Relational Query Optimization

Linda Wu

(CMPT 354 • 2004-2)

Topics

- Query evaluation plan
- Query block
- Relation algebra equivalence
- Cost estimation
- Enumeration of alternative plans
- Nested queries

References

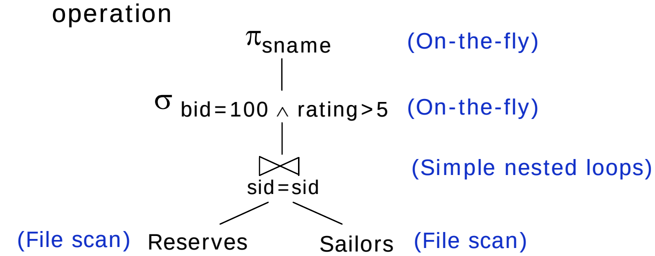
- Chapter 12: 12.1, 12.4.1, 12.4.2, 12.6.1
- Chapter 15

Overview of Query Optimization

- SQL queries are decomposed into smaller units: query blocks
- Query blocks are translated into extended relational algebra expressions
- A query optimizer optimizes a single block at a time
  - Enumerate alternative evaluation plans for the RA expression of the block
  - Estimate the cost of each enumerated plan, choose the plan with the lowest estimated cost
    - Ideally: find the best plan
    - Practically: avoid the worst plans

Query Evaluation Plan

- An extended relational algebra tree, with choice of algorithms for each relational operation



$$\pi_{sname}\,(\sigma_{bid=100 \wedge rating>5}\,(Reserves \bowtie Sailors))$$

Query Evaluation Plan (Cont.)

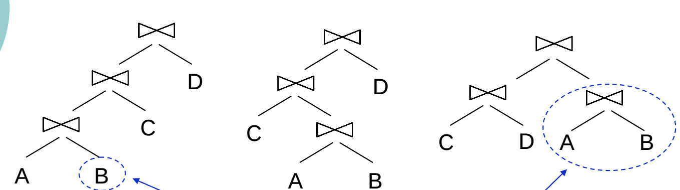
- Pipelining multiple-operator queries
  - The result of one operator is fed to the another operator without creating a temporary table (i.e., intermediate result is not materialized)
  - Save the cost of writing out the intermediate result and reading it back in
  - When the input table to a unary operator (π,σ) is pipelined into it, we say the operator is applied on-the-fly

Query Evaluation Plan (Cont.)

- Left-deep tree
  - On the plan tree, the right child of each join node is a base table
- Query optimizers typically concentrate on all left-deep plans
  - As # of joins increases, # of alternative plans increases: it is necessary to prune the space of alternative plans
  - Left-deep tree allows the generation of all fully-pipelined plans
  - Many alternative plans with less cost are ruled out!

Query Evaluation Plan (Cont.)

$A \bowtie B \bowtie C \bowtie D$



(Left-deep tree)

Right child: inner relation of join;
must be materialized

Query Block: Unit of Optimization

- Query block
  - An SQL query with no nesting and exactly one SELECT clause, one FROM clause, and at most one WHERE, GROUP BY and HAVING clauses
  - Nested block: usually treated as calls to a subroutine, made once per outer tuple

Outer block

```
SELECT  S.sid,  MIN (R.day)
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND
        B.color = 'red' AND S.rating =
                ( SELECT MAX (S2.rating)
                  FROM   Sailors S2 )
GROUP BY  S.sid
HAVING    COUNT (*) > 1
```

Nested block

Query Block (Cont.)

- Query block to relational algebra expression
  - SELECT $\rightarrow$ projection; WHERE $\rightarrow$ selection; FROM $\rightarrow$ cross product
  - A block is essential a $\sigma\pi\times$ algebra expression, with remaining operations carried out on the result of the $\sigma\pi\times$ expression

$\pi_{\text{S.sid, MIN(R.day)}}$ (
HAVING $_{\text{COUNT(*)>1}}$ (
GROUP BY $_{\text{S.sid}}$ (
$\pi_{\text{S.sid, R.day}}$ (
$\sigma_{\text{S.sid=R.sid} \wedge \text{R.bid=B.bid} \wedge \text{B.color='red'} \wedge \text{S.rating=valueFromNestedBlk}}$ (
Sailors $\times$ Reserves $\times$ Sailors))

Query Block (Cont.)

- Query block optimization
  - The optimizer find the best plan for the $\sigma\pi\times$ expression
    - Plans are enumerated by applying several equivalences between RA expressions
    - For each plan, estimate the I/O cost & result size for each operation in the tree
  - This plan is evaluated and the resulting tuples are sorted/hashed to implement the GROUP BY clause
  - The HAVING clause is applied to eliminate some groups
  - Aggregate operations in SELECT clause are computed for each remaining group

Relational Algebra Equivalences

- Allow us to choose different join orders and to push selections and projections ahead of joins

Selections: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$

$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ *(Commutative)*

Projections: $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$ *(Cascading)*

where $a_i \subseteq a_{i+1}$

Cross-products & joins: $R \times (S \times T) \equiv (R \times S) \times T$

$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ *(Associative)*

$R \times S \equiv S \times R$

$R \bowtie S \equiv S \bowtie R$ *(Commutative)*

RA Equivalences (Cont.)

- $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$
  if condition *c* only involves attributes in *a*
- $R \bowtie_c S \equiv \sigma_c(R \times S)$
- $\sigma_c(R \times S) \equiv \sigma_c(R) \times S, \quad \sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$
  if *c* appears in R but not S
- $\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$
  $\pi_a(R \bowtie S) \equiv \pi_{a1}(R) \bowtie \pi_{a2}(S)$
  if *a1* $\subset$ *a*, *a2* $\subset$ *a*, *a1* appear only in R, *a2* appear only in S (similar equivalences hold for selections)

Cost Estimation

- For each plan considered
  - Estimate the I/O cost of each operation in the plan tree
    - Need to know # of pages and available index (from system catalog)
  - Estimate the result size for each operation in the plan tree
    - The result of one operation is the input for the operation in the parent node, so, it affects the estimation of size and cost of the parent node
    - Use information about the input relations

Cost Estimation (Cont.)

- System catalogs
  - A collection of tables that contains the descriptive information for every table and index
- Information in catalog
  - For each relation: file name, file structure, attribute names/types, # tuples (NTuples), # pages (NPages), index name, PK, FK, ...
  - For each index: index name, index structure, search key attributes, # pages, low/high key values High(I)/Low(I), # distinct key values NKeys(I) ...
  - For each tree index: height

Cost Estimation (Cont.)

- Result size estimation (single relation query)
  - Result cardinality = (Max # tuples) * (product of all reduction factors)
    - Assume terms are independent
  - Maximum # tuples = product of the cardinalities of relations in the FROM clause
  - Reduction factor (RF) associated with each term reflects the impact of the term in reducing result size

```
SELECT  attribute list
FROM    relation list
WHERE   term1 AND ... AND termk
```


Cost Estimation (Cont.)

Terms	Reduction factor
column = value	1 / NKeys(I) or 1/10 Assuming uniform distribution of tuples among index key values
column1 = column2	1 / MAX(NKeys(I1), NKeys(I2)) or 1/10 Assuming each key value in the smaller index has a matching value in the other index
column > value	(High(I) – value) / (High(I) – Low(I))
column IN (list of values)	(RF of column=value) * (# of values in the list)

I, I1, I2: indexes on the corresponding columns

Cost Estimation (Cont.)

- I/O cost estimates for single relation query
  - Index *I* on primary key matches selection

    Height(I)+1 for a B+ tree, ≈1.2 for a hash index
  - Clustered index *I* matches one or more selection terms

    *(NPages(I)+NPages(R)) * (product of RFs of matching terms)*
  - Non-clustered index *I* matches one or more selection terms

    *(NPages(I)+NTuples(R)) * (product of RFs of matching terms)*
  - Sequential scan of file: *NPages(R)*

Cost Estimation (Cont.)

- Example 1: an index on rating
  - (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples
  - Clustered index

    cost = (1/NKeys(I)) * (NPages(I)+NPages(R))
    = (1/10) * (50+500)
  - Unclustered index

    Cost = (1/NKeys(I)) * (NPages(I)+NTuples(R))
    = (1/10) * (50+40000)
- Example 2: an index on sid
  - Clustered index: cost = 50+500
  - Unclustered index: cost = 50+40000
- Exampe 3: doing a file scan
  - We retrieve all file pages (500)

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating=8
```


Cost Estimation (Cont.)

- Result size estimates (multi-relation query)
  - Multirelation plans are built up by joining one new relation at a time
  - Cost of join method, plus estimation of join cardinality gives us both cost estimate and result size estimate

Enumeration of Alternative Plans

- Two main cases
  - Single-relation queries
    - Over a single relation
    - A combination of selects, projects, and aggregate operations; no join
  - Multi-relation queries
    - Over two or more relations
    - Requires join (or cross-product): expensive

Single-Relation Queries

- Each available access path is considered, and the one with the least estimated cost is chosen
  - Access path: a method of retrieving tuples, e,g., file scan, or, an index that match a selection
- The different operations are essentially carried out together
  - e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation

Single-Relation Queries (Cont.)

- Plan without index
  - Perform a file scan to retrieve tuples and apply the selection and projections on-the-fly
  - Write out tuples (as table *Temp)*
  - Sort *Temp*, and generate one answer tuple for each qualifying group on-the-fly
  - I/O cost
    - File scan: NPages(R)
    - Writing out *Temp* table: NPages(Temp) = NPages(R) * (size ratio) * (RF of selections)
    - Sorting: 3*NPages(Temp) (assume *Temp* can be sorted in 2 passes)

Single-Relation Queries (Cont.)

- Plans using indexes
  - Single-index access path
    - If several indexes match the selections, choose the most selective access path to retrieve data entries and then tuples, apply projections and remaining selection terms, (write the result to a temp relation), sort, compute aggregate values
  - Multiple-index access path
    - If several indexes of alt. (2)/(3) match the selection, use each index to retrieve a set of rids, intersect rids, sort rids by page id, retrieve tuples, apply projection and remaining selection terms, (write the result to a temp relation), sort, compute aggregate values

Single-Relation Queries (Cont.)

- Sorted index access path
  - If the list of grouping attribute is a prefix of a tree index, use the index to retrieve tuples in the order required by the GROUP BY clause, apply projections and selections and compute aggregate values on-the-fly
- Index-only access path
  - If all the attributes mentioned in the query are included in the search key for an index, index-only scan, apply projections and selections, (write the result to a temp relation), sort, compute aggregate values

Multi-Relation Queries

- Enumeration of left-deep plans
  - All enumerated left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join operation
  - N passes (if N relations joined)

    Pass 1: find best 1-relation plan for each relation; selection/projections considered as early as possible

    Pass 2: find best way to join result of each 1-relation plan (as outer) to another relation (all 2-relation plans)

    Pass N: find best way to join the result of a (N-1)-relation plan (as outer) to the N'th relation (all N-relation plans)

Multi-Relation Queries (Cont.)

- For each subset of relations, retain only
  - Cheapest plan overall, plus,
  - Cheapest plan for each "interesting order" of the tuples
- ORDER BY, GROUP BY, aggregates, etc.
  - Handled as a final step, using either an "interestingly ordered" plan or an additional sorting operator
- Avoid cross-product if possible
  - An N-1 relation plan is not combined with an additional relation unless there is a join condition between them, unless all terms in WHERE have been used up

Nested Queries

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition
- Outer block is optimized with the cost of "calling" nested block computation taken into account
- Implicit ordering imposed by nesting means that some good strategies are not considered
  - The non-nested version of the query is typically optimized better

Summary

- Query optimization is an important task in a relational DBMS
- Query optimization
  - Consider a set of alternative plans
  - Estimate cost of each plan that is considered
- Single-relation queries
  - All access paths considered, cheapest is chosen
- Multi-relation queries
  - Only left-deep plans considered, N passes