Relational Query Optimization

CMPSCI 445

Fall 2008

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.

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 plan space for cheapest (estimated) plan.
- Ideally: Want to find best plan. Practically: Avoid worst plans!

Basics of Query Optimization

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

- Selection conditions are first converted to *conjunctive normal form* (CNF):
  - (*day*<*8/9/94 OR bid*=*5 OR sid*=*3*) *AND* (*rname*=*'Paul' OR sid*=*3*)
- Query optimization interleaves FROM and WHERE into a plan tree.
- GROUP BY, HAVING, DISTINCT and ORDER BY are applied 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 attributes
- For each view:
  - view name and definition

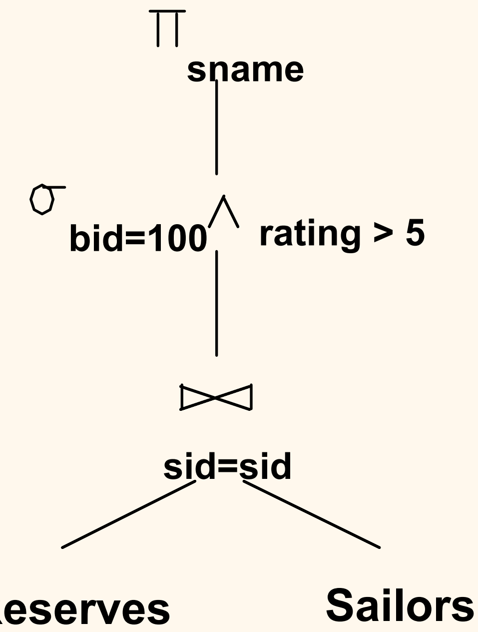
System Catalog (contd.)

- 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.

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:

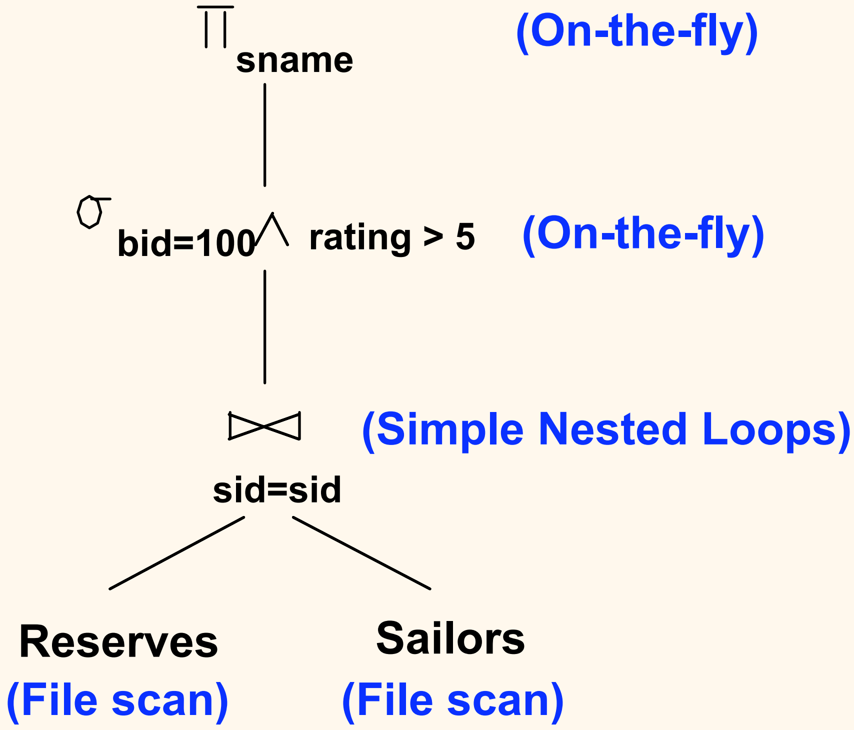


$\pi_{sname}\ (\alpha_{bid=100 \wedge rating>5}\ (\text{Reserves} \bowtie_{sid=sid}\ \text{Sailors}))$

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

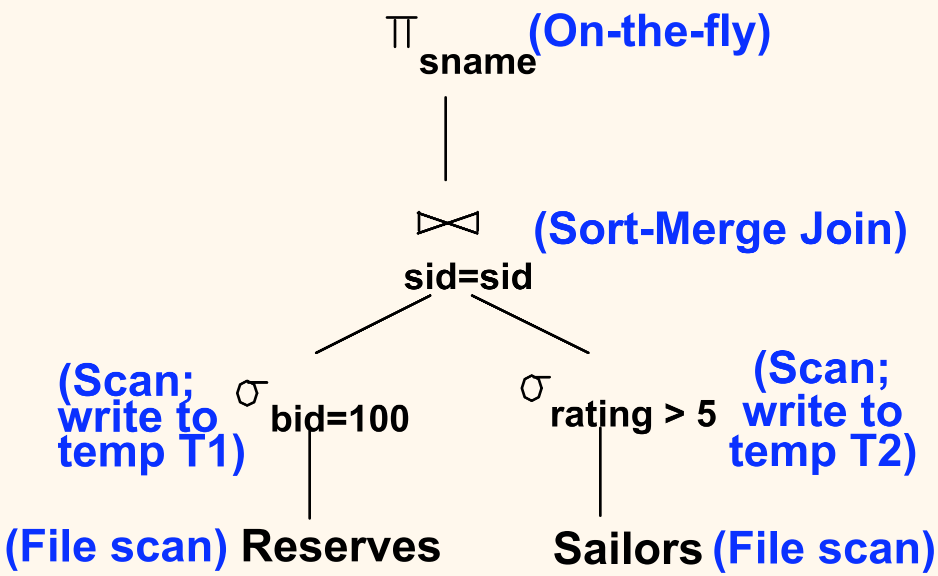
Query Evaluation Plan

- 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, etc.



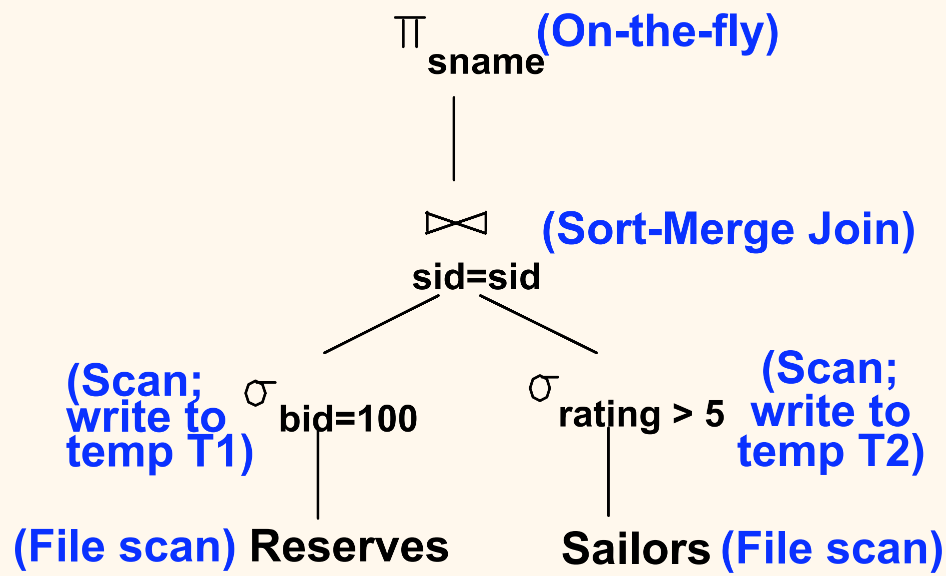
Alternative Plan 1 (No Indexes)

- Main difference: *push selections below the join.*
- *Materialization*: store a temporary relation T, if the subsequent join needs to scan T multiple times.
- With 5 buffers, cost of plan:



Alternative Plan 1 (No Indexes)

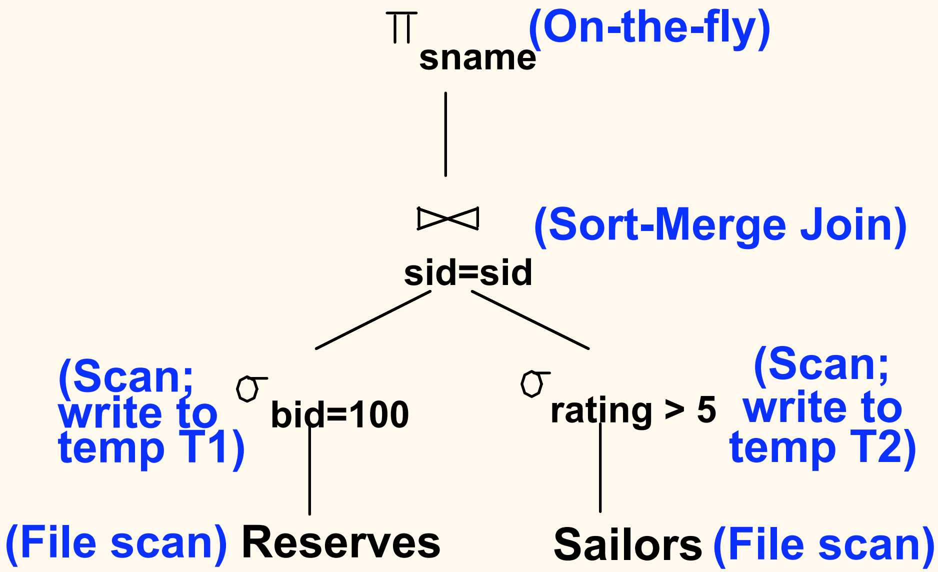
- Main difference: *push selections below the join.*
- *Materialization*: store a temporary relation T, if the subsequent join needs to scan T multiple times.
- With 5 buffers, cost of plan:



  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).

Alternative Plan 1 (No Indexes)

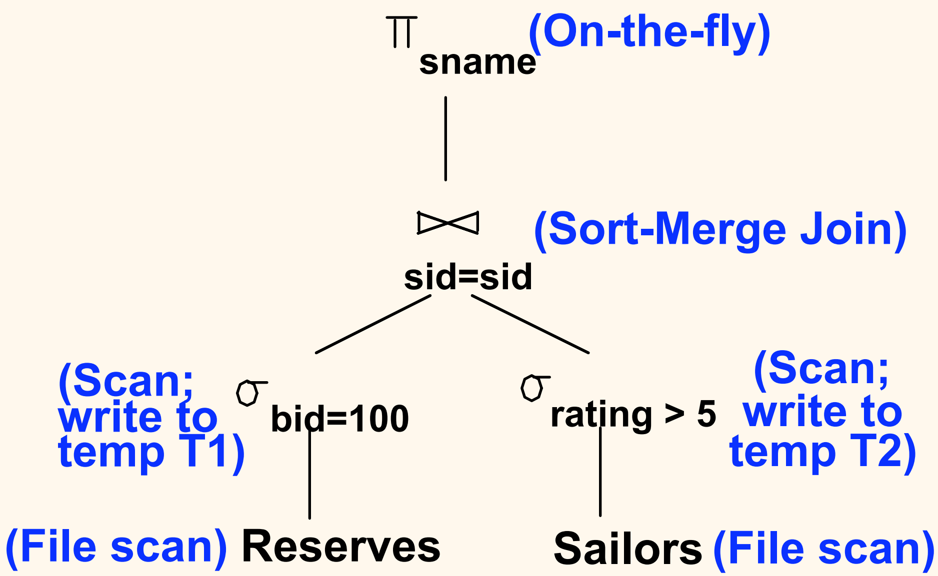
- Main difference: *push selections below the join.*
- *Materialization*: store a temporary relation T, if the subsequent join needs to scan T multiple times.
- With 5 buffers, cost of plan:



  - 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).

Alternative Plan 1 (No Indexes)

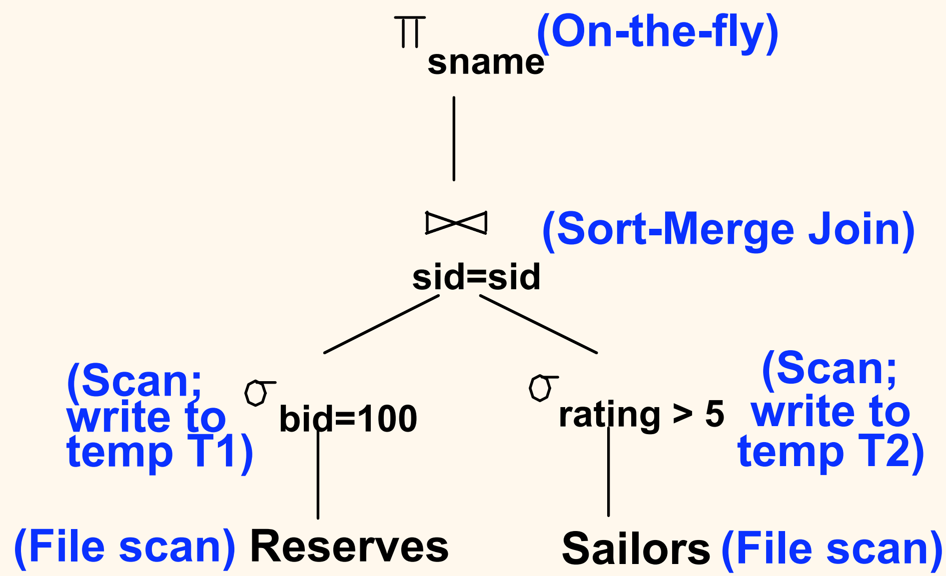
- Main difference: *push selections below the join.*
- *Materialization*: store a temporary relation T, if the subsequent join needs to scan T multiple times.
- With 5 buffers, cost of plan:



  - 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:

Alternative Plan 1 (No Indexes)

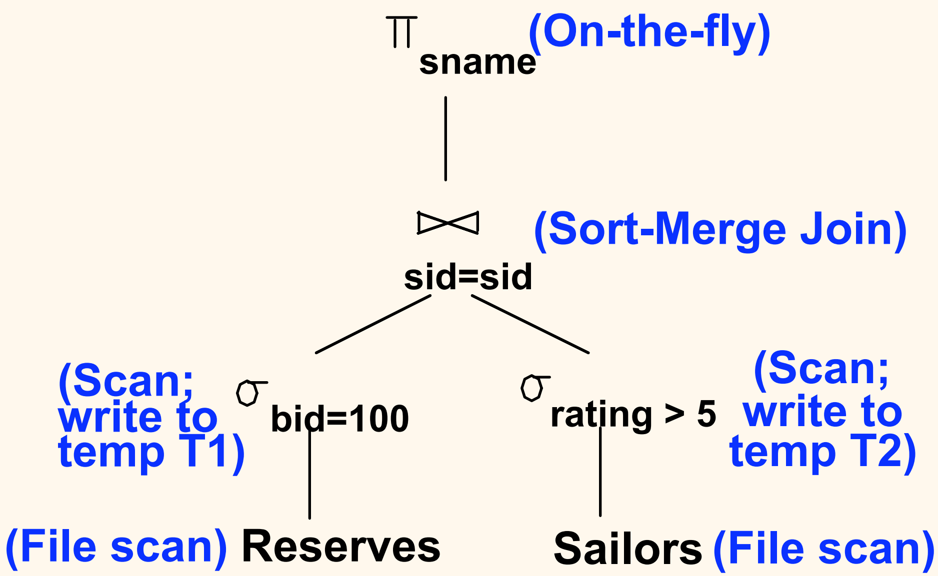
- Main difference: *push selections below the join.*
- *Materialization*: store a temporary relation T, if the subsequent join needs to scan T multiple times.
- With 5 buffers, cost of plan:



  - 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 I/Os.

Alternative Plan 1 (No Indexes)

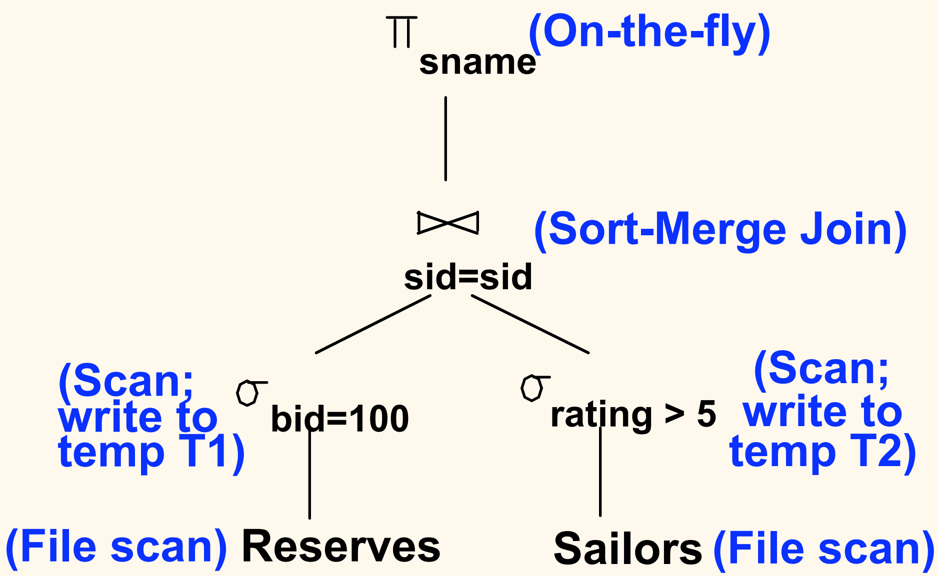
- Main difference: *push selections below the join.*
- *Materialization*: store a temporary relation T, if the subsequent join needs to scan T multiple times.
- With 5 buffers, cost of plan:



  - 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 I/Os.
  - BNL join:

Alternative Plan 1 (No Indexes)

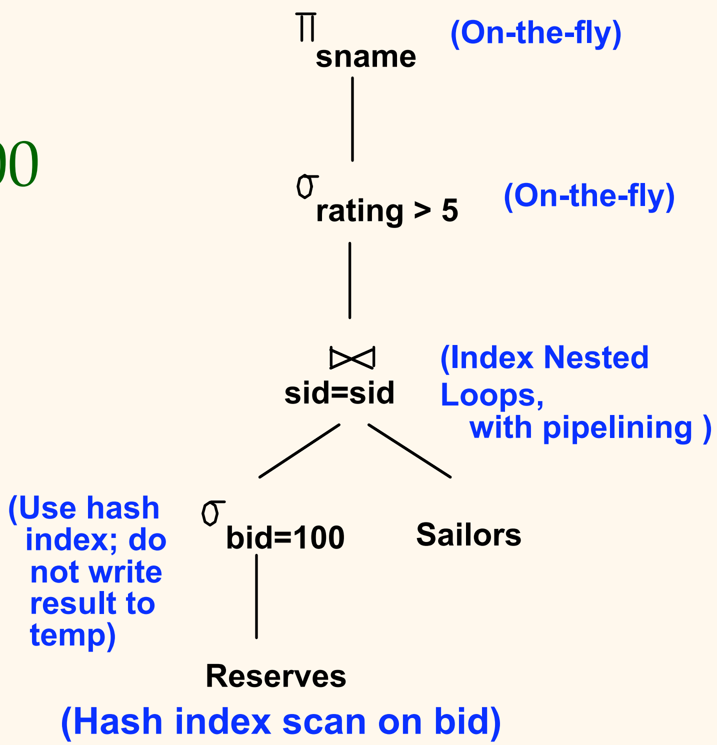
- Main difference: *push selections below the join.*
- *Materialization*: store a temporary relation T, if the subsequent join needs to scan T multiple times.
- With 5 buffers, cost of plan:



- 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 I/Os.
- BNL join:
  - join cost = 10+4*250, total cost = 2770.

Alternative Plan 2 (With Indexes)

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



- Decision not to push *rating*>5 before the join is based on availability of *sid* index on Sailors.
- 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 processing by the next op.
- *Pipelining*: Doesn't create a temporary relation, saves the cost of writing it out and reading it back. Can occur in two cases:
  - Unary operator: when the input is pipelined into it, say 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), and (2) computes its own results.
- Encapsulation: different access methods, join algorithms, and materialization vs. pipelining all encapsulated in the operator-specific code, transparent to the query executer.

Highlights of System R Optimizer

- Impact: Most widely used; works well for < 10 joins.
- Cost estimation: 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 plan: a tree of joins in which the inner is a base relation.
    - Left-deep plans naturally support *pipelining*.
  - Cartesian products avoided.
- 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*

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.

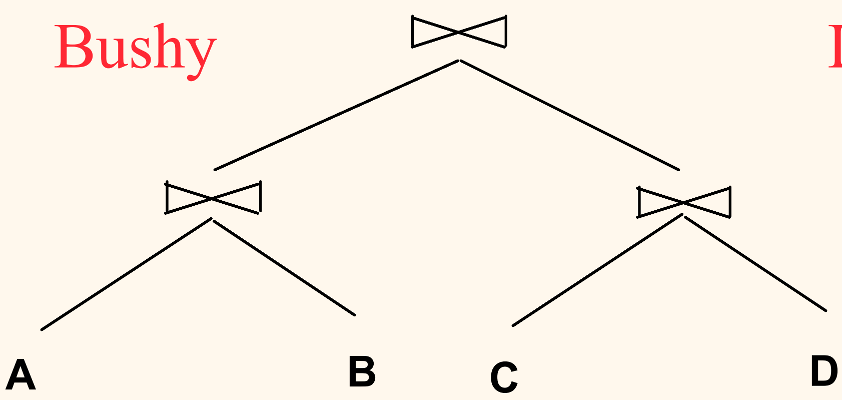
```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age IN
    (SELECT  MAX (S2.age)
     FROM  Sailors S2
     GROUP BY  S2.rating)
```

Outer block   *Nested block*

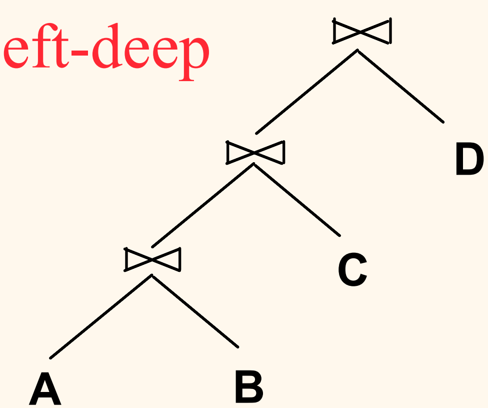
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (More discussion later.)

Plan Space

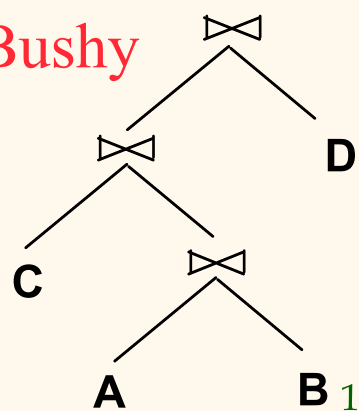
Bushy



Left-deep

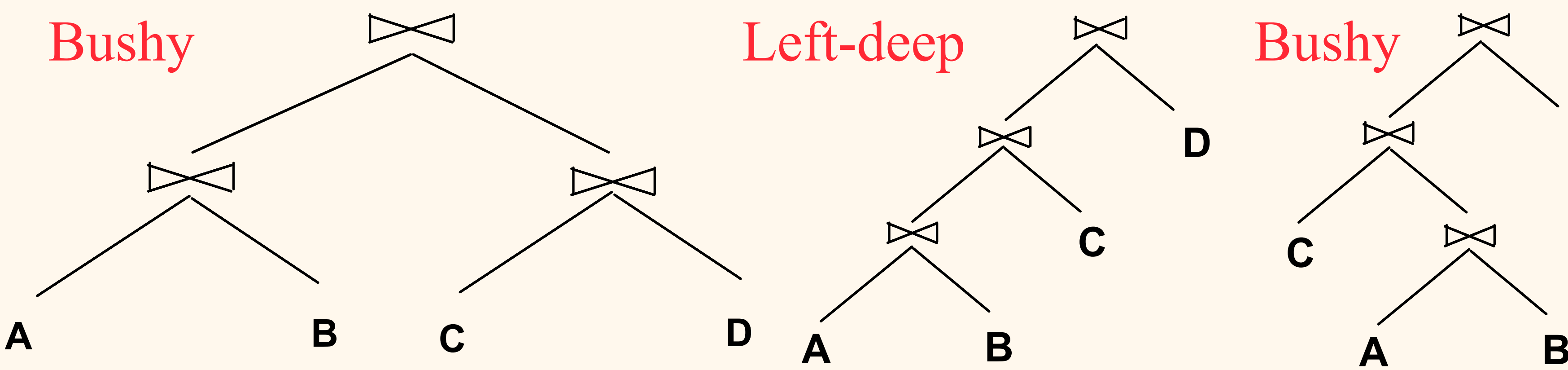


Bushy



Plan Space

- For each block, the plans considered are:
  - <u>All available access methods</u>, for each reln in FROM clause.
  - <u>All *left-deep join trees*</u>: i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations.
  - <u>All join methods</u>, for each join in the tree.
  - <u>Appropriate places for selections and projections</u>.



Relational Algebra Equivalences

- Allow us to (1) choose different join orders and to (2) `push' selections and projections ahead of joins.
- *Selections*: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$ *(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: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

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.
  - For selections and joins, assume independence of predicates.

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

- Consider a query block:

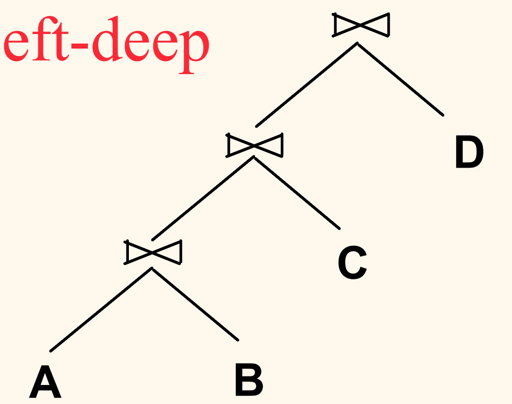
```
SELECT  attribute list
FROM  relation list
WHERE  term1 AND ... AND termk
```

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

Queries Over Multiple Relations

- As the number of joins increases, the number of alternative plans grows rapidly; *need to restrict the search space.*
- System R: *only left-deep join trees.*
  - Left-deep tree: inner is a base relation.
  - They 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).

Left-deep



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
  - Distribution of gender: male (40), female (4)
  - 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?
- Histogram: approximates a data distribution

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

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

System R: Limitation 2

- Assumption that predicates are independent.
  - *Result cardinality* = Max # tuples * product of each predicate's Reduction Factor.
- Often causes highly inaccurate estimates
  - Car DB: 10 makes, 100 models. Reduction factor of make='honda' and model='civic' is much higher than 1/10 * 1/100!
- Multi-dimensional histograms [PI'97, MVW'98, GKT'00]
- Dependency-based histograms using graphical models [DGR'01]
- …

Nested Queries With No Correlation

- A query may appear as an operand of a predicate of the form "expression operator query"; the query is called a *nested query* (*block*).
- *Nested query with no correlation*: the nested block does not contain a reference to tuple from the outer.
- Without correlation, 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)
```

Nested block to optimize:

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

Query Decorrelation

- Implicit ordering of these blocks means *nested iteration* only; some good strategies are not considered.
- The equivalent, non-nested version of the query is typically optimized better, e.g. *hash join* or *sort-merge*.
- Important task of optimizer: ***query decorrelation***!

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

<u>Equivalent non-nested query:</u>

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

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.
    - 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.