

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} :

```

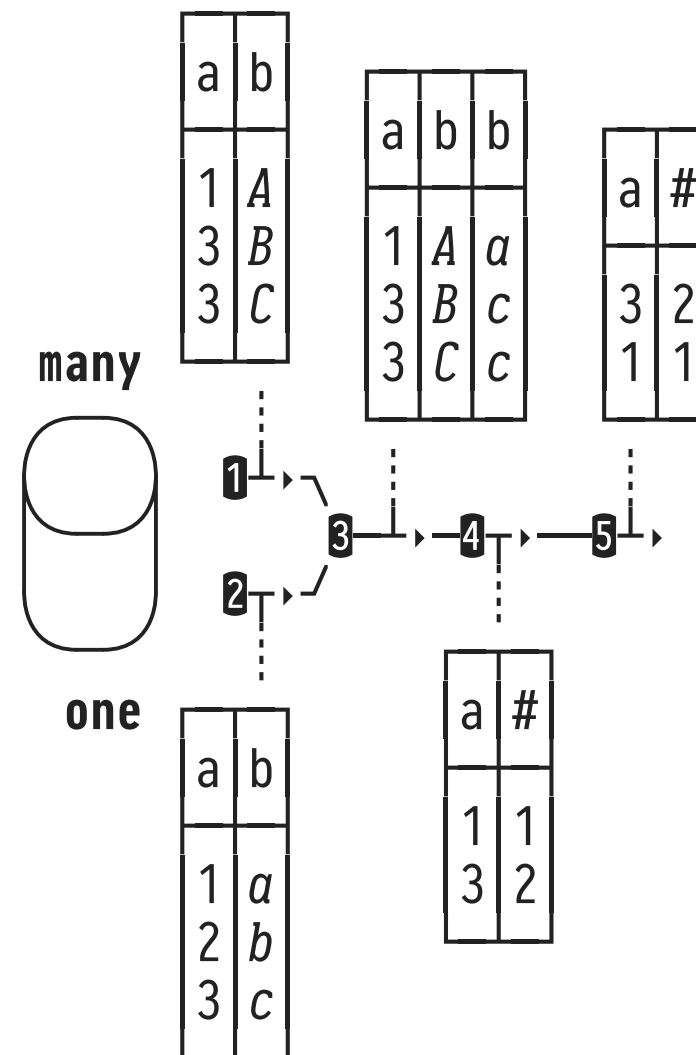
SELECT o.a, COUNT(*) AS "#"
FROM   one AS o, many AS m
WHERE  o.a = m.a
GROUP BY o.a
ORDER BY o.a DESC
  
```

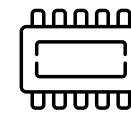
- Plan operators:

- 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



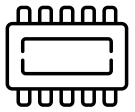


MonetDB: Full Materialization

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.



Data Dependencies in MAL Program for Q_{12}

```

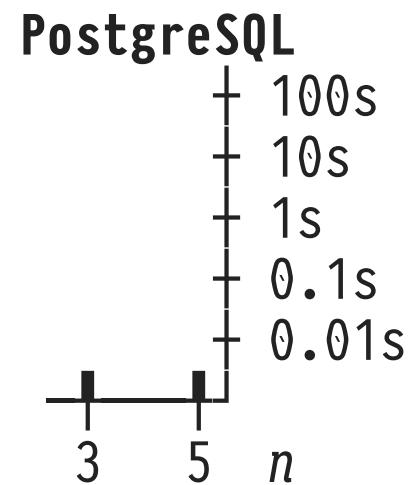
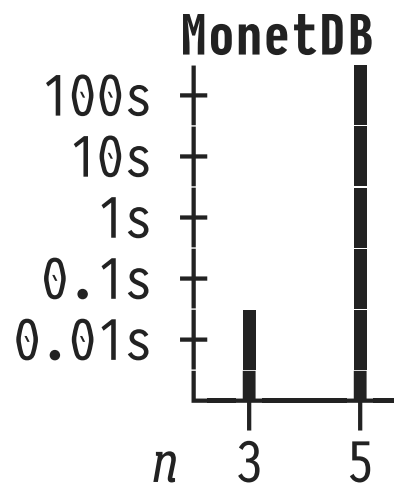
one      :bat[:oid] := sql.tid(sql, "sys", "one");
one_a0   :bat[:int] := sql.bind(sql, "sys", "one", "a", 0:int);
one_a   :bat[:int] := algebra.projection(one, one_a0);
} 1 Scan one.a
    1 // 2
many     :bat[:oid] := sql.tid(sql, "sys", "many");
many_a0  :bat[:int] := sql.bind(sql, "sys", "many", "a",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







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). 



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 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>.



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).

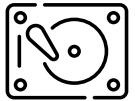


Volcano Iterator Model: API

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

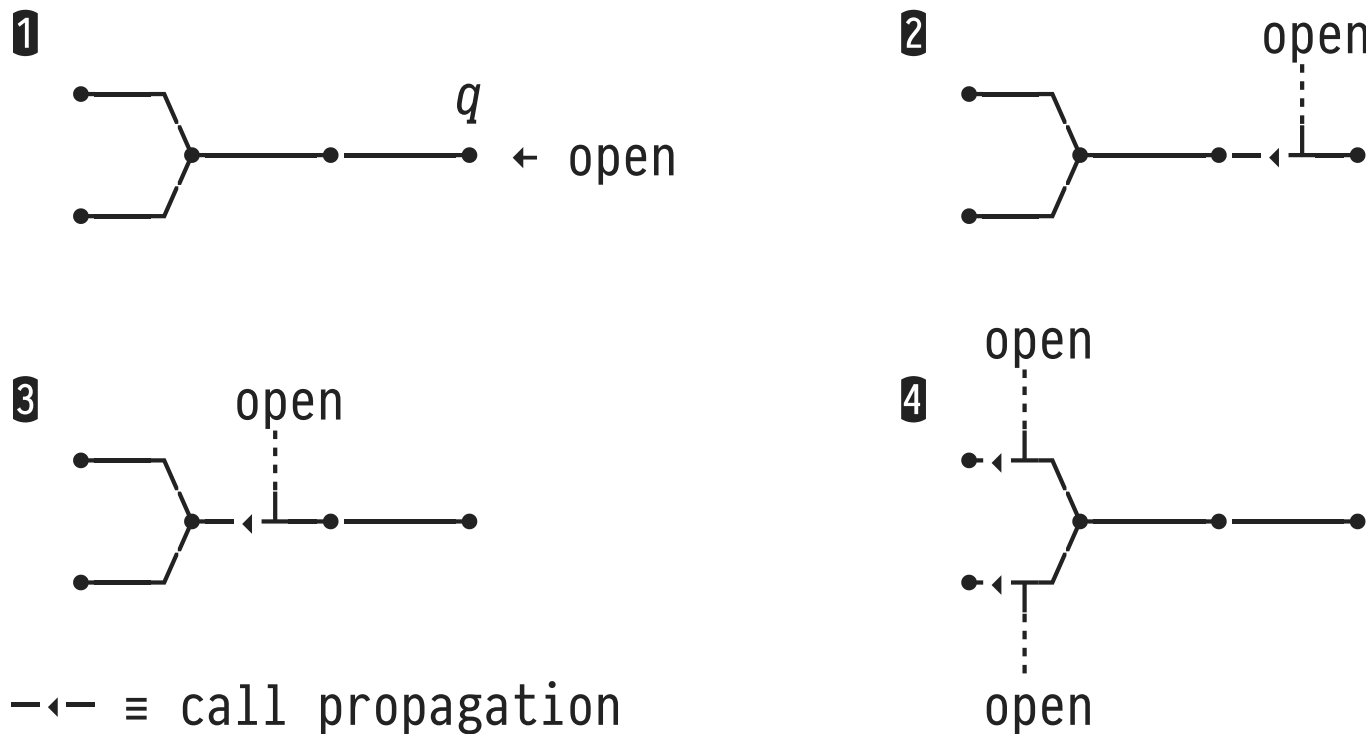
Use the Volcano iterator model API to *fully* evaluate a query. Operator q denotes the root of the query plan:

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

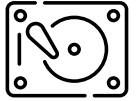
- To retrieve next result row only, simply call `next(q)`.
- May/must use `close(q)` to cancel query evaluation midway.



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



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



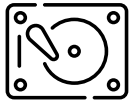
Pipelined Nested Loop Join (NLJ) ①

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 ← true;   } local NLJ state
    0              ← NL;    }
```

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



Pipelined Nested Loop Join (NLJ) ②

```

NLJ.next(outer,inner, $\theta$ ):
  forever
    if needNewOuter
      o  $\leftarrow$  next(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

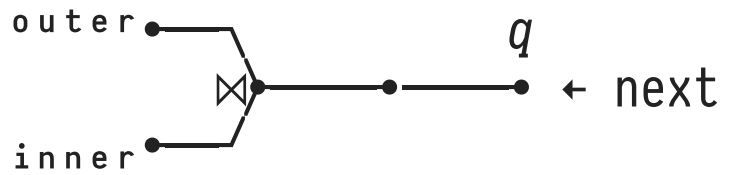
      i  $\leftarrow$  next(inner);               } i: current inner row
      if i =  $\text{NULL}$                          } no more inner rows,
        [ needNewOuter  $\leftarrow$  true;      } next time: read new outer
      else if o  $\theta$  i                     } join condition satisfied?
        [ return  $\langle o, i \rangle$ ;           } return single joined pair

```

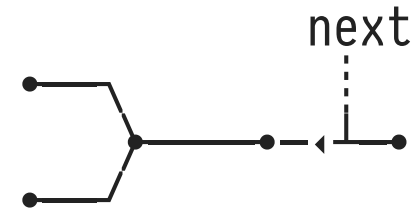


Volcano Iterator Model: Evaluating a NLJ Plan

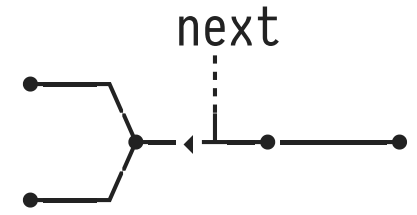
1



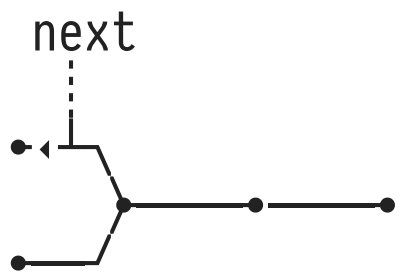
2



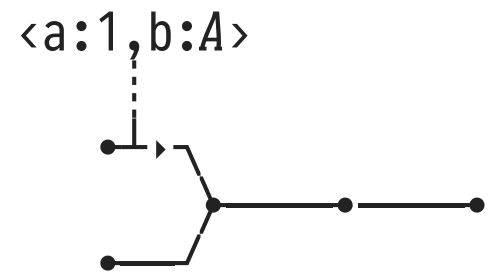
3



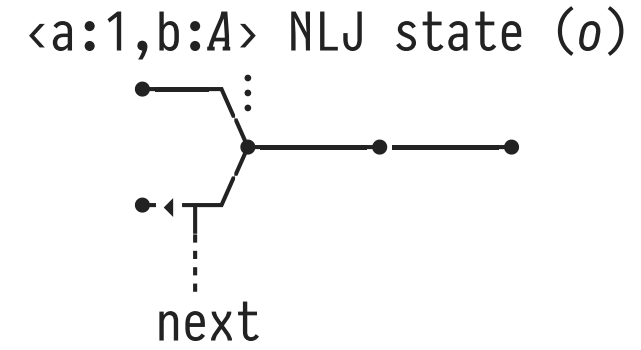
4



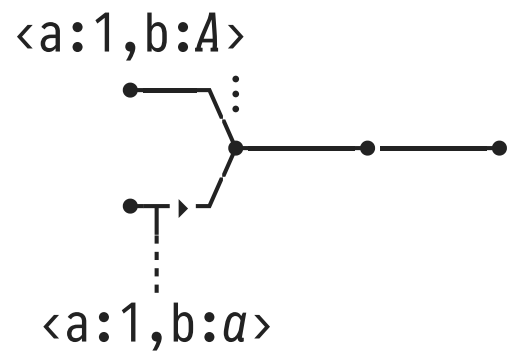
5



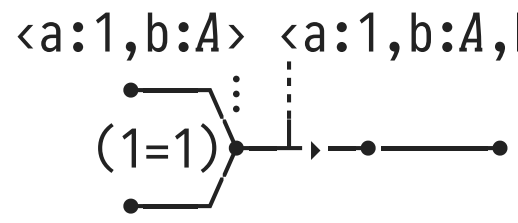
6



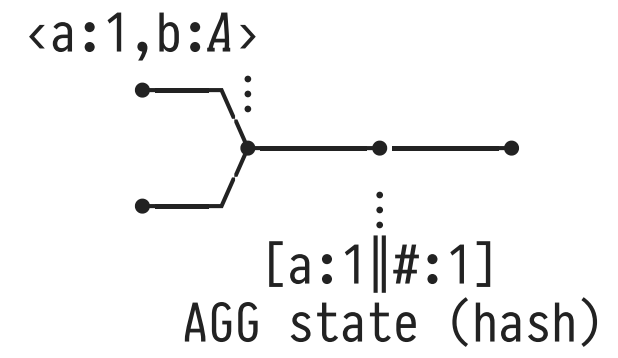
7

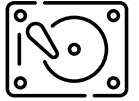


8



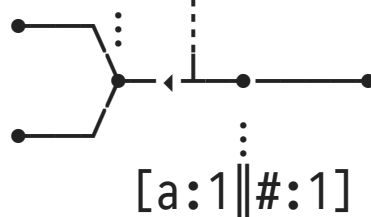
9



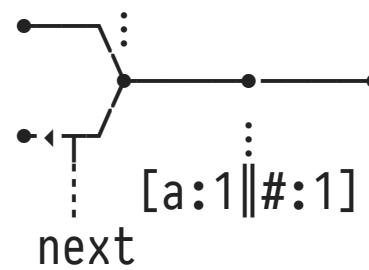


Volcano Iterator Model: Evaluating a NLJ Plan

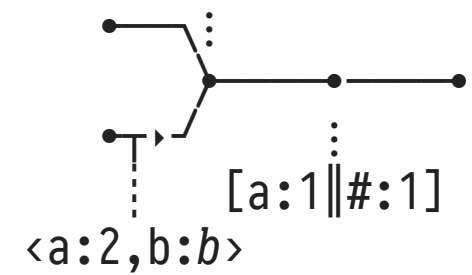
10 <a:1,b:A> next



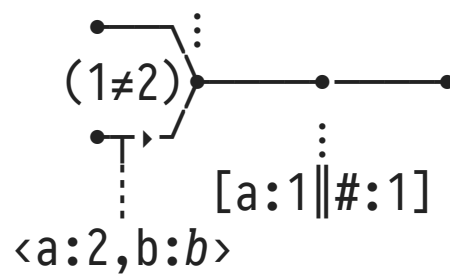
11 <a:1,b:A>



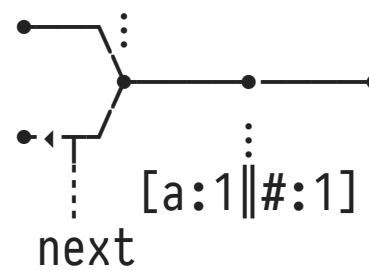
12 <a:1,b:A>



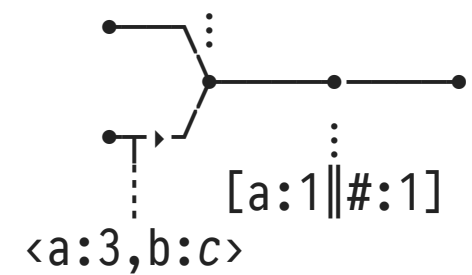
13 <a:1,b:A>



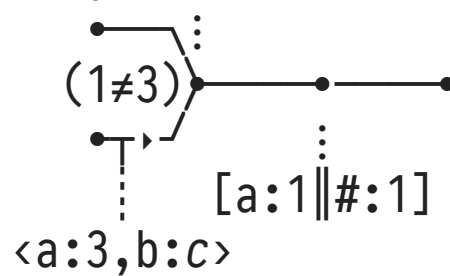
14 <a:1,b:A>



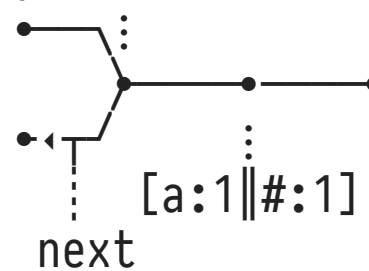
15 <a:1,b:A>



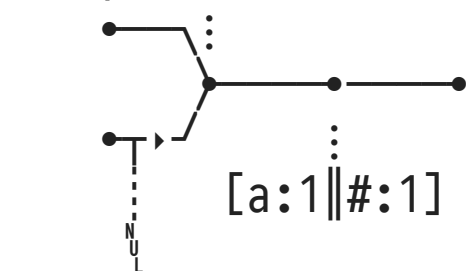
16 <a:1,b:A>



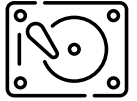
17 <a:1,b:A>



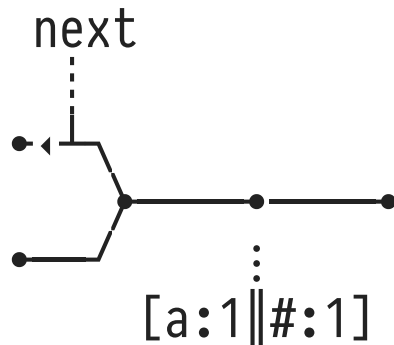
18 <a:1,b:A>



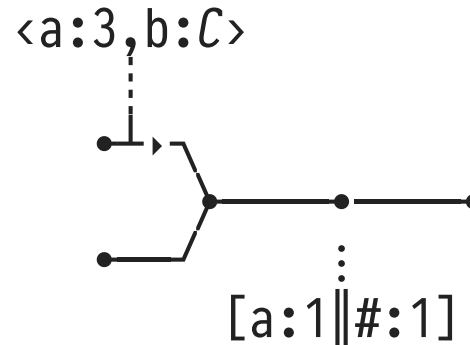
Volcano Iterator Model: Evaluating a NLJ Plan



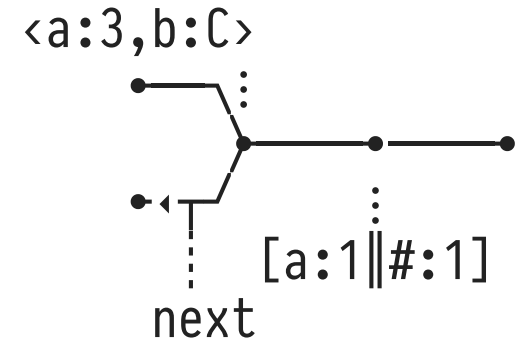
19



20

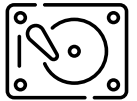


21



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).



Volcano Iterator Model at the SQL Level

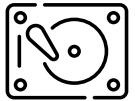
Via **cursors**, the SQL standard exposes the Volcano-style **open/next/close** API at the level of (Embedded) SQL:

```
-- 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.



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*.