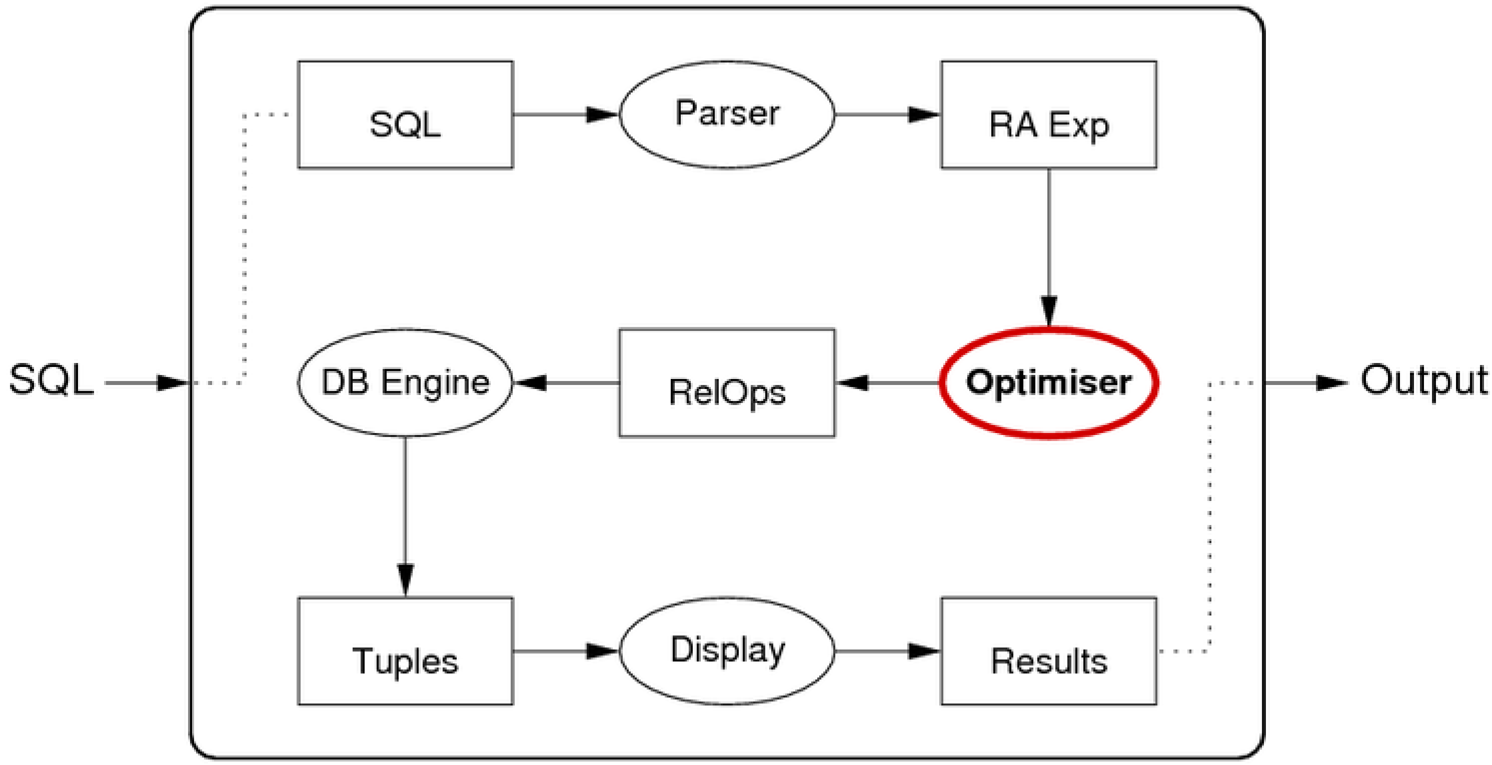
>>

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)

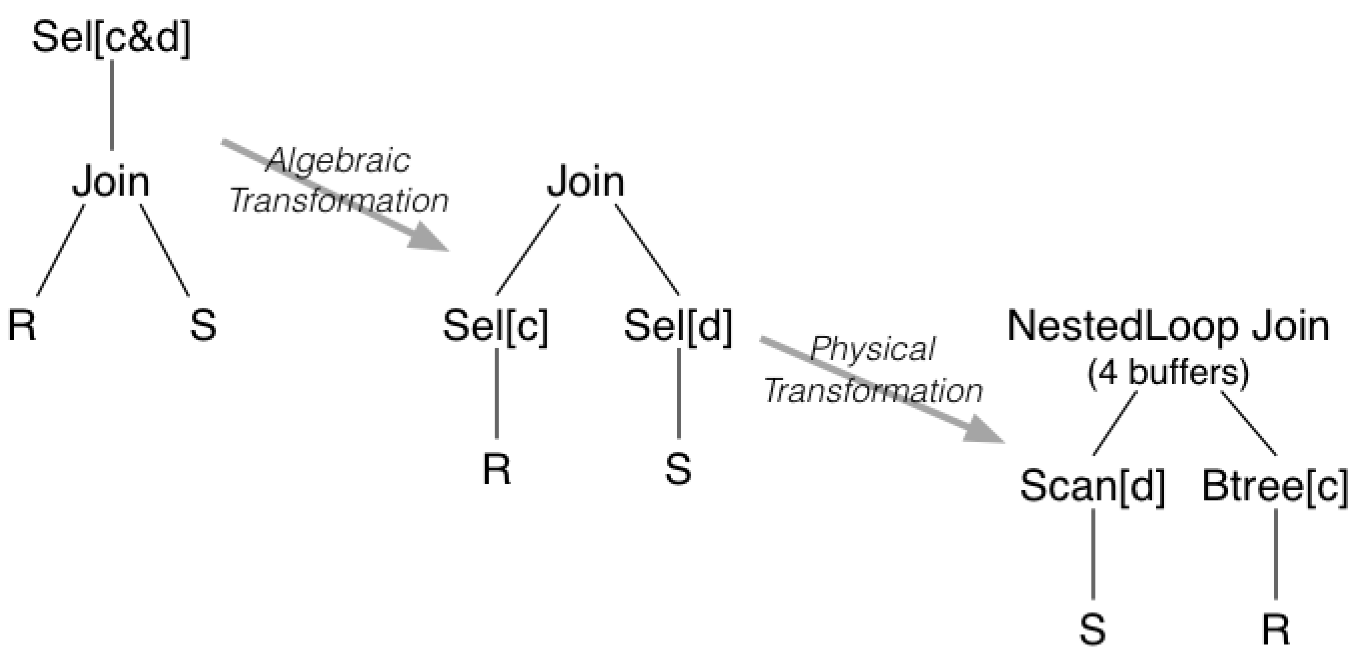
All driven by aim of minimising (or at least reducing) "cost".

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** $\Rightarrow$ **`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)`**
- $\sigma_{A \geqslant c}(R)$ and **R** has clustered index on **A**

  $\Rightarrow$ **`indexSearch[A=c](R)`** then scan
- $\sigma_{A \geqslant c}(R)$ and **R** is hashed on **A**

  $\Rightarrow$ **`linearSearch[A>=c](R)`**

<< ^ >>

❖ Choosing Access Methods (RelOps) (cont)

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)

<< ∧ >>

❖ 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:

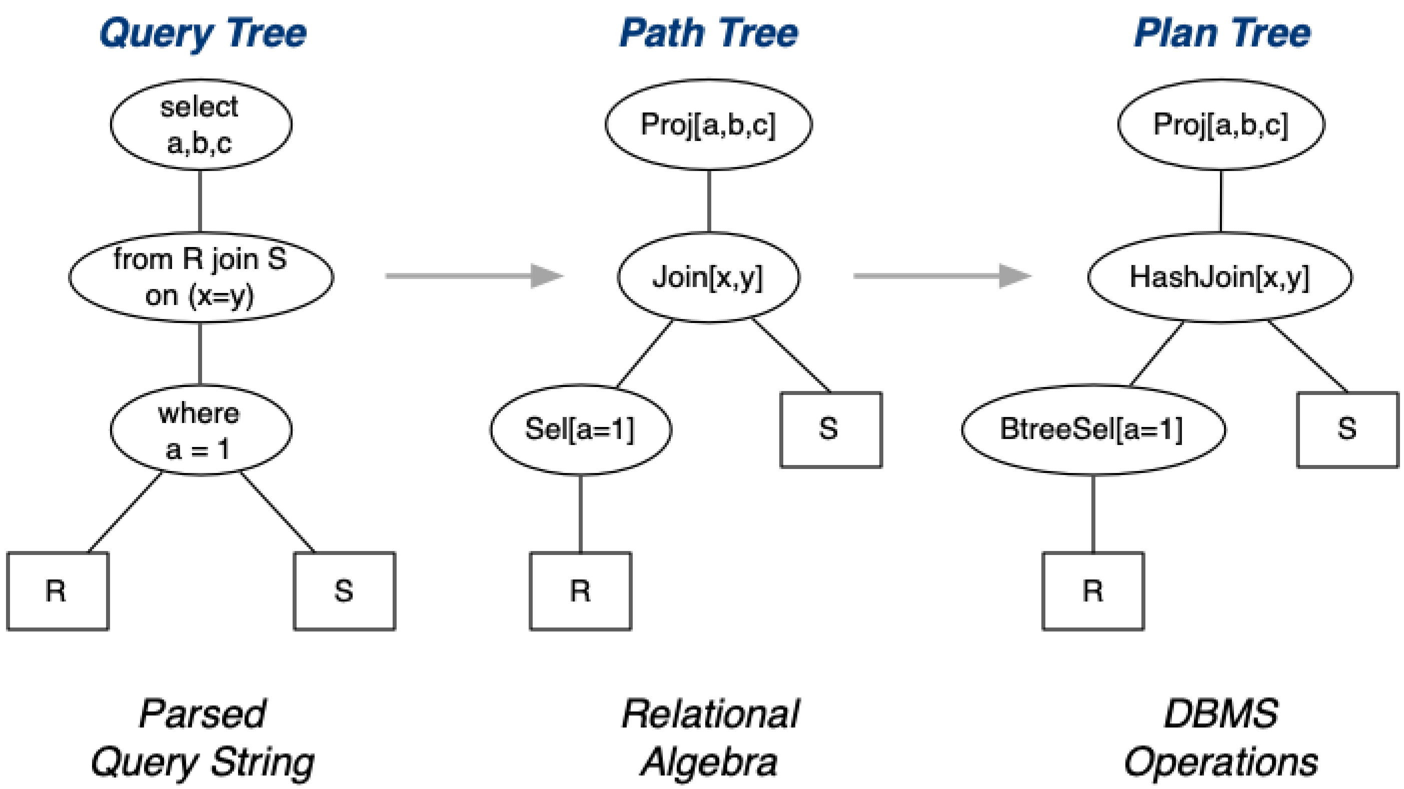
- using cost-based analysis of generated paths
- via one of two different path generators
- chooses least-cost path from all those considered

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
```



Produced: 5 Apr 2021