Randomized query optimization

Query optimization is a combinatorial optimization problem
- Each solution is a state in space, i.e. a node in a graph that includes all solutions.
- Each state has associated cost using some cost function.
- Goal: Find state with lowest cost.

**Randomized optimization**: Use various methods based on random generation of solutions followed by costing
Search space

Works well when many bottoms:

Hillclimbing good to guarantee to find one solution!
Terminology for randomized algorithms:

- Perform *random walks* through state space via a series of moves.
- A move originates in a *source state* and takes us into a *destination state*.
- The states reachable in one move from one state S is called *neighbours* of S.
- A move is *uphill* (*downhill*) if the cost of the source state is lower (higher) than the cost of the destination state.
- A state is *local minimum* if all neighbours are uphill.
- A state is *global minimum* if every other state is downhill.
- A state is a *plateau* if it has no downhill neighbour, but can reach downhill state without uphill moves.
Iterative improvement, II

- Idea:
  1. Start at random state.
  2. Move to randomly chosen downhill neighbour.
  3. Repeat until stopping condition reached.
- Repeat algorithm over and over.
- The more times, the more likely to reach global optimum.

II algorithm:
while not(stopping_condition) do
    S = random state
    while not(local_minimum(S)) do
        S' = random state in neighbours(S)
        if(cost(S')<cost(S)) then minS = S'
    return(minS)
Simulated annealing, SA
- Local optimization in II performs only downhill moves
- SA accept uphill moves too with some probability
- Avoid being caught in high cost local minimum
- Algorithm originally developed for annealing of crystals
SA idea/terminology
- Inner loop of SA called stage.
- Each stage performed under fixed value of parameter $T$, the temperature.
- Temperature controls probability of accepting uphill move, $P_u$
- $P_u = e^{-DC/T}$, where $DC$ difference in cost between old and new state.
- Higher temperature $\Rightarrow$ More likely to accept uphill move
- Higher $DC$ $\Rightarrow$ Less likely to accept uphill move
- The end of a stage reached when algorithm reached equilibrium.
- After each state lower $T$ according to some function.
- New stage begins.
- Stop algorithm when considered frozen, i.e. $T=0$. 
Simulated Annealing, SA, algorithm

S = S0
T = T0
minS = S
while not(frozen) do
    while not(equilibrum) do
        S’ = random state in neighbours(S)
        DC = cost(S’) - cost(S)
        if(DC < 0) then S = S’
        if(DC > 0) then S = S’ with probability \( e^{DC/T} \)
        if(cost(S) < cost(minS)) then minS = S
    T = reduce(T)
return(minS)
Two phase optimization

- Combination of II and simulated annealing
  A.Swami, *SIGMOD 1989: Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques*

- Amos II variant (II + sequence heuristics):
  [http://user.it.uu.se/~udbl/Theses/JoakimNasMSc.pdf](http://user.it.uu.se/~udbl/Theses/JoakimNasMSc.pdf)

```plaintext
  optmethod('randomopt');
  optlevel(50,1000);
```
Summary cost-based optimization

- With a good cost model it provides the optimal database execution plan
- Without it much less scalable query execution might occur
- Cost of optimization high
- There are alternative faster methods (e.g. randomized or heuristic optimization) but they give suboptimal plans
- Cost model need not be perfect as it is used only for comparing plans
- However, error in cost models may cause problems when:
  - Queries are large (errors multiplied)
  - There are statistical dependencies (independence assumed)
  - Costs are varying (e.g. network speed)
  - All data not known (e.g. parameterized queries, prepare in JDBC)
Prepared queries and the query cache

- Dynamic query compilation in program
- In JDBC (ODBC and other APIs)
- Idea:
  
  ```
  p = prepare("select name from person where name = ?")
  ...
  Execute(p, "Tore")
  ```

  Programmers make prepare statement in beginning of program. The compiled query is forgotten at end of session.

- Problems with `prepare`:
  - Programmer unaware of it!
  - Slow startup time for programs

- Modern DBMSs always have a `query cache`:
  - `Server` executes the preparations and saves in hashtable keyed by prepare string (including ?)
  - Saves start-up time
  - `prepare` followed by `execute` in loop efficient!
Dynamic query optimization

- Useful when
  - Queries are dynamic (i.e. dynamic strings sent to DBMS)
  - Parameterized queries (i.e. prepare in JDBC)
  - Cost changes during run
- Optimization of parameterized queries:
  prepare("select name from person where income > ?")
  Index on income
- Different plans depending on parameter value provided at execution time:
  Large value: Use index scan
  Small value: Use table scan
Dynamic query optimization

- One solution:

- Idea:
  Make several plans dependent on parameter
  Keep value intervals when plan applies
  Let prepare choose plan depending on actual parameter value

- Problem:
  Even slower query optimization
  Not useful when costs change dynamically (e.g. web)
Symmetric hash join

- Problems to solve:
  - Hash join favors one incoming argument
  - Not good in web environment
- For example:
  select s.sales+d.sales from SWStores s, DKsstores d
  where s.prod=d.prod
- Assume s and d accessed through slow network connections
- Hash join on s will stall if s blocks and vice versa
Symmetric hash join

- Solution:

- Idea:
  Make hash table on both operands
  Fill hash tables through two threads
  Emit tuples when match occurs

- Space overflow: spill tables to disk
Adaptive query optimization

- Problems to solve:
  - Startup time for queries
  - Dynamically changing costs
- Main paper:
- Idea:
  - Implement a pipelined multi-select-project-join operator, the *eddie* operator
  - One *eddie* operator adapts execution so to always work on data from incoming stream that delivers values
  - Work where data available in in-buffer
  - Buffer up intermediate results
Eddies (con.)

- Advantage:
  - Totally adaptive
  - Very low start-up cost

- Problems:
  - Eddie operator has overhead (25%)
  - Cost-based optimization generates better plans when cost model good
  - Adaptation may slow, bias towards first choice

- Improvement (STAIRS)
  A. Desphande & J. Hellerstein: *Lifting the Burden of History from Adaptive Query Processing*, VLDB 2004
  - Idea:
    - Break down *eddie* into smaller operators
    - Allow dynamic rollback-reconfigure-restart