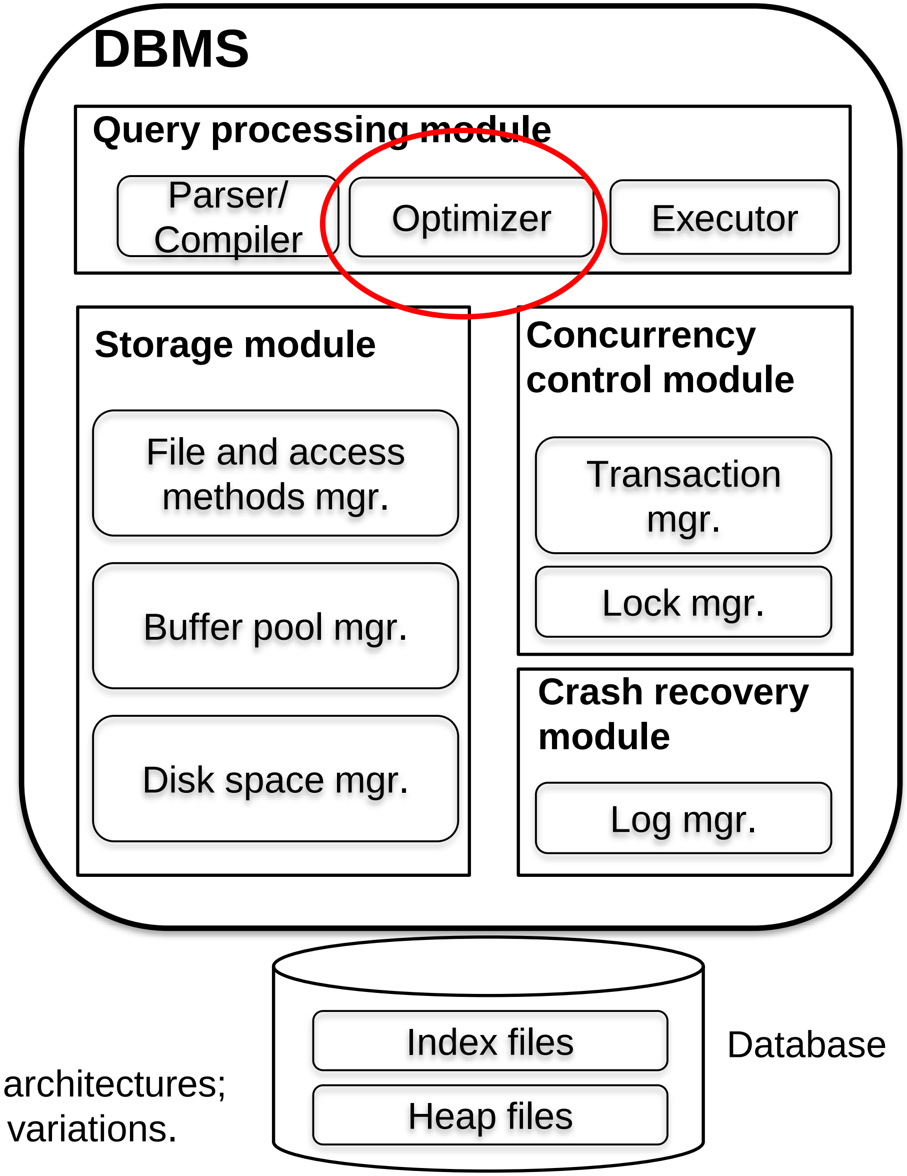
INFO20003 Database Systems

Dr Renata Borovica-Gajic

Lecture 13
Query Optimization Part I

Remember this? Components of a DBMS



TODAY & Next time

This is one of several possible architectures; each system has its own slight variations.

Coverage

- Overview
- Query optimization
- Cost estimation

Readings: Chapter 12 and 15, Ramakrishnan & Gehrke, Database Systems

THE UNIVERSITY OF MELBOURNE

Overview of Query Processing

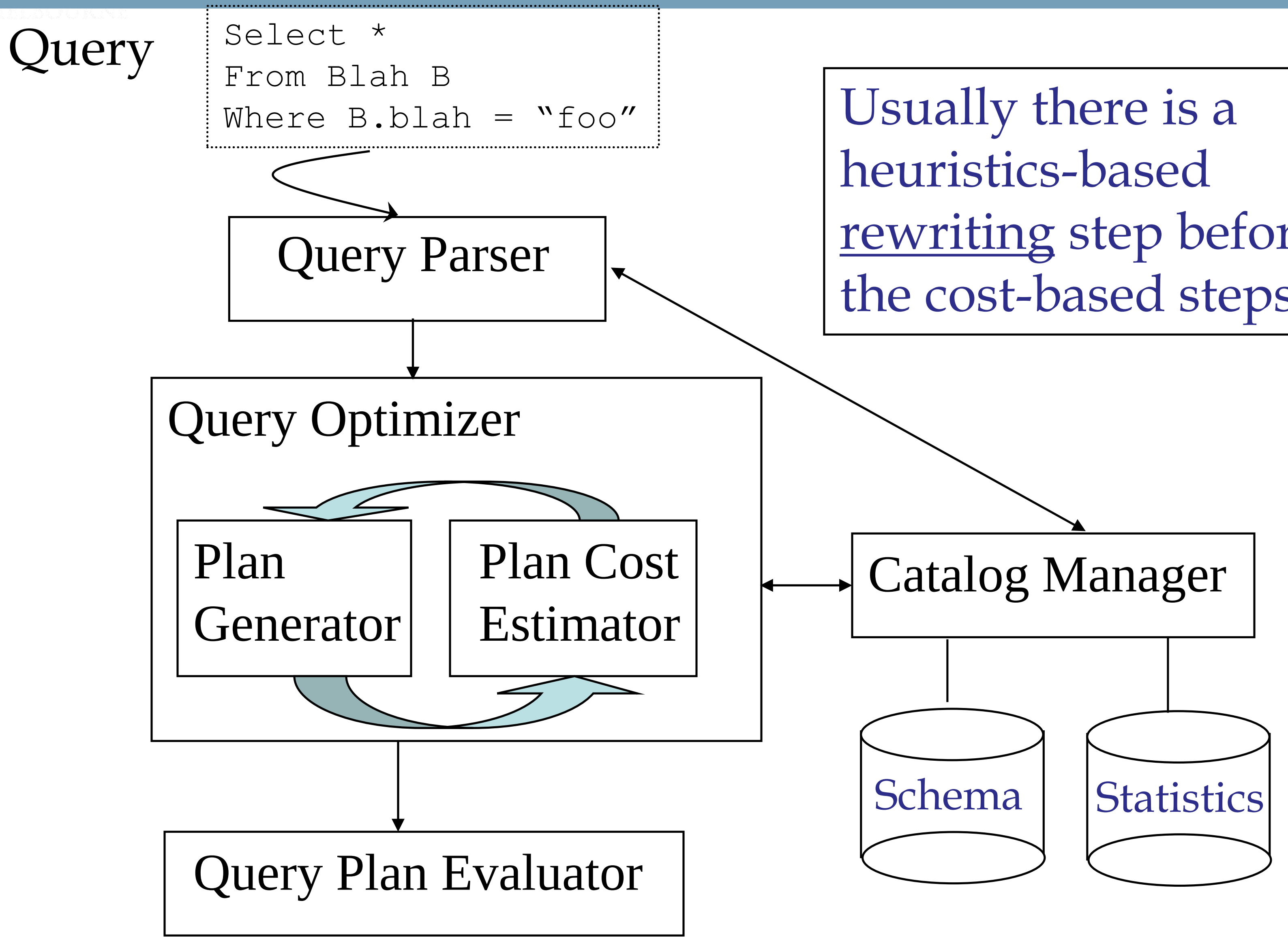
- Implementation of single Relational Operations
- Choices depend on indexes, memory, stats,…
- Joins
  - –Blocked nested loops:
    - •simple, exploits extra memory
  - –Indexed nested loops:
    - • best if one relation small and one indexed
  - –Sort/Merge Join
    - •good with small amount of memory, bad with duplicates
  - –Hash Join
    - •fast (enough memory), bad with skewed data

Query Optimization

- Typically many methods of executing a given query, all giving same answer
- Cost of alternative methods often **varies enormously**
- Desirable to find a low-cost execution strategy

- We will cover:
  - Relational algebra equivalences
  - Cost estimation
    - Result size estimation and reduction factors
    - Statistics and Catalogs
  - Enumerating alternative plans
- Will focus on “System R”-style optimizers

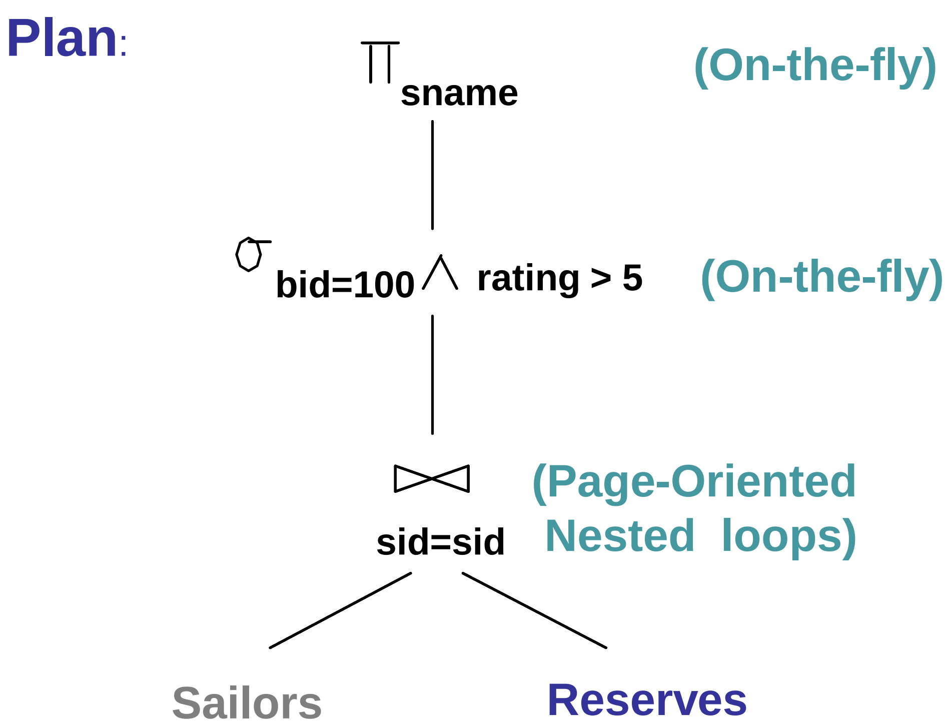
Refresh: Query execution



Usually there is a heuristics-based rewriting step before the cost-based steps.

Query Plans

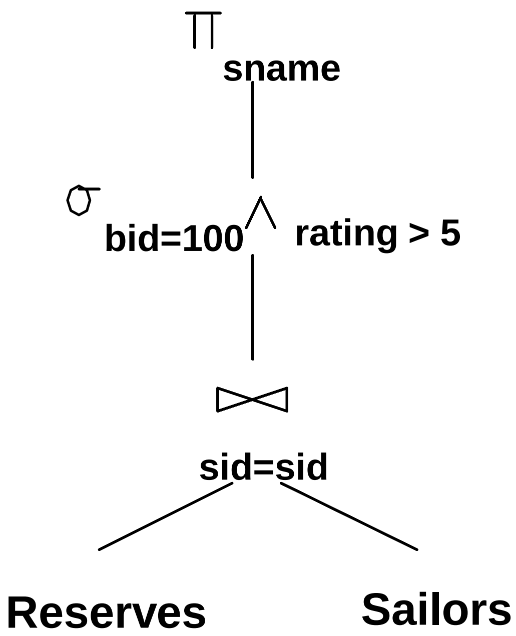
- A tree, with relational algebra operators as nodes
- Each operator labeled with choice of algorithm



* By convention, *outer* is on *left*.

Iterator Interface

- A note on implementation:



•Relational operators at nodes support uniform *iterator* interface:

Open( ), get_next( ), close( )

•Unary Operators – On Open() call Open() on child

•Binary Operators – call Open() on left child then on right

Query Optimization Overview

Query:

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

To optimize:

1. Query first broken into “blocks”
2. Each block converted to relational algebra
3. Then, for each block, several alternative query plans are considered
4. Plan with lowest estimated cost is selected

A Familiar 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)
Boats (<u>*bid*: integer</u>, *bname*: string, *color*: string)

Query Optimization

- Overview
- Query optimization
- Cost estimation

Readings: Chapter 15, Ramakrishnan & Gehrke, Database Systems

Step 1: Break query into Query Blocks

- Query block = unit of optimization
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple

  (This is an over-simplification, but serves for now)

Outer block

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

Nested block

Step 2: Converting query block into relational algebra expression

Query:

```
SELECT  S.sid
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
```

Relational algebra:

$$\pi_{\text{S.sid}}(\sigma_{\text{B.color = “red”}}(\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats}))$$

A Fancier Example …

- For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
AND S.rating = ( SELECT MAX (S2.rating) FROM Sailors S2)
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

Example translated to relational algebra

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
AND S.rating = ( SELECT MAX (S2.rating) FROM Sailors S2)
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

Inner Block

$$\pi_{\text{S.sid, MIN(R.day)}} \left(\text{HAVING}_{\text{COUNT(*)>2}} \left(\text{GROUP BY}_{\text{S.Sid}} \left(\sigma_{\text{B.color = “red”} \wedge \text{S.rating = val}} \left(\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats} \right)\right)\right)\right)$$

Select-Project-Join Optimization

- Core of every query is a select-project-join (SPJ) expression
- Other aspects, if any, carried out on result of SPJ core:
  - Group By (either sort or hash)
  - Having (apply filter on-the-fly)
  - Aggregation (easy once grouping done)
  - Order By (sorting is the name of the game)
- Not much room to exploit equivalences on non-SPJ parts
- Focus on optimizing SPJ core

Relational Algebra Equivalences

- Selections: $\sigma_{c_1 \wedge \cdots \wedge c_n}(R) \equiv \sigma_{c_1}\left(\ldots\left(\sigma_{c_n}(R)\right)\right)$ $(Cascade)$

  $\sigma_{c_1}\left(\sigma_{c_2}(R)\right) \equiv \sigma_{c_2}\left(\sigma_{c_1}(R)\right)$ $(Commute)$

- Projections: $\pi_{a_1}(R) \equiv \pi_{a_1}\left(\ldots\left(\pi_{a_n}(R)\right)\right)$ *(Cascade)*

  a_i is a set of attributes of R and $a_i \subseteq a_{i+1}$ for $i = 1 \ldots n-1$

- These equivalences allow us to 'push' selections and projections ahead of joins.

Examples …

$$\sigma_{age<18 \wedge rating>5} (Sailors)$$

$$\leftrightarrow \sigma_{age<18} (\sigma_{rating>5} (Sailors))$$

$$\leftrightarrow \sigma_{rating>5} (\sigma_{age<18} (Sailors))$$

$$\cancel{\pi_{age,rating} (Sailors) \leftrightarrow \pi_{age} (\pi_{rating} (Sailors))} \quad (?)$$

$$\pi_{age,rating} (Sailors) \leftrightarrow \pi_{age,rating} (\pi_{age,rating,sid} (Sailors))$$

Another Equivalence

- A projection commutes with a selection that only uses attributes retained by the projection

$$\pi_{\text{age, rating, sid}}\,(\sigma_{\text{age<18} \wedge \text{rating>5}}\,(\text{Sailors}))$$

$$\leftrightarrow \sigma_{\text{age<18} \wedge \text{rating>5}}\,(\pi_{\text{age, rating, sid}}\,(\text{Sailors}))$$

Equivalences Involving Joins

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \qquad \textit{(Associative)}$$

$$(R \bowtie S) \equiv (S \bowtie R) \qquad \textit{(Commutative)}$$

* These equivalences allow us to choose different join orders

Mixing Joins with Selections & Projections

- Converting selection + cross-product to join

$$\sigma_{S.sid = R.sid}\,(\text{Sailors x Reserves})$$

$$\leftrightarrow \text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves}$$

- Selection on just attributes of S commutes with R $\bowtie$ S

$$\sigma_{S.age<18}\,(\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

$$\leftrightarrow (\sigma_{S.age<18}\,(\text{Sailors})) \bowtie_{S.sid = R.sid} \text{Reserves}$$

- We can also "push down" projection (*but be careful*…)

$$\pi_{S.sname}\,(\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

$$\leftrightarrow \pi_{S.sname}\,(\pi_{sname,sid}(\text{Sailors}) \bowtie_{S.sid = R.sid} \pi_{sid}(\text{Reserves}))$$

What do you think? True or False?

1. $R \times S = S \times R$
2. $(R \times S) \times T = R \times (S \times T)$
3. $\sigma_p(R \cup S) = \sigma_p(R) \cup S$
4. $R \cup S = S \cup R$
5. $\sigma_p(R - S) = R - \sigma_p(S)$
6. $R \cup (S \cup T) = (R \cup S) \cup T$
7. $\sigma_{R.p \vee S.q} (R \bowtie S) =$

$$\left[(\sigma_p R) \bowtie S\right] \cup \left[R \bowtie (\sigma_q S)\right]$$

Query Rewriting

- Modern DBMS's may rewrite queries before the optimizer sees them
- Main purpose: de-correlate and/or flatten nested subqueries

- De-correlation:
  - Convert correlated subquery into uncorrelated subquery
- Flattening:
  - Convert query with nesting into query w/o nesting

Example: Decorrelating a Query

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

<u>Equivalent uncorrelated query:</u>

```
SELECT  S.sid
FROM Sailors S
WHERE  S.sid IN
   (SELECT  R.sid
    FROM  Reserves R
    WHERE  R.bid=103)
```

* Advantage: nested block only needs to be executed once (rather than once per S tuple)

Example: “Flattening” a Query

```
SELECT  S.sid
FROM Sailors S
WHERE  S.sid IN
  (SELECT  R.sid
   FROM  Reserves R
   WHERE  R.bid=103)
```

Equivalent non-nested query:

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

* Advantage: can use a join algorithm + optimizer can select among join algorithms & reorder freely

Query transformations: Summary

- Before optimizations, queries are flattened and de-correlated
- Queries are first broken into blocks
- Blocks are converted to relational algebra expressions
- Equivalence transformations are used to push down selections and projections

Query Optimization

- Overview
- Query optimization
- Cost estimation

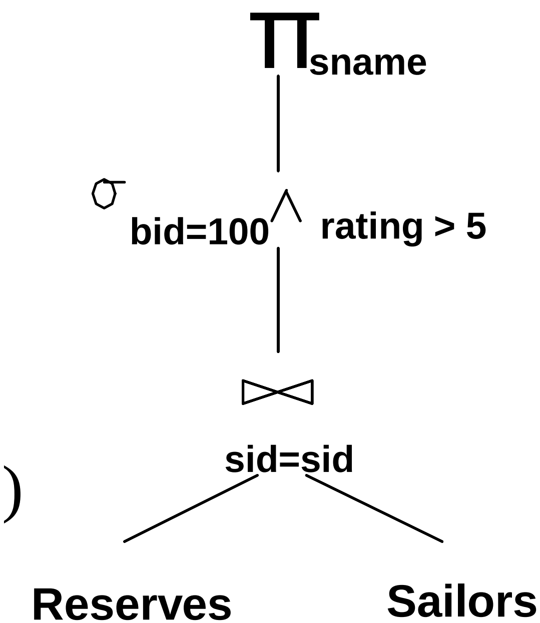
Readings: Chapter 15, Ramakrishnan & Gehrke, Database Systems

Recall: Query Optimization Overview

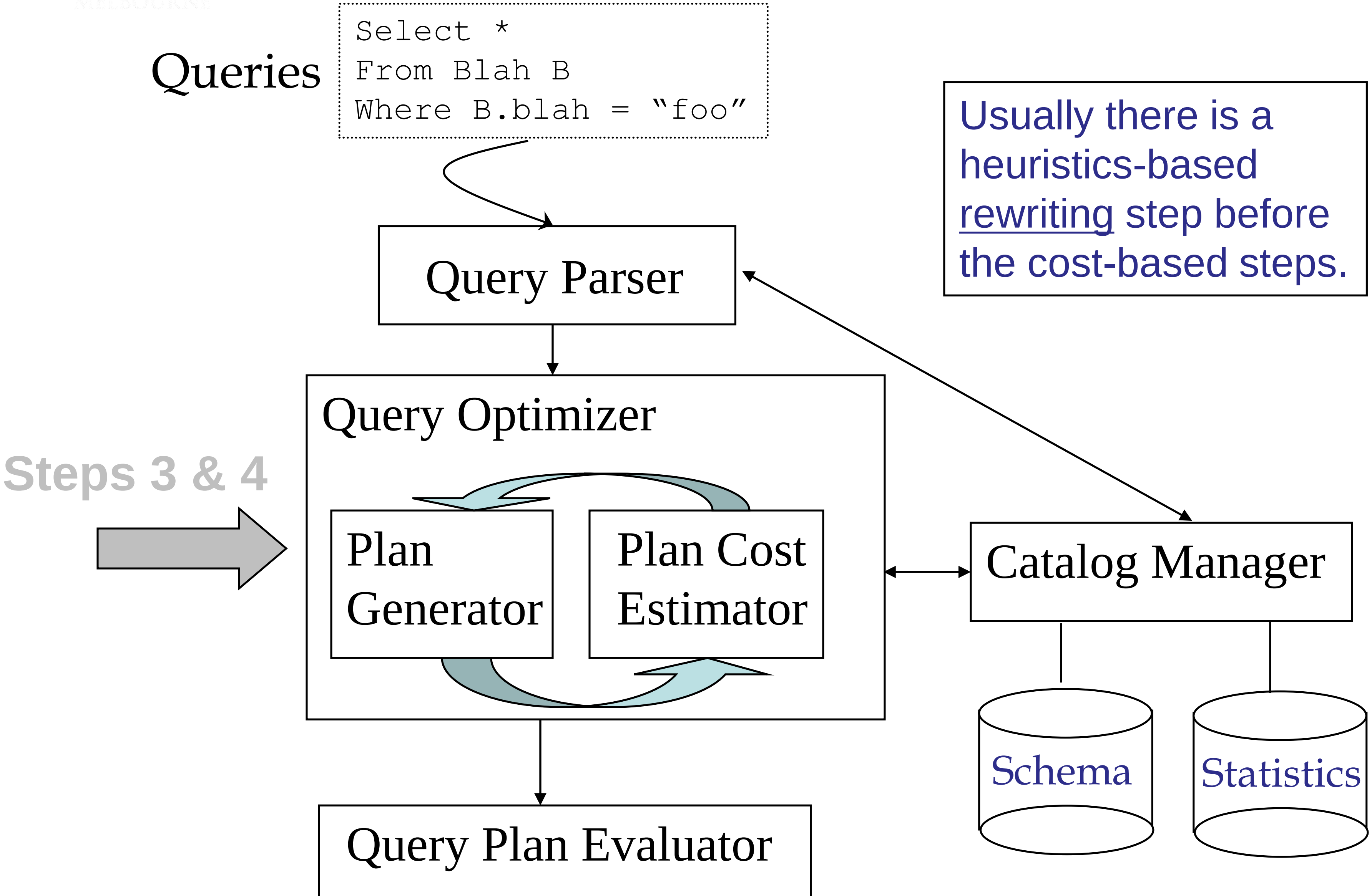
1. Query first broken into “blocks”
2. Each block converted to relational algebra
3. Then, for each block, several alternative query plans are considered
4. Plan with lowest estimated cost is selected

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

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



Cost-based Query Sub-System



Usually there is a heuristics-based rewriting step before the cost-based steps.

Two Main Issues

1. For a given query, what plans are considered?
   - Algorithm to search plan space for cheapest (estimated) plan.
2. How is the cost of a plan estimated?

- Ideally: Want to find best plan.
- Reality: Avoid worst plans!

Highlights of System R Optimizer

- Impact:
  - Most widely used currently; works well for < 10 joins
- Cost estimation:
  - Very inexact, but works okay in practice
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes
  - Considers combination of CPU and I/O costs
  - More sophisticated techniques known now
- Plan Space: Too large, must be pruned
  - Only the space of *left-deep plans* is considered
  - Cross products are avoided

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, 100 distinct bids
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages, 10 Ratings, 40.000 sids

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
  - In System R, cost is boiled down to a single number consisting of #I/O + *factor* * #CPU instructions

Statistics and Catalogs

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

Size Estimation and 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)* associated with each *term* reflects the impact of the *term* in reducing result size
- RF is usually called “selectivity”

Result Size Estimation for Selections

- *Result cardinality* = Max # tuples * product of all RF's
  (Implicit assumption that values are uniformly distributed and *terms* are independent!)
- Term *col=value* (given index I on *col*)
  RF = *1/NKeys(I)*
- Term *col>value*
  RF = *(High(I)-value)/(High(I)-Low(I))*
- *Note: if missing indexes, assume RF = 1/10*

Result Size Estimation for Joins

- Q: Given a join of R and S, what is the range of possible result sizes (in #of tuples)?
  - Hint: what if R_cols$\cap$S_cols = $\emptyset$?
  - R_cols$\cap$S_cols is a key for R (and a Foreign Key in S)?

Result Size Estimation for Joins

- General case: R_cols∩S_cols = {A} (and A is key for neither)
  - If NKeys(A,**S**) **>** NKeys(A,**R**)
    - Assume S values are a superset of R values, so each R value finds a matching value in S
    - Estimate each tuple r of R generates NTuples(S)/NKeys(A,S) result tuples, so…

      est_size = NTuples(R) * NTuples(S)/NKeys(A,**S**)
  - Else, if NKeys(A,**R**) **>** NKeys(A,**S**) … symmetric argument, yielding:

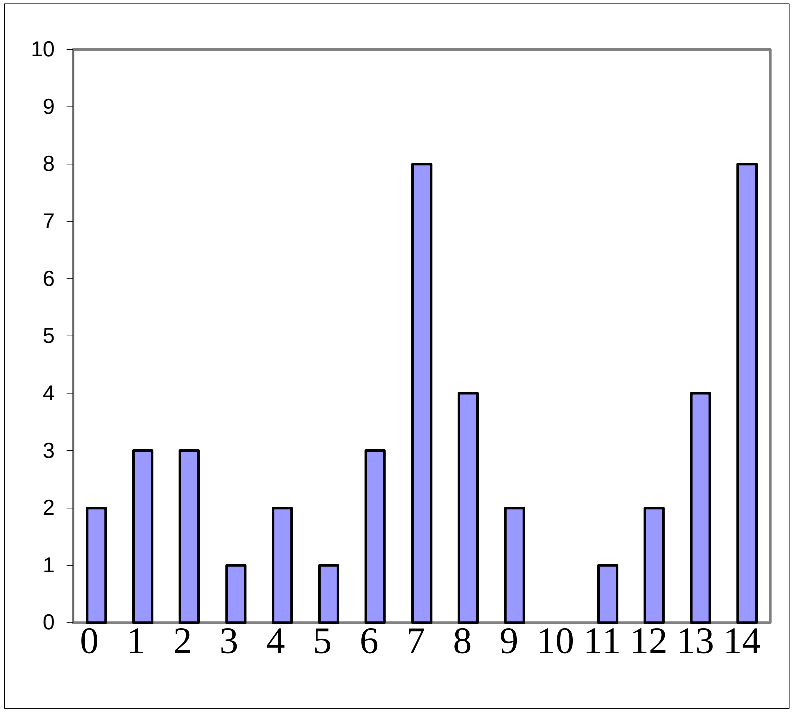
    est_size = NTuples(R) * NTuples(S)/NKeys(A,**R**)
  - Overall:

    est_size = NTuples(R)*NTuples(S)/MAX{NKeys(A,**S**), NKeys(A,**R**)}

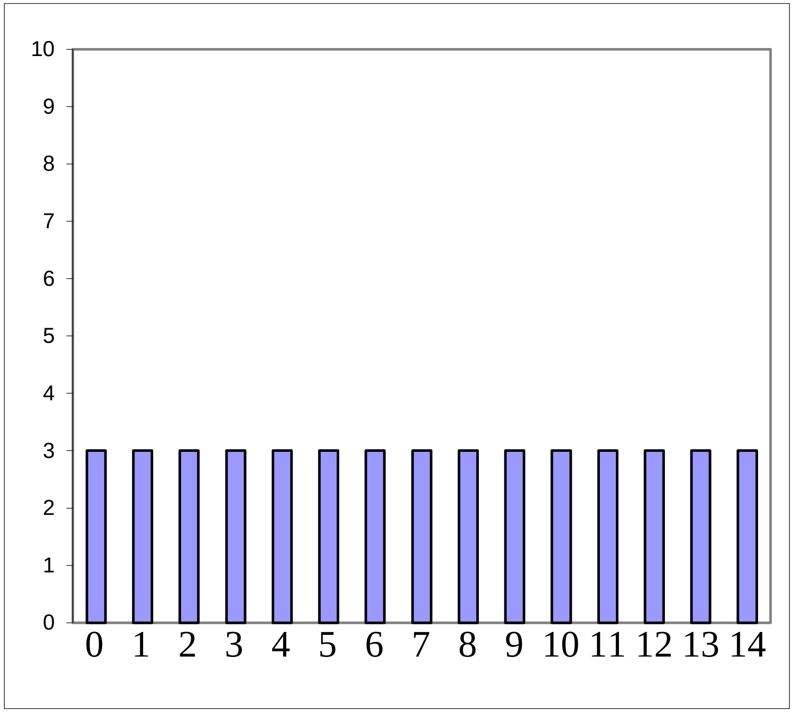
On the Uniform Distribution Assumption

- Assuming uniform distribution is rather crude

Distribution D



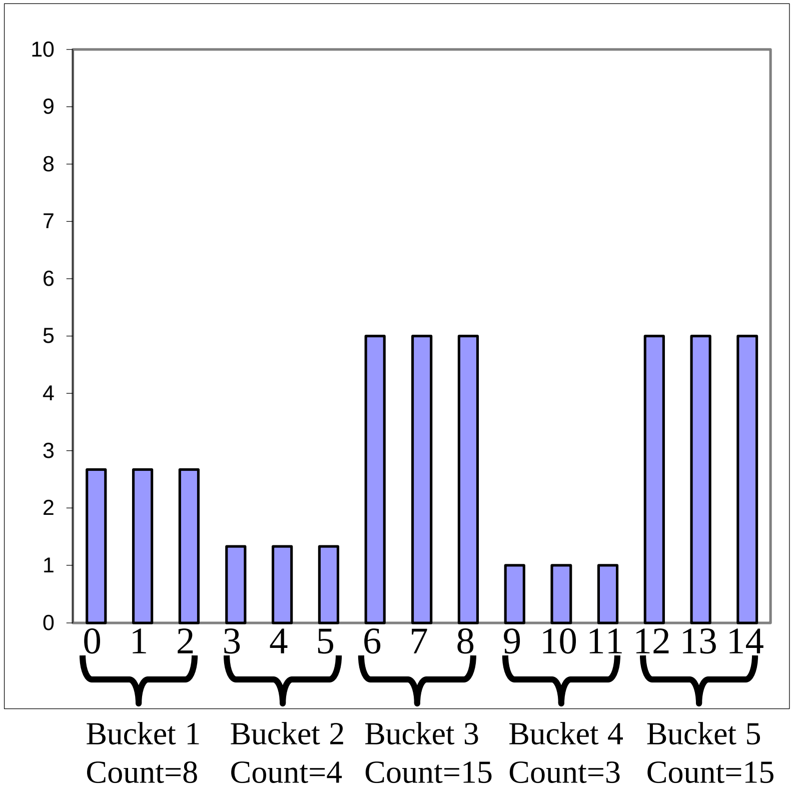
Uniform distribution approximating D



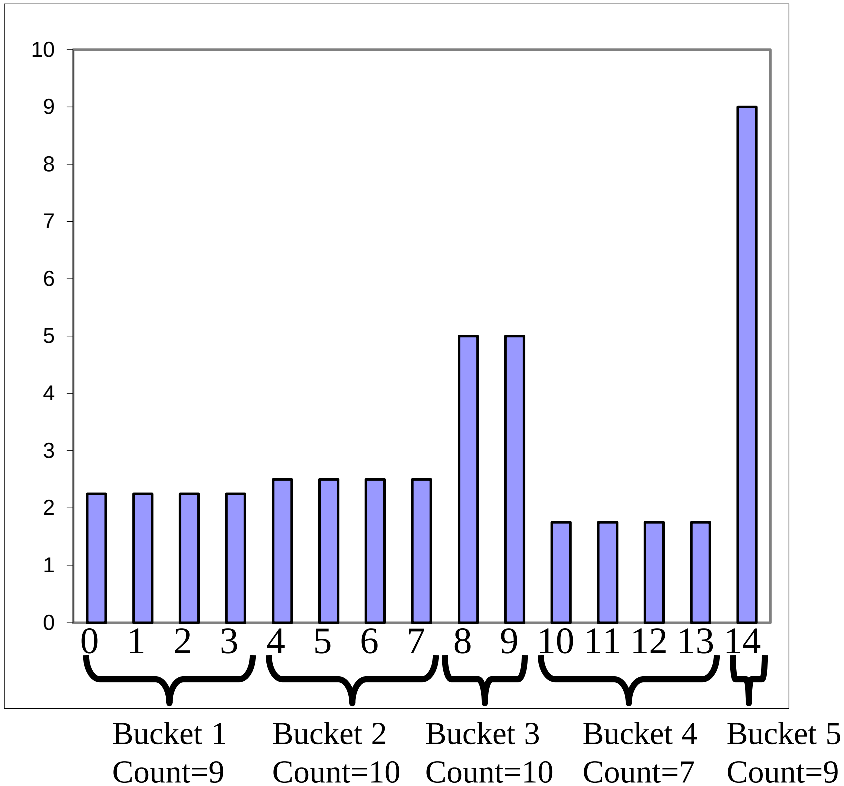
Histograms

- For better estimation, use a *histogram*

Equiwidth histogram



Equidepth histogram



Cost estimation: Summary

- The costs of possible strategies vary widely
- Estimate result sizes using statistics
- Estimate costs of each operator
- Focus on optimizing select-project-join (SPJ) blocks

What's examinable

- What is query optimization/steps?
- Equivalence classes
- Result size/cost estimation

- Important for Assignment 3 as well

Next Lecture

- Query optimization Part II
  - Plan enumeration