Last Class

Row-oriented database network protocols via JDBC/ODBC APIs are sufficient for queries that access a small number of tuples.

Large output queries / bulk export operations benefit from Arrow native columnar optimizations via ADBC.
NEXT TWO WEEKS

Optimizer Implementations
Query Rewriting
Plan Enumerations
Cost Models
For a given query, find a **correct** physical execution plan for that query with 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.
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.
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...
TODAY’S AGENDA

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

Define static rules that transform logical operators to a physical plan without a cost model.
→ Perform most restrictive selection early
→ Perform all selections before joins
→ Predicate/Limit/Projection pushdowns
→ Join ordering based on simple rules or cardinality estimates

Examples: INGRES (until mid-1980s) and Oracle (until mid-1990s), MongoDB, most new DBMSs.
LOGICAL QUERY OPTIMIZATION

Split Conjunctive Predicates
Predicate Pushdown
Replace Cartesian Products with Joins
Projection Pushdown

Source: Thomas Neumann
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.

```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"
```
SPLIT CONJUNCTIVE PREDICATES

Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.

```
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"
```
PREDICATE PUSHDOWN

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"

Move the predicate to the lowest point in the plan after Cartesian products.
Move the predicate to the lowest point in the plan after Cartesian products.

```
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"
```
Replace all Cartesian Products with inner joins using the join predicates.

```
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"
```
Replace all Cartesian Products with inner joins using the join predicates.

```
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"
```
Eliminate redundant attributes before pipeline breakers to reduce materialization cost.
Eliminate redundant attributes before pipeline breakers to reduce materialization cost.
Retrieve the names of people that appear on Andy's mixtape ordered by their artist id.

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

**Query #1**

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

**Query #2**

```sql
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, TEMP1
WHERE ARTIST.ID=APPEARS.ARTIST_ID
AND APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
AND ALBUM.NAME="Andy's OG Remix"
ORDER BY ARTIST.ID
```

---

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

Step #1: Decompose into single-value queries

Query #1

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

Query #2

```
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
```
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: Decompose into single-value queries**

**Query #1**

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

**Query #3**

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

**Query #4**

```
SELECT ARTIST.NAME
FROM ARTIST, TEMP2
WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```
Retrieve the names of people that appear on Andy's mixtape ordered by their artist id.

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

**Query #3**

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

**Step #2: Substitute the values from Query#1 → Query #3 → Query #4**

**Query #4**

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

**ALBUM_ID**

9999
INGRES OPTIMIZER

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

Step #1: Decompose into single-value queries

Step #2: Substitute the values from Query#1 → Query #3 → Query #4

Query #4
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: Decompose into single-value queries**

**Step #2: Substitute the values from**

Query #1 \(\rightarrow\) Query #3 \(\rightarrow\) Query #4

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

**Query #4**

```
SELECT ARTIST.NAME
FROM ARTIST, TEMP2
WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```
Retrieve the names of people that appear on Andy’s mixtape ordered by their artist id.

Step #1: Decompose into single-value queries

Step #2: Substitute the values from Query#1 → Query #3 → Query #4
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: Decompose into single-value queries**

**Step #2: Substitute the values from**

Query #1 → Query #3 → Query #4
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.
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 planning decisions.
→ Nearly impossible to generate good plans when operators have complex inter-dependencies.

Stonebraker gave the story of the query optimizer as an example. Relational queries were often highly complex. Let’s say you wanted your database to give you the name, salary, and job title of everyone in your Chicago office who did the same kind of work as an employee named Alien. (This example happens to come from Oracle’s 1981 user guide.) This would require the database to find information in the employee table and the department table, then sort the data. How quickly the database management system did this depended on how cleverly the system was constructed. “If you do it smart, you get the answer a lot quicker than if you do it stupid,” Stonebraker said.

He continued, “Oracle had a really stupid optimizer. They did the query in the order that you happened to type in the clauses. Basically, they blindly did it from left to right. The Ingres program looked at everything there and tried to figure out the best way to do it.” But Ellison found a way to neutralize this advantage. Stonebraker said, “Oracle was really shrewd. They said they had a syntactic optimizer, whereas the other guys had a semantic optimizer. The truth was, they had no optimizer and the other guys had an optimizer. It was very, very, very creative marketing. . . . They were very good at confusing the market.”

“What he’s using is semantics himself,” Ellison said. Just because Oracle did things differently, “Stonebraker decided we didn’t have an optimizer. [He seemed to think] the only kind of optimizer was his optimizer, and our approach to optimization wasn’t really optimization at all. That’s an interesting notion, but I’m not sure I buy that.”
HEURISTICS + COST-BASED SEARCH

First use static rules to perform initial logical→logical optimizations. Then enumerate plans using physical→logical transformations to find best plan according to a cost model.

Examples: System R, early IBM DB2, most open-source DBMSs.
PHYSICAL QUERY OPTIMIZATION

Transform a query plan's logical operators into physical operators.
→ Add more execution information
→ Select indexes / access paths
→ Choose operator implementations
→ Choose when to materialize (i.e., temp tables).

This stage must support cost model estimates.
PLAN ENUMERATION

Approach #1: Generative / Bottom-Up
→ Start with nothing and then iteratively assemble and add building blocks to generate a query plan.
→ **Examples:** System R, Starburst

Approach #2: Transformation / Top-Down
→ Start with the outcome that the query wants, and then transform it to equivalent alternative sub-plans to find the optimal plan that gets to that goal.
→ **Examples:** Volcano, Cascades
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" join tree that minimizes the estimated amount of work to execute the plan.
Retrieves the names of people that appear on Andy's mixtape ordered by their artist id.

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

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

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

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

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

```
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.

```
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**

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

**Step #3: Determine the join ordering with the lowest cost**

**ARTIST**: Sequential Scan

**APPEARS**: Sequential Scan

**ALBUM**: Index Look-up on NAME

ARTIST  XX  APPEARS  XX  ALBUM
APPEARS  XX  ALBUM  XX  ARTIST
ALBUM  XX  APPEARS  XX  ARTIST
APPEARS  XX  ARTIST  XX  ALBUM
ARTIST  ×  ALBUM  XX  APPEARS
ALBUM  ×  ARTIST  XX  APPEARS
⋮  ⋮   ⋮   ⋮
SYSTEM R OPTIMIZER

ARTIST ⋈ APPEARS ⋈ ALBUM

HASH_JOIN(A1, A3) -> MERGE_JOIN(A1, A3)
HASH_JOIN(A2, A3) -> MERGE_JOIN(A2, A3)
HASH_JOIN(A3, A2) -> MERGE_JOIN(A3, A2)

ARTIST.ID = APPEARS.ARTIST_ID
ALBUM.ID = APPEARS.ALBUM_ID
APPEARS.ALBUM_ID = ALBUM.ID
SYSTEM R OPTIMIZER

**Logical Op**

**Physical Op**

 ARTIST  ❁ APPEARS  ❁ ALBUM

---

**ARTIST**

**APPEARS**

**ALBUM**

---

**HASH JOIN(A1, A3)**

**ARTIST**

**APPEARS**

**ALBUM**

---

**HASH JOIN(A2, A3)**

**ARTIST**

**APPEARS**

**ALBUM**

---

**MERGE JOIN(A3, A2)**

**ARTIST**

**APPEARS**

**ALBUM**

---

ARTIST.ID = APPEARS.ARTIST_ID

ALBUM.ID = APPEARS.ALBUM_ID

APPEARS.ALBUM_ID = ALBUM.ID

---

CMU-DB

15-721 (Spring 2024)
Logical Op

Artists appear on an album.

$\text{ARTIST} \bowtie \text{APPEARS} \bowtie \text{ALBUM}$

Logical Op

$\text{HASH}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{MERGE}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{HASH}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{HASH}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$

Physical Op

$\text{ARTIST} \bowtie \text{APPEARS} \bowtie \text{ALBUM}$

Logical Op

$\text{HASH}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{MERGE}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{HASH}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{HASH}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$

Physical Op

$\text{ARTIST} \bowtie \text{APPEARS} \bowtie \text{ALBUM}$

Logical Op

$\text{HASH}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{MERGE}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{HASH}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{HASH}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$

Physical Op

$\text{ARTIST} \bowtie \text{APPEARS} \bowtie \text{ALBUM}$

Logical Op

$\text{HASH}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{MERGE}\_\text{JOIN}(\text{A1} \bowtie \text{A3}, \text{A2})$

$\text{HASH}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A2} \bowtie \text{A3}, \text{A1})$

$\text{HASH}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$

$\text{MERGE}\_\text{JOIN}(\text{A3} \bowtie \text{A2}, \text{A1})$
Logical Op

Physical Op

SYSTEM R OPTIMIZER

```
HASH_JOIN(A2⨝A3,A1)

APPEARS.ARTIST_ID=ARTIST.ID

ALBUM⨝APPEARS
ARTIST

HASH_JOIN(A2,A3)

ALBUM.ID=APPEARS.ALBUM_ID

ARTIST ALBUM APPEARS
```
The query has **ORDER BY** on **ARTIST.ID** but the logical plans do **not** contain sorting properties.

**Hack:** Keep track of best plans with and without data in proper physical form, and then check whether tacking on a sort operator at the end is better.
SEARCH TERMINATION

Approach #1: Wall-clock Time
→ Stop after the optimizer runs for some length of time.

Approach #2: Cost Threshold
→ Stop when the optimizer finds a plan that has a lower cost than some threshold.

Approach #3: Exhaustion
→ Stop when there are no more enumerations of the target plan. Usually done per sub-plan/group.

Approach #4: Transformation Count
→ Stop after a certain number of transformations have been considered.
HEURISTICS + COST-BASED 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.
→ Must take in consideration the physical properties of data in the cost model (e.g., sort order).
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

Framework to allow a DBMS implementer to write the declarative rules for optimizing queries.

→ Separate the search strategy from the data model.
→ Separate the transformation rules and logical operators from physical rules and physical operators.

The implementation of the optimizer's pattern matching method and transformation rules can be independent of its search strategy.
OPTIMIZER GENERATORS

Choice #1: Stratified Search
→ Planning is done in multiple stages (heuristics then cost-based search).
→ Examples: Starburst, CockroachDB

Choice #2: Unified Search
→ Perform query planning all at once.
→ Examples: Cascades, OPT++, SQL Server
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.
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 (bottoms-up) dynamic programming phase once query rewrite has completed.

**Example:** Latest version of IBM DB2
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 (bottoms-up) dynamic programming phase once query rewrite has completed.

Example: Latest version of IBM DB2

Perspectives

Four DB2 Code Bases?

Many years ago I worked on IBM DB2 and so I occasionally get the question, “how could you folks possibly have four relational database management system code bases?” Some go on to argue that a single code base would have been much more efficient. That’s certainly true. And, had we moved to a single code base, that engineering resource efficiency improvement would have led to a very different outcome in the database wars. I’m skeptical on this extension of the argument but the question is an interesting one and I wrote up a more detailed answer than usually possible off the cuff.

IBM Relational Database Code Bases

Few server manufacturers have the inclination and the resources needed to develop a relational database management system and yet IBM has internally developed and continues to support four independent, full-featured relational database products. A production-quality RDBMS with a large customer base is typically well over a million lines of code and represents a multi-year effort of hundreds and, in some cases, thousands of engineers. These are massive undertakings requiring special skills, so the question sometimes comes up, how could IBM possibly end up with four different RDBMS systems that don’t share components?

At least while I was at IBM, there was frequent talk of developing a single RDBMS code base for all supported hardware and operating systems. The reasons why this didn’t happen are at least partly social and historical, but there are also many strong technical challenges that make it difficult to reduce the clock and use a single code base. The diversity of the IBM hardware and operating platforms would have made it difficult, the deep exploitation of unique underlying platform characteristics like the single level cache on the AS/400 or the Systplex Data Sharing on System z would make it truly challenging, the implementation languages used by many of the RDBMS code bases...
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 (bottoms-up) dynamic programming phase once query rewrite has completed.

Example: Latest version of IBM DB2

There was a lot to be done and very little time. The pressure was mounting and we were looking at other solutions from a variety of different sources when the IBM Almaden database research team jumped in. They offered to put the entire Almaden database research team on the project, with a goal to both replace the OS/2 DBM optimizer and execution engine with Starburst (Database research project) components and to help solve the scaling and stability problems we were currently experiencing in the field. Taking a research code base is a dangerous step for any development team, but this proposal was different in that the authors would accompany the code base. Pat Selinger of IBM Almaden Research essentially convinced us that we would have a world-class optimizer and execution engine and we would have the full-time commitment from Pat, Bruce Lindsay, Guy Lohman, C. Mohan, Hamid Pirahesh, John McPherson and the rest of the IBM Almaden database research team working shoulder to shoulder with us in making this product successful.

Many years ago I worked on IBM DB2 and so I occasionally get the question, “how the heck could you folks possibly have four relational database management system code bases?” Some go on to argue that a single code base would have been much more efficient. That’s certainly true. And, had we moved to a single code base, that engineering resource efficiency improvement would have led to a very different outcome in the database wars. I’m skeptical on this extension of the argument but the question is an interesting one and I wrote up a more detailed answer than usually possible off the cuff.
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 because they are written in IBM's Query Graph Model (QGM) DSL.
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 many 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
Start with a logical plan of what we want the query to be.
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)
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)
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)
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)
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.
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.
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.
TOP-DOWN OPTIMIZATION

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**

**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.
CASCADES OPTIMIZER

Object-oriented implementation of the previous Volcano query optimizer.
→ **Top-down approach** (backward chaining) using branch-and-bound search.

Supports expression re-writing through a direct mapping function rather than an exhaustive search.
CASCADES: KEY IDEAS

Optimization tasks as data structures.
→ Patterns to match + Transformation Rule to apply

Rules to place property enforcers.
→ Ensures the optimizer generates correct plans.

Ordering of moves by promise.
→ Dynamic task priorities to find optimal plan more quickly.

Predicates as logical/physical operators.
→ Use same pattern/rule engine for expressions.
CASCades: Expressions

An **expression** represents some operation in the query with zero or more input expressions. 

→ Optimizer needs to quickly determine whether two expressions are equivalent.

**Logical Expression:** \[(A \bowtie B) \bowtie C\]

**Physical Expression:** \[(A_{Seq} \bowtie_{HJ} B_{Seq}) \bowtie_{NL} C_{Idx}\]

```sql
SELECT * FROM A
JOIN B ON A.id = B.id
JOIN C ON C.id = A.id;
```
A **group** is a set of logically equivalent logical and physical expressions that produce the same output.

→ All logical forms of an expression.
→ All physical expressions that can be derived from selecting the allowable physical operators for the corresponding logical forms.
A group is a set of logically equivalent logical and physical expressions that produce the same output.
→ All logical forms of an expression.
→ All physical expressions that can be derived from selecting the allowable physical operators for the corresponding logical forms.

Output: [ABC]

Logical Exps
1. (A⨝B)⨝C
2. (B⨝C)⨝A
3. (A⨝C)⨝B
4. A⨝(B⨝C)

Physical Exps
1. (A_{Seq}⨝_{NL}B_{Seq})⨝_{NL}C_{Seq}
2. (B_{Seq}⨝_{NL}C_{Seq})⨝_{NL}A_{Seq}
3. (A_{Seq}⨝_{NL}C_{Seq})⨝_{NL}B_{Seq}
4. A_{Seq}⨝_{NL}(C_{Seq}⨝_{NL}B_{Seq})
CASCADES: MULTI-EXPRESSION

Instead of explicitly instantiating all possible expressions in a group, the optimizer implicitly represents redundant expressions in a group as a multi-expression.

→ This reduces the number of transformations, storage overhead, and repeated cost estimations.

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical Multi-Exps</th>
<th>Physical Multi-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td><img src="image" alt="Logical Multi-Exps" /></td>
<td><img src="image" alt="Physical Multi-Exps" /></td>
</tr>
<tr>
<td>Properties: None</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CASCADES: RULES

A **rule** is a transformation of an expression to a logically equivalent expression.

→ **Transformation Rule**: Logical to Logical

→ **Implementation Rule**: Logical to Physical

Each rule is represented as a pair of attributes:

→ **Pattern**: Defines the structure of the logical expression that can be applied to the rule.

→ **Substitute**: Defines the structure of the result after applying the rule.
**CASCADES: RULES**

### Pattern

```
EQJOIN
   /   \
EQJOIN /    \ GROUP 3
   /      \
GROUP 1 /        \
   /          \
GROUP 2
```

- **Group**
- **Logical Expr**
- **Physical Expr**

### Transformation Rule

**Rotate Left-to-Right**

### Matching Plan

```
EQJOIN \[AB\] \[BC\]
   \[AB\] \[BC\]
   \[AB\] \[BC\]
```

### Implementation Rule

**EQJOIN**\[BC\]**\rightarrow****SORTMERGE**

```
A \[BC\] \[AB\] \[BC\] \[AB\] \[BC\]
   \[AB\] \[BC\] \[AB\] \[BC\] \[AB\] \[BC\]
```

- **EQJOIN**\[BC\]
- **SORTMERGE**

```
GET(A) \[BC\] \[AB\] \[BC\] \[AB\] \[BC\]
   \[AB\] \[BC\] \[AB\] \[BC\] \[AB\] \[BC\]
```

- **GET(A)**
- **GET(B)**
- **GET(C)**
CASCADeS: MEMO TABLE

Stores all previously explored alternatives in a compact graph structure / hash table.

Equivalent operator trees and their corresponding plans are stored together in groups.

Provides an overview of the optimizer's search progress that is used in multiple ways:
→ Transformation Result Memorization
→ Duplicate Group Detection
→ Property + Cost Management.
**PRINCIPLE OF OPTIMALITY**

Every sub-plan of an optimal plan is itself optimal.

This allows the optimizer to restrict the search space to a smaller set of expressions.

→ The optimizer never has to consider a plan containing sub-plan $P_1$ that has a greater cost than equivalent plan $P_2$ with the same physical properties.
**CASCADES: MEMO TABLE**

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td></td>
</tr>
<tr>
<td>[B]</td>
<td></td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Output: [ABC]  
Logical M-Exps:  
1. [AB]⨝[C]  
Physical M-Exps:  

Properties: None
# CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td></td>
</tr>
<tr>
<td>[B]</td>
<td></td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Output: [ABC]

Properties: None

Logical M-Exps
1. [AB]⨝ [C]

Physical M-Exps

Output: [AB]

Properties: None

Logical M-Exps
1. [A]⨝ [B]

Physical M-Exps

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td></td>
</tr>
<tr>
<td>[B]</td>
<td></td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Cost

Best Expr

[ABC]

[AB]

[A]

[B]

[C]
CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td></td>
</tr>
<tr>
<td>[B]</td>
<td></td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Output:

- [ABC]
  - Logical M-Exps
    1. [AB]⨝ [C]
  - Physical M-Exps

Output:

- [AB]
  - Logical M-Exps
    1. [A]⨝ [B]
  - Physical M-Exps

Output:

- [A]
  - Logical M-Exps
    1. GET(A)
  - Physical M-Exps
    1. SeqScan(A)
    2. IdxScan(A)
CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>SeqScan(A) 10</td>
</tr>
<tr>
<td>[B]</td>
<td></td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Cost: 10

Output: [A]
Properties: None

Logical M-Exps
1. GET(A)

Physical M-Exps
1. SeqScan(A)
2. IdxScan(A)

Output: [AB]
Properties: None

Logical M-Exps
1. [AB]⨝[C]

Physical M-Exps

Output: [ABC]

Logical M-Exps
1. [AB]⨝[C]

Physical M-Exps

Cost: 10
CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>SeqScan(A)</td>
</tr>
<tr>
<td>[B]</td>
<td></td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Cost: 10
CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>SeqScan(A) 10</td>
</tr>
<tr>
<td>[B]</td>
<td>SeqScan(B) 20</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Output: [ABC]
Properties: None
Logical M-Exps
1. [AB]⨝[C]

Physical M-Exps

Cost: 10

Output: [AB]
Properties: None
Logical M-Exps
1. [A]⨝[B]

Physical M-Exps

Cost: 20

Output: [A]
Properties: None
Logical M-Exps
1. GET(A)

Physical M-Exps

Cost: 10

Output: [B]
Properties: None
Logical M-Exps
1. GET(B)

Physical M-Exps

SeqScan(A) 10
SeqScan(B) 20
## CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td>1. [AB]⨝[C]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>SeqScan(A)</td>
<td>1. [A]⨝[B]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. [B]⨝[A]</td>
</tr>
<tr>
<td>[B]</td>
<td>SeqScan(B)</td>
<td>1. SeqScan(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. IdxScan(A)</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Properties:
- None

### Output:
- [AB]

### Logical M-Exps
- 1. [A]⨝[B]
- 2. [B]⨝[A]

### Physical M-Exps
- 1. SeqScan(A)
- 2. IdxScan(A)

### Cost: 10

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A]</td>
<td>1. GET(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Properties:
- None

### Output:
- [B]

### Logical M-Exps
- 1. GET(B)

### Physical M-Exps
- 1. SeqScan(B)
- 2. IdxScan(B)

### Cost: 20
CASCADEN: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>10</td>
</tr>
<tr>
<td>[B]</td>
<td>20</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Output: [ABC]
Logical M-Exps:
1. [AB]⨝[C]
Physical M-Exps:

Properties: None

Output: [AB]
Logical M-Exps:
1. [A]⨝[B]
2. [B]⨝[A]
Physical M-Exps:

Properties: None

Cost: 10

Output: [A]
Logical M-Exps:
1. GET(A)
Physical M-Exps:
1. SeqScan(A)
2. IdxScan(A)

Properties: None

Cost: 20

Output: [B]
Logical M-Exps:
1. GET(B)
Physical M-Exps:
1. SeqScan(B)
2. IdxScan(B)

Properties: None

Cost: 10

Cost: 20
CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>SeqScan(A) 10</td>
</tr>
<tr>
<td>[B]</td>
<td>SeqScan(B) 20</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

**Output:** [ABC]

**Properties:** None

**Logical M-Exps**
1. [AB] ⨝ [C]

**Physical M-Exps**

**Output:** [AB]

**Properties:** None

**Logical M-Exps**
1. [A] ⨝ [B]
2. [B] ⨝ [A]

**Physical M-Exps**
1. [A] ⨝_{NL} [B]
2. [A] ⨝_{HJ} [B]
3. [B] ⨝_{NL} [A]
4. [B] ⨝_{HJ} [A]

**Cost: 10**

**Output:** [A]

**Properties:** None

**Logical M-Exps**
1. GET(A)

**Physical M-Exps**
1. SeqScan(A)
2. IdxScan(A)

**Cost: 20**

**Output:** [B]

**Properties:** None

**Logical M-Exps**
1. GET(B)

**Physical M-Exps**
1. SeqScan(B)
2. IdxScan(B)
CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td>80</td>
</tr>
<tr>
<td>[A]</td>
<td>10</td>
</tr>
<tr>
<td>[B]</td>
<td>20</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

Cost: 50+(10+20)

Cost: 10

Cost: 20
**CASCADES: MEMO TABLE**

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td>80</td>
</tr>
<tr>
<td>[A]</td>
<td>10</td>
</tr>
<tr>
<td>[B]</td>
<td>20</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

**Output:** [ABC]  
**Logical M-Exps:**  
1. [A] \( \bowtie \) [B]  
2. [B] \( \bowtie \) [A]  
**Physical M-Exps:**  
1. [AB] \( \bowtie \) [C]  

**Cost:** 50 + (10 + 20)

**Output:** [AB]  
**Logical M-Exps:**  
1. [A] \( \bowtie \) [B]  
2. [B] \( \bowtie \) [A]  
**Physical M-Exps:**  
1. [A] \( \bowtie \) [B]  
2. [A] \( \bowtie \) [B]  
3. [B] \( \bowtie \) [A]  
4. [B] \( \bowtie \) [A]  

**Cost:** 10

**Output:** [A]  
**Logical M-Exps:**  
1. GET(A)  
**Physical M-Exps:**  
1. SeqScan(A)  
2. IdxScan(A)  

**Cost:** 20

**Output:** [B]  
**Logical M-Exps:**  
1. GET(B)  
**Physical M-Exps:**  
1. SeqScan(B)  
2. IdxScan(B)
### CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td>80</td>
</tr>
<tr>
<td>[A]</td>
<td>10</td>
</tr>
<tr>
<td>[B]</td>
<td>20</td>
</tr>
<tr>
<td>[C]</td>
<td>5</td>
</tr>
</tbody>
</table>

**Logical M-Exps**

1. `[AB]join[C]`

**Physical M-Exps**

1. `SeqScan(A)`
   - Cost: 10
2. `IdxScan(C)`
   - Cost: 5

**Logical M-Exps**

1. `[A]join[B]`
2. `[B]join[A]`

**Physical M-Exps**

1. `SeqScan(A)`
2. `IdxScan(A)`

**Logical M-Exps**

1. `[A]join[B]`
2. `[B]join[A]`

**Physical M-Exps**

1. `SeqScan(A)`
2. `IdxScan(A)`

**Logical M-Exps**

1. `[A]join[B]`
2. `[B]join[A]`

**Physical M-Exps**

1. `SeqScan(A)`
2. `IdxScan(A)`

**Logical M-Exps**

1. `GET(A)`

**Physical M-Exps**

1. `SeqScan(A)`
2. `IdxScan(A)`

**Logical M-Exps**

1. `GET(B)`

**Physical M-Exps**

1. `SeqScan(B)`
2. `IdxScan(B)`

**Logical M-Exps**

1. `GET(C)`

**Physical M-Exps**

1. `SeqScan(C)`
2. `IdxScan(C)`

**Cost: 10**

**Cost: 20**

**Cost: 50+(10+20)**

**Cost: 5**
### CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td></td>
</tr>
<tr>
<td>[AB]</td>
<td>80</td>
</tr>
<tr>
<td>[A]</td>
<td>10</td>
</tr>
<tr>
<td>[B]</td>
<td>20</td>
</tr>
<tr>
<td>[C]</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td>1. [AB]⨝ [C]</td>
<td>1. [AB]⨝ NL C</td>
</tr>
<tr>
<td></td>
<td>2. [BC]⨝ [A]</td>
<td>2. [BC]⨝ NL A</td>
</tr>
<tr>
<td></td>
<td>3. [AC]⨝ [B]</td>
<td>3. [AC]⨝ NL B</td>
</tr>
<tr>
<td></td>
<td>4. [B]⨝ [AC]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties:</th>
<th>None</th>
</tr>
</thead>
</table>

**Cost: 50+(10+20)**

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. [B]⨝ [A]</td>
<td>2. [B]⨝ NL [A]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. [B]⨝ NL [A]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. [B]⨝ NL [A]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties:</th>
<th>None</th>
</tr>
</thead>
</table>

**Cost: 5**

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[C]</td>
<td>1. GET(C)</td>
<td>1. SeqScan(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. IdxScan(C)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties:</th>
<th>None</th>
</tr>
</thead>
</table>

**Cost: 10**

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A]</td>
<td>1. GET(A)</td>
<td>1. SeqScan(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. IdxScan(A)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties:</th>
<th>None</th>
</tr>
</thead>
</table>

**Cost: 20**

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[B]</td>
<td>1. GET(B)</td>
<td>1. SeqScan(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. IdxScan(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties:</th>
<th>None</th>
</tr>
</thead>
</table>

**Cost: 5**
### CASCADES: MEMO TABLE

<table>
<thead>
<tr>
<th>Best Expr</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td>125</td>
</tr>
<tr>
<td>[AB]</td>
<td>80</td>
</tr>
<tr>
<td>[A]</td>
<td>10</td>
</tr>
<tr>
<td>[B]</td>
<td>20</td>
</tr>
<tr>
<td>[C]</td>
<td>5</td>
</tr>
</tbody>
</table>

### Cost: 40+(80+5)

**Output:** [ABC]

**Logical M-Exps**
1. [AB]⨝[C]
2. [BC]⨝[A]
3. [AC]⨝[B]
4. [B]⨝[AC]

**Physical M-Exps**
1. [AB]⨝NL[C]
2. [BC]⨝NL[A]
3. [AC]⨝NL[B]

### Cost: 50+(10+20)

**Output:** [AB]

**Logical M-Exps**
1. [A]⨝[B]
2. [B]⨝[A]

**Physical M-Exps**
1. [A]⨝NL[B]
2. [A]⨝HJ[A]
3. [B]⨝NL[A]
4. [B]⨝HJ[A]

### Cost: 5

**Output:** [C]

**Logical M-Exps**
1. GET(C)

**Physical M-Exps**
1. SeqScan(C)
2. IdxScan(C)

### Cost: 10

**Output:** [A]

**Logical M-Exps**
1. GET(A)

**Physical M-Exps**
1. SeqScan(A)
2. IdxScan(A)

### Cost: 20

**Output:** [B]

**Logical M-Exps**
1. GET(B)

**Physical M-Exps**
1. SeqScan(B)
2. IdxScan(B)
CASCADES IMPLEMENTATIONS

Standalone:
→ Wisconsin OPT++ (1990s)
→ Portland State Columbia (1990s)
→ Greenplum Orca (2010s)
→ Apache Calcite (2010s)

Integrated:
→ Microsoft SQL Server (1990s)
→ Tandem NonStop SQL (1990s)
→ CockroachDB (2010s)
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 length of time.

Examples: 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 (GEQO).

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
POSTGRES GENETIC OPTIMIZER

1st Generation

Cost: 300

Cost: 200

Cost: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

Cost: 300

Cost: 200

Cost: 100

Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

- Cost: 300 (Red X)
- Cost: 200
- Cost: 100

Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

Best: 100

Cost: 300

Cost: 200

Cost: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

- Cost: 300

2nd Generation

- Cost: 100

Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

- Cost: 300
- Cost: 200
- Cost: 100

2nd Generation

- Cost: 80
- Cost: 200
- Cost: 110

Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

- Cost: 300
- Cost: 200
- Cost: 100

2nd Generation

- Cost: 80
- Cost: 200
- Cost: 110

Best: 80
POSTGRES GENETIC OPTIMIZER

1st Generation

- Cost: 300
- Cost: 200
- Cost: 100

2nd Generation

- Cost: 80
- Cost: 200
- Cost: 110

Best: 80
POSTGRES GENETIC OPTIMIZER

1st Generation

1. Cost: 300
2. Cost: 200
3. Cost: 100

2nd Generation

1. Cost: 80
2. Cost: 200
3. Cost: 110

3rd Generation

1. Cost: 90
2. Cost: 160
3. Cost: 120

Best: 80
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 plan.
→ Must do extra work to ensure that query plans are deterministic.
→ Still must implement correctness rules.
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 plan.
→ Must do extra work to ensure that query plans are deterministic.
→ Still must implement correctness rules.

Still Not Efficient
- The work that we're performing per "relation" is not a constant! We consider many possibilities per "relation," throw away the ones that are clearly inferior, and keep the ones that look most promising.
- Still doesn't scale to large join problems. We're avoiding recomputation, but still searching a very large problem space.
- When the number of tables exceeds gego_threshold (by default, 12), we switch to GEQO, the "genetic query optimizer." It essentially tries a bunch of join orders at random and picks the best one. If you're lucky, it won't be too bad.
PARTING THOUGHTS

Query optimization is hard.
This difficulty is why NoSQL systems didn’t implement optimizers (at first).

Playlist of CMU-DB Query Optimizer talks:
→ https://cmudb.io/youtube-optimizers
Query optimization is hard. This is why NoSQL systems didn’t implement optimizers (at first).

Playlist of CMU - DB Query Optimizer talks:
→ https://cmudb.io/youtube

The Cascades Framework for Query Optimization at Microsoft

Nico Bruno
Cesar Galindo-Legaria
NEXT CLASS

German-style Unnesting Sub-Queries
German-style Dynamic Programming