DB 2

12 – Joins

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 ┊ Q_{11}: One-to-Many Joins

Join (⋈) is a core operation in query processing: given two tables,[1] 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…n rows of many: *1:n relationship*.
  - ⇒ Join size ∈ $\{0,\ldots,n \times |\text{one}|\}$ rows. Largest possible result if $n = |\text{many}|$ (Cartesian product).

[1] 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

one

a	b	c
a_1	c4ca…	3
a_2	c81e…	0
a_3	eccb…	2

# of join partners

many

a→one(a)	b	c
a_1	…849b	1
a_1	…849b	2
a_1	…849b	3
a_3	…baf3	1
a_3	…baf3	2

this is the c^{th} partner

- Join predicates:
  1. `one.a = many.a` (index-supported)
  2. `md5(one.a) = one.b || many.b` (`||`: string concat)

PostgreSQL: Join Algorithms

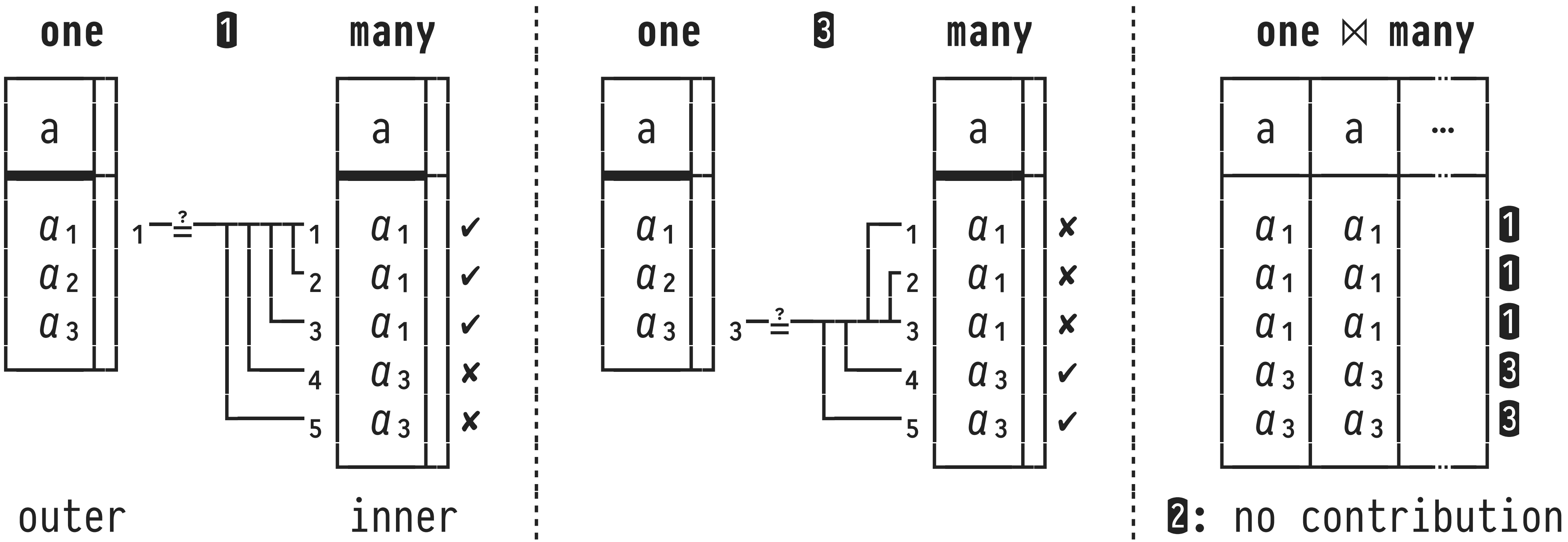
RDBMSs choose between several **join algorithms** based on

- the join **predicate type** (equi-join vs. θ-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 θ, 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 ❶…❸ over rows of **outer table** (here: one) once.
  - For every outer row, iterate over **inner table**.
- Performs |outer| × |inner| join predicate evaluations.

Nested Loop Join (NL⋈) — “The Fallback”

```
NLJ(outer,inner,θ):
 j = ϕ;
 for o ∈ outer
   │ for i ∈ inner
   │   │ if o θ i
   └   └   └ append ‹o,i› to j;
 return j;
```

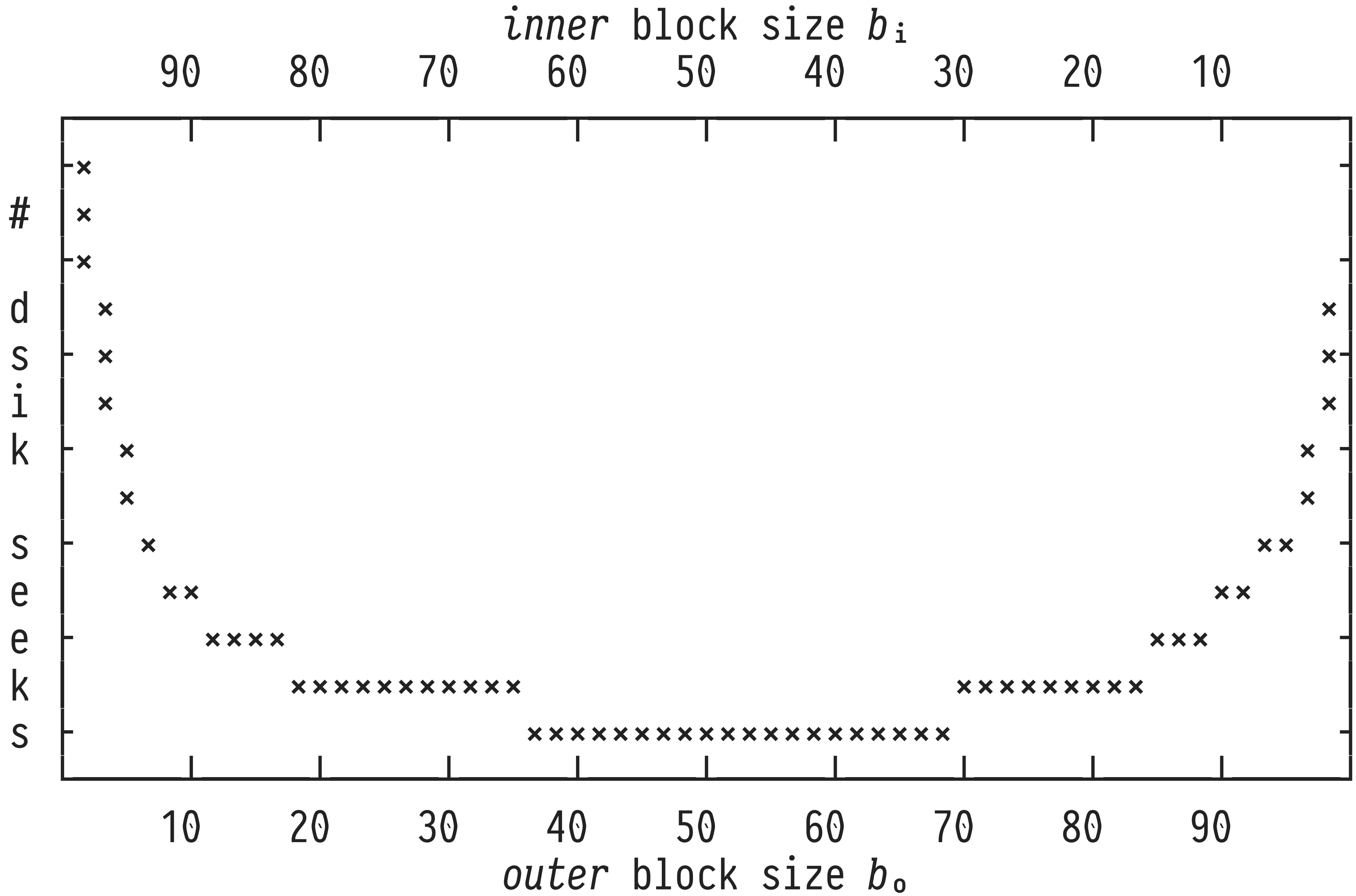
- No restrictions regarding $\theta \in \{=, <, \leqslant, <>, \ldots\}$.
- No restrictions regarding sort order of *outer*/*inner*.
- Preserves sort order of *outer*.
- Indexes on *outer*/*inner* are ignored.
- Benefits if *inner* can be iterated over quickly (*e.g.*, materialized and/or fits into database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

Block Nested Loop Join (BNL⋈)

```
BlockNLJ(outer,inner,θ):
 j = ϕ;
 foreach block (of size b_o) bo ∈ outer
   foreach block (of size b_i) bi ∈ inner
     for o ∈ bo                              ⎫
       for i ∈ bi                            ⎬ entirely performed
         if o θ i                            ⎪ inside the buffer
           append ⟨o,i⟩ to j;                ⎭
 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 $\lceil |outer|/b_\circ \rceil$ 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⋈)

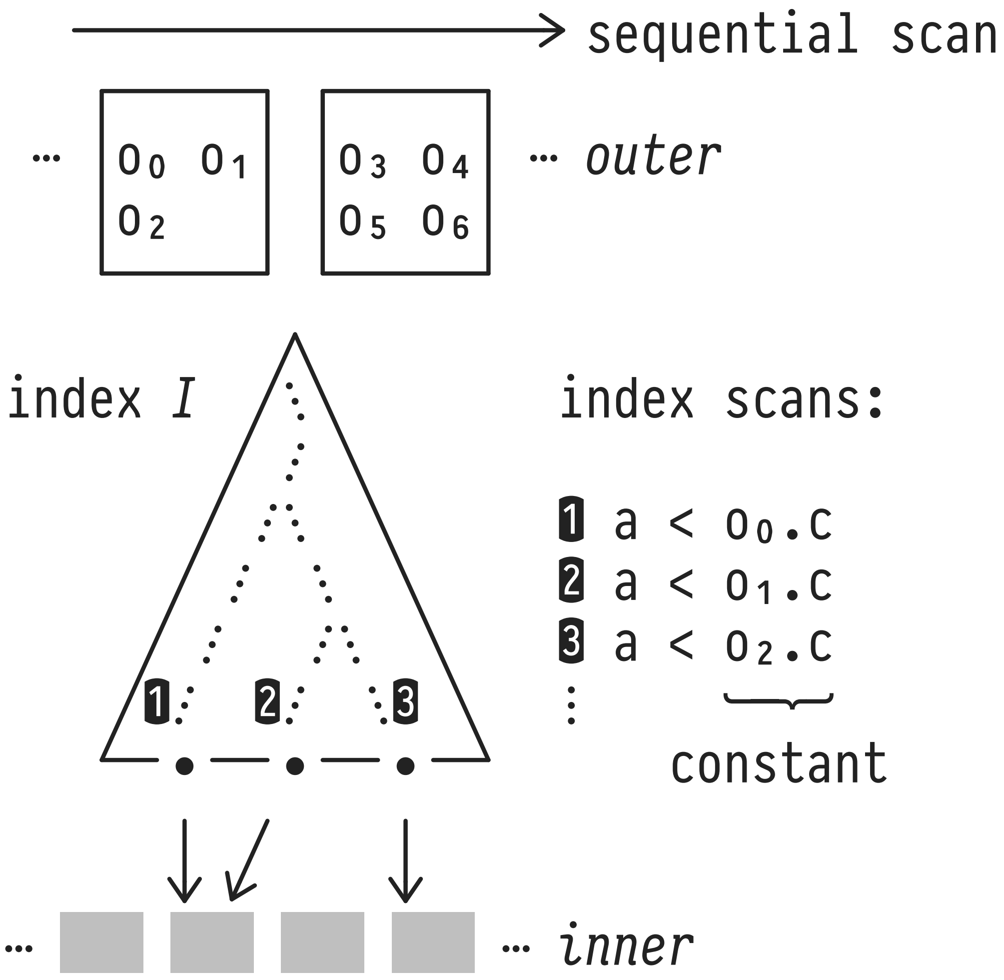
NL⋈ may be sped up considerably if the |*outer*| scans of *inner* can be turned into |*outer*| **index scans on** *inner*:

```
IndexNLJ(outer,inner,θ):
 j = ϕ;
 for o ∈ outer
   for i ∈ Index[Only]Scan(I, o θ □)
     append ‹o,i› to j;
 return j;
```

$\underbrace{o\ \theta\ \square}_{\text{index condition}}$

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

Index Nested Loop Join (INL⋈)




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

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

4 ¦ Merge Join

Join algorithm **Merge Join** supports equality join predicates ("equi-joins") of the form $c_l = c_r$:[2]

1. left input *must* be **sorted** by c_l, 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_l (and thus c_r).

N.B.: Merge Join's guaranteed output order can provide a true benefit during later query processing stages.

[2] Generalizations to predicates $c_l\ \theta\ c_r$ with $\theta \in \{<, \leqslant, \ldots\}$ have been defined but are seldomly found implemented in actual RDBMSs.

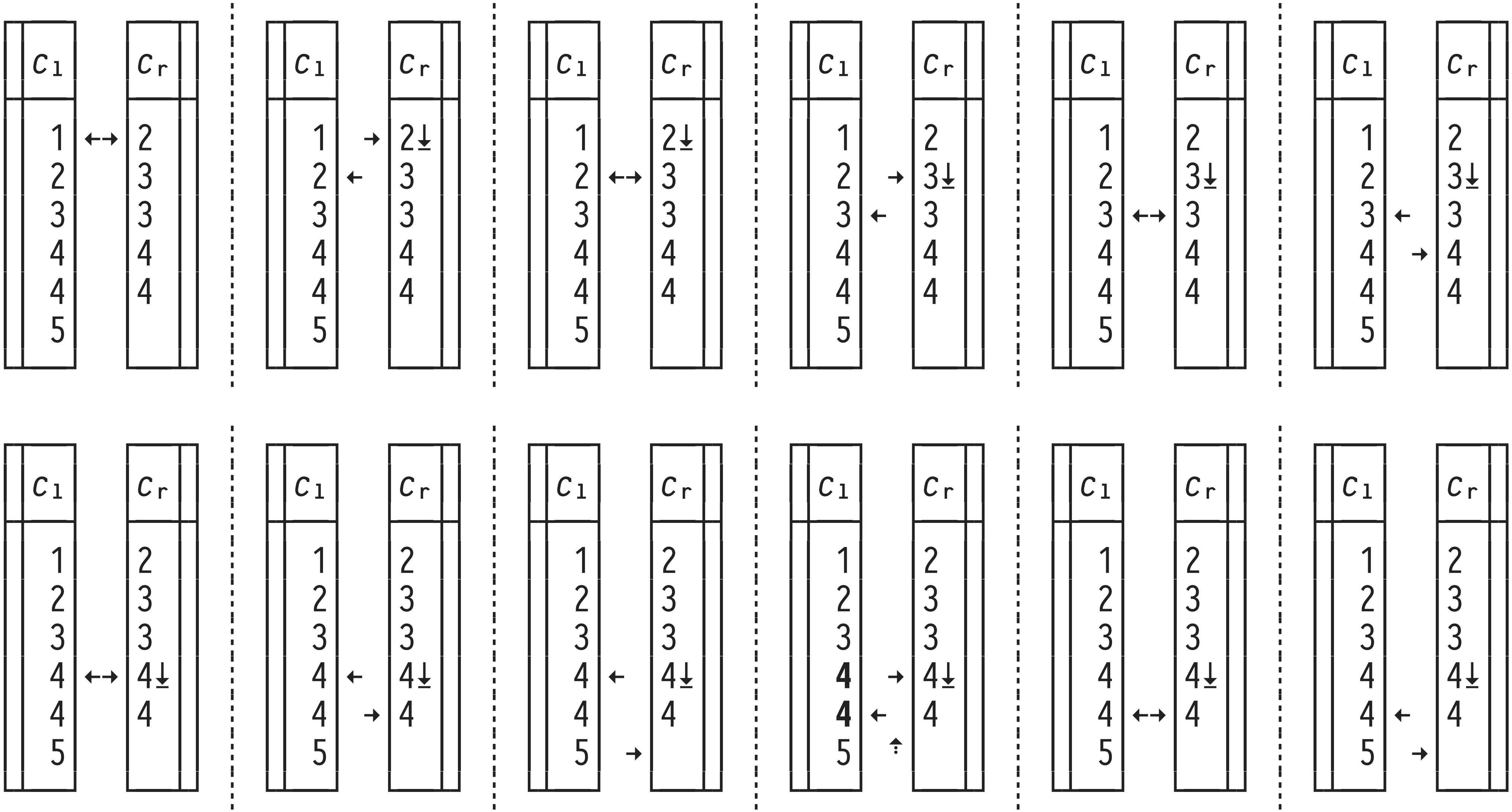
Merge Join Ingredients

Merge Join performs synchronized forward (≡ sorted) scans:[3]

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

[3] Arrow symbols ←, →, ⤓, ⇡ refer to the illustration on the next slide. Only the join columns c_l, c_r (of type int) are shown.

Merge Join: Synchronized Scan Pointers



Merge Join: Pseudo Code

```
MergeJoin(left,right,cₗ,cᵣ):
 j ← ∅;
 while left ≠ EOT ∧ right ≠ EOT        } reached end-of-table?
   while left.cₗ < right.cᵣ             ⎫
     advance left;                      ⎬ move scans forward
   while left.cₗ > right.cᵣ             ⎪ in lock step
     advance right;                     ⎭
   ⤓ ← right;                           } save current right pos
   while left.cₗ = ⤓.cᵣ                 } scan repeating left group
     right ← ⤓;                         } reset right scan
     while left.cₗ = right.cᵣ
       append ‹left,right› to j;
       advance right;
     advance left;
 return j;
```

Merge Join: Sorted Inputs

Merge Join requires inputs sorted on c_l/c_r. Options:

1. Introduce **explicit** **Sort** plan operator below Merge Join.
2. Input is **Index Scan** with key column prefix c_l/c_r.[4]
3. Input table is (perfectly) **clustered** on c_l/c_r.
4. **Subplan** below Merge Join delivers rows in c_l/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) ⌝
     ⌞                                                ⌟
```

[4] **Q:** Will Bitmap Index/Heap Scan also fit the bill here?

Merge Join: Re-Scanning the Right Input

Since Merge Join may need to reset the pointer in *right*, its subplan is required to support re-scanning of rows:

- Supported by Index Scan and in-memory buffers, but may be impossible and/or costly for complex subplans.
- 💡 Place Materialize above *right* to support re-scan:

```
                          QUERY PLAN
Merge Join (cost=⋯) (actual time=⋯ loops=⋯)
  -> ⌜ Subplan left (cost=⋯) (actual time=⋯ loops=1)  ⌝
     ⌞                                                ⌟
  -> Materialize (cost=⋯) (actual time=⋯ loops=1)
       -> ⌜ Subplan right (cost=⋯) (actual time=⋯ loops=1) ⌝
          ⌞                                                ⌟
```

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.

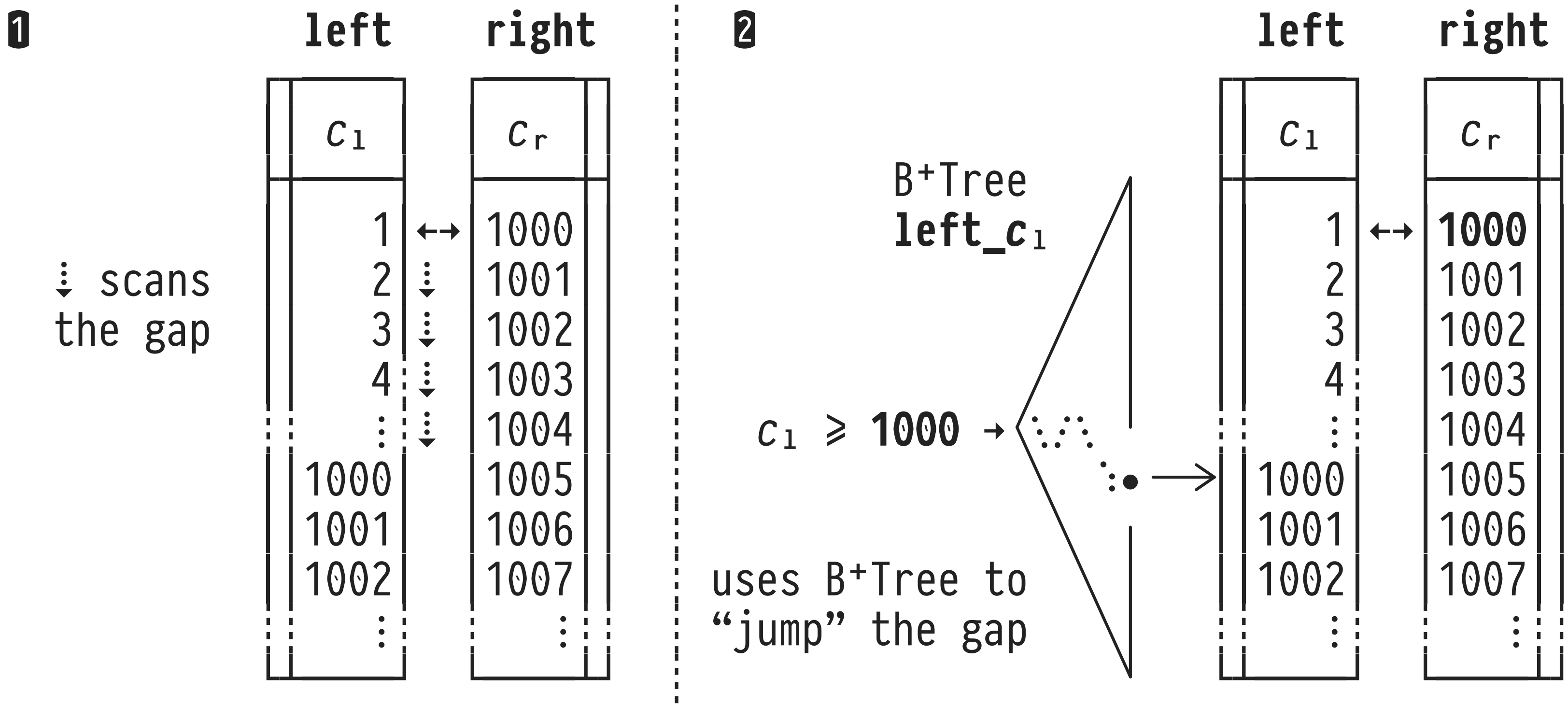
Merge Join: Low Memory Requirements

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×input + 1×output).
  - Requirement: *right* needs no re-scanning, *e.g.*, if *left*.c_l is unique.
  - See Merge Join plan property: Inner Unique: true.
  - Algorithm MergeJoinUnique(*left*,*right*,c_l,c_r) requires no management of ⭳ at all. **Q:** Simplified code?

Challenges for Merge Join

- Large groups of repeating values in *right* input (*i.e.*, positions of ⭳ and → diverge). **Q:** Worst case?
- Large *left*.c_l ↔ *right*.c_r gaps. Consider ❶:



5 ┆ Hash Join

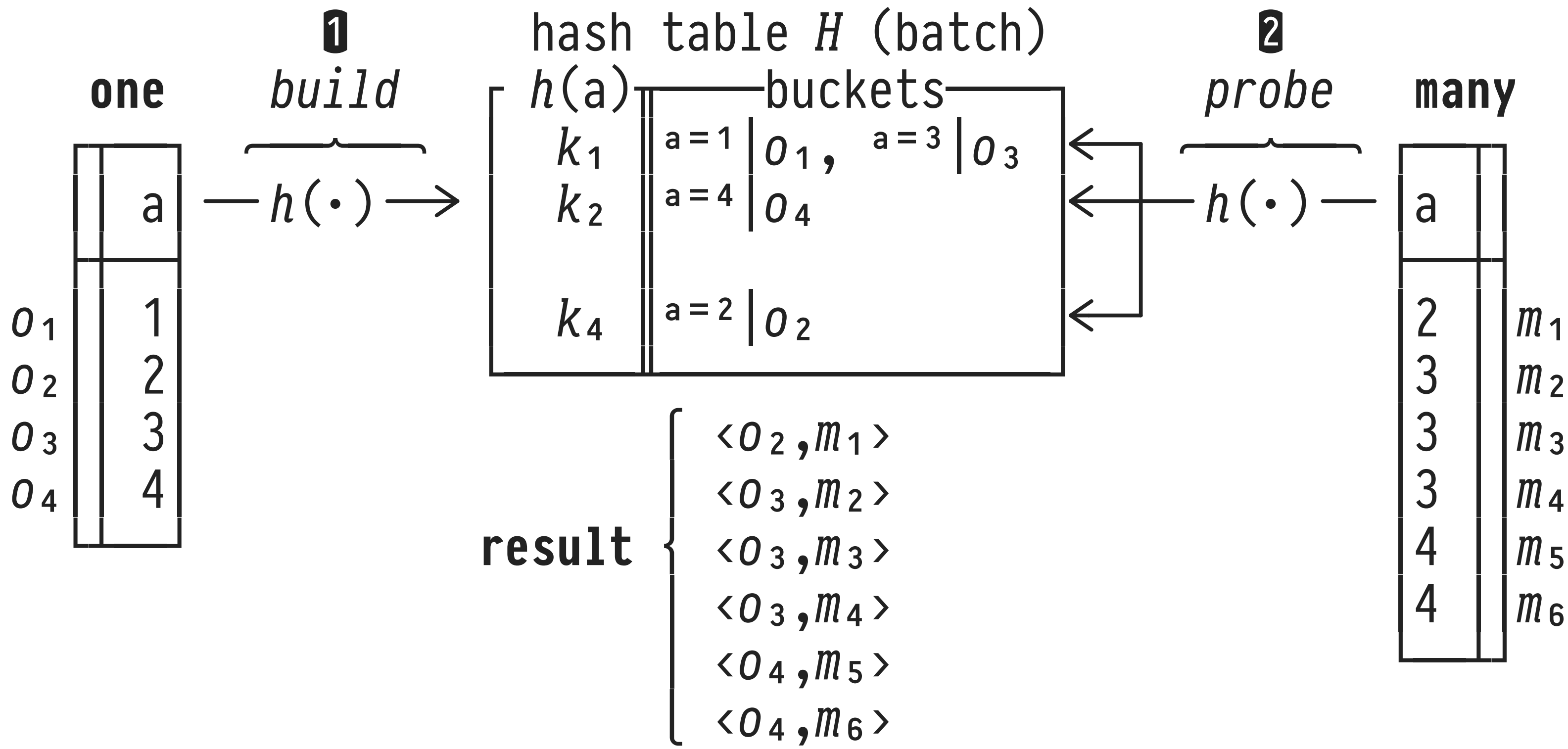
Equi-joins—*e.g.*, foreign-key joins—are arguably the most prominent kinds of relational join. Merge Join relies on *sorting* while **Hash Join** uses *hashing* to perform equi-joins:

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 $|H|$ > working memory, partition 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 $h(\cdot)$.
- **Probe:** Evaluate join predicate o.a = m.a for entries in hash bucket with key $k_i = h(\text{m.a})$ only.



Hash Join: Pseudo Code

```
HashJoin(build,probe,c₁,cᵣ):
  j ← ∅;
  H ← [];                                    } empty hash table

  for b ∈ build                              ⎫ ❶ build
   ⎣ insert b into bucket H[h(b.c₁)];        ⎭   phase

  for p ∈ probe                              ⎫
   │ for b ∈ H[h(p.cᵣ)]                      ⎬ ❷ probe
   │  │ if b.c₁ = p.cᵣ                       ⎪   phase
   ⎣  ⎣  ⎣ append ‹b,p› to j;                ⎭

  return j;
```

Hash Join: Execution Plan

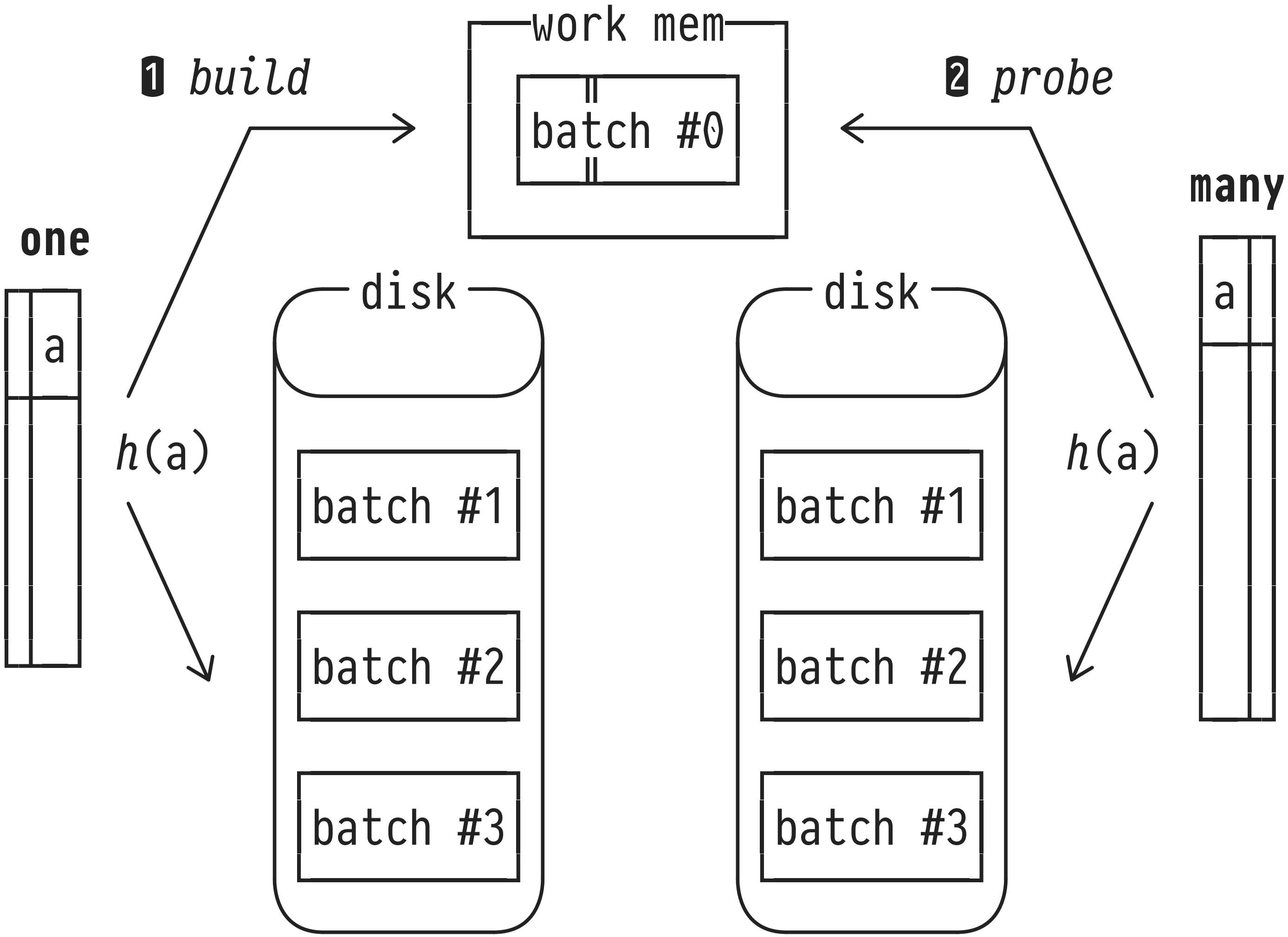
```
                         QUERY PLAN
Hash Join (cost=…) (actual time=… loops=…)
  Hash Cond: (… = …)
  -> ⌜ Subplan probe (cost=…) (actual time=… loops=1)  ⌝
     ⌞                                                 ⌟
  -> Hash (cost=…) (actual time=… loops=1)
      -> ⌜ Subplan build (cost=…) (actual time=… loops=1) ⌝
         ⌞                                                ⌟
```

- 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: Hybrid Hash Join

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

  batches #0: 00…
  #1: 01…
  #2: 10…
  #3: 11…

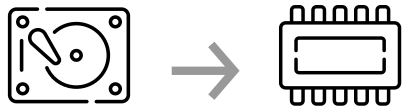


Hybrid Hash Join (With Skew)

- 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 subsequent **2^n-1 rounds.**

- 💡 Allocate additional **skew batch** in working memory:

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



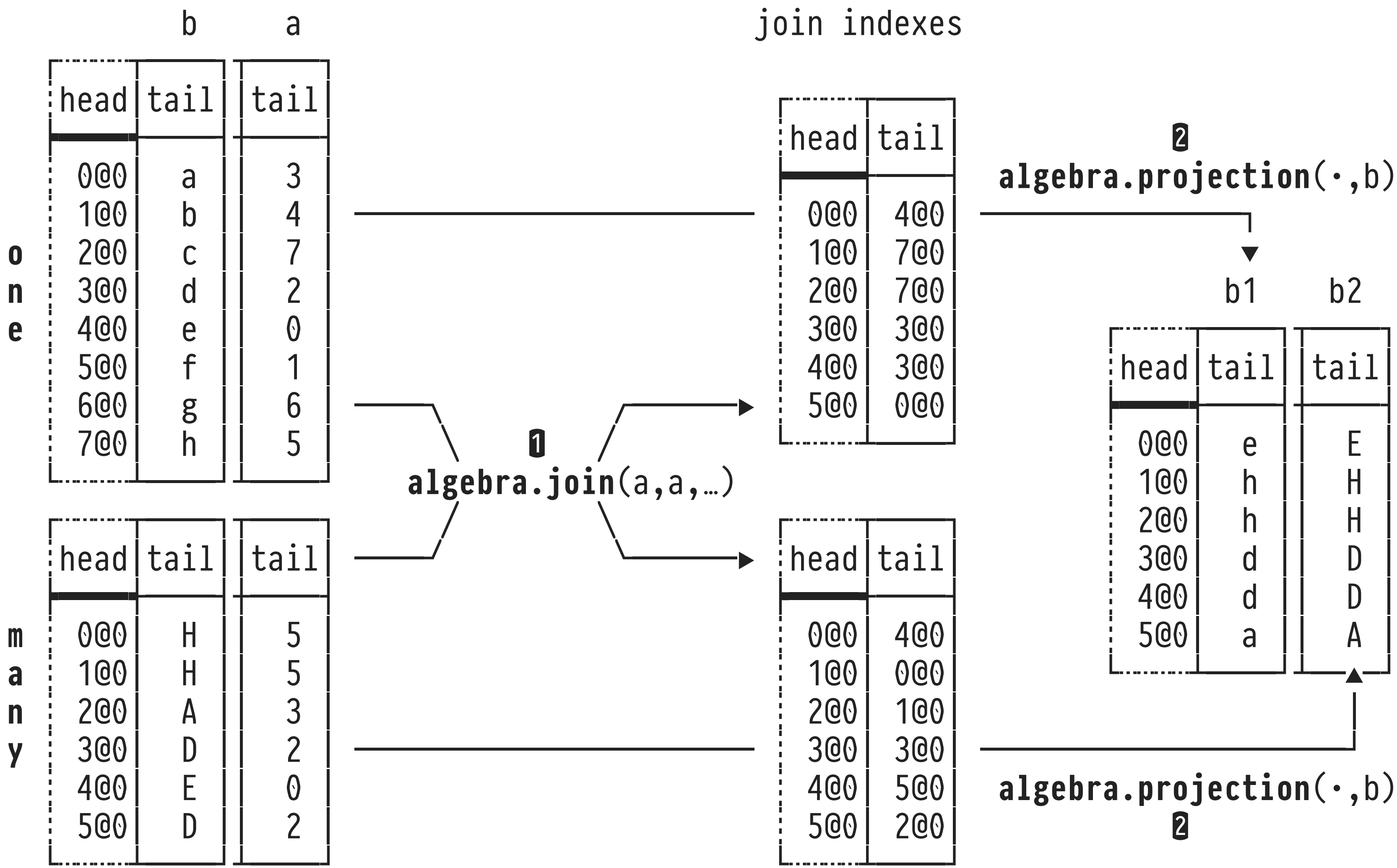
```sql
SELECT o.b AS b1, m.b AS b2
FROM   one AS o,
       many AS m
WHERE  o.a = m.a
```

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

In MonetDB, a join computes **join index** BATs[5] to identify rows in one, many that find a join partner.

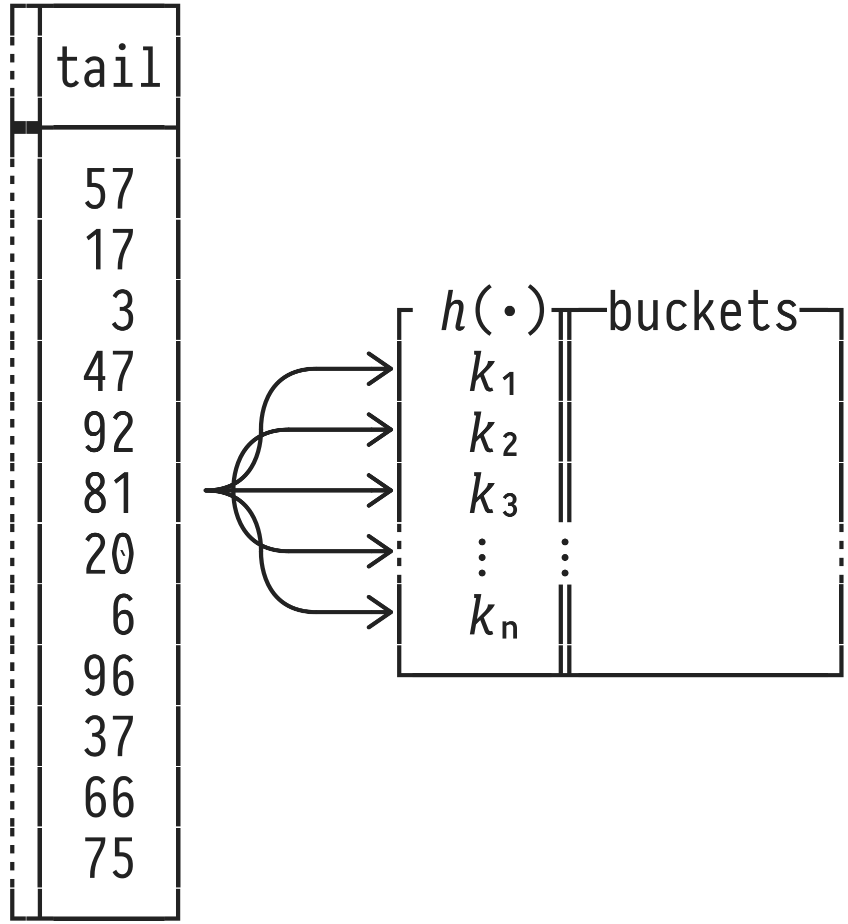
[5] Much like filtering is implemented in terms of **selection vectors**.

Equi-Joins and Join Indexes in MonetDB



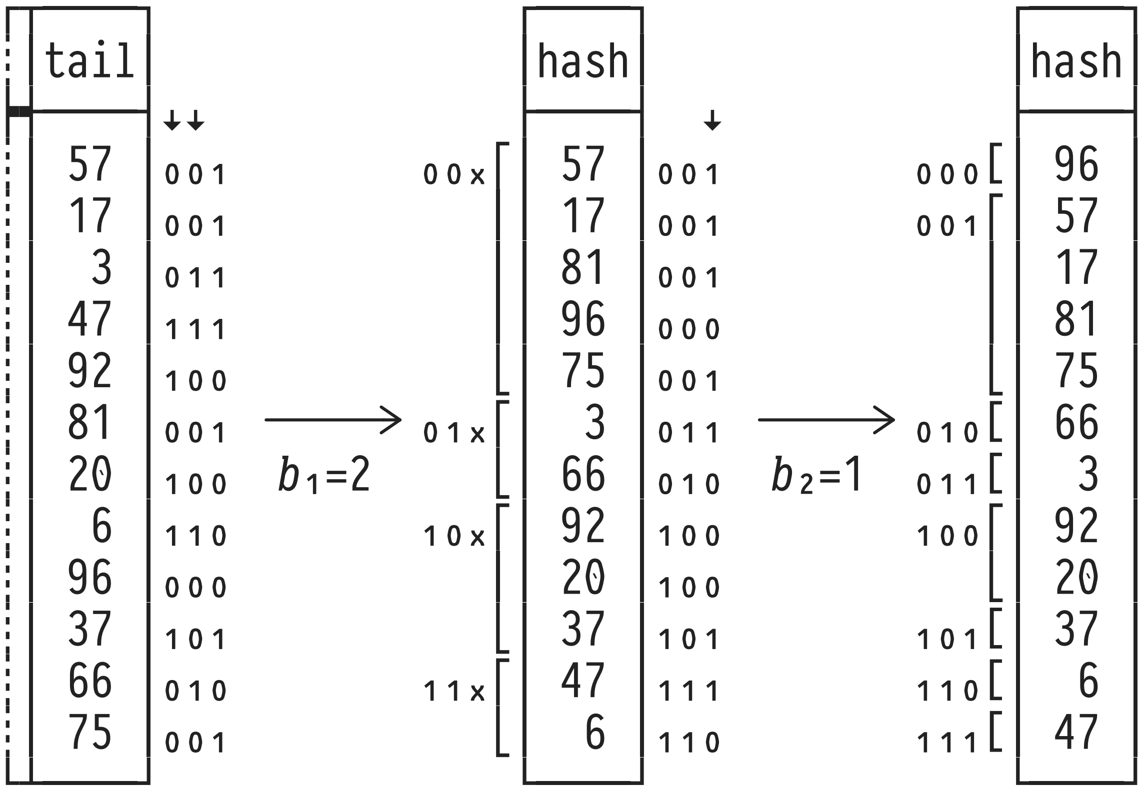
Partitioning BATs Into (Too) Many Buckets

- 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[6] misses. 👎
- 💡 Reduce number of buckets considered at any one time.



[6] 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:

  ❶ Define b_i such that

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

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

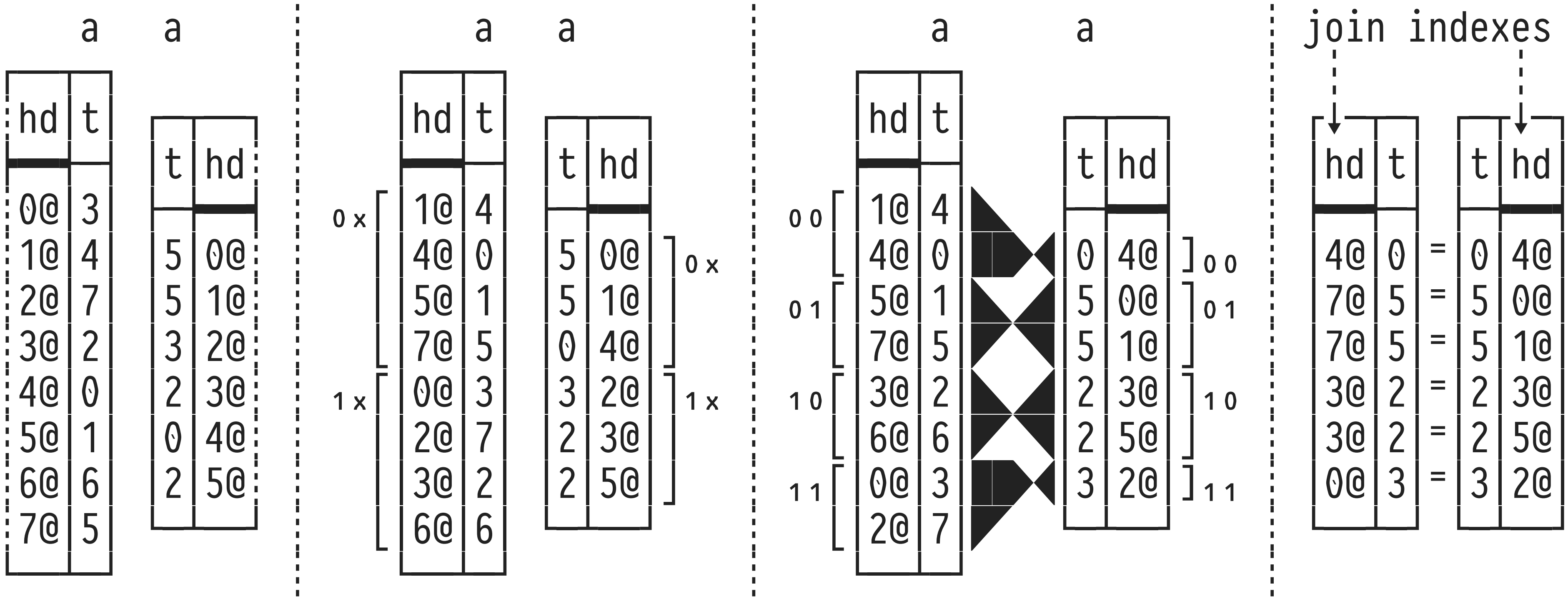
- # of buckets created:

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

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

Radix-Cluster Equi-Join in Q_{11} (o.a = m.a)

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



- Rows for cluster-local joins ⋈ fit into the CPU cache.