Query Optimization

CPS 216

Advanced Database Systems

Wavelets

- Mathematical tool for hierarchical decomposition of functions and signals
- Haar wavelets: recursive pair-wise averaging and differencing at different resolutions
  - Simplest wavelet basis, easy to implement

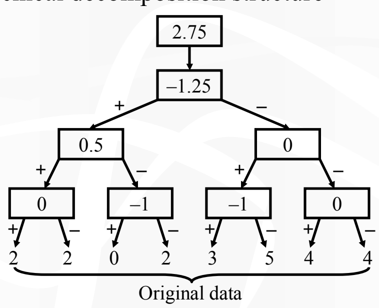
Resolution	Averages	Detail coefficients
3	[2, 2, 0, 2, 3, 5, 4, 4]	
2	[2, 1, 4, 4]	[0, –1, –1, 0]
1	[1.5, 4]	[0.5, 0]
0	[2.75]	[–1.25]

Haar wavelet decomposition: [2.75, –1.25, 0.5, 0, 0, –1, –1, 0]

2

Haar wavelet coefficients

- Hierarchical decomposition structure



3

Wavelet-based histogram

- Idea: use a compact subset of wavelet coefficients to approximate the data distribution (Matias et al., SIGMOD 1998)
  - The function to transform is the distribution function which maps v_i to f_i
- Steps
  - Compute cumulative data distribution function $C(v)$
    - $C(v)$ is the number of tuples with $R.A \leq v$
  - Compute wavelet transform of C
  - Coefficient thresholding: keep only the largest coefficients in absolute normalized value
    - For Haar wavelets, divide coefficients at resolution j by $2^{(j/2)}$

4

Using a wavelet-based histogram

- Q: $\sigma_{A > u \text{ AND } A \leq v} R$
- $| Q | = C(v) – C(u)$
- Search the tree to reconstruct $C(v)$ and $C(u)$
  - Worst case: two paths, $O(\log N)$, where N is the size of the domain
  - If we just store B coefficients, it becomes $O(B)$, but answers are now approximate
- What about Q: $\sigma_{A = v} R$?
  - Same as $\sigma_{A > v - 1 \text{ AND } A \leq v} R$

5

Summary of histograms

- Wavelet-based histograms are shown to work better than traditional bucket-based histograms
- The trick of using cumulative distribution for range query estimation also works for bucket-based histograms
- Trade-off: better accuracy ↔ bigger size; higher construction and maintenance costs

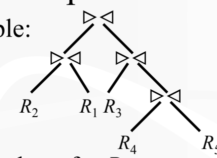
6

Cost-based query optimization

- Review
  - Algorithms for physical plan operators (sorting, hashing, indexing, …)
  - Query execution techniques (buffer management, pipelining using the iterator interface…)
  - Query rewrite techniques (relational algebra equivalences, unnesting, decorrelating SQL queries…)
  - Cost estimation techniques (I/O analysis of algorithms, histograms…)
- Mission: searching for an "optimal" plan
  - Focus on select-project-join query blocks
    - Join ordering is the most important subproblem

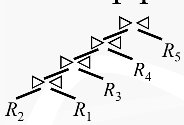
Search space

- "Bushy" plan example:



- How many plans are there for $R_1 \bowtie \ldots \bowtie R_n$?
  - Lots—close to $(n - 1)! \cdot 4^{(n-1)}$ (30240 for $n = 6$)
- There are more!
  - How about multiway joins?
  - How about different join methods?
  - How about placement of selection and projection?

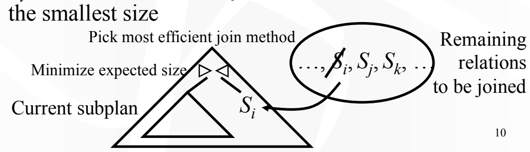
Left-deep plans



- Heuristic: consider only "left-deep" plans, wherein only the left child can be a join
  - Tend to be better than plans of other shapes
    - Many join algorithms scan inner (right) relation multiple times—you will not want it to be a complex subtree
- How many left-deep plans are there for $R_1 \bowtie \ldots \bowtie R_n$?
  - Significantly fewer, but still lots—n! (720 for $n = 6$)

A greedy algorithm

- $S_1, \ldots, S_n$
  - Say selections have been pushed down; i.e., $S_i = \sigma_p R_i$
- Start with the pair S_i, S_j with the smallest estimated size for $S_i \bowtie S_j$
- Repeat until no relation is left:
  Pick S_i from the remaining relations such that the join of S_i and the current result yields an intermediate result of the smallest size



Query optimization in System R

- A.k.a. Selinger-style query optimization
  - The classic paper on query optimization (Selinger et al., SIGMOD 1979)
- Basic ideas
  - Left-deep trees only
  - Bottom-up generation of plans
  - Interesting orders

Bottom-up plan generation

- Observation 1: Once we have joined k relations together, the method of joining this result further with another relation is independent of the previous join methods
- Observation 2: Any subplan of an optimal plan must also be optimal (otherwise we could replace the subplan to get a better overall plan)

» Not exactly accurate (next slide)

- Bottom-up generation of optimal plans
  - Compute the optimal plans for joining k relations together
    - Suboptimal plans are pruned
  - From these plans, derive the optimal plans for joining k+1 relations together

Motivation for "interesting order"

Example: $R(A, B) \bowtie S(A, C) \bowtie T(A, D)$

- Best plan for $R \bowtie S$: hash join (beats sort-merge join)
- Best overall plan: sort-merge join R and S, and then sort-merge join with T
  - Subplan of the optimal plan is not optimal!
- Why?
  - The result of the sort-merge join of R and S is sorted on A
  - This is an interesting order that can be exploited by later processing (e.g., join, duplicate elimination, GROUP BY, ORDER BY, etc.)!

13

Dealing with interesting orders

- When picking the optimal plan
  - Comparing their costs is not enough
    - Plans are not totally ordered by cost anymore
  - Comparing interesting orders is also needed
    - Plans are now partially ordered
    - Plan X is better than plan Y if
      - Cost of X is lower than Y
      - Interesting orders produced by X subsume those produced by Y
- Need to keep a set of optimal plans for joining every combination of k relations
  - Typically one for each interesting order

14

System-R algorithm

- Pass 1: Find the best single-relation plans
- Pass 2: Find the best two-relation plans by considering each single-relation plan (from Pass 1) as the outer relation and every other relation as the inner relation
  …
- Pass k: Find the best k-relation plans by considering each (k–1)-relation plan (from Pass k–1) as the outer relation and every other relation as the inner relation
  …
- Heuristics
  - Push selections and projections down
  - Process cross products at the end

15

Reasoning about predicates

- SELECT * FROM R, S, T
  WHERE $R.A = S.A$ AND $S.A = T.A$;
- Looks like a cross product between R and T
  - No join condition
- But there is really a join between R and T
  - $R.A = T.A$ is implied from the other two predicates
- A good optimizer should be able to detect this case and consider the possibility of joining R with T first

16

System-R algorithm example

- SELECT SID, CID
  FROM Student, Enroll, Course
  WHERE Student.age < 10
  AND Student.SID = Enroll.SID
  AND Enroll.CID = Course.CID
  AND Course.title LIKE '%data%';
- Primary keys/indexes
  - Student(SID), Enroll(CID, SID), Course(CID)
- Ordered, secondary indexes
  - Student(age), Course(title)

17

Example: pass 1

- Plans for {Student}
  - ☻ S1: Table scan, then filter (age < 10);
    cost 100; result ordered by SID ← interesting order
  - ☻ S2: Index scan using condition (age < 10);
    cost 5; result ordered by age ← not an interesting order
- Plans for {Enroll}
  - ☻ E1: Table scan;
    cost 1000; result ordered by CID, SID ← interesting order
- Plans for {Course}
  - ☻ C1: Table scan, then filter (title LIKE '%data%');
    cost 40; result ordered by CID ← interesting order
  - C2: Index scan, then filter (title LIKE '%data%');
    cost 160; result ordered by title ← not an interesting order

18

Example: pass 2

- Plans for {Student, Enroll}
  - – Extending best plans for {Student}
    - From S1: table scan, then filter (name = ’Bart’)
      - – Block-based nested loop join with Enroll; cost 1100
      - – Sort Enroll by SID, and merge join; cost 3100; ordered by SID ← no longer an interesting order
      - – … …
    - From S2: index scan using condition (name = ’Bart’)
      - ☻– Block-based nested loop join with Enroll; cost 1005
      - – … …
  - – Extending best plans for {Enroll} … …

Example: pass 2 continued

- Plans for {Student, Course}
  - – Ignore; it is a cross product
- Plans for {Enroll, Course}
  - – Extending best plans for {Course}
    - From C1: table scan, then filter (title LIKE ’%data%’)
      - ☻– Merge join; cost 1040
      - – … …
  - – Extending best plans for {Enroll} … …

Example: pass 3

- Finally, plans for {Student, Enroll, Course}
  - – Extending best plans for {Student, Enroll}
    - ☻• (INDEX-SCAN(Student) NLJ Enroll) NLJ FILTER(Course); cost …
    - … …
  - – Extending best plans for {Student, Course}
    - None!
  - – Extending best plans for {Enroll, Course}
    - (FILTER(Course) SMJ Enroll) NLJ (INDEX-SCAN(Student)); cost …
    - … …

Considering bushy plans

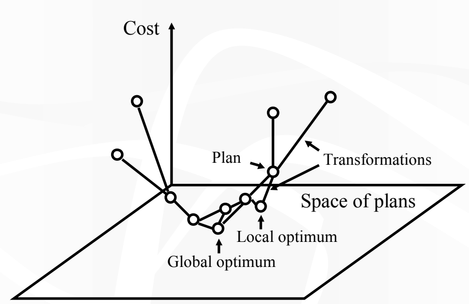
Straightforward generalization:

- Store all optimal 1-relation, 2-relation, …, and *k*-relation plans
- To find the optimal plan for *k*+1 relations
  - – For every possible partition of these relations into two groups, find the best ways of joining the optimal plans for the two groups
  - – Store the overall optimal plans

Optimizer “blow-up”

- A 20-way join will easily choke an optimizer using the System-R algorithm
- Solutions
  - – Heuristics-based query optimization
  - – Randomized query optimization (Ioannidis & Kang, SIGMOD 1990)

Search space revisited



Transformations

Relational algebra equivalences
(or query rewrite rules in general):

- Join method choice: $R \bowtie_{\text{method1}} S \rightarrow R \bowtie_{\text{method2}} S$
- Join commutativity: $R \bowtie S \rightarrow S \bowtie R$
- Join associativity: $(R \bowtie S) \bowtie T \rightarrow R \bowtie (S \bowtie T)$
- Left join exchange: $(R \bowtie S) \bowtie T \rightarrow R \bowtie (T \bowtie S)$
- Right join exchange: $R \bowtie (S \bowtie T) \rightarrow S \bowtie (R \bowtie T)$
- Why the last two redundant rules?
  - To avoid using the join commutativity rule, which does not change the cost of certain plans (e.g., sort-merge join)—creating plateaus in the plan space

Iterative improvement

- Repeat until some stopping condition (e.g., time runs out):
  - Start with a random plan
  - Repeatedly go downhill (i.e., pick a neighbor with a lower cost randomly) to get to a local optimum
- Return the smallest local optimum found

Simulated annealing

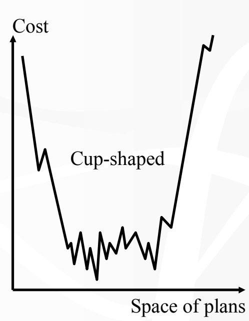
- Start with a plan and an initial temperature
- Repeat until temperature is 0:

  Repeat until some equilibrium (e.g., a fixed number of iterations):
  - Move to a random neighbor of the plan (an uphill move is allowed with probability $e^{-\Delta\text{cost}/\text{temperature}}$)
  - Reduce temperature
- Return the plan visited with the lowest cost

Two-phase optimization

- Phase I: run iterative improvement for a while to find a good local optimum
- Phase II: run simulated annealing with a low initial temperature to get more improvements
- Why does it tend to work better than both iterative improvement and simulated annealing?

Shape of the cost function



- An average local optimum has a much lower cost than an average plan
- The average distance between a random state and a local optimum is long
- There are lots of local optima
- Many local optima are connected together through low-cost plans within short distances

Comparison of randomized algorithms

- Iterative improvement
  - Too easily trapped in a local optimum
  - Too much work to restart
- Simulated annealing
  - Too much time spent on high-cost plans
- Two-phase
  - Phase I uses iterative improvement to get to the cup bottom quickly
  - Phase II uses simulated annealing to explore the cup bottom further