

# Lecture #23



Carnegie Mellon University

# ADVANCED DATABASE SYSTEMS

Optimizer Implementation  
(Part II)

@Andy\_Pavlo // 15-721 // Spring 2019

# DATABASE TALKS

---

## Amazon Redshift



- Ippokratis Pandis (PhD'07)
- Today @ 4:30pm
- DH 2315
- <https://db.cs.cmu.edu/events/spring-2019-ippokratis-pandis-phd07-amazon/>

## SAP HANA



- Anil Goel
- Thursday May 2<sup>nd</sup> @ 12:00pm
- CIC - 4th floor (ISTC Panther Hollow Room)
- <https://db.cs.cmu.edu/events/spring-2019-anil-goel-sap/>

# TODAY'S AGENDA

---

Cascades / Columbia

Plan Enumeration

Other Implementations



# QUERY OPTIMIZATION STRATEGIES

---

## **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: Randomized Search**

→ Academics in the 1980s, current Postgres

## **Choice #4: Stratified Search**

→ IBM's STARBURST (late 1980s), now IBM DB2 + Oracle

## **Choice #5: Unified Search**

→ Volcano/Cascades in 1990s, now MSSQL + Greenplum

# 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.

Implementation can be independent of the optimizer's search strategy.

Examples: Starburst, Exodus, Volcano, Cascades, OPT++

# 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 $\rightarrow$ logical and logical $\rightarrow$ physical transformations.

$\rightarrow$  No need for separate stages because everything is transformations.

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



# TOP-DOWN VS. BOTTOM-UP

---

## Top-down Optimization

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

## Bottom-up Optimization

- Start with nothing and then build up the plan to get to the final outcome that you want.
- Examples: System R, Starburst





# CASCADES OPTIMIZER

---

Object-oriented implementation of the Volcano query optimizer.

Simplistic expression re-writing can be through a direct mapping function rather than an exhaustive search.



Graefe

# CASCADES OPTIMIZER

---

Optimization tasks as data structures.

Rules to place property enforcers.

Ordering of moves by promise.

Predicates as logical/physical operators.



# CASCADES – EXPRESSIONS

A **expression** is an operator with zero or more input expressions.

```
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:  $(A_{Seq} \bowtie_{HJ} B_{Seq}) \bowtie_{NL} C_{Seq}$

# 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 that can be derived from selecting the allowable physical operators for the corresponding logical forms.

Output: [ABC]	Logical Exps	Physical Exps
	1. $(A \bowtie B) \bowtie C$	1. $(A_{Seq} \bowtie_{NL} B_{Seq}) \bowtie_{NL} C_{Seq}$
	2. $(B \bowtie C) \bowtie A$	2. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$
	3. $(A \bowtie C) \bowtie B$	3. $(A_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} B_{Seq}$
	4. $A \bowtie (B \bowtie C)$	4. $A_{Seq} \bowtie_{NL} (C_{Seq} \bowtie_{NL} B_{Seq})$
	⋮	⋮

# 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 that can be derived from selecting the allowable physical operators for the corresponding logical forms.

*Group*

	Logical Exps	Physical Exps
Output: [ABC]	1. $(A \bowtie B) \bowtie C$	1. $(A_{Seq} \bowtie_{NL} B_{Seq}) \bowtie_{NL} C_{Seq}$
	2. $(B \bowtie C) \bowtie A$	2. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$
	3. $(A \bowtie C) \bowtie B$	3. $(A_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} B_{Seq}$
	4. $A \bowtie (B \bowtie C)$	4. $A_{Seq} \bowtie_{NL} (C_{Seq} \bowtie_{NL} B_{Seq})$
	⋮	⋮

# 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 that can be derived from selecting the allowable physical operators for the corresponding logical forms.

<i>Group</i>	Output: [ABC]	Logical Exps	Physical Exps	<i>Equivalent Expressions</i>
		1. $(A \bowtie B) \bowtie C$ 2. $(B \bowtie C) \bowtie A$ 3. $(A \bowtie C) \bowtie B$ 4. $A \bowtie (B \bowtie C)$ $\vdots$	1. $(A_{Seq} \bowtie_{NL} B_{Seq}) \bowtie_{NL} C_{Seq}$ 2. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$ 3. $(A_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} B_{Seq}$ 4. $A_{Seq} \bowtie_{NL} (C_{Seq} \bowtie_{NL} B_{Seq})$ $\vdots$	

# 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.

Output: [ABC]	Logical Multi-Exps	Physical Multi-Exps
	1. $[AB] \bowtie [C]$	1. $[AB] \bowtie_{SM} [C]$
	2. $[BC] \bowtie [A]$	2. $[AB] \bowtie_{HJ} [C]$
	3. $[AC] \bowtie [B]$	3. $[AB] \bowtie_{NL} [C]$
	4. $[A] \bowtie [BC]$	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

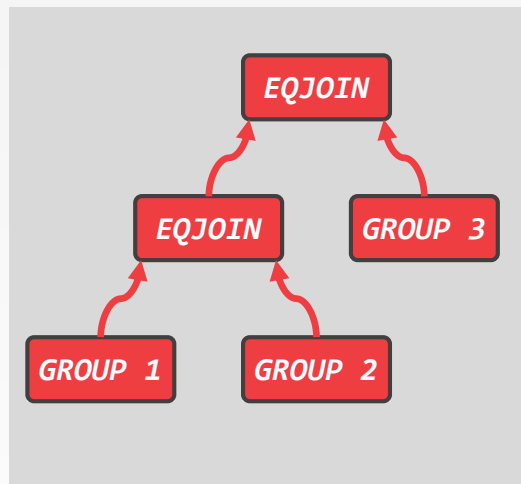
Each rule is represented as a pair of attributes:

- **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

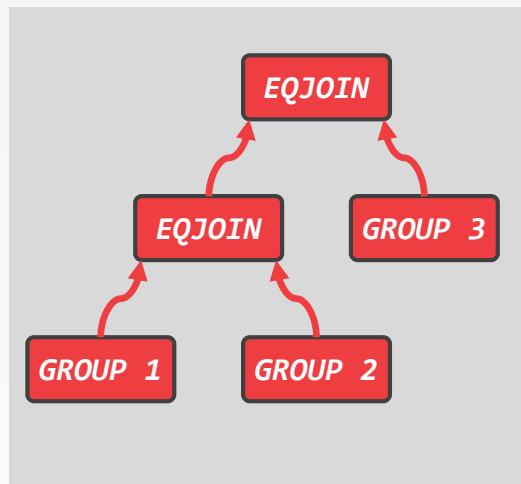
## Pattern



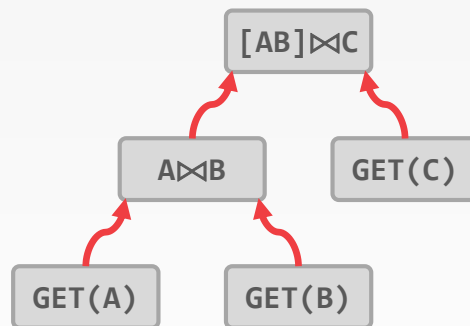
- *Group*
- *Logical Expr*
- *Physical Expr*

# CASCADES – RULES

## Pattern



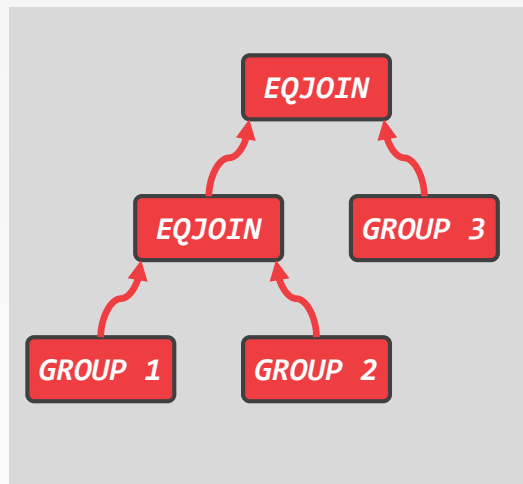
- Group*
- Logical Expr*
- Physical Expr*



## Matching Plan

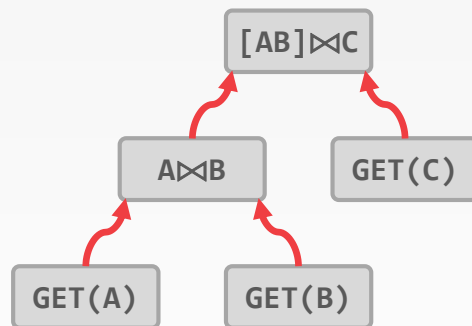
# CASCADES – RULES

## Pattern

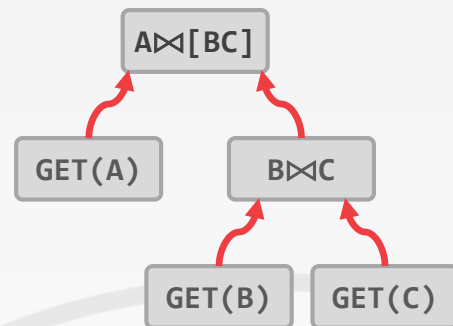


- Group
- Logical Expr
- Physical Expr

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

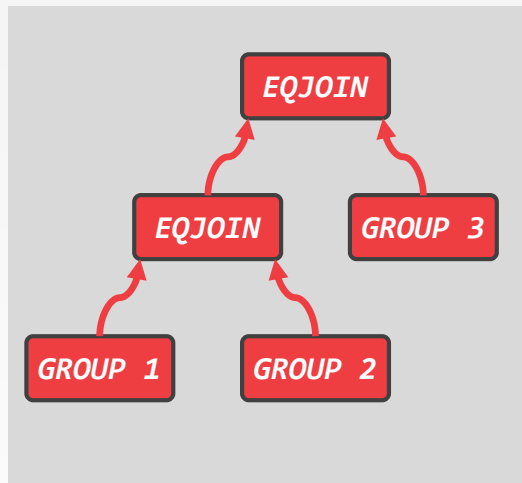


*Matching Plan*



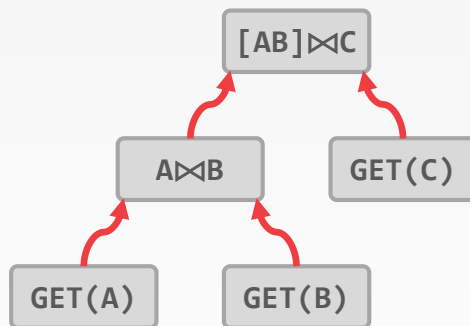
# CASCADES – RULES

## Pattern



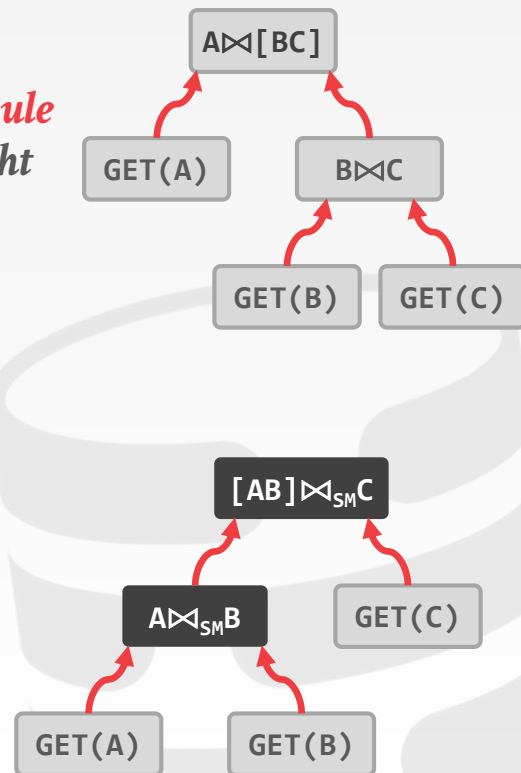
- Group**
- Logical Expr**
- Physical Expr**

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



*Matching Plan*

*Implementation Rule*  
*EQJOIN → SORTMERGE*



# 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 memoization, duplicate detection, and 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 **P1** that has a greater cost than equivalent plan **P2** with the same physical properties.

# CASCADES – MEMO TABLE

	<i>Best Expr</i>
[ABC]	
[AB]	
[A]	
[B]	
[C]	

	Logical M-Exps	Physical M-Exps
Output: [ABC]		

	Logical M-Exps	Physical M-Exps
Output: [AB]		

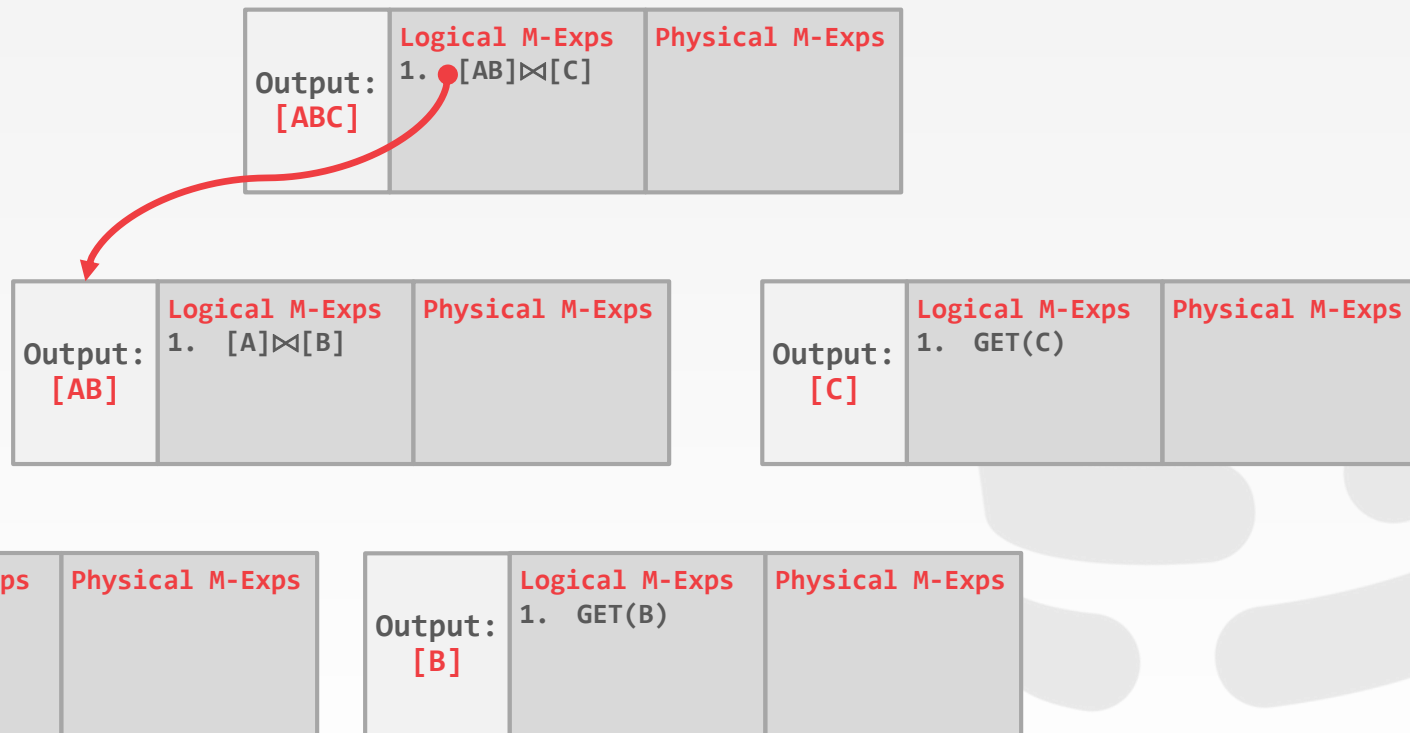
	Logical M-Exps	Physical M-Exps
Output: [C]	1. GET(C)	

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

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

# CASCADES – MEMO TABLE

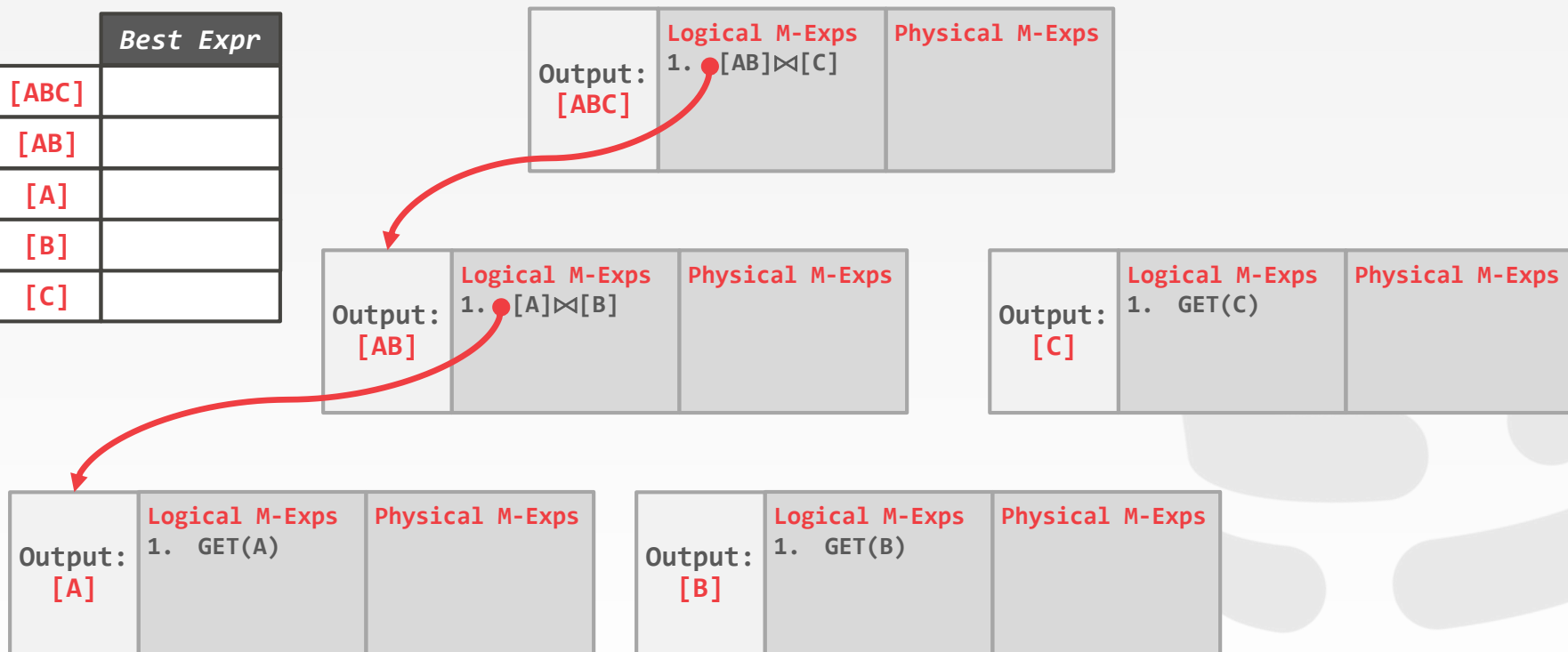
	Best Expr
[ABC]	
[AB]	
[A]	
[B]	
[C]	





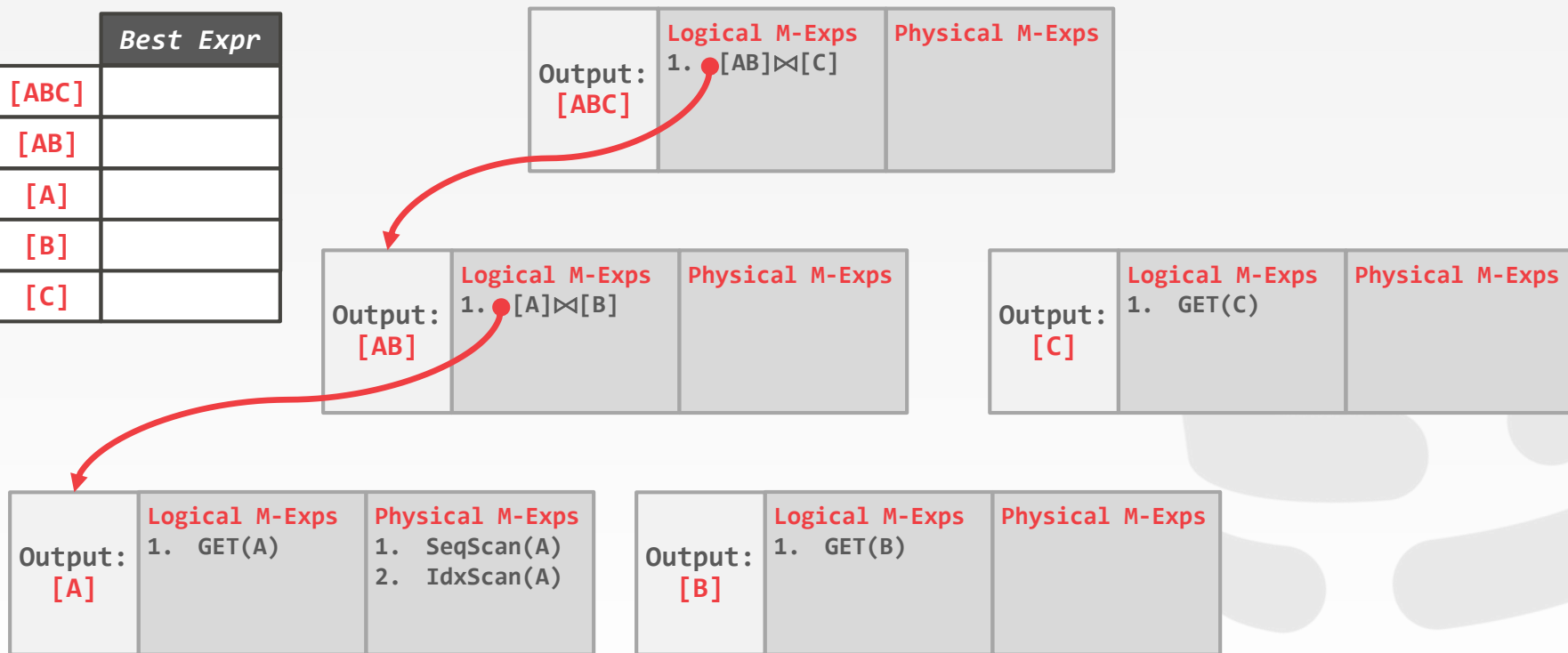
# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	
[A]	
[B]	
[C]	

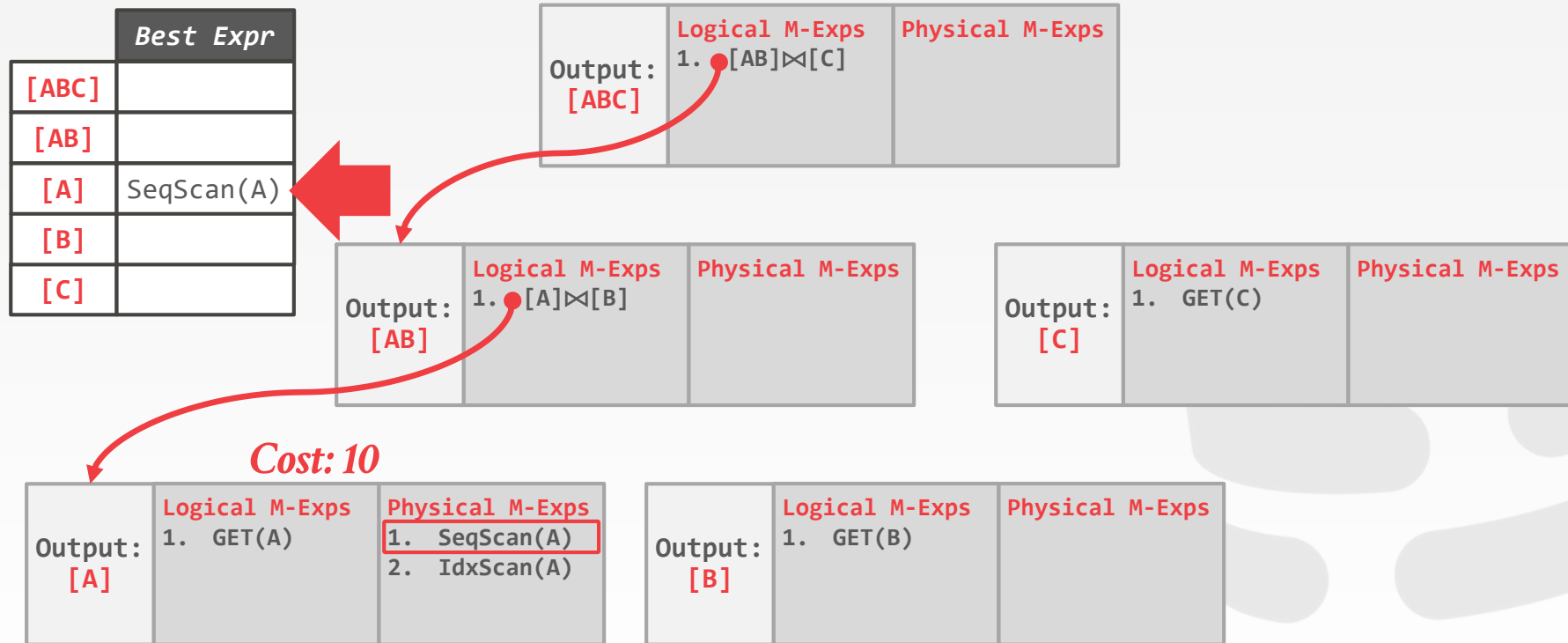


# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	
[A]	
[B]	
[C]	

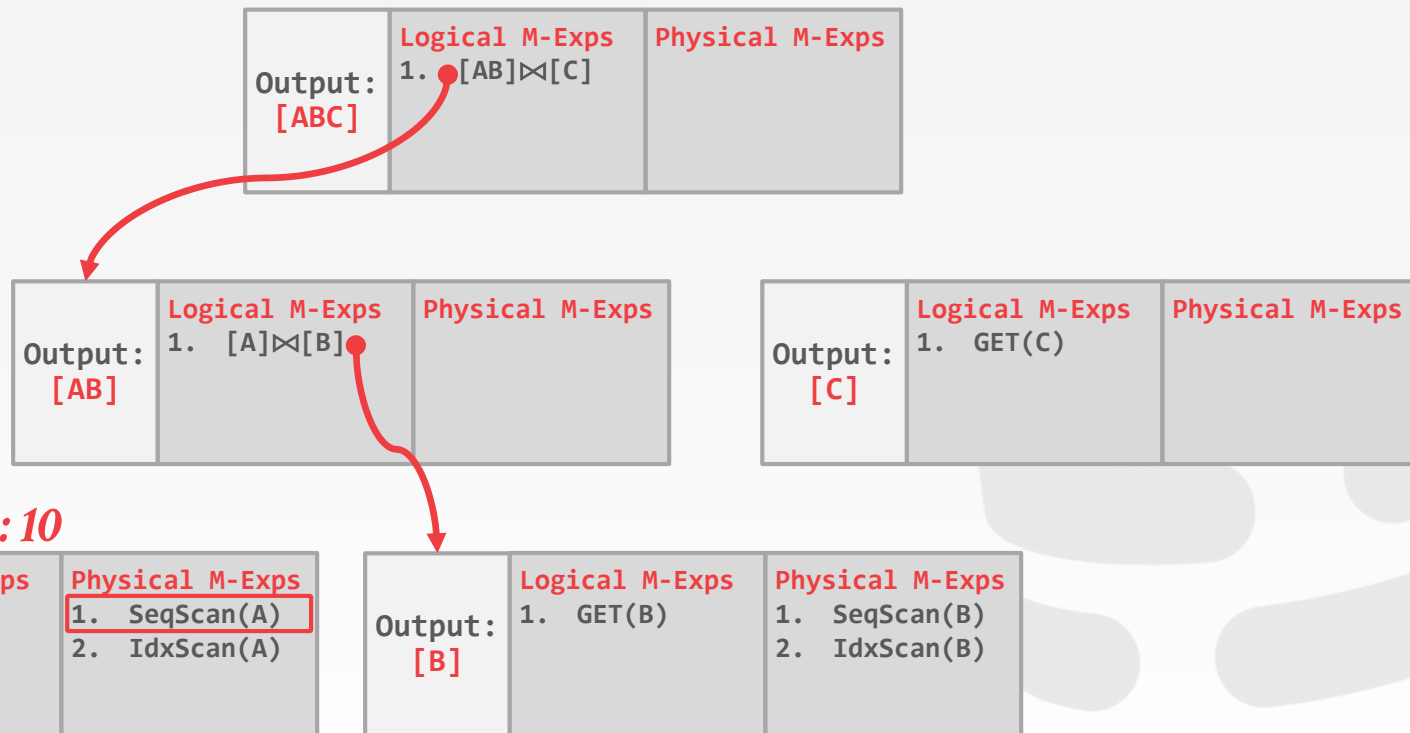


# CASCADES – MEMO TABLE



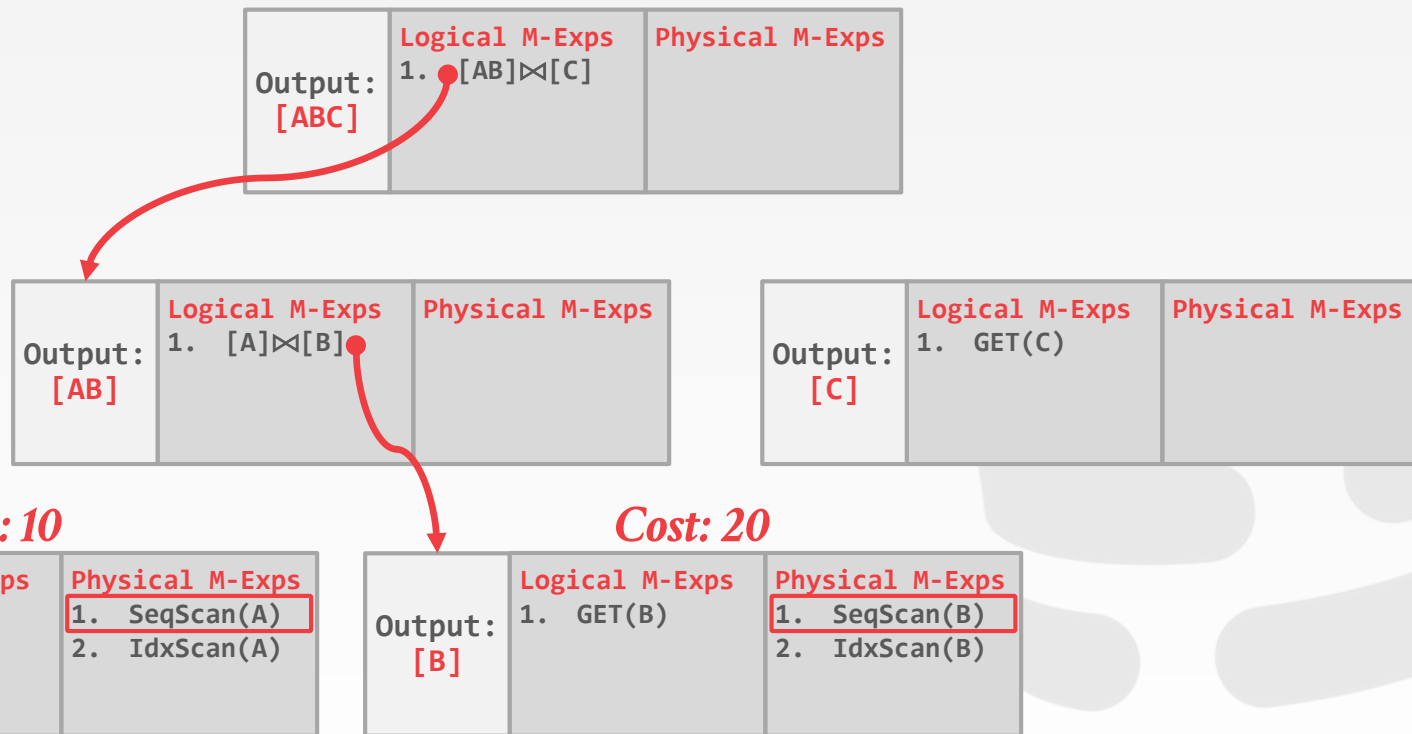
# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	
[A]	SeqScan(A)
[B]	
[C]	



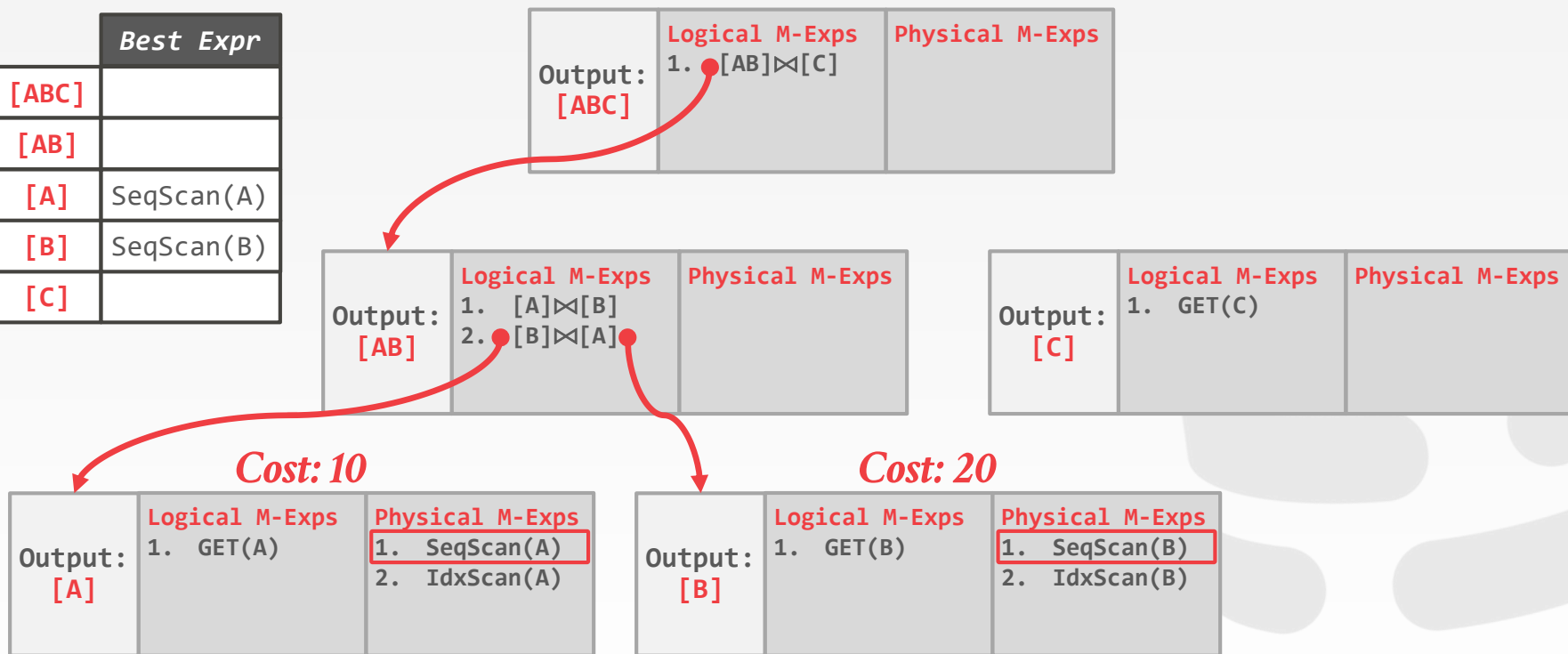
# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	
[A]	SeqScan(A)
[B]	SeqScan(B)
[C]	



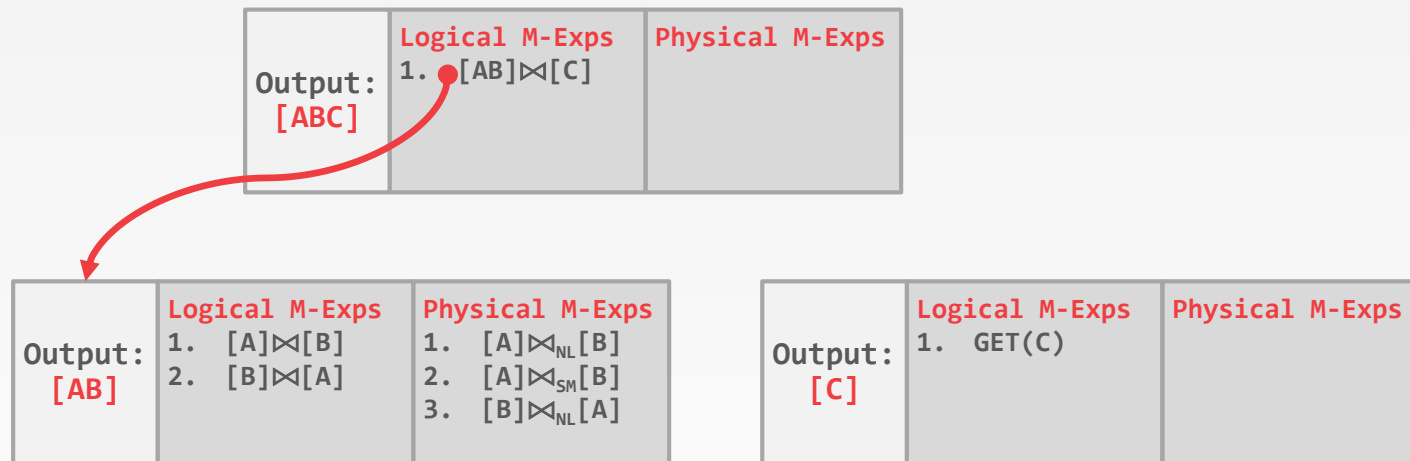
# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	
[A]	SeqScan(A)
[B]	SeqScan(B)
[C]	



# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	
[A]	SeqScan(A)
[B]	SeqScan(B)
[C]	



**Cost: 10**

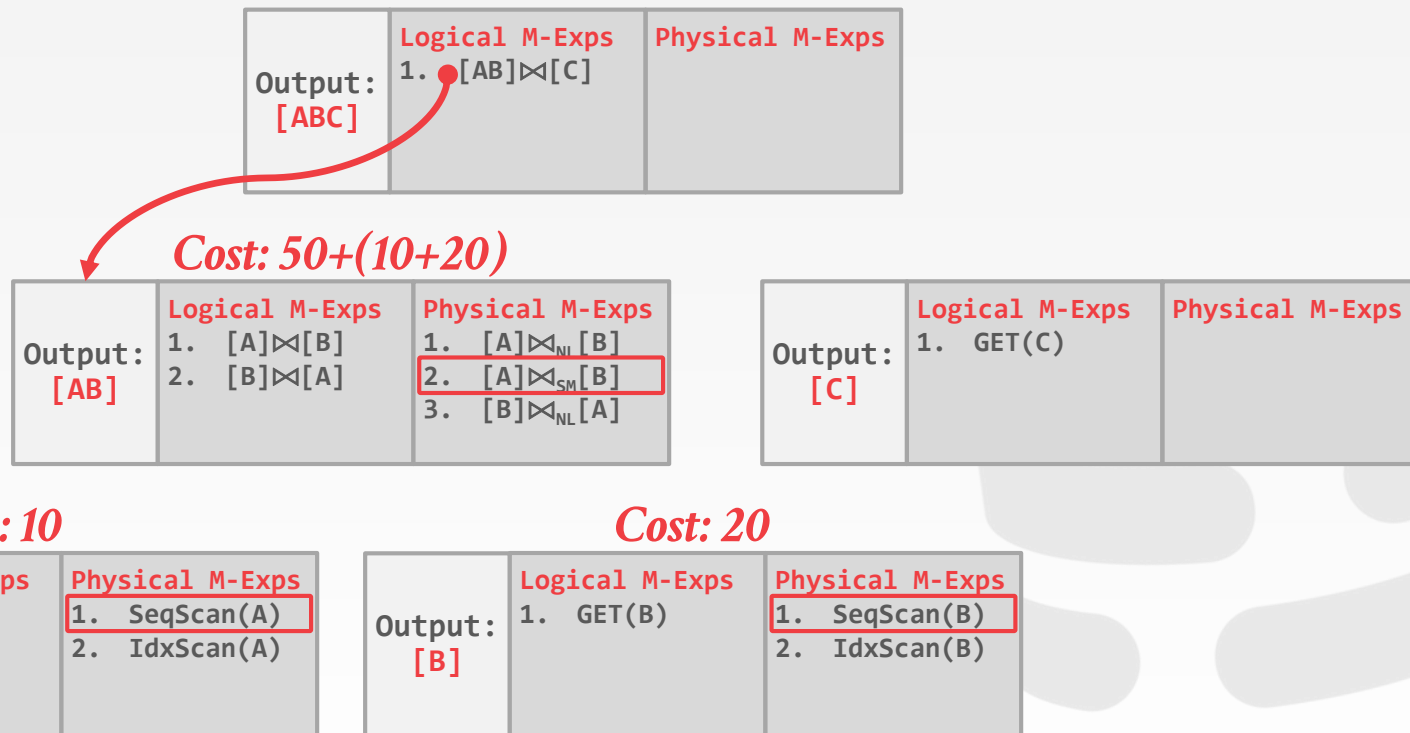
Output: [A]	Logical M-Exps 1. GET(A)	Physical M-Exps 1. SeqScan(A) 2. IdxScan(A)
----------------	-----------------------------	---

**Cost: 20**

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

# CASCADES – MEMO TABLE

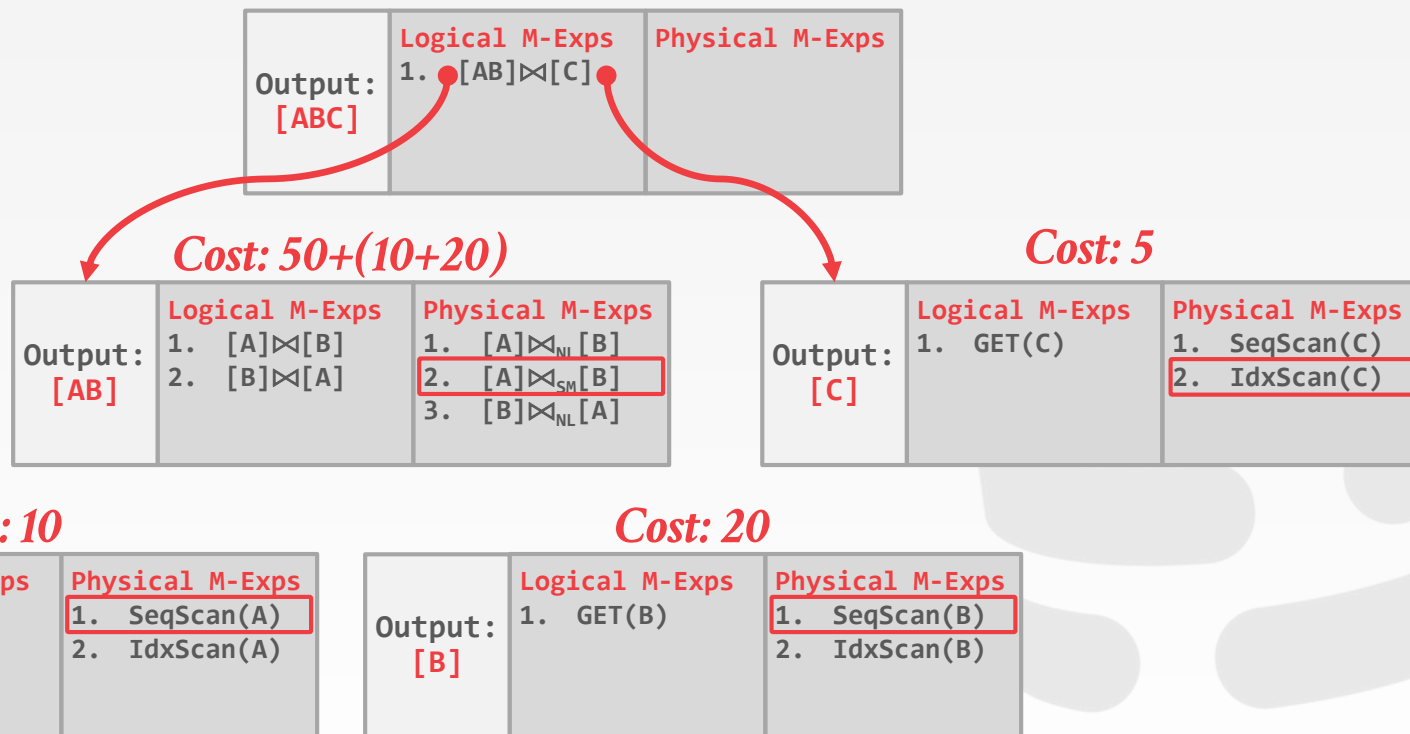
	Best Expr
[ABC]	
[AB]	$[A] \bowtie_{SM} [B]$
[A]	SeqScan(A)
[B]	SeqScan(B)
[C]	





# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	$[A] \bowtie_{SM} [B]$
[A]	SeqScan(A)
[B]	SeqScan(B)
[C]	IdxScan(C)



# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	$[A] \bowtie_{SM} [B]$
[A]	SeqScan(A)
[B]	SeqScan(B)
[C]	IdxScan(C)

	Logical M-Exps	Physical M-Exps
Output: [ABC]	1. $[AB] \bowtie [C]$ 2. $[BC] \bowtie [A]$ 3. $[AC] \bowtie [B]$ 4. $[B] \bowtie [AC]$	1. $[AB] \bowtie_{NL} C$ 2. $[BC] \bowtie_{NL} A$ 3. $[AC] \bowtie_{NL} B$ :

*Cost: 50+(10+20)*

	Logical M-Exps	Physical M-Exps
Output: [AB]	1. $[A] \bowtie [B]$ 2. $[B] \bowtie [A]$	1. $[A] \bowtie_{NL} [B]$ 2. $[A] \bowtie_{SM} [B]$ 3. $[B] \bowtie_{NL} [A]$

*Cost: 5*

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

*Cost: 10*

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

*Cost: 20*

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

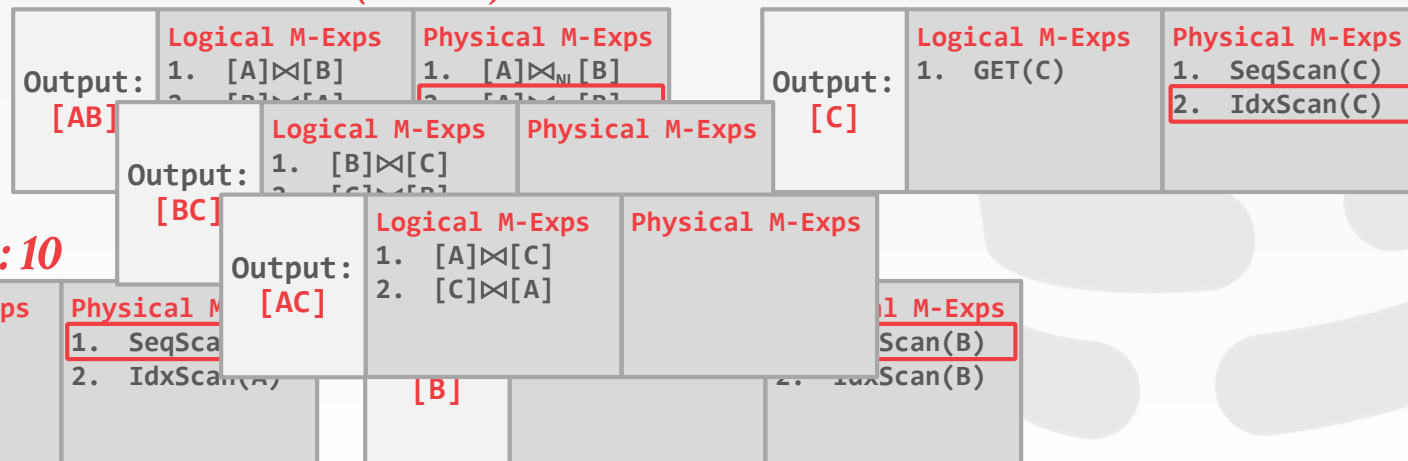
# CASCADES – MEMO TABLE

	Best Expr
[ABC]	
[AB]	$[A] \bowtie_{SM} [B]$
[A]	SeqScan(A)
[B]	SeqScan(B)
[C]	IdxScan(C)

	Logical M-Exps	Physical M-Exps
Output: [ABC]	1. $[AB] \bowtie [C]$ 2. $[BC] \bowtie [A]$ 3. $[AC] \bowtie [B]$ 4. $[B] \bowtie [AC]$	1. $[AB] \bowtie_{NL} C$ 2. $[BC] \bowtie_{NL} A$ 3. $[AC] \bowtie_{NL} B$ :

*Cost: 50+(10+20)*

*Cost: 5*



# 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: Transformation Exhaustion**

→ Stop when there are no more ways to transform the target plan. Usually done per group.

# CASCADES IMPLEMENTATIONS

---

## Standalone:

- Wisconsin OPT++ (1990s)
- Portland State Columbia (1990s)
- Pivotal Orca (2010s)
- Apache Calcite (2010s)

## Integrated:

- Microsoft SQL Server (1990s)
- Tandem NonStop SQL (1990s)
- Clustrix (2000s)
- CMU Peloton (2010s)



# OBSERVATION

---

All of the queries we have looked at so far have had the following properties:

- Equi/Inner Joins
- Simple join predicates that reference only two tables.
- No cross products

Real-world queries are much more complex:

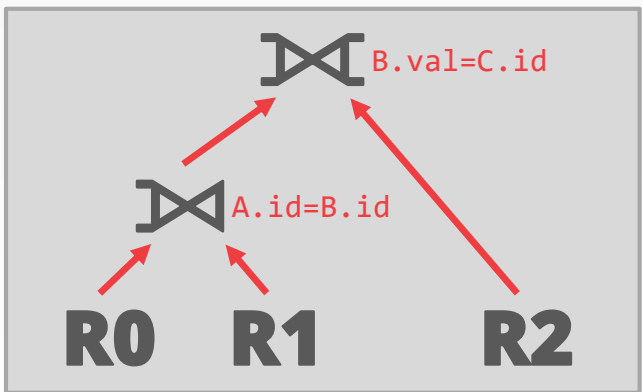
- Outer Joins
- Semi-joins
- Anti-joins



# REORDERING LIMITATIONS

```
SELECT * FROM A  
  LEFT OUTER JOIN B  
    ON A.id = B.id  
  FULL OUTER JOIN C  
    ON B.val = C.id);
```

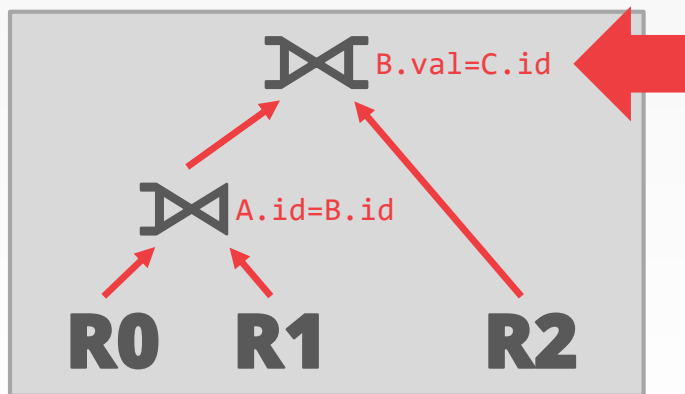
No valid reordering is possible.



Source: [Pit Fender](#)

# REORDERING LIMITATIONS

```
SELECT * FROM A
LEFT OUTER JOIN B
ON A.id = B.id
FULL OUTER JOIN C
ON B.val = C.id);
```



No valid reordering is possible.

The  $A \bowtie B$  operator is not commutative with  $B \bowtie C$ .

→ The DBMS does not know the value of **B.val** until after computing the join with **A**.



# PLAN ENUMERATION

---

How to generate different join orderings to feed into the optimizer's search model.

→ Need to be efficient to not slowdown the search.

**Approach #1: Generate-and-Test**

**Approach #2: Graph Partitioning**



# GERMANS

## Dynamic Programming Strikes Back

Guido Moerkotte  
University of Mannheim  
Mannheim, Germany  
moerkotte@informatik.uni-mannheim.de

Thomas Neumann  
Max-Planck Institute for Informatics  
Saarbrücken, Germany  
neumann@mpi-inf.mpg.de

### ABSTRACT

Two highly efficient algorithms are known for optimally ordering joins while avoiding cross products: DPtop, which is based on dynamic programming, and Top-Down Partition Search, based on memoization. Both have two severe limitations. They handle only (1) simple (binary) join predicates and (2) inner joins. However, real queries may contain complex join predicates, involving more than two relations, and outer joins as well as other non-inner joins.

Taking the most efficient known join-ordering algorithm, DPtop, as a starting point, we first develop a new algorithm, DPtop, which is capable to handle complex join predicates efficiently. We do so by modeling the query graph as a (view of a) hypergraph and then reason about its connected subgraphs. Then, we present a technique to exploit this capability to efficiently handle the widest class of non-inner joins dealt with so far. Our experimental results show that this reformulation of non-inner joins as complex predicates can improve optimization time by orders of magnitude, compared to known algorithms dealing with complex join predicates and non-inner joins. Once again, this gives dynamic programming a distinct advantage over current memoization techniques.

### Categories and Subject Descriptors

H.1.1 Systems: Query processing

### General Terms

Algorithms, Theory

### 1. INTRODUCTION

For the overall performance of a database management system, the cost-based query optimizer is an essential piece of software. One important and complex problem one cost-based query optimizer has to solve is to find the optimal join order. In their seminal paper, Selinger et al. not only introduced cost-based query optimization but also proposed

a dynamic programming algorithm to find the optimal join order for a given conjunctive query [21]. More precisely, they proposed to generate plans in the order of increasing size. Although they restricted the search space to left-deep trees, the general idea of their algorithm can be extended to the algorithm DPtop, which explores the space of bushy trees (see Fig. 1). The algorithm still forms the core of state-of-the-art commercial query optimizers like the one of DB2 [12].

Recently, we gave a thorough complexity analysis of DPtop [17]. We proved that DPtop has a runtime complexity which is much worse than the lower bound. This is mainly due to the tests marked by “\*” in Fig. 1, which fail far more often than they succeed. Furthermore, we proposed the algorithm DPtop, which exactly meets the lower bound. Experiments showed that DPtop is highly superior to DPtop. The core of their algorithm generates connected subgraphs in a bottom-up fashion.

The main competitor for dynamic programming is memoization, which generates plans in a top-down fashion. All known approaches needed tests similar to those shown for DPtop. Thus, with the advent of DPtop, dynamic programming became superior to memoization when it comes to generating optimal bushy join trees, which do not contain cross products. Challenged by this finding, DeRaaen and Topsi successfully devised a top-down algorithm that is capable of generating connected subgraphs by exploiting minimal cuts [7]. With this algorithm, called Top-Down Partition Search, memoization can be almost as efficient as dynamic programming.

However, both algorithms, DPtop and Top-Down Partition Search, are not ready yet to be used in practice: there exist two severe deficiencies in both of them. First, as has been argued in several places, hypergraphs must be handled by any join generator [1, 19, 25]. Second, plan generators have to deal with outer joins and join types [1, 19]. These operations are, in general, not freely nondistributable. That is, there might exist different orderings, which produce different results. This is not true for the regular, inner joins: any ordering gives the same result. Restricting the ordering to valid orderings for outer joins, that is those which produce the same result as the original query, has been the subject of the seminal work by Chandra-Levy and Borestein [10, 11, 26]. They also propose a dynamic programming algorithm that takes into account the integrality of outer joins. Their algorithm has been extended by Blugstein et al. to deal with hypergraphs [1]. A more practical approach has been proposed by Ren et al. [19]. They also include the

## Counter Strike: Generic Top-Down Join Enumeration for Hypergraphs

Pit Fender  
University of Mannheim  
Mannheim, Germany  
pfender@informatik.uni-mannheim.de

Guido Moerkotte  
University of Mannheim  
Mannheim, Germany  
moerkotte@informatik.uni-mannheim.de

### ABSTRACT

Finding the optimal execution order of join operations is a crucial task of today’s cost-based query optimizers. There are two approaches to identify the best plan: bottom-up and top-down join enumeration. But only the top-down approach allows for branch-and-bound pruning, which can improve compile time by several orders of magnitude while still preserving optimality. For both optimization strategies, efficient enumeration algorithms have been published. However, there are two severe limitations for the top-down approach. The published algorithms can handle only (1) simple (binary) join predicates and (2) inner joins. Since real queries may contain complex join predicates involving more than two relations, and outer joins as well as other non-inner joins, efficient top-down join enumeration cannot be used in practice yet. We develop a novel top-down join enumeration algorithm that overcomes these two limitations. Furthermore, we show that our new algorithm is competitive when compared to the state of the art in bottom-up processing even without playing out its advantage by making use of its branch-and-bound pruning capabilities.

### 1. INTRODUCTION

For a DBMS that provides support for a declarative query language like SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows it to be translated into many equivalent evaluation plans. Essential for the execution costs of a plan is the order of join operations, since the runtime of plans with different join orders can vary by several orders of magnitude. The search space considered here consists of all bushy join trees without cross products [19].

In principle, there are two approaches to find an optimal join order: bottom-up join enumeration via dynamic programming and top-down join enumeration through memoization. Both approaches face the same challenge: to efficiently find, for a given set of relations, all partitions into two subsets, such that both induce connected subgraphs and there exists an edge connecting the two subgraphs.

Currently, the following algorithms have been proposed: DPtop, an efficient dynamic programming-based algorithm [12], TDMCUTBRANCH [13], as well as TDMCUTBRANCH and TDMIN-

CUTCONSERVATIVE, two competitive top-down join enumeration strategies [6, 7, 5].

However, all four algorithms (DPtop, TDMCUTBRANCH, TDMIN-CUTBRANCH, TDMIN-CUTCONSERVATIVE) are not ready yet to be used in real-world scenarios because there exist two severe deficiencies in all of them. First, as has been argued in several places, hypergraphs must be handled by any plan generator [23, 25]. Second, plan generators have to deal with outer joins and join types [6, 7]. In general, these operations are not freely nondistributable: some orderings produce wrong results. The non-inner join reordering problem can be correctly reduced to hypergraphs [23, 25, 17]. Consequently, Moerkotte and Neumann [19] extended DPtop to DPtop to handle hypergraphs. Since DPtop is a bottom-up join enumeration algorithm, it cannot benefit from branch-and-bound pruning. On the other hand, branch-and-bound pruning can significantly speed up plan generation [12], while still guaranteeing plan optimality.

In this paper, we present a novel generic framework that can be used by any existing partitioning algorithm for top-down join enumeration to efficiently handle hypergraphs. The central idea is to smartly connect hypergraphs to simple graphs and introduce effective means to avoid inefficiencies. This way, any existing partitioning algorithm for simple graphs can be used. We show that TDMCUTBRANCH, resulting from instantiating our framework with the partitioning algorithm MINCUTBRANCH, is more efficient than existing partitioning algorithms for hypergraphs and as efficient as DPtop even without pruning. With pruning, TDMCUTBRANCH outperforms DPtop by a factor of 1.1 – 1.5.

This paper is organized as follows. Sec. 2 recalls some preliminaries. Sec. 3 shows a novel approach called TDMCUTBRANCH for handling hypergraphs. Sec. 4 presents our generic framework. Sec. 5 contains the experimental evaluation, and Sec. 6 concludes the paper.

### 2. PRELIMINARIES

Before we give the formal definitions necessary for our algorithm, let us demonstrate by means of a very simple example why hypergraphs are important for the case where join predicates connect more than two relations are necessary when reordering more than plan join. Consider the query

select \* from R0 join R1 on R0.A = R1.B  
full outer join R2 on R1.C = R2.D  
In a first step, it is translated into an initial ordering as follows:  
 $(R_0 \bowtie_{A=B} R_1) \bowtie_{C=D} R_2$

For this query, no valid reordering is possible. To prevent reordering, conflicts need to be detected and represented. A core of every conflict presentation is a set of relations, called TES, associated with each operator in the initial operator tree [13, 17, 11]. To

SELECT JOIN B 2013

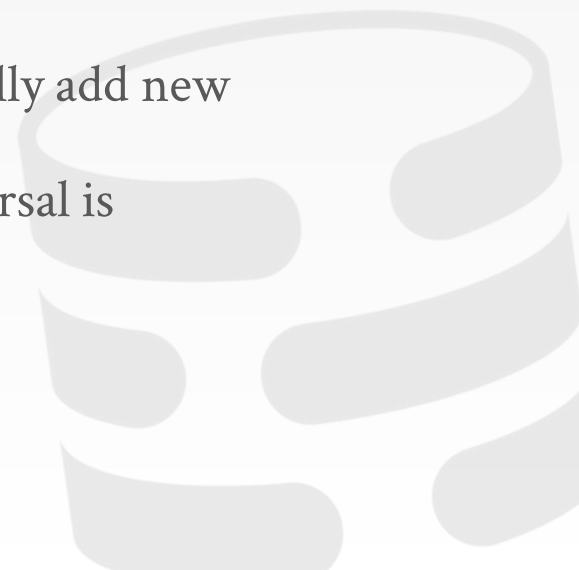
# DYNAMIC PROGRAMMING OPTIMIZER

---

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.



DYNAMIC PROGRAMMING STRIKES BACK  
SIGMOD 2008

# PREDICATE EXPRESSIONS

---

Predicates are defined as part of each operator.

- These are typically represented as an AST.
- Postgres implements them as flatten lists.

The same logical operator can be represented in multiple physical operators using variations of the same expression.

# PREDICATE PUSHDOWN

---

## **Approach #1: Logical Transformation**

- Like any other transformation rule in Cascades.
- Can use cost-model to determine benefit.

## **Approach #2: Rewrite Phase**

- Perform pushdown before starting search using an initial rewrite phase. Tricky to support complex predicates.

## **Approach #3: Late Binding**

- Perform pushdown after generating optimal plan in Cascades. Will likely produce a bad plan.

# PREDICATE MIGRATION

---

Observation: Not all predicates cost the same to evaluate on tuples.

```
SELECT * FROM foo
WHERE foo.id = 1234
      AND SHA_512(foo.val) = '...'
```

The optimizer should consider selectivity and computation cost when determining the evaluation order of predicates.



# PIVOTAL ORCA

---

Standalone Cascades implementation.

- Originally written for Greenplum.
- Extended to support HAWQ.

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

Supports multi-threaded search.

# ORCA – ENGINEERING

---

## 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 on for further debugging.

## Issue #2: Optimizer Accuracy

- Automatically check whether the ordering of the estimate cost of two plans matches their actual execution cost.



# 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.



# MEMSQL OPTIMIZER

---

## Rewriter

- Logical-to-logical transformations with access to the cost-model.

## Enumerator

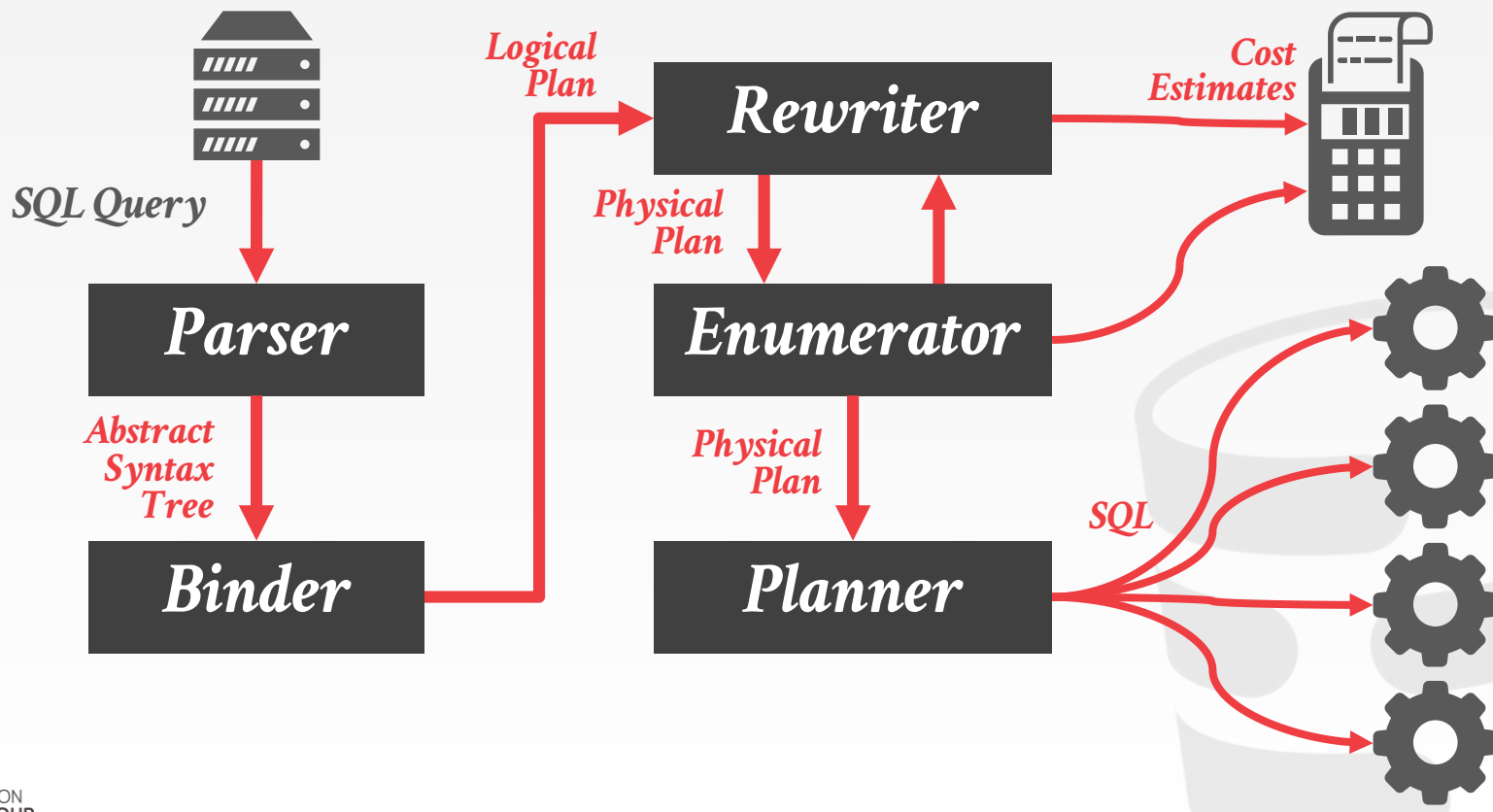
- Logical-to-physical transformations.
- Mostly join ordering.

## Planner

- Convert physical plans back to SQL.
- Contains MemSQL-specific commands for moving data.



# MEMSQL OPTIMIZER OVERVIEW



# PARTING THOUGHTS

---

This is the part of a DBMS that I least understand.  
Let me know if you are interested in exploring this topic more.

All of this relies on a good cost model.  
A good cost model needs good statistics.



# PARTING THOUGHTS

---

This is the part of a DBMS that I least understand.  
Let me know if you are interested in exploring this topic more.

All of this relies on a good cost model.  
A good cost model needs good statistics.



# NEXT CLASS

---

## Cost Models

