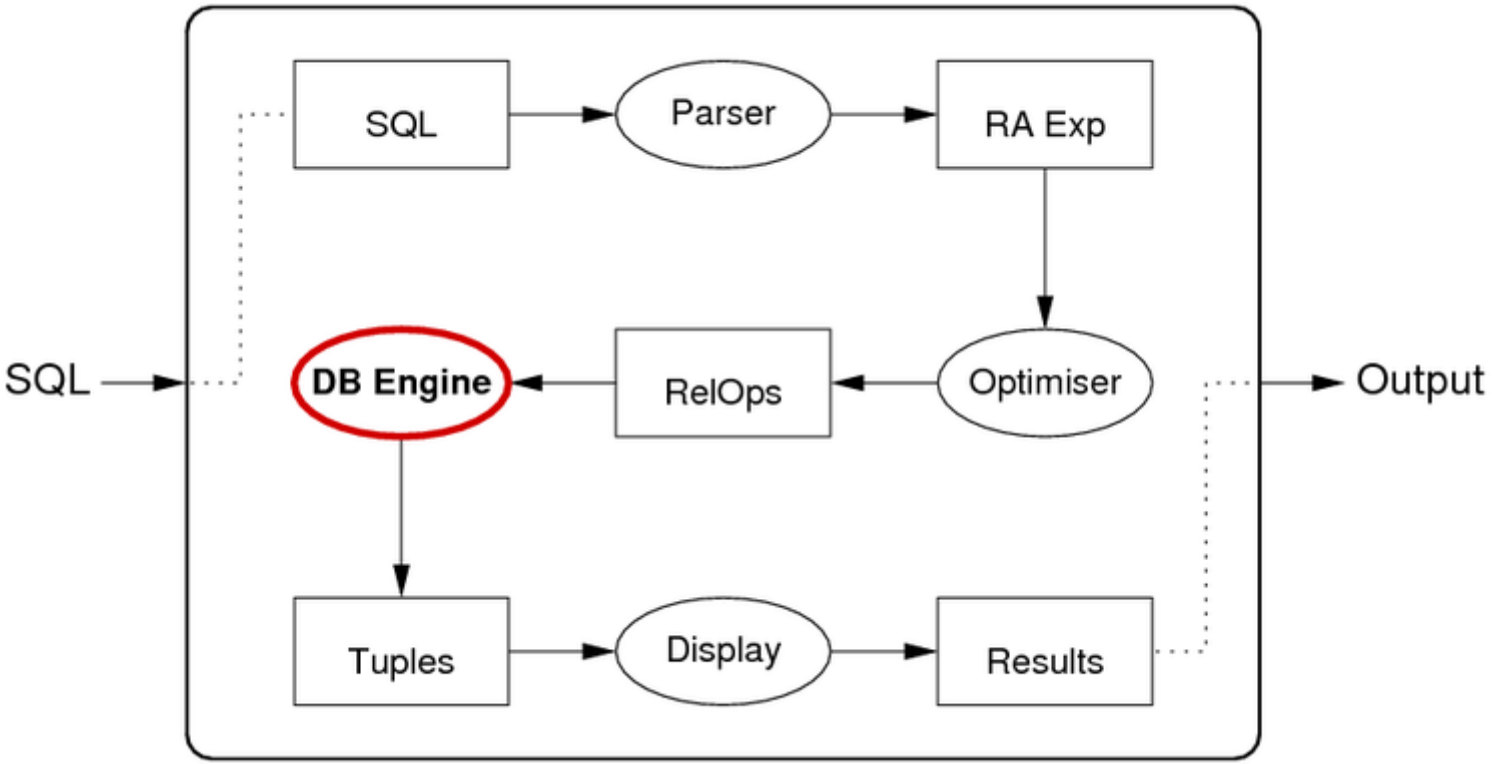
>>

Query Execution

/\ >>

❖ Query Execution

Query execution: applies evaluation plan → result tuples



❖ Query Execution (cont)

Example of query translation:

```
select s.name, s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and e.semester = '05s2';
```

maps to

$\pi_{name,id,course,mark}(Stu \bowtie_{e.student=s.id} (\sigma_{semester=05s2} Enr))$

maps to

```
Temp1  = BtreeSelect[semester=05s2](Enr)
Temp2  = HashJoin[e.student=s.id](Stu,Temp1)
Result = Project[name,id,course,mark](Temp2)
```

<< ^ >>

❖ Query Execution (cont)

A query execution plan:

- consists of a collection of RelOps
- executing together to produce a set of result tuples

Results may be passed from one operator to the next:

- materialization ... writing results to disk and reading them back
- pipelining ... generating and passing via memory buffers

<< ^ >>

❖ Materialization

Steps in materialization between two operators

- first operator reads input(s) and writes results to disk
- next operator treats tuples on disk as its input
- in essence, the **`Temp`** tables are produced as real tables

Advantage:

- intermediate results can be placed in a file structure
  (which can be chosen to speed up execution of subsequent operators)

Disadvantage:

- requires disk space/writes for intermediate results
- requires disk access to read intermediate results

<< ^ >>

❖ Pipelining

How pipelining is organised between two operators:

- operators execute "concurrently" as producer/consumer pairs
- structured as interacting iterators (open; while(next); close)

Advantage:

- no requirement for disk access (results passed via memory buffers)

Disadvantage:

- higher-level operators access inputs via linear scan, or
- requires sufficient memory buffers to hold all outputs

<< ^ >>

❖ Iterators (reminder)

Iterators provide a "stream" of results:

- **`iter = startScan(`***params*`)`**
  - set up data structures for iterator (create state, open files, ...)
  - *params* are specific to operator (e.g. reln, condition, #buffers, ...)
- **`tuple = nextTuple(iter)`**
  - get the next tuple in the iteration; return null if no more
- **`endScan(iter)`**
  - clean up data structures for iterator

Other possible operations: reset to specific point, restart, ...

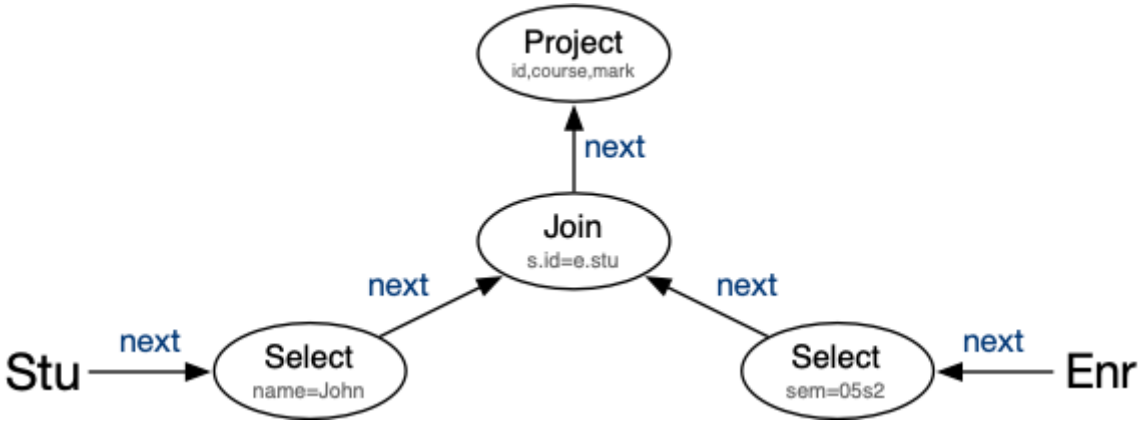
<< ^ >>

❖ Pipelining Example

Consider the query:

```
select s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and
       e.semester = '05s2' and s.name = 'John';
```

Evaluated via communication between RA tree nodes:



Note: likely that projection is combined with join in PostgreSQL

<< ^ >>

❖ Disk Accesses

Pipelining cannot avoid all disk accesses.

Some operations use multiple passes (e.g. merge-sort, hash-join).

- data is written by one pass, read by subsequent passes

Thus ...

- within an operation, disk reads/writes are possible
- between operations, no disk reads/writes are needed

<< ^ >>

❖ PostgreSQL Query Execution

Defs: **`src/include/executor`** and **`src/include/nodes`**

Code: **`src/backend/executor`**

PostgreSQL uses pipelining (as much as possible) ...

- query plan is a tree of **`Plan`** nodes
- each type of node implements one kind of RA operation
  (node implements specific access method via iterator interface)
- node types e.g. **`Scan`**, **`Group`**, **`Indexscan`**, **`Sort`**, **`HashJoin`**
- execution is managed via a tree of **`PlanState`** nodes
  (mirrors the structure of the tree of Plan nodes; holds execution state)

<< ^ >>

❖ PostgreSQL Executor

Modules in **`src/backend/executor`** fall into two groups:

`execXXX` (e.g. execMain, execProcnode, execScan)

- implement generic control of plan evaluation (execution)
- provide overall plan execution and dispatch to node iterators

`nodeXXX` (e.g. nodeSeqscan, nodeNestloop, nodeGroup)

- implement iterators for specific types of RA operators
- typically contains **`ExecInitXXX`**, **`ExecXXX`**, **`ExecEndXXX`**

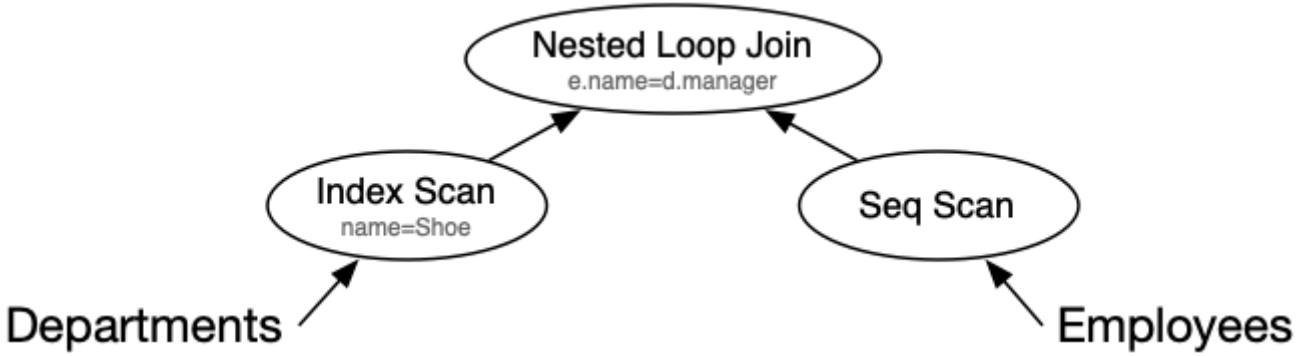
<< ^ >>

❖ Example PostgreSQL Execution

Consider the query:

```
-- get manager's age and # employees in Shoe department
select e.age, d.nemps
from   Departments d, Employees e
where  e.name = d.manager and d.name ='Shoe'
```

and its execution plan tree



<< ^ >>

❖ Example PostgreSQL Execution (cont)

Initially **`InitPlan()`** invokes **`ExecInitNode()`** on plan tree root.

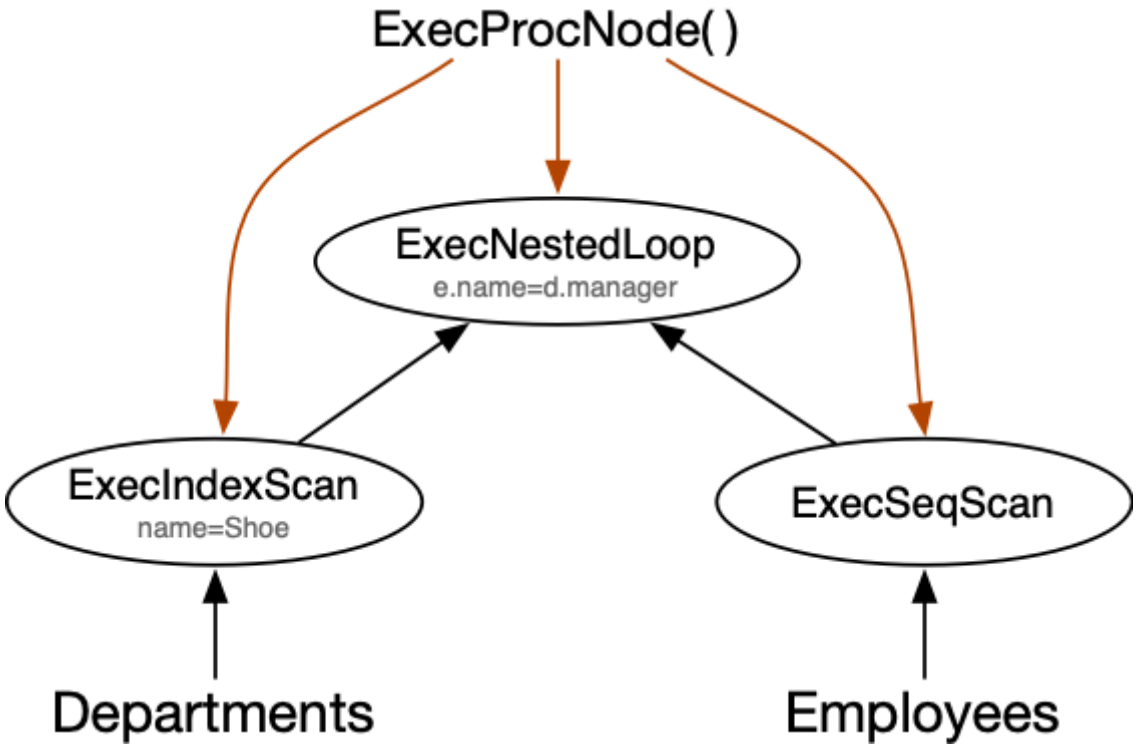
`ExecInitNode()` sees a **`NestedLoop`** node ...
so dispatches to **`ExecInitNestLoop()`** to set up iterator
then invokes **`ExecInitNode()`** on left and right sub-plans
in left subPlan, **`ExecInitNode()`** sees an **`IndexScan`** node
so dispatches to **`ExecInitIndexScan()`** to set up iterator
in right sub-plan, **`ExecInitNode()`** sees a **`SeqScan`** node
so dispatches to **`ExecInitSeqScan()`** to set up iterator

Result: a plan *state* tree with same structure as plan tree.

<< ^ >>

❖ Example PostgreSQL Execution (cont)

Plan state tree (collection of iterators):



<< ^

❖ Example PostgreSQL Execution (cont)

Then **`ExecutePlan()`** repeatedly invokes **`ExecProcNode()`**.

`ExecProcNode()` sees a **`NestedLoop`** node ...
  so dispatches to **`ExecNestedLoop()`** to get next tuple
  which invokes **`ExecProcNode()`** on its sub-plans
    in left sub-plan, **`ExecProcNode()`** sees an **`IndexScan`** node
      so dispatches to **`ExecIndexScan()`** to get next tuple
      if no more tuples, return END
      for this tuple, invoke **`ExecProcNode()`** on right sub-plan
        `ExecProcNode()` sees a **`SeqScan`** node
          so dispatches to **`ExecSeqScan()`** to get next tuple
          check for match and return joined tuples if found
          continue scan until end
        reset right sub-plan iterator

Result: stream of result tuples returned via **`ExecutePlan()`**

Produced: 6 Apr 2021