

DB 2

13 – Plan Evaluation

Summer 2018

Torsten Grust
Universität Tübingen, Germany

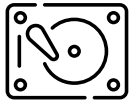
1 | Evaluating Query Plan Trees



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} :

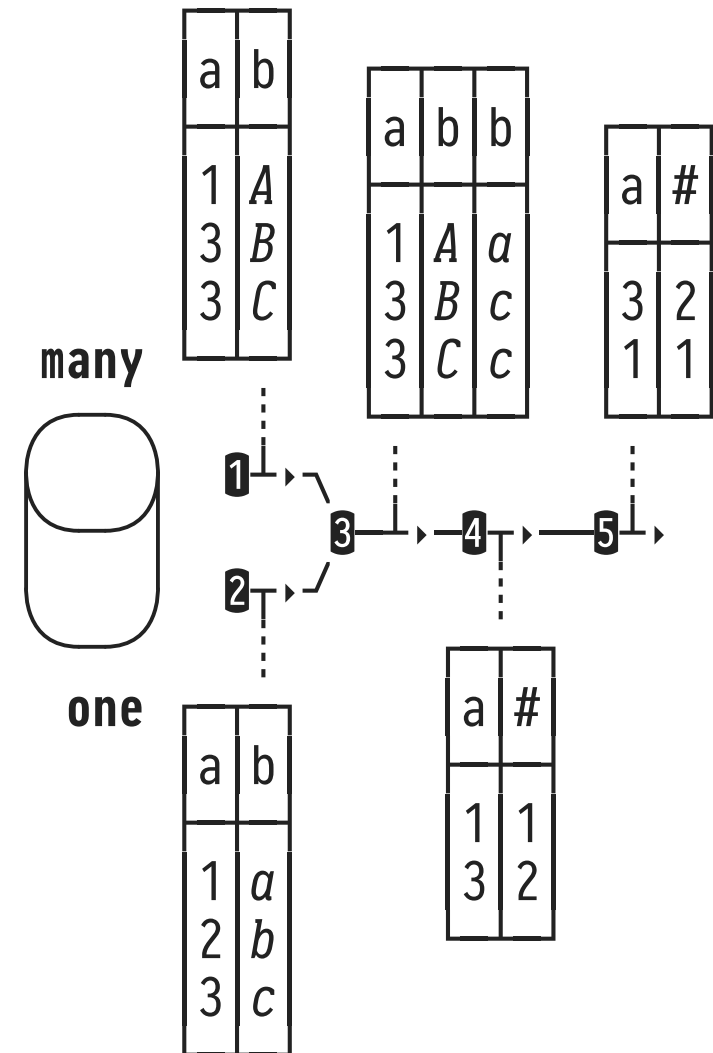
```
SELECT o.a, COUNT(*) AS "#"  
FROM   one AS o, many AS m  
WHERE  o.a = m.a  
GROUP BY o.a  
ORDER BY o.a DESC
```

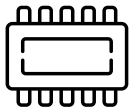
- Plan operators:

- 1 Seq Scan on **many** (outer of 3)
- 2 Seq Scan on **one** (inner of 3)
- 3 Nested Loop (Join Filter: $o.a = m.a$)
- 4 HashAggregate (Group Key: $o.a$)
- 5 Sort (Sort Key: $o.a$ DESC)

→ = direction of data flow

1...5 = evaluation order





MonetDB generates MAL programs that evaluate operators following a post-order traversal¹ 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 \Rightarrow OS-level paging and thus disk I/O.

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

MonetDB: Demonstrate that order of MAL operation follows a leaf-to-root (bottom-up, post-order) traversal of the plan tree. Display intermediate result BATs along the way.

-- ❶ Check input tables

```
\d one
CREATE TABLE "sys"."one" (
  "a" INTEGER NOT NULL,
  "b" CHARACTER LARGE OBJECT,
  CONSTRAINT "one_a_pkey" PRIMARY KEY ("a")
);
```

```
\d many
CREATE TABLE "sys"."many" (
  "a" INTEGER NOT NULL,
  "b" CHARACTER LARGE OBJECT
);
```

SELECT * FROM one;

+-----+	
a	b
+=====+	
3	a
4	b
7	c
2	d
0	e
1	f
6	g
5	h
+-----+	

SELECT * FROM many;

+-----+	
a	b
+=====+	
5	H
5	H
3	A
2	D
0	E
2	D
+-----+	

-- ❷ Query plan for Q₁₂

```

PLAN
  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;

```

```
| rel
```

```

project ( ← projection & sorting
  group by (
    join (
      table(sys.one) [ "one"."a" NOT NULL HASHCOL as "o"."a" ] COUNT ,
      table(sys.many) [ "many"."a" NOT NULL as "m"."a" ] COUNT
    ) [ "o"."a" NOT NULL HASHCOL = "m"."a" NOT NULL ]
  ) [ "o"."a" NOT NULL HASHCOL ] [ "o"."a" NOT NULL HASHCOL , sys.count() NOT NULL as "L3"."L3" ]
) [ "o"."a" NOT NULL, "L3" NOT NULL as "L4"."#" ] [ "o"."a" NOT NULL ]

```

sort criterion (DESC is implicit [ASC would be displayed])

-- ❸ MAL code for Q₁₂ (follow post-order traversal ov above plan tree, column BATs one.b/many.b never accessed)

```

EXPLAIN
  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;

```

```
[...]
```

```
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);

```

```
} ❶ Scan one.a
```

```

C_18:bat[:oid] := sql.tid(X_4, "sys", "many");
X_20:bat[:int] := sql.bind(X_4, "sys", "many", "a", 0:int);
X_25 := algebra.projection(C_18, X_20);

```

```
} ❷ Scan many.a
```

```

(X_26, X_27) := algebra.join(X_17, X_25, nil:BAT, nil:BAT, false, nil:lng);
X_32 := algebra.projection(X_26, X_17);

```

```
} ❸ (Hash) Equi-Join
```

```

(X_34, C_35, X_36) := group.groupdone(X_32);
X_37 := algebra.projection(C_35, X_32);
X_38:bat[:lng] := aggr.subcount(X_34, X_34, C_35, false);

```

```
} ❹ Group + Agg
```

```

(X_39, X_40, X_41) := algebra.sort(X_37, true, false);
X_44 := algebra.projection(X_40, X_38);
X_43 := algebra.projection(X_40, X_37);
[...]
```

-- 4 Readable and executable MAL code:

```

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

# 1 Scan one.a
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);
io.print(one_a);

# 2 Scan many.a
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);
io.print(many_a);

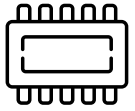
# 3 (Hash) Equi-Join      no candidate lists ↘      ↘ no outer ↘      ↘ no result size estimate
(left, right) := algebra.join(one_a, many_a, nil:bat, nil:bat, false, nil:lng);
joined_one_a :bat[:int] := algebra.projection(left, one_a);
io.print(joined_one_a);

# 4 Group + Agg
(grouped_one_a, group_keys, group_sizes) := group.groupdone(joined_one_a);
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);
#           values to aggregate ↗           ↗ group IDs           ↗ skip nils? [no: COUNT(*)]
io.print(keys_a, count);

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

```

Data Dependencies in MAL Program for Q_{12}



```

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);
} 1 Scan one.a

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);
} 2 Scan many.a

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

(grouped_one_a, group_keys, group_sizes) := group.groupdone(joined_one_a);
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);
} 4 Group + Agg

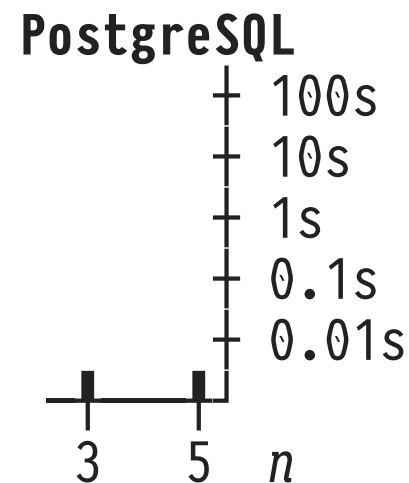
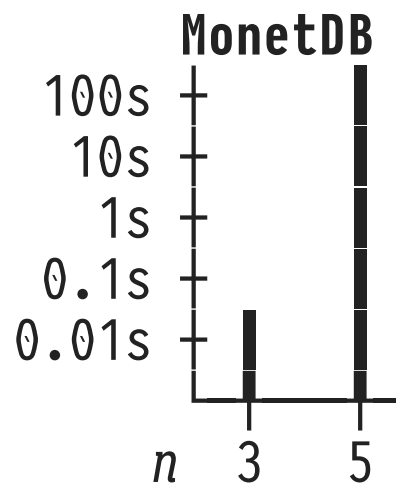
(sorted_a, oidx, gidx) := algebra.sort(keys_a, true, false);
result_a      :bat[:int] := algebra.projection(oidx, keys_a);
result_count:bat[:lng] := algebra.projection(oidx, count);
} 5 Sort

```


2 | Materialization vs. Demand-Driven Pipelining

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

```
SELECT 42 AS fortytwo
FROM   hundred AS h1, ..., hundred AS hn  -- Δ 100n rows
LIMIT 1
```



Demonstrate the materialized/demand-driven evaluation of query evaluation plans.

- Shell: use

```
$ top -r -stats pid,command,cpu,vsize,pageins -pid $(pgrep -f mserver5)
```

to observe memory allocation and paging behavior of MonetDB's server process.

-- 1 MonetDB: prepare input table

```
DROP TABLE IF EXISTS hundred;
```

```
CREATE TABLE hundred (i int);
```

```
INSERT INTO hundred(i)
```

```
  SELECT value
```

```
  FROM   generate_series(1, 101); -- 100 rows
```

```
\d hundred
```

-- Evaluate large cross-products (A materialization)

```
SELECT 42 AS fortytwo
```

```
FROM    hundred AS h1, hundred AS h2, hundred AS h3
```

```
LIMIT 1;
```

```
+-----+  
| fortytwo |
```

```
+-----+  
|      42 |
```

```
+-----+
```

```
1 tuple (17.635ms)
```

```
SELECT 42 AS fortytwo
```

```
FROM    hundred AS h1, hundred AS h2, hundred AS h3, hundred AS h4
```

```
LIMIT 1;
```

```
+-----+  
| fortytwo |
```

```
+-----+  
|      42 |
```

```
+-----+
```

```
1 tuple (1.8s)  $\approx 100 \times 17.635$  ms ✓
```

```
EXPLAIN
```

```
  SELECT 42 AS fortytwo
```

```
  FROM    hundred AS h1, hundred AS h2, hundred AS h3, hundred AS h4
```

```

LIMIT 1;
[...]
```

```

X_5 := sql.mvc();
C_8:bat[:oid] := sql.tid(X_5, "sys", "hundred");
X_11:bat[:int] := sql.bind(X_5, "sys", "hundred", "i", 0:int);
X_20 := algebra.projection(C_8, X_11);
(X_28, X_29) := algebra.crossproduct(X_20, X_20);    ▲ materialized cross product
X_30 := algebra.projection(X_28, X_20);
(X_39, X_40) := algebra.crossproduct(X_30, X_20);    ▲ materialized cross product
X_41 := algebra.projection(X_39, X_30);
(X_52, X_53) := algebra.crossproduct(X_41, X_20);    ▲ materialized cross product
X_54 := algebra.projection(X_52, X_41);
X_59 := algebra.project(X_54, 42:bte);
C_67 := algebra.subslice(X_59, 0:lng, 0:lng);
X_68 := algebra.projection(C_67, X_59);
[...]
```

```

-- ⚠ Does NOT terminate in reasonable time, need to kill mclient and mserver5
-- (huge virtual memory size [VSIZE], all PhysMem used, heavy swapping, < 30% CPU utilization)
SELECT 42 AS fortytwo
FROM   hundred AS h1, hundred AS h2, hundred AS h3, hundred AS h4, hundred AS h5
LIMIT 1;
```

```

-- 2 PostgreSQL: prepare input table
```

```

DROP TABLE IF EXISTS hundred;

CREATE TABLE hundred (i int);
INSERT INTO hundred(i)
  SELECT i
  FROM   generate_series(1, 100) AS i;

\d hundred
```

```

-- Evaluate large cross-products (demand-driven pipelining)
```

```

SELECT 42 AS fortytwo
FROM   hundred AS h1, hundred AS h2, hundred AS h3
LIMIT 1;
```

```

| fortytwo |
```

42

Time: 0.925 ms

```
SELECT 42 AS fortytwo
FROM   hundred AS h1, hundred AS h2, hundred AS h3, hundred AS h4
LIMIT 1;
```

fortytwo
42

Time: 0.543 ms

```
SELECT 42 AS fortytwo
FROM   hundred AS h1, hundred AS h2, hundred AS h3, hundred AS h4, hundred AS h5
LIMIT 1;
```

fortytwo
42

Time: 0.667 ms

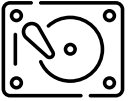
```
EXPLAIN (ANALYZE, BUFFERS, COSTS FALSE)
SELECT 42 AS fortytwo
FROM   hundred AS h1, hundred AS h2, hundred AS h3, hundred AS h4, hundred AS h5
LIMIT 1;
```

QUERY PLAN
Limit (actual time=0.099..0.099 rows=1 loops=1) Buffers: shared hit=5 ← essentially only constant buffer space needed -> Nested Loop (actual time=0.097..0.097 rows=1 loops=1) Buffers: shared hit=5 -> Nested Loop (actual time=0.083..0.083 rows=1 loops=1) Buffers: shared hit=4 -> Nested Loop (actual time=0.065..0.065 rows=1 loops=1) Buffers: shared hit=3 -> Nested Loop (actual time=0.046..0.046 rows=1 loops=1) Buffers: shared hit=2 -> Seq Scan on hundred h1 (actual time=0.022..0.022 rows=1 loops=1) Buffers: shared hit=1 -> Materialize (actual time=0.021..0.021 rows=1 loops=1) Buffers: shared hit=1



```
        -> Seq Scan on hundred h2 (actual time=0.010..0.010 rows=1 loops=1)
            Buffers: shared hit=1
    -> Materialize (actual time=0.018..0.018 rows=1 loops=1)
        Buffers: shared hit=1
        -> Seq Scan on hundred h3 (actual time=0.010..0.010 rows=1 loops=1)
            Buffers: shared hit=1
    -> Materialize (actual time=0.018..0.018 rows=1 loops=1)
        Buffers: shared hit=1
        -> Seq Scan on hundred h4 (actual time=0.010..0.010 rows=1 loops=1)
            Buffers: shared hit=1
    -> Materialize (actual time=0.012..0.012 rows=1 loops=1)
        Buffers: shared hit=1
        -> Seq Scan on hundred h5 (actual time=0.009..0.009 rows=1 loops=1)
            Buffers: shared hit=1
```

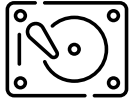
Planning time: 0.472 ms
Execution time: 0.191 ms

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 memory requirements (pass data row-by-row, not table at a time). 



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 Haskell², consider:

```
sum [ x/0 | x <- [1..10], x > 42 ]      ↦ 0.0
length [ x/0 | x <- [1..10] ]          ↦ 10
take 1 [ (x,y) | x <- [1..], y <- [1..] ] ↦ [(1,1)]
```

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

Demonstrate demand-driven evaluation and the NON-evaluation of parts of a query plan.

```
-- ❶ Check input tables
```

```
\d one
```

Table "public.one"

Column	Type	Collation	Nullable	Default
a	<u>integer</u>		not null	
b	<u>text</u>			
c	<u>integer</u>			

Indexes:

"one_a" 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	<u>integer</u>		not null	
b	<u>text</u>			
c	<u>integer</u>		not null	

Indexes:

"many_a_c" PRIMARY KEY, btree (a, c)

Foreign-key constraints:

"many_a_fkey" FOREIGN KEY (a) REFERENCES one(a)

```
-- ❷ Join query in which one leg yields the empty table ⇒ other leg
```

```
-- not evaluated at all
```

```
set enable_hashjoin = off;
```

```
EXPLAIN (VERBOSE, ANALYZE)
```

```
SELECT o.a / 0
```

```
FROM one AS o, many AS m
```

```
WHERE o.c = m.c
```

```
AND m.b = 'Ben Kenobi'; -- ← never satisfied ⇒ input leg 'many' yields no rows
```

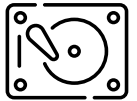

QUERY PLAN

```

Merge Join (cost=10869.80..10972.01 rows=5216 width=4) (actual time=107.661..107.661 rows=0 loops=1)
  Output: (o.a / 0)
  Merge Cond: (m.c = o.c)
    -> Sort (cost=10031.41..10031.55 rows=56 width=4) (actual time=107.660..107.660 rows=0 loops=1)
      Output: m.c
      Sort Key: m.c
      Sort Method: quicksort Memory: 25kB
      -> Seq Scan on public.many m (cost=0.00..10029.79 rows=56 width=4) (actual time=107.652..107.652 rows=0 loops=1)
        Output: m.c
        Filter: (m.b = 'Ben Kenobi'::text)
        Rows Removed by Filter: 505183
    -> Sort (cost=838.39..863.39 rows=10000 width=8) (never executed)
      Output: o.a, o.c
      Sort Key: o.c
      -> Seq Scan on public.one o (cost=0.00..174.00 rows=10000 width=8) (never executed)
        Output: o.a, o.c
Planning time: 0.175 ms
Execution time: 107.702 ms

```

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 ...)
                                   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).

Demonstrate how response and evaluation may differ/coincide for different (non-)blocking operator kinds.

```
-- 1 Check input table
```

\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)
```

```
-- 2 Seq Scan w/ Filter (fully pipelined)
```

EXPLAIN (ANALYZE, COSTS false)

```
SELECT m.b
FROM   many AS m
WHERE  m.a > 42;
```

QUERY PLAN
Seq Scan on many m (actual time=0.787..144.830 rows=502867 loops=1) Filter: (a > 42) Rows Removed by Filter: 2316 Planning time: 0.135 ms Execution time: 174.194 ms

```
-- 2 Sort Filter (blocking)
```

EXPLAIN (ANALYZE, COSTS false)

```
SELECT m.b
FROM   many AS m
WHERE  m.a > 42
ORDER BY m.b;
```

QUERY PLAN

```

Sort (actual time=2609.184..3473.799 rows=502867 loops=1)
  Sort Key: b
  Sort Method: external merge  Disk: 13312kB
  -> Seq Scan on many m (actual time=0.846..141.182 rows=502867 loops=1)
        Filter: (a > 42)
        Rows Removed by Filter: 2316
Planning time: 0.178 ms
Execution time: 3504.668 ms

```

```
-- 3 Aggregate (blocking, result tiny)
```

```

EXPLAIN (ANALYZE, COSTS false)
SELECT COUNT(m.b)
FROM   many AS m
WHERE  m.a > 42;

```

QUERY PLAN

```

Aggregate (actual time=228.496..228.496 rows=1 loops=1)
  -> Seq Scan on many m (actual time=0.498..125.060 rows=502867 loops=1)
        Filter: (a > 42)
        Rows Removed by Filter: 2316
Planning time: 0.147 ms
Execution time: 228.534 ms

```

```

-- 3 Grouped Aggregate over sorted input (⚠ first group(s) delivered BEFORE blocking Sort is done)
-- Q: How is this possible?
-- A: Grouping/aggregation folded into Sort's final merge phase)
--
-- See PostgreSQL source (src/backend/utils/sort/tuplesort.c):
-- ``When the caller requests random access to the sort result, we form
-- the final sorted run on a logical tape which is then "frozen", so
-- that we can access it randomly. When the caller does not need random
-- access, we return from tuplesort_performsort() as soon as we are down
-- to one run per logical tape. The final merge is then performed
-- on-the-fly as the caller repeatedly calls tuplesort_getXXX;      ← this is our next()
-- this saves one cycle of writing all the data out to disk and
-- reading it in.''

```

```
set enable_hashagg = off;
```

```

EXPLAIN (ANALYZE, COSTS false)
SELECT m.c, COUNT(m.b)

```

```
FROM many AS m
WHERE m.a > 42
GROUP BY m.c;
```

QUERY PLAN

GroupAggregate (actual time=406.329..612.823 rows=101 loops=1)

Group Key: c

! 406.329 < 505.390

-> Sort (actual time=402.426..505.390 rows=502867 loops=1)

Sort Key: c

Sort Method: external merge Disk: 15288kB

-> Seq Scan on many m (actual time=0.754..145.046 rows=502867 loops=1)

Filter: (a > 42)

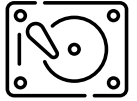
Rows Removed by Filter: 2316

Planning time: 0.233 ms

Execution time: 617.946 ms

```
set enable_hashagg = on;
```

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 **NULL** 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

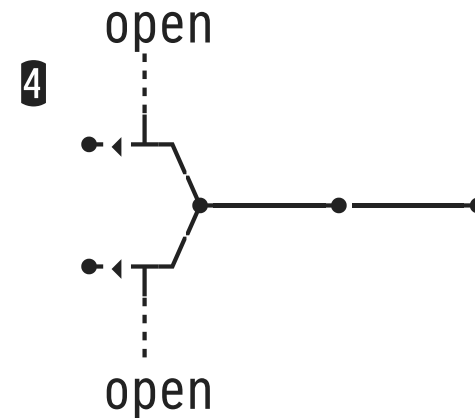
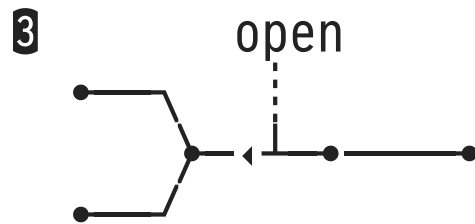
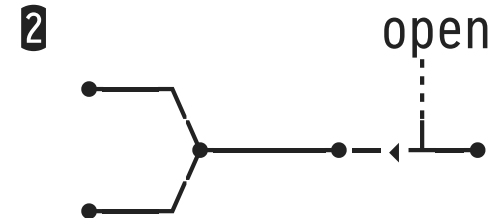
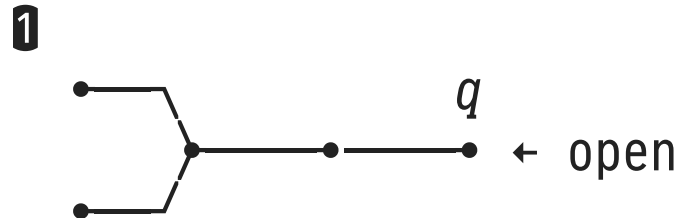


Use the Volcano iterator model API to *fully* evaluate a query. Operator q denotes the root of the query plan:

```
Eval( $q$ ):  
  open( $q$ );  
   $t \leftarrow \text{next}(\mathbf{q})$ ;  
  while  $t \neq \text{NUL}$  } \text{iterate while still rows to consume}  
  |   emit( $t$ ); } \text{ship current row to application}  
  |    $t \leftarrow \text{next}(\mathbf{q})$ ;  
  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()`



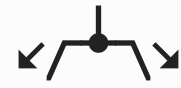
\dashleftarrow \equiv call propagation

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

Pipelined Nested Loop Join (NLJ) ①

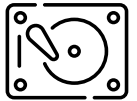


Implement `open()` and `close()` for the Nested Loop Join operator:

```
NLJ.open(outer,inner, $\theta$ ):  
    open(outer);           { may open() in //:   
    open(inner);           {  
                                outer inner  
  
    needNewOuter  $\leftarrow$  true;    { local NLJ state  
    o  $\leftarrow$   $\overset{N}{U}{\underset{L}{L}}$ ;    {
```

```
NLJ.close(outer,inner, $\theta$ ):  
    close(outer);  
    close(inner);
```

Pipelined Nested Loop Join (NLJ) ②

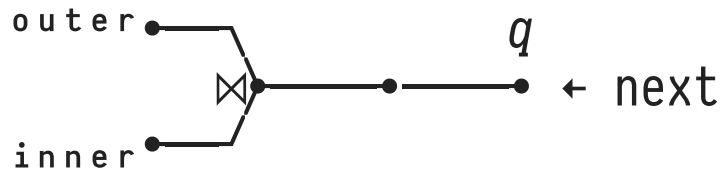


```
NLJ.next(outer,inner, $\theta$ ):  
  forever  
    if needNewOuter  
       $o \leftarrow \text{next}(\textit{outer});$            } o: current outer row  
      if  $o = \text{NULL}$                         } no more outer rows  
        return  $\text{NULL}$ ;                     }  $\Rightarrow$  join complete  
      needNewOuter  $\leftarrow$  false;  
      close(inner);                        } reset/rescan  
      open(inner);                          } inner input  
  
       $i \leftarrow \text{next}(\textit{inner});$            } i: current inner row  
      if  $i = \text{NULL}$                         } no more inner rows,  
        needNewOuter  $\leftarrow$  true;        } next time: read new outer  
      else if  $o \theta i$                      } join condition satisfied?  
        return  $\langle o, i \rangle$ ;                 } return single joined pair
```

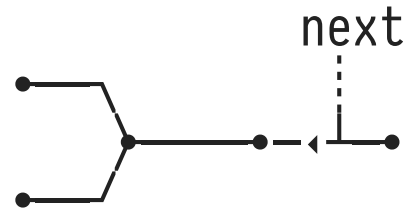
Volcano Iterator Model: Evaluating a NLJ Plan



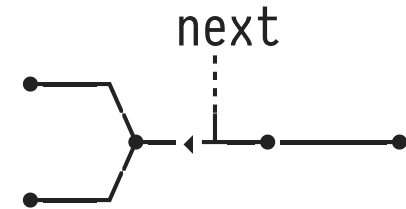
1



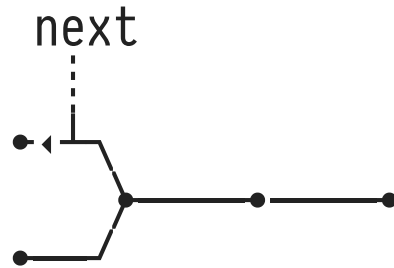
2



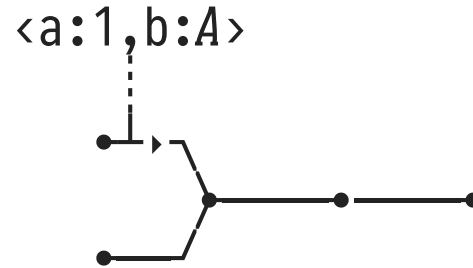
3



4

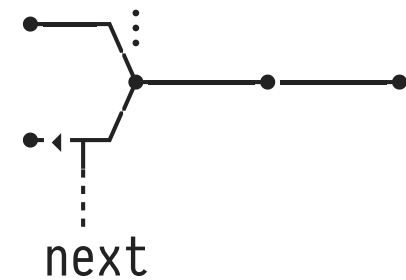


5



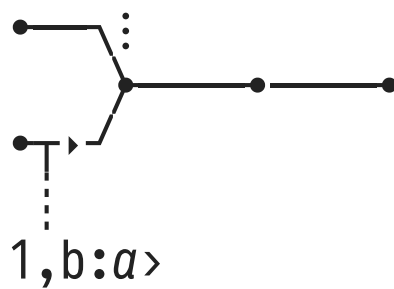
6

$\langle a:1, b:A \rangle$ NLJ state (o)



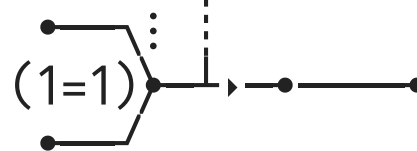
7

$\langle a:1, b:A \rangle$



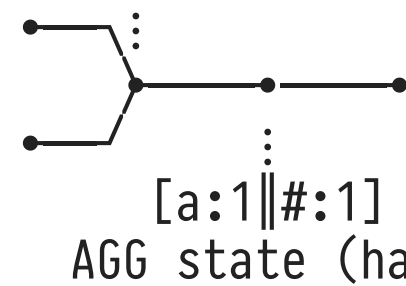
8

$\langle a:1, b:A \rangle$ $\langle a:1, b:A, b:a \rangle$



9

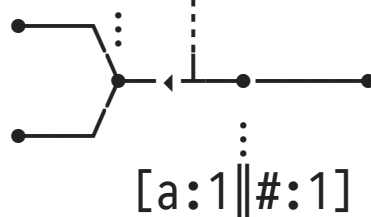
$\langle a:1, b:A \rangle$



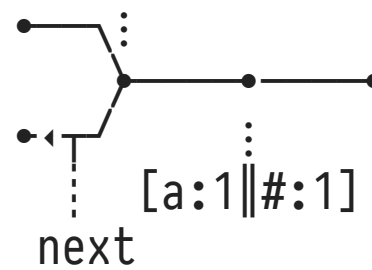
Volcano Iterator Model: Evaluating a NLJ Plan



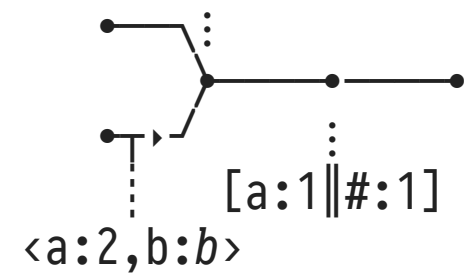
10 <a:1,b:A> next



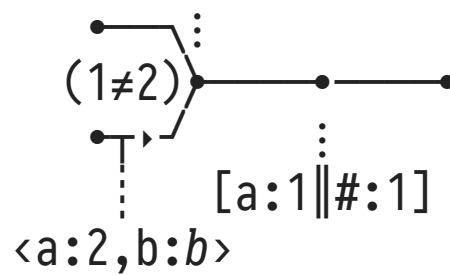
11 <a:1,b:A>



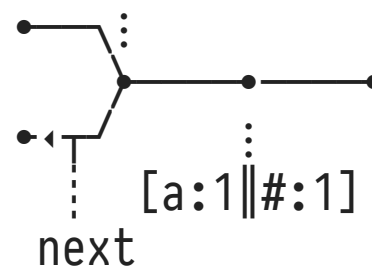
12 <a:1,b:A>



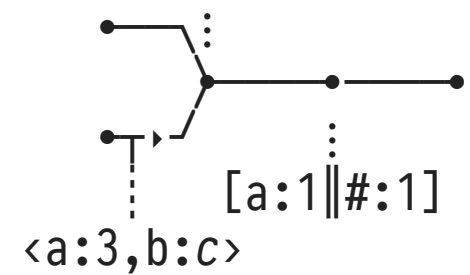
13 <a:1,b:A>



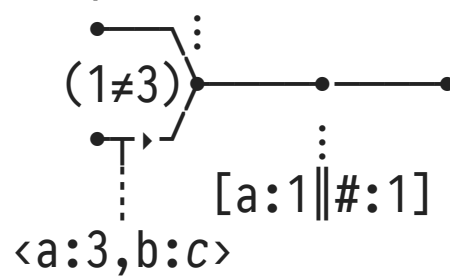
14 <a:1,b:A>



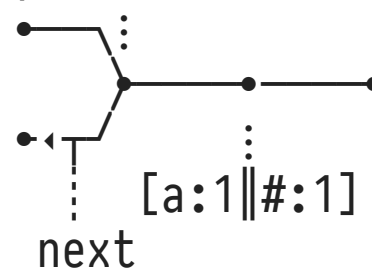
15 <a:1,b:A>



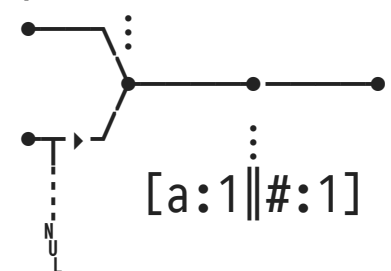
16 <a:1,b:A>



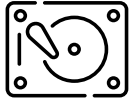
17 <a:1,b:A>



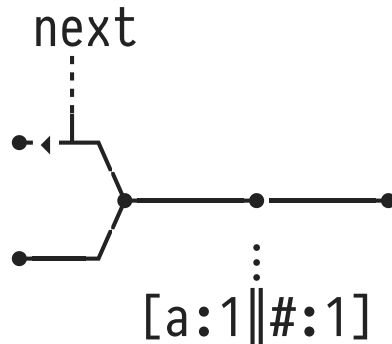
18 <a:1,b:A>



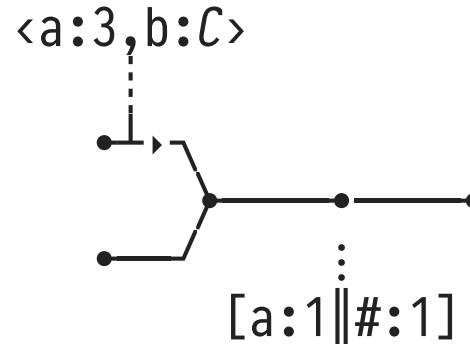
Volcano Iterator Model: Evaluating a NLJ Plan



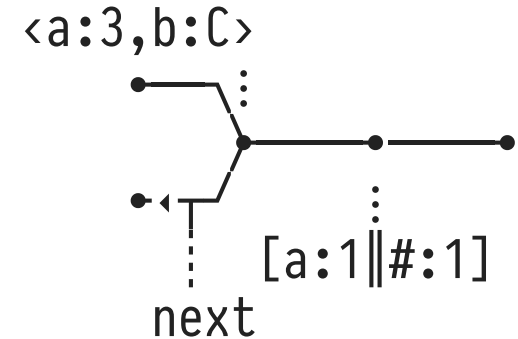
19



20



21

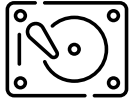


Quiz/Exercise: Think about how to implement the following plan operators in the Volcano iterator model:

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

- Seq Scan
 - Local state kept between method invocations: pointer into heap file.
 - `next()` reads rows from heap file until `Filter` condition is satisfied (or heap file exhausted).
- Limit
 - Local state kept between method invocations: current of value of `n`.
 - Decrement `n` on every invocation of `next()`, return `1` if `n = 0`.
- GroupAggregate
 - Local state kept between method invocations: none.
 - `next()` reads rows from input until group key changes, aggregates, then emits aggregate value.
 - Can pipeline aggregate values since input is sorted.
- Append
 - Local state kept between method invocations: none.
 - `next()` exhausts first input, then switches over to second input.

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
1 DECLARE <cursor> [ SCROLL ] CURSOR FOR <query>
--                               ^ cursor can move backwards

-- Evaluate plan to deliver the next/prior row (<n> rows)
2 FETCH [ NEXT | [ FORWARD | BACKWARD ] <n> ] FROM <cursor>

-- Release plan/intermediate buffers
3 CLOSE <cursor>
```

- Statements need to be issued within an SQL transaction.

Demonstrate the SQL-level Volcano-style cursor interface.

```
-- 1 Check input tables
\d one
      Table "public.one"

```

Column	Type	Collation	Nullable	Default
a	<u>integer</u>		not null	
b	<u>text</u>			
c	<u>integer</u>			

```

Indexes:
    "one_a" 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	<u>integer</u>		not null	
b	<u>text</u>			
c	<u>integer</u>		not null	

```

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

```

```
-- 2 Declare/fetch/close cursor for join query (within a SQL transaction)

BEGIN;

DECLARE pipeline SCROLL CURSOR FOR
  SELECT DISTINCT o.a, o.b || m.b AS md5
  FROM   one AS o, many AS m
  WHERE  o.a = m.a;

FETCH NEXT pipeline;
```

a	md5

1	c4ca4238a0b923820dcc509a6f75849b
---	----------------------------------

Time: 896.523 ms -- + ⚠ first fetch takes time (evaluate the blocking DISTINCT)

FETCH NEXT pipeline;

a	md5
2	c81e728d9d4c2f636f067f89cc14862c

Time: 0.177 ms -- + ⚠ subsequent fetches are immediate

FETCH FORWARD 3 pipeline;

a	md5
3	eccbc87e4b5ce2fe28308fd9f2a7baf3
4	a87ff679a2f3e71d9181a67b7542122c
5	e4da3b7fbbce2345d7772b0674a318d5

Time: 0.252 ms

FETCH BACKWARD 2 pipeline;

a	md5
4	a87ff679a2f3e71d9181a67b7542122c
3	eccbc87e4b5ce2fe28308fd9f2a7baf3

Time: 0.231 ms

CLOSE pipeline;

COMMIT;

⚠ **Nice:** Illustrate the implementation of plan operator **Append** (SQL: **UNION ALL**). **Q:** Let students speculate about the behavior:

```
-- Query will generate six rows (i = 1...6)
EXPLAIN
SELECT i FROM generate_series(1,3) AS i
UNION ALL
(SELECT i
FROM generate_series(10000000,4,-1) AS i
ORDER BY i
LIMIT 3);
```

QUERY PLAN

```
Append (cost=0.00..32.97 rows=1003 width=4)
-> Function Scan on generate_series i (cost=0.00..10.00 rows=1000 width=4)
-> Limit (cost=22.93..22.93 rows=3 width=4)
    -> Sort (cost=22.93..25.43 rows=1000 width=4)
        Sort Key: i_1.i
        -> Function Scan on generate_series i_1 (cost=0.00..10.00 rows=1000 width=4)
```

```
BEGIN;
```

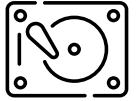
```
DECLARE pipeline CURSOR FOR
SELECT i FROM generate_series(1,3) AS i
UNION ALL
(SELECT i
FROM generate_series(10000000,4,-1) AS i
ORDER BY i
LIMIT 3);
```

```
FETCH NEXT pipeline; -- takes    22 ms [first row from left leg of UNION ALL]
FETCH NEXT pipeline; --         0.2 ms [subsequent rows from left leg]
FETCH NEXT pipeline; --         0.2 ms
FETCH NEXT pipeline; --        5000+ ms [first row from right leg, evaluates costly blocking Sort]
FETCH NEXT pipeline; --         0.2 ms [subsequent rows from right leg]
FETCH NEXT pipeline; --         0.2 ms
```

```
CLOSE pipeline;
```

```
COMMIT;
```

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*.