15-721 Database Management Systems

Query Optimization

Instructor: Anastassia Ailamaki
http://www.cs.cmu.edu/~natassa

Roadmap - detailed

- Processing steps - overview
- Single-table query optimization
- Join query optimization
- Nested queries

Query Processing Phases

- Parsing
- Optimization
- Code Generation
- Execution

Types of queries

(query block)

- Single-table
- 2-way join
- n-way join
- nested subqueries

Access Paths

- Segment (Relation) Scan - each page is accessed exactly once
- Index Scan (B+ Tree)
  - Clustered.
    - each index & data page: is touched once
  - Unclustered.
    - each index page: touched once
    - each tuple may be touched once, but each page may be fetched multiple times

Join Methods

- Nested Loops
- Sort-merge
- (Hash join)

- Access path is orthogonal choice

Useful Definitions

- A SARGable predicate:
  attribute op value
- A SARG (Search ARGument for scans) :
  a boolean expression of the SARGable predicates in disjunctive normal form:
  SARG1 or SARG2 or … or SARGn
  (SARG1 and … and SARGn) or
  (SARGn+1 and … and SARGq) or …

Definitions (cont.)

- A predicate (or set of predicates) matches an index when
  predicates are SARGable, and
  columns in the predicate are initial substring of index key

Example

- Index: name, location
  Predicates:
  "name = smith" matches index
  "name = smith or name = jones" matches
  "name = smith and location = San Jose" matches
  "(name = x and location = z) or (name = y and location = q)" matches

Definitions (cont.)

- An ordering of tuples is interesting if it is an ordering needed for a
  - GroupBy,
  - OrderBy, or
  - Join

Carnegie Mellon

Roadmap - detailed

- Processing steps - overview
- Single-table query optimization
- Join query optimization
- Nested queries

Carnegie Mellon

Single-Relation: Cost Model

- Cost of a Query - how would you measure it?

Carnegie Mellon

Single-Relation: Cost Model

- Cost of a Query =

  # page fetches + W*(#RSI Calls)
- (#RSI Calls) = #tuples returned by RSI
- W is a weighting factor
  - pages fetched vs. instructions executed

Single relation

- How to estimate #I/O's, for, say

  select *

  from EMP

  where salary > 30,000

Statistics

- What statistics would you need?

Statistics for Optimization

- NCARD (T) - cardinality of relation T in tuples
- TCARD (T) - number of pages containing tuples from T

Stats for Optimization (cont'd)

- P(T) = TCARD(T)/(# of non-empty pages in the segment)
  - If segments only held tuples from one relation there would be no need for P(T)
- ICARD(I) - number of distinct keys in index I
- NINDX(I) - number of pages in index I

Comments

- How / how-often would you update the stats?

Comments

- statistics not updated with each insert/delete/modify statement
- generated at load time
- update periodically using the update statistics command

Single relation

- How to estimate #I/O's, for, say

  select *
  from EMP
  where salary > 30,000

Step #1 of Query Optimization

- Calculate a selectivity factor 'F' for each boolean factor in the predicate list
- Single-relation access paths
- a1 = value; a1=a2; value1<=a1<=value2
- p or q; not p; p and q

Predicate Selectivity Estimation

attr = value	
attr1 = attr2	
val1 < attr < val2	
expr1 or expr2	
expr1 and expr2	
NOT expr	

Predicate Selectivity Estimation

attr = value	**F = 1/ICARD(attr index)** – if index exists **F = 1/10** otherwise
attr1 = attr2	
val1 < attr < val2	
expr1 or expr2	
expr1 and expr2	
NOT expr	

Predicate Selectivity Estimation

attr = value	**F = 1/ICARD(attr index)** – if index exists **F = 1/10** otherwise
attr1 = attr2	**F = 1/max(ICARD(I1),ICARD(I2))** or **F = 1/ICARD(Ii)** – if only index i exists, or **F = 1/10**
val1 < attr < val2	
expr1 or expr2	
expr1 and expr2	
NOT expr	

Predicate Selectivity Estimation

attr = value	**F = 1/ICARD(attr index)** – if index exists **F = 1/10** otherwise
attr1 = attr2	**F = 1/max(ICARD(I1),ICARD(I2))** or **F = 1/ICARD(Ii)** – if only index i exists, or **F = 1/10**
val1 < attr < val2	**F = (value2-value1)/(high key-low key)** **F = 1/4** otherwise
expr1 or expr2	
expr1 and expr2	
NOT expr	

Predicate Selectivity Estimation

attr = value	**F = 1/ICARD(attr index)** – if index exists **F = 1/10** otherwise
attr1 = attr2	**F = 1/max(ICARD(I1),ICARD(I2))** or **F = 1/ICARD(Ii)** – if only index i exists, or **F = 1/10**
val1 < attr < val2	**F = (value2-value1)/(high key-low key)** **F = 1/4** otherwise
expr1 or expr2	**F = F(expr1)+F(expr2)–F(expr1)*F(expr2)**
expr1 and expr2	
NOT expr	

Predicate Selectivity Estimation

attr = value	**F = 1/ICARD(attr index)** – if index exists **F = 1/10** otherwise
attr1 = attr2	**F = 1/max(ICARD(I1),ICARD(I2))** or **F = 1/ICARD(Ii)** – if only index i exists, or **F = 1/10**
val1 < attr < val2	**F = (value2-value1)/(high key-low key)** **F = 1/4** otherwise
expr1 or expr2	**F = F(expr1)+F(expr2)–F(expr1)*F(expr2)**
expr1 and expr2	**F = F(expr1) * F(expr2)**
NOT expr	**F = 1 – F(expr)**

Comments

- Query cardinality is the product of the relation cardinalities times the selectivities of the query's boolean factor

  QCARD= $|R_1|*|R_2|*$ ... * $|R_n|*F_{R1}*F_{R2}*$... * $F_{Rn,}$
- RSICARD (# RSI calls performed) = $|R_1|*|R_2|*$ ...* $|R_n|$*selectivity factors of all SARGABLE boolean factors

Step #2 of Query Optimization

- For each relation, calculate the **cost** of scanning the relation for each suitable index, and a segment scan
- What is produced:

  i) Cost C in the form of # pages fetched + W*RSICARD

  ii) Ordering of tuples the access path will produce

Costs per Access Path Case

Unique index matching equal predicate	**1+1+W**
Clustered index I matching >=1 preds	**F(preds)*(NINDX(I)+TCARD)+W*RSICARD**
Non-clustered index I matching >=1 preds	**F(preds)*(NINDX(I)+NCARD)+W*RSICARD** **…or if buffer pool large enough…** **F(preds)*(NINDX(I)+TCARD)+W*RSICARD**
Segment scan	**TCARD/P + W*RSICARD**

Roadmap - detailed

- Processing steps - overview
- Single-table query optimization
- Join query optimization
  - 2-way joins & n-way joins
- Nested queries

Joins - Definitions

- Outer relation - tuple retrieved first from here
- Inner relation - tuples retrieved (possible based on outer tuple join value)
- Join predicate - relates columns of inner/outer relations

Two join methods considered

- Nested loops - scan inner for each outer tuple
- Merge scans - scan in join column order (via index or after sorting)
- (cost formulas?)

Cost Formulae for Joins

Pi=access path

Nested Loops: $Cost_{NLjoin} = C_{outer} (P1) + N{*}C_{inner} (P2)$
N :# of outer tuples satisfying predicate

Merge Join: $Cost_{MSjoin} = C_{outer}(P1) + N{*}C_{inner} (P2)$
Since both are assumed to be sorted,
C_{inner} = #inner pages/N +W*RSICARD

Cost Formulae for Joins (cont'd)

Note: same except for $C_{inner}(P2)$ is cheaper (potentially) in merge joins case:

$Cost_{Sort} = Cost_{ScanPath} + Cost_{DoSortItself} + Cost_{WriteTempFile}$

(much more accurate cost functions in Shapiro and Graefe)

N-way joins

- N! orders for N-way join (in general)
- How would you start enumerating them?
  - R1 JOIN R2 JOIN R3 JOIN R4

N-way joins (cont'd)

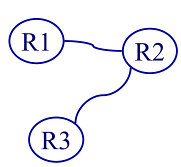
- N-way joins are performed as a sequence of 2-way joins
  - Can pipeline if no sort step is required
- (Heuristic: no (R1 JOIN R2) JOIN (R3 JOIN R4))

N-way joins (cont'd)

- Cartesian products (if any) done at end
- Join orders considered only when there is an inner - outer join predicate (and outer is all relations joined so far), except if all cross-products

Example

R1 join R2 and R2 join R3 on a different column



Forget

- R1 join R3 join R2
- R3 join R1 join R2

N-way joins (cont'd)

Important observation (dynamic programming):

- After k relations have been joined, method to add in (k+1)st is independent of the order for the first k (helps organize search)

Join Optimization Algorithm

1. Find best way to access each relation for each interesting tuple order and for the unordered case
2. Best way of join any relation to these -> pairs of relations
3. Find the best way adding a third rel. to the join
4. Continue adding additional relations via step 3
5. Choose cheapest path from root to leaf

Search Tree

- Tree for possible query processing strategies:
  - Root -> leaf path represents a way of processing query
  - Label edges with costs, orderings
  - Tree considers all reasonable options
    - Access paths
    - Orderings of tuples
    - Join Orderings
  - Trees for both nested loops and merge joins
  - Always take the cheapest way for the various interesting orders and prune more expensive equivalent plans

Optimization Example

- Assume the following database schema:
  - Emp (name, dno, job, salary), indices dno (clustered), job (unclustered)
  - Dept (dno, name, loc), indices dno (clustered)
  - Job (job, title) index job (clustered)

Optimization Example (cont'd)

- Consider optimization of the following query:

```
select  Emp.name, Emp.salary, Job.title, Dept.name
from    Emp, Dept, Job
where title="clerk" and location ="Denver"
        and Emp.dno = Dept.dno
        and Emp.job = Job.job
```


Optimization Example (cont.)

- Eligible predicates: Local predicates only
- "Interesting" orders: DNO, JOB

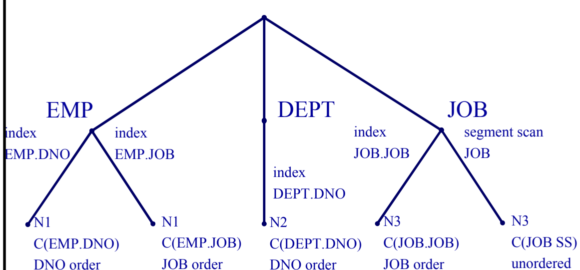
Access Paths for Single Relations

EMP:
- index EMP.DNO — N1, C(EMP.DNO)
- index EMP.JOB — N1, C(EMP.JOB)
- ~~segment scan on EMP — N1, C(EMP seg.scan)~~

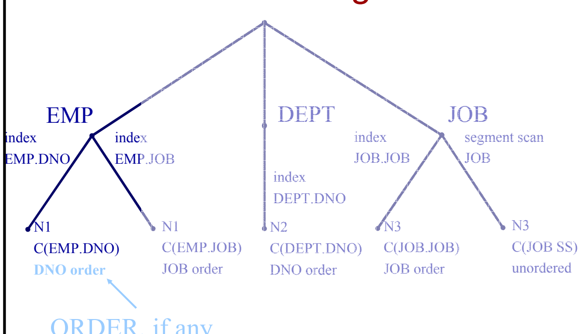
DEPT:
- index DEPT.DNO — N2, C(DEPT.DNO)
- ~~segment scan on DEPT — N2, C(DEPT seg.scan)~~

JOB:
- index JOB.JOB — N3, C(JOB.JOB)
- segment scan on JOB — N3, C(JOB seg.scan)

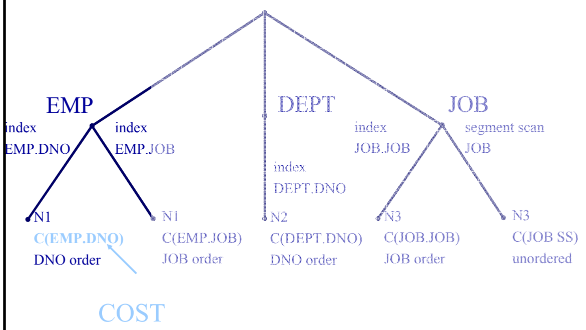
Search Tree for Single Relations



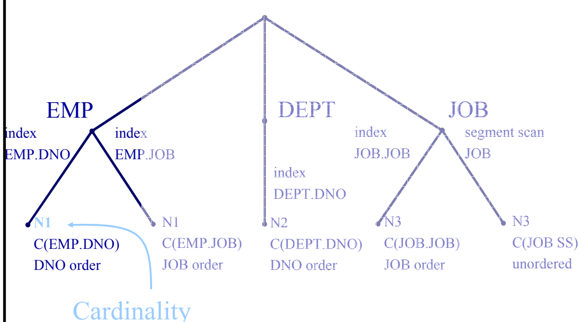
Search Tree for Single Relations



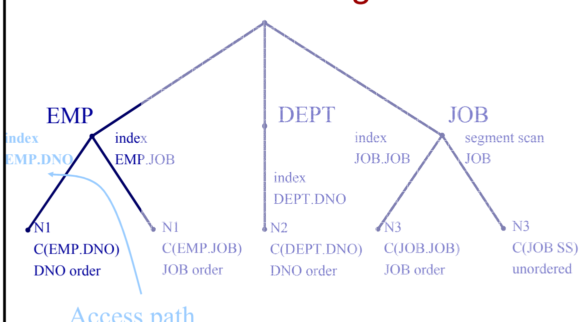
Search Tree for Single Relations



Search Tree for Single Relations

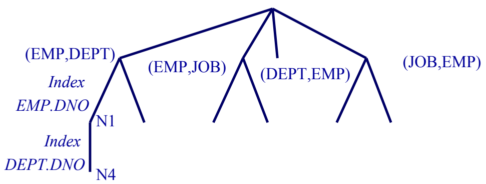


Search Tree for Single Relations

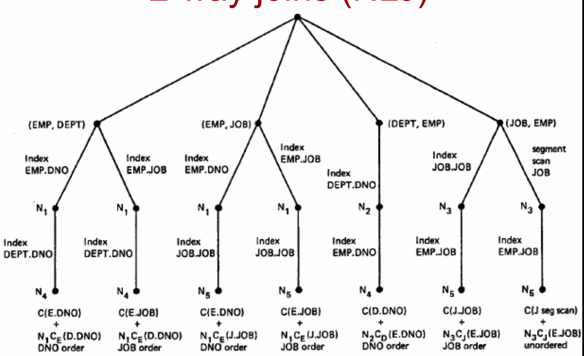


Next step

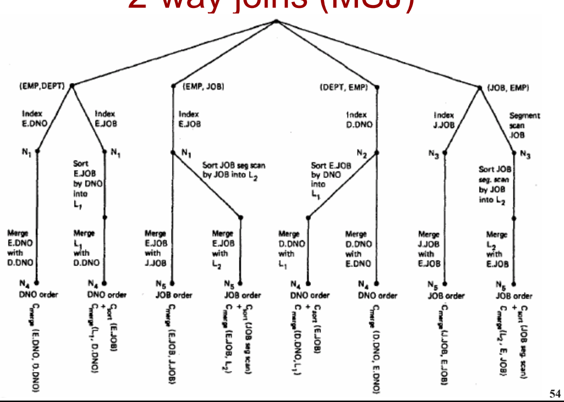
- Consider all 'allowed' 2-way joins - NL first



2-way joins (NLJ)



2-way joins (MSJ)



To add third relation

- Consider both EMP-DEPT and EMP-JOB solutions, find cheapest of each
- Then consider ways (NLJ, MSJ) to join third relation to the result

Complexity Considerations

- Exponential in N (the # of relations being joined
  - Fortunately N is pretty small (<= 3) in practice
  - How about # join methods considered?
- Pays off for compiled queries
- Can use heuristics for ad hoc queries
  - if the time spent optimizing exceeds the estimated execution time, quit optimizing and simply run the query

Roadmap - detailed

- Processing steps - overview
- Single-table query optimization
- Join query optimization
- Nested queries

Nested queries

- "Uncorrelated"

  select name from EMP
  where dno IN   (select dno
                   from DEPT
                   where loc = "Denver");
- **Q:** How optimize this query?

Nested queries

- "Uncorrelated"

  select name from EMP
  where dno IN   (select dno
                   from DEPT
                   where loc = "Denver");
- **Q:** How optimize this query?
- **A:** Subquery needs to be evaluated only once:
  - compute inner block first, replace cost into outer

Nested queries

- 'Correlated':

  select name from EMP X
  where salary > (select salary
                   from EMP
                   where EMP.number = X.manager)
- Must evaluate subquery for every tuple of the outer block!
- Complex (think of multiple nesting cases)

Conclusions

- Cost turns out to be good for most reasonable queries
  - Relative (not absolute) accuracy is what matters
- proposed use of statistics (recently: better statistics)

Conclusions cont'd

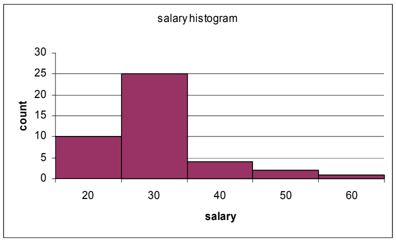
- consideration of CPU utilization and I/O activity
- selectivity factors, etc
- interesting orders save unnecessary sorting
- Today: CPU costs more important
  - Need to be factored in
- Optimization/execution times different
  - Interference

Addendum

- Uniformity and Independence assumptions
- Neither holds!
- Both lead to pessimistic results [Christodoulakis, TODS 84]
- How to avoid the uniformity assumption?

Addendum

- A: Histograms!



Addendum

- For details, see [Ioannidis & Poosala, SIGMOD 95]