Relational Query Optimization

Kathleen Durant PhD
Lecture 19
CS 3200
Northeastern University

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

Overview of Query Evaluation

- *Query Evaluation Plan*: *tree of relational algebra (R.A.) operators, with choice of algorithm for each operator.*
- Three main issues in query optimization:
  - *Plan space*: for a given query, what plans are considered?
    - Huge number of alternative, semantically equivalent plans.
  - *Plan cost*: how is the cost of a plan estimated?
  - *Search algorithm*: search the plan space for the cheapest (estimated) plan.
- Ideally: Want to find best plan. Practically: Avoid worst plans!

Representation of a SQL Command

```
SELECT          {DISTINCT} <list of columns>
FROM            <list of relations>
{WHERE          <list of "Boolean Factors">}
{GROUP BY <list of columns>
    {HAVING     <list of Boolean Factors>}}
{ORDER BY <list of columns>};
```

- Query Semantics:

1. Take Cartesian product (a.k.a. cross-product) of relns in FROM, projecting only those columns that appear in other clauses
2. If a WHERE clause exists, apply all filters in it
3. If a GROUP BY clause exists, form groups on the result
4. If a HAVING clause exists, filter groups with it
5. If an ORDER BY clause exists, make sure output is in the right order
6. If there is a DISTINCT modifier, remove duplicates

Basics of Query Optimization

```
SELECT        {DISTINCT} <list of columns>
FROM          <list of relations>
{WHERE        <list of "Boolean Factors">}
{GROUP BY <list of columns>
{HAVING       <list of Boolean Factors>}}
{ORDER BY <list of columns>};
```

- Convert selection conditions to *conjunctive normal form* (CNF):
  - (*day<8/9/94 OR bid=5 OR sid=3*) *AND* (*rname='Paul' OR sid=3*)
  - Why not disjunctive normal form?
- Interleave FROM and WHERE into a plan tree for optimization.
- Apply GROUP BY, HAVING, DISTINCT and ORDER BY at the end, pretty much in that order.

System Catalog

- System information: buffer pool size and page size.
- For each relation:
  - relation name, file name, file structure (e.g., heap file)
  - attribute name and type of each attribute
  - index name of each index on the relation
  - integrity constraints…
- For each index:
  - index name and structure (B+ tree)
  - search key attribute(s)
- For each view:
  - view name and definition

System Catalog (Contd.)

- Statistics about each relation (R) and index (I):
  - Cardinality: # tuples (NTuples) in *R*.
  - Size: # pages (NPages) in *R*.
  - Index Cardinality: # distinct key 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*.
  - More detailed info. (e.g., histograms). More on this later…
- Statistics updated periodically.
  - Updating whenever data changes is costly; lots of approximation anyway, so slight inconsistency ok.
- Intensive use in query optimization! Always keep the catalog in memory.

Schema for Examples

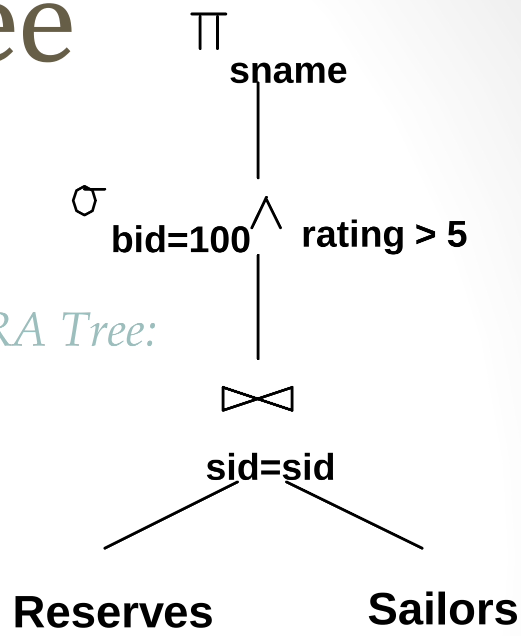
Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long
  - 100 tuples per page
  - 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long
  - 80 tuples per page
  - 500 pages.

Relational Algebra Tree

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

RA Tree:



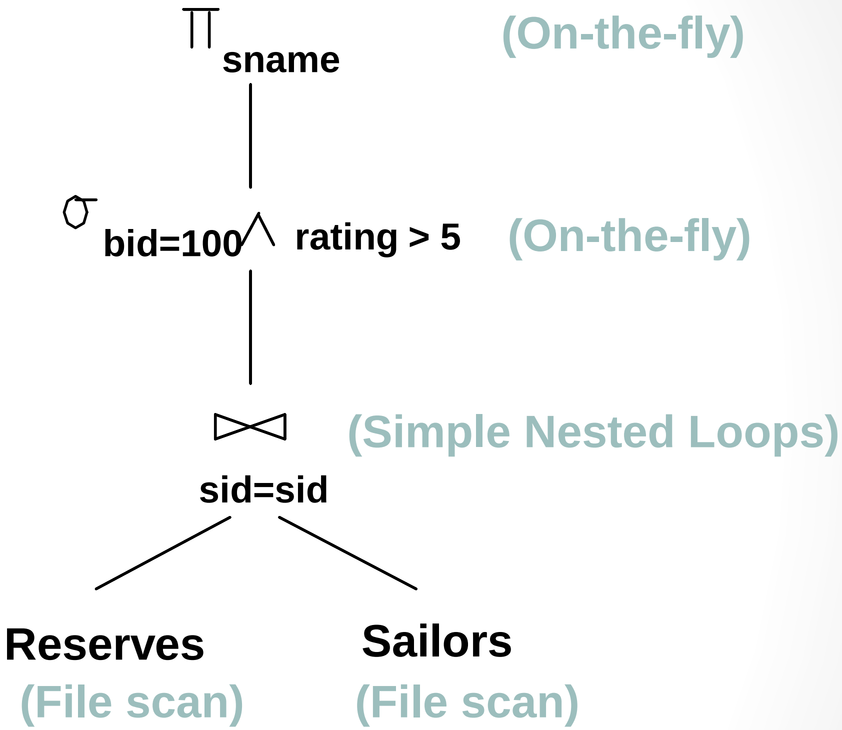
Expression in Relational Algebra (RA):

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

- The algebraic expression partially specifies how to evaluate the query:
  - Compute the natural join of Reserves and Sailors
  - Perform the selections
  - Project the *sname* field

Query Evaluation Plan

- *Query evaluation plan* is an extended RA tree, with additional annotations:
  - *access method* for each relation;
  - *implementation method* for each relational operator.
- Cost: 500+500*1000 I/Os
- Misses several opportunities:
  - Selections could have been `pushed' earlier.
  - No use is made of any available indexes.
  - More efficient join algorithm…



Equivalence Rules

- Conjunctive selection operations can be deconstructed into a sequence of individual selections

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

- Selection operations are commutative

$$\sigma_{c1 \wedge cn}(R) \equiv \sigma_{c2}(\ldots \sigma_{c1}(R))$$

- Only the last in a sequence of projection operations is needed, the others can be omitted

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

- Selections can be combined with Cartesian Products and joins

$$\sigma_{c1}(R_1 \text{ x } R_2) \equiv (R_1 \rhd\lhd_{c1} R_2)$$

Equivalence Rules

- Projection operation distributes over the join operation
  - If project involves only attributes L1 U L2

$$\pi_{L1 \cup L2}(R1 \bowtie_c R2) \equiv (\pi_{L1}(R1)) \bowtie_c (\pi_{L2}(R2))$$

- Set operations are commutative

$$(R1 \cup R2) \equiv (R2 \cup R1)$$

- Set operations are associative

$$(R1 \cup R2) \cup R3 \equiv R1 \cup (R2 \cup R3)$$

- Selection operation distributes over set operations
- Projection operation distributes over union

More Equivalence Rules

- A projection π commutes with a selection σ that only uses attributes retained by π, i.e., $\pi_a(\sigma_c(R)) = \sigma_c(\pi_a(R))$.
- Selection between attributes of the two relations of a cross-product converts cross-product to a join, i.e., $\sigma_c(R \times S) = R \bowtie_c S$
- A selection on attributes of R commutes with $R \bowtie S$,

  i.e., $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$.
- Similarly, if a projection follows a join $R \bowtie S$, we can `push' it by retaining only attributes of R (and S) that are (1) needed for the join or (2) kept by the projection.

Relational Algebra Equivalences

- Allow us (1) choose different join orders and to (2) `push' selections and projections ahead of joins.

Selections:

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

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

Projections:

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

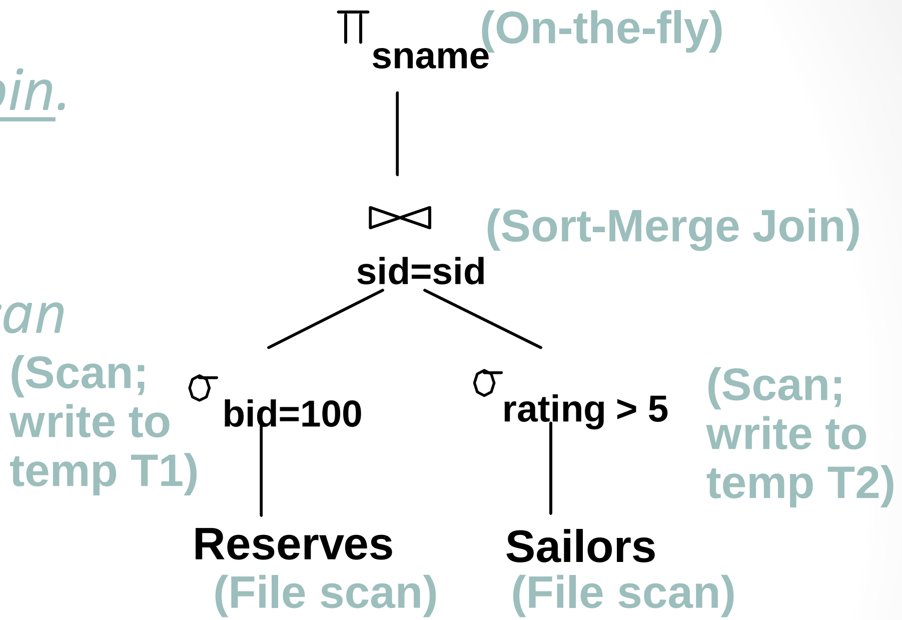
Joins:

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

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

Alternative Plan 1 (Selection Pushed Down)

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



- With 5 buffers, 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*3*250), merge (10+250), total = 3560 page I/Os.
  - Block Nested Loop Join: join cost = 10+4*250, total cost = 2770.

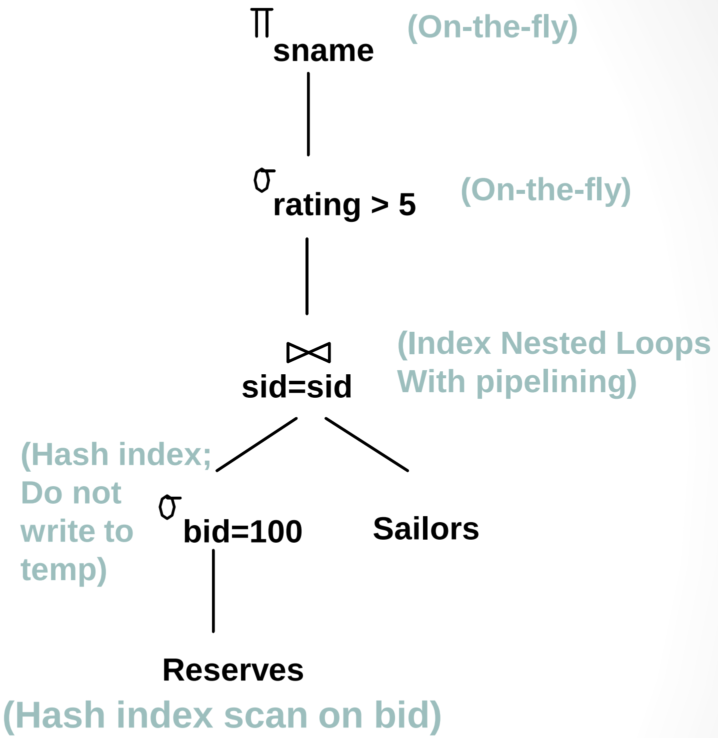
    250 page relation, 10 page relation, blocksize = 4

Access Methods

- An *access method (path)* is a method of retrieving tuples:
  - File scan, or index scan with the search key matching a selection in the query.
- A tree index *matches* (a conjunction of) terms if the attributes in the terms form a *prefix* of the search key.
  - E.g., Tree index on <*a, b, c*> matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.
- A hash index *matches* (a conjunction of) terms if there is a term *attribute = value* for *every* attribute in the search key of the index.
  - E.g., Hash index on <*a, b, c*> matches *a=5 AND b=3 AND c=5*; but it does not match *b=3, or a=5 AND b=3, or a>5 AND b=3 AND c=5*.

Alternative Plan 2 (Using Indexes)

- *Selection using index*: clustered index on *bid* of Reserves.
  - Retrieve 100,000/100 = 1000 tuples in 1000/100 = 10 pages.
- Indexed NLJ: *pipelining* the outer and *indexed lookup* on the inner.
  - The outer: scanned only once, pipelining, no need to materialize.
  - The inner: ioin column *sid* is a *key* for Sailors; *at most one* matching tuple, unclustered index on *sid* OK.



- Push *rating*>5 before the join? Need to use search arguments More on this later…
- Cost: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Pipelined Evaluation

- *Materialization*: Output of an *op* is saved in a temporary relation for uses (multiple scans) by the next op.
- *Pipelining*: No need to create a temporary relation. Avoid the cost of writing it out and reading it back. Can occur in two cases:
  - *Unary operator*: when the input is pipelined into it, the operator is applied *on-the-fly*, e.g. selection on-the-fly, project on-the-fly.
  - *Binary operator*: e.g., the outer relation in indexed nested loops join.

Iterator Interface for Execution

- A query plan, i.e., a tree of relational ops, is executed by calling operators in some (possibly interleaved) order.
- *Iterator Interface* for simple query execution:
  - Each operator typically implemented using a uniform interface: *open*, *get_next*, and *close*.
  - Query execution starts top-down (*pull-based*). When an operator is `pulled' for the next output tuples, it
    1. `pulls' on its inputs (opens each child node if not yet, gets next from each input, and closes an input if it is exhausted),
    2. computes its own results.
- *Encapsulation*
  - Encapsulated in the operator-specific code: access methods, join algorithms, and materialization vs. pipelining…
  - Transparent to the query executer.

Highlights of System R Optimizer

- Impact: most widely used; works well for < 10 joins.
- Cost of a plan: approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate <u>*cost of operations*</u> and <u>*result sizes*</u>.
  - Considers combination of CPU and I/O costs.
- Plan Space: too large, must be pruned.
  - Only considers the space of *left-deep plans*.
    - Left-deep plan: a tree of joins in which the inner is a base relation.
    - Left-deep plans naturally support <u>*pipelining*</u>.
  - Avoids cartesian products!
- Plan Search: dynamic programming (prunes useless subtrees).

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.

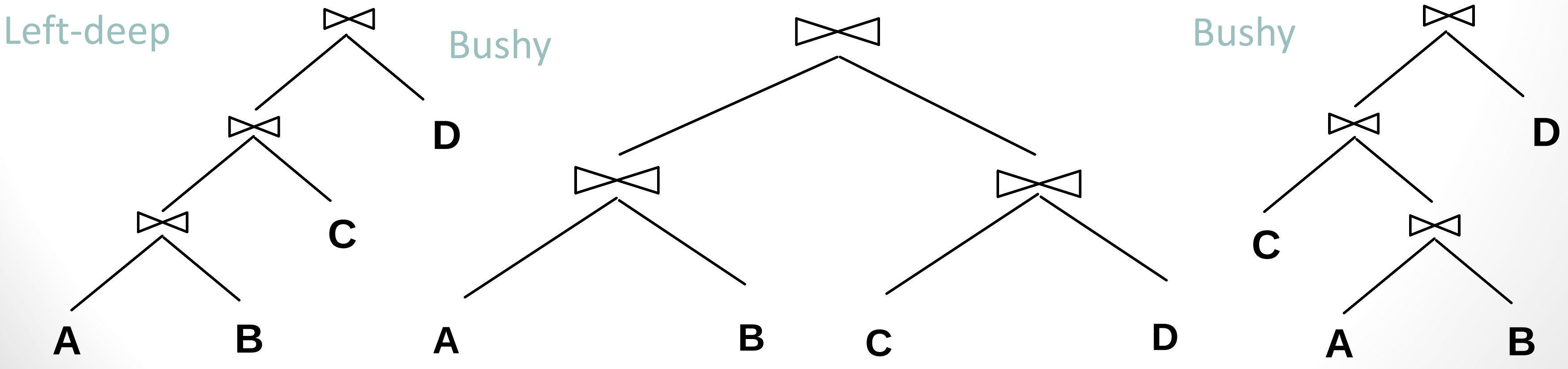
```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age IN
    (SELECT  MAX (S2.age)
     FROM  Sailors S2
     GROUP BY  S2.rating)
```

Outer block   *Nested block*

- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (More discussion later.)

Plan Space

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



Plan Space

- For each block, the plans considered are:
  - All available access methods, for each relation in FROM clause.
  - All *left-deep join trees*: all the ways to join the relations one-at-a-time, with the inner relation in the FROM clause.
    - Considering all permutations of N relations, N factorial!
    - But avoid cartesian products
    - e.g. R.a = S.a and R.b = T.b, how many left-deep trees?
  - All join methods, for each join in the tree.
  - Appropriate places for selections and projections.

Cost Estimation

- For each plan considered, must estimate its cost.
- Estimate *cost* of each operation in a plan tree:
  - Depends on input cardinalities.
  - We've discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
- Estimate *size of result* for each operation in tree:
  - Use information about the input relations.
  - For selections and joins, assume independence of predicates and uniform distribution of values.

Statistics in System Catalog

- Statistics about each relation (R) and index (I):
  - Cardinality: # tuples (NTuples) in *R*.
  - Size: # pages (NPages) in *R*.
  - Index Cardinality: # distinct key 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*.
  - More detailed info. (e.g., histograms). More on this later…

Size Estimation & Reduction Factors

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

- Consider a query block:
- *Reduction factor (RF) or Selectivity* of each *term*:
  - *Assumption 1*: *uniform distribution of the values*!
  - Term *col=value*: RF = *1/NKeys(I),* given index I on *col*
  - Term *col>value*: RF = *(High(I)-value)/(High(I)-Low(I))*
  - Term *col1=col2*: RF = *1/MAX(NKeys(I1), NKeys(I2))*
- *Max. number of tuples in result* = the product of the cardinalities of relations in the FROM clause.
- *Result cardinality* = Max # tuples * product of all RF's.
  - *Assumption 2*: *terms are independent*!

Queries over a Single Relation

- Queries over a single relation can consist of selection, projection, and aggregation.
- *Enumeration of alternative plans*:
  1. Each available access path (file/index scan) is considered, the one with least estimated cost is chosen.
  2. The various operations are often carried out together:
     - If an index is used for a selection, projection is done for each retrieved tuple.
     - The resulting tuples can be *pipelined* into the aggregate computation in the absence of GROUP BY; otherwise, hashing or sorting is needed for GROUP BY.

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 + (INPages'(I)+NPages(R)) * product of RF's of matching selections.* (*Treat INPages' as the number of leaf pages in the index.*)
- Non-clustered index I matching one or more selections:
  - *Cost of lookup + (INPages'(I)+NTuples(R)) * product of RF's of matching selections.*
- Sequential scan of file:
  - *NPages(R).*
- May add extra costs for GROUP BY and duplicate elimination in projection (if a query says DISTINCT).

Example

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

- If we have an index on *rating* ($1 \leq$ rating ≤ 10):
  - NTuples(R) /NKeys(I) = 40,000/10 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages'(I)+NPages(R)) = (1/10) * (50+500) pages retrieved, plus lookup cost.
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40,000) pages retrieved, plus lookup cost.
- If we have an index on *sid*:
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.
- Doing a file scan:
  - We retrieve all file pages (500).

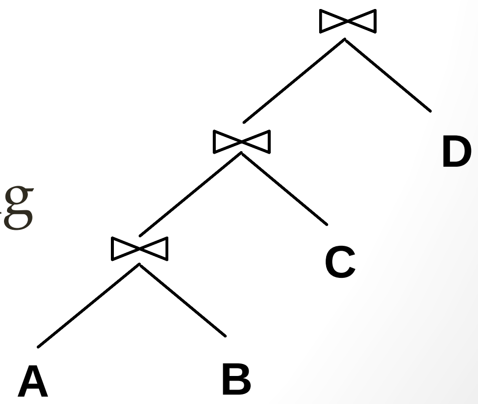
Queries Over Multiple Relations

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

❖ System R: (1) use *only left-deep join trees, where the* inner is a base relation, (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 (the sorting phase)
    - Two-phase hash join (the partitioning phase)

Left-deep



Place Search Algorithm

- Left-deep join plans differ in:
  - the order of relations,
  - the access path for each relation, and
  - the join method for each join.
- Many of these plans share common prefixes, so don't enumerate all of them. This is a job for…
- Dynamic Programming

"a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructure that takes much less time than naive methods."

Enumeration of Left-Deep Plans

- *Enumerate using N passes* (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation. Include index scans available on “SARGable” predicates.
    - SARGable – search argument able (can use an index).
    - SARGable operations would include =,>,<, BETWEEN, and some LIKE conditions. Non-SARGable operations would include <>,!=,NOT IN, OR, other LIKE conditions.
  - Pass 2: Find best ways to join result of each 1-relation plan (as *outer*) to another relation. *(All 2-relation plans.)*
  - …
  - Pass N: Find best ways 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 *unordered* plan, and
  - cheapest plan for each *interesting order* of the tuples, and discard all others.

Enumeration of Plans (Contd.)

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered' plan or an additional sorting operator.
- A *k*-way (*k*<*N*) plan is not combined with an additional relation unless there is a join condition between them.
  - Do it until all predicates in WHERE have been used up.
  - That is, avoid Cartesian products if possible.
- In spite of pruning plan space, still creates an exponential number of plans.

Cost Estimation for Multi-relation Plans

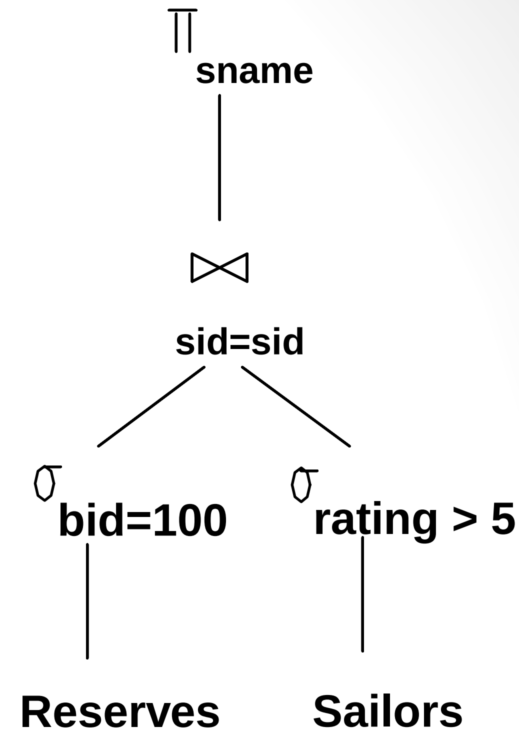
```
SELECT  attribute list
FROM  relation list
WHERE  term1 AND ... AND termk
```

- Consider a query block:
- *Reduction factor (RF)* is associated with each *term*.
- *Max number tuples in result* = the product of the cardinalities of relations in the FROM clause.
- *Result cardinality* = max # tuples * product of all RF's.
- Multi-relation plans are built up by joining one new relation at a time.
  - Cost of join method, plus estimate of join cardinality gives us both cost estimate and result size estimate.

Example: Pass 1

Sailors:
B+ tree on *rating*
Hash on *sid*

Reserves:
B+ tree on *bid*



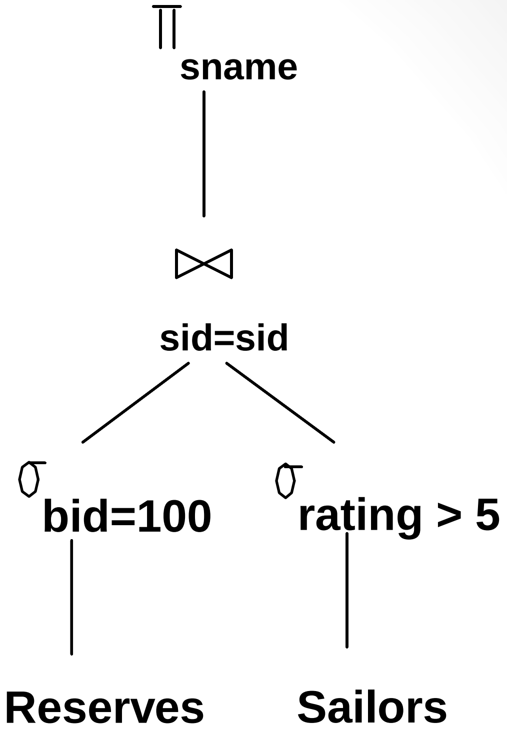
Pass 1

- *Sailors*:
  - B+ tree matches *rating>5*, and is probably cheapest.
  - However, if this selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper.
  - Still, B+ tree plan kept (because tuples are in *rating* order).
- *Reserves*: B+ tree on *bid* matches *bid=100*; cheapest.

Example: Pass 2

<u>Sailors:</u>
B+ tree on *rating*
Hash on *sid*

<u>Reserves:</u>
B+ tree on *bid*



<u>Pass 2</u>

- Consider each plan retained from Pass 1 as the *outer*, and consider how to join it with the (only) other relation.
- *Reserves as outer*: Hash index can be used to get Sailors tuples that satisfy *sid* = outer tuple's *sid* value.
    - rating > 5 is a *search argument* pushed to the index scan on Sailors.

System R: Limitation 1

- Uniform distribution of values:
  - Term *col=value* has RF *1/NKeys(I),* given index I on *col*
  - Term *col>value* has RF *(High(I)-value)/(High(I)-Low(I))*
- Often causes highly inaccurate estimates
  - E.g., distribution of gender: male (40), female (4)
  - E.g. distribution of age:

    0 (2), 1 (3), 2 (3), 3 (1), 4 (2), 5 (1), 6 (3), 7 (8), 8 (4), 9 (2), 10 (0), 11 (1), 12 (2), 13 (4), 14 (9). NKeys=15, count = 45.

    Reduction factor of age=14: 1/15? 9/45!
- *Histogram*: approximates a data distribution

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: equal counts of buckets favoring frequent values

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

Small errors for infrequent items: tolerable.

Now accurate for value 14: 9/45

System R: Limitation 2

- Predicates are independent:
  - *Result cardinality* = max # tuples * product of Reduction Factors of matching predicates.
- Often causes highly inaccurate estimates
  - E.g., Car DB: 10 makes, 100 models. RF of make='honda' and model='civic' >> than 1/10 * 1/100!
- Multi-dimensional histograms [PI'97, MVW'98, GKT'00]
  - Maintain counts and frequency in multi-attribute space.
- Dependency-based histograms [DGR'01]
  - Learn dependency between attributes and compute conditional probability P(model='civic' | make='honda')
  - Can use graphical models…

Nested Queries With No Correlation

- *Nested query (block)*: a query that appear as an operand of a predicate of the form "expression operator query".
- *Nested query with no correlation*: the nested block does not contain a reference to tuple from the outer.
  - A nested query needs to be evaluated *only once*.
  - The optimizer arranges it to be evaluated before the top level query.

```
SELECT  S.sname
FROM    Sailors S
WHERE  S.rating >
   (SELECT  Avg(rating)
    FROM  Sailors)
```

```
  (SELECT  Avg(rating)
   FROM  Sailors)
```

↓

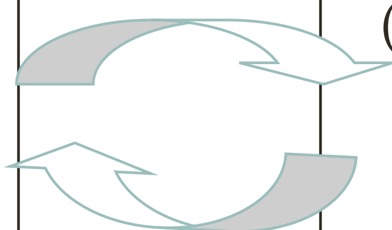
```
SELECT  S.sname
FROM    Sailors S
WHERE  S.rating > value
```

Nested Queries With Correlation

- Nested query with correlation: the nested block contains a reference to a tuple from the outer.
  - Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
  - The nested block is executed using ***nested iteration***, a tuple-at-a-time approach.

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

```
SELECT  S.sname
FROM    Sailors S
WHERE  EXISTS
   ( …)
```



Nested block to optimize:

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

Query Decorrelation

- Implicit ordering of nested blocks means *nested iteration* only.
- The equivalent, non-nested version of the query is typically optimized better, e.g. *hash join* or *sort-merge*.
- *Query decorrelation* is an important task of optimizer.

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

Equivalent non-nested query:

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

Query Decorrelation (Contd.)

- *Guideline: Use only one "query block", if possible.*

```
SELECT DISTINCT  *
FROM    Sailors S
WHERE S.sname IN
           (SELECT Y.sname
             FROM YoungSailors Y)
```

=

```
SELECT DISTINCT S.*
FROM    Sailors S,
             YoungSailors Y
WHERE S.sname = Y.sname
```

❖ *Not always possible ...*

```
SELECT *
FROM    Sailors S
WHERE S.sname IN
           (SELECT DISTINCT Y.sname
             FROM YoungSailors Y)
```

≠

```
SELECT S.*
FROM    Sailors S,
             YoungSailors Y
WHERE S.sname = Y.sname
```

Summary

- Query optimization is an important task in 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.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Summary (Contd.)

- Single-relation queries:
  - All access paths considered, cheapest is chosen.
  - *Issues*: Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order.
- Multiple-relation queries:
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
  - Next, for each 2-relation plan that is `retained', all ways of joining another relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only best plan for each interesting order of tuples is `retained'.

Rewriting SQL Queries

- Complicated by interaction of:
  - NULLs, duplicates, aggregation, sub-queries.
- ***Guideline**: Use only one "query block", if possible*

```
SELECT DISTINCT *
  FROM Sailors S
 WHERE S.sname IN
      (SELECT Y.sname
         FROM YoungSailors Y)
```

=

```
SELECT DISTINCT S.*
  FROM Sailors S,
       YoungSailors Y
 WHERE S.sname = Y.sname
```

❖ *Not always possible …*

```
SELECT *
  FROM Sailors S
 WHERE S.sname IN
      (SELECT DISTINCT Y.sname
         FROM YoungSailors Y)
```

≠

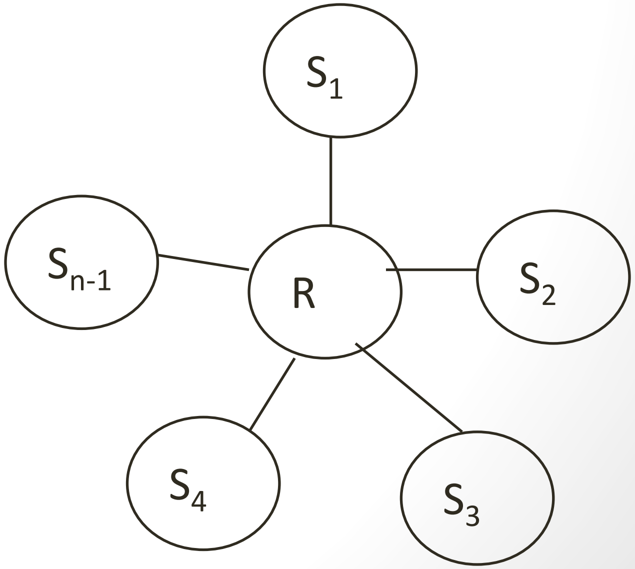
```
SELECT S.*
  FROM Sailors S,
       YoungSailors Y
 WHERE S.sname = Y.sname
```

Summary: Unnesting Queries

- DISTINCT at top level: *Can ignore duplicates*.
  - Can sometimes infer DISTINCT at top level! (e.g. subquery clause matches at most one tuple)
- DISTINCT in subquery w/o DISTINCT at top: *Hard to convert*.
- Subqueries inside OR: *Hard to convert.*
- ALL subqueries: *Hard to convert.*
  - EXISTS and ANY are just like IN.
- Aggregates in subqueries: *Tricky.*
- Good news: Some systems now rewrite under the covers (e.g. DB2).

Complexity of Plan Search

- Enumeration of all left-deep plans for an n-way join:
  $O(n!)$, where $n! \sim \sqrt{2\pi n}\,(\frac{n}{e})^n$ with a large n.
- Plans considered in System R:
  $O(2^{n-1})$, which occurs with a star join graph
  - $R.a_1 = S_1.a_1$
  - $R.a_2 = S_2.a_2$
  - …
  - $R.a_{n-1} = S_{n-1}.a_{n-1}$



Complexity with a Star Graph

- Total number of plans considered:
  - Pass 2: (n-1 choose 1) 2-relation subsets,
    for each subset, pick one as the outer reln in the join
    (best plan for the inner has been chosen in the previous pass).
  - Pass 3: (n-1 choose 2) 3-relation subsets,
    for each subset, pick one as the outer.
  - …
  - Pass n: (n-1 choose n-1) n-relation subsets,
    for each subset, pick one as the outer.
  - Total number of plans = $O(n \cdot 2^{n-1})$
- Maximum number of plans stored in a pass?