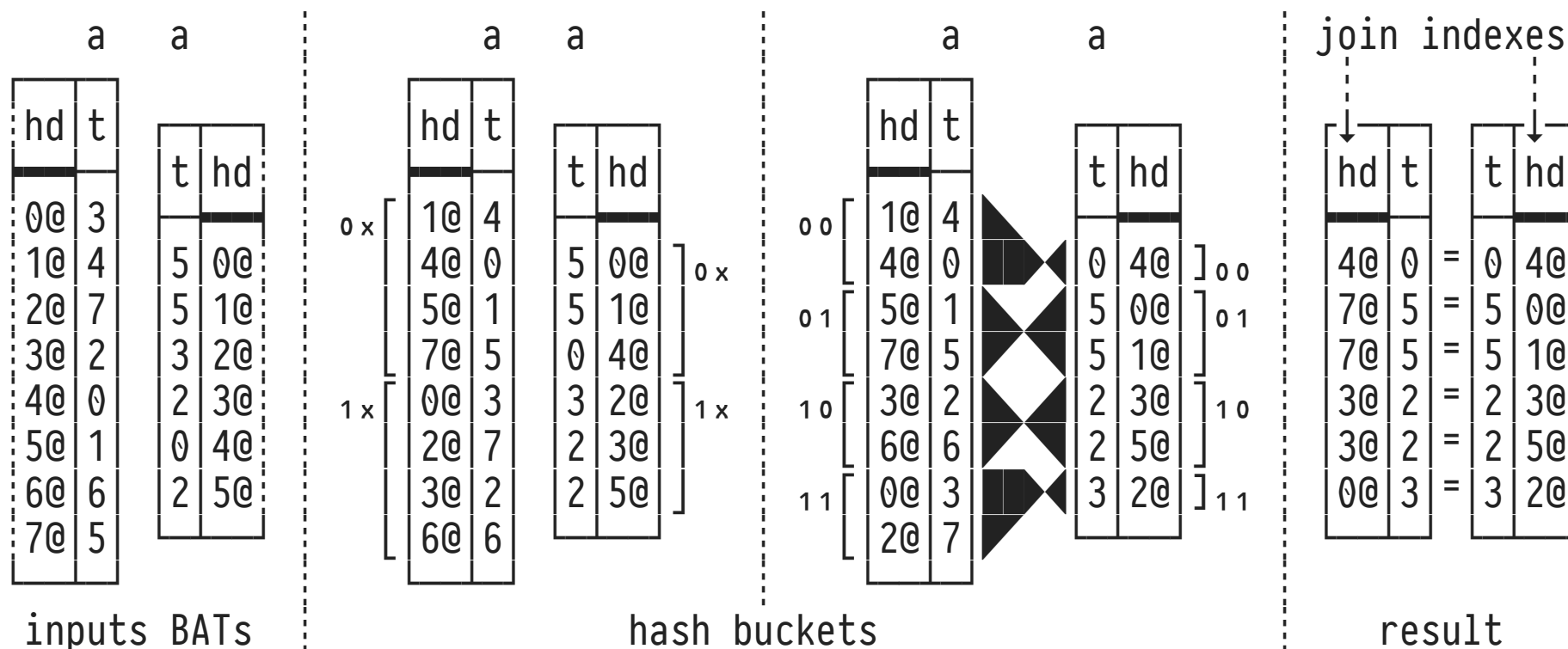
$$\prod_{i=1}^p 2^{b_i}$$

- Only write to  $2^{b_i}$  buckets in each pass to avoid cache thrashing and TLB misses.



# Radix-Cluster Equi-Join in $Q_{1,1}$ ( $\mathbf{o.a = m.a}$ )

- Two-pass ( $p = 2$ ) radix-clustering with  $b_1 = b_2 = 1$ :



- Rows for bucket-local joins fit into the CPU cache.