DB 2

12 – Joins

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 ┊ Q_{11}: One-to-Many Joins

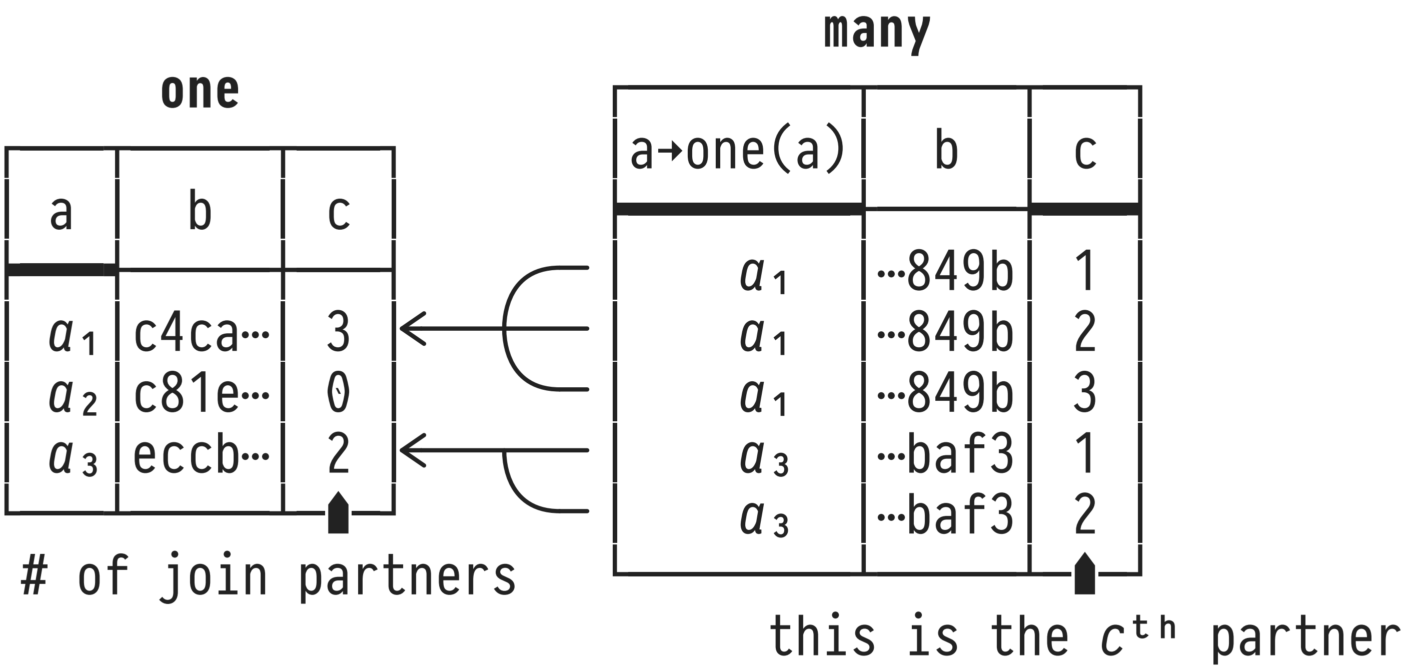
Join (⋈) is a core operation in query processing: given two tables,[1] form all pairs of related rows.

```
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)

```sql
DROP TABLE IF EXISTS one CASCADE;
DROP TABLE IF EXISTS many;

CREATE TABLE one  (a int PRIMARY KEY,
                   b text,
                   c int);
CREATE TABLE many (a int NOT NULL,
                   b text,
                   c int NOT NULL,
                   PRIMARY KEY (a,c),
                   FOREIGN KEY (a) REFERENCES one(a));

ALTER INDEX one_pkey RENAME TO one_a;
ALTER INDEX many_pkey RENAME TO many_a_c;

INSERT INTO one(a,b,c)
  SELECT i, left(md5(i::text), 16), random() * (100 + 1)    -- 1:100 relationship
  FROM   generate_series(1, 10000) AS i;

INSERT INTO many(a,b,c)
  SELECT o.a, right(md5(o.a::text), 16), i
  FROM   one AS o, LATERAL generate_series(0, o.c - 1) AS i;



ANALYZE one;
ANALYZE many;

SELECT avg(o.c) AS "average # of partners"
FROM   one AS o;
```

PostgreSQL: Join Algorithms

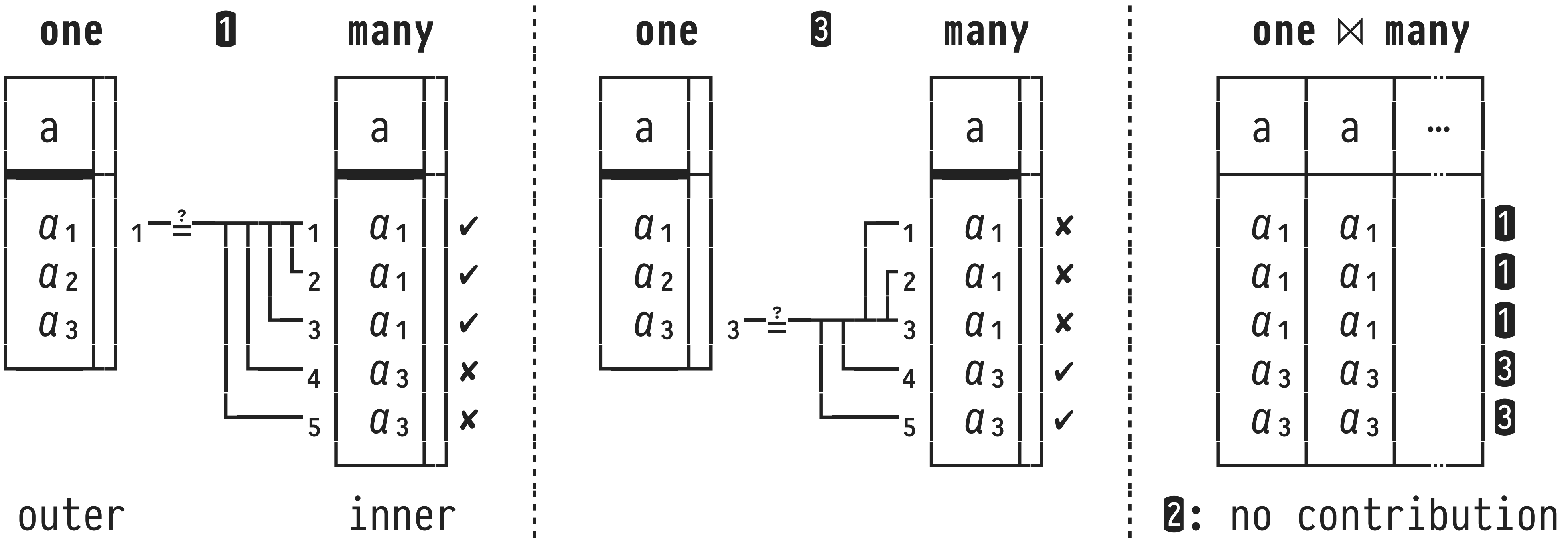
RDBMSs choose between several **join algorithms** based on

- 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)



❷: no contribution

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

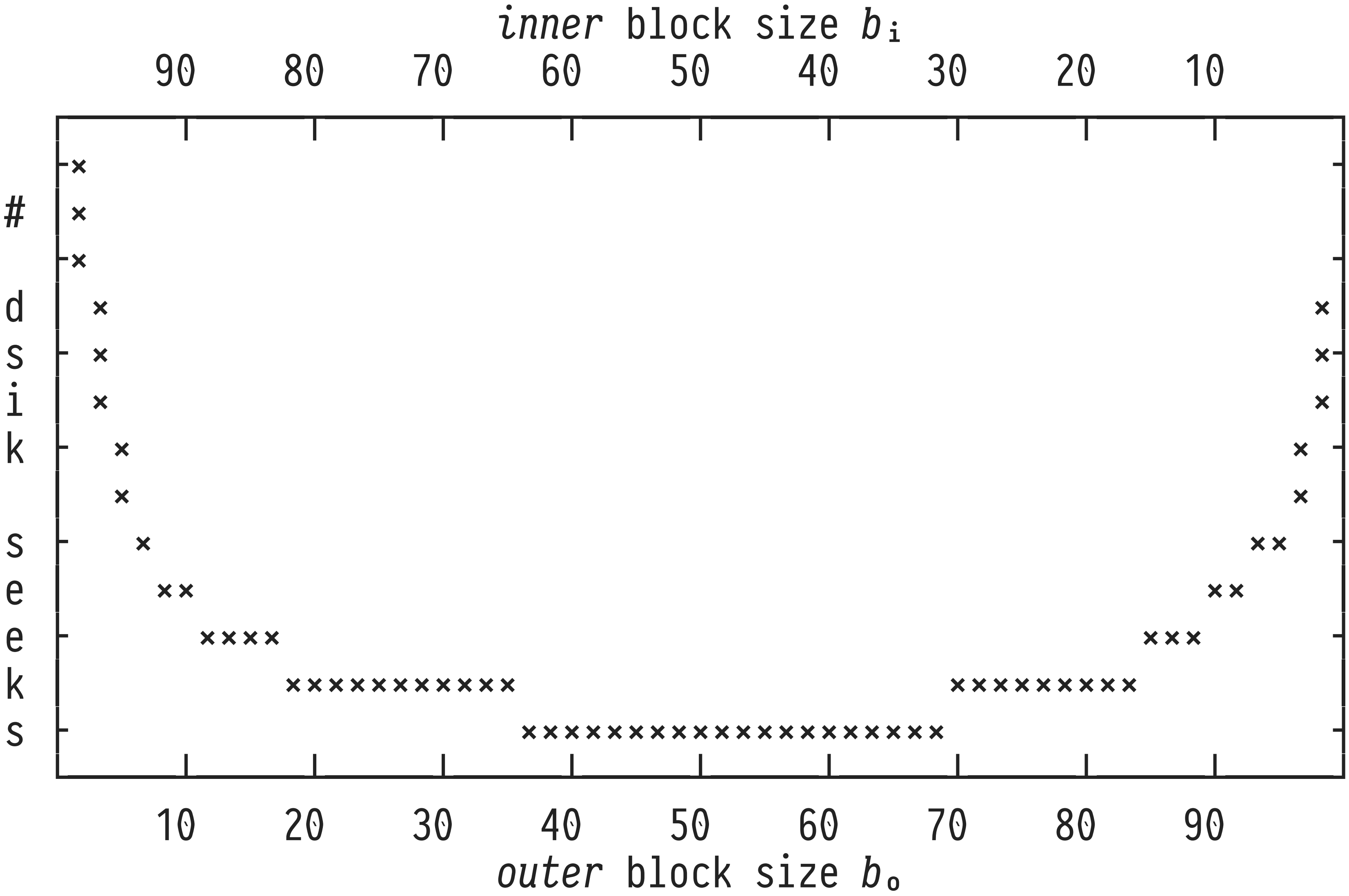
- No restrictions regarding $\theta \in \{=, <, \leqslant, <>, \ldots\}$. 👍
- No restrictions regarding sort order of *outer*/*inner*. 👍
- Preserves sort order of *outer*. 👍
- Indexes on *outer*/*inner* are ignored. 👎
- Benefits if *inner* can be iterated over quickly (*e.g.*, materialized and/or fits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

Block Nested Loop Join (BNL⋈)

```
BlockNLJ($outer$,$inner$,$\theta$):
  $j = \phi$;
  foreach block (of size $b_o$) $bo \in outer$
    foreach block (of size $b_i$) $bi \in inner$
      for $o \in bo$                              }
        for $i \in bi$                            }  entirely performed
          if $o\ \theta\ i$                       }  inside the buffer
            append $\langle o,i \rangle$ 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



- sample parameters: B = 100, $|\mathit{outer}|$ = 1000 blocks, $|\mathit{inner}|$ = 500 blocks
- plot: $\mathrm{seeks}(b_o) = \lceil |\mathit{outer}|/b_o \rceil + \lceil |\mathit{outer}|/b_o \rceil \times \lceil |\mathit{inner}|/(100 - b_o) \rceil$
    - $\mathrm{seeks}(5) = 1200$
    - $\mathrm{seeks}(10) = 600$
    - $\mathrm{seeks}(50) = 220$
    - $\mathrm{seeks}(90) = 561$
    - $\mathrm{seeks}(95) = 1010$

NL⋈: Materialization of the Inner Input

The inner NL⋈ input is scanned $\lceil |outer| / b_o \rceil$ 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)  ⌝
            ⌞                                            ⌟
```

Demonstrate the beneficial effect of Materialize in Nested Loops Join:

```
-- ❶ Input tables

  \d one
                Table "public.one"
┌────────┬─────────┬───────────┬──────────┬─────────┐
│ Column │  Type   │ Collation │ Nullable │ Default │
├────────┼─────────┼───────────┼──────────┼─────────┤
│ a      │ integer │           │ not null │         │
│ b      │ text    │           │          │         │
│ c      │ integer │           │          │         │
└────────┴─────────┴───────────┴──────────┴─────────┘
Indexes:
    "one_pkey" PRIMARY KEY, btree (a)
Referenced by:
    TABLE "many" CONSTRAINT "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)

  \d many
                Table "public.many"
┌────────┬─────────┬───────────┬──────────┬─────────┐
│ Column │  Type   │ Collation │ Nullable │ Default │
├────────┼─────────┼───────────┼──────────┼─────────┤
│ a      │ integer │           │ not null │         │
│ b      │ text    │           │          │         │
│ c      │ integer │           │ not null │         │
└────────┴─────────┴───────────┴──────────┴─────────┘
Indexes:
    "many_pkey" PRIMARY KEY, btree (a, c)
Foreign-key constraints:
    "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)


-- ❷ Evaluate Nested Loop Join with Materialize

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT *
    FROM   one AS o, many AS m
    WHERE  o.b < m.b AND m.c < 2 AND o.c < 2;
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                               QUERY PLAN                                                  │
├──────────────────────────────────────────────────────────────────────────────────────────────────────────┤
│ Nested Loop  (cost=0.00..64044.41 rows=1197918 width=50) (actual time=0.103..3091.830 rows=1516324 loops=1)
│   Output: o.a, o.b, o.c, m.a, m.b, m.c                      ⬆ joined row width 50 = 25 (one) + 25 (many)
│   Join Filter: (o.b < m.b) ⬅ evaluated without index support
│   Rows Removed by Join Filter: 1453076
```

```
  Buffers: shared hit=3755
  ->  Seq Scan on public.many m  (cost=0.00..9938.64 rows=19638 width=25) (actual time=0.018..90.615 rows=19796 loops=1)
        Output: m.a, m.b, m.c                                                                              ⬆
        Filter: (m.c < 2)                                                                       outer evaluated once
        Rows Removed by Filter: 480815
        Buffers: shared hit=3681
  ->  Materialize  (cost=0.00..199.91 rows=183 width=25) (actual time=0.000..0.011 rows=150 loops=19796)
        Output: o.a, o.b, o.c                                                                   ⬆
        Buffers: shared hit=74 ⬅ need 74 pages of work mem to materialize the subplan    inner evaluated MANY times
        ->  Seq Scan on public.one o  (cost=0.00..199.00 rows=183 width=25) (actual time=0.009..2.596 rows=150 loops=1)
              Output: o.a, o.b, o.c                                                                       ⬆
              Filter: (o.c < 2)                                                 Materialize subplan executed ONCE
              Rows Removed by Filter: 9850                                  (⚠ assign smaller table one as inner since
              Buffers: shared hit=74                                          we need less materialize memory that way)
Planning time: 0.155 ms
Execution time: 3204.972 ms ⬅ fast!
```

```
-- 3 Evaluate Nested Loop Join without Materialize

set enable_material = off;

EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT *
  FROM   one AS o, many AS m
  WHERE  o.b < m.b AND m.c < 2 AND o.c < 2;
```

```
                                         QUERY PLAN
Nested Loop  (cost=0.00..1863891.59 rows=1197918 width=50) (actual time=0.070..15642.764 rows=1516324 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Join Filter: (o.b < m.b)
  Rows Removed by Join Filter: 1453076
  Buffers: shared hit=552224
  ->  Seq Scan on public.one o  (cost=0.00..199.00 rows=183 width=25) (actual time=0.022..1.870 rows=150 loops=1)
        Output: o.a, o.b, o.c
        Filter: (o.c < 2)
        Rows Removed by Filter: 9850
        Buffers: shared hit=74
  ->  Seq Scan on public.many m  (cost=0.00..9938.64 rows=19638 width=25) (actual time=0.005..85.212 rows=19796 loops=150)
        Output: m.a, m.b, m.c                                                                                 ⬆
        Filter: (m.c < 2)                                                         inner Seq Scan + Filter evaluated MANY times
        Rows Removed by Filter: 480815                                          (⚠ assign smaller table one as outer since we
        Buffers: shared hit=552150                                                expect less iterations over inner that way)
Planning time: 0.148 ms
Execution time: 15763.267 ms ⬅ slow
```

```
set enable_material = on;
```

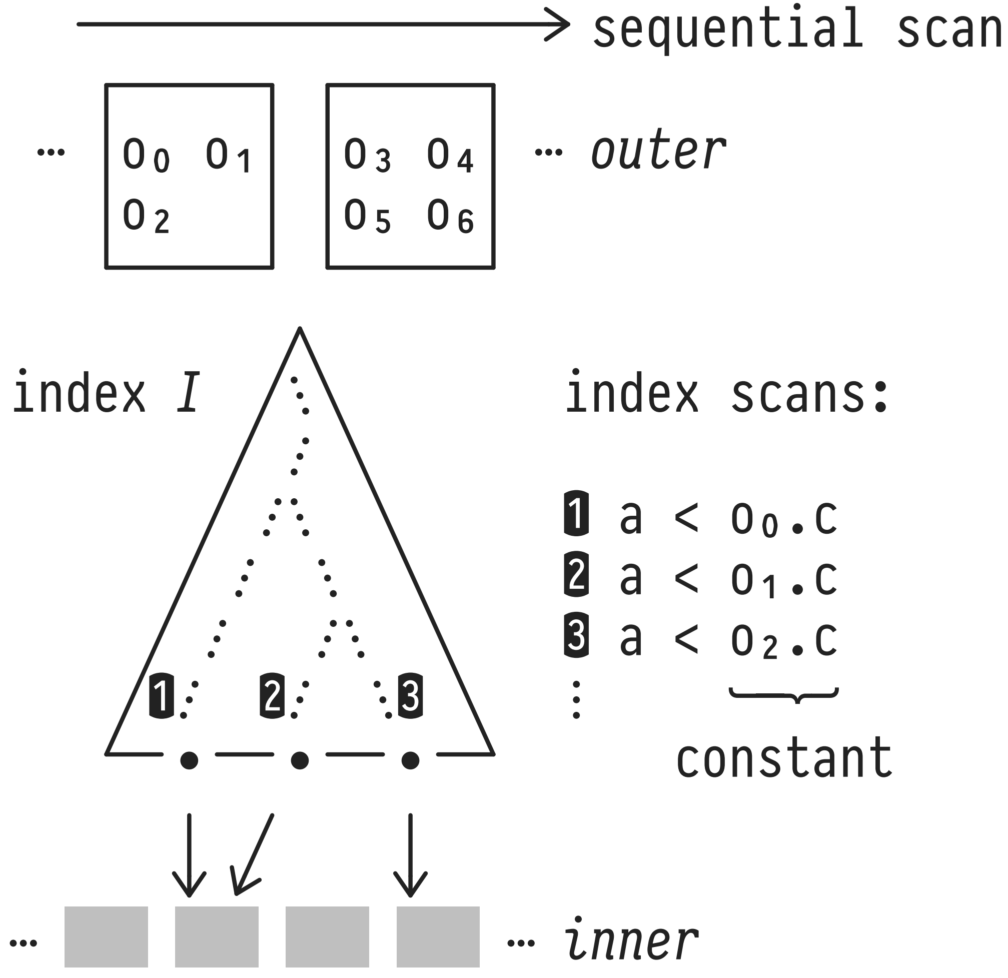
3 ¦ Index Nested Loop Join (INL⋈)

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 $i \in$ Index[Only]Scan($I$, $\underbrace{o\ \theta\ \square}_{\text{index condition}}$)
      append ‹$o$,$i$› to $j$;
  return $j$;
```

- *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⋈)



index scans:

❶ $a < o_0.c$
❷ $a < o_1.c$
❸ $a < \underbrace{o_2.c}_{\text{constant}}$
⋮

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

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

Demonstrate the use of INL⋈ in PostgreSQL.

```
-- ❶ Check indexes on table many
  \d many
                 Table "public.many"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text			
c	integer		not null	

```
Indexes:
    "many_a_c" PRIMARY KEY, btree (a, c)
Foreign-key constraints:
    "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)


-- ❷ Perform index nested loop join (predicate m.a < o.c is supported by index many_a_c)
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT *
    FROM   one AS o, many AS m
    WHERE  m.a < o.c AND o.b < 'a';
                          -- ^^^^^^^^^ added to keep result small
```

```
                                             QUERY PLAN
 Nested Loop  (cost=0.42..28933312.25 rows=1051283100 width=50) (actual time=0.202..7717.908 rows=16755613 loops=1)
   Output: o.a, o.b, o.c, m.a, m.b, m.c ⬅ ⚠️ no Join Filter
   ->  Seq Scan on public.one o  (cost=0.00..199.00 rows=6300 width=25) (actual time=0.049..17.195 rows=6249 loops=1)
         Output: o.a, o.b, o.c
         Filter: (o.b < 'a'::text) ⬅ predicate on table one pushed down
         Rows Removed by Filter: 3751
   ->  Index Scan using many_a_c on public.many m  (cost=0.42..2923.86 rows=166870 width=25) (actual time=0.016..0.661 rows=2681 loops=6249)
       ⬆ Output: m.a, m.b, m.c                                                                                              ⬆
         Index Cond: (m.a < o.c) ⬅ join condition turned into index condition                         6249 index scans performed
 Planning time: 0.283 ms
 Execution time: 8594.671 ms
```

```
-- ❸ ⚠️ Don't confuse the above with this plan which also uses Nested Loop Join + Index Scan (see Materialize)

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT *
    FROM   one AS o, many AS m
  WHERE  m.c < o.c AND m.a < 42;
```

```
                                                      QUERY PLAN
Nested Loop  (cost=0.42..375662.42 rows=8270000 width=50) (actual time=0.072..5792.337 rows=13165851 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Join Filter: (m.c < o.c) ⬅ join condition evaluated by expression evaluation, no index support
  Rows Removed by Join Filter: 7464149
  ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.019..3.585 rows=10000 loops=1)
        Output: o.a, o.b, o.c
  ->  Materialize  (cost=0.42..3344.62 rows=2481 width=25) (actual time=0.000..0.139 rows=2063 loops=10000)
      ⬆ Output: m.a, m.b, m.c
        ->  Index Scan using many_a_c on public.many m  (cost=0.42..3332.22 rows=2481 width=25) (actual time=0.045..1.375 rows=2063 loops=1)
            ⬆ Output: m.a, m.b, m.c                                                                                           ⬆
              Index Cond: (m.a < 42) ⬅ index only used to evaluate pushed down condition on table many              1 index scan performed
Planning time: 0.332 ms
Execution time: 6483.879 ms
```

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.

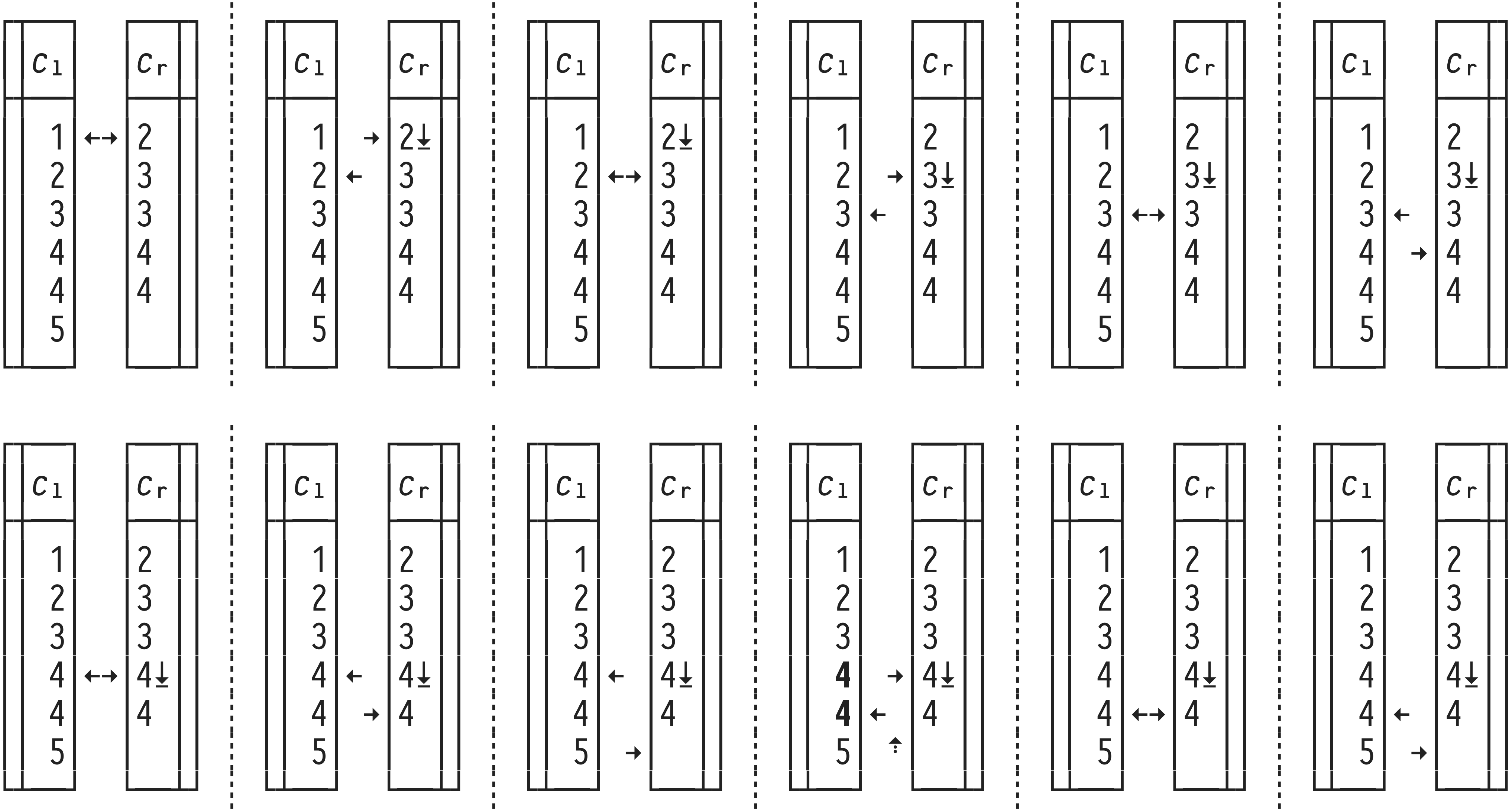
Merge Join Ingredients

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,$c_l$,$c_r$):
 j ← ∅;
 while left ≠ EOT ∧ right ≠ EOT          } 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;                     ⎭
   ⤓ ← right;                             } save current right pos
   while left.$c_l$ = ⤓.$c_r$                 } scan repeating left group
     right ← ⤓;                           } reset right scan
     while left.$c_l$ = right.$c_r$
       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?

- Root of subplan could be another Merge Join for example.
- Bitmap Index/Heap Scan will *not* fit the bill: the output of a Bitmap Heap Scan is ordered by RID (not by c_l/c_r).
- loop=1 in subplans: Merge Join scans both inputs only once.

Show plans that rely on Merge Join and establish sorted inputs in a variety of ways:

```
-- ❶ Check input tables one, many

  \d one
                Table "public.one"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text			
c	integer			

```
Indexes:
    "one_a" PRIMARY KEY, btree (a) CLUSTER
Referenced by:
    TABLE "many" CONSTRAINT "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)

  \d many
                Table "public.many"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text			
c	integer		not null	

```
Indexes:
    "many_a_c" PRIMARY KEY, btree (a, c) CLUSTER
Foreign-key constraints:
    "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)

  ANALYZE one;
  ANALYZE many;


-- ❷ Merge Join + Index Scan (left) + Sort (right)

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT o.a, o.b AS b1, m.b AS b2    -- replace m.b by m.c: Index Scan → Index Only Scan
    FROM   one AS o, many AS m
    WHERE  o.c = m.a;                   -- sort on m.a supported by index many_a_c
```

QUERY PLAN
Merge Join (cost=885.71..9406.59 rows=558525 width=38) (actual time=19.703..225.565 rows=545017 loops=1)

```
  Output: o.a, o.b, m.b
  Merge Cond: (m.a = o.c)
  ->  Index Scan using many_a_c on public.many m  (cost=0.42..21523.59 rows=500611 width=21) (actual time=0.038..6.158 rows=5511 loops=1)
      ☗ Output: m.a, m.b, m.c ⬅ join criterion m.a                                                                          ☗
  ->  Sort  (cost=838.39..863.39 rows=10000 width=25) (actual time=19.626..63.678 rows=545268 loops=1)
      ☗ Output: o.a, o.b, o.c                                                    ☗
        Sort Key: o.c ⬅ join criterion o.c
        Sort Method: quicksort  Memory: 1166kB
        ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.907..11.310 rows=10000 loops=1)
              Output: o.a, o.b, o.c
Planning time: 0.898 ms
Execution time: 257.058 ms
```

```
-- ❸ Carefully assess sort order of index scan to decide whether Sort is required

  set enable_hashjoin = off;

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT *
    FROM   one AS o, many AS m
    WHERE  o.c = m.c and m.a < 2;  -- m.a < 2 support by index many_a_c but output
                                   -- will NOT be sorted by m.c ⇒ Sort required
```

```
                                              QUERY PLAN

Merge Join  (cost=938.45..1037.04 rows=4848 width=50) (actual time=8.721..11.206 rows=3794 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Merge Cond: (m.c = o.c)
  ->  Sort  (cost=100.06..100.19 rows=52 width=25) (actual time=0.073..0.081 rows=38 loops=1)
      ☗ Output: m.a, m.b, m.c
        Sort Key: m.c ⬅ join criterion m.c
        Sort Method: quicksort  Memory: 27kB
        ->  Index Scan using many_a_c on public.many m  (cost=0.42..98.58 rows=52 width=25) (actual time=0.016..0.037 rows=38 loops=1)
              Output: m.a, m.b, m.c ⬅ index scan output sorted by a but NOT by c ⇒ Sort required
              Index Cond: (m.a < 2)
  ->  Sort  (cost=838.39..863.39 rows=10000 width=25) (actual time=8.639..9.352 rows=3795 loops=1)
      ☗ Output: o.a, o.b, o.c
        Sort Key: o.c ⬅ join criterion o.c
        Sort Method: quicksort  Memory: 1166kB
        ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.016..2.444 rows=10000 loops=1)
              Output: o.a, o.b, o.c
Planning time: 0.487 ms
Execution time: 11.619 ms
```

```
EXPLAIN (VERBOSE, ANALYZE)
  SELECT *
  FROM   one AS o, many AS m
  WHERE  o.c = m.c and m.a = 2;  -- m.a = 2 support by index many_a_c, index scans a = 2 group,
                                 -- output WILL be sorted by m.c ⇒ no Sort required
```

```
                                     QUERY PLAN
Merge Join  (cost=838.81..1043.53 rows=5127 width=50) (actual time=6.333..6.366 rows=54 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Inner Unique: true
  Merge Cond: (o.c = m.c)
  ->  Sort  (cost=838.39..863.39 rows=10000 width=25) (actual time=6.301..6.312 rows=55 loops=1)
      ☗ Output: o.a, o.b, o.c
        Sort Key: o.c ⬅ join criterion o.c
        Sort Method: quicksort  Memory: 1166kB
        ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.020..1.714 rows=10000 loops=1)
              Output: o.a, o.b, o.c
  ->  Index Scan using many_a_c on public.many m  (cost=0.42..103.88 rows=55 width=25) (actual time=0.025..0.026 rows=1 loops=1)
      ☗ Output: m.a, m.b, m.c
        Index Cond: (m.a = 2) ⬅ index scans a = 2 group only, inside that group, values are sorted by c
Planning time: 0.235 ms
Execution time: 6.419 ms
```

```
set enable_hashjoin = on;
```

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) ⌝
          ⌞                                                ⌟
```

Demonstrate the placement of Materialize above right subplan to support the re-scanning:

```
-- ➊ Subplan uses disk-based sort ⇒ use Materialize to support re-scanning of sort output

  set enable_hashjoin = off;
  show work_mem;
```

work_mem
4MB

```
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT *
    FROM   one AS o, many AS m
    WHERE  o.a = m.c;
```

```
                                                   QUERY PLAN
Merge Join  (cost=68106.27..76823.47 rows=500611 width=50) (actual time=335.405..684.592 rows=490665 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Merge Cond: (o.a = m.c)
  ->  Index Scan using one_a on public.one o  (cost=0.29..347.29 rows=10000 width=25) (actual time=0.020..0.244 rows=101 loops=1)
        Output: o.a, o.b, o.c
  ->  Materialize  (cost=68058.78..70561.83 rows=500611 width=25) (actual time=329.446..529.300 rows=500611 loops=1)
      ⬆ Output: m.a, m.b, m.c                                                                                   ⬆
        ->  Sort  (cost=68058.78..69310.30 rows=500611 width=25) (actual time=329.442..448.616 rows=500611 loops=1)
            ⬆ Output: m.a, m.b, m.c                                                                          ⬆
              Sort Key: m.c
              Sort Method: external merge  Disk: 18640kB ⬅ external sort, result in temporary disk file
              ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.014..70.973 rows=500611 loops=1)
                    Output: m.a, m.b, m.c
Planning time: 0.368 ms
Execution time: 716.761 ms
```

```
-- ➋ Increased work_mem to enable in-memory sort ⇒ resulting buffer supports re-scanning, no Materialize needed

  set work_mem = '64MB';  -- sufficient work memory to enable in-memory sort

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT *
    FROM   one AS o, many AS m
    WHERE  o.a = m.c;
```

```
                                                  QUERY PLAN
Merge Join  (cost=56125.77..63591.45 rows=500611 width=50) (actual time=234.355..466.343 rows=490665 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Merge Cond: (o.a = m.c)
  ->  Index Scan using one_a on public.one o  (cost=0.29..347.29 rows=10000 width=25) (actual time=0.015..0.248 rows=101 loops=1)
        Output: o.a, o.b, o.c
  ->  Sort  (cost=56078.28..57329.80 rows=500611 width=25) (actual time=230.546..301.350 rows=500611 loops=1)
      ⬆ Output: m.a, m.b, m.c
        Sort Key: m.c
        Sort Method: quicksort  Memory: 51399kB ⬅ in-memory sort, creates re-scannable memory buffer
        ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.013..69.545 rows=500611 loops=1)
              Output: m.a, m.b, m.c
Planning time: 0.323 ms
Execution time: 501.520 ms
```

```
set enable_hashjoin = on;
set work_mem = default;
```

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.

Demonstrate that PostgreSQL tracks interesting orders that have influence on subplans (even if these subplans themselves do not benefit). Overall plan cost improves.

```
-- ❶ No interesting order: use Hash Join (delivers rows in arbitrary order)

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT o.a, o.b || m.b AS b
    FROM   one AS o, many AS m
    WHERE  o.a = m.a;
```

```
                                        QUERY PLAN
Hash Join  (cost=299.00..16557.41 rows=500611 width=36) (actual time=6.360..335.689 rows=500611 loops=1)
☗ Output: o.a, (o.b || m.b)
  Inner Unique: true
  Hash Cond: (m.a = o.a)
  ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=21) (actual time=0.012..62.179 rows=500611 loops=1)
        Output: m.a, m.b, m.c
  ->  Hash  (cost=174.00..174.00 rows=10000 width=21) (actual time=6.319..6.319 rows=10000 loops=1)
        Output: o.a, o.b
        Buckets: 16384  Batches: 1  Memory Usage: 646kB
        ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=21) (actual time=0.010..2.975 rows=10000 loops=1)
              Output: o.a, o.b
Planning time: 0.578 ms
Execution time: 363.533 ms
```

```
-- ❷ Interesting order o.a that coincides with join condition: use Merge Join

  EXPLAIN (VERBOSE, ANALYZE)
    SELECT o.a, o.b || m.b AS b
    FROM   one AS o, many AS m
    WHERE  o.a = m.a
    ORDER BY o.a;             -- or: ORDER BY m.a
```

```
                                        QUERY PLAN
Merge Join  (cost=0.82..29405.04 rows=500611 width=36) (actual time=0.032..421.304 rows=500611 loops=1)
☗ Output: o.a, (o.b || m.b)
  Merge Cond: (o.a = m.a)
  ->  Index Scan using one_a on public.one o  (cost=0.29..347.29 rows=10000 width=21) (actual time=0.013..3.813 rows=10000 loops=1)
        Output: o.a, o.b, o.c
  ->  Index Scan using many_a_c on public.many m  (cost=0.42..21523.59 rows=500611 width=21) (actual time=0.013..164.813 rows=500611 loops=1)
        Output: m.a, m.b, m.c
Planning time: 0.407 ms
Execution time: 454.025 ms
```

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_1 is unique.
    - See Merge Join plan property: Inner Unique: true.
    - Algorithm MergeJoinUnique(*left*,*right*,c_1,c_r) requires no management of ↓ at all. **Q:** Simplified code?

Demonstrate that PostgreSQL chooses Merge Join if the join inputs are large (in the example below, the output has 1000 rows only and thus is small) AND at least one join criteion is unique (thus no rescanning):

```
-- ❶ Build large (10⁶ rows) tables left/right with unique join criteria

DROP TABLE IF EXISTS "left";
DROP TABLE IF EXISTS "right";

CREATE TABLE "left"  (a int, b text);
CREATE TABLE "right" (a int, b text);

INSERT INTO "left" (a,b)
  SELECT i, md5(i::text)
  FROM   generate_series(1, 1000000) AS i;

INSERT INTO "right" (a,b)
  SELECT i + 999000, md5(i::text)       -- ⚠ overlap of left.a and right.a of 1000 rows only
  FROM   generate_series(1, 1000000) AS i;

-- Table left:
--
-- ┌─────────┬───────────────────────────────────┐
-- │    a    │                 b                 │
-- ├─────────┼───────────────────────────────────┤
-- │       1 │ c4ca4238a0b923820dcc509a6f75849b  │
-- │       2 │ c81e728d9d4c2f636f067f89cc14862c  │
-- │       3 │ eccbc87e4b5ce2fe28308fd9f2a7baf3  │
-- [...]                                           ┐
-- │  999998 │ 755af25720023b2f852105910b125ecc  │ } 1000 rows of overlap with right
-- │  999999 │ 52c69e3a57331081823331c4e69d3f2e  │ │
-- │ 1000000 │ 8155bc545f84d9652f1012ef2bdfb6eb  │ ┘
-- └─────────┴───────────────────────────────────┘
--
-- Table right:
--
-- ┌─────────┬───────────────────────────────────┐
-- │    a    │                 b                 │
-- ├─────────┼───────────────────────────────────┤
-- │  999001 │ c4ca4238a0b923820dcc509a6f75849b  │ ┐
-- │  999002 │ c81e728d9d4c2f636f067f89cc14862c  │ } 1000 rows of overlap with left
-- │  999003 │ eccbc87e4b5ce2fe28308fd9f2a7baf3  │ │
-- [...]                                           ┘
-- │ 1998998 │ 755af25720023b2f852105910b125ecc  │
-- │ 1998999 │ 52c69e3a57331081823331c4e69d3f2e  │
-- │ 1999000 │ 8155bc545f84d9652f1012ef2bdfb6eb  │
-- └─────────┴───────────────────────────────────┘
```

```
CREATE UNIQUE INDEX left_a  ON "left"  USING btree (a);  -- the join columns ARE
CREATE UNIQUE INDEX right_a ON "right" USING btree (a);  -- indeed unique!

ANALYZE "left";
ANALYZE "right";

-- ❷ Equi-join of two large tables with unique join criteria

EXPLAIN (VERBOSE, ANALYZE)
  SELECT l.b AS b1, r.b AS b2
  FROM   "left" AS l, "right" AS r
  WHERE  l.a = r.a;
```

```
                                          QUERY PLAN
Merge Join  (cost=36782.51..46856.29 rows=1000000 width=66) (actual time=456.135..457.284 rows=1000 loops=1)
⬆ Output: l.b, r.b
  Inner Unique: true ⬅
  Merge Cond: (l.a = r.a)
  Buffers: shared hit=6094 read=4989  ⬅ 11083 buffer reads
  ->  Index Scan using left_a on left l  (cost=0.42..34317.43 rows=1000000 width=37) (actual time=0.171..290.515 rows=1000000 loops=1)
        Output: l.a, l.b
        Buffers: shared hit=6080 read=4989
  ->  Index Scan using right_a on right r  (cost=0.42..34317.43 rows=1000000 width=37) (actual time=0.032..0.279 rows=1000 loops=1)
        Output: r.a, r.b                                                                                                  ⬆
        Buffers: shared hit=14                                                                          only 1000 rows of right scanned,
Planning time: 0.513 ms                                                    once Merge Join has scanned to the end of left, it is obvious
Execution time: 457.405 ms ⬅ fast                                                                  that no more matches will be possible
```

```
-- ❸ Repeat equi-join, but now Merge Join disabled, default working memory

  set enable_mergejoin = off;

  show work_mem;
```

work_mem
4MB

```
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT l.b AS b1, r.b AS b2
    FROM   "left" AS l, "right" AS r
    WHERE  l.a = r.a;
```

```
                                        QUERY PLAN
```

```
Hash Join  (cost=38647.00..93044.99 rows=1000000 width=66) (actual time=671.588..1888.733 rows=1000 loops=1)
▲ Output: l.b, r.b
  Inner Unique: true
  Hash Cond: (r.a = l.a)
  Buffers: shared hit=4689 read=11979, temp read=14282 written=14220 ⬅ 45150 buffer slots accesses
  ->  Seq Scan on public.right r  (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.027..300.024 rows=1000000 loops=1)
        Output: r.b, r.a
        Buffers: shared hit=105 read=8229
  ->  Hash  (cost=18334.00..18334.00 rows=1000000 width=37) (actual time=669.885..669.885 rows=1000000 loops=1)
        Output: l.b, l.a         ▼
        Buckets: 65536  Batches: 32  Memory Usage: 2717kB
        Buffers: shared hit=4584 read=3750, temp written=7080
        ->  Seq Scan on public.left l  (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.065..272.988 rows=1000000 loops=1)
              Output: l.b, l.a
              Buffers: shared hit=4584 read=3750
Planning time: 0.372 ms
Execution time: 1888.986 ms ⬅ slow
```

```
-- Hash Join suffers if we reduce the available working memory

set work_mem = '64kB';

EXPLAIN (VERBOSE, ANALYZE)
  SELECT l.b AS b1, r.b AS b2
  FROM   "left" AS l, "right" AS r
  WHERE  l.a = r.a;
```

```
                                                                QUERY PLAN
Hash Join  (cost=38647.00..93044.99 rows=1000000 width=66) (actual time=2786.864..7105.444 rows=1000 loops=1)
▲ Output: l.b, r.b
  Inner Unique: true
  Hash Cond: (r.a = l.a)
  Buffers: shared hit=8503 read=8165, temp read=20486 written=16392 ⬅ 53546 buffer slots accesses
  ->  Seq Scan on public.right r  (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.036..436.015 rows=1000000 loops=1)
        Output: r.b, r.a
        Buffers: shared hit=169 read=8165
  ->  Hash  (cost=18334.00..18334.00 rows=1000000 width=37) (actual time=2622.111..2622.111 rows=1000000 loops=1)
        Output: l.b, l.a          ▼
        Buckets: 1024  Batches: 2048  Memory Usage: 42kB
        Buffers: shared hit=8334, temp written=6148
        ->  Seq Scan on public.left l  (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.027..387.493 rows=1000000 loops=1)
              Output: l.b, l.a
              Buffers: shared hit=8334
Planning time: 0.425 ms
Execution time: 7106.520 ms ⬅ super slow
```

```
-- ❹ Repeat equi-join, re-enable Merge Join, leave working memory constrained (64kB)

  set enable_mergejoin = on;

  show work_mem;
```

work_mem
64kB

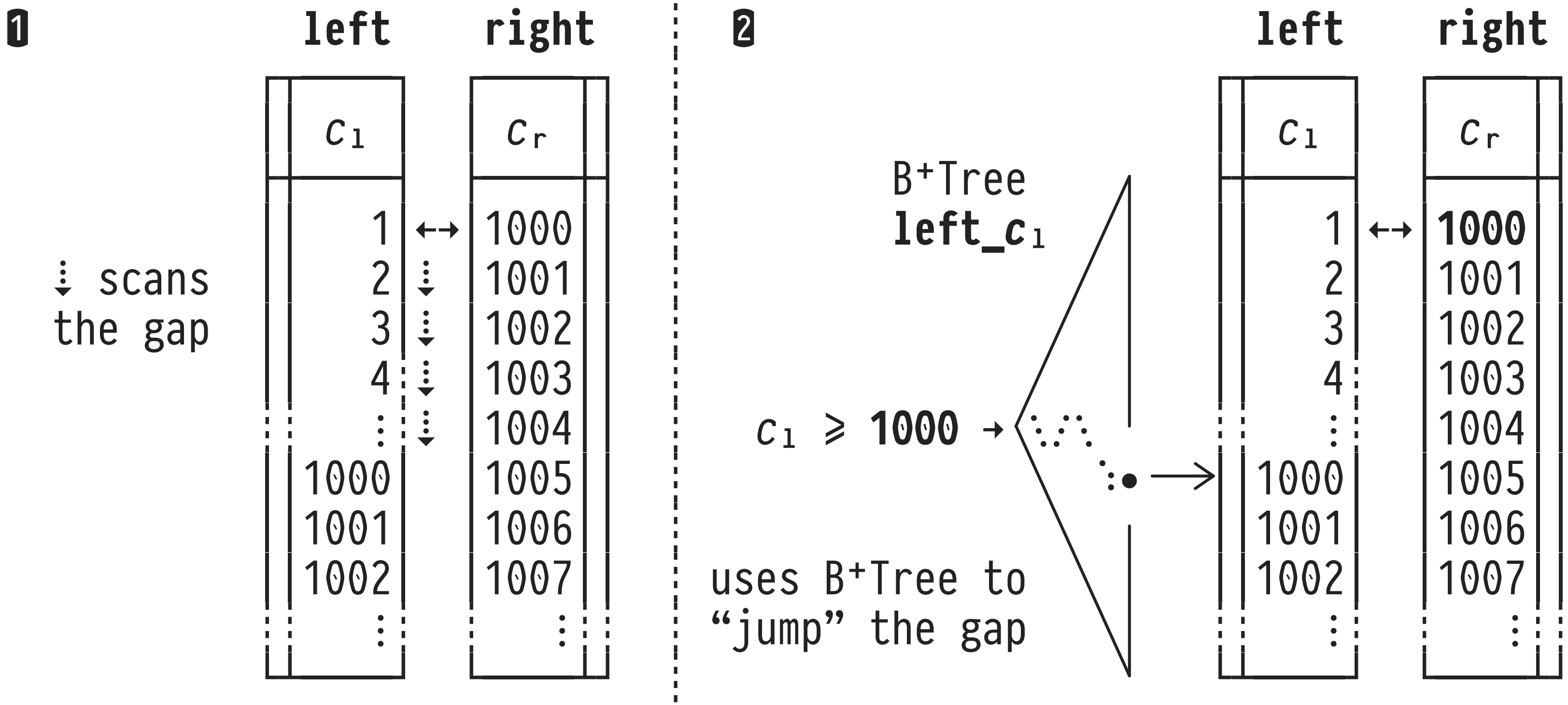
```
  EXPLAIN (VERBOSE, ANALYZE)
    SELECT l.b AS b1, r.b AS b2
    FROM   "left" AS l, "right" AS r
    WHERE  l.a = r.a;
```

```
                                        QUERY PLAN
Merge Join  (cost=36782.51..46856.29 rows=1000000 width=66) (actual time=397.223..398.391 rows=1000 loops=1)
☗ Output: l.b, r.b
  Inner Unique: true
  Merge Cond: (r.a = l.a)
  Buffers: shared hit=11083 ⬅ 11083 buffer reads, just like with 4MB of working memory
  ->  Index Scan using right_a on right r  (cost=0.42..34317.43 rows=1000000 width=37) (actual time=0.045..0.281 rows=1001 loops=1)
        Output: r.a, r.b
        Buffers: shared hit=14
  ->  Index Scan using left_a on left l  (cost=0.42..34317.43 rows=1000000 width=37) (actual time=0.045..251.302 rows=1000000 loops=1)
        Output: l.a, l.b
        Buffers: shared hit=11069
Planning time: 0.805 ms
Execution time: 398.538 ms ⬅ as fast as with 4MB working memory
```

```
  set work_mem = default;
```

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:



❶

⇣ scans the gap

left c_l	right c_r
1	1000
2	1001
3	1002
4	1003
⋮	1004
1000	1005
1001	1006
1002	1007
⋮	⋮

❷

B+Tree **left_c_l**

$c_l \geqslant$ **1000** →

uses B+Tree to “jump” the gap

left c_l	right c_r
1	**1000**
2	1001
3	1002
4	1003
⋮	1004
1000	1005
1001	1006
1002	1007
⋮	⋮

Demonstrate the “jumping the gap” technique. Recursive query simulates the lock-step movement of ←, →. Relies on B$^+$Tree index support on columns left.a, right.a.

Speed-up in this example essentially relies on value distribution in columns left.a, right.a (VERY large gap).

⚠️ Only works if columns left.a, right.a are unique (no support for repeating groups).

```
-- 1 Original scan-based Merge Join

  EXPLAIN (ANALYZE)
    SELECT l.b AS b1, r.b AS b2
    FROM   "left" AS l, "right" AS r
    WHERE  l.a = r.a;
```

```
                                                   QUERY PLAN
Merge Join  (cost=36782.73..46852.94 rows=1000000 width=66) (actual time=433.528..434.809 rows=1000 loops=1)
  Merge Cond: (l.a = r.a)
  ->  Index Scan using left_a on "left" l  (cost=0.42..34317.43 rows=1000000 width=37) (actual time=0.013..290.806 rows=1000000 loops=1)
  ->  Index Scan using right_a on "right" r  (cost=0.42..34317.43 rows=1000000 width=37) (actual time=0.011..0.304 rows=1000 loops=1)
Planning time: 0.281 ms
Execution time: 434.892 ms ⬅ slow ;-)
```

```
-- 2 B+Tree-based gap jumping

EXPLAIN (ANALYZE)
  WITH RECURSIVE merge(l, r) AS (
    SELECT
      (SELECT l FROM "left"  AS l ORDER BY l.a LIMIT 1),
      (SELECT r FROM "right" AS r ORDER BY r.a LIMIT 1)

    UNION ALL

    SELECT
      CASE WHEN (m.l).a < (m.r).a THEN
             (SELECT l1 FROM "left" AS l1 WHERE l1.a >= (m.r).a ORDER BY l1.a LIMIT 1) -- let ← jump forward using the index
           WHEN (m.l).a = (m.r).a THEN
             (SELECT l1 FROM "left" AS l1 WHERE l1.a >  (m.r).a ORDER BY l1.a LIMIT 1) -- let ← jump forward using the index
           ELSE m.l
      END,
      CASE WHEN (m.r).a < (m.l).a THEN
             (SELECT r1 FROM "right" AS r1 WHERE r1.a >= (m.l).a ORDER BY r1.a LIMIT 1) -- let → jump forward using the index (never executed)
           WHEN (m.r).a = (m.l).a THEN
             (SELECT r1 FROM "right" AS r1 WHERE r1.a >  (m.l).a ORDER BY r1.a LIMIT 1) -- let → jump forward using the index
```

```
          ELSE m.r
        END
      FROM   merge AS m
      WHERE  m IS NOT NULL -- m.l and/or m.r may be NULL if there is no larger value to jump forward to
    )
SELECT (m.l).b AS b1, (m.r).b AS b2
FROM   merge AS m
WHERE  (m.l).a = (m.r).a;
```

```
                                              QUERY PLAN

 CTE Scan on merge m  (cost=190.68..192.95 rows=1 width=64) (actual time=0.105..38.068 rows=1000 loops=1)
   Filter: ((l).a = (r).a)
   Rows Removed by Filter: 2
   CTE merge
     ->  Recursive Union  (cost=0.92..190.68 rows=101 width=64) (actual time=0.058..35.807 rows=1002 loops=1)
           ->  Result  (cost=0.92..0.93 rows=1 width=64) (actual time=0.057..0.057 rows=1 loops=1)
                 InitPlan 1 (returns $1)
                   ->  Limit  (cost=0.42..0.46 rows=1 width=65) (actual time=0.035..0.035 rows=1 loops=1)
                         ->  Index Scan using left_a on "left" l  (cost=⋯) (actual time=0.034..0.034 rows=1 loops=1)
                 InitPlan 2 (returns $2)
                   ->  Limit  (cost=0.42..0.46 rows=1 width=65) (actual time=0.019..0.019 rows=1 loops=1)
                         ->  Index Scan using right_a on "right" r  (cost=⋯) (actual time=0.017..0.017 rows=1 loops=1)
           ->  WorkTable Scan on merge m_1  (cost=0.00..18.77 rows=10 width=64) (actual time=0.033..0.034 rows=1 loops=1002)
                 Filter: (m_1.* IS NOT NULL)
                 Rows Removed by Filter: 0
                 SubPlan 3
                   ->  Limit  (cost=0.42..0.46 rows=1 width=65) (actual time=0.020..0.020 rows=1 loops=1)
                         ->➨Index Scan using left_a on "left" l1  (cost=⋯) (actual time=0.019..0.019 rows=1 loops=1)
                               Index Cond: (a >= (m_1.r).a)
                 SubPlan 4
                   ->  Limit  (cost=0.42..0.46 rows=1 width=65) (actual time=0.013..0.013 rows=1 loops=1000)
                         ->➨Index Scan using left_a on "left" l1_1  (cost=⋯) (actual time=0.013..0.013 rows=1 loops=1000)
                               Index Cond: (a > (m_1.r).a)
                 SubPlan 5
                   ->  Limit  (cost=0.42..0.46 rows=1 width=65) (never executed)
                         ->➨Index Scan using right_a on "right" r1  (cost=⋯) (never executed)
                               Index Cond: (a >= (m_1.l).a)
                 SubPlan 6
                   ->  Limit  (cost=0.42..0.46 rows=1 width=65) (actual time=0.015..0.015 rows=1 loops=1000)
                         ->➨Index Scan using right_a on "right" r1_1  (cost=⋯) (actual time=0.014..0.014 rows=1 loops=1000)
                               Index Cond: (a > (m_1.l).a)
 Planning time: 0.450 ms
 Execution time: 40.453 ms ⬅ fast
```

```
-- Table merge:
--  ┌──────────────────────┬──────────────────────┐
-- │                   l                    │                   r                    │
-- ├────────────────────────────────────────┼────────────────────────────────────────┤
-- │ (1,c4ca4238a0b923820dcc509a6f75849b)    │ (999001,c4ca4238a0b923820dcc509a6f75849b) │
-- │ (999001,9ea9c185834db3573483b76a48f25d0d) │ (999001,c4ca4238a0b923820dcc509a6f75849b) │
-- │ (999002,216fb4bed5d9d69a1d58aecaacaa25b3) │ (999002,c81e728d9d4c2f636f067f89cc14862c) │
-- │ (999003,b727d263986b2ff91f9ebba315e3c7c3) │ (999003,eccbc87e4b5ce2fe28308fd9f2a7baf3) │
-- │ (999004,aaee4bf603dd73e58906764d27c3d33a) │ (999004,a87ff679a2f3e71d9181a67b7542122c) │
-- [...]
```

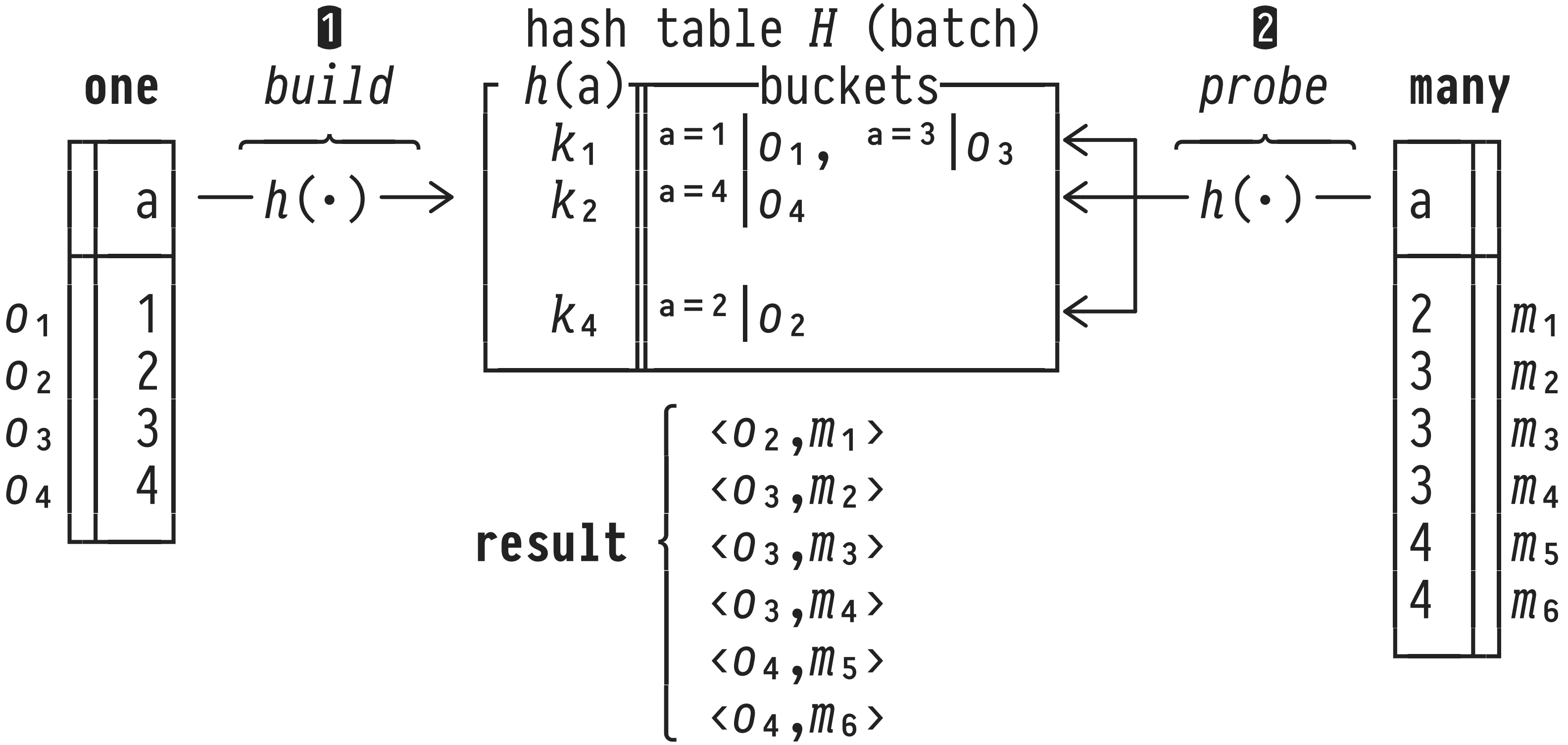
Equi-joins—*e.g.*, foreign-key joins—are arguably the most prominent kinds of relational join. Merge Join relies on *sorting* while **Hash Join** uses *hashing* to perform equi-joins:

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.



Demonstrate Hash Join in PostgreSQL plans:

```
-- ❶ Check input tables

  \d one
                 Table "public.one"
┌────────┬─────────┬───────────┬──────────┬─────────┐
│ Column │  Type   │ Collation │ Nullable │ Default │
├────────┼─────────┼───────────┼──────────┼─────────┤
│ a      │ integer │           │ not null │         │
│ b      │ text    │           │          │         │
│ c      │ integer │           │          │         │
└────────┴─────────┴───────────┴──────────┴─────────┘
Indexes:
    "one_a" PRIMARY KEY, btree (a) CLUSTER
Referenced by:
    TABLE "many" CONSTRAINT "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)

  \d many
                 Table "public.many"
┌────────┬─────────┬───────────┬──────────┬─────────┐
│ Column │  Type   │ Collation │ Nullable │ Default │
├────────┼─────────┼───────────┼──────────┼─────────┤
│ a      │ integer │           │ not null │         │
│ b      │ text    │           │          │         │
│ c      │ integer │           │ not null │         │
└────────┴─────────┴───────────┴──────────┴─────────┘
Indexes:
    "many_a_c" PRIMARY KEY, btree (a, c) CLUSTER
Foreign-key constraints:
    "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)


-- ❷ Equi-join is performed via Hash Join (the smaller one table becomes the build table)

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT o.*, m.*
    FROM   one AS o, many AS m
    WHERE  o.a = m.a;
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                                      QUERY PLAN                                                      │
├──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┤
│ Hash Join  (cost=299.00..15305.88 rows=500611 width=50) (actual time=6.796..320.629 rows=500611 loops=1)
│ ☗ Output: o.a, o.b, o.c, m.a, m.b, m.c
│   Inner Unique: true
│   Hash Cond: (m.a = o.a)
```

```
   Buffers: shared hit=3755
   ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.010..64.297 rows=500611 loops=1)
         Output: m.a, m.b, m.c
         Buffers: shared hit=3681
   ->  Hash  (cost=174.00..174.00 rows=10000 width=25) (actual time=6.772..6.772 rows=10000 loops=1)
        ⬆ Output: o.a, o.b, o.c    ⬇ all of H fits into working memory, no iteration needed
#buckets➡Buckets: 16384  Batches: 1  Memory Usage: 714kB ⬅ size of hash table H
         Buffers: shared hit=74
         ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.005..2.538 rows=10000 loops=1)
               Output: o.a, o.b, o.c
               Buffers: shared hit=74
 Planning time: 0.399 ms
 Execution time: 360.462 ms
```

```
-- ❸ Requiring less columns from the build table (semi-join): can build more compact hash table
  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT m.*
    FROM   one AS o, many AS m
    WHERE  o.a = m.a;
```

```
                                                             QUERY PLAN

 Hash Join  (cost=299.00..15305.88 rows=500611 width=25) (actual time=17.331..433.805 rows=500611 loops=1)
   Output: m.a, m.b, m.c
   Inner Unique: true
   Hash Cond: (m.a = o.a)
   Buffers: shared hit=3755
   ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.018..93.734 rows=500611 loops=1)
         Output: m.a, m.b, m.c
         Buffers: shared hit=3681
   ->  Hash  (cost=174.00..174.00 rows=10000 width=4) (actual time=17.279..17.279 rows=10000 loops=1)
         Output: o.a
         Buckets: 16384  Batches: 1  Memory Usage: 480kB ⬅ smaller hash table H
         Buffers: shared hit=74
         ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=4) (actual time=0.034..7.053 rows=10000 loops=1)
               Output: o.a ⬅ narrow build rows
               Buffers: shared hit=74
 Planning time: 1.248 ms
 Execution time: 488.840 ms
```

```
-- ❹ Reduce working memory: split build table in partitions, iterate build/probe phases

  set work_mem = '64kB';

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
```

```
    SELECT m.*
    FROM   one AS o, many AS m
    WHERE  o.a = m.a;
```

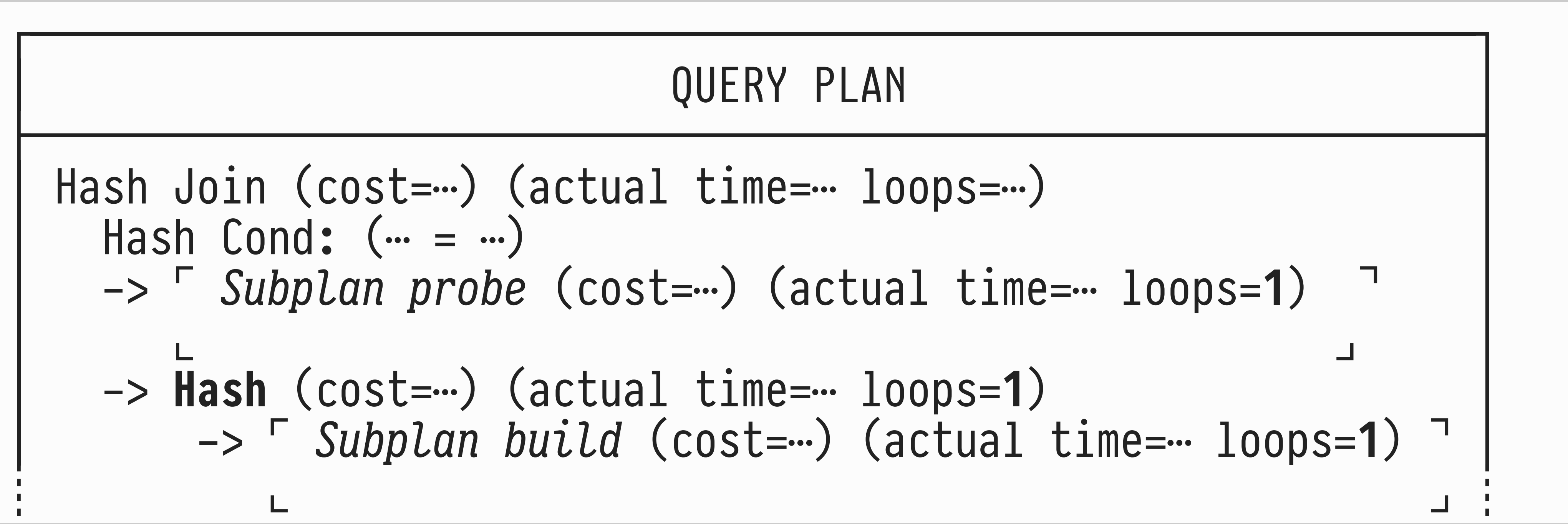
```
                                               QUERY PLAN
 Hash Join  (cost=339.00..22231.88 rows=500611 width=25) (actual time=8.740..452.992 rows=500611 loops=1)
   Output: m.a, m.b, m.c
   Inner Unique: true
   Hash Cond: (m.a = o.a)
   Buffers: shared hit=3755, temp read=2840 written=2810
   ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.008..72.534 rows=500611 loops=1)
         Output: m.a, m.b, m.c
         Buffers: shared hit=3681
   ->  Hash  (cost=174.00..174.00 rows=10000 width=4) (actual time=6.895..6.895 rows=10000 loops=1)
         Output: o.a             ⬇ iterate build/probe phases (iteration ≡ batch)
#buckets➡Buckets: 2048  Batches: 16  Memory Usage: 36kB
         Buffers: shared hit=74, temp written=15 ⬅ 15 of 16 batches written to disk
         ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=4) (actual time=0.007..2.436 rows=10000 loops=1)
               Output: o.a
               Buffers: shared hit=74
 Planning time: 0.279 ms
 Execution time: 493.360 ms
```

```
  set work_mem = default;
```

Hash Join: Pseudo Code

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

  for $b \in build$                                   } ❶ build
    insert $b$ into bucket $H[h(b.c_l)]$;                  phase

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

  return $j$;
```

Hash Join: Execution Plan



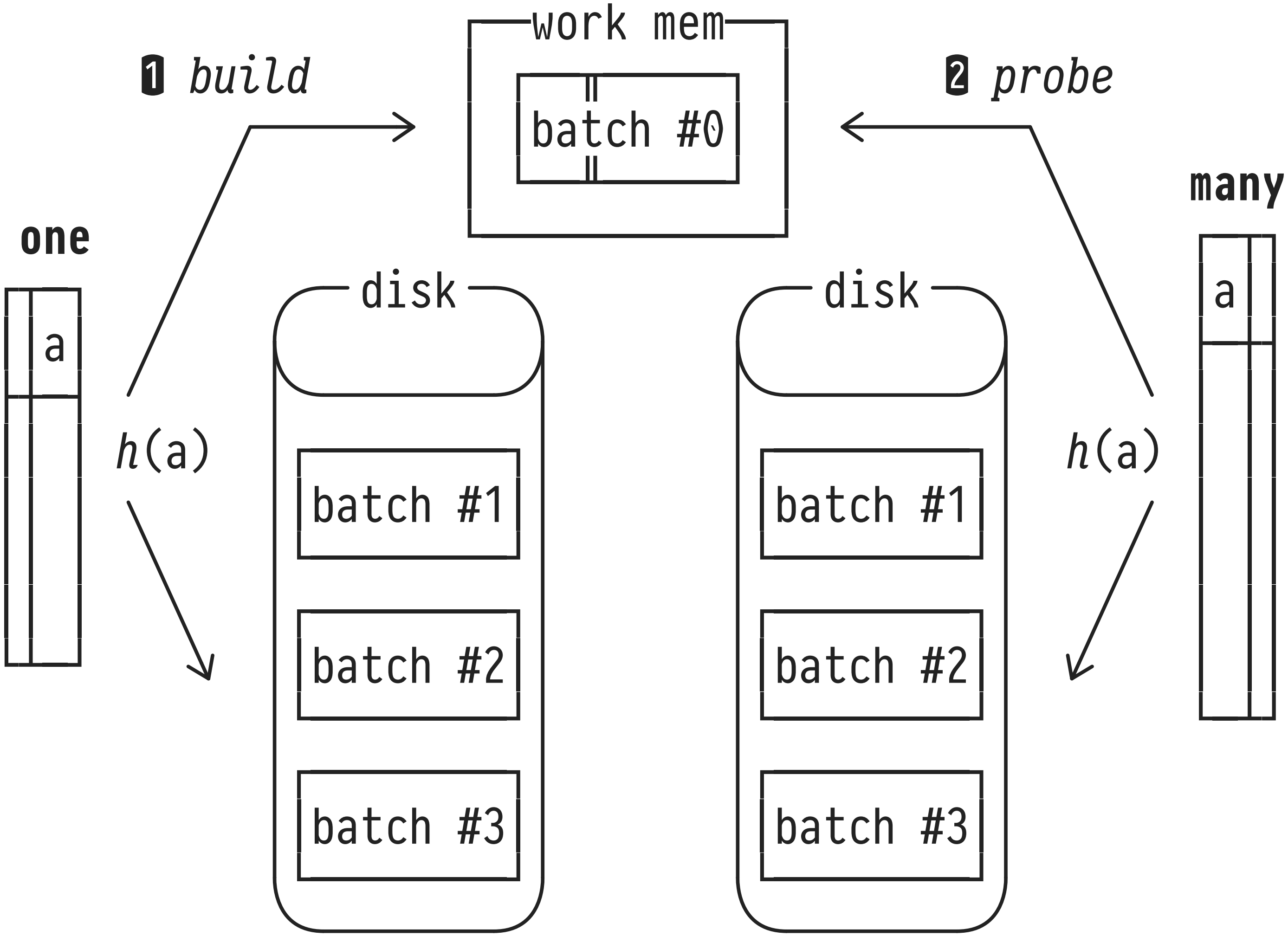
```
                              QUERY PLAN
Hash Join (cost=…) (actual time=… loops=…)
  Hash Cond: (… = …)
  -> ⌜ Subplan probe (cost=…) (actual time=… loops=1)  ⌝
     ⌞                                                ⌟
  -> Hash (cost=…) (actual time=… loops=1)
      -> ⌜ Subplan build (cost=…) (actual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

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

Demonstrate the creation of batches during Hash Join when the available working memory is decreased:

```
-- ❶ Check the input tables and working memory

  \d one
                 Table "public.one"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text			
c	integer			

```
Indexes:
    "one_a" PRIMARY KEY, btree (a) CLUSTER
Referenced by:
    TABLE "many" CONSTRAINT "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)

  \d many
                 Table "public.many"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text			
c	integer		not null	

```
Indexes:
    "many_a_c" PRIMARY KEY, btree (a, c) CLUSTER
Foreign-key constraints:
    "many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)


  show work_mem;
```

work_mem
4MB

```
-- ❷ Perform Hash Joins with decreasing memory

EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT *
  FROM   one AS o, many AS m
```

```
WHERE  o.a = m.a;
```

```
                                   QUERY PLAN
Hash Join  (cost=299.00..15305.88 rows=500611 width=50) (actual time=4.405..330.786 rows=500611 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Inner Unique: true
  Hash Cond: (m.a = o.a)
  Buffers: shared hit=3755
  ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.009..69.175 rows=500611 loops=1)
        Output: m.a, m.b, m.c
        Buffers: shared hit=3681
  ->  Hash  (cost=174.00..174.00 rows=10000 width=25) (actual time=4.361..4.361 rows=10000 loops=1)
        Output: o.a, o.b, o.c
     ➡ Buckets: 16384  Batches: 1  Memory Usage: 714kB
        Buffers: shared hit=74   ⬆
        ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.006..1.630 rows=10000 loops=1)
              Output: o.a, o.b, o.c
              Buffers: shared hit=74 ⬅ all 74 blocks of table one (build) have been read
Planning time: 0.291 ms
Execution time: 370.801 ms ⬅ fast
```

- PostgreSQL aims for a bucket length (rows per bucket) of ⩽ 10 to avoid long intra-bucket searches.

```
set work_mem = '512kB';

EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT *
  FROM   one AS o, many AS m
  WHERE  o.a = m.a;
```

```
                                   QUERY PLAN
Hash Join  (cost=368.00..22289.88 rows=500611 width=50) (actual time=9.001..422.845 rows=500611 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Inner Unique: true
  Hash Cond: (m.a = o.a)
  Buffers: shared hit=3755, temp read=1473 written=1471 ⬅ I/O on batches #1, #2, ...
  ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.013..69.465 rows=500611 loops=1)
        Output: m.a, m.b, m.c
        Buffers: shared hit=3681
  ->  Hash  (cost=174.00..174.00 rows=10000 width=25) (actual time=8.956..8.956 rows=10000 loops=1)
        Output: o.a, o.b, o.c   ⬇
smaller➡ Buckets: 8192  Batches: 2  Memory Usage: 368kB
|hash     Buffers: shared hit=74, temp written=28
```

```
table    ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.007..2.343 rows=10000 loops=1)
               Output: o.a, o.b, o.c
               Buffers: shared hit=74
 Planning time: 0.356 ms
 Execution time: 460.667 ms ⬅ slower
```

```
  set work_mem = '256kB';

  EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
    SELECT *
    FROM   one AS o, many AS m
    WHERE  o.a = m.a;
```

```
                                                      QUERY PLAN
 Hash Join  (cost=368.00..22289.88 rows=500611 width=50) (actual time=6.911..458.826 rows=500611 loops=1)
   Output: o.a, o.b, o.c, m.a, m.b, m.c
   Inner Unique: true
   Hash Cond: (m.a = o.a)
   Buffers: shared hit=3755, temp read=2253 written=2247 ⬅ even more I/O
   ->  Seq Scan on public.many m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.010..73.295 rows=500611 loops=1)
         Output: m.a, m.b, m.c
         Buffers: shared hit=3681
   ->  Hash  (cost=174.00..174.00 rows=10000 width=25) (actual time=6.653..6.653 rows=10000 loops=1)
         Output: o.a, o.b, o.c   ⬇
      ➡ Buckets: 4096  Batches: 4  Memory Usage: 182kB
         Buffers: shared hit=74, temp written=43
         ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.007..2.017 rows=10000 loops=1)
               Output: o.a, o.b, o.c
               Buffers: shared hit=74
 Planning time: 0.339 ms
 Execution time: 497.421 ms ⬅ even slower
```

```
-- ❸ Create a variant of table many which is super-heavily skewed:
--   all rows in table many have a = 1.  Expect build/probe rows
--   with a = 1 to be place in in-memory skew batch.  All probe
--   rows will hit the skew batch, no probe row will be place in
--   on-disk batches:

  CREATE TABLE many1 AS
    SELECT 1 AS a, m.b, m.c
    FROM   many AS m;

  ANALYZE many1;
```

```
-- Check column statistics (e.g., most common values)

SELECT attname, n_distinct, null_frac, most_common_vals
FROM   pg_stats
WHERE  tablename = 'many1' AND attname IN ('a', 'c');
```

attname	n_distinct	null_frac	most_common_vals
a	1	0	{1}
c	101	0	{0,1,6,3,4,9,8,7,5,2,10,15,11,18,16,13,12,14,19,20,21,22,23,17,26,24,30,29,27,25,28,32,31,33,38,37,34,...}

```
 # of distinct values   fraction of                    array of most common values (mcv) in column
 in columns             NULLs in column
```

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT *
  FROM   one AS o, many1 AS m
  WHERE  o.a = m.a;
```

```
                                        QUERY PLAN
Hash Join  (cost=368.00..22289.88 rows=500611 width=50) (actual time=5.355..337.678 rows=500611 loops=1)
  Output: o.a, o.b, o.c, m.a, m.b, m.c
  Inner Unique: true
  Hash Cond: (m.a = o.a)
  Buffers: shared hit=3755, temp written=46 ◀ significantly less I/O to/from batches, all processing skew batch
  ->  Seq Scan on public.many1 m  (cost=0.00..8687.11 rows=500611 width=25) (actual time=0.009..72.507 rows=500611 loops=1)
        Output: m.a, m.b, m.c
        Buffers: shared hit=3681
  ->  Hash  (cost=174.00..174.00 rows=10000 width=25) (actual time=5.330..5.330 rows=10000 loops=1)
        Output: o.a, o.b, o.c   ▼ four batches prepared, but no probe rows placed in batches (all join in skew batch)
        Buckets: 4096  Batches: 4  Memory Usage: 178kB
        Buffers: shared hit=74, temp written=43
        ->  Seq Scan on public.one o  (cost=0.00..174.00 rows=10000 width=25) (actual time=0.006..1.660 rows=10000 loops=1)
              Output: o.a, o.b, o.c
              Buffers: shared hit=74
Planning time: 0.299 ms
Execution time: 379.349 ms
```

```
set work_mem = default;
```

6 ¦ Q_{11}: Equi-Joins in MonetDB

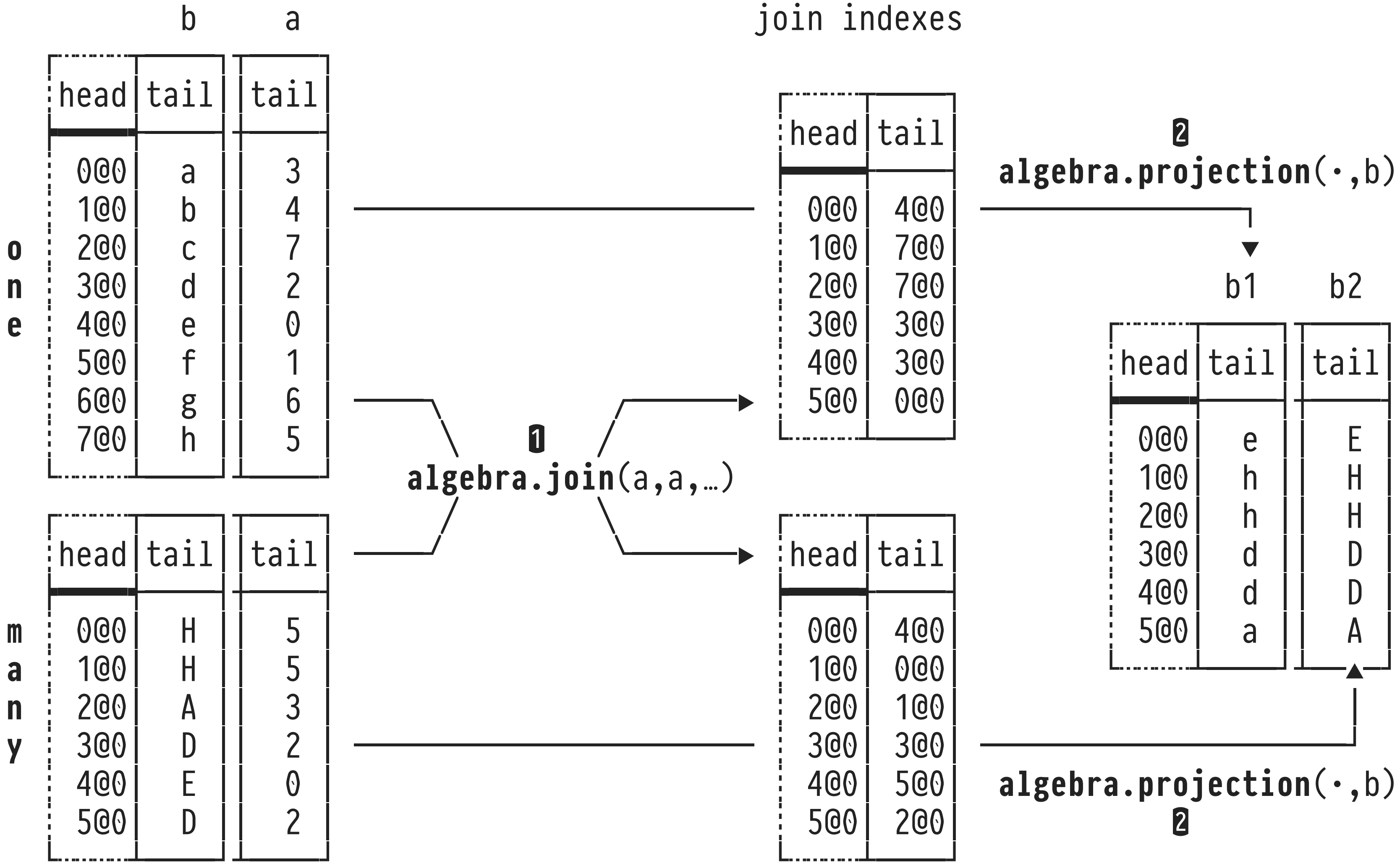
```
SELECT o.b AS b1, m.b AS b2
FROM   one AS o,
       many AS m
WHERE  o.a = m.a
```

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



one

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

many

head	tail	tail
0@0	H	5
1@0	H	5
2@0	A	3
3@0	D	2
4@0	E	0
5@0	D	2

1 **algebra.join**(a,a,…)

join indexes

head	tail
0@0	4@0
1@0	7@0
2@0	7@0
3@0	3@0
4@0	3@0
5@0	0@0

head	tail
0@0	4@0
1@0	0@0
2@0	1@0
3@0	3@0
4@0	5@0
5@0	2@0

2 **algebra.projection**(·,b)

	b1	b2
head	tail	tail
0@0	e	E
1@0	h	H
2@0	h	H
3@0	d	D
4@0	d	D
5@0	a	A

algebra.projection(·,b) 2

Demonstrate the evaluation of an equi-join. Prepare SQL input tables, then show prototypical MAL plan.

```
-- 1 Prepare tables one, many
  DROP TABLE IF EXISTS one;
  DROP TABLE IF EXISTS many;

  CREATE TABLE one  (a int PRIMARY KEY,
                     b text);
  CREATE TABLE many (a int NOT NULL,
                     b text);        -- don't declare FOREIGN KEY here (indexes/reorders column)

  INSERT INTO one(a,b) VALUES
    (3, 'a'),
    (4, 'b'),
    (7, 'c'),
    (2, 'd'),
    (0, 'e'),
    (1, 'f'),
    (6, 'g'),
    (5, 'h');

  INSERT INTO many(a,b) VALUES
    (5, 'H'),
    (5, 'H'),
    (3, 'A'),
    (2, 'D'),
    (0, 'E'),
    (2, 'D');

-- 2 Q11
  SELECT o.b AS b1, m.b AS b2
  FROM   one AS o,
         many AS m
  WHERE  o.a = m.a;
+------+------+
| b1   | b2   |
+======+======+
| a    | A    |
| d    | D    |
| d    | D    |
| e    | E    |
| h    | H    |
| h    | H    |
+------+------+

  EXPLAIN
```

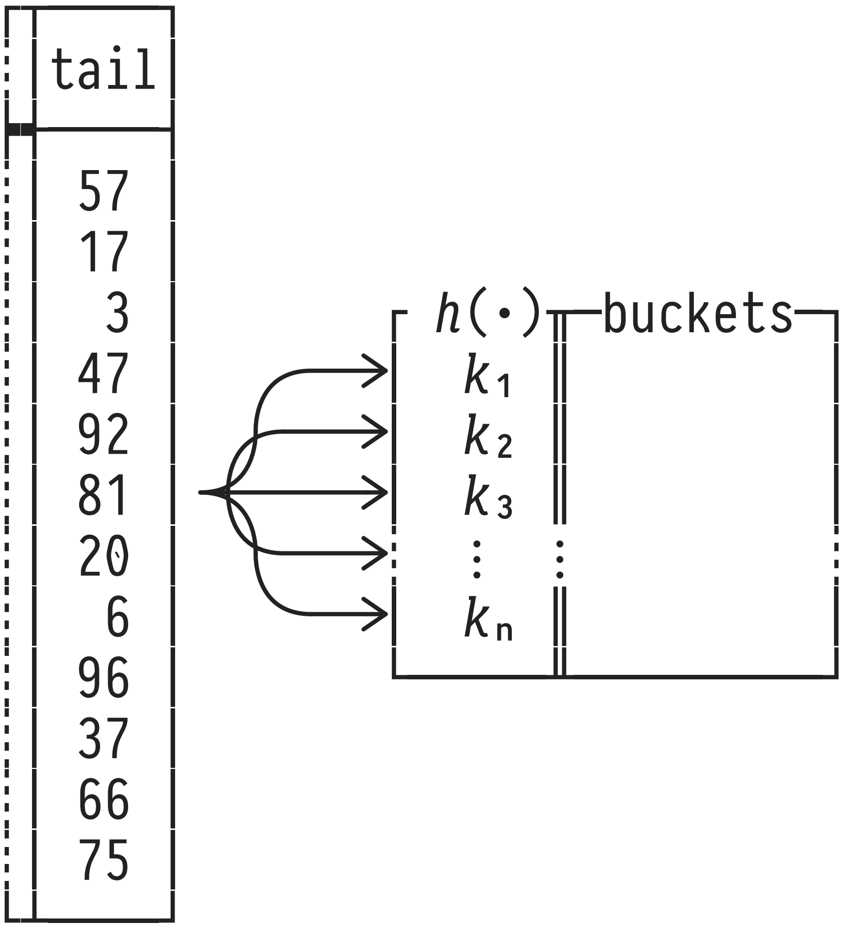
```
    SELECT o.b AS b1, m.b AS b2
    FROM   one AS o,
           many AS m
    WHERE  o.a = m.a;

[...]
X_4 := sql.mvc();
C_5:bat[:oid] := sql.tid(X_4, "sys", "one");
X_8:bat[:int] := sql.bind(X_4, "sys", "one", "a", 0:int);
X_17 := algebra.projection(C_5, X_8);
C_25:bat[:oid] := sql.tid(X_4, "sys", "many");
X_27:bat[:int] := sql.bind(X_4, "sys", "many", "a", 0:int);
X_32 := algebra.projection(C_25, X_27);
(X_39, X_40) := algebra.join(X_17, X_32, nil:BAT, nil:BAT, false, nil:lng);
X_33:bat[:str] := sql.bind(X_4, "sys", "many", "b", 0:int);
X_48:bat[:str] := algebra.projectionpath(X_40, C_25, X_33);
X_18:bat[:str] := sql.bind(X_4, "sys", "one", "b", 0:int);
X_46:bat[:str] := algebra.projectionpath(X_39, C_5, X_18);
[...]
```

Readable MAL plan:

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

one    :bat[:oid] := sql.tid(sql, "sys", "one");
one_a0 :bat[:int] := sql.bind(sql, "sys", "one", "a", 0:int);
one_a  :bat[:int] := algebra.projection(one, one_a0);

many   :bat[:oid] := sql.tid(sql, "sys", "many");
many_a0:bat[:int] := sql.bind(sql, "sys", "many", "a", 0:int);
many_a :bat[:int] := algebra.projection(many, many_a0);

# ❶ compute join index BATs for left/right input tables
#                             nil matches? (outer join semantics)      result size estimate
#                                                               ↓      ↓
(left,right) := algebra.join(one_a, many_a, nil:bat, nil:bat, false, nil:lng);
#                                            ↑        ↑
#                                          candidate BATs
io.print(left,right);

one_b0 :bat[:str] := sql.bind(sql, "sys", "one", "b", 0:int);
many_b0:bat[:str] := sql.bind(sql, "sys", "many", "b", 0:int);

# ❷ apply join index BATs to all required input columns
b1     :bat[:str] := algebra.projectionpath(left, one, one_b0);
b2     :bat[:str] := algebra.projectionpath(right, many, many_b0);
io.print(b1,b2);
```

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.



tail
57
17
3
47
92
81
20
6
96
37
66
75

[6] The CPU's *Translation Lookaside Buffer* stores recent translations from virtual into physical memory locations.

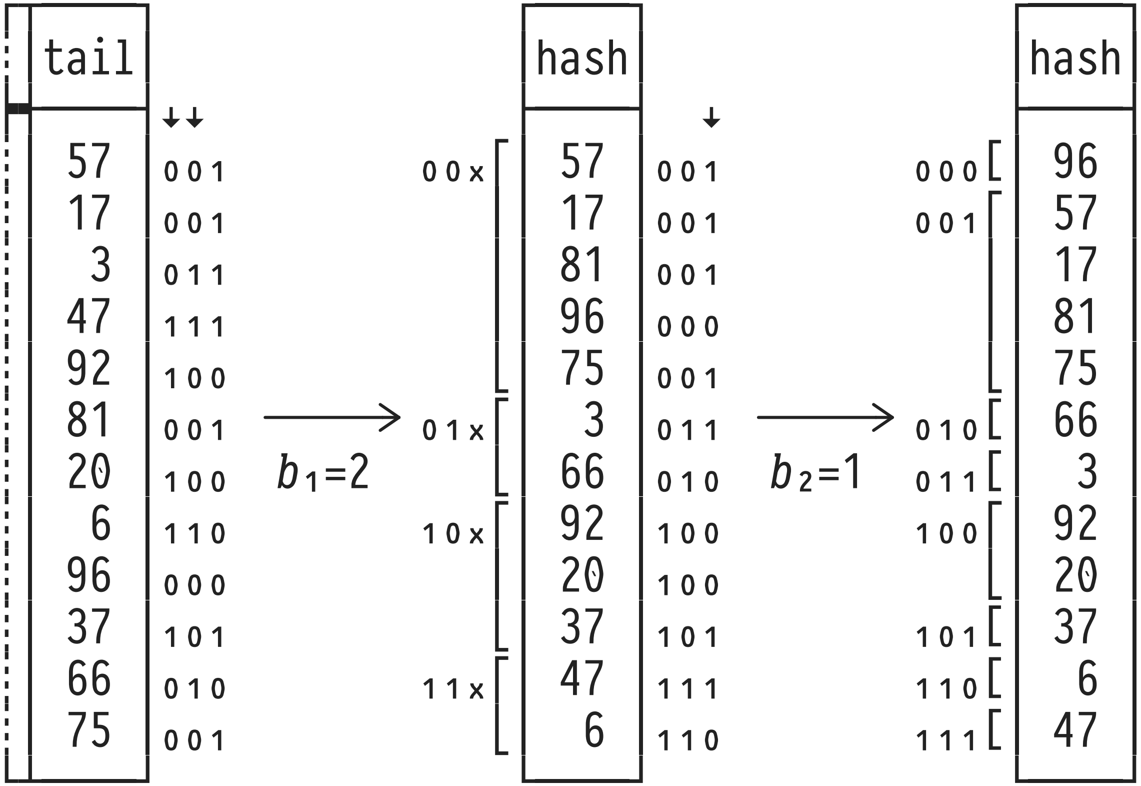
- **Cache line** size in Intel Core i7: 64 bytes, L2 cache: 256 kB ⇒ 4000 cache lines
- **TLB:**

  For many years now, processors have been working not with physical memory addresses, but with virtual addresses. Among other advantages, this approach lets more memory be allocated to a program than the computer actually has, keeping only the data necessary at a given moment in actual physical memory with the rest remaining on the hard disk. This means that for each memory access a virtual address has to be translated into a physical address, and to do that an enormous table is put in charge of keeping track of the correspondences. The problem is that this table gets so large that it can’t be stored on-chip–it’s placed in main memory, and can even be paged (part of the table can be absent from memory and itself kept on the hard disk).

  If this translation stage were necessary at each memory access, it would make access much too slow. As a result, engineers returned to the principle of physical addressing by adding a small cache memory directly on the processor that stored the correspondences for a few recently accessed addresses. This cache memory is called a Translation Lookaside Buffer (TLB).

  - Typical TLB:
    - 4096 entries (4kB pages or 2MB pages), separate/joint instruction/data TLBs
    - modern Intel Core i7 CPUs: two-level TLB
    - hit time: 1 clock cycle, miss penalty: 10+ clock cycles

Radix-Clustering



tail	↓↓		hash	↓		hash
57	001	00x	57	001	000	96
17	001		17	001	001	57
3	011		81	001		17
47	111		96	000		81
92	100		75	001		75
81	001	01x	3	011	010	66
20	100		66	010	011	3
6	110	10x	92	100	100	92
96	000		20	100		20
37	101		37	101	101	37
66	010	11x	47	111	110	6
75	001		6	110	111	47

- 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:

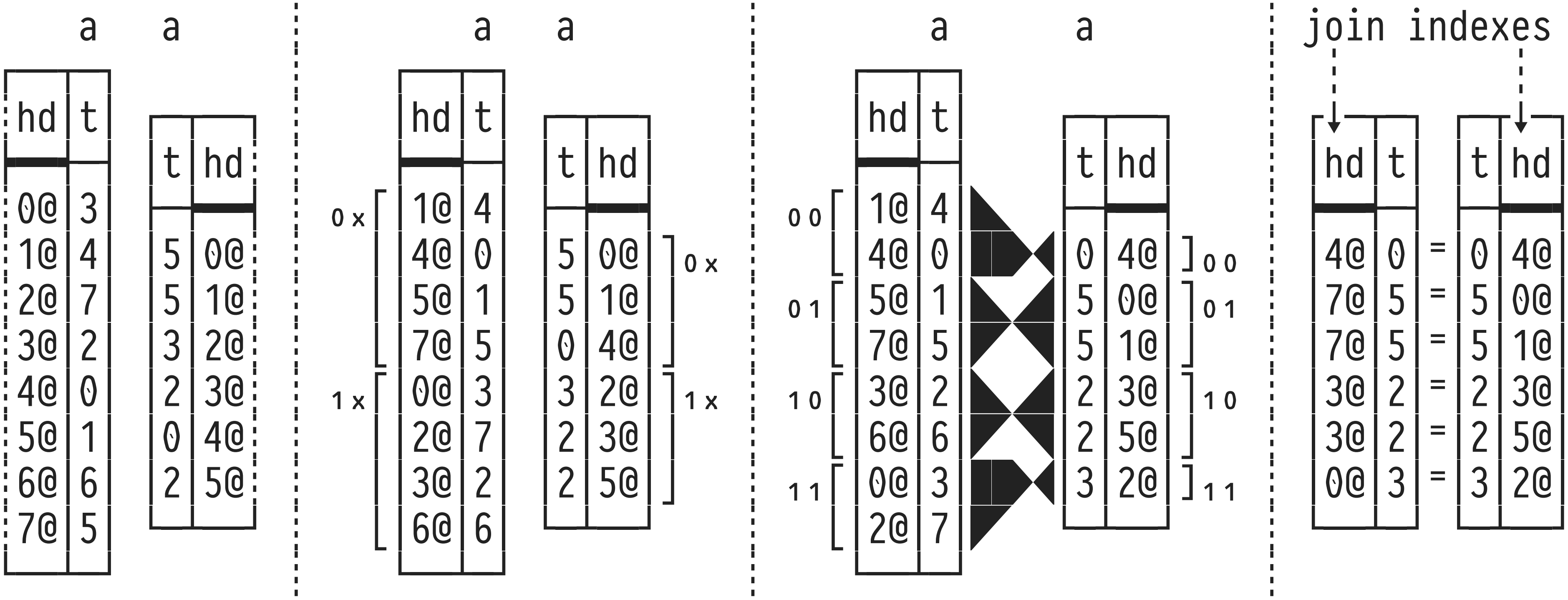
  $$\prod_{i=1}^{p} 2\hat{}b_i$$

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

N.B.: For simplicity, the above figure assumes that $h = \mathrm{id}$.

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.

- Why not use identity as hash function *h*? Why have a hash function at all?

  Radix clustering inspects few bits of join column values. Define a hash function that depends on *all* bits of the original value.