COMP9120

Relational Database Management Systems

Semester 2, 2016

Lecture 12. Query Optimization

Ramakrishnan & Gehrke: Ch 15, 12

Kifer, Bernstein, Lewis: Ch 11

Based on material by Alan Fekete, Uwe Roehm, Bryn Jeffries, and from textbooks by Silberschatz et al, Ramakrishnan et al, Kifer et al, and Ullman et al

Query Optimization

- A relational algebra expression may have many equivalent expressions
- Example:

```
SELECT BALANCE
  FROM account
WHERE balance < 2500
```

- Can be translated into

$$\sigma_{balance<2500}(\Pi_{balance}(account))$$

which is equivalent to

$$\Pi_{balance}(\sigma_{balance<2500}(account))$$

Query Optimization

Central Problems:

- Query is (by definition) declarative,
  e.g. it does not specify the execution order.

  But we need an executable plan.

- The goal of **real world query optimizers** is less to find the *optimal* plan,
  but more to avoid the *worst*.

  Time for query optimization adds to total
  query execution time.

Cost-based Query Optimization

- Generation of query-evaluation plans for an expression involves several steps:
  - Generating logically equivalent expressions
    - Use **equivalence rules** to transform an expression into an equivalent one.
  - Annotating resultant expressions to get alternative query plans
  - Choosing the cheapest plan based on **estimated cost**

- The overall process is called **cost-based optimization.**

THE UNIVERSITY OF SYDNEY

Statistics in the DB Catalog

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.

- Catalogs are updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

THE UNIVERSITY OF SYDNEY

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.

THE UNIVERSITY OF SYDNEY

Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time.
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple.
  - (This is an over-simplification, but serves for now.)

```
SELECT S.sname
  FROM Sailors S
 WHERE S.age IN
       (SELECT MAX (S2.age)
          FROM Sailors S2
         GROUP BY S2.rating)
```

Outer block

Nested block

- For each block, the plans considered are:
  - All available access paths, for each relation in FROM clause.
  - All *left-deep join trees* (i.e., all ways to join the relations one-at-a-time, with the inner relation in the FROM clause, considering all relation permutations and join methods.)

Equivalent Algebra Expressions

- Two relational algebra expressions are said to be **equivalent** if on every legal database instance the two expressions generate the same set of tuples
  - Note: order of tuples is irrelevant
  - In SQL, inputs and outputs are multisets of tuples, hence above definition on multiset of tuples
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

THE UNIVERSITY OF SYDNEY

Relational Algebra Equivalences

- Allow us to choose different join orders and to `push' selections and projections ahead of joins.
- Selections:

$$\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R)) \quad \text{(Cascade)}$$

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R)) \quad \text{(Commute)}$$

- Projections:

$$\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R))) \quad \text{(Cascade)}$$

- Joins:

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad \text{(Associative)}$$

$$(R \bowtie S) \equiv (S \bowtie R) \quad \text{(Commute)}$$

THE UNIVERSITY OF SYDNEY

More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
- A selection on just attributes of R commutes with R⋈S.
  - (i.e., $\sigma (R \bowtie S) \equiv \sigma (R) \bowtie S$)
- If a projection follows a join R ⋈ S, we can `push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.

Transformation Example

- Query: Find the names of all students who have enrolled in some course with 6 credit points.

$$\Pi_{name}(\sigma_{credit\_points=6}(course \bowtie (enrolled \bowtie student)))$$

- Transformation by commuting selection with join :

$$\Pi_{name}((\sigma_{credit\_points=6}(course)) \bowtie (enrolled \bowtie student)))$$

- Performing the selection as early as possible reduces the size of the relation to be joined.

Multiple Transformations Example

- Query: Find the names of all students which enrolled in a 6 credit_point course, whose grade is a distinction ('D').

$$\Pi_{name}(\sigma_{grade='D' \wedge credit\_points=6}(course \bowtie (enrolled \bowtie student)))$$

- Transformation using join association rule:

$$\Pi_{name}(\sigma_{grade='D' \wedge credit\_points=6}((course \bowtie enrolled) \bowtie student))$$

- Second form provides an opportunity to apply the "perform selections early" heuristic, resulting in the *sub*-expression

$$\sigma_{credit\_points=6}(course) \bowtie \sigma_{grade='D'}(enrolled)$$

- Thus a sequence of transformations can be useful

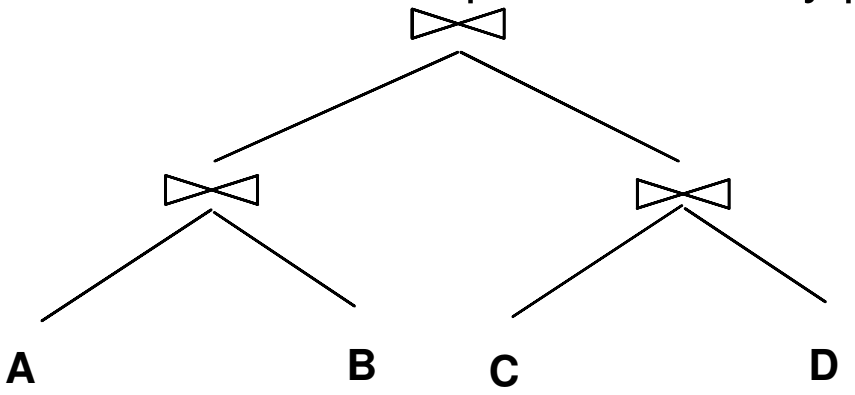
Optimization Strategies

- Ideally: find the optimal plan.
  - This approach is very expensive in time and space.
- Typically: Find a good plan, avoiding the worst plan.
  - do not generate all expressions
- Central Problem: Join Ordering (*Join Enumeration*)
- Typical Approach:
  - restrict to **left-deep join trees**

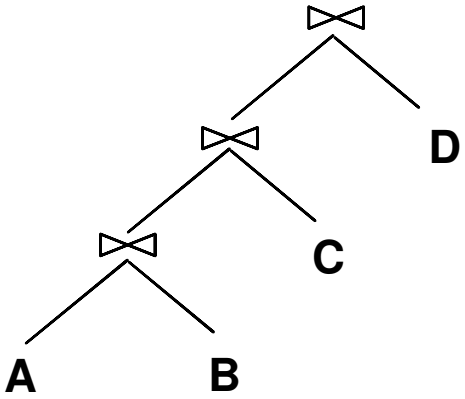
THE UNIVERSITY OF SYDNEY

Left-Deep Join Plans

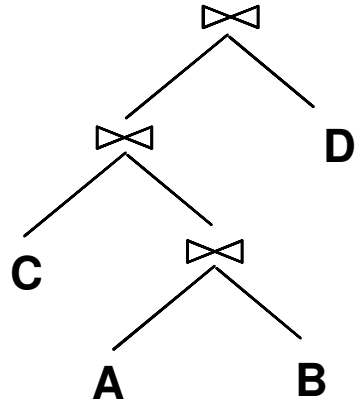
- Fundamental decision in System R: *only left-deep join trees* are considered.
- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).



bushy join plan



left-deep join plan



non-left-deep plan

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 of these blocks means that some good strategies are not considered. *The non-nested version of the query is typically optimized better.*

```
SELECT S.sname
  FROM Sailors S
 WHERE EXISTS
   (SELECT *
      FROM Reserves R
     WHERE  R.bid=103
       AND R.sid=S.sid)
```

Nested block to optimize:

```
SELECT *
  FROM Reserves R
 WHERE R.bid=103 AND
       S.sid=outer value
```

Equivalent non-nested query:

```
SELECT S.sname
  FROM Sailors S,Reserves R
 WHERE S.sid=R.sid
   AND R.bid=103
```

THE UNIVERSITY OF SYDNEY

Query Plan Cache

- Many DB applications use only a fixed set of queries.
- Parsing and optimising these queries again and again is costly
  - Hence: Most DBMS have a query plan cache
  - E.g. Oracle: If the same query (with the same constants) is issued again, use the previous plan

Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate *size of result* for each operation in tree!
    - Use information about the input relations.

- Database Statistics play a crucial role here
  - Our assumption: data values to be uniformly distributed (Note: This is typically not the case with many data sets)
  - How to determine a data distribution over large data sets?
  - How to keep those statistics up-to-date? (Periodic sampling)

Example

- The query

```
SELECT S.a, T.c
FROM S, T
WHERE T.a = S.a
      AND S.b = 5
```

Statistics

- S(a, b) has 10,000 rows each 50 bytes
  - 10,000 values for a
  - 2500 values for b
- T(c, a)
  - Foreign key (a) references S
  - 50,000 rows, each 40 bytes
  - 50,000 values for c
  - 10,000 values for a
- Assume a page is 4096 bytes, of which 4000 are useful for data records (the rest are header)

THE UNIVERSITY OF SYDNEY

Example

One Plan

- Scan data records of S, one page at a time
- For each page of S
  - Scan data records of T, one page at a time
  - For each page of T
    - Check each combination of a row in the S page with a row in the T page, to see if makes WHERE clause true
      - If so, output S.a, T.c

Cost estimate

- Cost to scan S
  - S has 4000/50=80 data records per page, so S is stored on 10000/80=125 pages (167 pages using 75% fill)
- Cost to scan T
  - T has 4000/40=100 data records per page, so T is stored on 50000/100=500 pages (667 pages using 75% fill)
- So we read 167 pages of S, and we scan T 167 times (doing 667 page reads each time)
- Total disk I/O is 167+167*667=111556 pages

This happens entirely in memory

Another Plan

- Scan data records of S, one page at a time
- For each page of S
  - Check each record to see whether S.b=5
  - If so, store record in Temp
- Scan Temp, for each page of Temp
  - Scan data records of T, one page at a time
  - For each page of T
    - Check each combination of a row in the temp page with a row in the T page, to see if makes WHERE clause true
      - If so, output S.a, T.c

This happens entirely in memory

Cost estimate

- Cost to scan S
  - S is stored on 167 pages
- Cost to store Temp
  - One in every 2500 rows of S will get to Temp
  - Temp has 4 rows, all fit in one page!
- Cost to scan Temp
  - 1 page read back in
- Cost to scan T
  - 667 pages of disk access
- So we scan T once (doing 667 page reads each time)
- Total disk I/O is 167+1+1+1*667=836 pages

Since Temp is so small, we could save this work and just keep it in memory

Example

A plan *if* S has clustered primary index on S.a

- Scan data records of T, one page at a time
- For each page of T, consider the records in the page
  - For each record of T in the page
  - Use index to fetch the record of S that has the appropriate value of S.a
    - Check whether S.b=5
      - If so, output S.a, T.c

Cost estimate

Assume index on S.a has 2 levels, excluding the leaf (record) pages.

- Cost to scan T
  - Read 667 pages
- Cost to look up row of S with given S.a
  - Read one page per level, then read the one data record that is pointed to (recall S.a is primary key of S)
  - Cost of lookup is 3 pages
- So we read 667 page of T, and we do index lookup on S once for each record in T (i.e., we do 50,000 index lookups)
- Total disk I/O is 667 + 50000*3 = 150667 pages

This happens entirely in memory

Example

A plan *if* S has unclustered secondary index on S.b, and T has unclustered secondary index on T.a

- Use index on S.b to find records with S.b = 5; store these in Temp
- Scan Temp, for each page of Temp, consider the records in the page
  - For each record of Temp in the page
  - Use index to fetch the records of T that have the appropriate value of T.a
    - For each such record, output S.a, T.c

This happens entirely in memory

Cost estimate

- **Assume index on S.b has 3 levels, and index on T.a has 3 levels (entries don't include records)**
- Cost to use index to find rows with S.b = 5
  - Read one page per level, then fetch the data records pointed to
  - There are 3 levels, and there will be 4 records in Temp
  - Cost for the lookup is 3+4 = 7 pages read
- Cost to write Temp and read back in: 1+1 = 2 pages
- Cost to use index to find rows of T with given T.a
  - Read one page per level, then fetch the data records pointed to
  - There will be 50000/10000 = 5 records with a given value of T.a
  - Cost of a lookup is 3 + 5 = 8 pages
- Total disk I/O is 7 + 1+ 1 + 4*8 = 41 pages

Choosing a plan

- If the appropriate indices existed, we would choose the fourth plan among those examined so far, as it has the lowest cost
- If the indices didn't exist, we can only consider plans that don't depend on them
  - Maybe we would choose the second plan
- In general, there are many possible plans to be considered
  - Query processor must be systematic in examining them
- Note that sometimes, a plan that doesn't use an index is faster than another which does use an index
  - There is no general rule to say which will end up cheaper
- The difference in cost between plans can be several orders of magnitude!

Summary

- Query optimization is an important task in a relational DBMS
- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered and choose the least expensive
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.