DB 2

13 – Plan Evaluation

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 ¦ Evaluating Query Plan Trees

The evaluation of a (complex) query plan requires a coordinated execution of the plan's operators:

- Is data **pushed** from the leaves (*e.g.*, Seq Scan, Index Scan) towards the plan root?
- Or does an operator **pull** the intermediate results from its upstream child operators?
- What kind of data flows across the plan's edges? **Entire tables or columns**? **Single rows**?
- Does the plan execute in one shot or can we **demand** the "next result row" when we are ready to consume it?
  - Can operators remember/resume from their current state?

Query Q_{12} and its (Moderately Complex) Plan

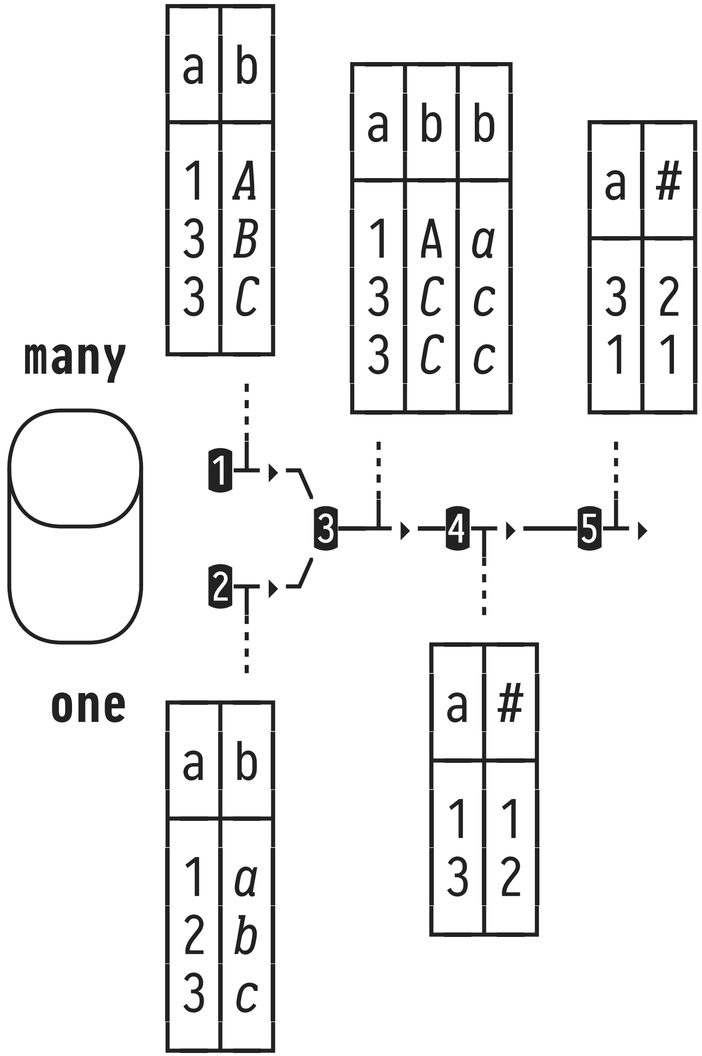
- Q_{12}:

```
SELECT o.a, COUNT(*) AS "#"
FROM   one AS o, many AS m
WHERE  o.a = m.a
GROUP BY o.a
ORDER BY o.a DESC
```

- Plan operators:

  ❶ Seq Scan on **many** (outer of ❸)
  ❷ Seq Scan on **one** (inner of ❸)
  ❸ Nested Loop (Join Filter: o.a = m.a)
  ❹ HashAggregate (Group Key: o.a)
  ❺ Sort (Sort Key: o.a DESC)

  ─▸─ ≡ direction of data flow
  ❶…❺ ≡ evaluation order



MonetDB: Full Materialization

MonetDB generates MAL programs that evaluate operators following a post-order traversal[1] of the query plan tree.

- Leaf nodes evaluated first, downstream nodes consume BATs generated by child nodes. Root operator evaluated last.
- Each operator consumes entire BATs, generates and **fully materializes** its result BAT(s) [cf. previous slide].
  - 👍 Tight code loops process entire columns. **Instruction *and* data locality**, predictable memory access.
  - 👎 **Size of intermediate results** may exceed available RAM ⇒ OS-level paging and thus disk I/O.

[1] Recall: data-flow dependency analysis enables the // evaluation of ❶ and ❷.

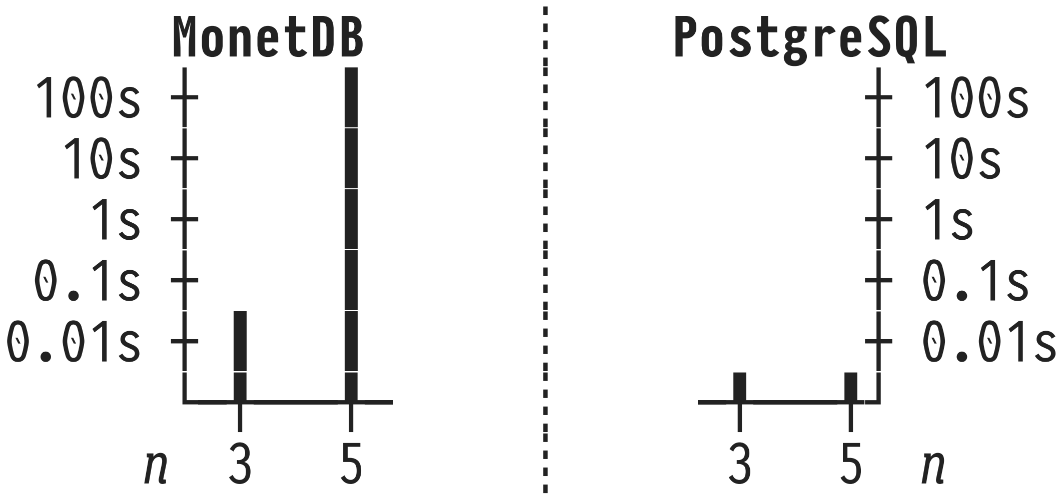
Data Dependencies in MAL Program for Q_{12}

```
one      :bat[:oid] := sql.tid(sql, "sys", "one");                    } 1 Scan one.a
one_a0 :bat[:int] := sql.bind(sql, "sys", "one", "a", 0:int);
one_a    :bat[:int] := algebra.projection(one, one_a0);
                                                                        1 // 2
many     :bat[:oid] := sql.tid(sql, "sys", "many");                   } 2 Scan many.a
many_a0:bat[:int] := sql.bind(sql, "sys", "many", "a", 0:int);
many_a  :bat[:int] := algebra.projection(many, many_a0);

(left, right) := algebra.join(one_a, many_a, nil:bat, nil:bat, false, nil:lng);   } 3 Equi-Join
joined_one_a:bat[:int] := algebra.projection(left, one_a);

(grouped_one_a, group_keys, group_sizes) := group.groupdone(joined_one_a);        } 4 Group + Agg
keys_a:bat[:int] := algebra.projection(group_keys, joined_one_a);
count:bat[:lng] := aggr.subcount(grouped_one_a, grouped_one_a, group_keys, false);

(sorted_a, oidx, gidx) := algebra.sort(keys_a, true, false);                      } 5 Sort
result_a     :bat[:int] := algebra.projection(oidx, keys_a);
result_count:bat[:lng] := algebra.projection(oidx, count);
```

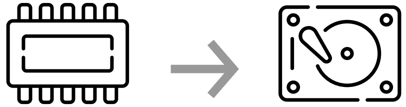
2 ┆ Materialization vs. Demand-Driven Pipelining

Consider Q_{13}, returning the single value 42:

```sql
SELECT 42 AS fortytwo
FROM   hundred AS h₁, …, hundred AS hₙ  -- ⚠ 100ⁿ rows
LIMIT  1
```



Volcano-Style Demand-Driven Pipelining



PostgreSQL implements the **Volcano Iterator Model:**

- Operator **demands** its subplan to produce the next row (*i.e.*, the plan root drives the query evaluation).
- Operator delivers results **one row at a time**, avoids intermediate result materialization (if possible ⚠️):
  - Reduces query *response* time (first row delivered immediately, do not wait until result is complete). 👍
  - Reduces memory requirements (pass data row-by-row, not table at a time). 👍

Demand-Driven Evaluation and Call by Need

Volcano-style **demand-driven** pipelining bears some resemblance with **call-by-need** evaluation of (functional) programming languages:

- If function $f(e_1,e_2)$ does not (always) need the value of expression e_2, then $f(42,1/0)$ may evaluate just fine.
- With the demand-driven evaluation in Haskell[2], consider:

```
sum [ x/0 | x <- [1..10], x > 42 ]   ⇢  0.0
length [ x/0 | x <- [1..10] ]        ⇢  10
```

[2] Haskell is a *lazily* evaluated functional programming language, see http://haskell.org.

Query Response vs. Evaluation Time

In PostgreSQL's EXPLAIN output, query **response** (first row) and **evaluation time** (all rows) are distinguished:

```
⋮
Seq Scan on many m (actual time=0.747..139.172 rows=502867 …)
```

$\underbrace{0.747}_{}..\underbrace{139.172}_{}$ **response**/**evaluation** time

- Both times may…
  - … differ substantially (pipelined evaluation),
  - … coincide (**blocking** operators—*e.g.*, Sort—evaluate in full first, then deliver all rows from intermediate result buffer).

Volcano Iterator Model: API

In Volcano-style demand-driven query evaluation, operators implement a simple API of three main methods:

1. **open()**: Initialize operator and its internal state, forward open() request to upstream subplans as well.
2. **next()**: If required, forward next() upstream to request more input rows. Then deliver next output row (or ␀ if result complete).
3. **close()**: Release operator-internal state, forward close() request to upstream subplans as well.

Volcano-style call protocol: (open() next()* close())+.

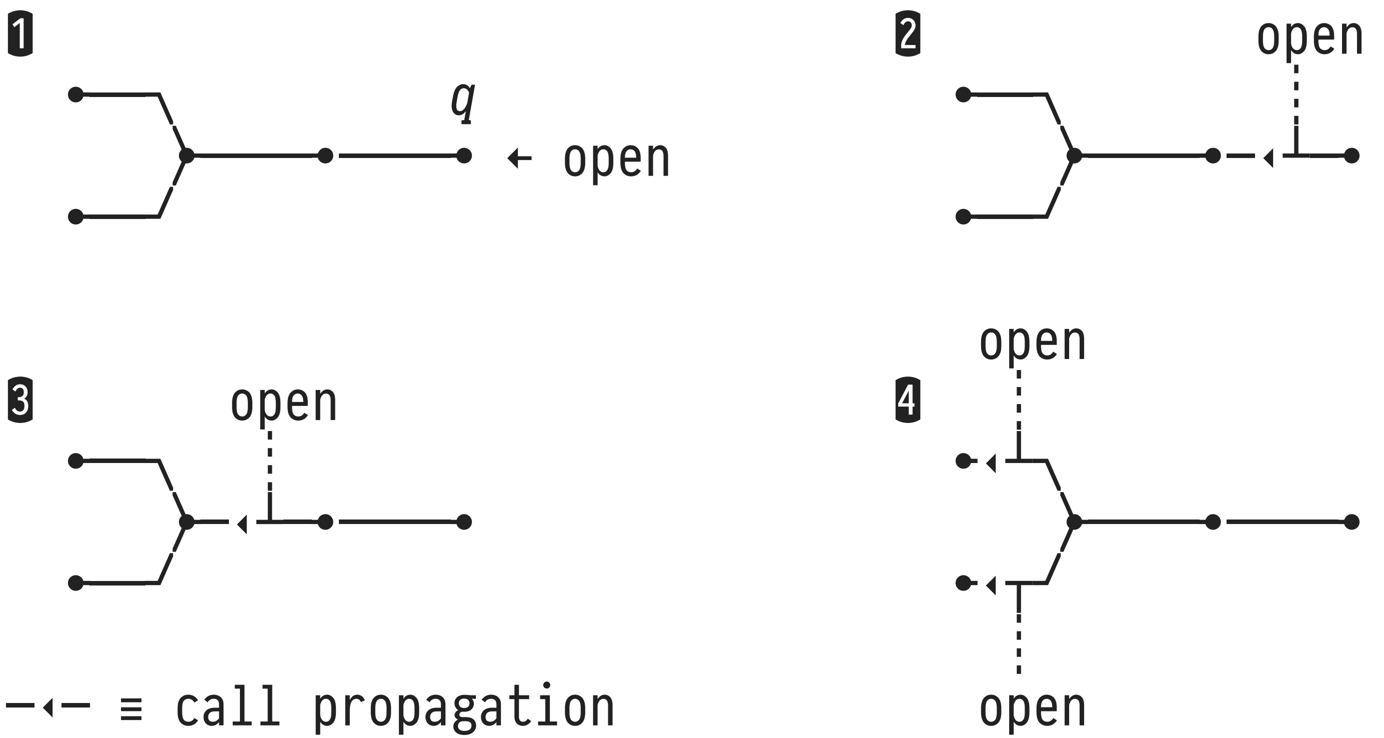
Volcano Iterator Model: Query Evaluation Driver

Use the Volcano iterator model API to *fully* evaluate a query. Operator q denotes the root of the query plan:

```
Eval(q):
  open(q);
  t ← next(q);
  while t ≠ NULL            } iterate while still rows to consume
    emit(t);                } ship current row to application
    t ← next(q);
  close(q);
```

- To retrieve next result row only, simply call next(q).
- May/must use close(q) to cancel query evaluation midway.

Volcano Iterator Model: Forwarding open()/close()



- Each operator instance (•) allocates and releases its own copy of state that is kept between method invocations.

Pipelined Nested Loop Join (NLJ) ❶

Implement open() and close() for the Nested Loop Join operator:

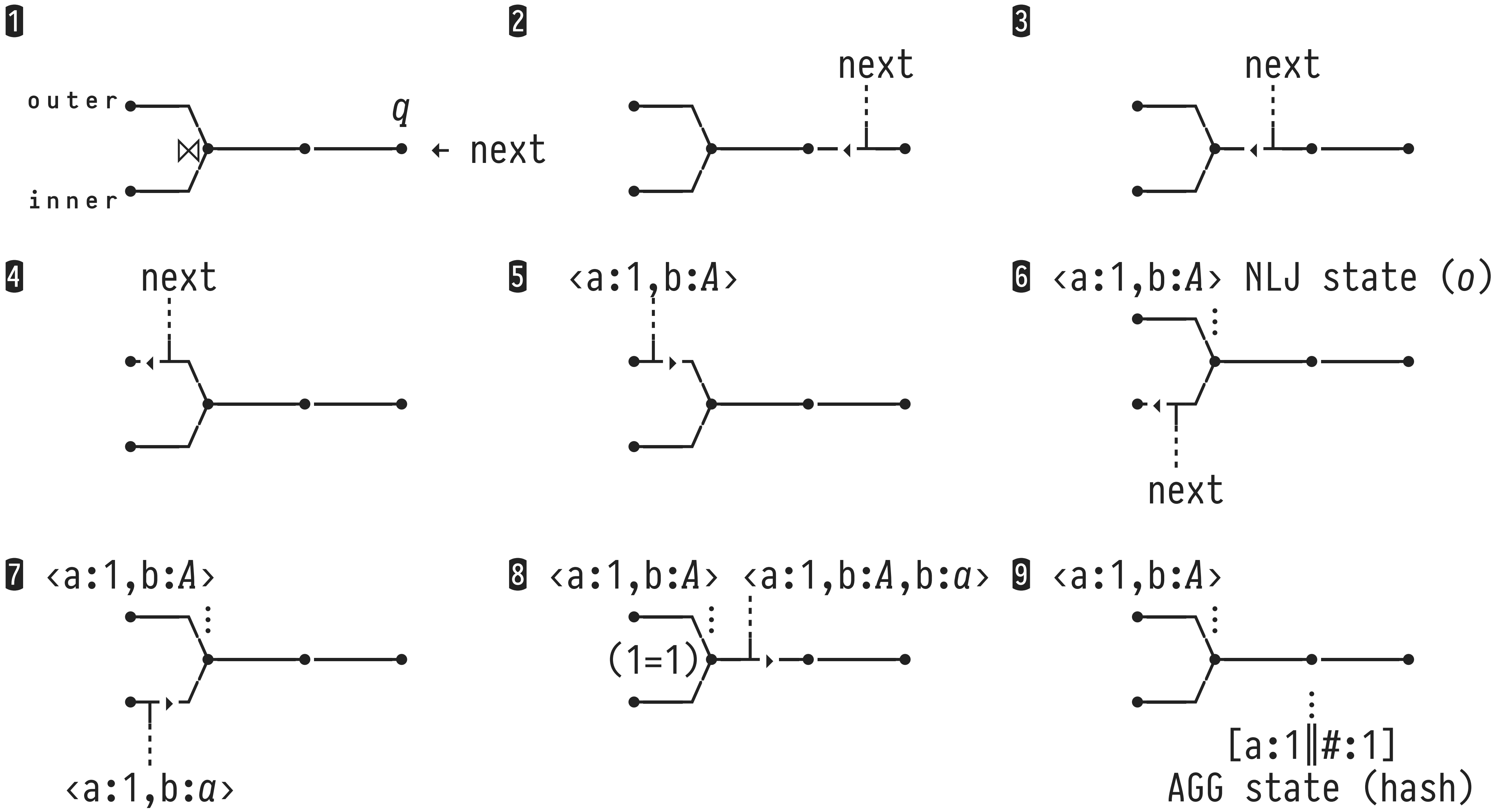
```
NLJ.open(outer,inner,θ):
  open(outer);                 ⎫ may open() in ∥:
  open(inner);                 ⎭                   outer inner

  needNewOuter ← true;         ⎫ local NLJ state
  o            ← NULL;         ⎭
```

```
NLJ.close(outer,inner,θ):
  close(outer);
  close(inner);
```

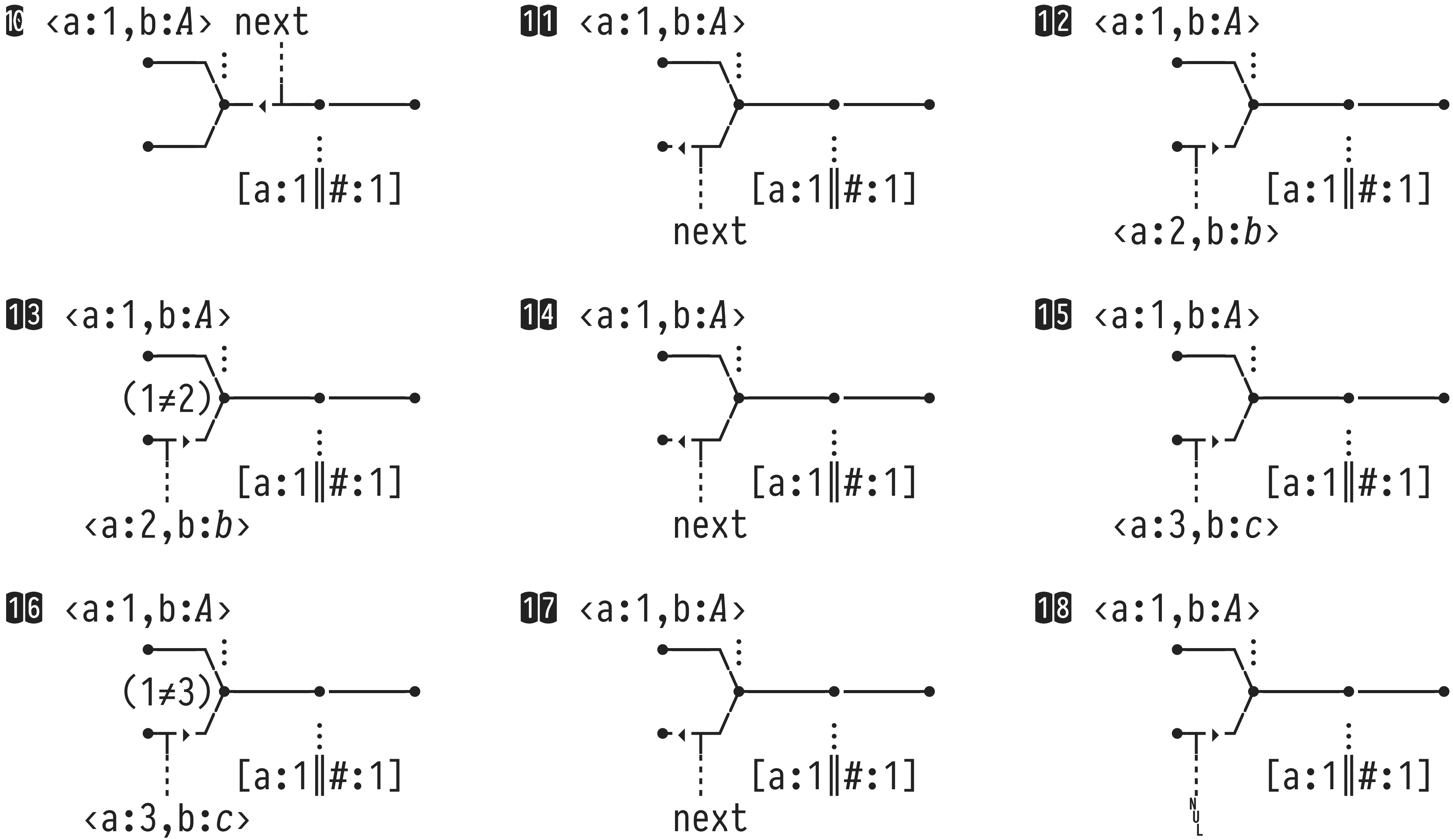
Pipelined Nested Loop Join (NLJ) ❷

```
NLJ.next(outer,inner,θ):
  forever
    if needNewOuter
      o ← next(outer);          } o: current outer row
      if o = NULL               ⎫ no more outer rows
        return NULL;            ⎭  ⇒ join complete
      needNewOuter ← false;
      close(inner);             ⎫ reset/rescan
      open(inner);              ⎭  inner input

    i ← next(inner);            } i: current inner row
    if i = NULL                 ⎫ no more inner rows,
      needNewOuter ← true;      ⎭  next time: read new outer
    else if o θ i               } join condition satisfied?
          return ‹o,i›;         } return single joined pair
```

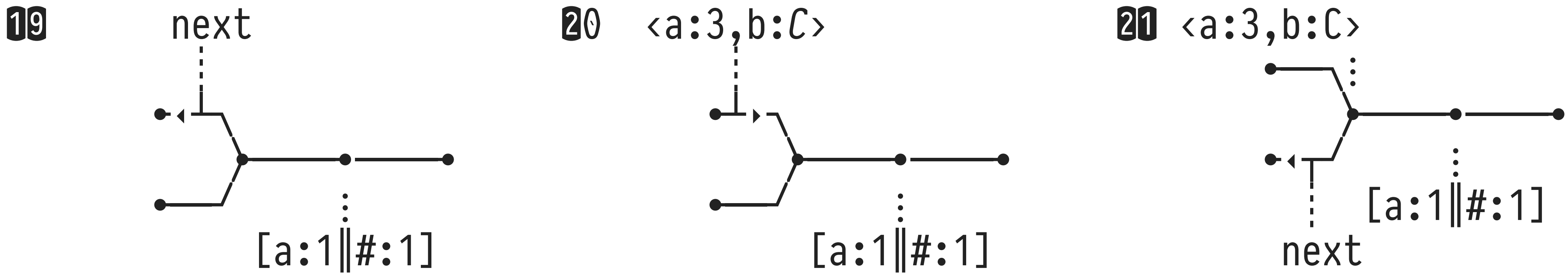
Volcano Iterator Model: Evaluating a NLJ Plan



Volcano Iterator Model: Evaluating a NLJ Plan



Volcano Iterator Model: Evaluating a NLJ Plan



Quiz/Exercise: Think about how to implement the following plan operators in the Volcano iterator model:

- Seq Scan (with Filter condition),
- Limit (given a row limit *n*),
- GroupAggregate (over input sorted by the Group Key),
- Append (SQL: UNION ALL).

Volcano Iterator Model at the SQL Level

Via **cursors**, the SQL standard exposes the Volcano-style open/next/close API at the level of (Embedded) SQL:

```
-- Generate query plan, no evaluation yet
❶ DECLARE ‹cursor› [ SCROLL ] CURSOR FOR ‹query›
--                     ⬑ cursor can move backwards

-- Evaluate plan to deliver the next/prior row (‹n› rows)
❷ FETCH [ NEXT | [ FORWARD | BACKWARD ] ‹n› ] FROM ‹cursor›

-- Release plan/intermediate buffers
❸ CLOSE ‹cursor›
```

- Statements need to be issued within an SQL transaction.

Volcano-Style Iteration has its Cost

- Effectively, **multiple operators are active at one time.**
  - Aggregate intermediate state (memory) may be large.
  - Method call forwarding incurs function call overhead.
  - Frequent switches between code blocks due to row-by-row processing, CPU instruction cache misses are likely.

- 💡 Few modern RDBMSs (*X100* aka *VectorWise*[3]) seek middle ground between full materialization and pipelining:
  - Build demand-driven pipeline between operators, but…
  - … pass **vectors of rows**—typically the size of the CPU's data cache—between operators.

[3] See *MonetDB/X100—A DBMS In The CPU Cache* and *MonetDB/X100: Hyper-Pipelining Query Execution*.