TODAY’S AGENDA

Background
Implementation Design Decisions
Optimizer Search Strategies
QUERY OPTIMIZATION

For a given query, find a **correct** execution plan that has the lowest "**cost**".

This is the part of a DBMS that is the hardest to implement well (proven to be NP-Complete).

No optimizer truly produces the "optimal" plan
→ Use estimation techniques to guess real plan cost.
→ Use heuristics to limit the search space.
ARCHITECTURE OVERVIEW

SQL Query → SQL Rewriter (Optional) → System Catalog

SQL Query → Parser → Abstract Syntax Tree

SQL Rewriter (Optional) → Annotated AST

System Catalog → Binder → Annotated AST

Binder → Tree Rewriter (Optional) → Annotated AST

Tree Rewriter (Optional) → Optimizer

Optimizer → Cost Estimates

Optimizer → Physical Plan

Parser

Abstract Syntax Tree
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 a particular 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.
RELATIONAL ALGEBRA EQUIVALENCES

Two relational algebra expressions are said to be equivalent if on every legal database instance the two expressions generate the same set of tuples.

Example: \((A \bowtie (B \bowtie C)) = (B \bowtie (A \bowtie C))\)
OBSERVATION

Query planning for OLTP queries is easy because they are **sargable**.
→ 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.

We will focus on OLAP queries in this lecture.
COST ESTIMATION

Generate an estimate of the cost of executing a plan for the current state of the database.
→ Interactions with other work in DBMS
→ Size of intermediate results
→ Choices of algorithms, access methods
→ Resource utilization (CPU, I/O, network)
→ Data properties (skew, order, placement)

We will discuss this more next week…
DESIGN DECISIONS

Optimization Granularity
Optimization Timing
Prepared Statements
Plan Stability
Choice #1: Single Query
→ Much smaller search space.
→ DBMS cannot reuse results across queries.
→ In order to account for resource contention, the cost model must account for what is currently running.

Choice #2: Multiple Queries
→ More efficient if there are many similar queries.
→ Search space is much larger.
→ Useful for scan sharing.
**OPTIMIZATION TIMING**

**Choice #1: Static Optimization**
- Select the best plan prior to execution.
- Plan quality is dependent on cost model accuracy.
- Can amortize over executions with prepared stmts.

**Choice #2: Dynamic Optimization**
- Select operator plans on-the-fly as queries execute.
- Will have re-optimize for multiple executions.
- Difficult to implement/debug (non-deterministic)

**Choice #3: Hybrid Optimization**
- Compile using a static algorithm.
- If the error in estimate > threshold, re-optimize
PREPARED STATEMENTS

```
SELECT A.id, B.val
FROM A, B, C
WHERE A.id = B.id
   AND B.id = C.id
   AND A.val > 100
   AND B.val > 99
   AND C.val > 5000
```
PREPARE myQuery AS
  SELECT A.id, B.val
  FROM A, B, C
  WHERE A.id = B.id
  AND B.id = C.id
  AND A.val > 100
  AND B.val > 99
  AND C.val > 5000
EXECUTE myQuery;
What should be the join order for A, B, and C?

```sql
PREPARE myQuery(int, int, int) AS
SELECT A.id, B.val
FROM A, B, C
WHERE A.id = B.id
  AND B.id = C.id
  AND A.val > ?
  AND B.val > ?
  AND C.val > ?

EXECUTE myQuery(100, 99, 5000);
```
PREPARED STATEMENTS

Choice #1: Re-Optimize
→ Rerun optimizer each time the query is invoked.
→ Tricky to reuse existing plan as starting point.

Choice #2: Multiple Plans
→ Generate multiple plans for different values of the parameters (e.g., buckets).

Choice #3: Average Plan
→ Choose the average value for a parameter and use that for all invocations.
PLAN STABILITY

Choice #1: Hints
→ Allow the DBA to provide hints to the optimizer.

Choice #2: Fixed Optimizer Versions
→ Set the optimizer version number and migrate queries one-by-one to the new optimizer.

Choice #3: Backwards-Compatible Plans
→ Save query plan from old version and provide it to the new DBMS.
OPTIMIZATION SEARCH STRATEGIES

Heuristics
Heuristics + Cost-based Join Order Search
Randomized Algorithms
Stratified Search
Unified Search
HEURISTIC-BASED OPTIMIZATION

Define static rules that transform logical operators to a physical plan.

→ Perform most restrictive selection early
→ Perform all selections before joins
→ Predicate/Limit/Projection pushdowns
→ Join ordering based on cardinality

Example: Original versions of INGRES and Oracle (until mid 1990s)
CREATE TABLE ARTIST (  
   ID INT PRIMARY KEY,  
   NAME VARCHAR(32)  
);  

CREATE TABLE ALBUM (  
   ID INT PRIMARY KEY,  
   NAME VARCHAR(32) UNIQUE  
);  

CREATE TABLE APPEARS (  
   ARTIST_ID INT  
   REFERENCES ARTIST(ID),  
   ALBUM_ID INT  
   REFERENCES ALBUM(ID),  
   PRIMARY KEY  
   (ARTIST_ID, ALBUM_ID)  
);
Retrieve the names of people that appear on Andy's mixtape

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
AND APPEARS.ALBUM_ID=ALBUM.ID
AND ALBUM.NAME="Andy's OG Remix"
```
Step #1: Decompose into single-variable queries

SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
AND APPEARS.ALBUM_ID=ALBUM.ID
AND ALBUM.NAME="Andy's OG Remix"

Retrieve the names of people that appear on Andy's mixtape

Q1
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
FROM ALBUM
WHERE ALBUM.NAME="Andy's OG Remix"

Q2
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, TEMP1
WHERE ARTIST.ID=APPEARS.ARTIST_ID
AND APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
INGRES OPTIMIZER

Retrieve the names of people that appear on Andy's mixtape

**Step #1: Decompose into single-variable queries**

**Q1**

```sql
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
FROM ALBUM
WHERE ALBUM.NAME = "Andy's OG Remix"
```

**Q3**

```sql
SELECT APPEARS.ARTIST_ID INTO TEMP2
FROM APPEARS, TEMP1
WHERE APPEARS.ALBUM_ID = TEMP1.ALBUM_ID
```

**Q4**

```sql
SELECT ARTIST.NAME
FROM ARTIST, TEMP2
WHERE ARTIST.ARTIST_ID = TEMP2.ARTIST_ID
```
Retrieve the names of people that appear on Andy's mixtape

**Q1**

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
FROM ALBUM
WHERE ALBUM.NAME="Andy's OG Remix"
```

**Q3**

```
SELECT APPEARS.ARTIST_ID INTO TEMP2
FROM APPEARS, TEMP1
WHERE APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
```

**Q4**

```
SELECT ARTIST.NAME
FROM ARTIST, TEMP2
WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

**Step #1: Decompose into single-variable queries**

**Step #2: Substitute the values from Q1→Q3→Q4**
Retrieve the names of people that appear on Andy's mixtape

**Step #1: Decompose into single-variable queries**

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
    AND APPEARS.ALBUM_ID=ALBUM.ID
    AND ALBUM.NAME="Andy's OG Remix"
```

**Step #2: Substitute the values from Q1→Q3→Q4**

```
SELECT APPEARS.ARTIST_ID
FROM APPEARS
WHERE APPEARS.ALBUM_ID=9999
```

Q4

```
SELECT ARTIST.NAME
FROM ARTIST, TEMP2
WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

ALBUM_ID

9999
Retrieve the names of people that appear on Andy's mixtape

\[
\begin{align*}
\text{SELECT} & \quad \text{ARTIST.NAME} \\
\text{FROM} & \quad \text{ARTIST, APPEARS, ALBUM} \\
\text{WHERE} & \quad \text{ARTIST.ID=APPEARS.ARTIST_ID} \\
\text{AND} & \quad \text{APPEARS.ALBUM_ID=ALBUM.ID} \\
\text{AND} & \quad \text{ALBUM.NAME="Andy's OG Remix"}
\end{align*}
\]

**Step #1: Decompose into single-variable queries**

**Step #2: Substitute the values from Q1→Q3→Q4**
Retrieve the names of people that appear on Andy's mixtape

\[
\text{SELECT ARTIST.NAME} \\
\text{FROM ARTIST, APPEARS, ALBUM} \\
\text{WHERE ARTIST.ID=APPEARS.ARTIST_ID} \\
\text{AND APPEARS.ALBUM_ID=ALBUM.ID} \\
\text{AND ALBUM.NAME=}"Andy's OG Remix""
\]

**Step #1: Decompose into single-variable queries**

**Step #2: Substitute the values from Q1→Q3→Q4**
HEURISTIC-BASED OPTIMIZATION

Advantages:
→ Easy to implement and debug.
→ Works reasonably well and is fast for simple queries.

Disadvantages:
→ Relies on magic constants that predict the efficacy of a planning decision.
→ Nearly impossible to generate good plans when operators have complex inter-dependencies.
HEURISTICS + COST-BASED JOIN SEARCH

Use static rules to perform initial optimization. Then use dynamic programming to determine the best join order for tables.

→ First cost-based query optimizer
→ **Bottom-up planning** (forward chaining) using a divide-and-conquer search method

Example: System R, early IBM DB2, most open-source DBMSs
SYSTEM R OPTIMIZER

Break query up 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” tree that minimizes the estimated amount of work to execute the plan.
SYSTEM R OPTIMIZER

Retrieve the names of people that appear on Andy’s mixtape ordered by their artist id.

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
    AND APPEARS.ALBUM_ID=ALBUM.ID
    AND ALBUM.NAME="Andy's OG Remix"
ORDER BY ARTIST.ID
```

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

- **ARTIST**: Sequential Scan
- **APPEARS**: Sequential Scan
- **ALBUM**: Index Look-up on NAME
Retrieve the names of people that appear on Andy’s mixtape ordered by their artist id.

```sql
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
 AND APPEARS.ALBUM_ID=ALBUM.ID
 AND ALBUM.NAME="Andy's OG Remix"
ORDER BY ARTIST.ID
```

**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 × APPEARS × ALBUM
- APPEARS × ALBUM × ARTIST
- ALBUM × APPEARS × ARTIST
- APPEARS × ARTIST × ALBUM
- ARTIST × ALBUM × APPEARS
- ALBUM × ARTIST × APPEARS
- ...
SYSTEM R OPTIMIZER

Retrieve the names of people that appear on Andy's mixtape ordered by their artist id.

```sql
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
AND APPEARS.ALBUM_ID=ALBUM.ID
AND ALBUM.NAME="Andy's OG Remix"
ORDER BY ARTIST.ID
```

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

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

ARTIST ⇚ APPEARS ⇚ ALBUM

ARTIST ⇚ APPEARS ⇚ ALBUM

ALBUM ⇚ APPEARS ⇚ ARTIST
**SYSTEM R OPTIMIZER**

Hash Join
ARTIST.ID=APPEARS.ARTIST_ID

SortMerge Join
ARTIST.ID=APPEARS.ARTIST_ID

SortMerge Join
ALBUM.ID=APPEARS.ALBUM_ID

Hash Join
ALBUM.ID=APPEARS.ALBUM_ID

• • •
Hash Join
ARTIST.ID=APPEARS.ARTIST_ID

Hash Join
ALBUM.ID=APPEARS.ALBUM_ID
Hash Join
ARTIST.ID=APPEARS.ARTIST_ID

Hash Join
APPEARS.ALBUM_ID=ALBUM.ID

SortMerge Join
APPEARS.ALBUM_ID=ALBUM.ID

SortMerge Join
APPEARS.ARTIST_ID=ARTIST.ID

Hash Join
APPEARS.ARTIST_ID=ARTIST.ID

Hash Join
ARTIST.ID=APPEARS.ARTIST_ID
The query uses `ORDER BY` on `ARTIST.ID` but the logical plans do not contain any information about sorting.
The query uses ORDER BY on ARTIST.ID but the logical plans do not contain any information about sorting.
HEURISTICS + COST-BASED JOIN SEARCH

Advantages:
→ Usually finds a reasonable plan without having to perform an exhaustive search.

Disadvantages:
→ All the same problems as the heuristic-only approach.
→ Left-deep join trees are not always optimal.
→ Have to take in consideration the physical properties of data in the cost model (e.g., sort order).
RANDOMIZED ALGORITHMS

Perform a random walk over a solution space of all possible (valid) plans for a query.

Continue searching until a cost threshold is reached or the optimizer runs for a particular length of time.

Example: Postgres’ genetic algorithm.
SIMULATED ANNEALING

Start with a query plan that is generated using the heuristic-only approach.

Compute random permutations of operators (e.g., swap the join order of two tables)
→ Always accept a change that reduces cost
→ Only accept a change that increases cost with some probability.
→ Reject any change that violates correctness (e.g., sort ordering)
More complicated queries use a **genetic algorithm** that selects join orderings.

At the beginning of each round, generate different variants of the query plan.

Select the plans that have the lowest cost and permute them with other plans. Repeat.

→ The mutator function only generates valid plans.

Source: [Postgres Documentation](https://www.postgresql.org)
RANDOMIZED ALGORITHMS

Advantages:
→ Jumping around the search space randomly allows the optimizer to get out of local minimums.
→ Low memory overhead (if no history is kept).

Disadvantages:
→ Difficult to determine why the DBMS may have chosen a particular plan.
→ Have to do extra work to ensure that query plans are deterministic.
→ Still have to implement correctness rules.
Observation

Writing query transformation rules in a procedural language is hard and error-prone.

→ No easy way to verify that the rules are correct without running a lot of fuzz tests.
→ Generation of physical operators per logical operator is decoupled from deeper semantics about query.

A better approach is to use a declarative DSL to write the transformation rules and then have the optimizer enforce them during planning.
OPTIMIZER GENERATORS

Use a rule engine that allows transformations to modify the query plan operators. The physical properties of data is embedded with the operators themselves.

Choice #1: Stratified Search
→ Planning is done in multiple stages

Choice #2: Unified Search
→ Perform query planning all at once.
STRATIFIED SEARCH

First rewrite the logical query plan using transformation rules.
→ The engine checks whether the transformation is allowed before it can be applied.
→ Cost is never considered in this step.

Then perform a cost-based search to map the logical plan to a physical plan.
STARBURST OPTIMIZER

Better implementation of the System R optimizer that uses declarative rules.

**Stage #1: Query Rewrite**
→ Compute a SQL-block-level, relational calculus-like representation of queries.

**Stage #2: Plan Optimization**
→ Execute a System R-style dynamic programming phase once query rewrite has completed.

**Example: Latest version of IBM DB2**
STARBURST OPTIMIZER

**Advantages:**
→ Works well in practice with fast performance.

**Disadvantages:**
→ Difficult to assign priorities to transformations
→ Some transformations are difficult to assess without computing multiple cost estimations.
→ Rules maintenance is a huge pain.
UNIFIED SEARCH

Unify the notion of both logical→logical and logical→physical transformations. → No need for separate stages because everything is transformations.

This approach generates a lot more transformations so it makes heavy use of memoization to reduce redundant work.
VOLCANO OPTIMIZER

General purpose cost-based query optimizer, based on equivalence rules on algebras.
→ Easily add new operations and equivalence rules.
→ Treats physical properties of data as first-class entities during planning.
→ **Top-down approach** (backward chaining) using branch-and-bound search.

Example: Academic prototypes
**TOP-DOWN VS. BOTTOM-UP**

**Top-down Optimization**
→ Start with the final outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal.
→ Example: Volcano, Cascades

**Bottom-up Optimization**
→ Start with nothing and then build up the plan to get to the final outcome that you want.
→ Examples: System R, Starburst
Start with a logical plan of what we want the query to be.

**ARTIST** $\Join$ **APPEARS** $\Join$ **ALBUM**

**ORDER-BY(ARTIST.ID)**
VOLCANO OPTIMIZER

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

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

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

→ Logical → Physical:
  JOIN(A,B) to HASH_JOIN(A,B)
Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

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

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

→ Logical→Physical:
  JOIN(A,B) to HASHJOIN(A,B)
VOLCANO OPTIMIZER

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse 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.
Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse 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.
**VOLCANO OPTIMIZER**

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse 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.
Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse 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.
The optimizer needs to enumerate all possible transformations without repeating.
Go from logical to physical plan as fast as possible, then try alternative plans.
→ Use a top-down rules engine that performs branch-and-bound pruning.
→ Use memoization to cache equivalent operators.
VOLCANO OPTIMIZER

Advantages:
→ Use declarative rules to generate transformations.
→ Better extensibility with an efficient search engine.
  Reduce redundant estimations using memoization.

Disadvantages:
→ All equivalence classes are completely expanded to
generate all possible logical operators before the
optimization search.
→ Not easy to modify predicates.
PARTING THOUGHTS

Query optimization is **hard**.

This is why the NoSQL systems didn’t implement it (at first).
NEXT CLASS

Optimizers! First Blood, Part II

Cascades / Orca / Columbia