Query Optimization

Principles of Modern Database Systems
2007

Tore Risch
Dept. of information technology
Uppsala University
Sweden
Query execution plan

Query execution plan is functional program with primitives:
  Tuple scan operator
  Tuple selection operator
  Various index scan operators
  Various join algorithm operators
  Sort operator
  Duplicate elimination operator
  Stop after N tuples operator

.....

Normally *pipelined* execution
  *Streams* of tuples produced as intermediate results
  Avoid building large main memory data structures
  Intermediate results can sometimes be *materialized* too
Degrees of freedom:

*Plan enumeration*: Generating all different possible execution plans

Choice of, e.g.:
- scan tuples vs traverse indexes
- choose indexes to traverse
- choose join order
- choose algorithms used for joins
- resources restricted by available main memory
- possible materialization of intermediate results
- intermediate results need sorting
- duplicate elimination of intermediate results

...
Data statistics

- Used statistics to estimate size of intermediate results:
- Size of tables
- Number of different column values
- Histogram of distributions of column values
  E.g. selectivity of AGE>xxx, etc.
- Classically rough models that still work rather well since models used only for comparing different execution strategies - not for getting the exact execution costs.
- Data independence assumed – major source of estimate errors
Cost of maintaining data statistics

- Cost of maintaining data statistics
- Cheap: e.g. size of relation, depth of B-tree.
- Expensive: e.g. distribution on non-indexed columns,
- Occasional statistics updates – works well for steady-state
- Statistics not always up-to-date
- Wrong statistics -> sub-optimal but still correct plans
Dynamic programming:

```plaintext
optmethod('exhaustive');
dyprogopt(query)
    queue = priority queue containing queue nodes, qnode, of
    partial plans (qnode.partial),
    remaining parts of query (qnode.rest),
    and costs (qnode.cost)
initialize queue to qnode(nil,query,0);
while(true)
    if(queue empty) error("Query not executable");
    bestplan = subplan in queue with lowest cost;
    queue = remove(bestplan, queue);
    if(bestplan.rest empty)return bestplan;
    for each new queue node, nq,
        constructed from bestplan.partial
        extended with new partial neighbour plan, np,
        picked from bestplan.rest
        nq.partial = bestplan.partial + np;
        nq.rest = bestplan.rest - nq;
        nq.cost = bestplan.cost + cost(nq); (approx)
    add nq to queue;
```
Object-relational optimizers

10.2 User defined foreign functions
   select name from emp where northof(loc, 60)
   Can define own selection function:
   northof(locx, locy)

10.3 Associate function computing selectivity of foreign function

10.7 Associate function computing cost of foreign function

Also needed:
- Query transformation rules that recognize UDF patterns to simplify query
- Rewrites to utilize special indexing when applicable.
Amos II foreign functions

In Amos II:
create function sqrt(number x)->number y as multidirectional
  (‘bf’ foreign ‘SQRT’ cost {2,0.5})
  (‘fb’ foreign ‘SQUARE’ cost {1,1});
select sqrt(2.0); -> SQRT called.
select y where sqrt(y)=2; -> SQUARE called
select true where sqrt(4.0)=2.0; -> SQUARE called
Costs functions can be (foreign) Amos II functions.
Object-relational optimizers

10.4 User defined negators

\[ \text{not} (\text{close}(x, \text{loc}(5,5))) \iff \text{apart}(x, \text{loc}(5,5)) \]

10.6 User defined index updates

select … where \( \text{readness}(\text{picture}) < 0.1 \)
readiness evaluated when picture inserted or updated!

select … where \( \text{north}(\text{loc}) > 60 \)
north evaluated when picture inserted or updated.

10.9 User defined indexing

E.g. R-trees,
Requires API on server
Access to locks, recovery, page management
Object-relational optimizers

10.8 Smart handling of expensive predicates (functions)
Relational optimizer assumes all predicates cheap
  -> always evaluate (filter) early (selection pushing)
Functions such as readiness(..) may be expensive
  -> evaluate after all cheap filters (selection pulling)
=> Need optimizer handling expensive predicates
     properly (pull expensive predicates).
=> J.Hellerstein: *Optimization Techniques for Queries with Expensive Methods*
    How to modify traditional dynamic programming optimizer to handle expensive predicates.
10.10 Expression flattening

Basic idea: Functions/views are macro-expanded

Amos II expands *derived* functions.

create function foo(Date d)->bag of Emp e
as select e where startdate(e)>d;

Select name(e) from Emp e
where e = foo('…') and salary(e)>18000;

Use B-tree index on salary rather than
first evaluating foo if that requires a scan.

Traditional optimizer *expand views*, here functions too.
Object-relational optimizers

10.15 User defined aggregation operators
  Good idea.
  In Amos II: foreign functions
  Problem: Optimization

Conclusion:
  Object-relational optimizers must support extensibility of query language and of storage structures.
  Requires extended query optimizer compared to traditional relational optimizers.
Optimizing large queries

- Don’t optimize at all, order of predicates significant
- Optimize partly, i.e. up to ca 8 joins, leave rest unoptimized
- Heuristic methods
- 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)

...
Optimizing the optimizer (meta-optimization):

Naïve approach (trying all execution orders and indexes): \(O(|Q|!)\)

Dynamic programming \(O(|Q|^2) - O(3^{|Q|})\) generates optimal plan.

Normally used. System R style optimizer.

Hillclimbing \(O(|Q|^2)\) may generate suboptimal plans.

Randomized methods \(O(|Q|^2)\) converge to optimal plan.

Adaptive methods, modify plan dynamically by monitoring.

Does not rely on static statistics.