Access Path Selection in System R

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

Query Processing Phases

- Parsing
- Optimization
- Code Generation
- Execution

Access Paths

- Segment (Relation) Scan - each page is accessed exactly once
- Index Scan (B+ Tree)
  - <u>Clustered.</u>
  - each index page is touched once
  - each data page is touched once
  - <u>Unclustered</u>.
  - each index page is 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 <u>interesting</u> if it is an ordered needed for a
  - GroupBy,
  - OrderBy, or
  - Join

Single-Relation: Cost Model

- Cost of a Query = # page fetches + W(#RSI Calls)
- W is a weighting factor
  - pages fetched vs. instructions executed
  - low for I/O bound machines
  - high for CPU bound machines

Statistics for Optimization

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

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

Step #1 of Query Optimization

- Calculate a selectivity factor ‘F’ for each boolean factor in the predicate list
- Single-relation access paths
- Formulae on the board

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= $|R_1|*|R_2|* \ldots * |R_n|*F_{R1}*F_{R2}*\ldots * F_{Rn,}$

- 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 + 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**

Joins - Definitions

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

Two join methods considered

- <u>Nested loops</u> - scan inner for each outer tuple
- <u>Merge scans</u> - scan in join column order (via index or after sorting)
- N-way joins are performed as a sequence of 2-way joins
  - Can pipeline if no sort step is required

Join Order Issues

- Cardinality of result is the same regardless of the join order
- N! orders for N-way join (in general)
- After k relations have been joined, method to add in (k+1)st is independent of the order for the 1st k (helps organize search)
- 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

Consider

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

Forget

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

Join Optimization Algorithm

1. Find best way to access <u>each</u> relation for each interesting tuple order and for the unordered case
2. Best way of join any relation to these if found ➔ produces solutions for joining pairs of relations
3. Find the best way of joining sets of three relations by considering all sets of two relations and joining in each third relation permitted by the join order heuristic
4. Continue adding additional relations via step 3
5. Choose cheapest path from root to leaf

Cost Formulae for Joins

Pi=access path

<u>Nested Loops:</u> $Cost_{NLjoin} = C_{outer}(P1) + N*C_{inner}(P2)$
N is the number of outer tuples satisfying predicate

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

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

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

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)

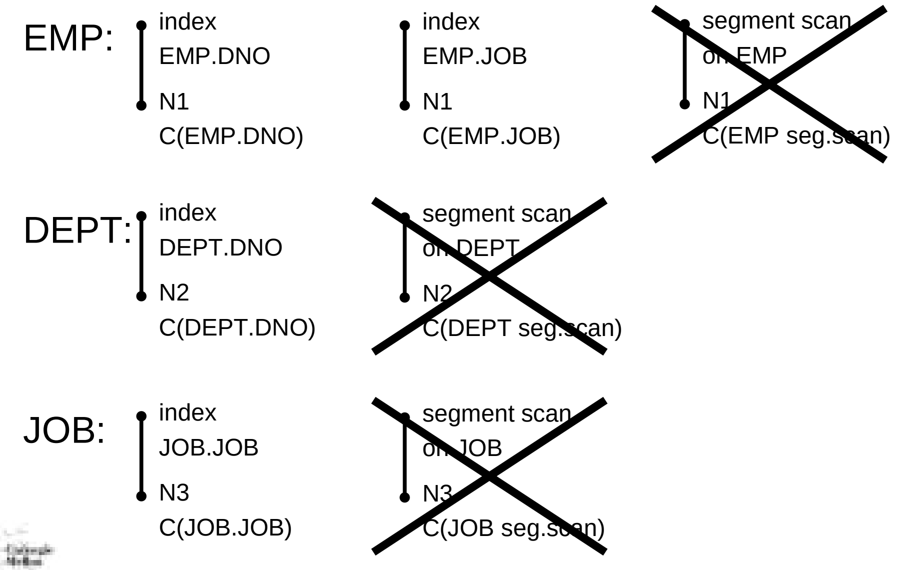
  Job (job, title) index job (clustered)
- Consider optimization of the following query:

```
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
```

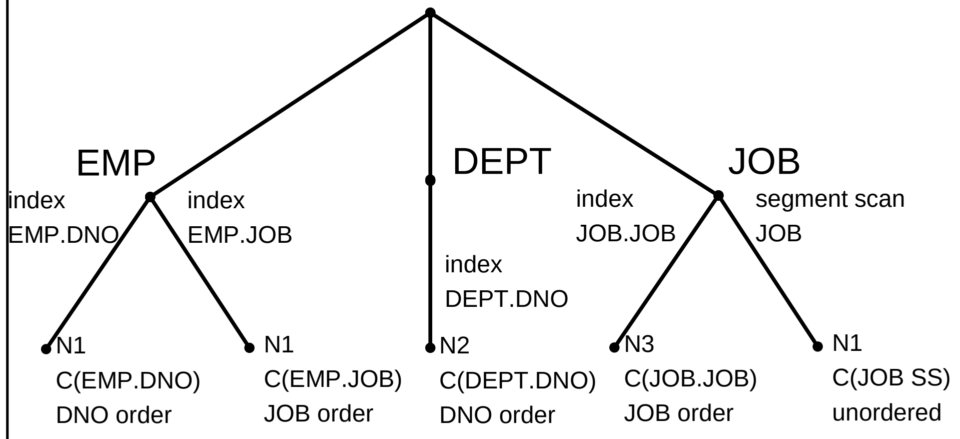
Optimization Example (cont.)

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

Access Paths for Single Relations



Search Tree for Single Relations



- It gets really complex as soon as we start considering joins (look at paper)!

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 estimated execution time exceeds the time spent optimizing, quit optimizing and simply run the query

Closing Remarks

- They also deal with “nested queries”, both simple ones and “correlated” ones
- Cost turns out to be good for most reasonable queries
  - Relative (not absolute) accuracy is what matters
- use of statistics (newer, better work out now)
- consideration of CPU utilization and I/O activity
- selectivity factors, etc
- interesting orders save sorting unnecessarily

Selectivity Histograms

- First found in Commercial INGRES (Koi, Ph.D. Thesis)
- Divide attribute domains into fixed range buckes, count number of hits for each bucket:
- Given a range query, base the selectivity estimate on the histogram data

Example

Histogram on age (152 values total)

range	**# of values in range**
0-9	3
10-19	7
20-29	62
...	**...**
90-99	1

consider the selection:

EMP[15 <= AGE <= 25]

Est. Selectivity = [(5*7)/10+(6*62)/10]/152 = 0.27
(instead of using 0.10 w/ uniform assumption!)

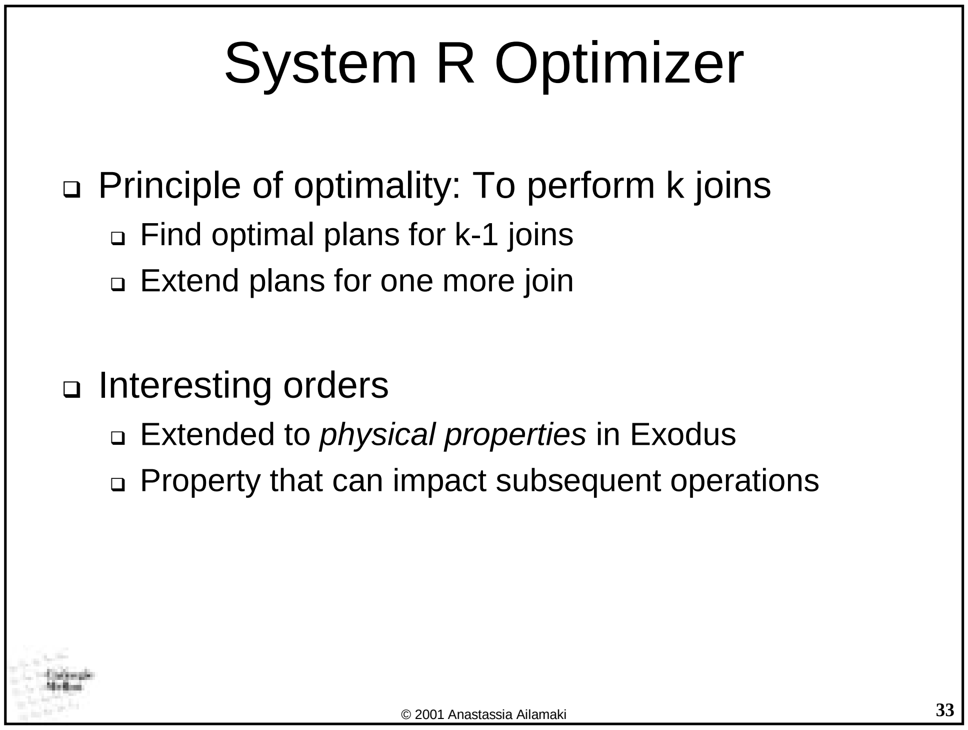
Overview of Query Optimization

- Chaudhuri, PODS 1998
- Query optimization =

search space of plans +
(low-cost plans)

cost estimation technique +
(accurate)

enumeration algorithm
(efficient)



System R Optimizer

- Principle of optimality: To perform k joins
  - Find optimal plans for k-1 joins
  - Extend plans for one more join
- Interesting orders
  - Extended to *physical properties* in Exodus
  - Property that can impact subsequent operations