**Query Processing Steps**

Query

PARSER (parsing and semantic checking as in any compiler)

Parse tree (~ data structure representing tuple calculus expression)

OPTIMIZER (very advanced)

Execution plan (annotated relation algebra expression)

EXECUTOR (execution plan interpreter)

Query result

---

**Query Optimizer and Execution Steps**

Tuple calculus

VIEW EXPANSION

Tuple calculus

REWRITES

Tuple calculus

COST-BASED QUERY OPTIMIZATION

Execution plan using *annotated (physical) relational algebra* ('code to execute query’ in book)
### The Query Processing problem

- **Transform:**
  
  High-Level Declarative Query $\rightarrow$ Low-Level Execution Plan

  Normally: Relational Calculus $\rightarrow$ Annotated Physical Relational Algebra *(code in book)*

- The execution plan is a (functional) program that is interpreted by the *evaluation engine* to produce the query result

- Problem: For every query there may be very many possible execution plans: $O(2^{|Q|})$ where $|Q|$ is number of operations in query

- The optimal plan can be millions of times faster than an unoptimized plan!

  The complexity of optimal plan improved.

  E.g from $O(N^2)$ to $O(1)$, where $N$ is size of database!

- Query optimization may have huge payoff!

  However: Query optimization time may be significant!

### Query execution plan

- Query execution plan is functional program with evaluation primitives:

  Tuple scan operator
  
  Tuple selection operator
  
  Various index scan operators
  
  Various join algorithms
  
  Sort operator
  
  Duplicate elimination operator
  
  ..... 

- Normally *pipelined* execution

  *Streams* of tuples produced as intermediate results

  Intermediate results can sometimes be *materialized* too
Cost-based query optimization

- Cost-based query optimization:
  1. Generate all possible execution plans (heuristics to avoid some unlikely ones)
  2. Estimate the cost of executing each of the generated plans
  3. Choose the cheapest one
- Optimization criteria
  # of disk blocks read (dominates)
  CPU usage
  Normally weighted average of different criteria.

The Query Processing Problem

- Degrees of freedom:
  Query plan must be efficient and correct
  Choice of:
  scan tuples
  traverse index
  order of joining tables
  algorithms used for join
  available main memory
  materialization of intermediate results
  pipelining intermediated results (streaming)
  sort intermediate results
  duplicate elimination
Query Cost Models

- Basic costs parameters
  - Cost of accessing disk block
  - Data transfer rates
  - Clustering of data tuples on disk
  - Sort order of data tuples on disk
- Cost models of basic evaluation primitives
  - Cost of scanning disk segment containing tuples
  - Performance models for different index access methods (tree structures - hashing)
  - Performance models for different join methods
  - Cost of sorting intermediate results
- Total cost of an execution plan
  - The total cost depends on frequency of primitive operations invoked.
  - The frequency of invocation of primitive operations in execution plans depends on size of intermediate results.
  - Intermediate results estimated by statistical models.

Data statistics

- Used statistics to estimate size of intermediate results:
  - Size of tabless
  - Number of different column values
  - Histogram of distributions of column values
  - Models for estimating sizes of selections
    - E.g. selectivity of PNR=xxxx, AGE>xxx, etc.
    - Models for estimating sizes of results from joins
  - The models are often very rough
    - Work rather well since models used only for comparing different execution strategies - not for getting the exact execution costs.
- Cost of maintaining data statistics
  - Cheap: e.g size of relation, depth of B-tree.
  - Expensive: e.g. distribution on non-indexed columns, histograms
- Occasional statistics updates when load is low
  - Statistics not always up-to-date
  - Wrong statistics -> sub-optimal but correct plans
Complexity of optimizer

- Standard ‘System R’ cost-based optimization method
  NP hard in general ($O(2^n)$) where $|Q|$ is size of query
  Dynamic programming (semi-exhaustive search) optimizes in $O(|Q|^2)$ in best case.
  Works well up to ca 8 joins
- Heuristic methods $O(|Q|^2)$
  Not optimal plan
- Randomized and genetic methods $O(|Q|^2)$
  Converges to optimal plan
- Optimization timing
  Static
    - Canned queries by SQL preprocessor
  Dynamic
    - Optimize for every query
  Hybride
    - Optimize at program startup ($prepare$ in ODBC/JDBC)
    - Choose among several precompiled plans

Optimizing large queries

- Don’t optimize at all, i.e. order of predicates significant (old Oracle)
- Optimize partly, i.e. up to ca 8 joins, leave rest unoptimized (new Oracle)
- Heuristic methods (Ingres)
- Randomized (Monte Carlo) methods (research papers)
- Hybride methods, mix dynamic programming, heuristic, randomized
- User breaks down large queries to many small queries manually (often necessary for translating relational representations to complex object structures in application programs)