Relational Query Optimization

Yanlei Diao

Slide Content Courtesy of R. Ramakrishnan, J. Gehrke, and J. Hellerstein

Overview of Query Evaluation

- *Query evaluation plan*: tree of *relational algebra* operators, with choice of algorithm for each operator.

- *Query optimization*: given a query, many plans are possible
  - Ideally, find the most efficient plan.
  - In practice, avoid worst plans in practice.

Outline of topics

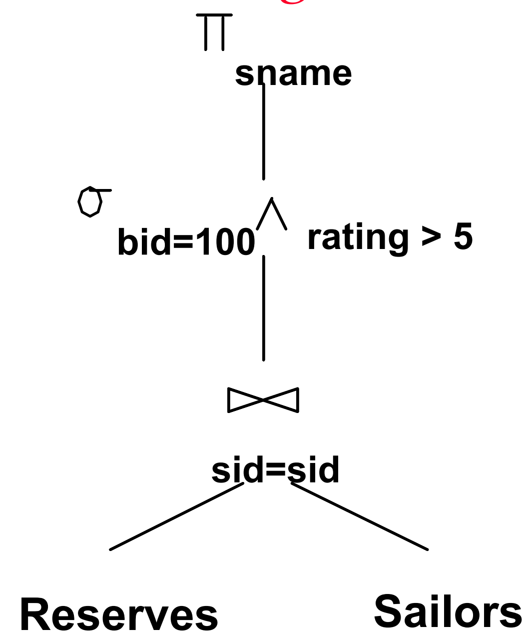
- Query plans and equivalences

- Query optimization issues
  - Plan space
  - Cost estimation
  - Plan search

- Handling nested queries

Relational Algebra Tree

Relational Algebra Tree:



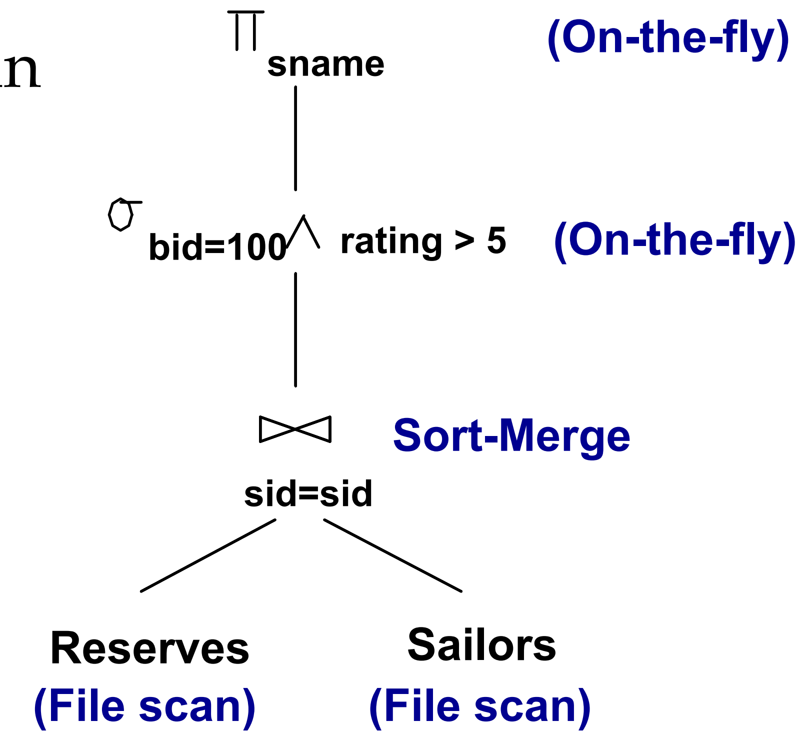

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid=S.sid AND
        R.bid=100 AND S.rating>5
```

Expression in Relational Algebra (RA):

$\pi_{sname}\ (\alpha_{bid=100 \wedge rating>5}\ (\text{Reserves} \bowtie_{sid=sid} \text{Sailors}))$

Query Evaluation Plan

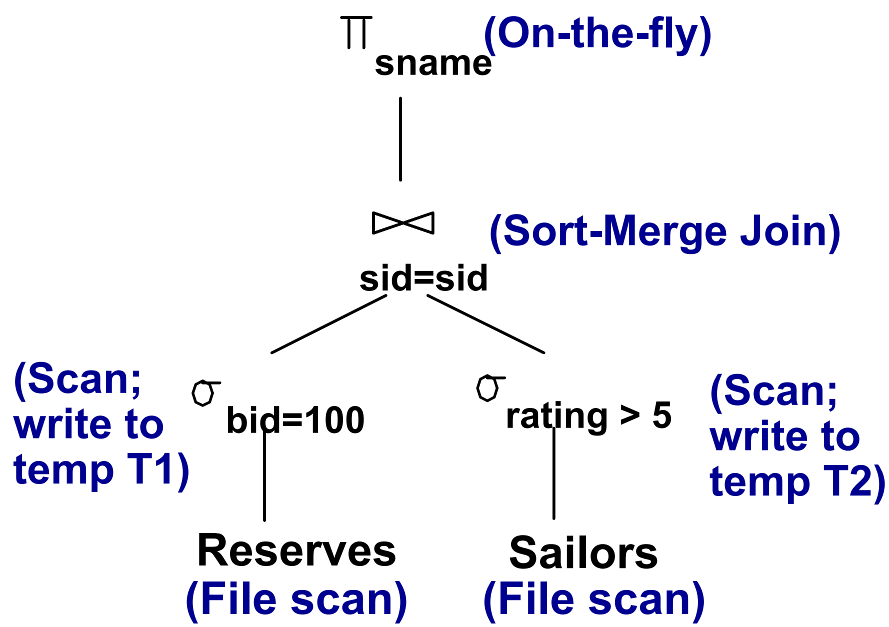
- *Query evaluation plan* extends an RA tree with:
  - *access method* for each relation;
  - *implementation method* for each other operator.



- What are the missed opportunities?
  - Selections could have been \`pushed' earlier.
  - Use of indexes.
  - More efficient joins.

Query Plan 1 (Selection Pushed Down)

- *Push selections below the join.*
- *Materialization* vs. *Pipelining*:
  - Store a temporary relation T, if the subsequent join needs to *scan T multiple times*.
  - The opposite is pipelining.



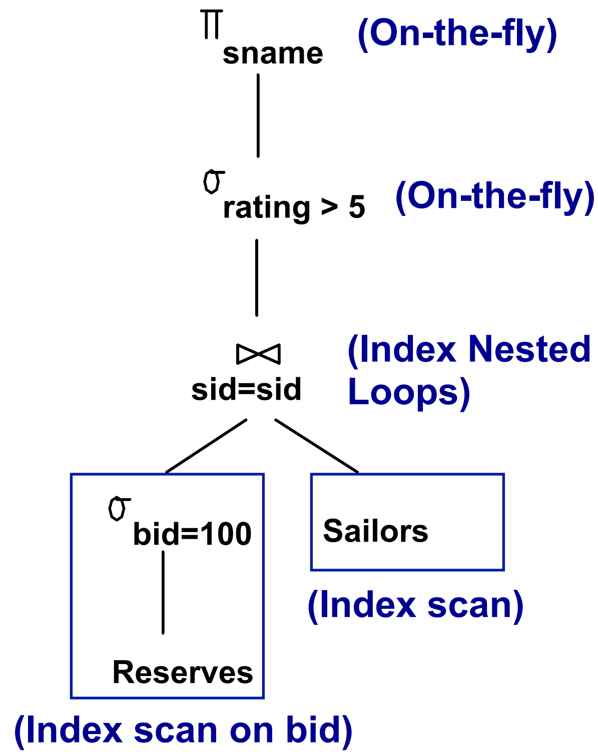
- With <u>5 buffer pages</u>, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort-Merge join: Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250).
  - Total = 4060 page I/Os.

Indexes

- A **tree** index *matches* (a conjunction of) terms if the attributes in the terms form a *prefix* of the search key.
  - Tree index on <*a, b, c*>
  - *a*=5 *AND b*=3 ?
  - *a*=5 *AND b*>6 ?
  - *b*=3 ?

Query Plan 3 (Using Indexes)

- *Selection using index*: clustered index on *bid* of Reserves.
  - Retrieve 100,000/100 = 1000 tuples
  - Clustering: read 1000/100 = 10 pages.
- Indexed NLJ: *pipeline* the outer and *index lookup* on sid of Sailors.
  - The outer: no need to materialize.
  - The inner: *sid* is a *key*; *at most one* match tuple, unclustered index OK.



- Cost:
  - Selection of Reserves tuples (~10 I/Os).
  - For each tuple, get matching Sailor tuple (1000*(2~3)).
  - Total = 2010~3010 I/Os.

Outline

- Query plans and equivalences
- Query optimization issues
  - Plan space
  - Cost estimation
  - Plan search
- Handling nested queries

Three Main Issues in Optimization

- Given a query block, three main optimization issues:
  - *Plan space*: what plans are considered?
  - *Plan cost*: what is the cost of a given plan?
  - *Search algorithm*: how do we search the plan space for the cheapest estimated plan?

(1) *Cost Estimation*

- For each plan considered, must estimate its cost.

- Estimate *cost* of each operation in a plan tree:
  - Depends on input cardinalities.
  - Depends on the method (sequential scan, index scan, join...)

- Estimate *size of result* for each operation in tree:
  - Use statistics about input relations.
  - Estimate the *reduction factors* of predicates: *reduction factor (RF) or selectivity* of each *predicate* reflects the impact of the *term* in reducing result size.

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

Statistics in System Catalog

- Statistics about each relation (R) and index (I):
  - Relation cardinality: # tuples (NTuples) in *R*
  - Relation size: # pages (NPages) in *R*

  - Index cardinality: # distinct values (NKeys) in *I*
  - Index size: # pages (INPages) in *I*
  - Index height: # nonleaf levels (IHeight) of *I*
  - Index range: low/high key values (Low/High) in *I*

  - Number of distinct values in an attribute (NKeys)
  - Histogram for an attribute

Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - *Cost of lookup* = *Height(I)+1 for a B+ tree*, ≈ *1.2 for hash index*
  - *Cost of record retrieval* = *1*
- Clustered index I matching one or more selections:
  - *Cost of lookup* + *product of RF's of matching terms (RF-terms) * (INPages'(I)+NPages(R))*
- Non-clustered index I matching one or more selections:
  - *Cost of lookup* + *RF-terms * INPages'(I)* + *min(RF-terms * NTuples(R), NPages(R))*
- Sequential scan of file: *NPages(R)*
- May add extra costs for GROUP BY and duplicate elimination (if a query says DISTINCT)

Equiwidth Histograms

Equiwidth: buckets of equal size.

Frequency	8/3	4/3	15/3	3/3	15/3
Counts	8	4	15	3	15
Buckets	0 1 2	3 4 5	6 7 8	9 10 11	12 13 14

Still not accurate for value 14: 5/45

Equidepth Histograms

Equidepth: equal counts of buckets.

Small errors for infrequent items: tolerable.

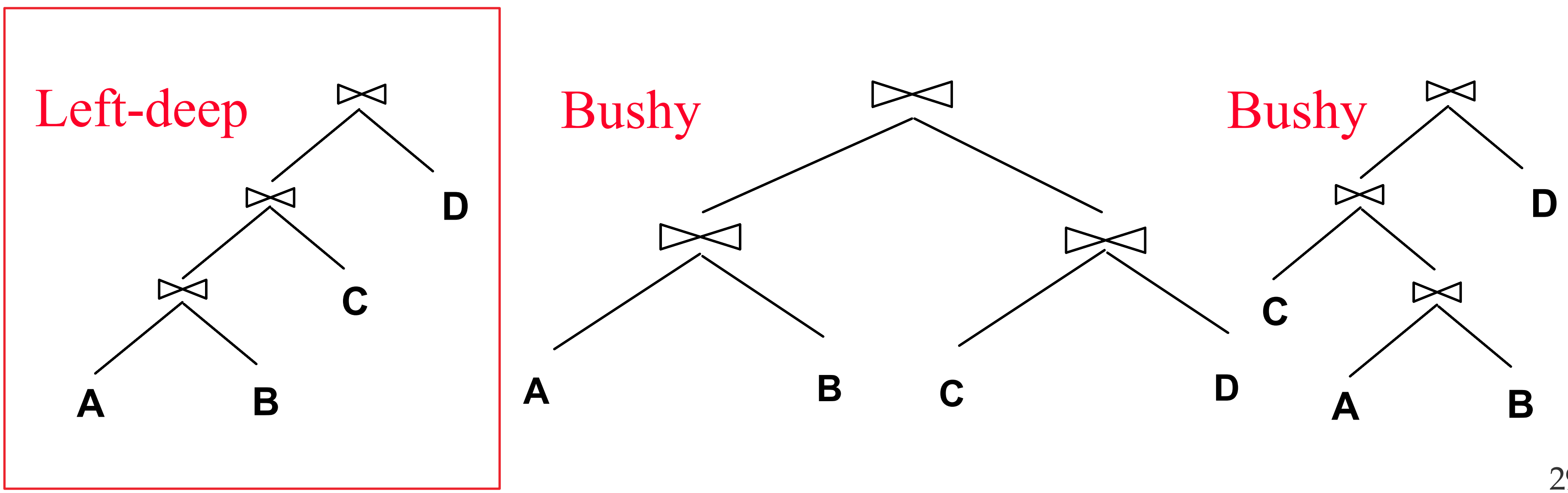
Frequency	9/4	10/4	10/2	7/4	9/1
Counts	9	10	10	7	9
Buckets	0 1 2 3	4 5 6 7	8 9	10 11 12 13	14

Now accurate for value 14: 9/45

- Favors *frequent* values.
- Implementation:
  - Boundaries of 5 buckets {0, 4, 8, 10, 14, 14}
  - Count of tuples for each bucket
  - Number of distinct values for each bucket

(2) *Plan Space*

- For each query block, the plans considered are:
  - All *access methods*, for each reln in the FROM clause.
  - All *left-deep join trees*: all the ways to join the relns one-at-a-time, with the inner reln in the FROM clause.
    - All permutations of N relns: **N factorial**!



Plan Space

- For each block, the plans considered are:
  - All *access methods*, for each reln in FROM clause.
  - All *left-deep join trees*: all the ways to join the relns one-at-a-time, with the inner reln in the FROM clause.
    - All permutations of N relns: N factorial!
    - But avoid **Cartesian products**!

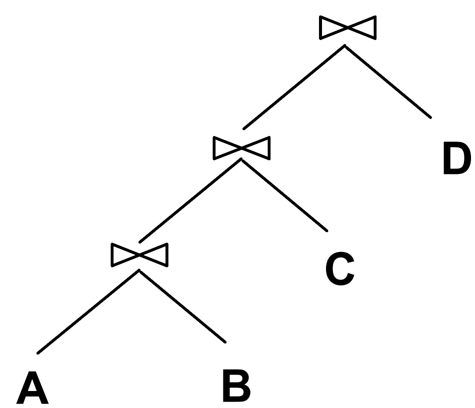
      Join R, S, T w. R.a = S.a and S.b = T.b, how many left-deep trees?
  - All *join methods*, for each join in the tree.
  - Appropriate *places for selections and projections*.

(3) Plan Search

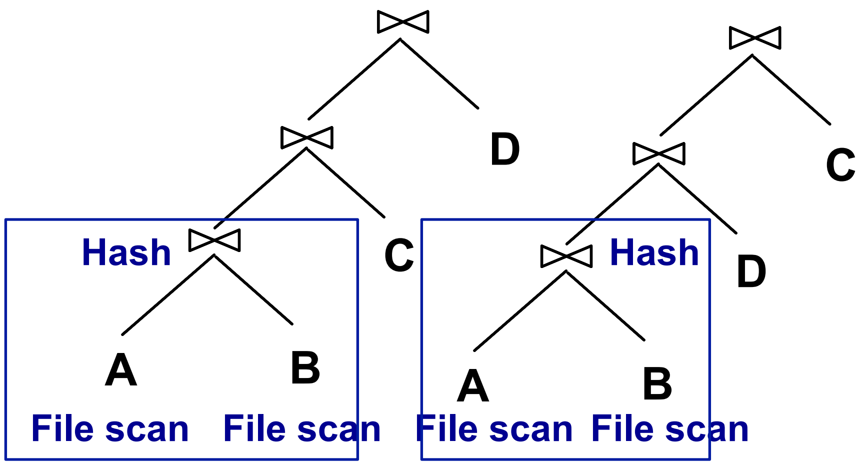
- As the number of joins increases, the number of alternative plans grows rapidly.

- System R: (1) use *only left-deep join trees*, (2) avoid ***Cartesian products***.
  - Allow *pipelined* plans; intermediate results not written to temporary files.
  - Not all left-deep trees are fully pipelined!
    - Sort-Merge join: at least sorting phase
    - Two-phase hash join: partitioning phase

Left-deep



Search Algorithm



- Left-deep join plans :
  - Differ in the *order* of relations, *access method* for each relation, *join method* for each join.
  - But may share common prefixes, so don't enumerate all. Instead use
- Dynamic Programming

"a method for solving problems that exhibit the properties of *overlapping subproblems* and *optimal substructures*"

  - Find the best plans to access A, B, C, D individually
  - Find the best plans for joining A-B, A-C, A-D, B-A, B-C, B-D, C-A, C-B, C-D, D-A, D-B, D-C; store the best for (A-B), (A-C), (A-D), (B-C), (B-D), (C-D)
  - Repeat this for 3 relation sets…
  - Repeat this for 4 relation sets…
  - **This procedure is revised if given specific join predicates of A,B,C,D (i.e., left deep trees but avoiding Cartesian products).**

An Example Star Schema

- Dynamic Programming

  "a method for solving problems that exhibit the properties of *overlapping subproblems* and *optimal substructures*"

  - Find the best plans to access A, B, C, D individually
  - Find the best plans for joining A-B, A-C, A-D, B-A, C-A, D-A,; store the best for (A-B), (A-C), (A-D)
  - Repeat this for 3 relation sets…
  - Repeat this for 4 relation sets…