Query Optimization

Schema for Examples

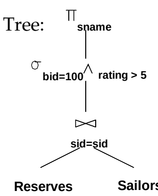
Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string)

- Similar to old schema; *rname* added for variations.
- 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.

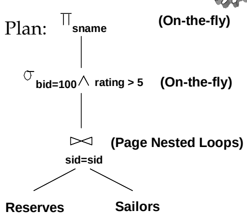
Motivating Example

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

RA Tree:

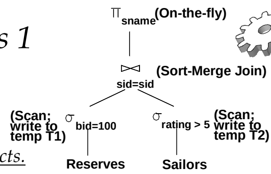


Plan:



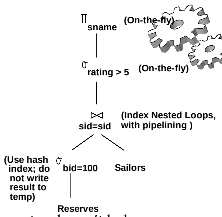
- Cost: 500+500*1000 I/Os
- By no means the worst plan!
- But can do better (how?)

Alternative Plans 1 (No Indexes)



- ***Main difference:*** *<u>push selects.</u>*
- 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 T1 (2*2*10), sort T2 (2*3*250), merge (10+250)
  - Total: 3560 page I/Os.
- If we used BNL join, join cost = 10+4*250, total cost = 2770.
- If we `push' projections, T1 has only *sid*, T2 only *sid* and *sname*:
  - T1 fits in 3 pages, cost of BNL drops to under 250 pages, total < 2000.

Alternative Plans 2 With Indexes



- With clustered index on *bid* of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- INL with ***<u>pipelining</u>*** (outer is not materialized).
  - –Projecting out unnecessary fields from outer doesn't help.
- 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.

Overview of Query Optimization

- <u>*Plan*</u>: *Tree of R.A. ops, with choice of alg for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, 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 best plan. Practically: Avoid worst plans!
- We will study the System R approach.

Outline

- Relational algebra equivalences
- Statistics and size estimation
- Plan enumeration and cost estimation
- Nested queries

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

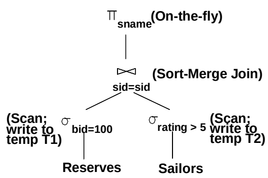
More Equivalences

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

Outline

- Relational algebra equivalences
- Statistics and size estimation
- Plan enumeration and cost estimation
- Nested queries

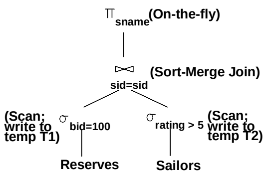
Example Plan



Statistics and Catalogs

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Example Plan



Size Estimation and Reduction Factors

- Consider a query block:

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

- What is maximum # tuples possible in result?
- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * product of all RF's.
  - Implicit assumption that *terms* are independent!
  - 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))*

Reduction Factors & Histograms

- For better estimation, use a histogram

No. of Values	2	3	3	1	8	2	1
Value	0-.99	1-1.99	2-2.99	3-3.99	4-4.99	5-5.99	6-6.99

equiwidth

No. of Values	2	3	3	3	3	2	4
Value	0-.99	1-1.99	2-2.99	3-4.05	4.06-4.67	4.68-4.99	5-6.99

equidepth

Outline

- Relational algebra equivalences
- Statistics and size estimation
- Plan enumeration and cost estimation
- Nested queries

Highlights of System R Optimizer

- Impact:
  - Most widely used currently; 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 plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

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 estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
- We'll discuss the System R cost estimation approach.
  - Very inexact, but works ok in practice.
  - More sophisticated techniques known now.

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).

Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - *Cost is Height(I)+1 for a B+ tree, about 1.2 for hash index.*
- Clustered index I matching one or more selects:
  - *(NPages(I)+NPages(R)) * product of RF's of matching selects.*
- Non-clustered index I matching one or more selects:
  - *(NPages(I)+NTuples(R)) * product of RF's of matching selects.*
- Sequential scan of file:
  - *NPages(R).*

Note: *Typically, no duplicate elimination on projections! (Exception: Done on answers if user says DISTINCT.)*

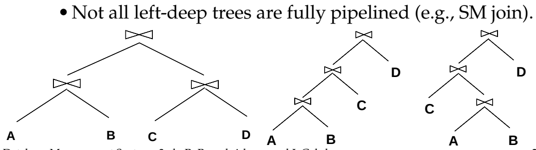
Example

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

- If we have an index on *rating*:
  - (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) pages are retrieved. (This is the ***cost***.)
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) pages are retrieved.
- 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

- 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).

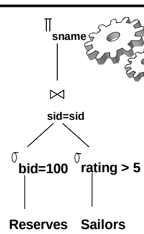


Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method 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.

Example

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



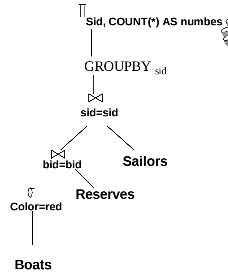
- Pass1:
  - *Sailors*: B+ tree matches *rating>5*, and is probably cheapest. However, if this selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper.
    - Still, B+ tree plan kept (because tuples are in *rating* order).
  - *Reserves*: B+ tree on *bid* matches *bid=500*; cheapest.
- Pass 2:
  - We consider each plan retained from Pass 1 as the outer, and consider how to join it with the (only) other relation.
    - e.g., *Reserves as outer*: Hash index can be used to get Sailors tuples that satisfy *sid* = outer tuple's *sid* value.

Enumeration of Plans (Contd.)

- N-1 way plan not combined with a relation unless there is a join condition between them
  - Unless all predicates in WHERE have been used up!
  - i.e., avoid Cartesian products if possible.
  - In spite of this pruning, plan space is still exponential in # tables
- ORDER BY, GROUP BY, aggregates etc. handled as a final step
  - Use an `interestingly ordered' plan
  - Or use an additional sorting operator

Example

Sailors:
 Hash, B+ on *sid*
Reserves:
 Clustered B+ tree on *bid*
 B+ on *sid*
Boats
 B+, Hash on *color*



Select S.sid, COUNT(*) AS numbes
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = "red"
GROUP BY S.sid

Pass 1

- Best plan for accessing each relation regarded as the first relation in an execution plan
  - Reserves, Sailors: File Scan
  - Boats: B+ tree & Hash on color

Pass 2

- For each of the plans in pass 1, generate plans joining another relation as the inner, using all join methods
  - File Scan Reserves (outer) with Boats (inner)
  - File Scan Reserves (outer) with Sailors (inner)
  - File Scan Sailors (outer) with Boats (inner)
  - File Scan Sailors (outer) with Reserves (inner)
  - Boats hash on color with Sailors (inner)
  - Boats Btree on color with Sailors (inner)
  - Boats hash on color with Reserves (inner)
  - Boats Btree on color with Reserves (inner)
- Retain cheapest plan for each pair of relations
  - Also "interesting order" plans even if they are not cheapest

Pass 3

- For each of the plans retained from Pass 2, taken as the outer, generate plans for the inner join
  - eg Boats hash on color with Reserves (bid) (inner) (sortmerge)) inner Sailors (B-tree sid) sort-merge

Add cost of aggregate

- Cost to sort the result by sid, if not returned sorted

Outline

- Relational algebra equivalences
- Statistics and size estimation
- Plan enumeration and cost estimation
- Nested queries

Nested Queries

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

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
- Outer block is optimized with the cost of `calling' nested block computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered. *The non-nested version of the query is typically optimized better.*

Nested block to optimize:
SELECT *
FROM Reserves R
WHERE R.bid=103
  AND S.sid= *outer value*

Equivalent non-nested query:
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
  AND R.bid=103