CMSC 424 – Database design
Lecture 18
Query optimization

Mihai Pop

Admin

- Homework 3 due

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Dynamic Programming in Optimization

- To find best join tree for a set of *n* relations:
  - To find best plan for a set *S* of *n* relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of *S*.
  - Recursively compute costs for joining subsets of *S* to find the cost of each plan. Choose the cheapest of the $2^n - 1$ alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on R_i using best choice of indices on R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Dynamic programming

Join Order Optimization Algorithm

```
procedure findbestplan($S$)
    if ($bestplan[S].cost \neq \infty$)
        return $bestplan[S]$
    // else $bestplan[S]$ has not been computed earlier, compute it now
    if ($S$ contains only 1 relation)
        set $bestplan[S].plan$ and $bestplan[S].cost$ based on the best way
        of accessing $S$  /* Using selections on S and indices on S */
    else for each non-empty subset $S1$ of $S$ such that $S1 \neq S$
        P1= findbestplan($S1$)
        P2= findbestplan($S - S1$)
        A = best algorithm for joining results of $P1$ and $P2$
        cost = $P1.cost + P2.cost$ + cost of $A$
        if $cost < bestplan[S].cost$
            $bestplan[S].cost$ = cost
            $bestplan[S].plan$ = "execute $P1.plan$; execute $P2.plan$;
                                  join results of $P1$ and $P2$ using $A$"
    return $bestplan[S]$
```

Dynamic programming example

- Enumerate all equivalent expressions for:

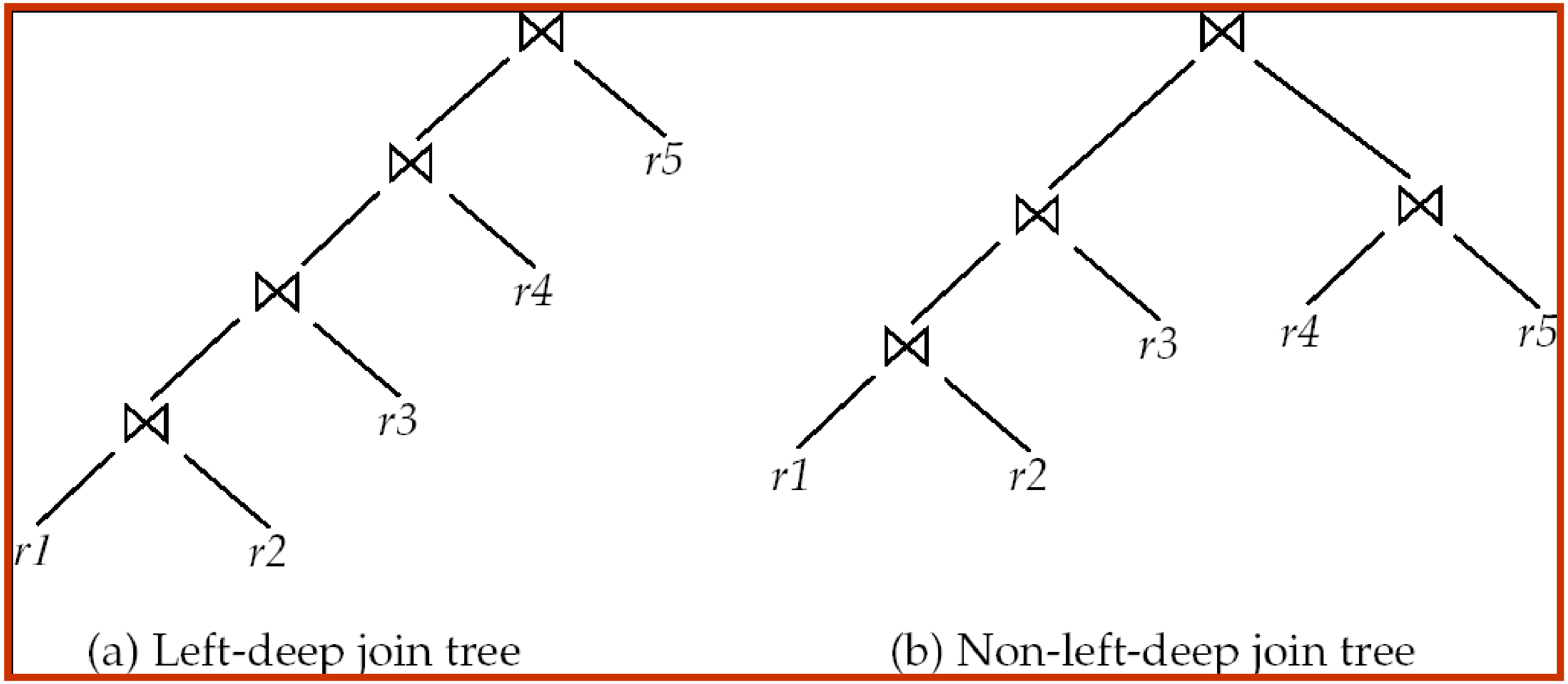
$A \bowtie B \bowtie C \bowtie D \bowtie E$
$A \bowtie (B \bowtie C \bowtie D \bowtie E)$
$A \bowtie (B \bowtie (C \bowtie D \bowtie E))$

$A \bowtie (B \bowtie (C \bowtie (D \bowtie E)))$ remember the best of two ways to
$A \bowtie (B \bowtie (C \bowtie (E \bowtie D)))$ represent $D \bowtie E$

$A \bowtie (B \bowtie ((D \bowtie E) \bowtie C))$ here we can use the precomputed
expressions for $D \bowtie E$ and
store the best of different ways to
represent $C \bowtie D \bowtie E$

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - Replace "**for each** non-empty subset $S1$ of S such that $S1 \neq S$"
    - By: **for each** relation r in S
      let S1 = S – r .
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which in turn may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing overall cost
  - Sort order may also be useful for order by and for grouping
- Not sufficient to find the best join order for each subset of the set of n given relations
  - must find the best join order for each subset, for each interesting sort order
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

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 (i.e. with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g. nested subqueries

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

Optimizing Nested Subqueries**

- Nested query example:

```
select customer_name
from borrower
where exists (select *
              from depositor
              where depositor.customer_name =
                            borrower.customer_name)
```

- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**

Optimizing nested subqueries

- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause
  - Such evaluation is called **correlated evaluation**
  - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery
- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques

Optimizing Nested Subqueries (Cont.)

- E.g.: earlier nested query can be rewritten as
  select *customer_name*
  from *borrower, depositor*
  where *depositor.customer_name* = *borrower.customer_name*
  - Note: the two queries generate different numbers of duplicates (why?)
    - Borrower can have duplicate customer-names
    - Can be modified to handle duplicates correctly as we will see
- In general, it is not possible/straightforward to move the entire nested subquery from clause into the outer level query from clause
  - A temporary relation is created instead, and used in body of outer level query

Optimizing Nested Subqueries (Cont.)

In general, SQL queries of the form below can be rewritten as shown

- Rewrite: **select** …
  from L_1
  where P_1 **and exists** (**select** *
  from L_2
  where P_2)
- To: **create table** t_1 as
  select distinct V
  from L_2
  where P_2^1

  select …
  from L_1, t_1
  where P_1 **and** P_2^2
  - P_2^1 contains predicates in P_2 that do not involve any correlation variables
  - P_2^2 reintroduces predicates involving correlation variables, with relations renamed appropriately
  - V contains all attributes used in predicates with correlation variables

Optimizing Nested Subqueries (Cont.)

- In our example, the original nested query would be transformed to

  create table t_1 **as**
      select distinct *customer_name*
      from *depositor*

  select *customer_name*
  from *borrower*, t_1
  where t_1.*customer_name* = *borrower.customer_name*

- The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation.**

Optimizing nested subqueries

- Decorrelation is more complicated when
  - the nested subquery uses aggregation, or
  - when the result of the nested subquery is used to test for equality, or
  - when the condition linking the nested subquery to the other
    query is **not exists**,
  - and so on.

Materialized Views**

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view
  create view *branch_total_loan*(*branch_name, total_loan*) **as**
  select *branch_name,* **sum**(*amount*)
  from *loan*
  group by *branch_name*
- Materializing the above view would be very useful if the total loan amount is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - **Changes to database relations are used to compute changes to the materialized view, which is then updated**
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Above methods are directly supported by many database systems
    - Avoids manual effort/correctness issues