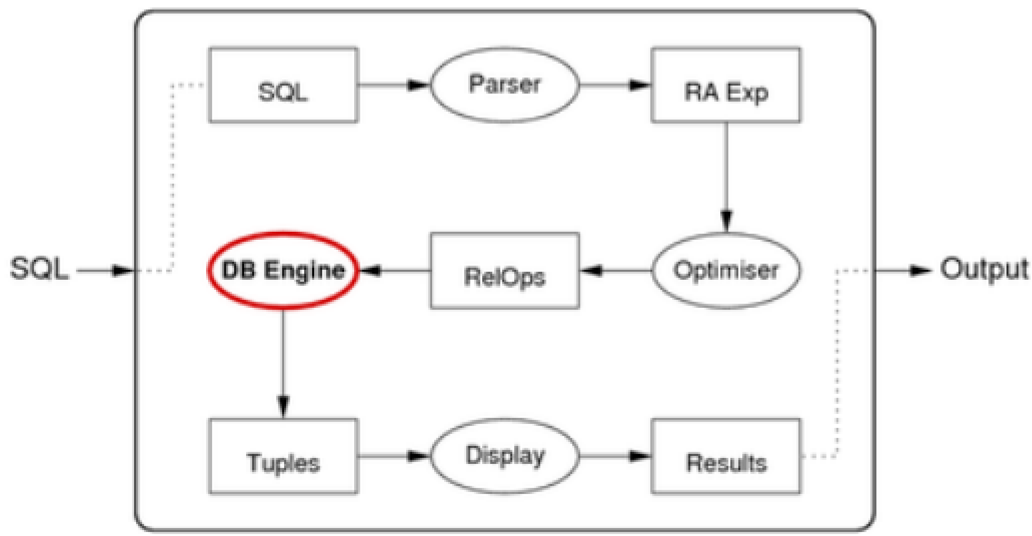
# course 10 query process part 2-query execution

## 1. Query execution

query execution:

***applies evaluation plans generated from Optimiser → produce a set of result tuples***



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)
```

a query execution consist of :

- consists of sequence of operations
- each operations is a RA operator

result may be passed from one RA operator to another, with two methods

- materialization: writing results to disk and reading them back
- Pipelining: generating and passing results one-at-a-time

## 1.1 Materialization

process:

- first operator reads inputs and writes result to disk
- next operators treats tuples results on disk as its input

advantages:

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

disadvantages:

- require disk space to store intermediate results
- and require extra time to access disk (read/write operations)

Example:

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

might be executed as

```
Temp1  = BtreeSelect[semester=05s2](Enrolment)
Temp2  = BtreeSelect[name=John](Student)
         -- indexes on name and semester
         -- produce sorted Temp1 and Temp2
Temp3  = SortMergeJoin[e.student=s.id](Temp1,Temp2)
         -- SMJoin especially effective, since
         -- Temp1 and Temp2 are already sorted
Result = Project[name,id,course,mark](Temp3)
```

## 1.2 Pipelining

process:

- blocks execute "concurrently" as **producer/consumer** pairs
  - first operator acts as producer and the second as consumer
- structured as interacting iterators (open; while(next); close)

Advantage:

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

Disadvantage:

- each operator accesses inputs via **linear scan**
  - (in materialization, intermediate data are store in specific data structure, like sorted structure)

examples:

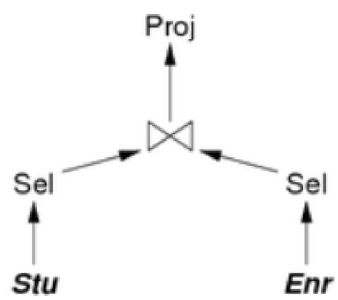
Consider the query:

```
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';
```

which maps to the RA expression

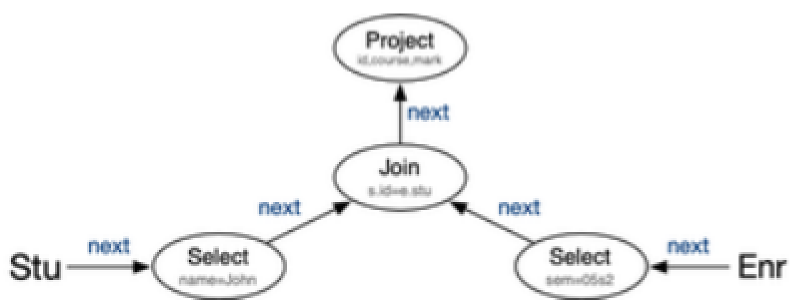
*Proj[id,course,mark](Join[student=id](Sel[05s2](Enr),Sel[John](Stu)))*

which could represented by the RA expression tree




the RA tree is shown below:

Modelled as communication between RA tree nodes:




```
System:
  iter0 = open(Result)
  while (Tup = next(iter0)) { display Tup }
  close(iter0)
  Result:
  iter1 = open(Join)
  while (T = next(iter1))
  { T' = project(T); return T' }
  close(iter1)

Sel1:
```

```
  iter4 = open(Btree(Enrolment,'semester=05s2'))
  while (A = next(iter4)) { return A }
  close(iter4)

Join: -- nested-loop join
  iter2 = open(Sel1)
  while (R = next(iter2) {
  iter3 = open(Sel2)
  while (S = next(iter3))
  { if (matches(R,S) return (RS) }
  close(iter3) // better to reset(iter3)
  }
  close(iter2)

Sel2:
  iter5 = open(Btree(Student,'name=John'))
  while (B = next(iter5)) { return B }
  close(iter5)
```

Piplines can be executed as:

- ***Demand-driven***
  producers wait until consumers request tuples
- ***Producer-driven***
  producers generate tuples until output buffer full, then wait

In both cases, top-level driver is request for result tuples. **In parallel-processing systems, iterators could run concurrently.**

## 1.3 Disk Accesses

sometimes we are not rigidly only use single Materializtion / pipelining methods, we will use both two methods together.

you know, for Pipelining methods, although between operations there are no disk read/write needed, disk may frequently accessed within one operation.

so **sophisticated query optiisers** might be:

> *if operation X writes its results to a file with structure S,*
> *the subsequent operation Y will proceed much faster*
> *than if Y reads X's output tuple–at–a–time*

In this case, it could materialize X's output in an S-file.
Produces a pipeline/materialization hybrid query execution.

Example:

- selection writes output into an indexed file (Btree)
- later join can then be implemented as efficient index–join

example:

Example: (pipeline/materialization hybrid)

```
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';
```

might be executed as

```
System:
  exec(Sel2) -- creates Temp1
  iter0 = open(Result)
  while (Tup = next(iter0)) { display Tup }
  close(iter0)

Result:
  iter1 = open(Join)
  while (T = next(iter1))
    { T' = project(T); return T' }
  close(iter1)

Join: -- index join
  iter2 = open(Sel1)
  while (R = next(iter2) {
    iter3 = open(Btree(Temp1,'id=R.student'))
    while (S = next(iter3)) { return (RS) }
    close(iter3)
  }
  close(iter2)

Sel1:
  iter4 = open(Btree(Enrolment,'semester=05s2'))
  while (A = next(iter4)) { return A }
  close(iter4)
```

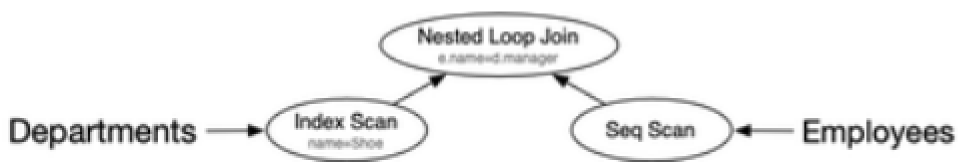
```
Sel2:
  iter5 = open(Btree(Student,'name=John'))
  createBtree(Temp1,'id')
  while (B = next(iter5)) { insert(B,Temp1) }
  close(iter5)
```

## 1.4 PostgreSQL Execution

for example:

```
-- 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'
```

this query's plan tree is :




this produces a tree with three nodes:

- **NestedLoop** with join condition
- **IndexScan** on Departments with selection
- **SeqScan** on Employees

Initially, function InitPlan() invokes ExecInitNode() on plan tree root.

```
ExecInitNode() sees a NestedLoop node ...
    so dispatches to ExecInitNestLoop() to set up iterator
    and 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 aSeqScan node
            so dispatches to ExecInitSeqScan() to set up iterator
```

result: a **plan state tree** with same structure as plan tree

Execution: function 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 the 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
            reset right sub-plan iterator
```

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

# 2. Performance Tuning

Performance can be considered at two times:

- *during* schema design
  - typically towards the end of schema design process
  - requires schema transformations such as *denormalisation*
- *outside* schema design
  - typically after application has been deployed/used
  - requires adding/modifying data structures such as *indexes*

## 2.1 Denormalisation

## 2.2 indexes

## 2.3 query tuning