CSE 444: Database Internals

Lecture 12
Query Optimization (part 3)

Guest Lecture: Sudeepa Roy

Sellinger Optimizer

- Problem:
  - How to order a series of N joins

  e.g. SELECT from A, B, C, D
  WHERE A.a = B.b AND A.c = D.d AND B.e = C.f

- How many logical/physical plans?
  - N! ways to order joins (e.g. ABCD, ACBD, ….)
  - Many plans per ordering (e.g. (((AB)C)D), ((AB)(CD)))
  - Multiple access paths for each relation
    - E.g. index-scan vs. file scan

- Naïve approach does not scale
  - E.g. N = 20, no. of join orders is 20! = 2.4 x 10^{18}, and many more plans

Ack: some slides are from Sam Madden's class at MIT

Sellinger Optimizer Properties

- Review: what we have done this week
- Only left-deep plan: (((AB)C)D)
  - Skeleton fixed, need to find the optimal order
- Push down selection
- Don't consider cartesian product
- Cost of a plan is IO + CPU
- Concept of *interesting order* during plan enumeration
  - Same order as that requested by ORDER BY or GROUP GY
  - Attributes that appear in equi-join predicates
    - They can speed-up a sort-merge join later

Dynamic Programming

R = set of relations to join (e.g. {A, B, C, D})

For d in {1 … |**R** |}

For **S** in {all length-d subsets of **R**}

optJoin(S) = $\min_{A \in S}$

[cost(**optJoin(S – {A})**)

+ $\min_{\text{access methods}}$ **[access cost for A]**

+ $\min_{\text{join methods}}$ **cost of joining (S – {A}) to A**]

Choice of relation for Logical plan

Cached from previous iterations

Physical implementation choices

Note: We are using optimality of subproblems. Why?

Example

- optJoin(A, B, C, D) **R** = {A, B, C, D}
- Assume all joins are nested-loop

- All subsets of size d = 1
  - {A}: best way to access A (seq. scan, index scan, predicate pushdown)
  - Similarly for {B}, {C}, {D}

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Hash index scan	100
B	Seq. scan	50
C	Seq scan	120
D	B+tree scan	400

Example

- optJoin(A, B, C, D)
- All subsets of size d = 2

  S = {A, B} : AB or BA
  - consider least cost option to access inner relation
  - Only one option for join (Nested loop)

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		
{A, B}	BA	156
{B, C}	BC	98
……..		

  - Similarly for S = {B, C}, {C, D}, {A, C}, {A, D}, {B, D}
- Total logical options: choose(N, 2) * 2

Example

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
….		
{A, B}	BA	156
{B, C}	BC	98
….		
{A, B, C}	BAC	500
….		

- optJoin(A, B, C, D)
- All subsets of size d = 3

S = {A, B, C} :

– Remove A, compute least cost join {B, C} to A

– Remove B, compute least cost join {A, C} to B

– Remove A, compute least cost join {A, B} to C

Similarly for S = {A, B, D}, {A C, D}, {B, C, D}, ….

optJoin(B, C) and its cost are already cached in table

Note: A little more general in simpledb-lab4, compares cost of joining {B, C} to A and also A to {B, C}

Total logical options: choose(N, 3) x 3 ---------- (x 2 in simpledb)

Example

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
{A, B}	BA	156
{B, C}	BC	98
{A, B, C}	BAC	500
{B, C, D}	DBC	150
……..		

- optJoin(A, B, C, D)
- Only one subset of size d = 4

S = {A, B, C, D} :

– Remove A, compute least cost join {B, C, D} to A

– Remove B, compute least cost join {A, C, D} to B

– Remove C, compute least cost join {A, B, D} to C

– Remove D, compute least cost join {A, B, C} to D

optJoin(B, C, D) and its cost are already cached in table

- Final answer is a plan with min-cost of these four
- Total logical options: choose(N, 4) x 4 (x 2 in simpledb)

Complexity

- No. of different subsets considered:
  - For a fixed value of d, Choose(N, d) choices of subsets S of size d
  - For a fixed choice of S, |S| = d, d choices of the inner relation A to be joined with S - {A}
- Total #logical options considered
  - Choose(N, 1) + Choose(N, 2) * 2 + ….. + Choose (N, N) * N

    <= N $\Sigma_{d = 1 \text{ to } N}$ Choose (N, d)

    <= **$N\ 2^N$**
  - #Options double in simpleDB
  - N = 20, cost = 2.1×10^7
  - Much smaller than the no. of left deep trees = N! = 20! = 2.4×10^{18}
- If there are m ways of doing the physical join, then #physical options = $O(mN2^N)$, also another factor for multiple "interesting orders"

Why Left-Deep and Not Right-Deep

- Asymmetric, cost depends on the order
  - Left: Outer relation Right: Inner relation
  - For nested-loop-join,

    we try to load the outer(typically smaller) relation in memory, then read the inner relation one page at a time

    B(R) + B(R) * B(S)
  - For index-join,

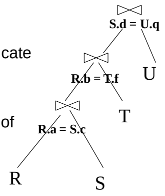
    we assume right (inner) relation has index

Why Left-Deep and Not Right-Deep

- Advantages of left-deep trees?
  1. Fits well with standard join algorithms (nested loop, one-pass), more efficient
  2. One pass join: Uses smaller memory
     1. ((R, S), T), can reuse the space for R while joining (R, S) with T
     2. (R, (S, T)): Need to hold R, compute (S, T), then join with R, worse if more relations
  3. Nested loop join, consider top-down iterator next()
     1. ((R, S), T), Reads the chunks of (R, S) once, reads stored base relation T multiple times
     2. (R, (S, T)): Reads the chunks of R once, reads computed relation (S, T) multiple times, either more time or more space

Implementation in SimpleDB (lab4)

1. JoinOptimizer.java (and the classes used there)
2. Returns vector of "LogicalJoinNode"
   a) Two base tables, two join attributes, predicate
   b) e.g. R(a, b), S(c, d), T(a, f), U(p, q)
   c) (R, S, R.a, S.c, =)
   d) Recall that SimpleDB stores all attributes of R, S after their join R.a, R.b, S.c, S.d



3. Output vector looks like:

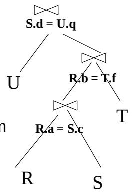
   <(R, S, R.a, S.c), (R, T, R.b, T.f), (S, U, S.d, U.q)>

Implementation in SimpleDB (lab4)

Any advantage of returning pairs?

- Flexibility to consider all linear plans

<(R, S, R.a,S.c), (R, T, R.b, T.f), (U, S, U.q, S.d)>



More Details:

1. You mainly need to implement “orderJoin(..)”
2. “CostCard” data structure stores a plan, its cost and cardinality: you would need to estimate them
3. “PlanCache” stores the table in dyn. Prog:

Maps a set of LJN to
a vector of LJN (best plan for the vector),
its cost,
and its cardinality

LJN = LogicalJoinNode

The Index Selection Problem

- So far
  - Given a physical plan, compute its cost
  - Given some choices of indexes for each relation, find the best logical/physical plan (Sellinger)
- Now
  - How to automatically choose indexes for relations
  - Index Selection Problem! (recall from 344)
  - Adv of index: search Disadv.: update
  - What are the parameters to consider?

The Index Selection Problem

- Given a database schema (tables, attributes)
- Given a “query workload”:
  - Workload = a set of (query, frequency) pairs
    - Either from log, or from the application programmer
  - The queries may be both SELECT and updates
  - Frequency = either a count, or a percentage
- Select a set of indexes that optimizes the workload
  - Either candidates are suggested to the programmer or some indexes are automatically created

In general this is a very hard problem

Basic Index Selection Guidelines

- Consider queries in workload in order of importance
  - If a query is only executed 1 out of 10000 times, we can ignore it
- Consider relations accessed by query
  - No point indexing other relations
- Look at WHERE clause for possible search key
  - Selection or join condition, selectivity of conditions
- Try to choose indexes that speed-up multiple queries

Basic Index Selection Guidelines

- And then consider the following…
  1. Which search key
  2. Multi attribute keys (covering index)
  3. Cluster or Unclustered
  4. Hash Index or B+ tree Index
  5. Query vs. Updates

---

1. Which Search Key

- Make some attribute K a search key if the WHERE clause contains:
  - An exact match on K
  - A range predicate on K
  - A join on K

2. Multi-attribute Keys

Consider creating a multi-attribute key K1, K2, … for a relation if

1. WHERE clause has matches on K1, K2, …
   - But also consider separate indexes
2. SELECT clause contains only K1, K2, ..
   - A covering index is one that can be used exclusively to answer a query without accessing the actual relation
   - e.g. index R(K1,K2) covers the query:

```
SELECT K2 FROM R WHERE K1=55
```

You will know about the other considerations

(Cluster or Unclustered, Hash Index or B+ tree Index, Query vs. Updates)

later in the lecture on “Database Tuning”