LAST CLASS

Choice #1: Heuristics
→ INGRES, Oracle (until mid 1990s)

Choice #2: Heuristics + Cost-based Join Search
→ System R, early IBM DB2, most open-source DBMSs

Choice #3: Stratified Search
→ IBM’s STARBURST (late 1980s), now IBM DB2 + Oracle

Choice #4: Unified Search
→ Volcano/Cascades in 1990s, now MSSQL + Greenplum

Choice #5: Randomized Search
→ Academics in the 1980s, current Postgres
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.
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.
**TOP-DOWN VS. BOTTOM-UP**

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

**Bottom-up Optimization**
→ Start with nothing and then build up the plan to get to the outcome that you want.
→ **Examples**: System R, Starburst
TODAY'S AGENDA

Unified Search
Randomized Search
Real-World Implementations
Unnesting Subqueries
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: \((A_{\text{Seq}} \bowtie_{\text{HJ}} B_{\text{Seq}}) \bowtie_{\text{NL}} C_{\text{Idx}}\)
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.

<table>
<thead>
<tr>
<th>Output:</th>
<th>Logical Exps</th>
<th>Physical Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ABC]</td>
<td>1. ((A \bowtie B) \bowtie C)</td>
<td>1. (A_{\text{Seq}} \bowtie_{\text{NL}} B_{\text{Seq}} \bowtie_{\text{NL}} C_{\text{Seq}})</td>
</tr>
<tr>
<td></td>
<td>2. ((B \bowtie C) \bowtie A)</td>
<td>2. ((B \bowtie_{\text{Seq}} \bowtie_{\text{NL}} C_{\text{Seq}}) \bowtie_{\text{NL}} A_{\text{Seq}})</td>
</tr>
<tr>
<td></td>
<td>3. ((A \bowtie C) \bowtie B)</td>
<td>3. ((A \bowtie_{\text{Seq}} \bowtie_{\text{NL}} C_{\text{Seq}}) \bowtie_{\text{NL}} B_{\text{Seq}})</td>
</tr>
<tr>
<td></td>
<td>4. (A \bowtie (B \bowtie C))</td>
<td>4. (A_{\text{Seq}} \bowtie_{\text{NL}} (C \bowtie_{\text{Seq}} \bowtie_{\text{NL}} B_{\text{Seq}}))</td>
</tr>
</tbody>
</table>

Properties: None
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]

**Properties:** None

<table>
<thead>
<tr>
<th>Logical Exps</th>
<th>Physical Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (A⨝B)⨝C</td>
<td>1. (A_{Seq}⨝<em>{NL}B</em>{Seq})⨝<em>{NL}C</em>{Seq}</td>
</tr>
<tr>
<td>2. (B⨝C)⨝A</td>
<td>2. (B_{Seq}⨝<em>{NL}C</em>{Seq})⨝<em>{NL}A</em>{Seq}</td>
</tr>
<tr>
<td>3. (A⨝C)⨝B</td>
<td>3. (A_{Seq}⨝<em>{NL}C</em>{Seq})⨝<em>{NL}B</em>{Seq}</td>
</tr>
<tr>
<td>4. A⨝(B⨝C)</td>
<td>4. A_{Seq}⨝<em>{NL}(C</em>{Seq}⨝<em>{NL}B</em>{Seq})</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
</tr>
</tbody>
</table>
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: [ABC]</th>
<th>Logical Multi-Exps</th>
<th>Physical Multi-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties: None</td>
<td>1. [AB]⨝[C]</td>
<td>1. [AB]⨝_{SM}[C]</td>
</tr>
<tr>
<td></td>
<td>2. [BC]⨝[A]</td>
<td>2. [AB]⨝_{HJ}[C]</td>
</tr>
<tr>
<td></td>
<td>3. [AC]⨝[B]</td>
<td>3. [AB]⨝_{NL}[C]</td>
</tr>
<tr>
<td></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

Transformation Rule
Rotate Left-to-Right

Matching Plan

Implementation Rule
EQJOIN→SORTMERGE

Group

Logical Expr

Physical Expr

CMU-DB
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]
Properties: None

Logical M-Exps
1. [AB]⨝[C]

Physical M-Exps

Output: [AB]
Properties: None

Logical M-Exps
1. [A]⨝[B]

Physical M-Exps

## 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] \times [C]  
**Physical M-Exps:**

**Output:** [AB]  
**Logical M-Exps:**  
1. [A] \times [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>

**Logical M-Exps**

1. \([\text{AB}] \bowtie [\text{C}]\)

**Physical M-Exps**

Output: [AB]

Properties: None

Cost: 10

**Logical M-Exps**

1. \([\text{A}] \bowtie [\text{B}]\)

**Physical M-Exps**

Output: [A]

Properties: None

**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></td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

**Cost: 10**

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

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

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

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

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

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>[AB]</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>SeqScan(A)</td>
</tr>
<tr>
<td>[B]</td>
<td>SeqScan(B)</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

**Logical M-Exps**
1. [AB]⨝[C]

**Physical M-Exps**

**Cost: 10**

<table>
<thead>
<tr>
<th>Output: [A]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. GET(A)</td>
<td></td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

**Cost: 20**

<table>
<thead>
<tr>
<th>Output: [B]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. SeqScan(B)</td>
<td></td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

**Cost: 10**

<table>
<thead>
<tr>
<th>Output: [AB]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. [AB]⨝[C]</td>
<td></td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

**Cost: 20**

<table>
<thead>
<tr>
<th>Output: [ABC]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
## 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>

### Logical M-Exps

Output: [ABC]

- Properties: None

Logical M-Exps:
1. [AB] ⨝ [C]

### Physical M-Exps

Output: [ABC]

- Properties: None

Physical M-Exps:

Cost: 10

Output: [AB]

- Properties: None

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

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)
2. IdxScan(A)

Physical M-Exps:

Cost: 20

Output: [B]

- Properties: None

Logical M-Exps:
1. SeqScan(B)
2. IdxScan(B)

Physical M-Exps:
# 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>10</td>
</tr>
<tr>
<td>[B]</td>
<td>20</td>
</tr>
<tr>
<td>[C]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output: [ABC]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. [AB]⨝[C]</td>
<td></td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output: [AB]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. [A]⨝[B]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. [B]⨝[A]</td>
<td></td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output: [A]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. GET(A)</td>
<td>SeqScan(A)</td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output: [B]</th>
<th>Logical M-Exps</th>
<th>Physical M-Exps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. GET(B)</td>
<td>SeqScan(B)</td>
</tr>
<tr>
<td>Properties:</td>
<td>None</td>
<td>20</td>
</tr>
</tbody>
</table>

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>10</td>
</tr>
<tr>
<td>[A]</td>
<td>SeqScan(A)</td>
</tr>
<tr>
<td>[B]</td>
<td>SeqScan(B)</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**:
  - SeqScan(A)

### Cost: 20

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

**Output:** [ABC]

**Logical M-Exps**
1. [AB] ⨝ [C]

**Physical M-Exps**

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

**Output:** [AB]

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

**Physical M-Exps**

**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)

**Cost: 20**

**Output:** [C]

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

**Physical M-Exps**

**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>5</td>
</tr>
</tbody>
</table>

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: 5

Output: [C]
Logical M-Exps
1. GET(C)
Physical M-Exps
1. SeqScan(C)
2. IdxScan(C)
Properties: None

Cost: 50+(10+20)

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

Cost: 5

Output: [C]
Logical M-Exps
1. GET(C)
Physical M-Exps
1. SeqScan(C)
2. IdxScan(C)
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: 5

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

Cost: 5

Output: [C]
Logical M-Exps
1. GET(C)
Physical M-Exps
1. SeqScan(C)
2. IdxScan(C)
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: 5

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

Cost: 50+(10+20)

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

Cost: 5

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

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

**Output:** [ABC]  

Properties: None

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

### Cost: 5

**Output:** [C]  

Properties: None

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

### Cost: 10

**Output:** [A]  

Properties: None

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

### Cost: 20

**Output:** [B]  

Properties: None

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

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>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[B]  
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)
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

- **Best**: 100

  - **Cost**: 300
    - NL
      - R
      - S
      - T
  - **Cost**: 200
    - HJ
      - S
      - T
      - R
  - **Cost**: 100
    - HJ
      - S
      - R
      - T
POSTGRES GENETIC OPTIMIZER

1st Generation

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

Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

Cost: 300

Cost: 200

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
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 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 `geqo_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.
DYNAMIC PROGRAMMING OPTIMIZER

Model the query as a hypergraph and then incrementally expand to enumerate new plans.

Algorithm Overview:
→ Iterate connected sub-graphs and incrementally add new edges to other nodes to complete query plan.
→ Use rules to determine which nodes the traversal is allowed to visit and expand.
REAL-WORLD IMPLEMENTATIONS

Microsoft SQL Server
Apache Calcite
Greenplum Orca
CockroachDB
SingleStore
Snowflake

Cascades
First Cascades implementation started in 1995.
→ Derivatives are used in many MSFT database products.
→ All transformations are written in C++. No DSL.
→ Scalar / expression transformations are written in procedural code and not rules.

DBMS applies transformations in multiple stages with increasing scope and complexity.
→ The goal is to leverage domain knowledge to apply transformations that you always want to do first to reduce the search space.
Sub-Query Removal
Outer Joins to Inner Joins
Predicate Pushdown
Empty Result Pruning

Simplification / Normalization

Tree-to-Tree Transformations

Cost-based Search Initialization

Pre-Exploration

Exploration

Multi-Stage Cost-Based Search

Stage 1: Trivial Plan
Stage 2: Quick Plan (Parallel)
Stage 3: Full Plan (Parallel)

Engine-Specific Transformations

MICROSOFT SQL SERVER

Stage 1: Trivial Plan
Stage 2: Quick Plan (Parallel)
Stage 3: Full Plan (Parallel)

Simplification / Normalization

Tree-to-Tree Transformations

Cost-based Search Initialization

Pre-Exploration

Exploration

Multi-Stage Cost-Based Search

Engine-Specific Transformations

Source: Nico Bruno + Cesar Galindo-Legaria
**MICROSOFT SQL SERVER**

**Optimization #1:** Timeouts are based on the number of transformations not wallclock time.
→ Ensures that overloaded systems do not generate different plans than under normal operations.

**Optimization #2:** Pre-populate the Memo Table with potentially useful join orderings.
→ Heuristics that consider relationships between tables.
→ Syntactic appearance in query.
APACHE CALCITE

Standalone extensible query optimization framework for data processing systems.
→ Support for pluggable query languages, cost models, and rules.
→ Does not distinguish between logical and physical operators. Physical properties are provided as annotations.

Originally part of LucidDB.
GREENPLUM ORCA

Standalone Cascades implementation in C++.  
→ Originally written for Greenplum.  
→ Extended to support HAWQ.

A DBMS integrates Orca by implementing API to send catalog + stats + logical plans and then retrieve physical plans.

Supports multi-threaded search.
Issue #1: Remote Debugging
→ Automatically dump the state of the optimizer (with inputs) whenever an error occurs.
→ The dump is enough to put the optimizer back in the exact same state later for further debugging.

Issue #2: Optimizer Accuracy
→ Automatically check whether the ordering of the estimate cost of two plans matches their actual execution cost.
Custom Cascades implementation written in 2018. All transformation rules are written in a custom DSL (OptGen) and then codegen into Go-lang. → Can embed Go logic in rule to perform more complex analysis and modifications.

Also considers scalar expression (predicates) transformations together with relational operators.
Custom Cascades implementation in CockroachDB.

All transformation rules are written in a custom DSL (OptGen) and then codegen into Go language.

→ Can embed Go logic into transformation analysis and modifications.

Also considers scalar expression transformations together with relational operators.

Source: Rebecca Taft

DSL: OptGen

```go
// ConstructNot constructs an expression for the Not operator.
func (_f *Factory) ConstructNot(m input ScalarExpr) opt ScalarExpr {

    // [EliminateNot]
    
    _not, _ := input.(*memo.NotExpr)
    if _not != nil {
        input := _not.Input
        if _f.matchedRule == nil || _f.matchedRule(opt.EliminateNot) {
            _expr := input
            return _expr
        }
    }

    // ... other rules ...

e := _f.memo.MemoizeNot(input)
    return _f.onConstructScalar(e)
}```
SQL allows a nested **SELECT** subquery to exist (almost?) anywhere in another query.

→ Projection, **FROM, WHERE, LIMIT, HAVING**

→ Results of the inner subquery are passed to the outer query.

Such nesting enables more expressive queries without having to use separate queries to prepare intermediate results.
SUBQUERIES

SQL allows a nested **SELECT** subquery to exist (almost?) anywhere in another query.

→ Projection, **FROM, WHERE, LIMIT, HAVING**
→ Results of the inner subquery are passed to the outer query.

Such nesting enables more expressive queries without having to use separate queries to prepare intermediate results.

**Key Distinction: Uncorrelated vs. Correlated**
An uncorrelated subquery does not reference any attributes from the (calling) outer query.

The DBMS logically executes it once and reuses the result for all tuples in the outer query.
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.

```
SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);
```
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.

```
SELECT name, major
FROM students AS s1
WHERE score =
  (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);
```

<table>
<thead>
<tr>
<th>name</th>
<th>major</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>GZA</td>
<td>CompSci</td>
<td>90</td>
</tr>
<tr>
<td>RZA</td>
<td>CompSci</td>
<td>80</td>
</tr>
<tr>
<td>ODB</td>
<td>Streets</td>
<td>100</td>
</tr>
</tbody>
</table>

`s1.major='CompSci'`
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.

```
SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);
```

<table>
<thead>
<tr>
<th>name</th>
<th>major</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>GZA</td>
<td>CompSci</td>
<td>90</td>
</tr>
<tr>
<td>RZA</td>
<td>CompSci</td>
<td>80</td>
</tr>
<tr>
<td>ODB</td>
<td>Streets</td>
<td>100</td>
</tr>
</tbody>
</table>

s1.major='CompSci'  MAX(s2.score)=90
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.

```
SELECT name, major
FROM students AS s1
WHERE score =
(SELECT MAX(s2.score)
FROM students AS s2
WHERE s2.major = s1.major);
```
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.

```
SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);
```
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.

```
SELECT name, major
FROM students AS s1
WHERE score = 
  (SELECT MAX(s2.score)
   FROM students AS s2
   WHERE s2.major = s1.major);
```
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.
A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

The DBMS logically evaluates the subquery on each tuple in the outer query because the result can change per tuple.

```sql
SELECT name, major FROM students AS s1
WHERE score =
(SELECT MAX(s2.score) FROM students AS s2
WHERE s2.major = s1.major);
```
Almost every DBMS that supports uses heuristics that identify specific query plan patterns to decorrelate nested subqueries.

The goal is to move the subquery up a level so that the DBMS can execute it as a join.
Almost every DBMS that supports
query optimization uses
heuristics that identify
query plan patterns to
decorrelate nested subqueries.

The subquery in the
WHERE clause can
be moved up a level
in the query
as

\[
R \ A^\otimes E = R \otimes_{\text{true}} E, \quad \text{(1)}
\]

if no parameters in \( E \) resolved from \( R \)

\[
R \ A^\otimes (\sigma_p E) = R \otimes_p E, \quad \text{(2)}
\]

if no parameters in \( E \) resolved from \( R \)

\[
R \ A^\times (\sigma_p E) = \sigma_p (R \ A^\times E), \quad \text{(3)}
\]

\[
R \ A^\times (\pi_v E) = \pi_v \cup \text{columns}(R)(R \ A^\times E), \quad \text{(4)}
\]

\[
R \ A^\times (E_1 \cup E_2) = (R \ A^\times E_1) \cup (R \ A^\times E_2), \quad \text{(5)}
\]

\[
R \ A^\times (E_1 - E_2) = (R \ A^\times E_1) - (R \ A^\times E_2), \quad \text{(6)}
\]

\[
R \ A^\times (E_1 \times E_2) = (R \ A^\times E_1) \bowtie_{\text{R.key}} (R \ A^\times E_2), \quad \text{(7)}
\]

\[
R \ A^\times (G_{A,F} E) = G_{A \cup \text{columns}(R),F}(R \ A^\times E), \quad \text{(8)}
\]

\[
R \ A^\times (G_F^1 E) = G_{\text{columns}(R),F'}(R \ A^\times E), \quad \text{(9)}
\]
Almost every DBMS that supports

using heuristics to identify specific query plan patterns to decorrelate nested subqueries. The goal is to move the subquery up a level so that the DBMS can execute it as a join.

Orthogonal Optimization of Subqueries and Aggregation SIGMOD 2001

11. Subquery Flattening

When a subquery occurs as the FROM clause of a SELECT, the simplest behavior is to evaluate the subquery into a temporary table, then run the outer SELECT against the temporary table. Such a plan can be suboptimal since the temporary table will not have any indexes and the outer query which is likely to fail will be forced to do a full table scan on the temporary table.

To overcome this problem, SQLite attempts to flatten subqueries in the FROM clause of a SELECT. This involves inserting the entire clause of the subquery into the FROM clause of the outer query and rewriting expressions in the outer query that refer to the result set of the subquery. For example:

```
SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);
```

The query can be rewritten using query flattening as:

```
SELECT s1.name, s1.major
FROM students AS s1
JOIN (
    SELECT major,
    MAX(s2.score) AS max_score
    FROM students
    GROUP BY major)
AS s2
ON s1.major = s2.major
AND s1.score = s2.max_score
```

Query flattening is an important optimization when views are used in each use of a view is translated into a subquery.
HEURISTIC REWRITING

Advantages:
→ Transformed queries are more efficient.
→ Decision to decorrelate can be a cost-based decision.
→ Easy to control decorrelation by enabling/disabling rules.

Disadvantages:
→ Hard to write rules for all possible correlations scenarios.
→ Changing a small part of a query can make rules ineffective.
→ Maintaining transformation rules is a difficult.
→ Handling all edge cases is exceedingly difficult.

Source: Mayank Baranwal
General-purpose method to eliminate all dependent joins by manipulating the query plan until the RHS no longer depends on the LHS.

The optimizer then converts dependent joins to regular joins.

→ Some queries switch from a $O(n^2)$ nested-loop join to a $O(n)$ hash join.
FLATTENING CORRELATED QUERIES

SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);

Introduce a dependent join operator to execute RHS once for every tuple in LHS.

Source: Mark Raasveldt
FLATTENING CORRELATED QUERIES

Introduce a **dependent join** operator to execute RHS once for every tuple in LHS.

**Source:** Mark Raasveldt
Introduce a **dependent join** operator to execute RHS once for every tuple in LHS.

Source: Mark Raasveldt
New **dependent join** relational algebra operator that denotes a correlated subquery.

→ Evaluate RHS of the join for every tuple on the LHS.

→ The operator combine results from every execution and return them as its output.
FLATTENING CORRELATED QUERIES

SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);

Push dependent join down into the RHS of the plan.

Only need to execute RHS once for every unique combination of correlated columns.

→ Duplicate Elimination Scan

Source: Mark Raasveldt
Pull dependent join down into the RHS of the plan.

Only need to execute RHS once for every unique combination of correlated columns.

→ Duplicate Elimination Scan

**Source:** Mark Raasveldt
Keeping pushing dependent join as far down into the plan as is possible.

Source: Mark Raasveldt
Keeping pushing dependent join as far down into the plan as is possible.
SELECT name, major
FROM students AS s1
WHERE score =
(SELECT MAX(s2.score)
FROM students AS s2
WHERE s2.major = s1.major);

Keeping pushing dependent join as far down into the plan as is possible.

Source: Mark Raasveldt
SELECT name, major
FROM students AS s1
WHERE score =
(SELECT MAX(s2.score)
FROM students AS s2
WHERE s2.major = s1.major);

Convert the **dependent join** operator into a **cross product**.
FLATTENING CORRELATED QUERIES

Convert the **dependent join** operator into a **cross product**.

```
SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);
```

Source: Mark Raasveldt
Convert the **dependent join** operator into a **cross product**.

```
SELECT name, major
FROM students AS s1
WHERE score =
  (SELECT MAX(s2.score)
   FROM students AS s2
   WHERE s2.major = s1.major);
```
FLATTENING CORRELATED QUERIES

Convert the **dependent join** operator into a **cross product**.

Then convert the **cross product** into an **inner join**.

Source: Mark Raasveldt
SELECT name, major
FROM students AS s1
WHERE score =
  (SELECT MAX(s2.score)
   FROM students AS s2
   WHERE s2.major = s1.major);

Convert the dependent join operator into a cross product.

Then convert the cross product into an inner join.

Source: Mark Raasveldt
FLATTENING CORRELATED QUERIES

SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);

Remove duplicate elimination scan entirely.

Remove the filter above the new join.

Source: Mark Raasveldt
SELECT name, major
FROM students AS s1
WHERE score =
  (SELECT MAX(s2.score)
   FROM students AS s2
   WHERE s2.major = s1.major);

Remove duplicate elimination scan entirely.

Remove the filter above the new join.

Source: Mark Raasveldt
SELECT name, major
FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
     FROM students AS s2
     WHERE s2.major = s1.major);

Remove duplicate elimination scan entirely.

Remove the filter above the new join.

Source: Mark Raasveldt
SELECT name, major
FROM students AS s1
WHERE score =
  (SELECT MAX(s2.score)
   FROM students AS s2
   WHERE s2.major = s1.major);

Remove duplicate elimination scan entirely.

Remove the filter above the new join.

Source: Mark Raasveldt
SELECT name, major 
FROM students AS s1 
WHERE score = 
  (SELECT MAX(s2.score) 
    FROM students AS s2 
    WHERE s2.major = s1.major);

Remove duplicate elimination scan entirely.

Remove the filter above the new join.

Source: Mark Raasveldt
Only HyPer, Umbra, and DuckDB correctly unnest correlated sub-queries.

All the optimizer strategies we discussed assume that the optimizer has one shot at choosing a plan. But what happens if the DBMS discovers that the cost estimates don't match reality when it starts processing data?
PARTING THOUGHTS

Only HyPer, Umbra, and DuckDB correctly unnest correlated sub-queries.

All the optimizer strategies we discussed assume that the optimizer has one shot at choosing a plan. But what happens if the DBMS discovers that the cost estimates don't match reality when it starts processing data?
Adaptive Query Optimization