CSE 541: Database Systems I

Query Optimization

Slides Credit: Andy Pavlo, Tianzheng Wang

Query Optimization

- SQL is designed as a declarative language.
  - Users tell the DBMS what answer they want, but NOT how to get it.
- There can be a huge difference in performance:
  - Hours vs. seconds vs. milliseconds.
- First implemented in IBM System R in 1970s.
  - People argued that the DBMS could never choose a query plan better than what a human could write.
- Many concepts and designs from R are still used today.

Logical vs. Physical Plans

- Goal of a query optimizer
- Logical algebra expression → the optimal equivalent physical algebra expression.

- Physical operators define a specific execution strategy using an access path.
  - They can depend on the physical format of the data that they process (i.e., sorting, compression).
  - Not always a 1:1 mapping from logical to physical.

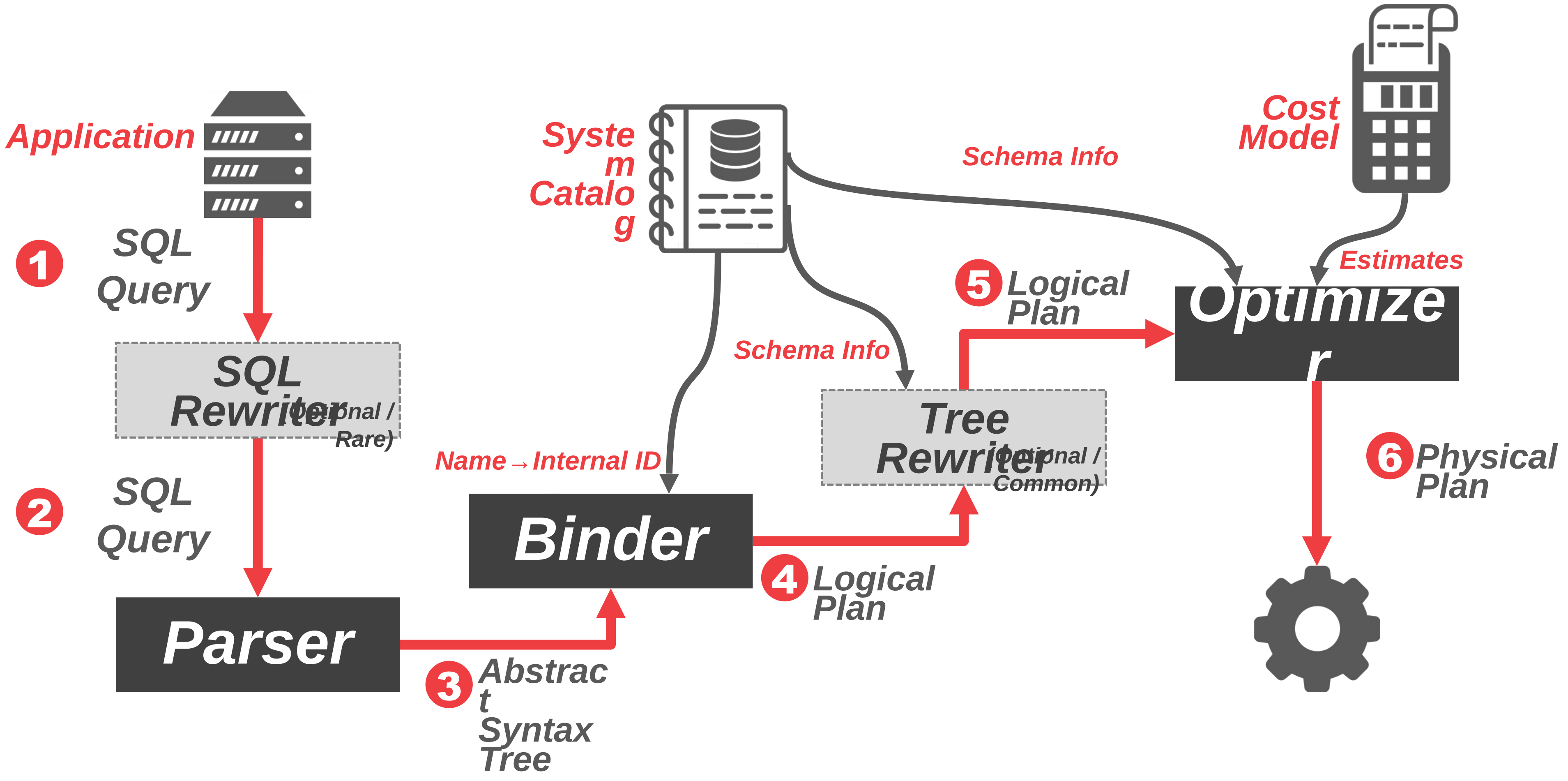
- Optimal may not be practical.
- Avoid the worst.

QO is NP-Hard

- This is the hardest part of building a DBMS.
- If you are good at this, you will get paid $$$.

- People are starting to look at employing ML to improve the accuracy and efficacy of optimizers.
  - IBM DB2 tried this with [LEO]() in the early 2000s…
- Active research in the DB community is happening as well.

DMBS Frontend Overview



Workflow of Query Optimization

SQL query

⬇

Query blocks

Basic unit of optimization

⬇

Two relational algebra expressions that generate the same results

Identify <u>equivalent</u> relational algebra expressions

⬇

Compare implementation of equivalent relational algebra expressions

➔ Generates several query evaluation plans

⬇

Estimate the cost of each query plan and choose the <u>cheapest</u>

- "Thinks the optimizer"
- Typically important to avoid the worst plans

Query Blocks: Basic Optimization Unit

SQL queries are parsed into a collection of query blocks

Query block: a SQL query with no nesting and

- With exactly one SELECT clause, FROM clause
- With a most one WHERE, GROUP BY, and HAVING clause

- The query optimizer concentrates on optimizing a single query block at a time

Example Query

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

Full SQL query:

```
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(*)>1
```

Nested query

Example Query

Outer block:

```
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=(reference to nested block)
GROUP BY  S.sid
HAVING    COUNT(*)>1
```

Nested block:

```
SELECT MAX (S2.rating)
FROM   Sailors.S2
```

- The optimizer chooses an evaluation plan for each block
  - Express it as a relational algebra expression (choose from potentially many)
  - Based on information available in the catalog, e.g., length of records/fields, relation statistics, index availability.

Query Optimizer

- **Heuristics / Rules**
  - Rewrite the query to remove stupid/inefficient things.
  - These techniques may need to examine the catalog, but they do NOT need to examine data.
- **Cost-based Search**
  - Use a model to estimate the cost of executing a plan.
  - Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.

Relational Algebra Equivalences

Allow the optimizer to evaluate any equivalent (and hopefully cheaper) expression and still get correct results

Selections:

- Cascading of selections: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ \ldots\ \sigma_{cn}(R))$
  - Left to right: replace a selection that has several conjuncts with several smaller selections
  - Right to left: combine several selections into one

- Commutative: $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$
  - OK to test c1 and c2 in either order

Relational Algebra Equivalence

<u>Projections:</u>

- Cascading projections: $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$
- Successively eliminating columns is equivalent to eliminating all but the columns retained by the final projection, if $a_i \subseteq a_{i+1}$ for i in 1…n -1

<u>Cross-Products and Joins:</u>

- Commutativity
  - $R \times S \equiv S \times R$
  - $R \bowtie S \equiv S \bowtie R$
- Associativity
  - $R \times (S \times T) \equiv (R \times S) \times T$
  - $R \times \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$

➔Relations can be joined in any order (aka “order independence”)

Relational Algebra Equivalences

Equivalences involving more than one operator:

- A projection commutes with a selection that only uses attributes retained by the projection
  - $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$
  - Every attribute in *c* must be included in *a*

- Selection between attributes of the two arguments of a cross-product converts cross-product to a join
  - $R \bowtie_c S \equiv \sigma_c(R \times S)$

- A selection on just attributes of R commutes with R × S / R ⋈ S
  - $\sigma_c(R \times S) \equiv \sigma_c(R) \times S$
  - $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$

Attributes in *c* must appear only in R and not in S

Common Heuristics

Rewrite queries based on relational algebra equivalences

- Some common heuristics are always applied
  - Most of the time giving better plans
  - Keep the query plan space smaller

Selection cascades and pushdown:

- Apply selections as soon the relevant columns are available
- $\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(R \bowtie_{sid=sid} S))$
- $\pi_{sname}(\sigma_{bid=100}(R \bowtie_{sid=sid} \sigma_{rating>5} S))$
- One selection condition cascaded into two and one (rating > 5) pushed down to join
  - Reduce join input size ➔ join becomes cheaper
  - Assumption: selection is cheaper than join

Common Heuristics

Projections:

- Keep only the columns needed to evaluate downstream operators
- $\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(R \bowtie_{sid=sid} S))$
- $\pi_{sname}(\pi_{sid}(\sigma_{bid=100}(R)) \bowtie_{sid=sid} \pi_{sname,sid}(\sigma_{rating>5} S)))$

Avoid Cartesian products:

- Given a choice, do joins rather than cross-products
  - R(a, b), S(b, c), T(c, d)
  - Favour $(R \bowtie S) \bowtie T$ over $(R \times S) \bowtie T$
- Not always the best, e.g., for small tables
- Used in System R

Expression Rewriting

- An optimizer transforms a query's expressions (e.g., WHERE clause predicates) into the optimal/minimal set of expressions.

- Implemented using a pattern-matching rule engine.
  - Search for expressions that match a pattern.
  - When a match is found, rewrite the expression.
  - Halt if there are no more rules that match.

Expression Rewriting Examples

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
SELECT * FROM A;
```

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL
);
```

- Join Elimination

```
SELECT * FROM A;
  FROM A AS A1 JOIN A AS A2
    ON A1.id = A2.id;
```

Expression Rewriting Examples

- Ignoring Projections

```
SELECT * FROM A;
 WHERE EXISTS(SELECT val FROM A
AS A2
                 WHERE A1.id =
A2.id);
```

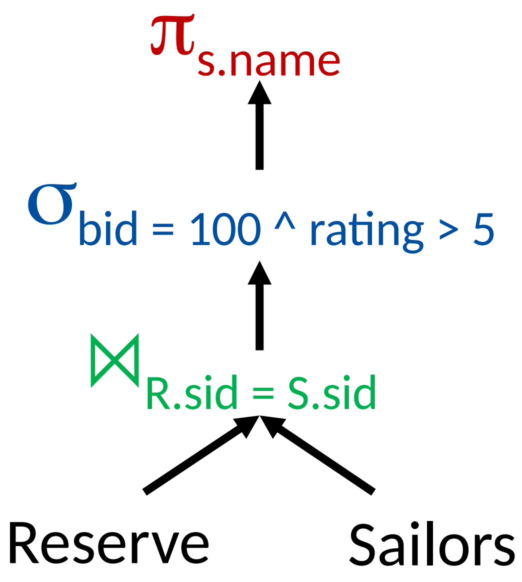
- Merging Predicates

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 150;
    OR val BETWEEN 50 AND 150;
```

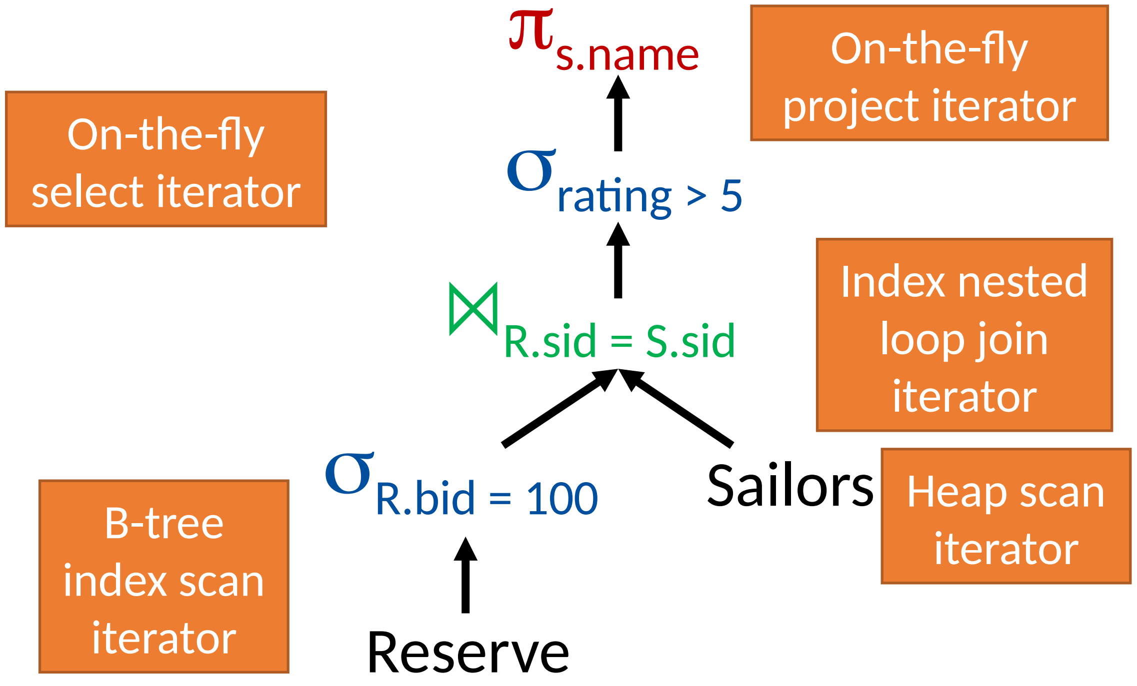
```
CREATE TABLE A (
  id INT PRIMARY
KEY,
  val INT NOT NULL
);
```

Logic Plan → Physical Plan

Logical query plan:



(Optimized) physical query plan:



- Relational algebra equivalences: logical
  - Need actual implementations (physical equivalences)

Physical Equivalences

Table access with single-table selections and projections:

- Heap scan
- Index scan (if index is available on specified columns)

Equijoins:

- Block nested loops join: simple and can utilize extra memory
- Index nested loops join: good if one relation is very small the other has index
- Sort-merge join: good with small memory, equal-sized tables
- Grace hash join: better than sorting with one small table

Non-equijoins:

- Only choice: nested loops join algorithms
- Block nested loops join typically preferred (most efficient nested loops join algorithm)

Enumerating Alternative Plans

Two main cases:

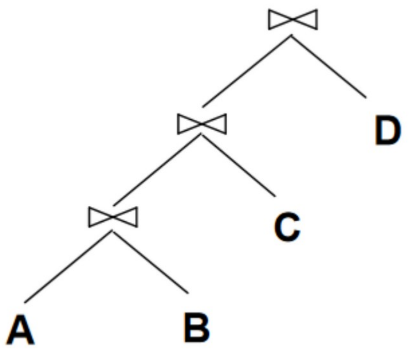
- Single-relation plans
- Multiple-relation plans

Single-relation plans:

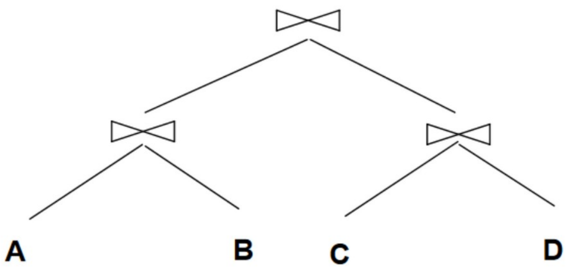
- Queries consist of a combination of select, project, and aggregate operations
- Consider each available access path (file scan/index)
    - Choose the one with the least estimated cost
- Different operations are carried out together
    - E.g., with an index for selection, projection is done for each retrieved tuple, which is pipelined into the aggregate operator

Enumerating Alternative Plans

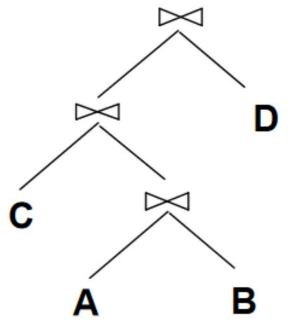
Query: A ⋈ B ⋈ C ⋈ D



“Left-deep” (linear)



“Bushy tree” (non-linear)



Linear tree: at least one child of each join node is a base table

Left-deep tree/plan: the right child of each join node is a base table

- Allow to generate fully pipelined plans
  - Intermediate results not written to temporary files
  - Not all left-deep trees are fully pipelined (e.g., sort-merge join)
- Optimizers typically only consider left-deep plans (System R style)

Enumerating Left-Deep Plans

Left-deep plans differ only in

- The order of relations
- The access method for each relation
- Method for each join

Enumerate using N passes (for joining N relations):

- Pass 1: Find best 1-relation plan for each relation
    - Done when first accessing the relation, before any joins
    - Do selections and projections as early as possible
    - May retain the cheapest plan for each different ordering of produced tuples
        - Useful for subsequent steps, e.g., sort-merge join, GROUP BY, ORDER BY

Enumerating Left-Deep Plans

Enumerate using N passes (for joining N relations):

- Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation
    - All 2-relation plans
    - Suppose Pass 1 generated relation A (outer) and B is the inner relation
    - Examine the list of selections in the WHERE clause to find:
        - Selections that involve only B and can be applied before join
        - Selections that define the join
        - Selections that involve attributes in other relations (can be done only after join)
    - Note: tuples generated by outer plan are assumed to be pipelined into the join

Enumerating Left-Deep Plans

Enumerate using N passes (for joining N relations):

- Pass 3: Find the best way to join the result of each 2-relation plan (as outer) to another relation
  - All 3-relation plans
- Pass N: Find best way to join the result of an (N-1)-relation plan (as outer) to the N'th relation
  - All N-relation plans

Enumerating Left-Deep Plans

- For each subset of relations, retain only
  - The cheapest plan overall, plus
  - The cheapest plan for each interesting order of the tuples
- ORDER BY, GROUP BY, aggregates etc.
  - Handled as a **final step** using either an “interestingly ordered” plan or an additional sorting operator
- Avoid Cartesian products in early stages if possible

Note: In spite of pruning plan space, this approach is still exponential in the number of tables.

Query Optimizer

- **Heuristics / Rules**
  - Rewrite the query to remove stupid/inefficient things.
  - These techniques may need to examine the catalog, but they do <u>NOT</u> need to examine data.
- **Cost-based Search**
  - Use a model to estimate the cost of executing a plan.
  - Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.

Cost Model Components

- **Choice #1: Physical Costs**
  - Predict CPU cycles, # of I/Os, cache misses, DRAM consumption, etc
  - Depends heavily on hardware.
- **Choice #2: Logical Costs**
  - Estimate result sizes per operator
  - Independent of the operator algorithm
  - Need estimation for operator result size
- **Choice #3: Algorithmic Costs**
  - Complexity of the operator algorithm implementation

Disk-based DBMS Cost Model

- Disk accesses will always **dominate** the execution time of a query.
  - CPU costs are negligible
  - Must consider sequential vs. random I/O

- This is easier to model if the DBMS has **full control** over buffer management.
  - Know the replacement strategy, pinning and assume exclusive access to disk.

Cost Estimation

For each plan considered, must estimate:

- The cost of each operation in the plan tree
  - Depends on input cardinalities
  - Already discussed partially, e.g., sequential scan, index scan, joins

- The size of result for each operation in the tree
  - The output of an operator can become the input of another
  - Use information about the input relations
  - For selections and joins, often assume independence of predicates

Estimating Result Sizes

- Consider a query block like:

```
SELECT attribute list
FROM   relation list
WHERE  term1 ∧ term2 ∧ term3 ∧ . . .  ∧ term n
```

- Maximum possible result size: product of the cardinalities of each table listed in the FROM clause
- Each term listed in the WHERE clause reduces some of the tuples from result set

➔ The key is to model the effect of the WHERE clause

Estimating Result Sizes

Reduction factor (RF): ratio of expected result size to input size considering a selection term

- Estimated result size = product of all terms’ RF * maximum size

- Simple but with assumptions
  - Conditions tested by each term is statistically independent
  - Uniform distribution of values
  - There are more sophisticated statistics and methods proposed
    - E.g., keep histograms of values per column

Cost Estimation for Single-Relation Plans

Index I on primary key matches selection:

- Cost = Height(I) + 1 for a B+-tree, about 1.2 for hash table

Clustered index I matching one or more selects:

- (#Pages(I) + #Pages(R)) * product of RFs of matching selects

Non-clustered index I matching one or more selects:

- (#Pages(I) + #Tuples(R)) * product of RFs of matching selects

Sequential scan of file: #Pages(R)

Statistics

- The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.
- Different systems update them at different times.

- Manual invocations:
  - Postgres/SQLite: ANALYZE
  - Oracle/MySQL: ANALYZE TABLE
  - SQL Server: UPDATE STATISTICS
  - DB2: RUNSTATS

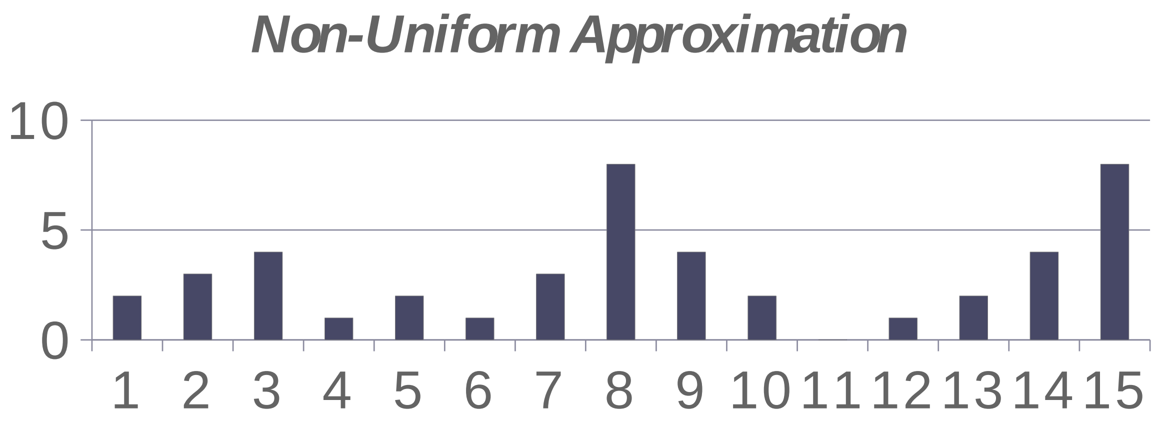
- For each relation R, the DBMS maintains the following information:
  - N_R: Number of tuples in R.
  - V(A, R) : Number of distinct values for attribute A.

Selection Cardinality Estimation

- **Assumption #1: Uniform Data**
  - The distribution of values (except for the heavy hitters) is the same.
- **Assumption #2: Independent Predicates**
  - The predicates on attributes are independent
- **Assumption #3: Inclusion Principle**
  - The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.

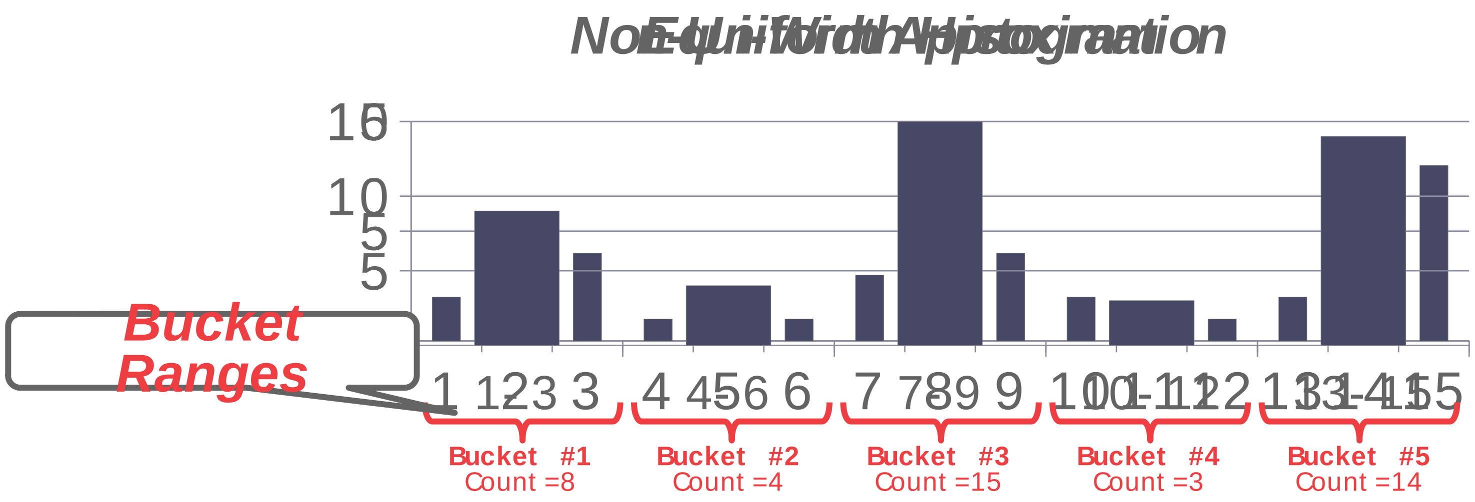
What if non-uniform?

- Our formulas are nice, but we assume that data values are uniformly distributed.
- We can use **histograms**!!



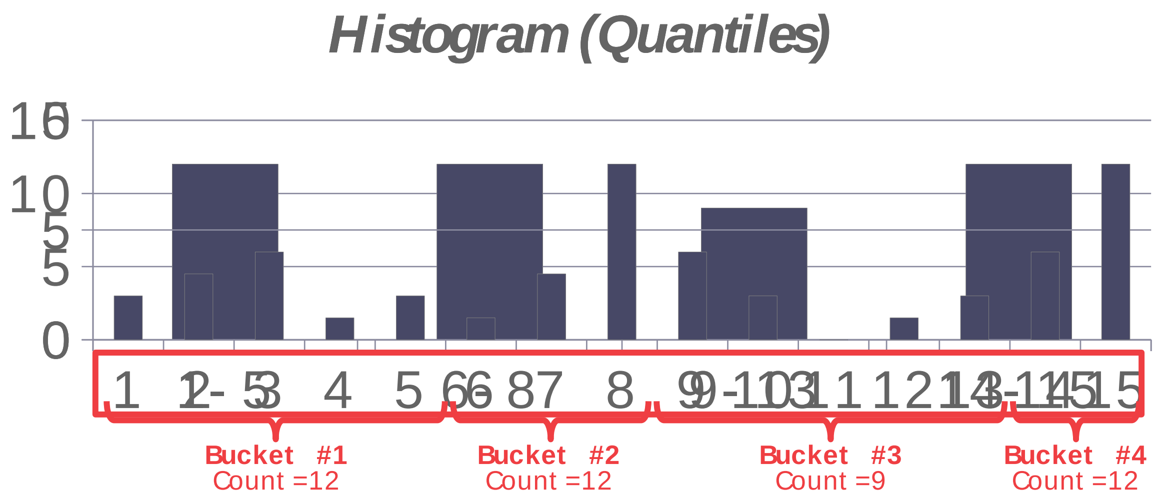
Equal-width Histograms

- All buckets have the same width (i.e., the same number of values).



Equal-depth Histograms

- Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



Other Options

- **Sketches**: Probabilistic data structures that generate approximate statistics about a data set.
- Cost-model can replace histograms with sketches to improve its selectivity estimate accuracy.

- Most common examples:
  - Count-Min Sketch (1988): Approximate frequency count of elements in a set.
  - HyperLogLog (2007): Approximate the number of distinct elements in a set.

- **Sampling**: Modern DBMSs also collect samples from tables to estimate selectivities.
- Update samples when the underlying tables changes significantly.

Cost Estimation Example: Single Relation

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

- If an index is available on rating
  - (1/NKeys(I)) * #Tuples(R) = (1/10) * 40000 tuples retrieved
  - Clustered index: (1/#Keys(I)) * (#Pages(I) + #Pages(R)) = (1/10) * (50 + **500**) pages are retrieved
    - This is the cost
  - Unclustered index: (1/#Keys(I)) * (#Pages(I)+#Tuples(R)) = (1/10) * (50 + **40000**) pages are retrieved
- If an index is available on sid: Have to retrieve all tuples/pages
  - With a clustered index, cost is 50 + 500
  - With an unclustered index, cost is 50 + 40000
- If doing a file scan: cost is 500 (retrieving all pages)

Cost Estimation for Multiple-relation Plans

Query block:

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

- Built up by joining one new relation at a time
- Cost of join method, plus estimation of join cardinality give both cost and result size estimates

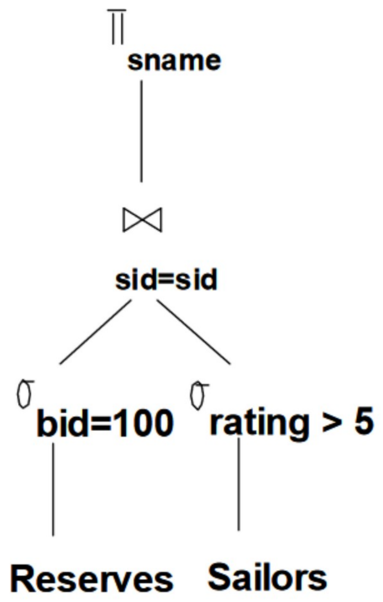
Cost Estimation Example: Multiple Relations

Sailors table:

- B+-tree index on rating
- Hash index on sid

Reserves table:

B+-tree index on bid



Pass 1:

- Sailors: B+-tree matches rating > 5
  - If the index is unclustered, file scan may be cheaper
  - Here assume the B+-tree plan is kept
- Reserves: B+-tree matches bid = 100 ➔ cheapest choice

Pass 2:

- Consider the result of each plan retained in Pass 1 as the outer relation, and see how it joins with the only other inner relation
- E.g., Reserves as outer, use hash index to get Sailors tuples (probe hash table with Reserves tuple's sid)

Beyond Single Query Block: Nested Subqueries

- Nested queries are optimized independently
  - Nested loops evaluation
- Outer block
  - Considered as providing a selection condition
  - Should take into account the cost of “calling” the nested block

Example: Find the names of sailors with the highest rating

```
SELECT S.sname
FROM   Sailors S
WHERE  S.rating = (SELECT MAX(S2.rating)
                                    FROM
 Sailors S2)
```

Nested query returns a single tuple:

- Replaced with the computation result
- As if the query had e.g., **S.rating = 8** originally

Nested Subqueries

- Sometimes the nested query may return a relation

Example: Find the names of sailors who have reserved boat 103

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

Nested query returns a relation:

- A set of sailor IDs (**sids**)
- Nested query is evaluated only once
- Outer block checks whether S.sid is in sids, using a join of S and sids
  - Can be optimized using index on S.sid, theoretically
  - Often always index nested loops join in practice

Nested Subqueries

- Sometimes nested queries need to be evaluated more than once

Example: A different version of the previous query

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

Correlated query: variable in outer block used in nested block

- May evaluate the nested block for each tuple in S

Nested Subqueries

- A nested query often has an equivalent version without nesting
- A correlated query often has an equivalent version without correlation

Example: An equivalent query without nesting

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

- Optimizer often not able to recognize, and tend to poorly handle nested/correlated queries
- Up to the user to write “good” queries