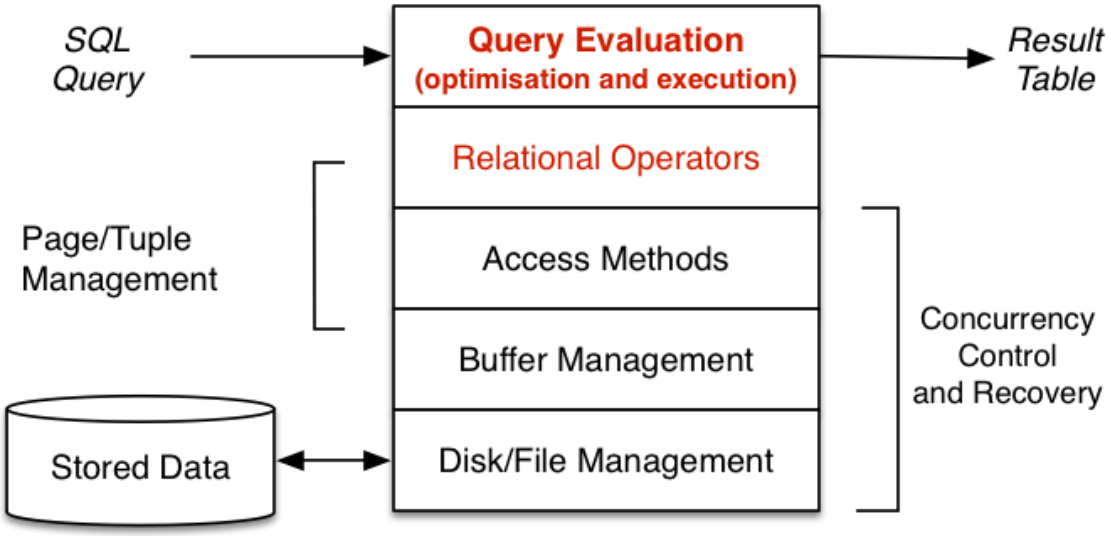
Query Processing

❖ Query Processing



❖ Query Processing (cont)

A query in SQL:

- states *what* kind of answers are required (declarative)
- does not say *how* they should be computed (procedural)

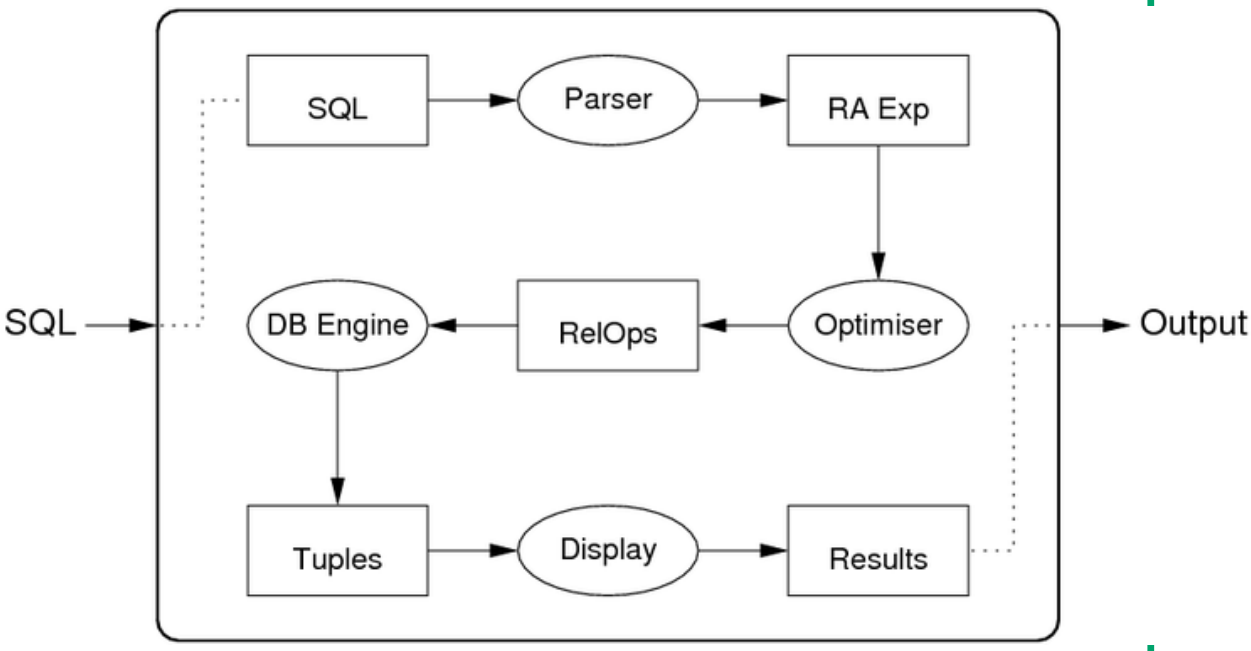
A query evaluator/processor :

- takes declarative description of query (in SQL)
- parses query to internal representation (relational algebra)
- determines plan for answering query (expressed as DBMS ops)
- executes method via DBMS engine (to produce result tuples)

Some DBMSs can save query plans for later re-use.

❖ Query Processing (cont)

Internals of the query evaluation "black-box":



❖ Query Processing (cont)

DBMSs provide several "flavours" of each RA operation.

For example:

- several "versions" of selection (σ) are available
- each version is effective for a particular kind of selection, e.g

```
select * from R where id = 100   -- hashing
select * from S                  -- Btree index
where age > 18 and age < 35
select * from T                  -- MALH file
where a = 1 and b = 'a' and c = 1.4
```

Similarly, π and $\bowtie$ have versions to match specific query types.

❖ Query Processing (cont)

We call these specialised version of RA operations RelOps.

One major task of the query processor:

- given a RA expression to be evaluated
- find a combination of RelOps to do this efficiently

Requires the query translator/optimiser to consider

- information about relations (e.g. sizes, primary keys, ...)
- information about operations (e.g. selection reduces size)

RelOps are realised at execution time

- as a collection of inter-communicating nodes
- communicating either via pipelines or temporary relations

❖ Terminology Variations

Relational algebra expression of SQL query

- intermediate query representation
- logical query plan

Execution plan as collection of RelOps

- query evaluation plan
- query execution plan
- physical query plan

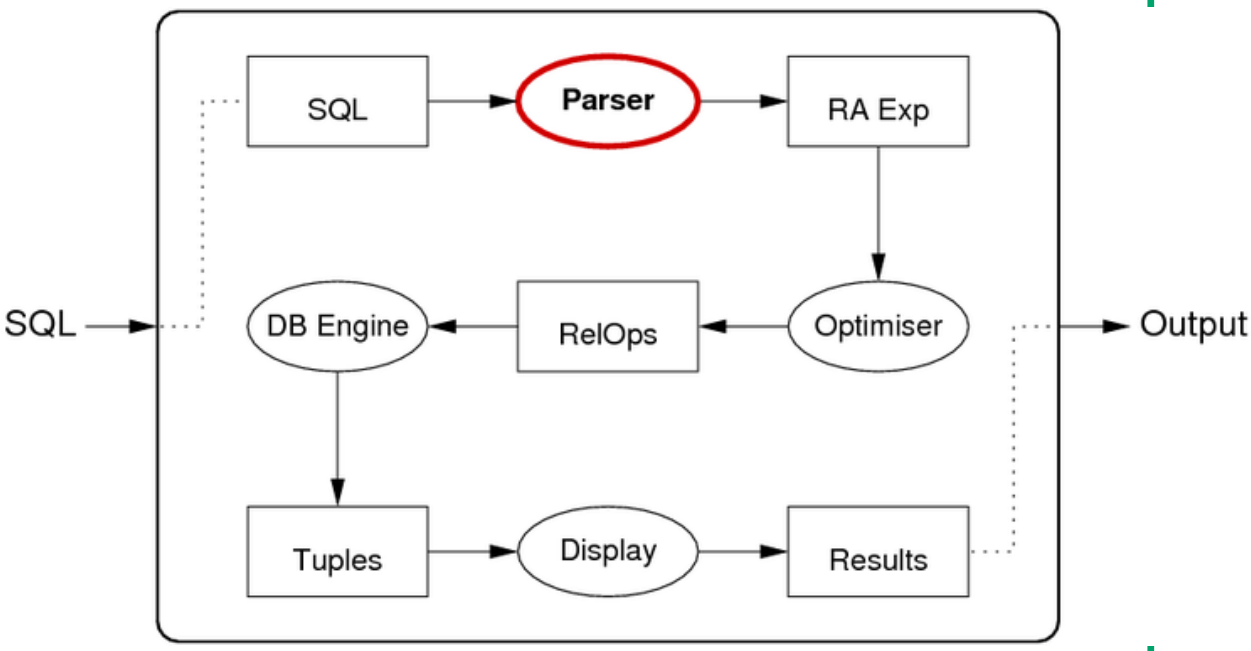
Representation of RA operators and expressions

- σ = *Select* = *Sel*, Π = *Project* = *Proj*
- $R \bowtie S$ = *R Join S* = *Join(R,S)*, $\wedge$ = &, $\vee$ = |

Query Translation

❖ Query Translation

Query translation: SQL statement text → RA expression



❖ Query Translation (cont)

Translation step: SQL text → RA expression

Example:

```
SQL: select name from Students where id=7654321;
-- is translated to
RA:  Proj[name](Sel[id=7654321]Students)
```

Processes: lexer/parser, mapping rules, rewriting rules.

Mapping from SQL to RA may include some optimisations, e.g.

```
select * from Students where id = 54321 and age > 50;
-- is translated to
Sel[age>50](Sel[id=54321]Students)
-- rather than ... because of index on id
Sel[id=54321&age>50](Students)
```

❖ Parsing SQL

Parsing task is similar to that for programming languages.

Language elements:

- keywords: **`create`**, **`select`**, **`from`**, **`where`**, ...
- identifiers: **`Students`**, **`name`**, **`id`**, **`CourseCode`**, ...
- operators: **`+`**, **`-`**, **`=`**, **`<`**, **`>`**, **`AND`**, **`OR`**, **`NOT`**, **`IN`**, ...
- constants: **`'abc'`**, **`123`**, **`3.1`**, **`'01-jan-1970'`**, ...

PostgreSQL parser ...

- implemented via lex/yacc (**`src/backend/parser`**)
- maps all identifiers to lower-case (A-Z → a-z)
- needs to handle user-extendable operator set
- makes extensive use of catalog (**`src/backend/catalog`**)

❖ Expression Rewriting Rules

Since RA is a well-defined formal system

- there exist many algebraic laws on RA expressions
- which can be used as a basis for expression rewriting
- in order to produce equivalent (more-efficient?) expressions

Expression transformation based on such rules can be used

- to simplify/improve SQL→RA mapping results
- to generate new plan variations to check in query optimisation

❖ Relational Algebra Laws

Commutative and Associative Laws:

- $R \bowtie S \leftrightarrow S \bowtie R, \quad (R \bowtie S) \bowtie T \leftrightarrow R \bowtie (S \bowtie T)$ (natural join)
- $R \cup S \leftrightarrow S \cup R, \quad (R \cup S) \cup T \leftrightarrow R \cup (S \cup T)$
- $R \bowtie_{Cond} S \leftrightarrow S \bowtie_{Cond} R$ (theta join)
- $\sigma_c (\sigma_d (R)) \leftrightarrow \sigma_d (\sigma_c (R))$

Selection splitting (where c and d are conditions):

- $\sigma_{c \wedge d}(R) \leftrightarrow \sigma_c (\sigma_d (R))$
- $\sigma_{c \vee d}(R) \leftrightarrow \sigma_c(R) \cup \sigma_d(R)$

❖ Relational Algebra Laws (cont)

Selection pushing ($\sigma_c(R \cup S)$ and $\sigma_c(R \cup S)$):

- $\sigma_c(R \cup S) \leftrightarrow \sigma_c R \cup \sigma_c S$, $\sigma_c(R \cap S) \leftrightarrow \sigma_c R \cap \sigma_c S$

Selection pushing with join ...

- $\sigma_c (R \bowtie S) \leftrightarrow \sigma_c(R) \bowtie S$ (if c refers only to attributes from R)
- $\sigma_c (R \bowtie S) \leftrightarrow R \bowtie \sigma_c(S)$ (if c refers only to attributes from S)

If *condition* contains attributes from both R and S:

- $\sigma_{c' \wedge c''} (R \bowtie S) \leftrightarrow \sigma_{c'}(R) \bowtie \sigma_{c''}(S)$
- c' contains only R attributes, c'' contains only S attributes

❖ Relational Algebra Laws (cont)

Rewrite rules for projection ...

All but last projection can be ignored:

- $\Pi_{L1}\ (\ \Pi_{L2}\ (\ ...\ \Pi_{Ln}\ (R)))\ \ \rightarrow\ \ \Pi_{L1}\ (R)$

Projections can be pushed into joins:

- $\Pi_{L}\ (R\ \bowtie_{c} S)\ \ \leftrightarrow\ \ \Pi_{L}\ (\ \Pi_{M}(R)\ \bowtie_{c}\ \Pi_{N}(S)\ )$

where

- M and N must contain all attributes needed for c
- M and N must contain all attributes used in L ($L \subset M \cup N$)

❖ Query Rewriting

Subqueries ⇒ convert to a join

Example: (on schema Courses(id,code,...), Enrolments(cid,sid,...), Students(id,name,...)

```
select c.code, count(*)
from   Courses c
where  c.id in (select cid from Enrolments)
group  by c.code
```

becomes

```
select c.code, count(*)
from   Courses c join Enrolments e on c.id = e.cid
group  by c.code
```

❖ Query Rewriting (cont)

But not all subqueries can be converted to join, e.g.

```
select e.sid as student_id, e.cid as course_id
from   Enrolments e
where  e.sid = (select max(id) from Students)
```

has to be evaluated as

Val = max[id]Students

Res = $\pi_{(sid,cid)}(\sigma_{sid=Val}$Enrolments)

❖ Query Rewriting (cont)

In PostgreSQL, views are implemented via rewrite rules

- a reference to view in SQL expands to its definition in RA

Example:

```
create view COMP9315studes as
select stu,mark from Enrolments where course='COMP9315';
-- students who passed
select stu from COMP9315studes where mark >= 50;
```

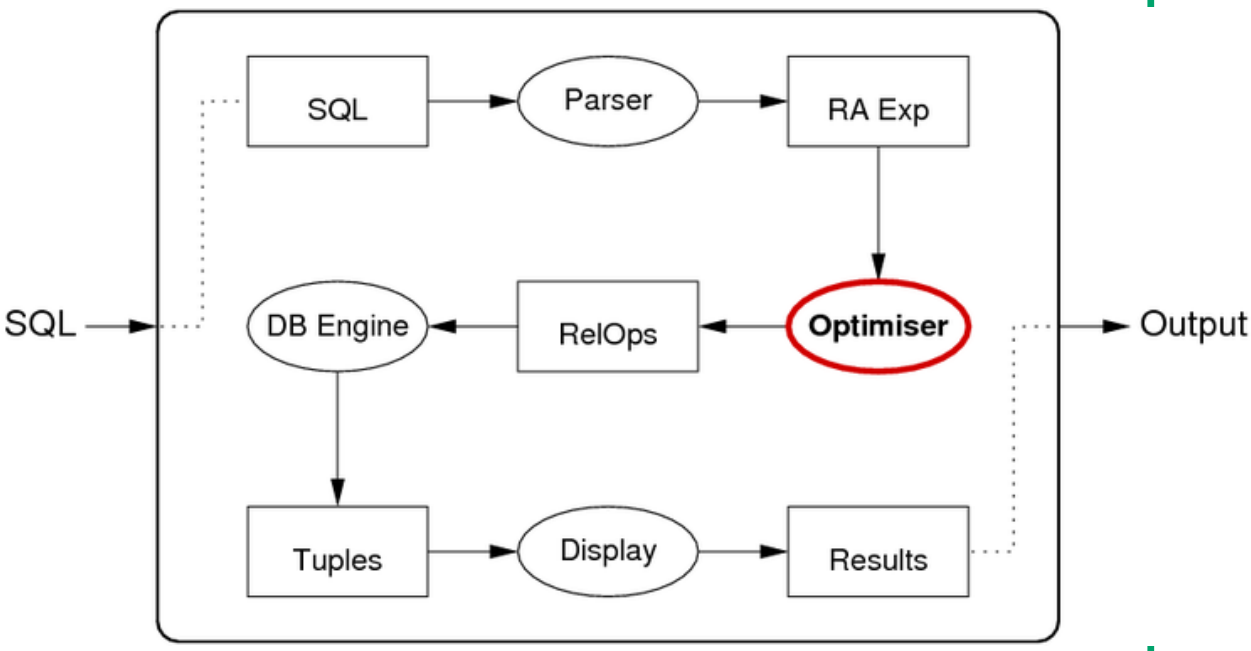
is represented in RA by

```
COMP9315studes
  = Proj[stu,mark](Sel[course=COMP9315](Enrolments))
-- with query ...
Proj[stu](Sel[mark>=50](COMP9315studes))
-- becomes ...
Proj[stu](Sel[mark>=50](
  Proj[stu,mark](Sel[course=COMP9315](Enrolments))))
-- which could be rewritten as ...
Proj[stu](Sel[mark>=50 & course=COMP9315]Enrolments)
```

Query Optimisation

❖ Query Optimisation

Query optimiser: RA expression → efficient evaluation plan



❖ Query Optimisation (cont)

Query optimisation is a critical step in query evaluation.

The query optimiser

- takes relational algebra expression from SQL compiler
- produces sequence of RelOps to evaluate the expression
- query execution plan should provide efficient evaluation

"Optimisation" is a misnomer since query optimisers

- aim to find a good plan ... but maybe not optimal

Observed Query Time = Planning time + Evaluation time

❖ Query Optimisation (cont)

Why do we not generate optimal query execution plans?

Finding an optimal query plan ...

- requires exhaustive search of a space of possible plans
- for each possible plan, need to estimate cost (not cheap)

Even for relatively small query, search space is *very large*.

Compromise:

- do limited search of query plan space (guided by heuristics)
- *quickly* choose a *reasonably efficient* execution plan

❖ Approaches to Optimisation

Three main classes of techniques developed:

- algebraic (equivalences, rewriting, heuristics)
- physical (execution costs, search-based)
- semantic (application properties, heuristics)

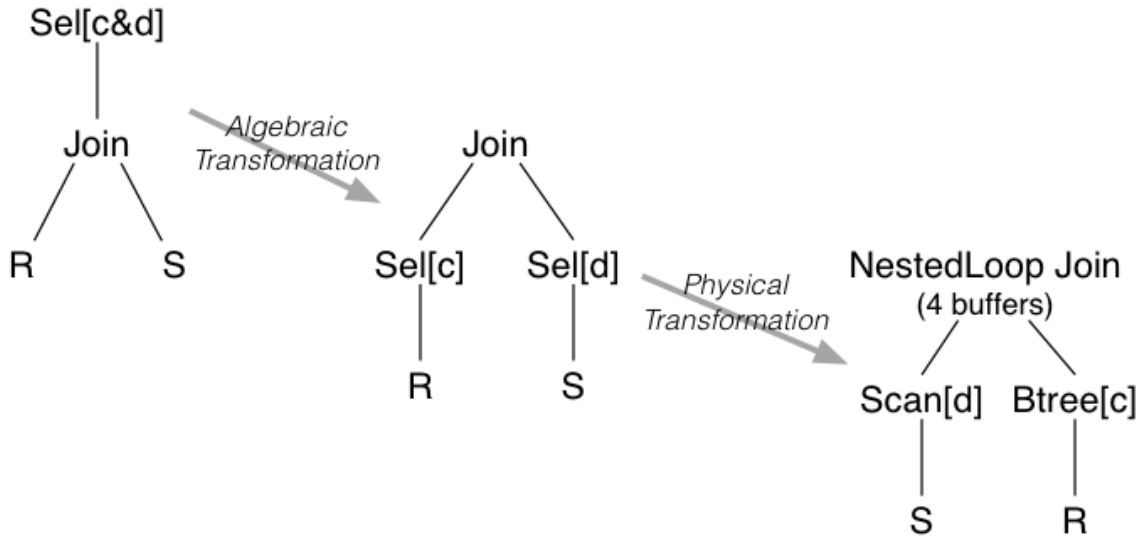
All driven by aim of minimising (or at least reducing) "cost".

Real query optimisers use a combination of algrebraic+physical.

Semantic QO is good idea, but expensive/difficult to implement.

❖ Approaches to Optimisation (cont)

Example of optimisation transformations:



For join, may also consider sort/merge join and hash join.

❖ Cost-based Query Optimiser

Approximate algorithm for cost-based optimisation:

```
translate SQL query to RAexp
for enough transformations RA' of RAexp {
   while (more choices for RelOps) {
      Plan = {};  i = 0;  cost = 0
      for each node e of RA' (recursively) {
         ROp = select RelOp method for e
         Plan = Plan ∪ ROp
         cost += Cost(ROp) // using child info
      }
      if (cost < MinCost)
         { MinCost = cost;  BestPlan = Plan }
   }
}
```

Heuristics: push selections down, consider only left-deep join trees.

❖ Cost Models and Analysis

The cost of evaluating a query is determined by:

- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves *estimating*:

- size of intermediate results
- number of disk reads/writes

❖ Choosing Access Methods (RelOps)

Performed for each node in RA expression tree ...

Inputs:

- a single RA operation (σ, π, $\bowtie$)
- information about file organisation, data distribution, ...
- list of operations available in the database engine

Output:

- specific DBMS operation to implement this RA operation

❖ Choosing Access Methods (RelOps) (cont)

Example:

- RA operation: $Sel_{[name='John' \wedge age>21]}(Student)$
- **`Student`** relation has B-tree index on **`name`**
- database engine (obviously) has B-tree search method

giving

```
tmp[i]   := BtreeSearch[name='John'](Student)
tmp[i+1] := LinearSearch[age>21](tmp[i])
```

Where possible, use pipelining to avoid storing **`tmp[i]`** on disk.

❖ Choosing Access Methods (RelOps) (cont)

Rules for choosing σ access methods:

- $\sigma_{A=c}(R)$ and **R** has index on **A** ⇒ **indexSearch[A=c](R)**
- $\sigma_{A=c}(R)$ and **R** is hashed on **A** ⇒ **hashSearch[A=c](R)**
- $\sigma_{A=c}(R)$ and **R** is sorted on **A** ⇒ **binarySearch[A=c](R)**
- $\sigma_{A \geq c}(R)$ and **R** has clustered index on **A** ⇒ **indexSearch[A=c](R)** then scan
- $\sigma_{A \geq c}(R)$ and **R** is hashed on **A** ⇒ **linearSearch[A>=c](R)**

❖ Choosing Access Methods (RelOps) (cont)

Rules for choosing ⋈ access methods:

- *R* ⋈ *S* and **R** fits in memory buffers ⇒ **bnlJoin(R,S)**
- *R* ⋈ *S* and **S** fits in memory buffers ⇒ **bnlJoin(S,R)**
- *R* ⋈ *S* and **R**,**S** sorted on join attr ⇒ **smJoin(R,S)**
- *R* ⋈ *S* and **R** has index on join attr ⇒ **inlJoin(S,R)**
- *R* ⋈ *S* and no indexes, no sorting ⇒ **hashJoin(R,S)**

(**bnl** = block nested loop; **inl** = index nested loop; **sm** = sort merge)

❖ PostgreSQL Query Optimization

Input: tree of **`Query`** nodes returned by parser

Output: tree of **`Plan`** nodes used by query executor

- wrapped in a **`PlannedStmt`** node containing state info

Intermediate data structures are trees of **`Path`** nodes

- a path tree represents one evaluation order for a query

All **`Node`** types are defined in **`include/nodes/*.h`**

❖ PostgreSQL Query Optimization (cont)

Query optimisation proceeds in two stages (after parsing)...

Rewriting:

- uses PostgreSQL's rule system
- query tree is expanded to include e.g. view definitions

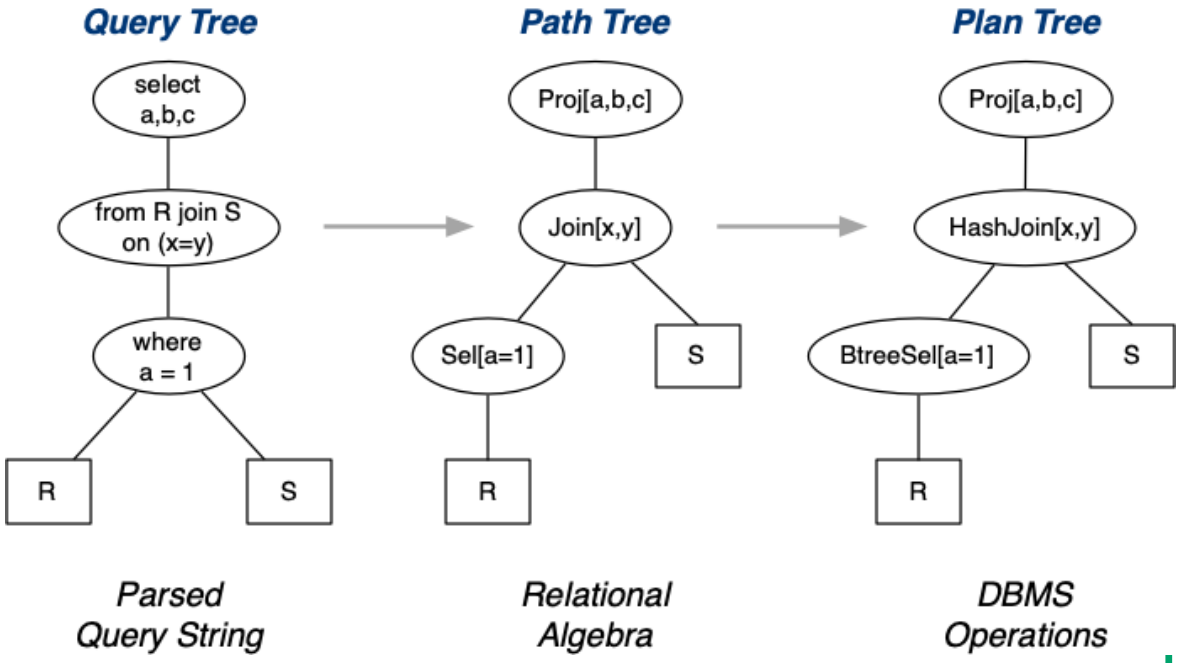
Planning and optimisation:

- using cost-based analysis of generated paths
- via one of two different path generators
- chooses least-cost path from all those considered

Then produces a `Plan` tree from the selected path.

❖ PostgreSQL Query Optimization (cont)

```
select a,b,c from R join S on (x=y) where a = 1
```



Query Cost Estimation

❖ Query Cost Estimation

Without executing a plan, cannot always know its precise cost.

Thus, query optimisers estimate costs via:

- cost of performing operation (dealt with in earlier lectures)
- size of result (which affects cost of performing next operation)

Result size estimated by statistical measures on relations, e.g.

r_S	cardinality of relation *S*
R_S	avg size of tuple in relation *S*
V(A,S)	# distinct values of attribute *A*
min(A,S)	min value of attribute *A*
max(A,S)	max value of attribute *A*

❖ Estimating Projection Result Size

Straightforward, since we know:

- number of tuples in output

  $r_{out} = |\ \Pi_{a,b,..}(T)\ | = |\ T\ | = r_T$ (in SQL, because of bag semantics)

- size of tuples in output

  R_{out} = sizeof(*a*) + sizeof(*b*) + ... + tuple-overhead

Assume page size *B*, $b_{out} = ceil(r_T / c_{out})$, where $c_{out} = floor(B/R_{out})$

If using **`select distinct`** ...

- $|\ \Pi_{a,b,..}(T)\ |$ depends on proportion of duplicates produced

❖ Estimating Selection Result Size

Selectivity = fraction of tuples expected to satisfy a condition.

Common assumption: attribute values uniformly distributed.

Example: Consider the query

```
select * from Parts where colour='Red'
```

If $V(colour,Parts)=4$, $r=1000 \Rightarrow |\sigma_{colour=red}(Parts)|=250$

In general, $|\sigma_{A=c}(R)| \cong r_R / V(A,R)$

Heuristic used by PostgreSQL: $|\sigma_{A=c}(R)| \cong r/10$

❖ Estimating Selection Result Size (cont)

Estimating size of result for e.g.

```
select * from Enrolment where year > 2015;
```

Could estimate by using:

- uniform distribution assumption, *r*, min/max years

Assume: min(year)=2010, max(year)=2019, $|Enrolment| = 10^5$

- 10^5 from 2010-2019 means approx 10000 enrolments/year
- this suggests 40000 enrolments since 2016

Heuristic used by some systems: $|\sigma_{A>c}(R)| \cong r/3$

❖ Estimating Selection Result Size (cont)

Estimating size of result for e.g.

```
select * from Enrolment where course <> 'COMP9315';
```

Could estimate by using:

- uniform distribution assumption, *r*, domain size

e.g. | *V(course,Enrolment)* | = 2000, | $\sigma_{A<>c}(E)$ | = *r* * *1999/2000*

Heuristic used by some systems: | $\sigma_{A<>c}(R)$ | ≅ *r*

❖ Estimating Selection Result Size (cont)

How to handle non-uniform attribute value distributions?

- collect statistics about the values stored in the attribute/relation
- store these as e.g. a histogram in the meta-data for the relation

So, for part colour example, might have distribution like:

`White`: 35% **`Red`**: 30% **`Blue`**: 25% **`Silver`**: 10%

Use histogram as basis for determining # selected tuples.

Disadvantage: cost of storing/maintaining histograms.

❖ Estimating Selection Result Size (cont)

Summary: analysis relies on operation and data distribution:

E.g. `select * from R where a = k;`

Case 1: *uniq(R.a)* ⇒ 0 or 1 result

Case 2: r_R tuples && *size(dom(R.a))* = *n* ⇒ r_R / *n* results

E.g. `select * from R where a < k;`

Case 1: *k* ≤ *min(R.a)* ⇒ 0 results

Case 2: *k* > *max(R.a)* ⇒ ≅ r_R results

Case 3: *size(dom(R.a))* = *n* ⇒ ? *min(R.a)* ... *k* ... *max(R.a)* ?

❖ Estimating Join Result Size

Analysis relies on semantic knowledge about data/relations.

Consider equijoin on common attr: $R \bowtie_a S$

Case 1: *values(R.a)* $\cap$ *values(S.a)* = {} $\Rightarrow$ *size*($R \bowtie_a S$) = *0*

Case 2: *uniq(R.a)* and *uniq(S.a)* $\Rightarrow$ *size*($R \bowtie_a S$) $\leq$ *min*(|*R*|, |*S*|)

Case 3: *pkey(R.a)* and *fkey(S.a)* $\Rightarrow$ *size*($R \bowtie_a S$) $\leq$ |*S*|

❖ Cost Estimation: Postscript

Inaccurate cost estimation can lead to poor evaluation plans.

Above methods can (sometimes) give inaccurate estimates.

To get more accurate cost estimates:

- more time ... complex computation of selectivity
- more space ... storage for histograms of data values

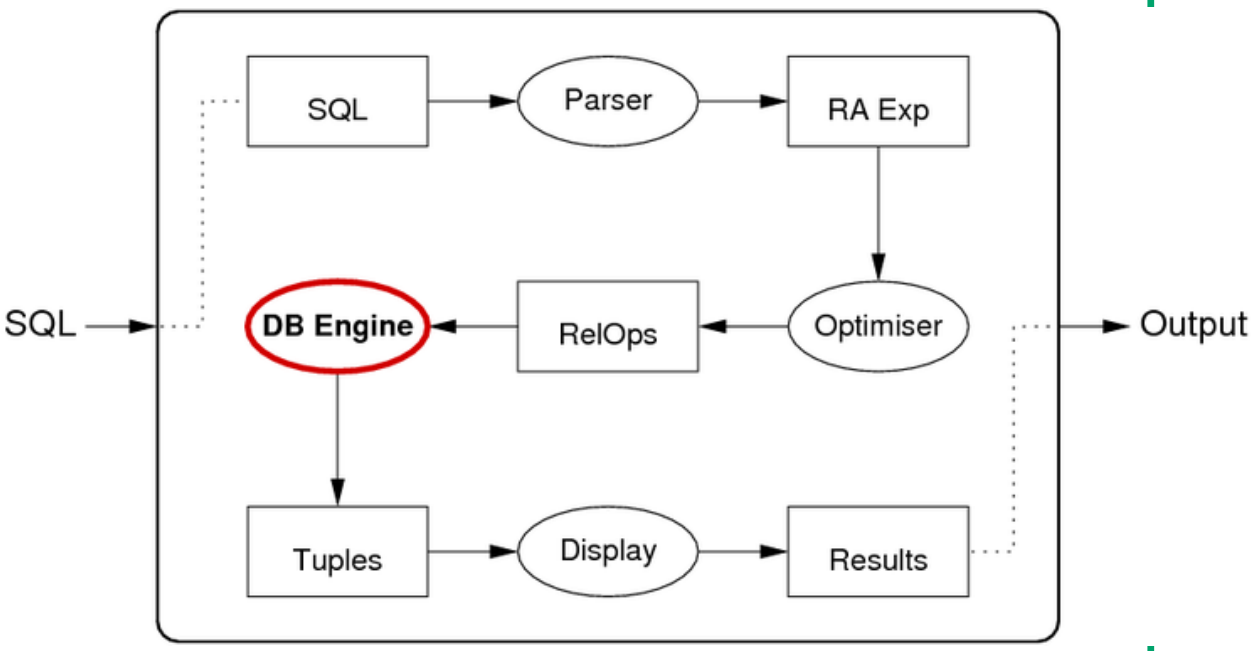
Either way, optimisation process costs more (more than query?)

Trade-off between optimiser performance and query performance.

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

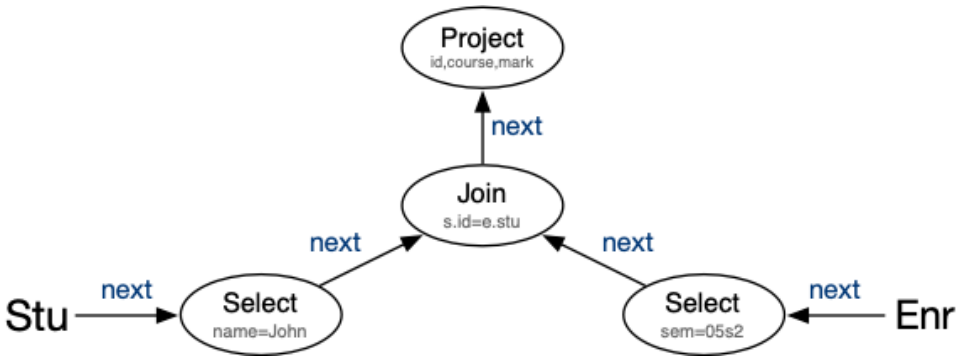
Other possible operations: reset to specific point, restart, ...

❖ 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)

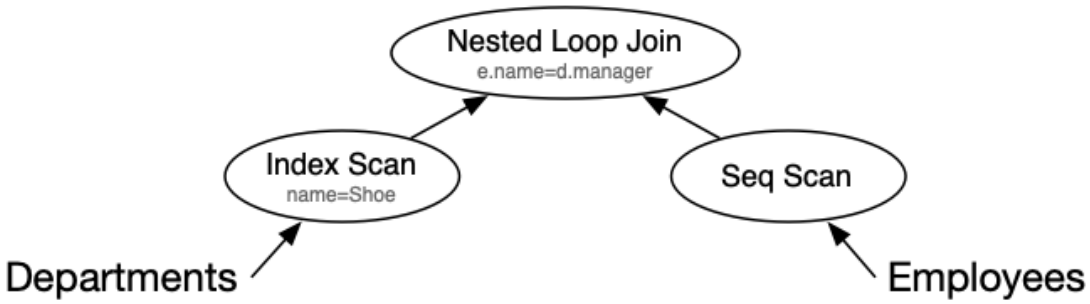
- implement iterators for specific types of RA operators
- typically contains **`ExecInitXXX`**, **`ExecXXX`**, **`ExecEndXXX`**

❖ 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)

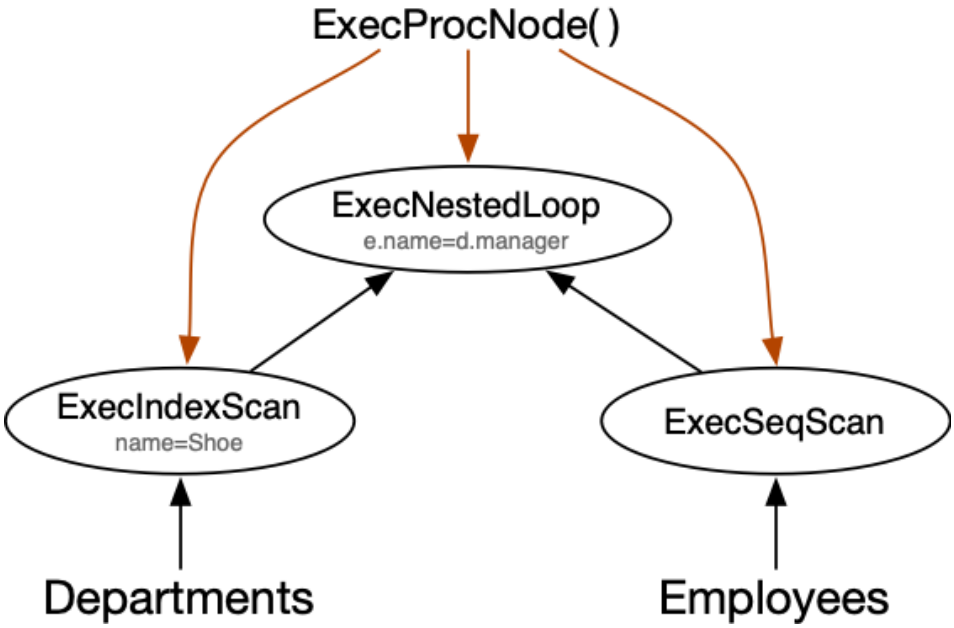
Initially **`InitPlan()`** invokes **`ExecInitNode()`** on plan tree root.

`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()`**

Query Performance Tuning

- Query Performance Tuning
- PostgreSQL Query Tuning
- EXPLAIN examples

❖ Query Performance Tuning

What to do if the DBMS takes "too long" to answer some queries?

Improving performance may involve any/all of:

- making applications using the DB run faster
- lowering response time of queries/transactions
- improving overall transaction throughput

Remembering that, to some extent ...

- the query optimiser removes choices from DB developers
- by making its own decision on the optimal execution plan

❖ Query Performance Tuning (cont)

Tuning requires us to consider the following:

- which queries and transactions will be used?
  (e.g. check balance for payment, display recent transaction history)
- how frequently does each query/transaction occur?
  (e.g. 80% withdrawals; 1% deposits; 19% balance check)
- are there time constraints on queries/transactions?
  (e.g. EFTPOS payments must be approved within 7 seconds)
- are there uniqueness constraints on any attributes?
  (define indexes on attributes to speed up insertion uniqueness check)
- how frequently do updates occur?
  (indexes slow down updates, because must update table *and* index)

❖ Query Performance Tuning (cont)

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

Difficult to predict what query optimiser will do, so ...

- implement queries using methods which *should* be efficient
- observe execution behaviour and modify query accordingly

❖ PostgreSQL Query Tuning

PostgreSQL provides the **`explain`** statement to

- give a representation of the query execution plan
- with information that may help to tune query performance

Usage:

```
EXPLAIN [ANALYZE] Query
```

Without **`ANALYZE`**, **`EXPLAIN`** shows plan with estimated costs.

With **`ANALYZE`**, **`EXPLAIN`** executes query and prints real costs.

Note that runtimes may show considerable variation due to buffering.

❖ EXPLAIN examples

Using the following database ...

```
CourseEnrolments(student, course, mark, grade, ...)
Courses(id, subject, semester, homepage)
People(id, family, given, title, name, ..., birthday)
ProgramEnrolments(id, student, semester, program, wam, ...)
Students(id, stype)
Subjects(id, code, name, longname, uoc, offeredby, ...)
```

with a view defined as

```
create view EnrolmentCounts as
 select s.code, c.semester, count(e.student) as nstudes
   from Courses c join Subjects s on c.subject=s.id
        join Course_enrolments e on e.course = c.id
  group by s.code, c.semester;
```

❖ EXPLAIN examples (cont)

Some database statistics:

```
         tab_name          | n_records
---------------------------+-----------
 courseenrolments          |    503120
 courses                   |     71288
 people                    |     36497
 programenrolments         |    161110
 students                  |     31048
 subjects                  |     18799
```

❖ EXPLAIN examples (cont)

Example: Select on non-indexed attribute

```
uni=# explain
uni=# select * from Students where stype='local';
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on students
             (cost=0.00..562.01 rows=23544 width=9)
   Filter: ((stype)::text = 'local'::text)
```

where

- **`Seq Scan`** = operation (plan node)
- **`cost`**=*StartUpCost*`..`*TotalCost*
- **`rows`**=*NumberOfResultTuples*
- **`width`**=*SizeOfTuple* (# bytes)

❖ EXPLAIN examples (cont)

More notes on **`explain`** output:

- each major entry corresponds to a plan node
  - e.g. **`Seq Scan`**, **`Index Scan`**, **`Hash Join`**, **`Merge Join`**, ...
- some nodes include additional qualifying information
  - e.g. **`Filter`**, **`Index Cond`**, **`Hash Cond`**, **`Buckets`**, ...
- **`cost`** values in **`explain`** are estimates (notional units)
- **`explain analyze`** also includes actual time costs (ms)
- costs of parent nodes include costs of all children
- estimates of #results based on sample of data

❖ EXPLAIN examples (cont)

Example: Select on non-indexed attribute with actual costs

```
uni=# explain analyze
uni=# select * from Students where stype='local';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on students
             (cost=0.00..562.01 rows=23544 width=9)
             (actual time=0.052..5.792 rows=23551 loops=1)
   Filter: ((stype)::text = 'local'::text)
   Rows Removed by Filter: 7810
 Planning time: 0.075 ms
 Execution time: 6.978 ms
```

❖ EXPLAIN examples (cont)

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=100250;
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using student_pkey on student
            (cost=0.00..8.27 rows=1 width=9)
            (actual time=0.049..0.049 rows=0 loops=1)
   Index Cond: (id = 100250)
 Planning Time: 0.274 ms
 Execution Time: 0.109 ms
```

❖ EXPLAIN examples (cont)

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=1216988;
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using students_pkey on students
            (cost=0.29..8.30 rows=1 width=9)
            (actual time=0.011..0.012 rows=1 loops=1)
   Index Cond: (id = 1216988)
 Planning time: 0.273 ms
 Execution time: 0.115 ms
```

❖ EXPLAIN examples (cont)

Example: Join on a primary key (indexed) attribute (2016)

```
uni=# explain analyze
uni-# select s.id,p.name
uni-# from Students s, People p where s.id=p.id;
                        QUERY PLAN
-----------------------------------------------------------
Hash Join (cost=988.58..3112.76 rows=31048 width=19)
          (actual time=11.504..39.478 rows=31048 loops=1)
  Hash Cond: (p.id = s.id)
  -> Seq Scan on people p
         (cost=0.00..989.97 rows=36497 width=19)
         (actual time=0.016..8.312 rows=36497 loops=1)
  -> Hash (cost=478.48..478.48 rows=31048 width=4)
          (actual time=10.532..10.532 rows=31048 loops=1)
          Buckets: 4096  Batches: 2  Memory Usage: 548kB
      ->  Seq Scan on students s
              (cost=0.00..478.48 rows=31048 width=4)
              (actual time=0.005..4.630 rows=31048 loops=1)
 Planning Time: 0.691 ms
 Execution Time: 44.842 ms
```

❖ EXPLAIN examples (cont)

Example: Join on a primary key (indexed) attribute (2018)

```
uni=# explain analyze
uni-# select s.id,p.name
uni-# from Students s, People p where s.id=p.id;
                        QUERY PLAN
-----------------------------------------------------------
Merge Join  (cost=0.58..2829.25 rows=31361 width=18)
            (actual time=0.044..25.883 rows=31361 loops=1)
  Merge Cond: (s.id = p.id)
  ->  Index Only Scan using students_pkey on students s
            (cost=0.29..995.70 rows=31361 width=4)
            (actual time=0.033..6.195 rows=31361 loops=1)
        Heap Fetches: 31361
  ->  Index Scan using people_pkey on people p
            (cost=0.29..2434.49 rows=55767 width=18)
            (actual time=0.006..6.662 rows=31361 loops=1)
Planning time: 0.259 ms
Execution time: 27.327 ms
```