Query Optimization
**ADMINISTRIVIA**

**Mid-term Exam** is on Wednesday October 17\textsuperscript{th} → See [mid-term exam guide](#) for more info.

**Project #2 – Checkpoint #2** is due Friday October 19\textsuperscript{th} @ 11:59pm.
Remember that SQL is declarative.
\[\rightarrow\] User tells the DBMS what answer they want, not how to get the answer.

There can be a big difference in performance based on plan is used:
\[\rightarrow\] See last week: 1.3 hours vs. 0.45 seconds
First implementation of a query optimizer. People argued that the DBMS could never choose a query plan better than what a human could write.

A lot of the concepts from **System R**’s optimizer are still used today.
QUERY OPTIMIZATION

Heuristics / Rules
→ Rewrite the query to remove stupid / inefficient things.
→ Does not require a cost model.

Cost-based Search
→ Use a cost model to evaluate multiple equivalent plans and pick the one with the lowest cost.
QUERY PLANNING OVERVIEW

SQL Query → Parser → Binder → Rewriter (Optional) → Optimizer

- **Parser**: Converts SQL Query to Abstract Syntax Tree
- **Binder**: Assigns names to identifiers
- **Rewriter**: May optimize the query
- **Optimizer**: Generates Annotated AST
- **System Catalog**: Provides metadata
- **Cost Model**: Calculates costs

**Query Plan**
TODAY'S AGENDA

Relational Algebra Equivalences
Plan Cost Estimation
Plan Enumeration
Nested Sub-queries
Mid-Term Review
Two relational algebra expressions are equivalent if they generate the same set of tuples.

The DBMS can identify better query plans without a cost model.

This is often called **query rewriting**.
SELECT s.name, e.cid
FROM student AS s, enrolled AS e
WHERE s.sid = e.sid
AND e.grade = 'A'

\[ \pi_{s.name, e.cid} \sigma_{s.sid = e.sid \land \text{grade} = 'A'}^\circ \text{student} \bowtie \text{enrolled} \]

\[ \pi_{s.name, e.cid} \sigma_{s.sid = e.sid \land \text{grade} = 'A'}^\circ \text{student} \bowtie \text{enrolled} \]
RELATIONAL ALGEBRA EQUIVALENCES

\[
\pi_{name, cid} (\sigma_{\text{grade} = 'A'} (\text{student} \bowtie \text{enrolled}))
\]

\[
\quad \equiv \quad
\]

\[
\pi_{name, cid} (\text{student} \bowtie (\sigma_{\text{grade} = 'A'} (\text{enrolled})))
\]
RELATIONAL ALGEBRA EQUIVALENCES

Selections:
→ Perform filters as early as possible.
→ Reorder predicates so that the DBMS applies the most selective one first.
→ Break a complex predicate, and push down

\[ \sigma_{p_1 \land p_2 \land \ldots \land p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots \sigma_{p_n}(R))) \]

Simplify a complex predicate
→ \((X=Y \land Y=3) \rightarrow X=3 \land Y=3\)
RELATIONAL ALGEBRA EQUIVALENCES

Projections:

→ Perform them early to create smaller tuples and reduce intermediate results (if duplicates are eliminated)
→ Project out all attributes except the ones requested or required (e.g., joining keys)

This is not important for a column store...
**PROJECTION PUSHDOWN**

```sql
SELECT s.name, e.cid
FROM student AS s, enrolled AS e
WHERE s.sid = e.sid
AND e.grade = 'A'
```
Impossible / Unnecessary Predicates

SELECT * FROM A WHERE 1 = 0;
MORE EXAMPLES

Impossible / Unnecessary Predicates

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

SELECT * FROM A WHERE 1 = 0; X

SELECT * FROM A WHERE 1 = 1;
MORE EXAMPLES

Impossible / Unnecessary Predicates

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

SELECT * FROM A WHERE 1 = 0; X

SELECT * FROM A;
MORE EXAMPLES

Impossible / Unnecessary Predicates

- \[
\text{SELECT } * \text{ FROM } A \text{ WHERE } 1 = 0; \]
- \[
\text{SELECT } * \text{ FROM } A; \]

Join Elimination

- \[
\text{SELECT } A1.* \\
\text{FROM } A \text{ AS } A1 \text{ JOIN } A \text{ AS } A2 \\
\text{ON } A1.id = A2.id; \]
Impossible / Unnecessary Predicates

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

Join Elimination

SELECT * FROM A;
MORE EXAMPLES

Ignoring Projections

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

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

Ignoring Projections

```
SELECT * FROM A;
```

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

Source: Lukas Eder
MORE EXAMPLES

Ignoring Projections

```sql
SELECT * FROM A;
```

Merging Predicates

```sql
SELECT * FROM A
WHERE val BETWEEN 1 AND 100
  OR val BETWEEN 50 AND 150;
```
MORE EXAMPLES

Ignoring Projections

```sql
SELECT * FROM A;
```

Merging Predicates

```sql
SELECT * FROM A
WHERE val BETWEEN 1 AND 150;
```
Joins:
→ Commutative, associative

\[ R \bowtie S = S \bowtie R \]
\[ (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \]

How many different orderings are there for an \( n \)-way join?
How many different orderings are there for an $n$-way join?

**Catalan number $\approx 4^n$**

→ Exhaustive enumeration will be too slow.

We’ll see in a second how an optimizer limits the search space...
COST ESTIMATION

How long will a query take?
→ CPU: Small cost; tough to estimate
→ Disk: # of block transfers
→ Memory: Amount of DRAM used
→ Network: # of messages

How many tuples will be read/written?
What statistics do we need to keep?
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**
STATISTICS

For each relation $R$, the DBMS maintains the following information:

$\rightarrow N_R$: Number of tuples in $R$.

$\rightarrow V(A, R)$: Number of distinct values for attribute $A$. 
DERIVABLE STATISTICS

The selection cardinality $SC(A, R)$ is the average number of records with a value for an attribute $A$ given $N_R / V(A, R)$.

Note that this assumes *data uniformity*.

→ 10,000 students, 10 colleges – how many students in SCS?
SELECTION STATISTICS

Equality predicates on unique keys are easy to estimate.

What about more complex predicates? What is their selectivity?

```
SELECT * FROM people
WHERE id = 123
```

```
SELECT * FROM people
WHERE val > 1000
```

```
SELECT * FROM people
WHERE age = 30
AND status = 'Lit'
```
**COMPLEX PREDICATES**

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

Formula depends on type of predicate:
- → Equality
- → Range
- → Negation
- → Conjunction
- → Disjunction
The **selectivity** \( \text{sel} \) of a predicate \( P \) is the fraction of tuples that qualify.

**Formula depends on type of predicate:**
- Equality
- Range
- Negation
- Conjunction
- Disjunction
Assume that $V(\text{age, people})$ has five distinct values (0–4) and $N_R = 5$

Equality Predicate: $A = \text{constant}$

$\rightarrow \text{sel}(A = \text{constant}) = SC(P) / V(A, R)$

$\rightarrow$ Example: $\text{sel}(\text{age}=2) =$

```
SELECT * FROM people
WHERE age = 2
```
Assume that $V(\text{age, people})$ has five distinct values (0–4) and $N_R = 5$

Equality Predicate: $A=\text{constant}$

→ $\text{sel}(A=\text{constant}) = SC(P) / V(A, R)$

→ Example: $\text{sel}(\text{age}=2) =$

```
SELECT * FROM people
WHERE age = 2
```
Assume that $V(\text{age,people})$ has five distinct values (0–4) and $N_R = 5$

Equality Predicate: $A=\text{constant}$

$\rightarrow sel(A=\text{constant}) = SC(P) / V(A,R)$

$\rightarrow$ Example: $sel(\text{age}=2) = 1/5$

SELECT * FROM people
WHERE age = 2
Range Query:

→ sel(A>=a) = (A_{\text{max}} - a) / (A_{\text{max}} - A_{\text{min}})

→ Example: sel(age>=2) = (4 - 2) / (4 - 0)

= 1/2

SELECT * FROM people
WHERE age >= 2
Negation Query:

→ sel(not P) = 1 - sel(P)
→ Example: sel(age != 2)

SELECT * FROM people
WHERE age != 2
SELECTIONS – COMPLEX PREDICATES

Negation Query:
→ sel(not P) = 1 - sel(P)
→ Example: sel(age != 2) = 1 - (1/5) = 4/5

Observation: Selectivity ≈ Probability

SELECT * FROM people
WHERE age != 2
Conjunction:
→ \( \text{sel}(P1 \land P2) = \text{sel}(P1) \cdot \text{sel}(P2) \)
→ \( \text{sel}(\text{age}=2 \land \text{name LIKE 'A%'}) \)

This assumes that the predicates are independent.
SELECTIONS – COMPLEX PREDICATES

Conjunction:
\[ \Rightarrow \text{sel}(P_1 \land P_2) = \text{sel}(P_1) \cdot \text{sel}(P_2) \]
\[ \Rightarrow \text{sel}(\text{age}=2 \land \text{name LIKE 'A%'}) \]

This assumes that the predicates are independent.

```
SELECT * FROM people
WHERE age = 2
AND name LIKE 'A%'
```
Conjunction:

\[ \text{sel}(P1 \land P2) = \text{sel}(P1) \cdot \text{sel}(P2) \]

\[ \text{sel}(\text{age}=2 \land \text{name LIKE 'A%'}) \]

This assumes that the predicates are independent.

```
SELECT * FROM people
WHERE age = 2
AND name LIKE 'A%
```
SELECTIONS – COMPLEX PREDICATES

Disjunction:

\[ \text{sel}(P_1 \lor P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \lor P_2) \]

\[ \text{sel}(\text{age}=2 \ \text{OR} \ \text{name LIKE 'A%'}) \]

This again assumes that the selectivities are independent.
Disjunction:

\[ \rightarrow \text{sel}(P_1 \lor P_2) \]
\[ = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \lor P_2) \]
\[ = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \cdot \text{sel}(P_2) \]
\[ \rightarrow \text{sel}(\text{age}=2 \text{ OR name LIKE 'A%'}) \]

This again assumes that the selectivities are independent.
RESULT SIZE ESTIMATION FOR JOINS

Given a join of $R$ and $S$, what is the range of possible result sizes in # of tuples?

In other words, for a given tuple of $R$, how many tuples of $S$ will it match?
RESULT SIZE ESTIMATION FOR JOINS

General case: \( R_{\text{cols}} \cap S_{\text{cols}} = \{ A \} \) where \( A \) is not a key for either table.

→ Match each \( R \)-tuple with \( S \)-tuples:

\[
\text{estSize} \approx \frac{N_R \cdot N_S}{V(A,S)}
\]

→ Symmetrically, for \( S \):

\[
\text{estSize} \approx \frac{N_R \cdot N_S}{V(A,R)}
\]

Overall:

→ \( \text{estSize} \approx \frac{N_R \cdot N_S}{\max(V(A,S), V(A,R))} \)
COST ESTIMATIONS

Our formulas are nice but we assume that data values are uniformly distributed.

Uniform Approximation

# of occurrences

Distinct values of attribute
Our formulas are nice but we assume that data values are uniformly distributed.

Non-Uniform Approximation

Bucket Ranges:
- Bucket #1: 1-3, Count=8
- Bucket #2: 4-6, Count=4
- Bucket #3: 7-9, Count=15
- Bucket #4: 10-12, Count=3
- Bucket #5: 13-15, Count=14
HISTOGRAMS WITH QUANTILES

A histogram type wherein the "spread" of each bucket is same.

Equi-width Histogram (Quantiles)
HISTOGRAMS WITH QUANTILES

A histogram type wherein the "spread" of each bucket is same.

Equi-width Histogram (Quantiles)
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

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

<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>56</td>
<td>Rested</td>
</tr>
<tr>
<td>1002</td>
<td>Kanye</td>
<td>40</td>
<td>Weird</td>
</tr>
<tr>
<td>1003</td>
<td>Tupac</td>
<td>25</td>
<td>Dead</td>
</tr>
<tr>
<td>1004</td>
<td>Bieber</td>
<td>23</td>
<td>Crunk</td>
</tr>
<tr>
<td>1005</td>
<td>Andy</td>
<td>37</td>
<td>Lit</td>
</tr>
</tbody>
</table>

1 billion tuples
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

\[
\text{sel}(\text{age} > 50) = \\
\begin{array}{ccc}
1001 & Obama & 56 \\
1003 & Tupac & 25 \\
1005 & Andy & 37 \\
\end{array}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{AVG(age)} \\
\text{FROM} & \quad \text{people} \\
\text{WHERE} & \quad \text{age} > 50
\end{align*}
\]
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

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

\[
\begin{align*}
\text{SELECT} & \quad \text{AVG}(\text{age}) \\
\text{FROM} & \quad \text{people} \\
\text{WHERE} & \quad \text{age} > 50
\end{align*}
\]

<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>56</td>
<td>Rested</td>
</tr>
<tr>
<td>1002</td>
<td>Kanye</td>
<td>40</td>
<td>Weird</td>
</tr>
<tr>
<td>1003</td>
<td>Tupac</td>
<td>25</td>
<td>Dead</td>
</tr>
<tr>
<td>1004</td>
<td>Bieber</td>
<td>23</td>
<td>Crunk</td>
</tr>
<tr>
<td>1005</td>
<td>Andy</td>
<td>37</td>
<td>Lit</td>
</tr>
</tbody>
</table>

1 billion tuples
OBSERVATION

Now that we can (roughly) estimate the selectivity of predicates, what can we actually do with them?
After performing rule-based rewriting, the DBMS will 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

Simple heuristics are often good enough for this.
OLTP queries are especially easy.
Query planning for OLTP queries is easy because they are *sargable*.

→ Search *Argument Able*

→ It is usually just picking the best index.

→ Joins are almost always on foreign key relationships with a small cardinality.

→ Can be implemented with simple heuristics.
MULTI-RELATION QUERY PLANNING

As number of joins increases, number of alternative plans grows rapidly → We need to restrict search space.

Fundamental decision in System R: only left-deep join trees are considered. → Modern DBMSs do not always make this assumption anymore.
MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.
MULTI-RELATION QUERY PLANNING

Fundamental decision in System R: Only consider left-deep join trees.
MULTI-RELATION QUERY PLANNING

Fundamental decision in System R: Only consider left-deep join trees.

Allows for fully pipelined plans where intermediate results are not written to temp files. → Not all left-deep trees are fully pipelined.
MULTI-RELATION QUERY PLANNING

Enumerate the orderings
→ Example: Left-deep tree #1, Left-deep tree #2…

Enumerate the plans for each operator
→ Example: Hash, Sort-Merge, Nested Loop…

Enumerate the access paths for each table
→ Example: Index #1, Index #2, Seq Scan…

Use **dynamic programming** to reduce the number of cost estimations.
Dynamic Programming

Hash Join
R.a = S.a

SortMerge Join
R.a = S.a

Hash Join
T.b = S.b

SortMerge Join
T.b = S.b

Hash Join
T.b = S.b

SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
Hash Join
R.a = S.a

Hash Join
T.b = S.b

SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
DYNAMIC PROGRAMMING

SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
CANDIDATE PLAN EXAMPLE

How to generate plans for search algorithm:
→ Enumerate relation orderings
→ Enumerate join algorithm choices
→ Enumerate access method choices

No real DBMSs does it this way. It’s actually more messy…

```sql
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
```
CANDIDATE PLANS

Step #1: Enumerate relation orderings
CANDIDATE PLANS

Step #1: Enumerate relation orderings

Prune plans with cross-products immediately!
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices

Do this for the other plans.
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices

Do this for the other plans.
CANDIDATE PLANS

Step #3: Enumerate access method choices

Do this for the other plans.
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 result to 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 = '2018-10-15'  
)

SELECT name  
FROM sailors AS S, reserves AS R  
WHERE S.sid = R.sid  
AND R.day = '2018-10-15'
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(*) > 1
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 a temporary table that are discarded after the query finishes.
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 Block
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 = (SELECT MAX(S2.rating) FROM sailors S2)
GROUP BY S.sid
HAVING COUNT(*) > 1
### DECOMPOSING QUERIES

<table>
<thead>
<tr>
<th>SQL Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT MAX(rating) FROM sailors</td>
</tr>
<tr>
<td>SELECT S.sid, MIN(R.day)</td>
</tr>
<tr>
<td>FROM sailors S, reserves R, boats B</td>
</tr>
<tr>
<td>WHERE S.sid = R.sid</td>
</tr>
<tr>
<td>AND R.bid = B.bid</td>
</tr>
<tr>
<td>AND B.color = 'red'</td>
</tr>
<tr>
<td>AND S.rating = ###</td>
</tr>
<tr>
<td>GROUP BY S.sid</td>
</tr>
<tr>
<td>HAVING COUNT(*) &gt; 1</td>
</tr>
</tbody>
</table>

---

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

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

Outer Block
CONCLUSION

Filter early as possible.
Selectivity estimations
→ Uniformity
→ Independence
→ Histograms
→ Join selectivity
Dynamic programming for join orderings
Rewrite nested queries
Query optimization is really hard…
Midterm Exam

**Who:** You

**What:** Midterm Exam

**When:** Wed Oct 17th 12:00pm - 1:20pm

**Where:** Posner Mellon Auditorium

**Why:** [https://youtu.be/xgMiaIPxSlc](https://youtu.be/xgMiaIPxSlc)
MIDTERM

What to bring:
→ CMU ID
→ Calculator
→ One 8.5x11" page of notes (double-sided)

What not to bring:
→ Live animals
→ Your wet laundry
MIDTERM

Covers up to Joins (inclusive).
→ Closed book, one sheet of notes (double-sided)
→ Please email Andy if you need special accommodations.

https://15445.courses.cs.cmu.edu/fall2018/midterm-guide.html
RELATIONAL MODEL

Integrity Constraints
Relation Algebra
SQL

Basic operations:
→ SELECT / INSERT / UPDATE / DELETE
→ WHERE predicates
→ Output control

More complex operations:
→ Joins
→ Aggregates
→ Common Table Expressions
STORAGE

Buffer Management Policies
→ LRU / MRU / CLOCK

On-Disk File Organization
→ Heaps
→ Linked Lists

Page Layout
→ Slotted Pages
→ Log-Structured
HASHING

Static Hashing
→ Linear Probing
→ Robin Hood
→ Cuckoo Hashing

Dynamic Hashing
→ Extendible Hashing
→ Linear Hashing

Comparison with B+Trees
TREE INDEXES

B+Tree
→ Insertions / Deletions
→ Splits / Merges
→ Difference with B-Tree
→ Latch Crabbing / Coupling

Radix Trees

Skip Lists
SORTING

Two-way External Merge Sort
General External Merge Sort
Cost to sort different data sets with different number of buffers.
QUERY PROCESSING

Processing Models
→ Advantages / Disadvantages

Join Algorithms
→ Nested Loop
→ Sort-Merge
→ Hash
NEXT CLASS

Parallel Query Execution