Lecture #15

Query Planning & Optimization

SPRING 2024  Prof. Jignesh Patel
ADMINISTRIVIA

Project #3 is due Sun April 7, 2024 @ 11:59pm

Mid-Term
→ See me during OH for exam viewing

Final Exam
→ Thu May 2, 2024, @ 05:30pm-08:30pm
SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

**Catalog**

- **EMP** (ssn, ename, addr, sal, did)
  - 10,000 records
  - 1,000 pages

- **DEPT** (did, dname, floor, mgr)
  - 500 records
  - 50 pages

**Total: 2M I/Os**

- **4 reads, 1 write**
  - π<sub>ename</sub>

- **2,000 + 4 writes**
  - (10K/500 = 20 emps per dept)

- **1,000,000 + 2,000 writes**
  - (FK join, 10K tuples in temp T2)

- **50 + 50,000 + 1,000,000 writes**
  - (write to temp file T1)
  - 5 tuples per page in T1

**Query**

```sql
SELECT ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'
```
SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

Query

Total: 54K I/Os

Read temp T2
4 reads + 1 writes

Read temp T1
2,000 reads +4 writes

Page NL, write to temp T1
50 + 50,000 + 2000 writes

Catalog

<table>
<thead>
<tr>
<th>Table</th>
<th>Type</th>
<th>Records</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP (ssn, ename, addr, sal, did)</td>
<td>clustered</td>
<td>10,000</td>
<td>1,000</td>
</tr>
<tr>
<td>DEPT (did, dname, floor, mgr)</td>
<td>clustered</td>
<td>500</td>
<td>50</td>
</tr>
</tbody>
</table>

EMP (ssn, ename, addr, sal, did)

σ
dname = 'Toy'

π
ename

EMP did = DEPT did

EMP

DEPT
SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

Catalog

<table>
<thead>
<tr>
<th></th>
<th>clustered</th>
<th>nonclustered</th>
<th>nonclustered</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP (ssn, ename, addr, sal, did)</td>
<td>▲</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>10,000 records</td>
<td>1,000 pages</td>
<td></td>
</tr>
<tr>
<td>DEPT (did, dname, floor, mgr)</td>
<td>▲</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td></td>
<td>500 records</td>
<td>50 pages</td>
<td></td>
</tr>
</tbody>
</table>

Query

w/ Materialization
Total: 7,159 I/Os
w/ Pipelining
Total: 3,151 I/Os

Read temp T2
4 reads + 1 writes

Read temp T1
2,000 reads + 4 writes

Sort-merge join (50 buffers)
3*(|Emp|+|Dept|) = 3150 + 2000 writes

π_{ename}

σ_{dname = 'Toy'}

EMP.did = DEPT.did

EMP

DEPT
SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

Catalog

<table>
<thead>
<tr>
<th>Clustered</th>
<th>Nonclustered</th>
<th>Nonclustered</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP (ssn, ename, addr, sal, did)</td>
<td>▲</td>
<td>△</td>
</tr>
<tr>
<td>10,000 records</td>
<td>1,000 pages</td>
<td></td>
</tr>
<tr>
<td>DEPT (did, dname, floor, mgr)</td>
<td>▲</td>
<td>△</td>
</tr>
<tr>
<td>500 records</td>
<td>50 pages</td>
<td></td>
</tr>
</tbody>
</table>

Access: Index (name)
3 reads + 1 write

Read temp T1, NL-IDX Join
1 + 3 (idx) + 20 (ptr chase) + 4 writes

πename

σdname = 'Toy'

Swap

Total: 37 I/Os
Annotated RA Tree a.k.a. The Physical Plan

Simple projection
π_{ename}
Estimates: output cardinality = 20, ...

NL-IDX using unclustered index on EMP.id
Estimates: output cardinality = 20, ...

Access Path: Unclustered B-tree
Estimates: output cardinality = 1, ...

To the scheduler to run the query

Access Path: File Scan
Estimates: output cardinality = 10K

EMP

σ_{dname = 'Toy'}

DEPT

EMP.did = DEPT.did

Pipeline

Unclustered B-tree

Access Path:

File Scan
Query Optimization (QO)

1. Identify candidate equivalent trees (logical). It is an NP-hard problem, so the space is large.

2. For each candidate, find the execution plan tree (physical). We need to estimate the cost of each plan.

3. Choose the best overall (physical) plan.

Practically: Choose from a subset of all possible plans.
LOGICAL VS. PHYSICAL PLANS

The optimizer generates a mapping of a **logical** algebra expression to 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.
QUERY OPTIMIZATION

Heuristics / Rules

→ Rewrite the query to remove (guessed) inefficiencies; e.g, always do selections first, or push down projections as early as possible.
→ These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search

→ Use a model to estimate the cost of executing a plan.
→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.
Predicate Pushdown

\[ \pi_{ename} \left( \sigma_{dname = 'Toy'} (DEPT \bowtie EMP) \right) \] rewrite \[ \pi_{ename} \left( EMP \bowtie \sigma_{dname = 'Toy'} (DEPT) \right) \]
Replace Cartesian Product

... (σ_{DEPT.did = EMP.did} (DEPT X EMP))

... (EMP ⋈_{did} DEPT)
Projection Pushdown

\[ \pi_{ename} (\ldots \bowtie did EMP) \]

\[ \pi_{ename, did} (\pi_{ename, did} (\pi_{ename, did} EMP)) \]
Equivalence

\[ \sigma_{p_1} (\sigma_{p_2}(R)) \equiv \sigma_{p_2} (\sigma_{p_1}(R)) \quad (\sigma \text{ commutativity}) \]

\[ \sigma_{p_1 \land p_2 \ldots \land p_n}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(\ldots \sigma_{p_n}(R))) \quad (\text{cascading } \sigma) \]

\[ \Pi_{a_1}(R) \equiv \Pi_{a_1}(\Pi_{a_2}(\ldots \Pi_{a_k}(R)\ldots)), \quad a_i \subseteq a_{i+1} \quad (\text{cascading } \Pi) \]

\[ R \bowtie S \equiv S \bowtie R \quad (\text{join commutativity}) \]

\[ R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad (\text{join associativity}) \]

\[ \sigma_p (R \times S) \equiv (R \bowtie_p S), \quad \text{if } P \text{ is a join predicate} \]

\[ \sigma_p (R \times S) \equiv \sigma_{p_1}(\sigma_{p_2}(R) \bowtie_{p_4} \sigma_{p_3}(S)), \quad \text{where } P = p_1 \land p_2 \land p_3 \land p_4 \]

\[ \Pi_{A_1, A_2, \ldots, A_n}(\sigma_p(R)) \equiv \Pi_{A_1, A_2, \ldots, A_n}(\sigma_p(\Pi_{A_1, \ldots, A_n, B_1, \ldots, B_M}R)), \quad \text{where } B_1 \ldots B_M \text{ are columns in } P \]

\[ \ldots \]
ARCHITECTURE OVERVIEW

1. SQL Query
2. Abstract Syntax Tree
3. Logical Plan
4. Physical Plan

Application → SQL Query → Parser → Binder → Optimizer → Cost Model

System Catalog → Schema Info → Estimates
QUERY OPTIMIZATION

Heuristics / Rules

→ Rewrite the query to remove inefficient patterns.
→ These techniques may need to examine catalog, but they do not need to examine data.

Examples: predicate pushdown, replace cartesian product, projection pushdown …

Cost-based Search

→ Use a model to estimate the cost of executing a plan.
→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.
COST-BASED QUERY OPTIMIZATION

Let’s start with a certain style of QO: cost-based, bottom-up QO (the classic System-R optimizer approach)

Approach: Enumerate different plans for the query and estimate their costs.
→ Single relation.
→ Multiple relations.
→ Nested sub-queries.

It chooses the best plan it has seen for the query after exhausting all plans or some timeout.
SINGLE-RELATION QUERY PLANNING

Pick the best access method.
→ Sequential Scan
→ Binary Search (clustered indexes)
→ Index Scan

Predicate evaluation ordering.

Simple heuristics are often good enough for this.
SYSTEM R OPTIMIZER

Break the query into blocks and generate the logical operators for each block.

For each logical operator, generate a set of physical operators that implement it.
→ All combinations of join algorithms and access paths

Then, iteratively construct a “left-deep” join tree that minimizes the estimated amount of work to execute the plan.

System-R optimizer does NOT consider this “shape”
**SYSTEM R OPTIMIZER**

### Step #1: Choose the best access paths to each table

- **ARTIST**: Sequential Scan
- **APPEARS**: Sequential Scan
- **ALBUM**: Index Look-up on `NAME`

### Step #2: Enumerate all possible join orderings for tables

- ARTIST  \(\times\)  APPEARS  \(\times\)  ALBUM
- APPEARS  \(\times\)  ALBUM  \(\times\)  ARTIST
- ALBUM  \(\times\)  APPEARS  \(\times\)  ARTIST
- APPEARS  \(\times\)  ARTIST  \(\times\)  ALBUM
- ARTIST  \(\times\)  ALBUM  \(\times\)  APPEARS
- ALBUM  \(\times\)  ARTIST  \(\times\)  APPEARS

### Step #3: Determine the join ordering with the lowest cost
SYSTE R R OPTIMIZER

ARTIST \(\bowtie\) APPEARS \(\bowtie\) ALBUM

\[\text{HASH}\_\text{JOIN}(A1, A3)\]
\[\text{HASH}\_\text{JOIN}(A2, A3)\]
\[\text{SM}\_\text{JOIN}(A3, A2)\]

\[\text{ARTIST.ID=APPEARS.ARTIST.ID}\]
\[\text{ALBUM.ID=APPEARS.ALBUM.ID}\]
\[\text{APPEARS.ALBUM.ID=ALBUM.ID}\]
SYSTEM R OPTIMIZER

ARTIST ⋈ APPEARS ⋈ ALBUM

HASH_JOIN(A1, A3) ⋈ SM_JOIN(A1, A3, A2)

HASH_JOIN(A2, A3, A1) ⋈ SM_JOIN(A2, A3, A1)

HASH_JOIN(A3, A2, A1) ⋈ SM_JOIN(A3, A2, A1)

ARTIST ⋈ APPEARS ⋈ ALBUM

ALBUM ⋈ APPEARS ⋈ ARTIST

APPEARS.ALBUM_ID = ALBUM.ID

APPEARS.ARTIST_ID = ARTIST.ID

ARTIST.ID = APPEARS.ARTIST_ID

ALBUM.ID = APPEARS.ALBUM_ID

APPEARS.ALBUM_ID = ALBUM.ID

21
SYSTEM R OPTIMIZER
SYSTEM R OPTIMIZER

ARTIST \Join \APPEARS \Join \ALBUM

HASH_JOIN(A2, A3, A1)

APPEARS.ARTIST_ID=ARTIST.ID

ALBUM\Join\APPEARS
ARTIST

ALBUM.ID=APPEARS.ALBUM_ID

HASH_JOIN(A2, A3)

ARTIST ALBUM APPEARS
The query has `ORDER BY on ARTIST.ID` but the plans do not carry an explicit notion of the sorting properties.
$\Pi_{\text{year}, \text{artist\_name}, \text{album\_name}}$

Hash Join ...
Estimates: output cardinality = ...

$\bowtie$

Hash Join ...
output cardinality = ...

Unclustered B-tree
output cardinality = …

$\sigma_{\text{genre} = \text{Blues}}$

$\bowtie$

Hash Join ...
output cardinality = …

$\bowtie$

File Scan
output cardinality = 10K

$\bowtie$

File Scan
output cardinality = 10K

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$

$\bowtie$
MULTI-RELATION QUERY PLANNING

Choice #1: Bottom-up Optimization
→ Start with nothing and then build up the plan to get to the outcome that you want.

Choice #2: Top-down Optimization
→ Start with the outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal.

We just saw an example of this, the System R approach.
BOTTOM-UP OPTIMIZATION

Use static rules to perform initial optimization. Then use dynamic programming to determine the best join order for tables using a divide-and-conquer search method.

Examples: IBM System R, DB2, MySQL, Postgres, most open-source DBMSs.
TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be. Perform a branch-and-bound search to traverse the plan tree by converting logical operators into physical operators.

→ Keep track of global best plan during search.
→ Treat physical properties of data as first-class entities during planning.

Example: MSSQL, Greenplum, CockroachDB
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ **Logical** → **Logical**:
  
  \[ \text{JOIN}(A,B) \] to \[ \text{JOIN}(B,A) \]

→ **Logical** → **Physical**:
  
  \[ \text{JOIN}(A,B) \] to \[ \text{HASH}_\text{JOIN}(A,B) \]

ARTIST ⊙ APPEARS ⊙ ALBUM
ORDER-BY(ARTIST.ID)
Invoke rules to create new nodes and traverse the tree.

→ **Logical → Logical:**
  JOIN(A, B) to JOIN(B, A)

→ **Logical → Physical:**
  JOIN(A, B) to HASH_JOIN(A, B)
Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
   
   JOIN(A,B) to JOIN(B,A)

→ **Logical→Physical:**
   
   JOIN(A,B) to HASHJOIN(A,B)
Invoke rules to create new nodes and traverse the tree.

→ **Logical**→**Logical**:  
JOIN(A,B) to JOIN(B,A)

→ **Logical**→**Physical**:  
JOIN(A,B) to HASH_JOIN(A,B)
Invoke rules to create new nodes and traverse the tree.

→ Logical → Logical:
  JOIN(A, B) to JOIN(B, A)

→ Logical → Physical:
  JOIN(A, B) to HASH_JOIN(A, B)
Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
JOIN(A,B) to JOIN(B,A)

→ **Logical→Physical:**
JOIN(A,B) to HASH_JOIN(A,B)
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
  
  JOIN(A,B) to JOIN(B,A)

→ **Logical→Physical:**
  
  JOIN(A,B) to HASH_JOIN(A,B)
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ Logical→Logical:
  JOIN(A,B) to JOIN(B,A)

→ Logical→Physical:
  JOIN(A,B) to HASH_JOIN(A,B)
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ Logical→Logical:
  JOIN(A,B) to JOIN(B,A)

→ Logical→Physical:
  JOIN(A,B) to HASH_JOIN(A,B)
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ Logical → Logical:
  JOIN(A,B) to JOIN(B,A)

→ Logical → Physical:
  JOIN(A,B) to HASH_JOIN(A,B)
Invoke rules to create new nodes and traverse the tree.

→ \textbf{Logical}$\rightarrow$\textbf{Logical}:
  
  JOIN(A,B) to JOIN(B,A)

→ \textbf{Logical}$\rightarrow$\textbf{Physical}:
  
  JOIN(A,B) to HASH\_JOIN(A,B)

Can create “enforcer” rules that require input to have certain properties.
Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
  \[ \text{JOIN}(A, B) \] to \[ \text{JOIN}(B, A) \]

→ **Logical→Physical:**
  \[ \text{JOIN}(A, B) \] to \[ \text{HASH JOIN}(A, B) \]

Can create “enforcer” rules that require input to have certain properties.
Invoke rules to create new nodes and traverse the tree.

→ Logical→Logical:
JOIN(A,B) to JOIN(B,A)

→ Logical→Physical:
JOIN(A,B) to HASH_JOIN(A,B)

Can create “enforcer” rules that require input to have certain properties.
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ **Logical**→**Logical**:
JOIN(A,B) to JOIN(B,A)

→ **Logical**→**Physical**:
JOIN(A,B) to HASH_JOIN(A,B)

Can create “enforcer” rules that require input to have certain properties.

Invoke rules to create new nodes and traverse the tree.

→ **Logical**→**Logical**:
JOIN(A,B) to JOIN(B,A)

→ **Logical**→**Physical**:
JOIN(A,B) to HASH_JOIN(A,B)

Can create “enforcer” rules that require input to have certain properties.
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ Logical→Logical:
  JOIN(A,B) to JOIN(B,A)

→ Logical→Physical:
  JOIN(A,B) to HASH_JOIN(A,B)

Can create “enforcer” rules that require input to have certain properties.
Invoke rules to create new nodes and traverse the tree.

→ Logical → Logical:
  \[ \text{JOIN}(A,B) \text{ to } \text{JOIN}(B,A) \]

→ Logical → Physical:
  \[ \text{JOIN}(A,B) \text{ to } \text{HASH\_JOIN}(A,B) \]

Can create “enforcer” rules that require input to have certain properties.

\[ \text{ARTIST} \Join \text{APPEARS} \Join \text{ALBUM} \]

\[ \text{ORDER\_BY}\!(\text{ARTIST}\!.\!\text{ID}) \]
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ Logical→Logical:
JOIN(A,B) to JOIN(B,A)

→ Logical→Physical:
JOIN(A,B) to HASH_JOIN(A,B)

Can create “enforcer” rules that require input to have certain properties.
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ Logical→Logical:
  JOIN(A, B) to JOIN(B, A)

→ Logical→Physical:
  JOIN(A, B) to HASH_JOIN(A, B)

Can create “enforcer” rules that require input to have certain properties.
Life so far ... single block QO

Often, we get nested queries.
→ We could optimize each block using the methods we have discussed.
→ However, this may be inefficient since we optimize each block separately without a global approach.

What if we could flatten a nested query into a single block and optimize it?
→ Then, apply single-block query optimization methods.
→ Even if one can’t flatten to a single block, flattening to fewer blocks is still beneficial.
NESTED SUB-QUERIES

The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

Two Approaches:
→ Rewrite to de-correlate and/or flatten them.
→ Decompose nested query and store results in a temporary table.
NESTED SUB-QUERIES: REWRITE

```
SELECT name FROM sailors AS S
WHERE EXISTS (  
    SELECT * FROM reserves AS R  
    WHERE S.sid = R.sid  
    AND R.day = '2022-10-25'
)
```

```
SELECT name
FROM sailors AS S, reserves AS R
WHERE S.sid = R.sid
AND R.day = '2022-10-25'
```
DECOMPOSING QUERIES

For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.

Sub-queries are written to temporary tables that are discarded after the query finishes.
**DECOMPOSING QUERIES**

```sql
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
```
DECOMPOSING QUERIES

**Select Statement 1:**
```
SELECT MAX(rating) FROM sailors
```

**Select Statement 2:**
```
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
```
DECOMPOSING QUERIES

SELECT MAX(rating) FROM sailors

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 = ###
GROUP BY S.sid
HAVING COUNT(*) > 1
**DECOMPOSING QUERIES**

**Inner Block**

```sql
SELECT MAX(rating) FROM sailors
```

```sql
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 = ###
GROUP BY S.sid
HAVING COUNT(*) > 1
```

**Outer Block**

```sql
SELECT S.sid
FROM sailors S, reserves R, boats B
WHERE S.sid = R.sid
  AND R.bid = B.bid
  AND B.color = 'red'
  AND S.rating = ###
GROUP BY S.sid
HAVING COUNT(*) > 1
```
EXPRESSION REWRITING

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

Implemented using if/then/else clauses or 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

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0
```
EXPRESSION REWRITING

Impossible / Unnecessary Predicates

```sql
SELECT * FROM A WHERE false;
```
EXPRESSION REWRITING

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE false;
```

```
SELECT * FROM A WHERE NOW() IS NULL;
```
EXPRESSION REWRITING

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE false;
SELECT * FROM A WHERE false;
```
EXPRESSION REWRITING

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE false;
SELECT * FROM A WHERE false;
SELECT * FROM A WHERE RANDOM() IS NULL;
```
**EXPRESSION REWRITING**

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE false;
```

```
SELECT * FROM A WHERE false;
```

```
SELECT * FROM A WHERE RANDOM() IS NULL;
```

Merging Predicates

```
SELECT * FROM A
WHERE val BETWEEN 1 AND 100
  OR val BETWEEN 50 AND 150;
```
EXPRESSION REWRITING

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE false;
SELECT * FROM A WHERE false;
SELECT * FROM A WHERE RANDOM() IS NULL;
```

Merging Predicates

```
SELECT * FROM A
WHERE val BETWEEN 1 AND 150;
```
How do we calculate the cost of the plans?

We have formulas for the operator algorithms (e.g. the cost formulae for hash join, sort merge join, …), but we also need to estimate the size of the output that an operator produces.
The DBMS uses a cost model to predict the behavior of a query plan given a database state. → This is an internal cost that allows the DBMS to compare one plan with another.

It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information.
COST MODEL COMPONENTS

Choice #1: Physical Costs
→ Predict CPU cycles, I/O, cache misses, RAM consumption, network messages...
→ Depends heavily on hardware.

Choice #2: Logical Costs
→ Estimate output size per operator.
→ Independent of the operator algorithm.
→ Need estimations for operator result sizes.
POSTGRES COST MODEL

Uses a combination of CPU and I/O costs that are weighted by “magic” constant factors.

Default settings are obviously for a disk-resident database without a lot of memory:
→ Processing a tuple in memory is $400x$ faster than reading a tuple from disk.
→ Sequential I/O is $4x$ faster than random I/O.
POSTGRES COST MODEL

Uses a combination of CPU and I/O costs that are weighted by “magic” constant factors.

Default settings are obviously for a disk-resident database without a lot of memory:
→ Processing a tuple in memory is $400\times$ faster than reading a tuple from disk.
→ Sequential I/O is $4\times$ faster than random I/O.
19.7.2. Planner Cost Constants

The cost variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, seq_page_cost is conventionally set to 1.0 and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

**Note:** Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

**seq_page_cost (floating point)**

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see `ALTER TABLESPACE`).

**random_page_cost (floating point)**
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**
The *selectivity* \((sel)\) of a predicate \(P\) is the fraction of tuples that qualify.

**Equality Predicate:** \(A=constant\)

\[ sel(A=constant) = \frac{\text{#occurrences}}{|R|} \]

SELECT * FROM people
WHERE age = 9

![Bar chart showing the distribution of age values](chart.png)
**SELECTION CARDINALITY**

The **selectivity** \((\text{sel})\) of a predicate \(P\) is the fraction of tuples that qualify.

**Equality Predicate:** \(A=\text{constant}\)

\[ \text{sel}(A=\text{constant}) = \frac{\text{#occurrences}}{|R|} \]

→ Example: \(\text{sel}(\text{age}=9) = \)

![Graph showing the distribution of ages with distinct values and occurrences]
The selectivity \( \text{sel} \) of a predicate \( P \) is the fraction of tuples that qualify.

**Equality Predicate:** \( A = \text{constant} \)

\[ \text{sel}(A = \text{constant}) = \frac{\# \text{occurrences}}{|R|} \]

→ Example: \( \text{sel}(\text{age}=9) = \)

```
SELECT * FROM people
WHERE age = 9
```
The **selectivity** (\(sel\)) of a predicate \(P\) is the fraction of tuples that qualify.

**Equality Predicate: \(A=constant\)**

\[
\rightarrow \text{sel}(A=\text{constant}) = \frac{\#\text{occurrences}}{|R|}
\]

\[
\rightarrow \text{Example: sel(age=9) = } \frac{4}{45}
\]

---

**SELECT \(*\) FROM people**

**WHERE age = 9**

---

**# of occurrences**

**Distinct values of attribute**

**SC(age=9)=4**
SELECTION CARDINALITY

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.
Consider a database of automobiles:
→ # of Makes = 10, # of Models = 100

And the following query:
→ (make="Honda" AND model="Accord")

With the independence and uniformity assumptions, the selectivity is:
→ $1/10 \times 1/100 = 0.001$

But since only Honda makes Accords the real selectivity is
$1/100 = 0.01$
STATISTICS

Choice #1: Histograms
→ Maintain an occurrence count per value (or range of values) in a column.

Choice #2: Sketches
→ Probabilistic data structure that gives an approximate count for a given value.

Choice #3: Sampling
→ DBMS maintains a small subset of each table that it then uses to evaluate expressions to compute selectivity.
Our formulas are nice, but we assume that data values are uniformly distributed.

**Histogram**

- 15 Keys $\times$ 32-bits = 60 bytes
- Distinct values of attribute
EQUI-WIDTH HISTOGRAM

Maintain counts for a group of values instead of each unique key. All buckets have the same width (i.e., same # of value).

Non-Uniform Approximation

Bucket Ranges

- Bucket #1: Count=8
- Bucket #2: Count=4
- Bucket #3: Count=15
- Bucket #4: Count=3
- Bucket #5: Count=14
EQUI-DEPTH HISTOGRAMS

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

*Histogram (Quantiles)*
EQUI-DEPTH HISTOGRAMS

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

Histogram (Quantiles)
EQUI-DEPTH HISTOGRAMS

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

Histogram (Quantiles)
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:
→ **HyperLogLog** (2007): Approximate the number of distinct elements in a set.
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

\[
\text{sel}(\text{age}>50) = \frac{1}{3}
\]

**Table Sample**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Obama</td>
<td>61</td>
<td>Rested</td>
</tr>
<tr>
<td>1003</td>
<td>Tupac</td>
<td>25</td>
<td>Dead</td>
</tr>
<tr>
<td>1005</td>
<td>Andy</td>
<td>41</td>
<td>Illin</td>
</tr>
</tbody>
</table>

```sql
SELECT AVG(age) 
FROM people 
WHERE age > 50
```
CONCLUSION

• Query optimization is critical for a database system.

• SQL -> logical plan -> physical plan.

• Flatten queries before going to the optimization part. Expression handling is also important.

• QO enumeration can be bottom-up or top-down.

• Need to cost each plan, so need cost-estimation methods.
Essential Query Optimization papers


**Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE 1993: 209-218**

**Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34**

**Umeshwar Dayal: Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. VLDB 1987: 197-208**
Suggestions if you are going to build a QO

**Rule 1: Read lots of papers, especially from the 80s & 90s.**
→ Expect new combinations, only partially new core inventions.

**Rule 2: Early on, test various workloads on the QO.**
→ QOs harden over time as they “see” new workloads. Let them see more ASAP.

**Rule 3: Throw away the initial one (or two) and start anew.**
→ The hard part is going to be nitty-gritty details like data structures and pointers to shared objects; e.g., the list of predicates and the query graph structure, … You will NOT get this right in the first pass. Don’t try to patch; be prepared to rewrite.
Transactions!

→ aka the second hardest part about database systems