Carnegie Mellon University

# OPTIMIZE!

Database Query Optimization

## Cascades Query Optimizer

# ERRATA

Volcano Clarification
→ Optimization Phases
→ Enforcers

**Send Corrections: db-mistakes@cs.cmu.edu**

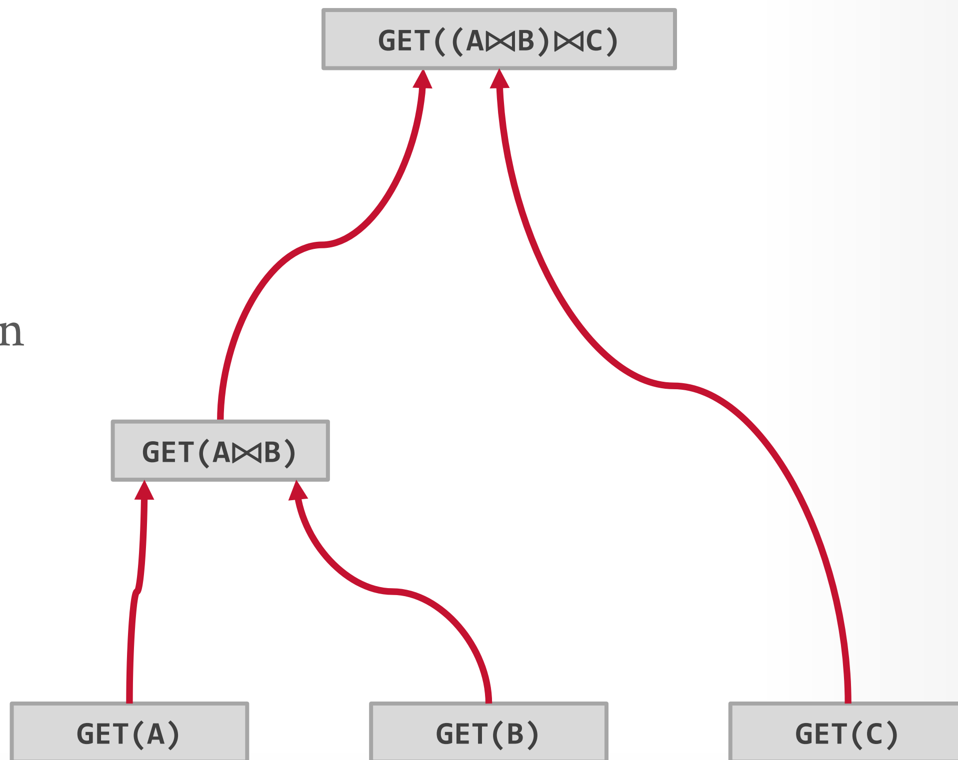
Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES

Start with a logical plan of what result the query needs to produce.

**Generation Phase:**
→ Apply transformation rules to generate all possible logical expression alternatives.

**Cost Analysis Phase:**
→ Apply implementation rules to generate physical operators.



Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES

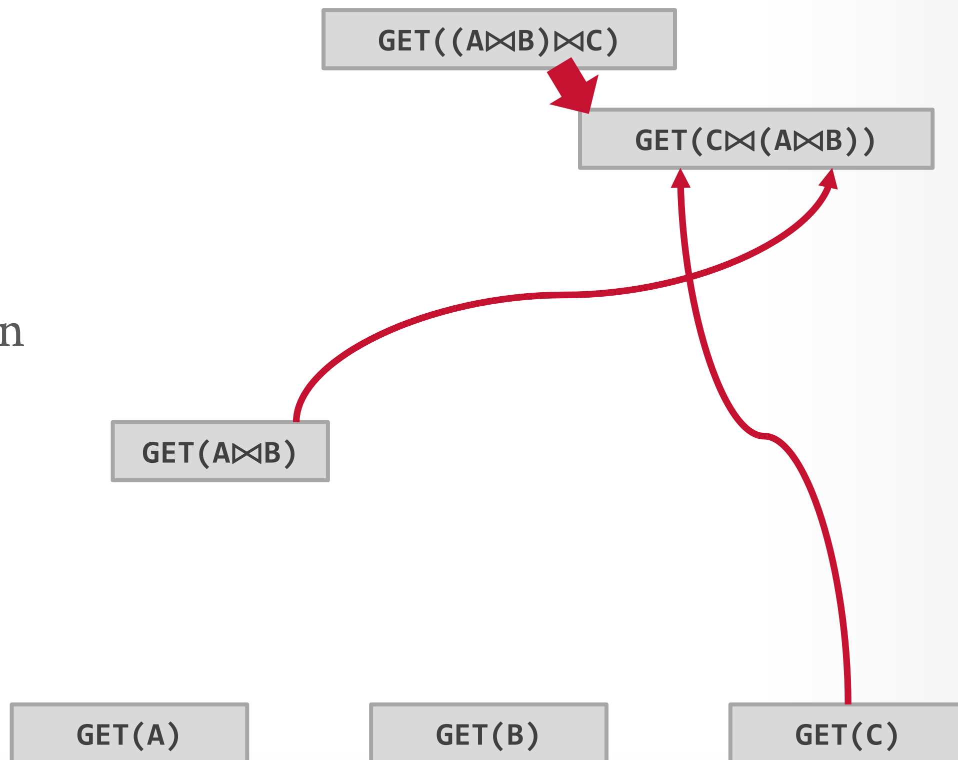
Start with a logical plan of what result the query needs to produce.

## Generation Phase:

→ Apply transformation rules to generate all possible logical expression alternatives.

## Cost Analysis Phase:

→ Apply implementation rules to generate physical operators.



Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES



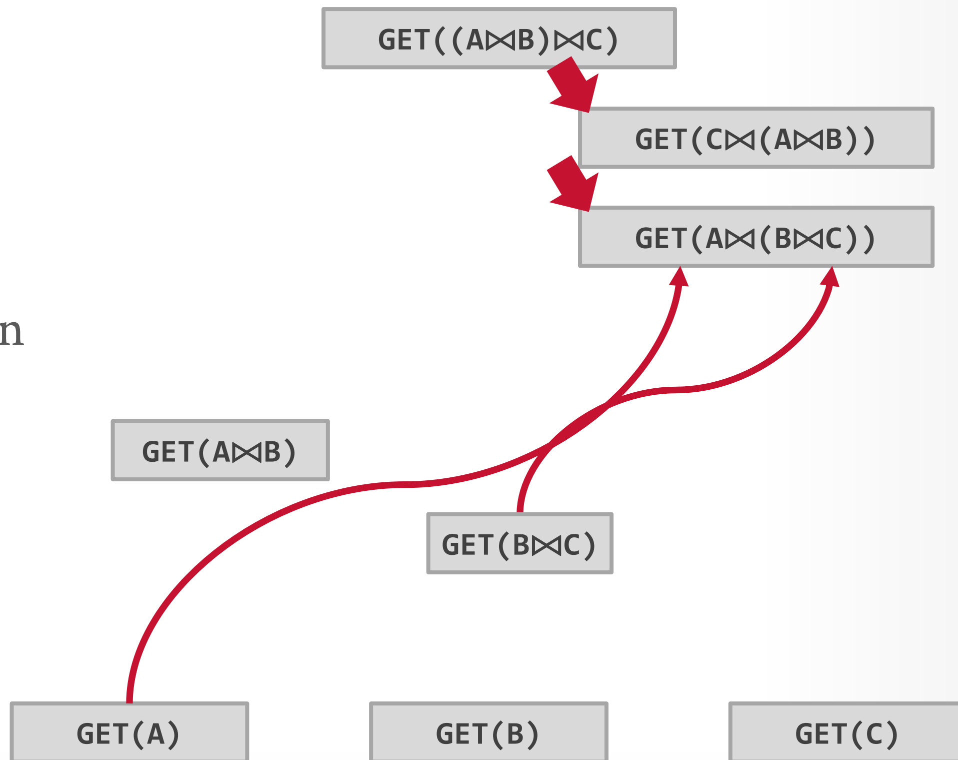
Start with a logical plan of what result the query needs to produce.

**Generation Phase:**

→ Apply transformation rules to generate all possible logical expression alternatives.

**Cost Analysis Phase:**

→ Apply implementation rules to generate physical operators.



Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES

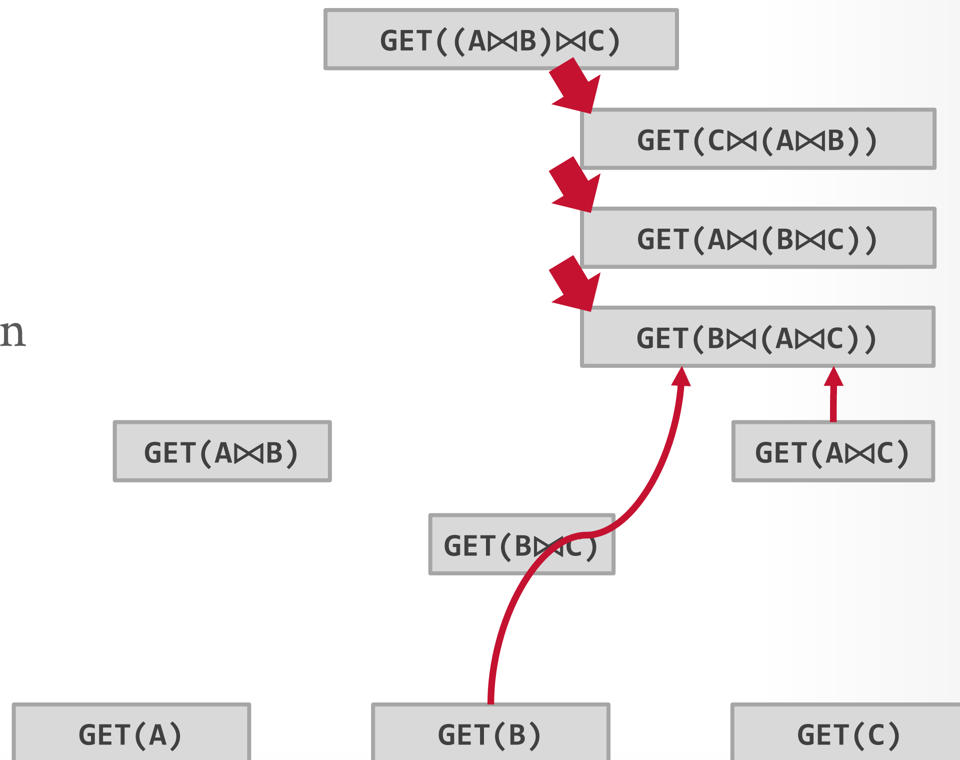
Start with a logical plan of what result the query needs to produce.

## Generation Phase:

→ Apply transformation rules to generate all possible logical expression alternatives.

## Cost Analysis Phase:

→ Apply implementation rules to generate physical operators.



*Logical Op*
*Physical Op*
*Enforcer*

# VOLCANO: PHASES

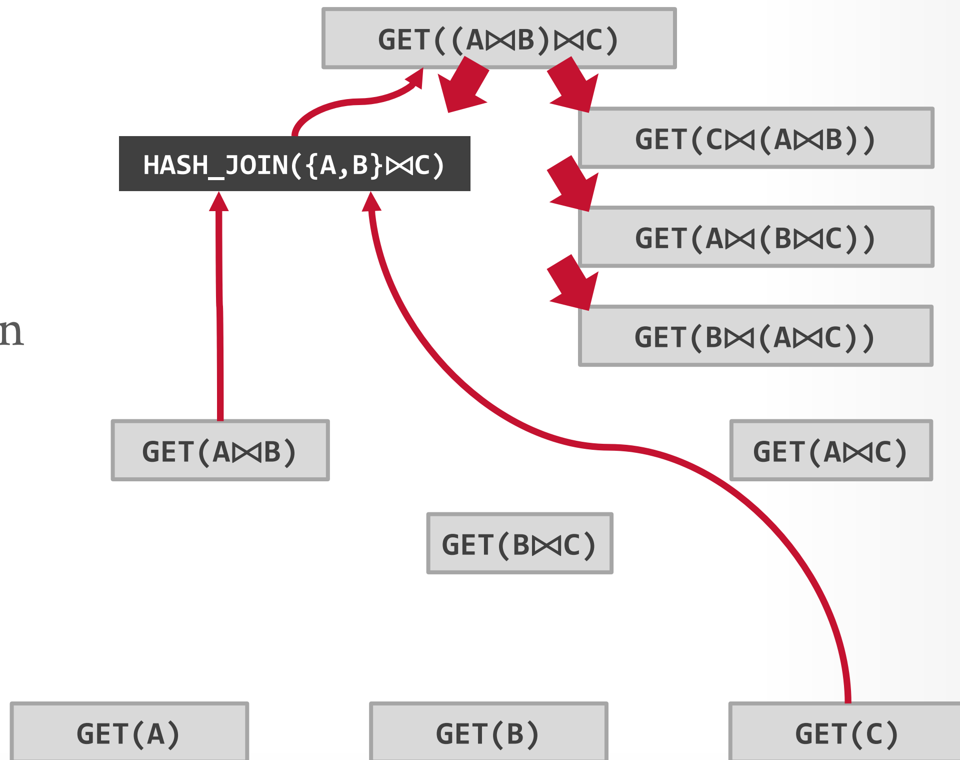
Start with a logical plan of what result the query needs to produce.

## Generation Phase:

→ Apply transformation rules to generate all possible logical expression alternatives.

## Cost Analysis Phase:

→ Apply implementation rules to generate physical operators.



Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES

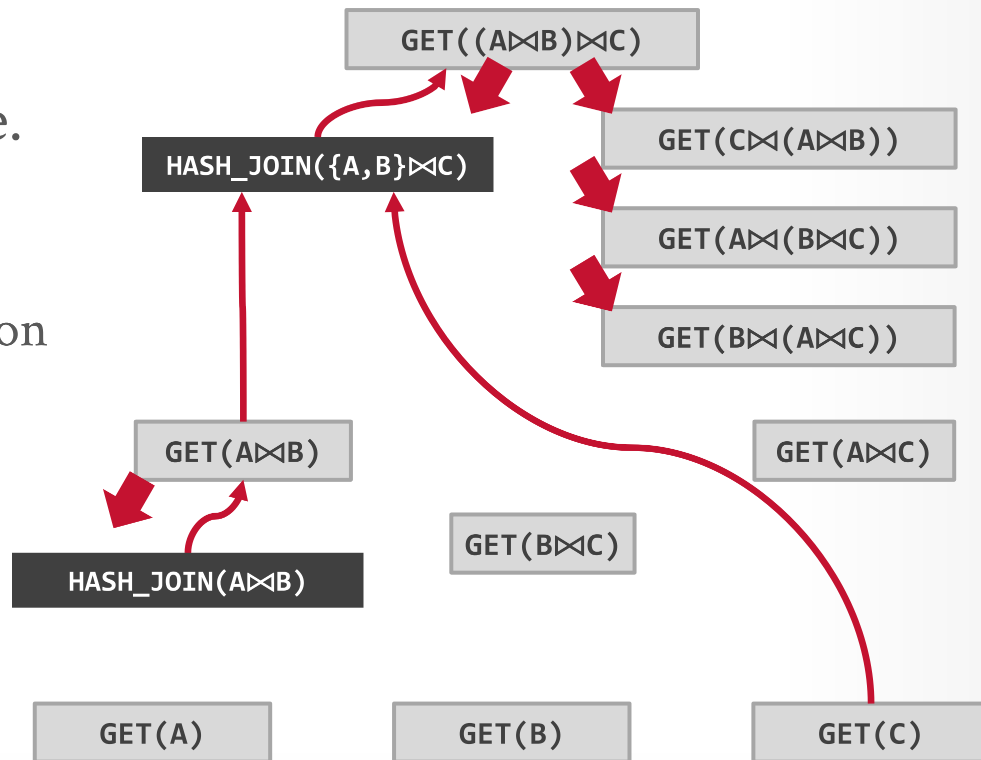
Start with a logical plan of what result the query needs to produce.

## Generation Phase:

→ Apply transformation rules to generate all possible logical expression alternatives.

## Cost Analysis Phase:

→ Apply implementation rules to generate physical operators.



Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES

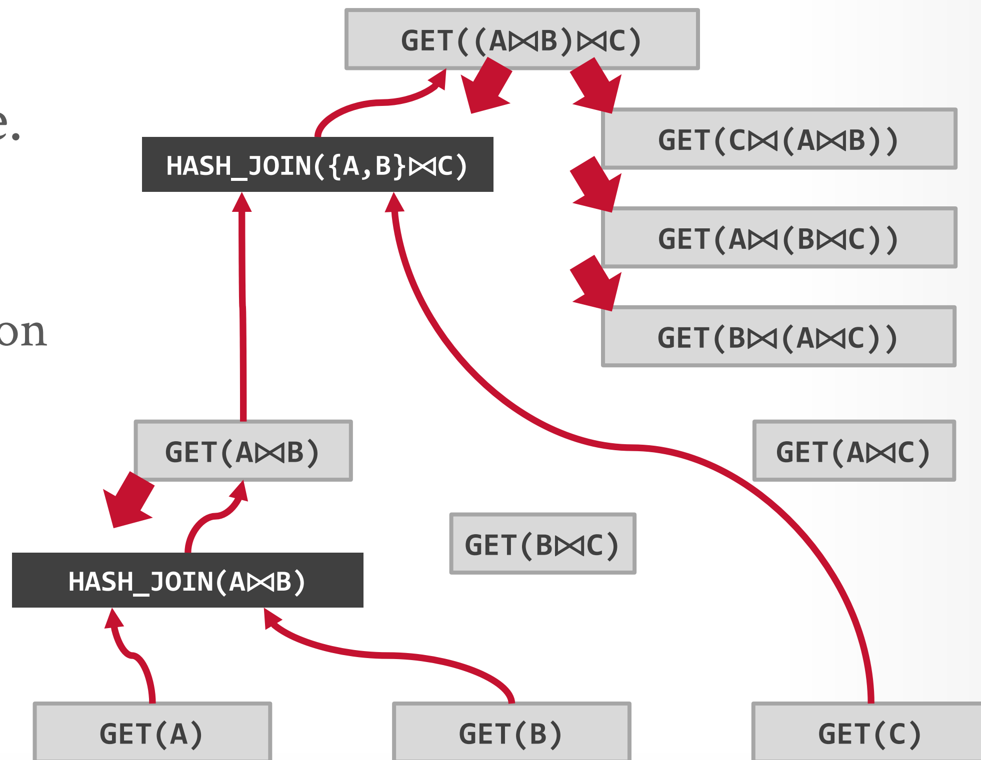
Start with a logical plan of what result the query needs to produce.

## Generation Phase:

→ Apply transformation rules to generate all possible logical expression alternatives.

## Cost Analysis Phase:

→ Apply implementation rules to generate physical operators.



Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES



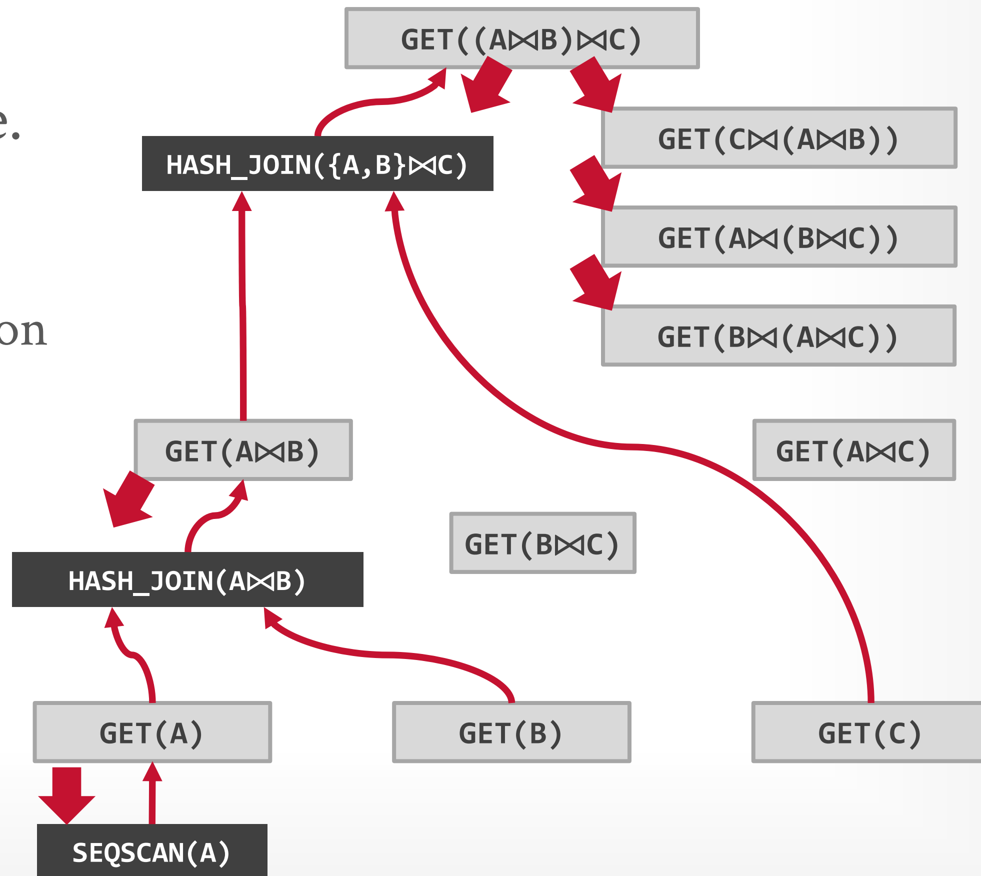
Start with a logical plan of what result the query needs to produce.

**Generation Phase:**

→ Apply transformation rules to generate all possible logical expression alternatives.

**Cost Analysis Phase:**

→ Apply implementation rules to generate physical operators.



*Logical Op*
*Physical Op*
*Enforcer*

# VOLCANO: PHASES

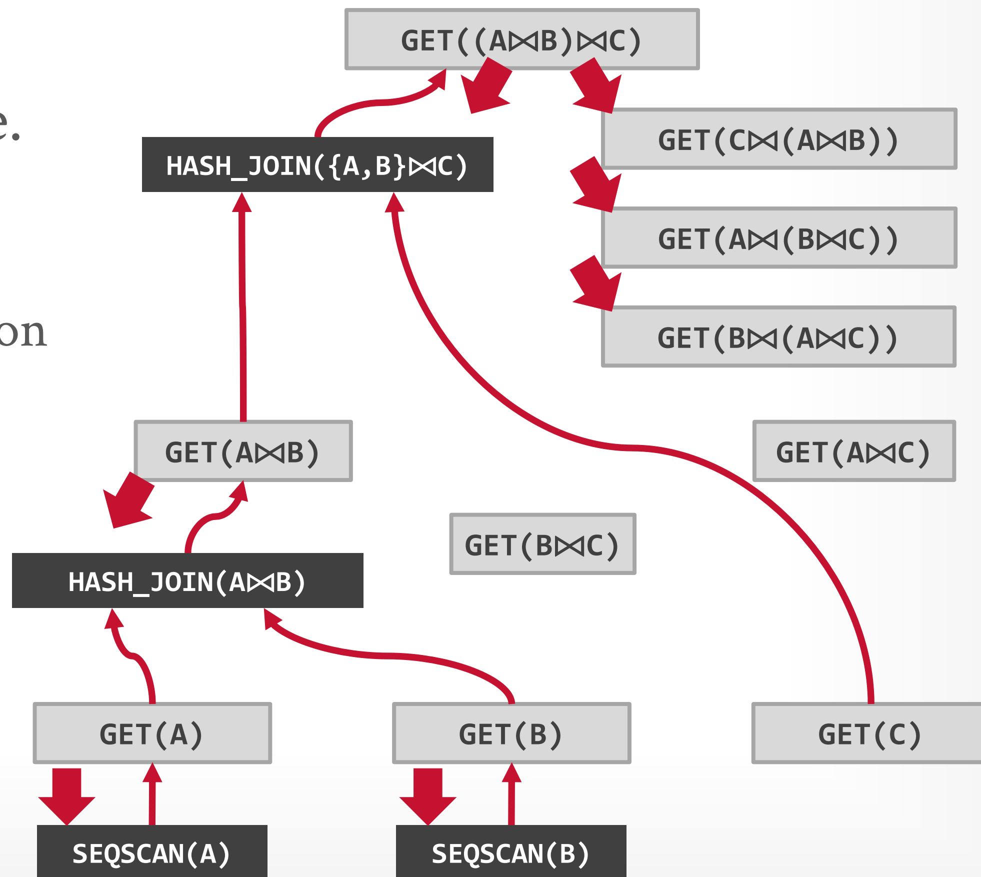
Start with a logical plan of what result the query needs to produce.

## Generation Phase:

→ Apply transformation rules to generate all possible logical expression alternatives.

## Cost Analysis Phase:

→ Apply implementation rules to generate physical operators.



Logical Op
Physical Op
Enforcer

# VOLCANO: PHASES



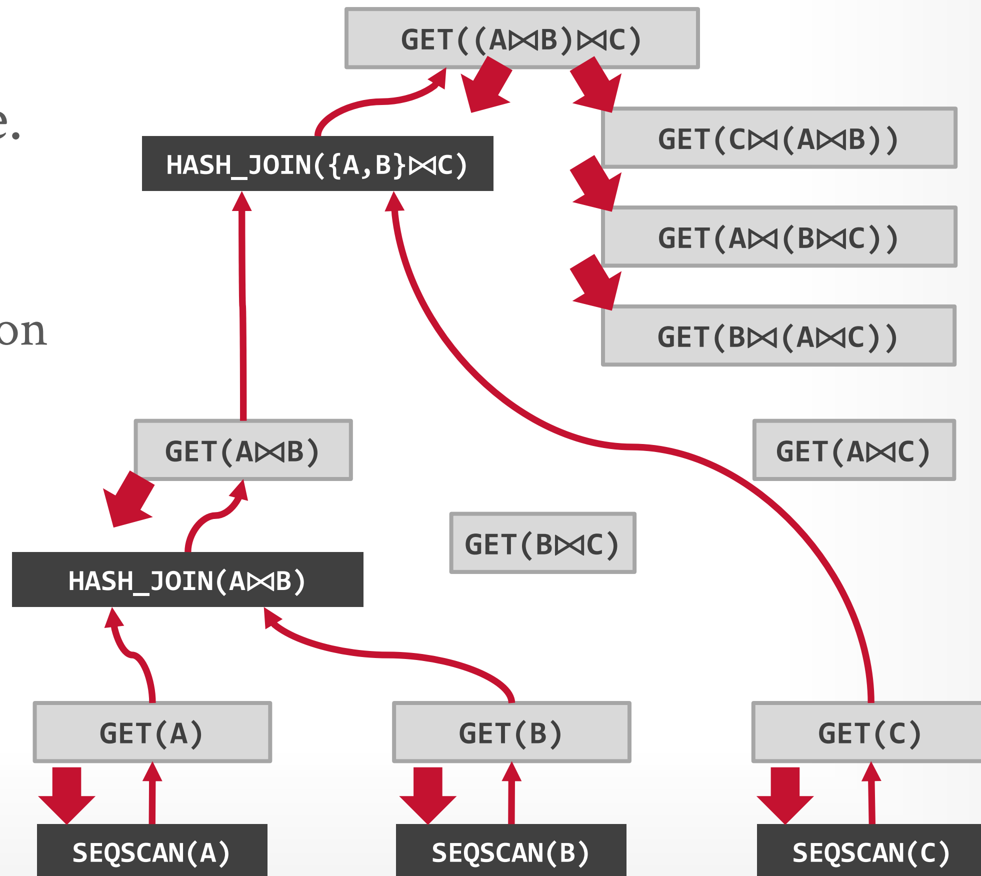
Start with a logical plan of what result the query needs to produce.

**Generation Phase:**

→ Apply transformation rules to generate all possible logical expression alternatives.

**Cost Analysis Phase:**

→ Apply implementation rules to generate physical operators.



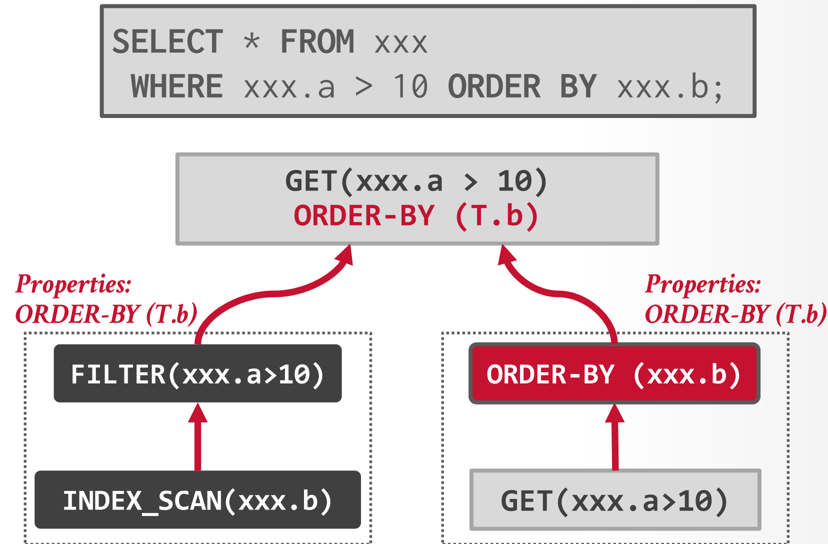
*Logical Op*
*Physical Op*
*Enforcer*

# VOLCANO: ENFORCERS



Enforcers are physical operators that ensure the properties of the output of a sub-plan / expression.

Volcano's rule engine has additional logical to avoid considering operators below it in the plan that satisfy its property requirements.
→ Example: **INDEX_SCAN(xxx.b)**



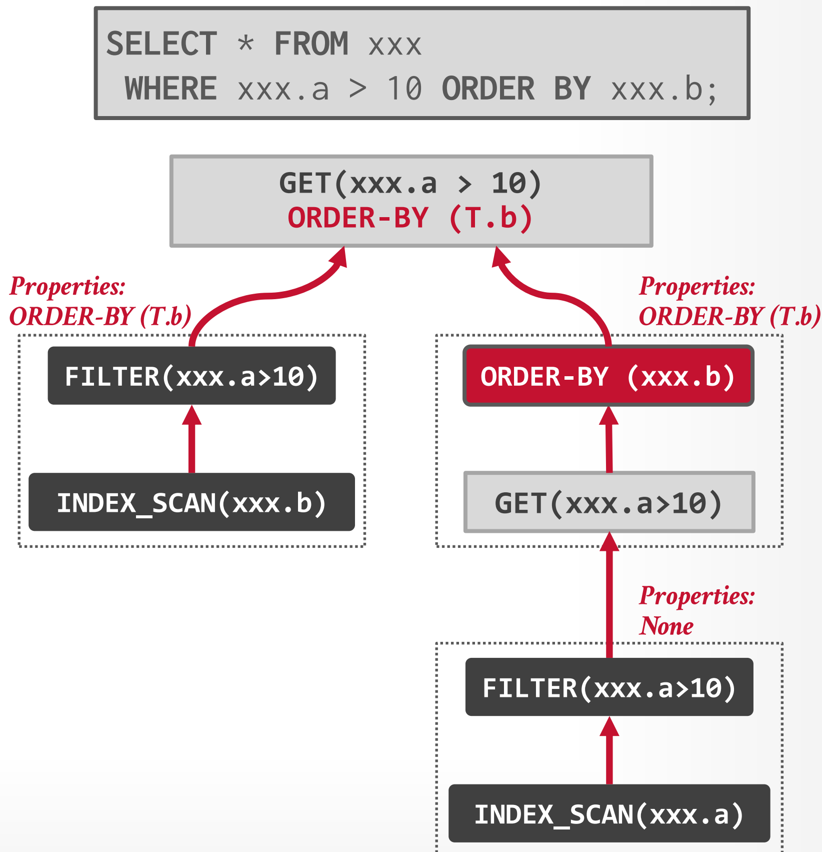

Source: EQOP Book

*Logical Op*
*Physical Op*
*Enforcer*

# VOLCANO: ENFORCERS

Enforcers are physical operators that ensure the properties of the output of a sub-plan / expression.

Volcano's rule engine has additional logical to avoid considering operators below it in the plan that satisfy its property requirements.
→ Example: **INDEX_SCAN(xxx.b)**



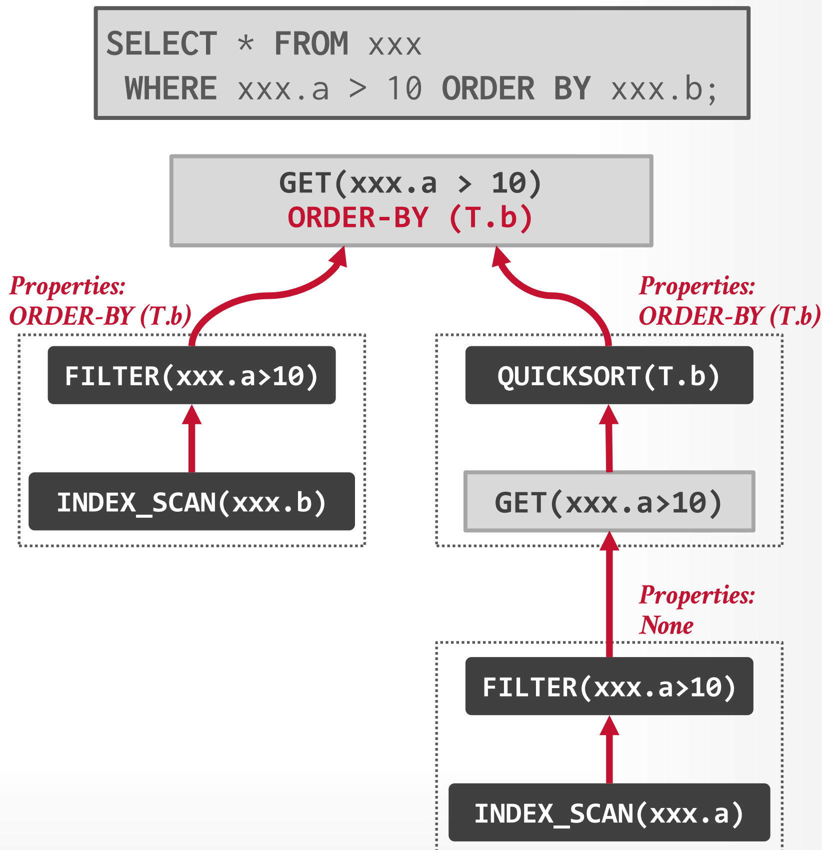

Source: EQOP Book

*Logical Op*
*Physical Op*
*Enforcer*

# VOLCANO: ENFORCERS

Enforcers are physical operators that ensure the properties of the output of a sub-plan / expression.

Volcano's rule engine has additional logical to avoid considering operators below it in the plan that satisfy its property requirements.
→ Example: **INDEX_SCAN(xxx.b)**




Source: EQOP Book

# TODAY'S AGENDA

Cascades Overview
Tasks / Scheduling
Optimizations
Implementations
Project #1

# CASCADES OPTIMIZER

Object-oriented implementation of the previous Volcano query optimizer.

→ **Top-down approach** (backward chaining) using branch-and-bound search.

→ Depth-first search ordering of tasks via a stack.

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

*Graefe*

THE CASCADES FRAMEWORK FOR QUERY OPTIMIZATION
*IEEE DATA ENGINEERING BULLETIN 1995*

EFFICIENCY IN THE COLUMBIA DATABASE QUERY OPTIMIZER
*PORTLAND STATE UNIVERSITY MS THESIS 1998*

# 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.

**Unified representation of rules & operators.**
→ Single search engine for logical and physical operators.

EFFICIENCY IN THE COLUMBIA DATABASE QUERY OPTIMIZER
*PORTLAND STATE UNIVERSITY MS THESIS 1998*

# 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.

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON C.id = A.id;
```

**Logical Expression:** $(A \bowtie B) \bowtie C$

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

# CASCADES: GROUPS

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 derived from selecting allowable physical operators for the corresponding logical forms.

**Group**

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
| --- | --- | --- |
| | 1. $(A \bowtie B) \bowtie C$ | 1. $(A_{Seq} \bowtie_{SM} B_{Seq}) \bowtie_{SM} C_{Seq}$ |
| | 2. $(B \bowtie C) \bowtie A$ | 2. $(A_{Seq} \bowtie_{HJ} B_{Seq}) \bowtie_{HJ} C_{Seq}$ |
| | 3. $(A \bowtie C) \bowtie B$ | 3. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$ |
| | 4. $A \bowtie (B \bowtie C)$ | 4. $(A_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} B_{Seq}$ |
| | ⋮ | 5. ⋮ |

# CASCADES: GROUPS

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 derived from selecting allowable physical operators for the corresponding logical forms.

| Group | Logical Exprs | Physical Exprs |
| --- | --- | --- |
| Output: {ABC} | 1. $(A \bowtie B) \bowtie C$ | 1. $(A_{Seq} \bowtie_{SM} B_{Seq}) \bowtie_{SM} C_{Seq}$ |
| | 2. $(B \bowtie C) \bowtie A$ | 2. $(A_{Seq} \bowtie_{HJ} B_{Seq}) \bowtie_{HJ} C_{Seq}$ |
| Properties: None | 3. $(A \bowtie C) \bowtie B$ | 3. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$ |
| | 4. $A \bowtie (B \bowtie C)$ | 4. $(A_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} B_{Seq}$ |
| | ⋮ | 5. ⋮ |

*Equivalent Expressions*

# CASCADES: GROUPS

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 derived from selecting allowable physical operators for the corresponding logical forms.

# CASCADES: GROUPS

Instead of explicitly instantiating all possible expressions in a group, the optimizer implicitly represents redundant expressions in a group with a placeholder (e.g., **{ABC}**).

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

| Output: **{ABC}** Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | 1. $\{AB\} \bowtie_{SM} \{C\}$ |
| | 2. {BC}⋈{A} | 2. $\{AB\} \bowtie_{HJ} \{C\}$ |
| | 3. {AC}⋈{B} | 3. $\{AB\} \bowtie_{NL} \{C\}$ |
| | 4. {A}⋈{BC} | 4. $\{BC\} \bowtie_{SM} \{A\}$ |
| | ⋮ | ⋮ |

# 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

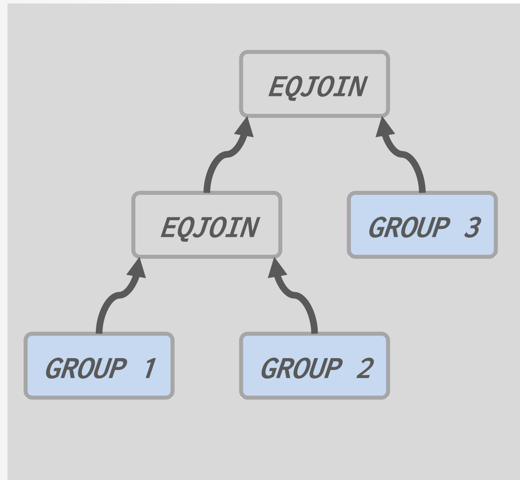
Each rule is represented as a pair of attributes:

→ **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

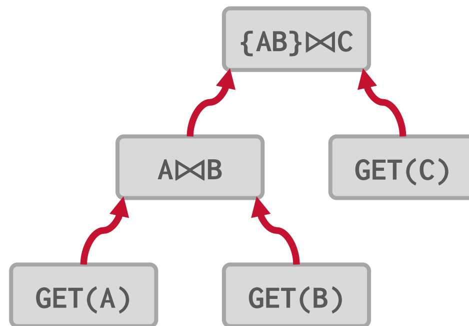
*Pattern*




*Group*

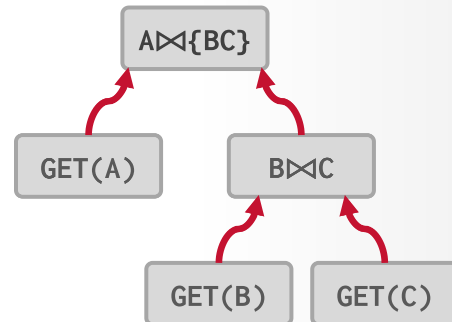
*Logical Expr*

*Physical Expr*

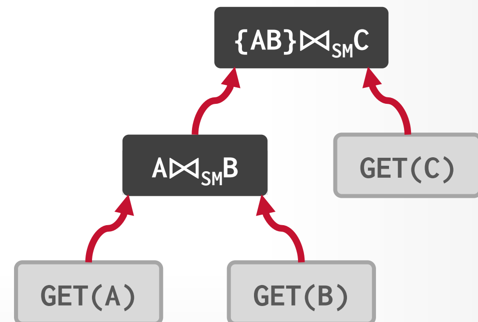



*Matching Plan*

*Transformation Rule*
*Rotate Left-to-Right*




*Implementation Rule*
*EQJOIN→SORTMERGE*



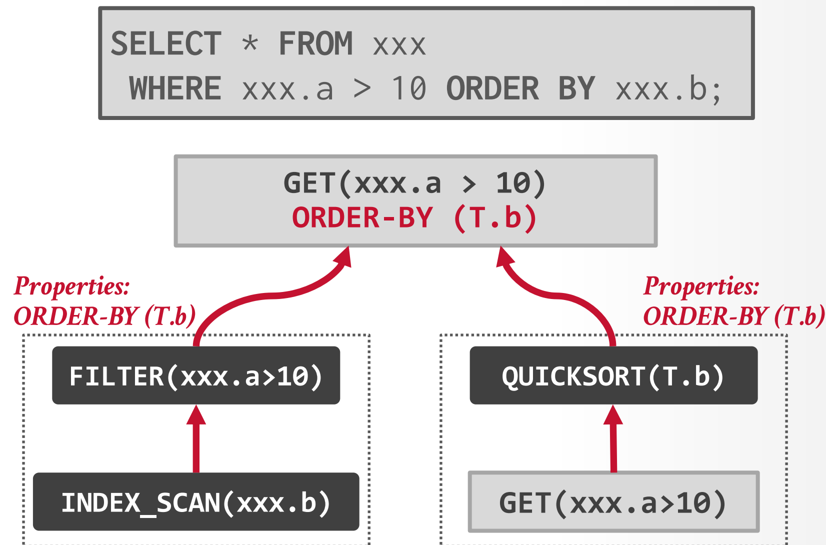
# CASCADES: ENFORCERS

Represent enforcers as rules that change physical properties of a plan.

→ An enforcer inserts physical operators that change the physical properties of the plan.

Cascades does not require special casing to implement enforcers.

→ Volcano integrated enforcers as operators via ad hoc code.



# CASCADES: TASKS

A **task** is a fine-grained unit of work that represents an operation in the query optimization process.

→ Break down optimization into smaller, manageable pieces to allow for more flexible and efficient exploration.

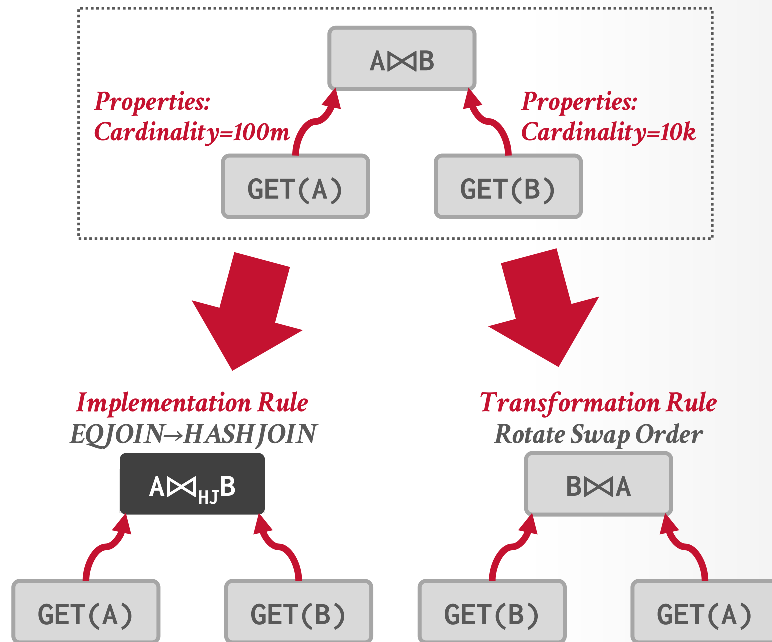
The order that a task is invoked can be customized via **promise**.

# CASCADES: PROMISES

A task's **promise** is the estimated benefit of a move on a given expression relative to other tasks.

→ Example: Give a higher priority to a join order swap rule if the outer table expression is larger than inner table expression.

Optimizer must still ensure tasks execute in the order defined by their dependencies.

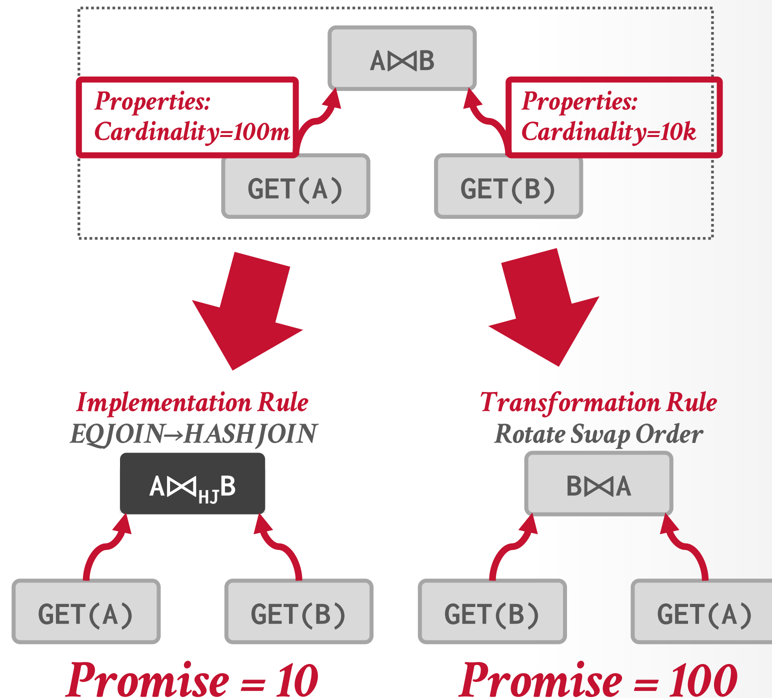


# CASCADES: PROMISES

A task's **promise** is the estimated benefit of a move on a given expression relative to other tasks.

→ Example: Give a higher priority to a join order swap rule if the outer table expression is larger than inner table expression.

Optimizer must still ensure tasks execute in the order defined by their dependencies.

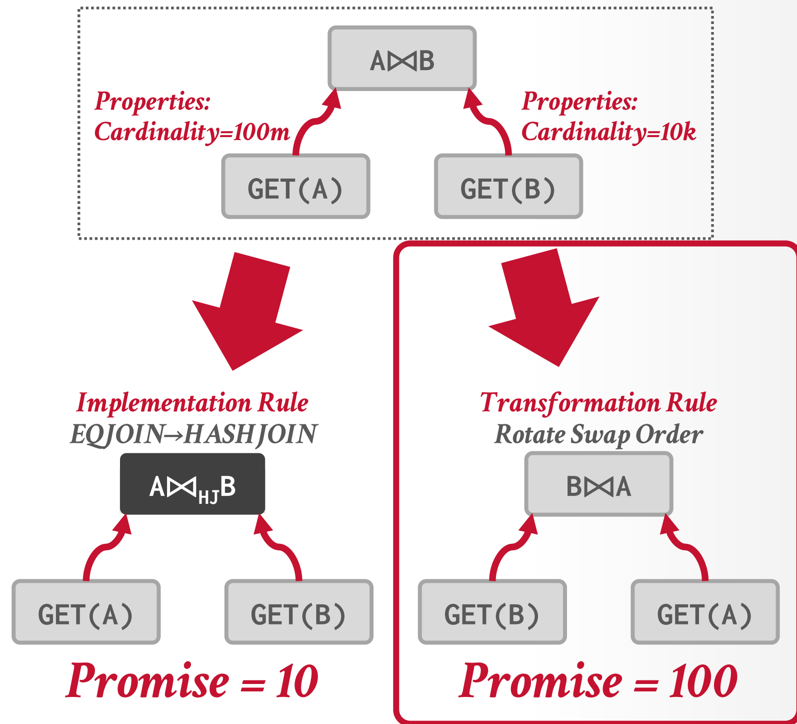


# CASCADES: PROMISES

A task's **promise** is the estimated benefit of a move on a given expression relative to other tasks.

→ Example: Give a higher priority to a join order swap rule if the outer table expression is larger than inner table expression.

Optimizer must still ensure tasks execute in the order defined by their dependencies.



# CASCADES: TASKS

**#1 – Optimize Group:**
→ Generate best physical plan for a group.

**#2 – Optimize Expression:**
→ Generate best physical plan for a specific expression.

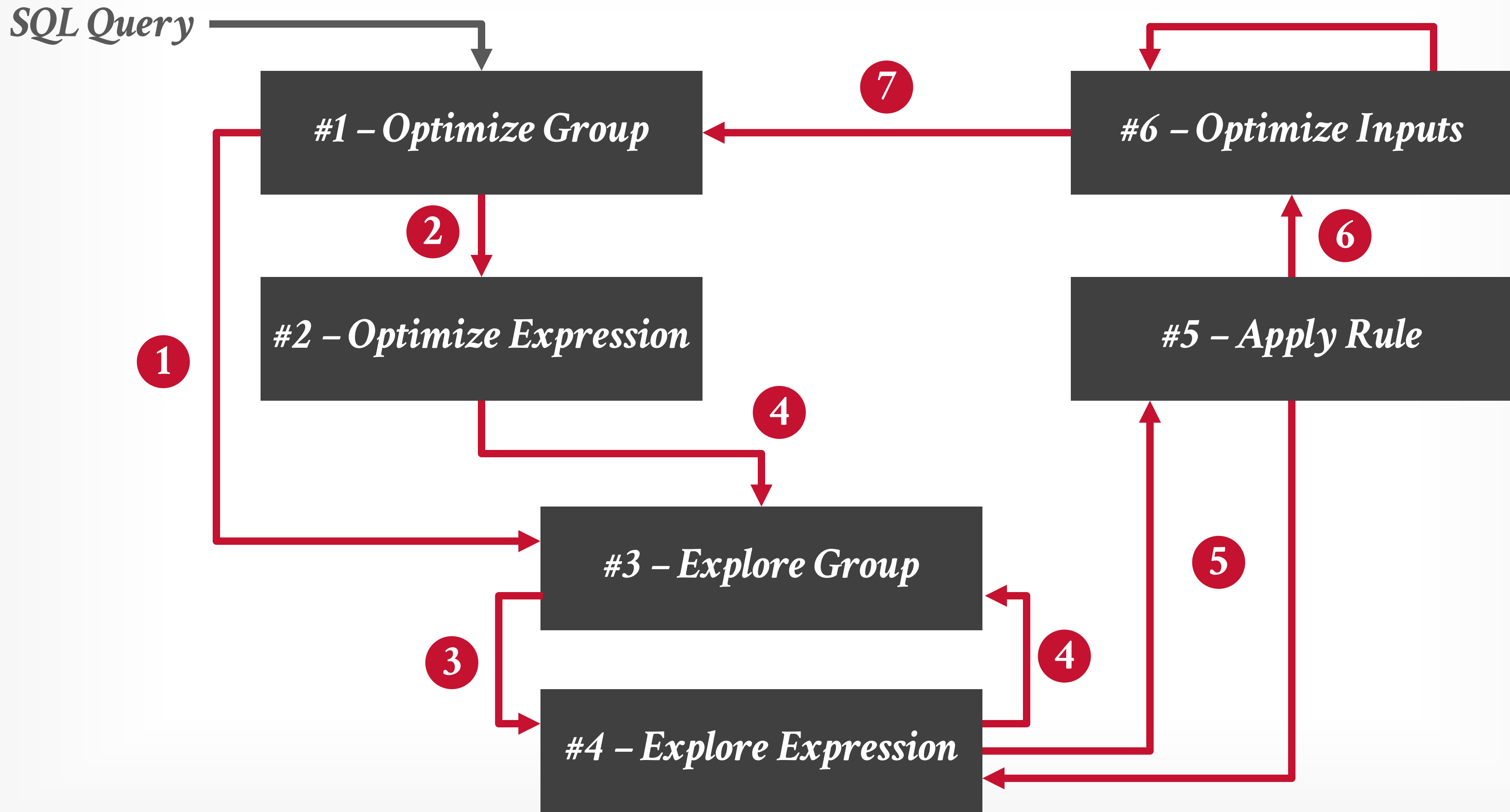
**#3 – Explore Group:**
→ Generate logical expressions for a group.

**#4 – Explore Expression:**
→ Generate logical transformations for a specific expression.

**#5 – Apply Rule:**
→ Apply a rule to an input expression.

**#6 – Optimize Inputs:**
→ Optimize the inputs of a given expression.

# CASCADES: TASK FLOW



# CASCADES: TASK-BASED SEARCH

Optimizer maintains a LIFO stack of tasks to perform actions on groups and expressions.

Stack ensures expressions are derived after the best plans of its input expressions are derived.

- → Removes sequential ordering of independent optimizations.
- → Tasks are stored in the heap rather than in the program stack to reduce OOM errors.

# 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
→ Superfluous Rule Evaluation

# 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 **P1** that has a greater cost than equivalent plan **P2** with the same physical properties.

EXPLOITING UPPER AND LOWER BOUNDS IN TOP-DOWN QUERY OPTIMIZATION
*IDEAS 2001*

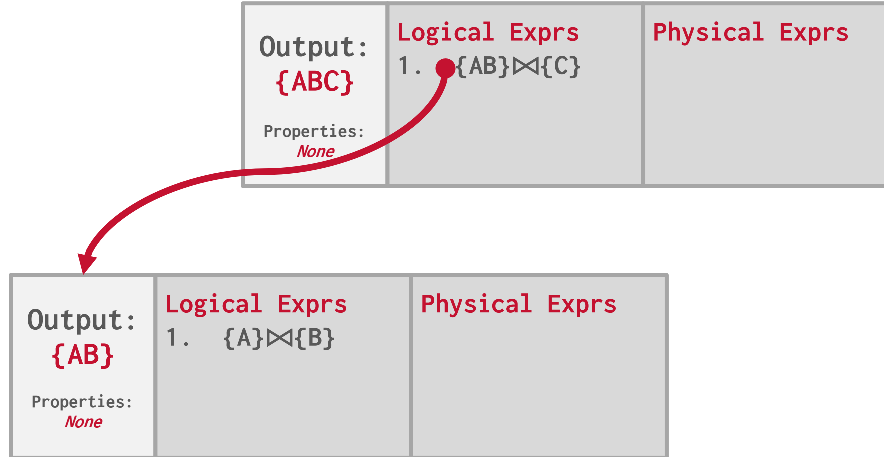
# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | | |
| {B} | | |
| {C} | | |

| Output: {ABC}<br>Properties: *None* | Logical Exprs<br>1. {AB}⋈{C} | Physical Exprs |
|---|---|---|

# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | | |
| {B} | | |
| {C} | | |



# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | | |
| {B} | | |
| {C} | | |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⨝{C} | |

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⨝{B} | |

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | |

# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | | |
| {B} | | |
| {C} | | |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B} | |

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | SeqScan(A) | 10 |
| {B} | | |
| {C} | | |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B} | |

***Cost: 10***

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A)<br>2. IdxScan(A) |

# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | SeqScan(A) | 10 |
| {B} | | |
| {C} | | |

Output: {ABC}
Properties: *None*
Logical Exprs
1. {AB}⋈{C}
Physical Exprs

Output: {AB}
Properties: *None*
Logical Exprs
1. {A}⋈{B}
Physical Exprs

**Cost: 10**

Output: {A}
Properties: *None*
Logical Exprs
1. GET(A)
Physical Exprs
1. SeqScan(A)
2. IdxScan(A)

Output: {B}
Properties: *None*
Logical Exprs
1. GET(B)
Physical Exprs

# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | SeqScan(A) | 10 |
| {B} | | |
| {C} | | |

| Output: {ABC}<br>Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

| Output: {AB}<br>Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B} | |

**Cost: 10**

| Output: {A}<br>Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

| Output: {B}<br>Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B) |
| | | 2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | | |

| Output: {ABC} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

| Output: {AB} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B} | |

**Cost: 10**

| Output: {A} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

**Cost: 20**

| Output: {B} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B) |
| | | 2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | | |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B}<br>2. {B}⋈{A} | |

***Cost: 10***

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A)<br>2. IdxScan(A) |

***Cost: 20***

| Output: {B} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B)<br>2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | | |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B}<br>2. {B}⋈{A} | |

***Cost: 10***

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A)<br>2. IdxScan(A) |

***Cost: 20***

| Output: {B} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B)<br>2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | | |
| {AB} | | |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | | |

| Output: {ABC} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

| Output: {AB} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B} | 1. {A}$\bowtie_{NL}${B} |
| | 2. {B}⋈{A} | 2. {A}$\bowtie_{HJ}${B} |
| | | 3. {B}$\bowtie_{NL}${A} |
| | | 4. {B}$\bowtie_{HJ}${A} |

**Cost: 10**

| Output: {A} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

**Cost: 20**

| Output: {B} Properties: None | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B) |
| | | 2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | | |
| {AB} | {A}⋈$_{HJ}${B} | 80 |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | | |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

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

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B} | 1. {A}⋈$_{NL}${B} |
| | 2. {B}⋈{A} | 2. {A}⋈$_{HJ}${B} |
| | | 3. {B}⋈$_{NL}${A} |
| | | 4. {B}⋈$_{HJ}${A} |

**Cost: 10**

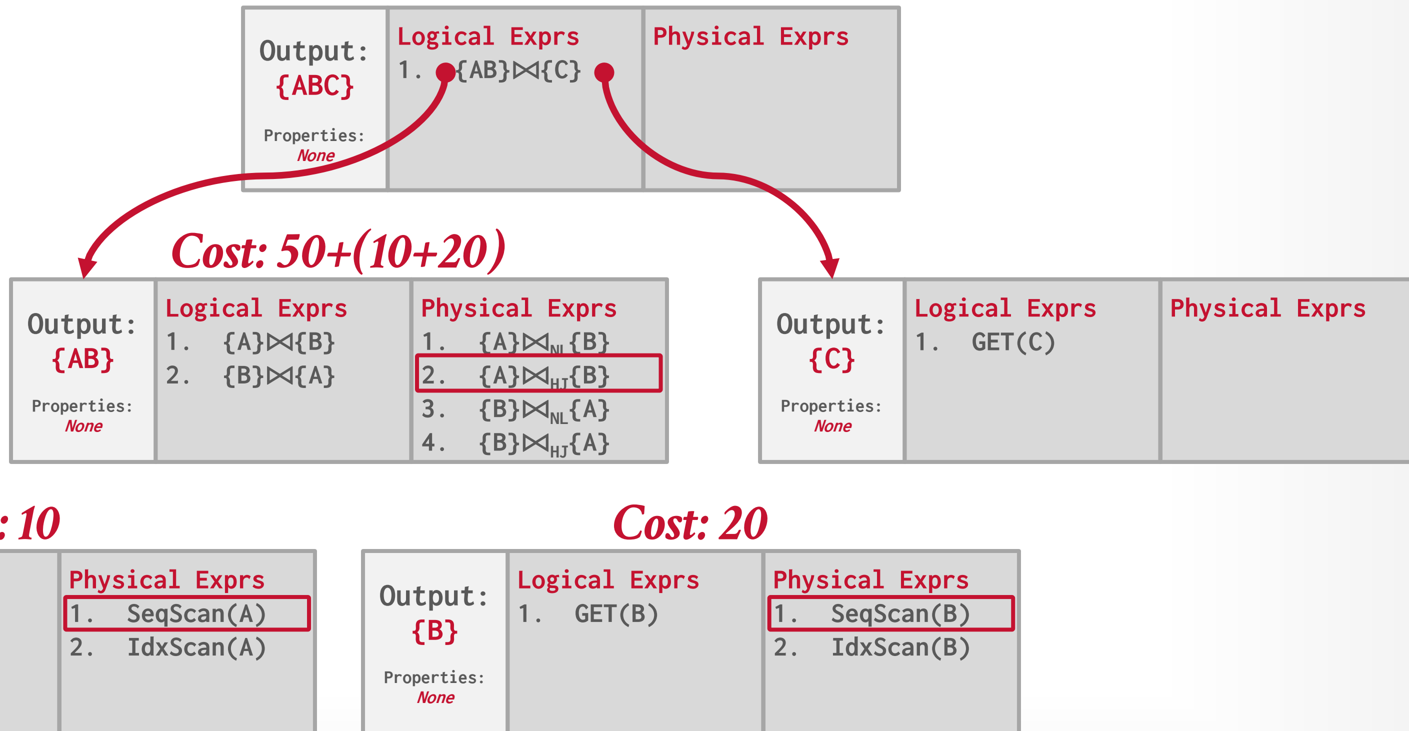
| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

**Cost: 20**

| Output: {B} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B) |
| | | 2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | $\{A\} \bowtie_{HJ} \{B\}$ | 80 |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | | |



# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | | |
| {AB} | {A}⋈$_{HJ}${B} | 80 |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | IdxScan(C) | 5 |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {AB}⋈{C} | |

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

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. {A}⋈{B} | 1. {A}⋈$_{NL}${B} |
| | 2. {B}⋈{A} | 2. {A}⋈$_{HJ}${B} |
| | | 3. {B}⋈$_{NL}${A} |
| | | 4. {B}⋈$_{HJ}${A} |

**Cost: 5**

| Output: {C} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(C) | 1. SeqScan(C) |
| | | 2. IdxScan(C) |

**Cost: 10**

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

**Cost: 20**

| Output: {B} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B) |
| | | 2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | *Best Expr* | *Cost* |
|---|---|---|
| {ABC} | | |
| {AB} | $\{A\} \bowtie_{HJ} \{B\}$ | 80 |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | IdxScan(C) | 5 |

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. $\{AB\} \bowtie \{C\}$ | 1. $\{AB\} \bowtie_{NL} C$ |
| | 2. $\{BC\} \bowtie \{A\}$ | 2. $\{BC\} \bowtie_{NL} A$ |
| | 3. $\{AC\} \bowtie \{B\}$ | 3. $\{AC\} \bowtie_{NL} B$ |
| | 4. $\{B\} \bowtie \{AC\}$ | ⋮ |

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

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. $\{A\} \bowtie \{B\}$ | 1. $\{A\} \bowtie_{NL} \{B\}$ |
| | 2. $\{B\} \bowtie \{A\}$ | 2. $\{A\} \bowtie_{HJ} \{B\}$ |
| | | 3. $\{B\} \bowtie_{NL} \{A\}$ |
| | | 4. $\{B\} \bowtie_{HJ} \{A\}$ |

**Cost: 5**

| Output: {C} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(C) | 1. SeqScan(C) |
| | | 2. IdxScan(C) |

**Cost: 10**

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

**Cost: 20**

| Output: {B} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B) |
| | | 2. IdxScan(B) |

# CASCADES: MEMO TABLE

| | Best Expr | Cost |
|---|---|---|
| {ABC} | $(\{A\}\bowtie_{HJ}\{B\})\bowtie_{HJ}\{C\}$ | 125 |
| {AB} | $\{A\}\bowtie_{HJ}\{B\}$ | 80 |
| {A} | SeqScan(A) | 10 |
| {B} | SeqScan(B) | 20 |
| {C} | IdxScan(C) | 5 |

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

| Output: {ABC} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. $\{AB\}\bowtie\{C\}$ | 1. $\{AB\}\bowtie_{NL}C$ |
| | 2. $\{BC\}\bowtie\{A\}$ | 2. $\{BC\}\bowtie_{NL}A$ |
| | 3. $\{AC\}\bowtie\{B\}$ | 3. $\{AC\}\bowtie_{NL}B$ |
| | 4. $\{B\}\bowtie\{AC\}$ | ⋮ |

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

| Output: {AB} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. $\{A\}\bowtie\{B\}$ | 1. $\{A\}\bowtie_{NL}\{B\}$ |
| | 2. $\{B\}\bowtie\{A\}$ | 2. $\{A\}\bowtie_{HJ}\{B\}$ |
| | | 3. $\{B\}\bowtie_{NL}\{A\}$ |
| | | 4. $\{B\}\bowtie_{HJ}\{A\}$ |

**Cost: 5**

| Output: {C} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(C) | 1. SeqScan(C) |
| | | 2. IdxScan(C) |

**Cost: 10**

| Output: {A} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(A) | 1. SeqScan(A) |
| | | 2. IdxScan(A) |

**Cost: 20**

| Output: {B} Properties: *None* | Logical Exprs | Physical Exprs |
|---|---|---|
| | 1. GET(B) | 1. SeqScan(B) |
| | | 2. IdxScan(B) |

# OBSERVATION

Promises enable Cascades to potentially target more beneficial transformations more quickly in the search process.

Cascades' flexible architecture enables other optimizations to reduce search times.

# SIMPLIFICATION RULES

Some rules simplify the logical query plan and almost always reduce its cost.

Instead of creating alternative expressions, a **simplification rule** replaces the expression with the transformed expression.

→ Removes the need to retain unnecessary state.

→ Equivalent to Starburst rewrite rules.

Source: EQOP Book

# MACRO RULES

To reduce the complexity of the search space and find a better plan more quickly, the optimizer can support **macro rules** that apply multiple transformations in a single rule.

This goes against the spirit of simple, independent rules as defined in the Volcano paper.

Source: EQOP Book

# PARALLEL SEARCH

If tasks are independent, then the optimizer can execute them in parallel on multiple threads.

→ Memo table is the shared state of the optimization process.

→ Need to ensure internal data structures are thread-safe.

AFAIK, Orca is the only multi-threaded Cascades optimizer implementation.

We will discuss this more in Lecture #17.

Source: EQOP Book

# CASCADES IMPLEMENTATIONS

**Standalone:**

- → Wisconsin OPT++ (1990s)
- → Portland State Columbia (1990s)
- → Greenplum Orca (2010s)
- → CMU optd (2025)

**Integrated:**

- → Microsoft SQL Server (1990s)
- → Tandem NonStop SQL (1990s)
- → Clustrix (2000s)
- → CockroachDB (2010s)
- → Snowflake (2010s)
- → Databricks (2010s)

# MICROSOFT SQL SERVER

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.

# MSSQL: MULTI-STAGE OPTIMIZATION

Sub-Query Removal
Outer Joins to Inner Joins
Predicate Pushdown
Empty Result Pruning

***Simplification / Normalization*** → ***Pre-Exploration*** → ***Exploration*** → ***Post-Optimization***

***Tree-to-Tree Transformations***

***Cost-based Search Initialization***

Trivial Plan Short-circuit
Projection Normalization
Statistics Identification/Collection
Initial Cardinality Estimates
Join Collapsing

Stage1: Trivial Plan
Stage2: Quick Plan (Parallel)
Stage3: Full Plan (Parallel)

***Multi-Stage Cost-Based Search***

***Engine-Specific Transformations***

Source: Nico Bruno + Cesar Galindo-Legaria

# MSSQL: OPTIMIZATIONS

**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.

Source: Nico Bruno + Cesar Galindo-Legaria

# GREENPLUM ORCA

Standalone Cascades implementation in C++.

→ Originally written for Greenplum.
→ Extended to support HAWQ.
→ Supports multi-threaded search.

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

Open-sourced in 2010s but then closed-sourced in 2024 after Broadcom acquisition.

ORCA: A MODULAR QUERY OPTIMIZER ARCHITECTURE FOR BIG DATA
*SIGMOD 2014*

# COCKROACHDB

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.

Source: Rebecca Taft

# COCKROACHDB

Custom Cascades im

All transformation r
DSL (OptGen) and t
→ Can embed Go logic i
analysis and modifica

Also considers scalar
transformations toge

## DSL: Optgen

```go
// ConstructNot constructs an expression for the Not operator.
func (_f *Factory) ConstructNot(input opt.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.mem.MemoizeNot(input)
  return _f.onConstructScalar(e)
}
```

Source: Rebecca Taft

# PARTING THOUGHTS

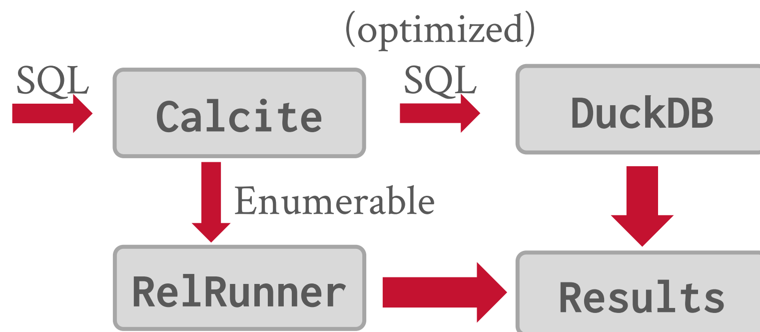
Andy prefers the Cascades' architecture because it is conceptually cleaner than other implementations.

→ The quality of a plan is still highly dependent on accurate statistics and cost estimations.

The Germans disagree with top-down search…

# PROJECT #1 – QUERY OPTIMIZER EVALUATION

You will use Apache Calcite to optimize SQL queries that are then executed on Calcite (via the Enumerable adapter) and DuckDB.



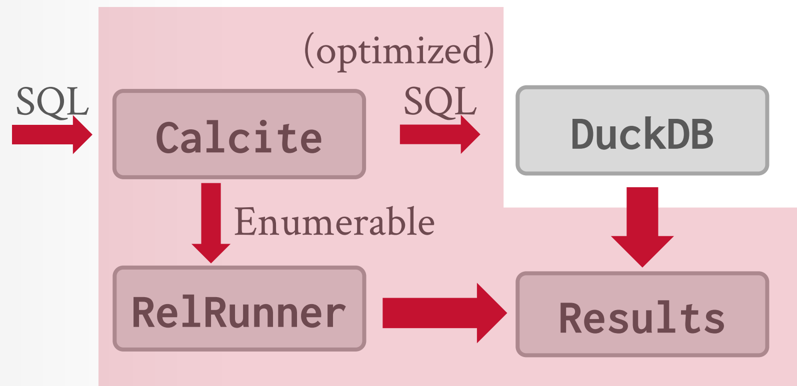

**Due Date: Friday Feb 28th**

**https://15799.courses.cs.cmu.edu/spring2025/project1.html**

# PROJECT #1 – TASKS

We provide starter code to build and run a Calcite application.

You need to implement the highlighted region.




```
Graded with: optimize.sh workload.tgz output_dir

For each foo.sql in workload.tgz/queries,
write to output_dir the following files:

foo.sql               [Original SQL query]
foo.txt               [Logical Calcite Plan]
foo_optimized.txt     [Enumerable Calcite Plan]
foo_results.csv       [Results from foo_optimized]
foo_optimized.sql     [Optimized SQL query]
```

You are graded against the reference solution on:

(1) RelRunner result correctness

(2) DuckDB performance with disable_optimizer

# DEVELOPMENT HINTS

Follow the roadmap.

Use our code for serializing the `txt` and `csv` files.

Beware of case sensitivity issues (e.g., table name).

The "standard library" of rules does **not** include operation-specific rules. You may need them.

If there's limited support for optimizing or deparsing `Foo`, maybe you can turn `Foo` into `Bar`. Then optimize and/or deparse that instead.

# THINGS TO NOTE

Do **not** hardcode based on the provided workload's data distribution – we test on a different workload.

Limit your memory usage to avoid autograder crashes: `java -Xmx4096m jar App.jar APP_ARGS`

Post your questions on Piazza.

Make sure you submit a reflection that documents what you tried. Bullet points are fine.

# NEXT CLASS

**Cascades Transformations**

→ Everything but sub-queries.
→ You do not need to submit a reading synopsis.