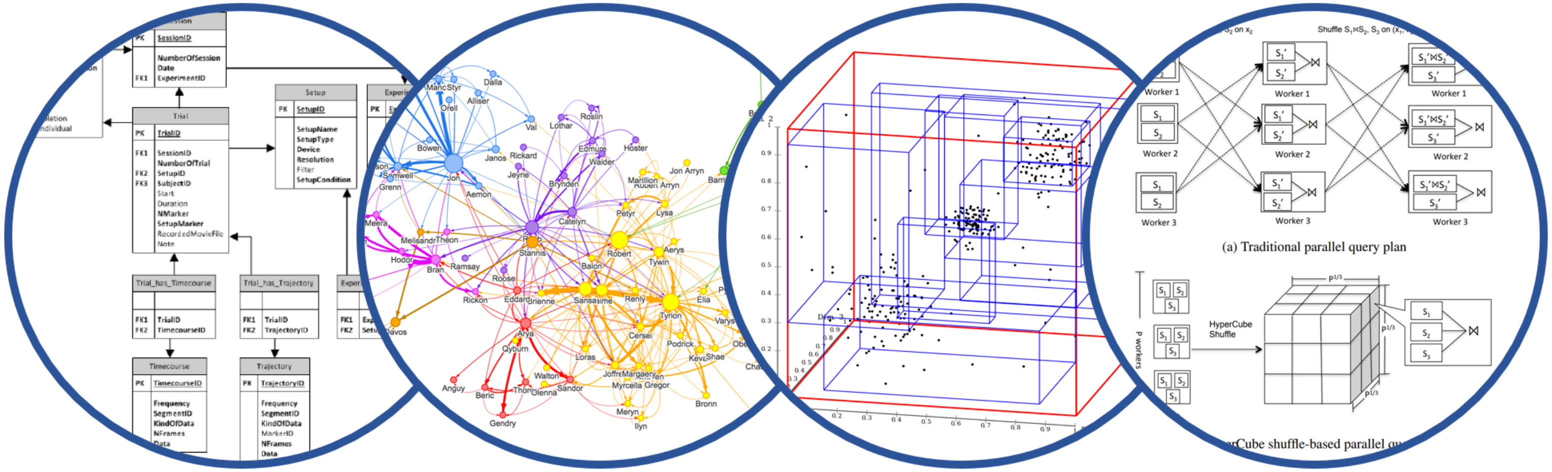


Database System Internals

Query Optimization Review

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

- I’m aware that students in class are affected by current events
- To help, we make two changes:
  - Cancel HW6 (apologies to 4 students who submitted)
  - Final report becomes extra credit
- Please do focus on Lab5: you will learn a lot
- Please fill out the course evaluation form: https://uw.iasystem.org/survey/225399

Final Project Instructions (Lab 5)

See course website for details!

1. Design and implementation:
   - There is **a mandatory part** and **extensions**
   - Design, implement, and evaluate extension (see specs)
2. Testing and evaluation
   - For your extension, write your own JUnit tests
   - Feel free to also write scripts
3. Final report – Extra credit

Final Report (Lab 5)

Extra credit (Spring'20) but highly recommended!

- Single-column & single-spaced
- Write your name!
- Structure of the final report
  - Sec 1. Overall System Architecture (2 pages)
    - Can reuse text from lab write-ups
  - Sec 2. Detailed design of the query optimizer and your extension (2 pages)
    - Include an **analysis** of the query plans that your system generates in different scenarios.
  - Sec 3. Discussion (0.5-1 page)

Selinger Optimizer History

- 1960's: first database systems
  - Use tree and graph data models
- 1970: Ted Codd proposes relational model
  - E.F. Codd. A relational model of data for large shared data banks. Communications of the ACM, 1970
- 1974: System R from IBM Research
  - One of first systems to implement relational model
- 1979: Seminal query optimizer paper by P. Selinger et. al.
  - Invented cost-based query optimization
  - Dynamic programming algorithm for join order computation

Next Example Acks

Implement variant of Selinger optimizer in SimpleDB

Designed to help you understand how this would work in SimpleDB

Many following slides from Sam Madden at MIT

Selinger Optimizer

Problem:

- How to order a series of joins over N tables A,B,C,...
  E.g. A.a = B.b AND A.c = D.d AND B.e = C.f
- N! ways to order joins; e.g. ABCD, ACBD, ...
- $C_{N-1} = \frac{1}{N}\binom{2(N-1)}{N-1}$ plans/ordering; e.g.
  (((AB)C)D),((AB)(CD)))
- Multiple implementations (hash, nested loops)
- Naïve approach does not scale
  - E.g. N = 20, #join orders 20! = 2.4 x 10^{18} ; many more plans

Selinger Optimizer

- Only left-deep plan: (((AB)C)D) – eliminate C_{N-1}.
- Push down selections
- Don't consider cartesian products
- Dynamic programming algorithm

Dynamic Programming

```
OrderJoins(…):
R = set of relations to join
For d = 1 to N:    /* where N = |R| */
    For S in {all size-d subsets of R}:
        optjoin(S) = (S – a) join a,
            where a is the single relation that minimizes:
                cost(optjoin(S – a)) +
                min.cost to join (S – a) with a +
                min.access cost for a
```

SimpleDB Lab5:
you implement orderJoins

Use: enumerateSubsets

Use:
computeCostAndCardOfSubplan

Note: optjoin(S-a) is cached from previous iterations

Example

- **orderJoins(A, B, C, D)**
- Assume all joins are Nested Loop

Subplan S	optJoin(S)	Cost(OptJoin(S))
A		

Example

- **orderJoins(A, B, C, D)**
- Assume all joins are NL

- d = 1
  - A = best way to access A (sequential scan, predicate-pushdown on index, etc)
  - B = best way to access B
  - C = best way to access C
  - D = best way to access D

- Total number of steps: **choose(N, 1)**

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

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B

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

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB

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

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB

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

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB
  - {C,D} = CD or DC
  - {A,C} = AC or CA
  - {B,D} = BD or DB
  - {A,D} = AD or DA

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

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB
  - {C,D} = CD or DC
  - {A,C} = AC or CA
  - {B,D} = BD or DB
  - {A,D} = AD or DA

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

- Total number of steps: 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
……..		

- **orderJoins(A, B, C, D)**
- d = 3
    - {A,B,C} =
      Remove A: compare A({B,C}) to ({B,C})A

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
……..		

- **orderJoins(A, B, C, D)**

- d = 3
  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A

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

Example

- **orderJoins(A, B, C, D)**

- d = 3
  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A
    Remove B: compare B({A,C}) to ({A,C})B
    Remove C: compare C({A,B}) to ({A,B})C

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

Example

- **orderJoins(A, B, C, D)**

- d = 3
  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A
    Remove B: compare B({A,C}) to ({A,C})B
    Remove C: compare C({A,B}) to ({A,B})C

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

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
……..		

- **orderJoins(A, B, C, D)**
- d = 3
  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A
    Remove B: compare B({A,C}) to ({A,C})B
    Remove C: compare C({A,B}) to ({A,B})C
  - {A,B,D} =
    Remove A: compare A({B,D}) to ({B,D})A
    …
  - {A,C,D} =…
  - {B,C,D} =…
- Total number of steps: choose(N, 3) × 3 × 2

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

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
………		

- **orderJoins(A, B, C, D)**
- d = 4
  - {A,B,C,D} =

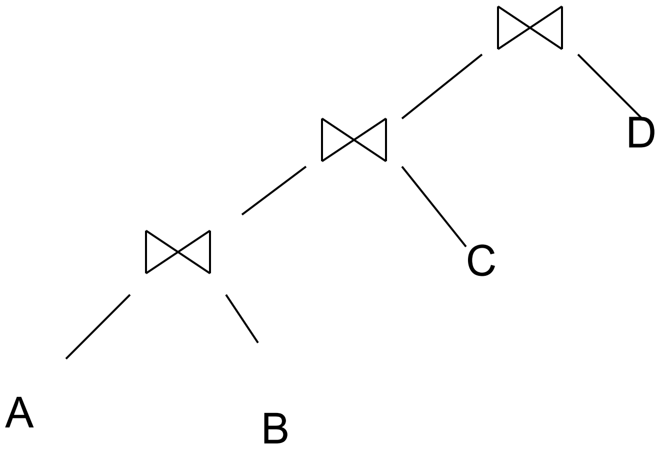
Remove A: compare A({B,C,D}) to ({B,C,D})A
Remove B: compare B({A,C,D}) to ({A,C,D})B
Remove C: compare C({A,B,D}) to ({A,B,D})C
Remove D: compare D({A,B,C}) to ({A,B,C})D

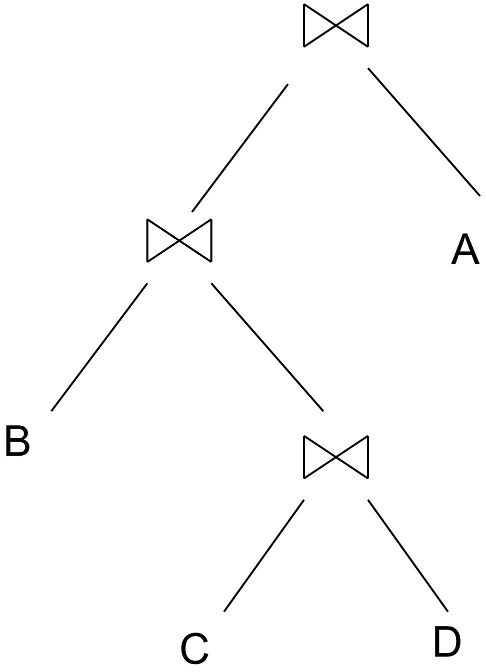
optJoin(B, C, D) and its cost are already cached in table

- Total number of steps: choose(N, 4) × 4 × 2

Discussion

- We kept the slides from Sam Madden from MIT, however they use inconsistently left-linear trees and linear trees
- For linear: both (BCD)A, A(BCD)
- For left linear: only (BCD)A, (ACD)B…
- For bushy: include (AB)(CD), etc





Complexity

- Total #subsets considered
  - Choose(N, 1) + Choose(N, 2) + ….. + Choose (N, N)
  - All nonempty subsets of a size N set: $2^N - 1$
  - Equivalently: number of binary strings of size N, except 00…0: ~~000~~, 001, 010, 011, 100, 101, 110, 111

Complexity

- Total #subsets considered
  - Choose(N, 1) + Choose(N, 2) + ….. + Choose (N, N)
  - All nonempty subsets of a size N set: $2^N - 1$
  - Equivalently: number of binary strings of size N, except 00…0: ~~000~~, 001, 010, 011, 100, 101, 110, 111
- For each subset of size d:
  - d ways to remove one element
  - 2 ways for compute AB or BA (except when d=2, when we already accounted for that – why?)

Complexity

- Total #subsets considered
  - Choose(N, 1) + Choose(N, 2) + ….. + Choose (N, N)
  - All nonempty subsets of a size N set: $2^N - 1$
  - Equivalently: number of binary strings of size N, except 00…0: ~~000~~, 001, 010, 011, 100, 101, 110, 111
- For each subset of size d:
  - d ways to remove one element
  - 2 ways for compute AB or BA (except when d=2, when we already accounted for that – why?)
- Total #plans considered
  - Choose(N, 1) + 2 Choose(N, 2) + ….. + N Choose (N, N)
  - Equivalently: total number of 1's in all strings of size N
  - $N\ 2^{N-1}$ because every 1 occurs 2^{N-1} times
  - Need to further multiply by 2, to account for AB or BA

Interesting Orders

- Some query plans produce data in sorted order
    - E.g scan over a primary index, merge-join
    - Called *interesting order*
- Next operator may use this order
    - E.g. can be another merge-join
- For each subset of relations, compute multiple optimal plans, one for each interesting order
- Increases complexity by factor k+1, where k=number of interesting orders

Why Left-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) or B(R) + B(R)/M * B(S)
- For index-join,

  we assume right (inner) relation has index

Why Left-Deep

- Advantages of left-deep trees?
    1. Fits well with standard join algorithms (nested loop, one-pass), more efficient

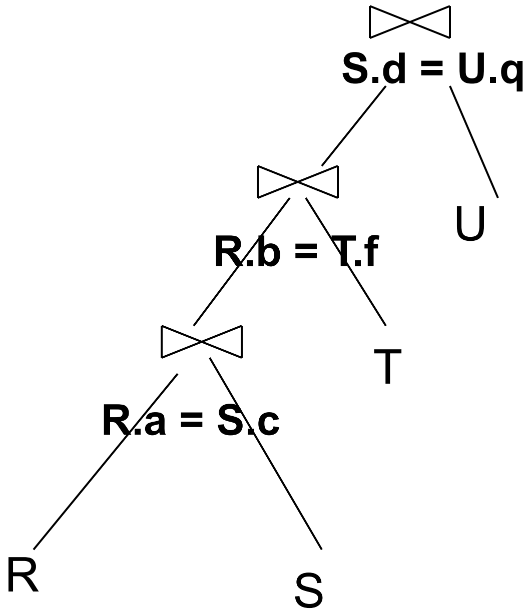
    2. One pass join: Uses smaller memory
        1. ((R, S), T), can reuse the space for R while joining (R, S) with T
        2. (R, (S, T)): Need to hold R, compute (S, T), then join with R, worse if more relations

    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 (lab5)

1. JoinOptimizer.java (and the classes used there)

2. Returns vector of "LogicalJoinNode"
   Two base tables, two join attributes, predicate
   e.g. R(a, b), S(c, d), T(a, f), U(p, q)
   (R, S, R.a, S.c, =)
   Recall that SimpleDB keeps all attributes of
   R, S after their join R.a, R.b, S.c, S.d



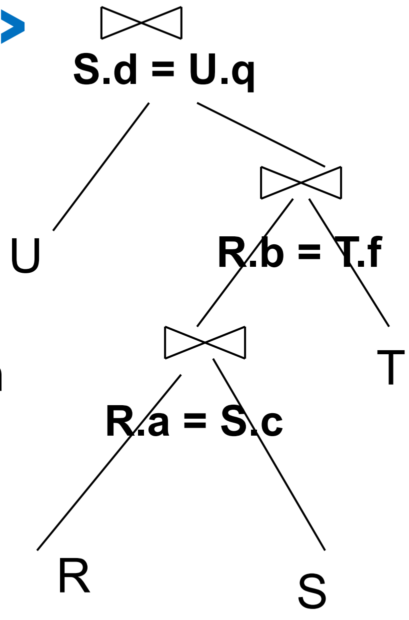
3. Output vector looks like:
   <(R, S, R.a, S.c), (R, T, R.b, T.f), (S, U, S.d, U.q)>

Implementation in SimpleDB (lab5)

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 "orderJoins(..)"
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