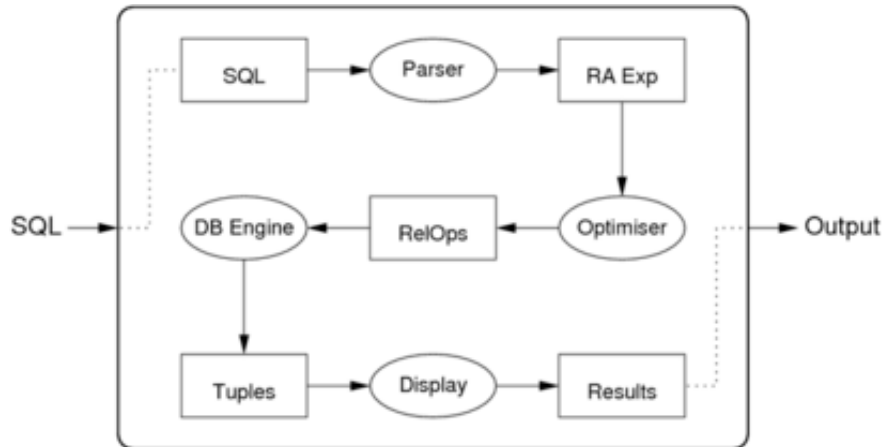


# Week 08 Lectures

## Query Processing So Far

1/79



### ... Query Processing So Far

2/79

Steps in processing an SQL statement

- parse, map to relation algebra (RA) expression
- transform to more efficient RA expression
- instantiate RA operators to DBMS operations
- execute DBMS operations (aka query plan)

4-3-5-2

Cost-based optimisation:

- generate possible query plans (via rewriting/heuristics)
- estimate cost of each plan (using costs of operations)
- choose the lowest-cost plan (... and choose quickly)

## Expression Rewriting Rules

3/79

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

不是 RA

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

4/79

Commutative and Associative Laws:

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

5/79

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

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

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

6/79

Rewrite rules for projection ...

All but last projection can be ignored:

- $\pi_{L1}(\pi_{L2}(\dots \pi_{Ln}(R))) \rightarrow \pi_{L1}(R)$

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  $L$  ( $L \subset M \cup N$ )

### ... Relational Algebra Laws

7/79

Subqueries  $\Rightarrow$  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
```

### ... Relational Algebra Laws

8/79

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

$Val = \max[id]Students$

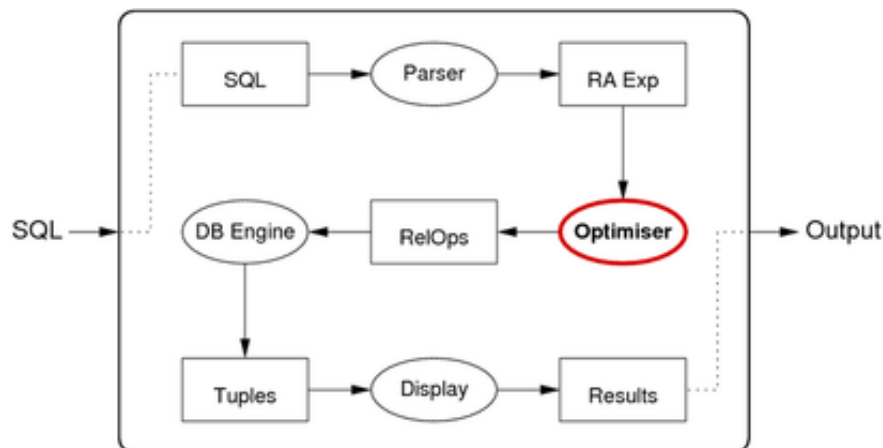
$Res = \pi_{(sid,cid)}(\sigma_{sid=Val}Enrolments)$

## Query Optimisation

### Query Optimisation

10/79

Query optimiser: RA expression  $\rightarrow$  efficient evaluation plan



### ... Query Optimisation

11/79

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

12/79

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

13/79

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".

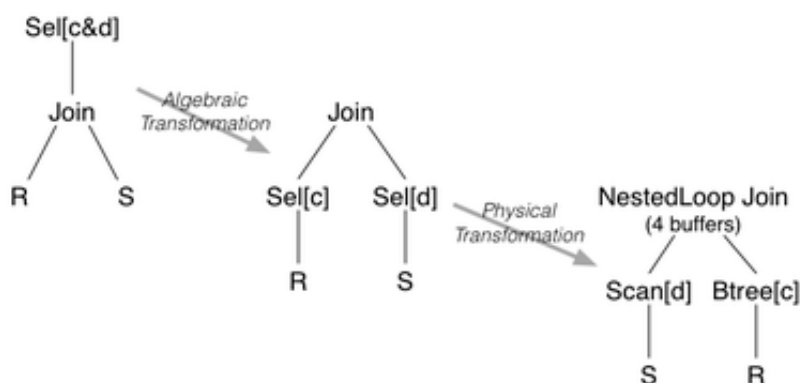
Real query optimisers use a combination of algebraic+physical.

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

### ... Approaches to Optimisation

14/79

Example of optimisation transformations:



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

## Cost-based Query Optimiser

15/79

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 U ROp
      cost += Cost(ROp) // using child info
    }
    if (cost < MinCost)
      { MinCost = cost; BestPlan = Plan }
  }
}

```

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

## Exercise 1: Alternative Join Plans

16/79

Consider the schema

```

Students(id,name,...)   Enrol(student,course,mark)
Staff(id,name,...)      Courses(id,code,term,lic,...)

```

the following query on this schema

```

select c.code, s.id, s.name
from   Students s, Enrol e, Courses c, Staff f
where  s.id=e.student and e.course=c.id
      and c.lic=f.id and c.term='11s2'
      and f.name='John Shepherd'

```

Show some possible evaluation orders for this query.

## Cost Models and Analysis

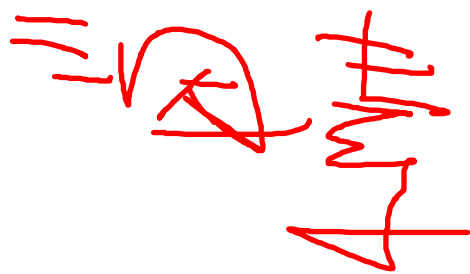
17/79

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 secondary storage accesses



## Choosing Access Methods (RelOps)

18/79

Performed for each node in RA expression tree ...

Inputs:

- a single RA operation ( $\sigma$ ,  $\pi$ ,  $\bowtie$ )
- 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)

19/79

#### 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)

20/79

Rules for choosing  $\sigma$  access methods:

几种不同 access

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

针对不同的operation  
用不同的 access

### ... Choosing Access Methods (RelOps)

21/79

Rules for choosing  $\bowtie$  access methods:

- $R \bowtie S$  and R fits in memory buffers  $\Rightarrow$  bnlJoin(R,S)
- $R \bowtie S$  and S fits in memory buffers  $\Rightarrow$  bnlJoin(S,R)
- $R \bowtie S$  and R,S sorted on join attr  $\Rightarrow$  smJoin(R,S)
- $R \bowtie S$  and R has index on join attr  $\Rightarrow$  inlJoin(S,R)
- $R \bowtie S$  and no indexes, no sorting  $\Rightarrow$  hashJoin(R,S)

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

## Cost Estimation

22/79

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

23/79

Straightforward, since we know:

- number of tuples in output

$$r_{out} = |\pi_{a,b,\dots}(T)| = |T| = r_T \quad (\text{in SQL, because of bag semantics})$$

- size of tuples in output

$$R_{out} = \text{sizeof}(a) + \text{sizeof}(b) + \dots + \text{tuple-overhead}$$

Assume page size  $B$ ,  $b_{out} = \lceil r_T / c_{out} \rceil$ , where  $c_{out} = \lfloor B/R_{out} \rfloor$

If using `select distinct ...`

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

## Estimating Selection Result Size

24/79

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

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

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

### ... Estimating Selection Result Size

25/79

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:  $\min(\text{year})=2010$ ,  $\max(\text{year})=2019$ ,  $|Enrolment|=10^5$

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

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

### ... Estimating Selection Result Size

26/79

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

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

## Exercise 2: Selection Size Estimation

27/79

Assuming that

- all attributes have uniform distribution of data values
- attributes are independent of each other

Give formulae for the number of expected results for

1. `select * from R where not A=k`
2. `select * from R where A=k and B=j`
3. `select * from R where A in (k,l,m,n)`

where  $j, k, l, m, n$  are constants.

Assume:  $V(A,R) = 10$  and  $V(B,R)=100$  and  $r=1000$

如果不是 uniform  
distribute

### ... Estimating Selection Result Size

28/79

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

29/79

Summary: analysis relies on operation and data distribution:

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

Case 1:  $\text{uniq}(R.a) \Rightarrow 0 \text{ or } 1 \text{ result}$

Case 2:  $r_R \text{ tuples \&\& } \text{size}(\text{dom}(R.a)) = n \Rightarrow r_R / n \text{ results}$

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

Case 1:  $k \leq \min(R.a) \Rightarrow 0 \text{ results}$

Case 2:  $k > \max(R.a) \Rightarrow r_R \text{ results}$

Case 3:  $\text{size}(\text{dom}(R.a)) = n \Rightarrow ? \min(R.a) \dots k \dots \max(R.a) ?$

## Estimating Join Result Size

30/79

Analysis relies on semantic knowledge about data/relations.

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

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

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

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

## Exercise 3: Join Size Estimation

31/79

How many tuples are in the output from:

- `select * from R, S where R.s = S.id`  
where `S.id` is a primary key and `R.s` is a foreign key referencing `S.id`
- `select * from R, S where R.s <> S.id`  
where `S.id` is a primary key and `R.s` is a foreign key referencing `S.id`
- `select * from R, S where R.x = S.y`  
where `R.x` and `S.y` have no connection except that  $\text{dom}(R.x) = \text{dom}(S.y)$

Under what conditions will the first query have maximum size?

## Cost Estimation: Postscript

32/79

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.

## PostgreSQL Query Optimiser

### Overview of QOpt Process

34/79

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

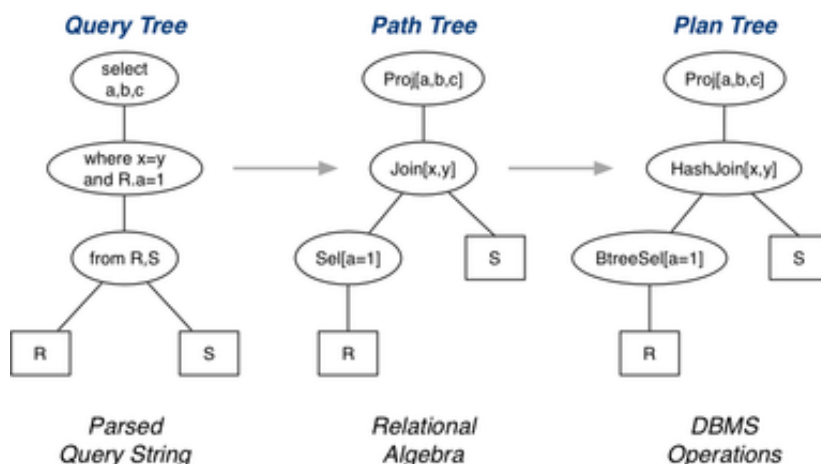
Intermediate data structures are trees of **Path** nodes

- a path tree represents one evaluation order for a query

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

### ... Overview of QOpt Process

35/79



### QOpt Data Structures

36/79

Generic **Path** node structure:

```
typedef struct Path
{
    NodeTag    type;           // scan/join/...
    NodeTag    pathtype;       // specific method
    RelOptInfo *parent;        // output relation
    PathTarget *pathtarget;    // list of Vars/Exprs, width
    // estimated execution costs for path ...
    Cost       startup_cost;    // setup cost
    Cost       total_cost;      // total cost
    List       *pathkeys;       // sort order
} Path;
```

PathKey = (opfamily:Oid, strategy:SortDir, nulls\_first:bool)

## ... QOpt Data Structures

37/79

Specialised **Path** nodes (simplified):

```
typedef struct IndexPath
{
    Path    path;
    IndexOptInfo *indexinfo; // index for scan
    List    *indexclauses; // index select conditions
    ...
    ScanDirection indexscandir; // used by planner
    Selectivity indexselectivity; // estimated #results
} IndexPath;

typedef struct JoinPath
{
    Path    path;
    JoinType jointype; // inner/outer/semi/anti
    Path    *outerpath; // outer part of the join
    Path    *innerpath; // inner part of the join
    List    *restrictinfo; // join condition(s)
} JoinPath;
```

## Query Optimisation Process

38/79

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.

## Top-down Trace of QOpt

39/79

Top-level of query execution: **backend/tcop/postgres.c**

```
exec_simple_query(const char *query_string)
{
    // lots of setting up ... including starting xact
    parsetree_list = pg_parse_query(query_string);
    foreach(parsetree, parsetree_list) {
        // Query optimisation
        querytree_list = pg_analyze_and_rewrite(parsetree,...);
        plantree_list = pg_plan_queries(querytree_list,...);
        // Query execution
        portal = CreatePortal(...plantree_list...);
        PortalRun(portal,...);
    }
}
```

```
// lots of cleaning up ... including close xact
}
```

Assumes that we are dealing with multiple queries (i.e. SQL statements)

### ... Top-down Trace of QOpt

40/79

**query\_planner()** produces plan for a select/join tree

- make list of tables used in query
- split **where** qualifiers ("quals") into
  - restrictions (e.g. `r.a=1`) ... for selections
  - joins (e.g. `s.id=r.s`) ... for joins
- search for quals to enable merge/hash joins
- invoke **make\_one\_rel()** to find best path/plan

Code in: **backend/optimizer/plan/planmain.c**

### ... Top-down Trace of QOpt

41/79

**make\_one\_rel()** generates possible plans, selects best

- generate scan and index paths for base tables
  - using of restrictions list generated above
- generate access paths for the entire join tree
  - recursive process, controlled by **make\_rel\_from\_joinlist()**
- returns a single "relation", representing result set

Code in: **backend/optimizer/path/allpaths.c**

## Join-tree Generation

42/79

**make\_rel\_from\_joinlist()** arranges path generation

- switches between two possible path tree generators
- path tree generators finally return best cost path

Standard path tree generator (**standard\_join\_search()**):

- "exhaustively" generates join trees (like System R)
- starts with 2-way joins, finds best combination
- then adds extra table to give 3-table join, etc.

Code in: **backend/optimizer/path/{allpaths.c,joinrels.c}**

### ... Join-tree Generation

43/79

Genetic query optimiser (**geqo**):

- uses genetic algorithm (GA) to generate path trees
- handles joins involving `> geqo_threshold` relations
- goals of this approach:
  - find near-optimal solution
  - examine far less than entire search space

Code in: **backend/optimizer/geqo/\*.c**

## ... Join-tree Generation

44/79

Basic idea of genetic algorithm:

```
Optimize(join)
{
    t = 0
    p = initialState(join) // pool of (random) join orders
    for (t = 0; t < #generations; t++) {
        p' = recombination(p) // get parts of good join orders
        p'' = mutation(p') // generate new variations
        p = selection(p'', p) // choose best join orders
    }
}
```

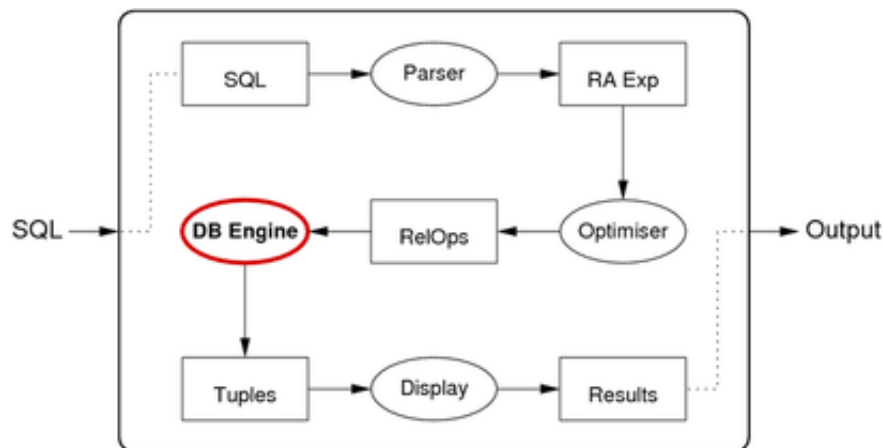
#generations determined by size of initial pool of join orders

## Query Execution

46/79

### Query Execution

Query execution: applies evaluation plan → result tuples



## ... Query Execution

47/79

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

48/79

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

49/79

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

50/79

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)

51/79

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

52/79

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';
```

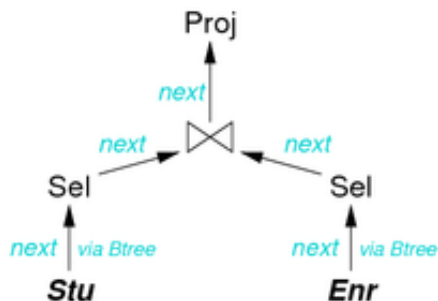
which maps to the RA expression

$$Proj_{[id, course, mark]}(Join_{[student=id]}(Sel_{[05s2]}(Enr), Sel_{[John]}(Stu)))$$

## ... Pipelining Example

53/79

Evaluated via communication between RA tree nodes:



Note: likely that projection is combined with join in PostgreSQL

## Disk Accesses

54/79

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

## PostgreSQL Query Execution

56/79

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

Code: **src/backend/executor**

PostgreSQL uses pipelining ...

- 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

57/79

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**

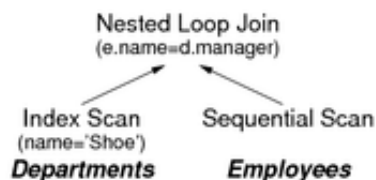
## Example PostgreSQL Execution

58/79

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'
```

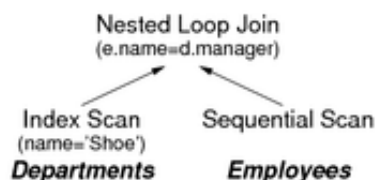
and its execution plan tree



## ... Example PostgreSQL Execution

59/79

The execution plan tree



contains three nodes:

- `NestedLoop` with join condition (`Outer.manager = Inner.name`)
- `IndexScan` on `Departments` with selection (`name = 'Shoe'`)
- `SeqScan` on `Employees`



### ... Example PostgreSQL Execution

60/79

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

61/79

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

`ExecProcNode()` sees a `NestedLoop` node ...  
 so dispatches to `ExecNestLoop()` 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

### Performance Tuning

63/79

How to make a database-backed system perform "better"?

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

### ... Performance Tuning

64/79

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)

## ... Performance Tuning

65/79

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

66/79

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

67/79

Database

```
people(id, family, given, title, name, ..., birthday)
courses(id, subject, semester, homepage)
course_enrolments(student, course, mark, grade, ...)
subjects(id, code, name, longname, uoc, offeredby, ...)
...
```

where

table_name	n_records
people	150963

```

courses          |      34955
course_enrolments |     1812317
subjects         |      33377
...

```

### ... EXPLAIN Examples

68/79

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

69/79

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

70/79

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

71/79

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)
  Total runtime: 0.1 ms
```

---

### ... EXPLAIN Examples

72/79

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: 0.066 ms
  Execution time: 0.026 ms
```

---

### ... EXPLAIN Examples

73/79

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)
  Total runtime: 41.0 ms
```

---

### ... EXPLAIN Examples

74/79

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

```

---

### ... EXPLAIN Examples

75/79

Example: Join on a non-indexed attribute (2016)

```

uni=# explain analyze
uni=# select s1.code, s2.code
uni=# from Subjects s1, Subjects s2
uni=# where s1.offeredBy=s2.offeredBy;
               QUERY PLAN
-----
Merge Join (cost=4449.13..121322.06 rows=7785262 width=18)
  (actual time=29.787..2377.707 rows=8039979 loops=1)
  Merge Cond: (s1.offeredby = s2.offeredby)
-> Sort (cost=2224.57..2271.56 rows=18799 width=13)
  (actual time=14.251..18.703 rows=18570 loops=1)
  Sort Key: s1.offeredby
  Sort Method: external merge  Disk: 472kB
-> Seq Scan on subjects s1
  (cost=0.00..889.99 rows=18799 width=13)
  (actual time=0.005..4.542 rows=18799 loops=1)
-> Sort (cost=2224.57..2271.56 rows=18799 width=13)
  (actual time=15.532..1100.396 rows=8039980 loops=1)
  Sort Key: s2.offeredby
  Sort Method: external sort  Disk: 552kB
-> Seq Scan on subjects s2
  (cost=0.00..889.99 rows=18799 width=13)
  (actual time=0.002..3.579 rows=18799 loops=1)
Total runtime: 2767.1 ms

```

---

### ... EXPLAIN Examples

76/79

Example: Join on a non-indexed attribute (2018)

```

uni=# explain analyze
uni=# select s1.code, s2.code
uni=# from Subjects s1, Subjects s2
uni=# where s1.offeredBy = s2.offeredBy;
               QUERY PLAN
-----
Hash Join  (cost=1286.03..108351.87 rows=7113299 width=18)
  (actual time=8.966..903.441 rows=7328594 loops=1)
  Hash Cond: (s1.offeredby = s2.offeredby)
-> Seq Scan on subjects s1
  (cost=0.00..1063.79 rows=17779 width=13)
  (actual time=0.013..2.861 rows=17779 loops=1)
-> Hash  (cost=1063.79..1063.79 rows=17779 width=13)
  (actual time=8.667..8.667 rows=17720 loops=1)
  Buckets: 32768  Batches: 1  Memory Usage: 1087kB
-> Seq Scan on subjects s2
  (cost=0.00..1063.79 rows=17779 width=13)
  (actual time=0.009..4.677 rows=17779 loops=1)
Planning time: 0.255 ms
Execution time: 1191.023 ms

```

---

### ... EXPLAIN Examples

77/79

### Example: Join on a non-indexed attribute (2018)

```

uni=# explain analyze
uni=# select s1.code, s2.code
uni=# from Subjects s1, Subjects s2
uni=# where s1.offeredBy = s2.offeredBy and s1.code < s2.code;
          QUERY PLAN
-----
Hash Join  (cost=1286.03..126135.12 rows=2371100 width=18)
           (actual time=7.356..6806.042 rows=3655437 loops=1)
    Hash Cond: (s1.offeredby = s2.offeredby)
    Join Filter: (s1.code < s2.code)
    Rows Removed by Join Filter: 3673157
    -> Seq Scan on subjects s1
           (cost=0.00..1063.79 rows=17779 width=13)
           (actual time=0.009..4.602 rows=17779 loops=1)
    -> Hash  (cost=1063.79..1063.79 rows=17779 width=13)
           (actual time=7.301..7.301 rows=17720 loops=1)
           Buckets: 32768  Batches: 1  Memory Usage: 1087kB
           -> Seq Scan on subjects s2
                 (cost=0.00..1063.79 rows=17779 width=13)
                 (actual time=0.005..4.452 rows=17779 loops=1)
Planning time: 0.159 ms
Execution time: 6949.167 ms

```

## Exercise 4: EXPLAIN examples

78/79

Using the database described earlier ...

```

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

```

```

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;

```

predict how each of the following queries will be executed ...

Check your prediction using the EXPLAIN ANALYZE command.

1. select max(birthday) from People
2. select max(id) from People
3. select family from People order by family
4. select distinct p.id, pname  
from People s, CourseEnrolments e  
where s.id=e.student and e.grade='FL'
5. select \* from EnrolmentCounts where code='COMP9315';

Examine the effect of adding ORDER BY and DISTINCT.

Add indexes to improve the speed of slow queries.