CS222/CS122C: Principles of Data Management

UCI, Fall 2019
Notes #13

Query Optimization (System-R)

Instructor: Chen Li

System R Optimizer

- Impact:
  - Most widely used currently; works well for < 10 joins.
- Cost estimation: An approximate art at best (☺).
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- Plan Space: Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

Overview of Query Optimization

- *Plan:* *Tree of relational algebra ops annotated with the chosen algorithm for each op.*
  - Then at runtime, when an operator is `pulled' for its next output tuple, it `pulls' on its inputs and computes them.
- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find very best available plan. (Reality: Avoid picking one of the worst plans!)
- We will study the System R approach.

Query Blocks: Units of Optimization

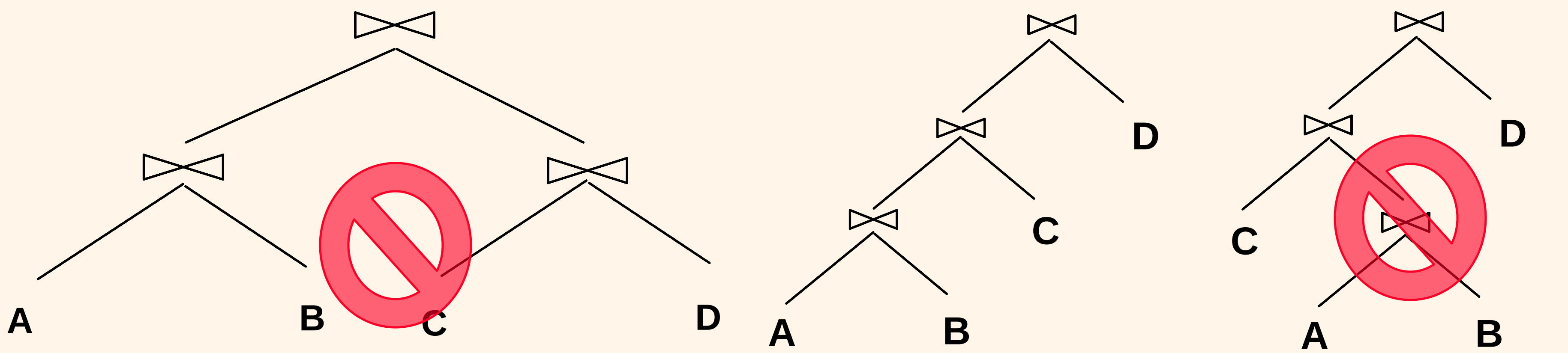
- An SQL query is parsed into a collection of *query blocks*, and they are optimized one block at a time.
- Nested blocks usually treated as calls to a subroutine, once per outer tuple. (Over-simplification, but will serve our purposes.)

```
SELECT  S.sname                    Outer
FROM  Sailors S                    block
WHERE  S.age IN
   (SELECT  MAX (age)              Nested
    FROM  Sailors                   block
    WHERE rating = S.rating)
```

- Query rewrite phase, before cost-based optimization phase, tries to “flatten” nested queries where it can (exposing joins).

For each block

- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - 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).



Relational Algebra Equivalences

- Allow us to choose different join orders and to \`push' selections and projections ahead of joins.
- *Selections*: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$ (*Cascade*)

  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (*Commute*)
- *Projections:* $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$ (*Cascade*)
- *Joins:* $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (*Associative*)

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

Show that: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

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 the cross-product to a join.
- A selection on just attributes of R commutes with R $\bowtie$ S. (i.e., σ (R $\bowtie$ S) $\equiv$ σ (R) $\bowtie$ S).
- Similarly, if a projection follows a join R $\bowtie$ S, we can push it down (earlier) by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.

Enumeration of Alternative Plans

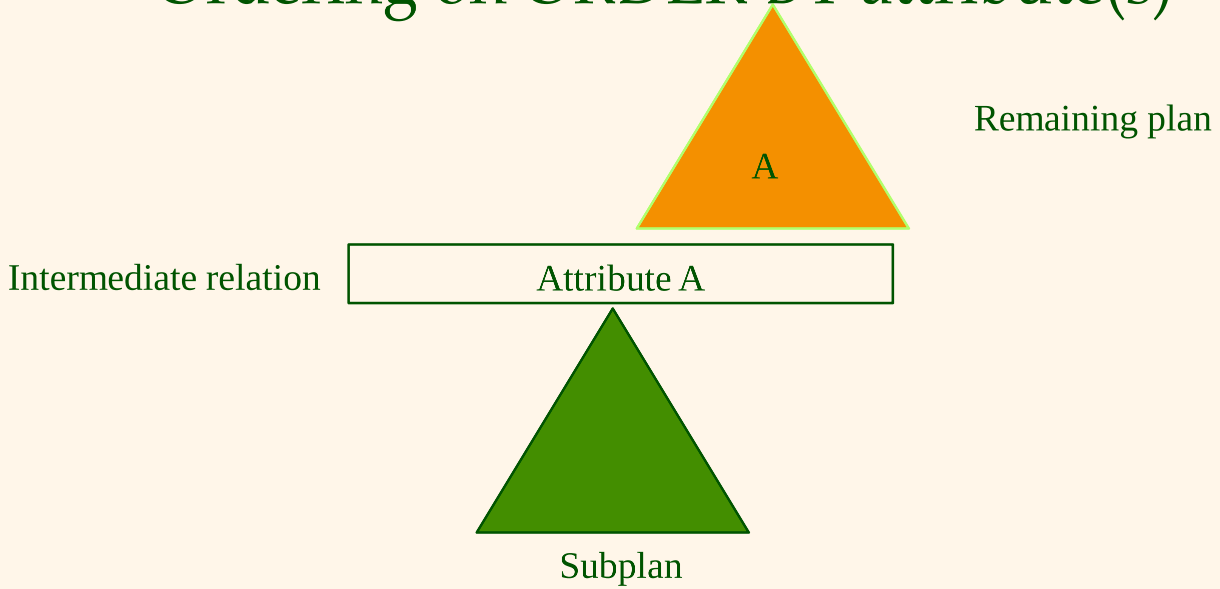
- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - 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).

Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method chosen for each join.
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - 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 result of a (N-1)-relation plan (as outer) to the N'th relation. *(All N-relation plans.)*
- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each ***interesting order*** of the tuples.

Interesting Orders

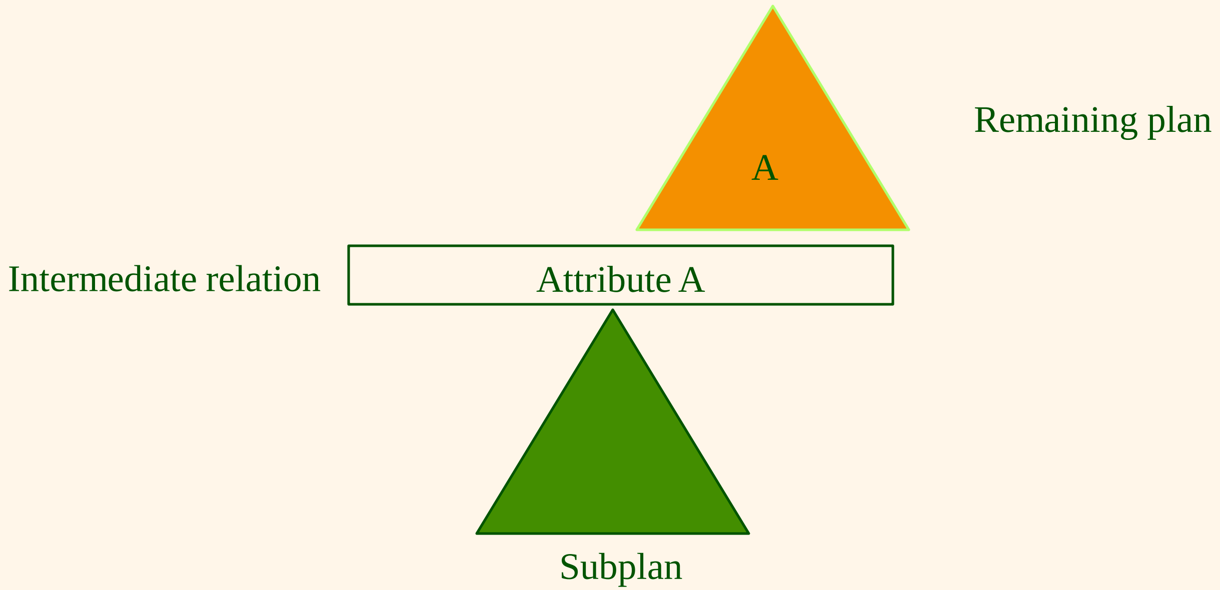
- A given data order is deemed "interesting" if it has the potential to save work (i.e., lower cost) later on.
  - Ordering on join attribute(s)
  - Ordering on GROUP BY attribute(s)
  - Ordering on DISTINCT attribute(s)
  - Ordering on ORDER BY attribute(s)



Plan pruning using interesting Orders

Plan X	Plan Y
Generates an interesting order on attribute A	No interesting order on attribute A

- If cost(X) < cost(Y): Keep plan X; remove Y.
- If cost(X) > cost(Y): keep both plans



Enumeration of Plans (Contd.)

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an "interestingly ordered" plan or via an additional sorting operator.
- An N-1 way plan will not be combined with an additional relation unless there is a join condition between them, unless *all* WHERE predicates have been used up.
  - i.e., avoid Cartesian products if possible!
- In spite of pruning plan space, this approach is still exponential in the # of tables.

Example

```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
GROUP BY S.sid
```

Available indexes:

S.Sid: B+ tree

R.sid: B+ tree; R.bid: B+ tree (clustered)

B.color: B+ tree

Example: Pass 1

```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
GROUP BY S.sid
```

- Access relation Sailors
  - (1) Scan
  - (2) B-tree (sid) scan: interesting order on sid
    - Sid is used in a later join
  - (3) B-tree search on an sid constant
  - If Cost of (1) < cost of (2): keep (1)
  - (2) is kept due to its interesting order on sid
  - (3) is kept since it is needed for an index-based NLJ
- Access relation Reserves
  - (1) B-tree scan on sid (interesting order)
  - (2) B-tree search on an sid constant
  - (3) B-tree scan on bid (interesting order)
  - (4) B-tree search on a bid constant
  - (1) and (3) are kept: interesting order
  - (2) and (4) are kept: needed for an index-based NLJ

Example: Pass 1

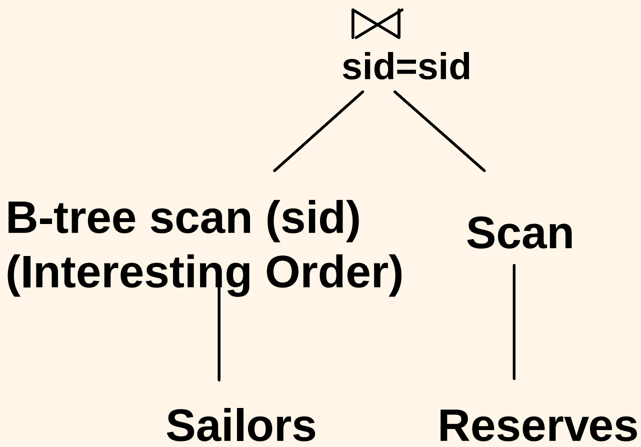
```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’
GROUP BY S.sid
```

- Access relation Boats
  - (1) Scan
  - (2) B-tree on “color” using “color = red “;
  - If Cost of (2) < cost of (1): keep (2) only

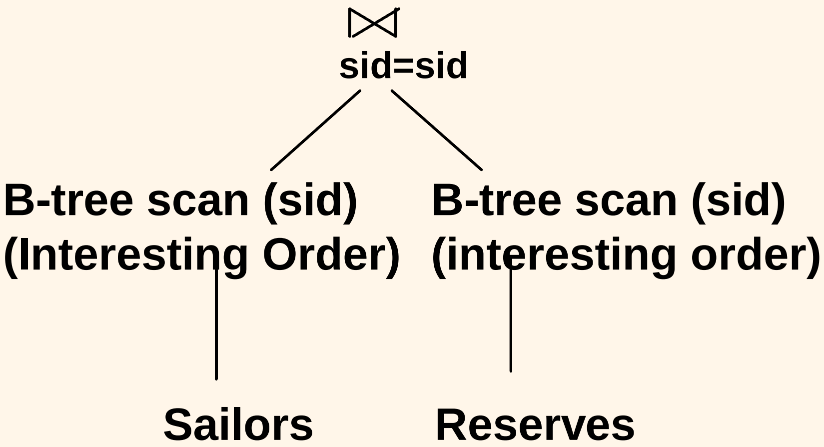
Example: Pass 2, join S and R

```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
GROUP BY S.sid
```

- For each access method of R, for each access method of S, consider all possible join methods:



1) Sort R, join S and R using sort-merge (benefits of interesting order of S)
2) Hash join
3) …



1) Sort-merge
2) Hash join
3) …

Example: Pass 2, join R and S

```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
GROUP BY S.sid
```

- Do the same analysis for R join S
- For all the plans, choose the cheapest ones with interesting orders

Example: Pass 2, consider (R,B)

```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’
GROUP BY S.sid
```

- Do the same analysis for R join B and B join R
- Ignore (S, B) since it’s a cross product

Example: Pass 3, consider (S,R,B)

```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
GROUP BY S.sid
```

- Consider ((S,R) join B), since we only consider left-deep trees
  - For "(S,R)", consider the best plans from pass 2 (with their interesting orders)
  - Consider various ways to join B
- Consider ((R,B) join S)
- Choose the best plans with interesting orders

Example: Pass 4, consider GROUP BY

```
SELECT S.sid, count(*)
FROM  Sailors S, Reserves R, Boats B
WHERE R.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’
GROUP BY S.sid
```

- For each plan from (S, R, B), consider different GROUP BY plans
- Pick the best one as the FINAL plan!

System-R Optimizer summary

- Left-deep trees only
- Avoid Cartesian products
- All access paths considered, cheapest chosen.
- Push selection down as much as possible
- Deal with GROUP BY/Aggregation/Order by at the end
- Keep "interesting order" for each plan
- Use dynamic programming to do plan enumeration

Summary

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