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 2nd @ 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→logical and logical→physical transformations.

→ No need for separate stages because everything is transformations.

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

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

THE CASCADES FRAMEWORK FOR QUERY OPTIMIZATION
IEEE DATA ENGINEERING BULLETIN 1995

CASCADES OPTIMIZER

Optimization tasks as data structures.

Rules to place property enforcers.

Ordering of moves by promise.

Predicates as logical/physical operators.

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

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⋈B)⋈C	1. $(A_{Seq} \bowtie_{NL} B_{Seq}) \bowtie_{NL} C_{Seq}$
	2. (B⋈C)⋈A	2. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$
	3. (A⋈C)⋈B	3. $(A_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} B_{Seq}$
	4. A⋈(B⋈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

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

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. $A_{Seq} \bowtie_{NL} (C_{Seq} \bowtie_{NL} B_{Seq})$
	⋮	⋮

Equivalent Expressions

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]⋈[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.

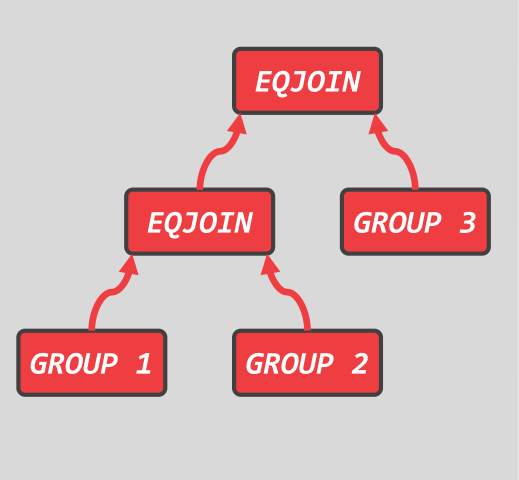
→ **Transformation Rule:** Logical to Logical
→ **Implementation Rule:** Logical to Physical

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



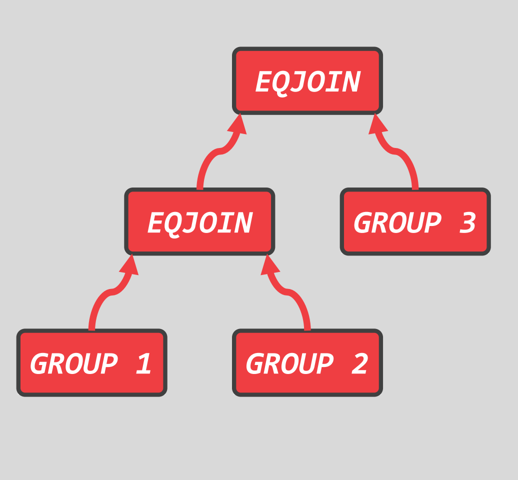
Group

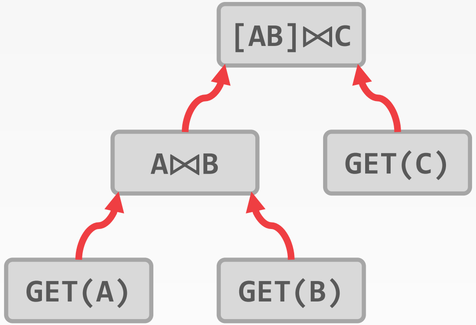
Logical Expr

Physical Expr

CASCADES – RULES

Pattern



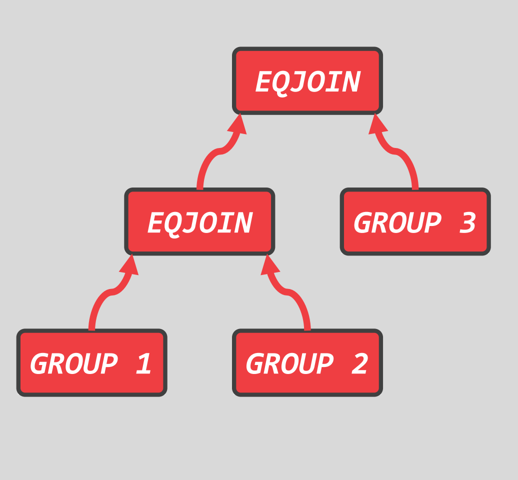


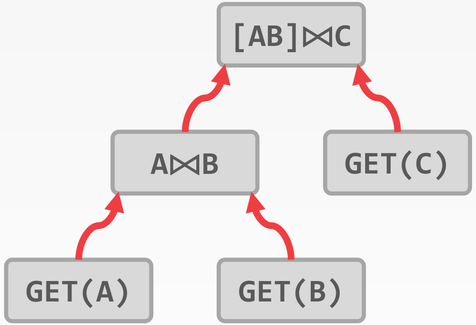
Matching Plan

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

CASCADES – RULES

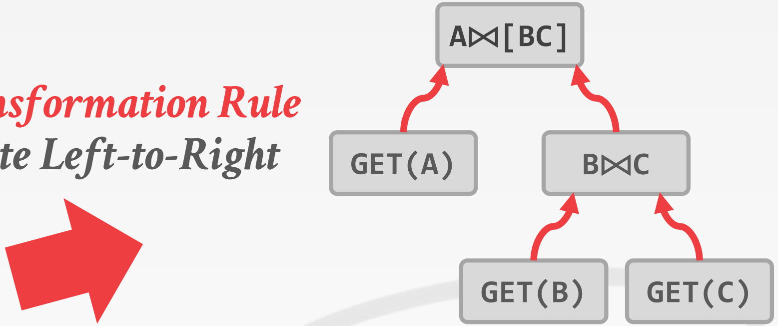
Pattern





Matching Plan

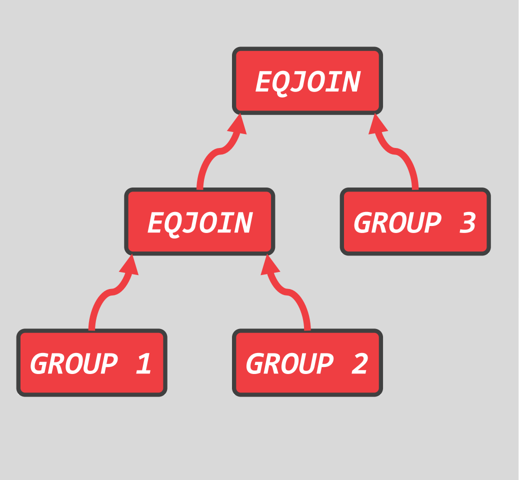
Transformation Rule
Rotate Left-to-Right

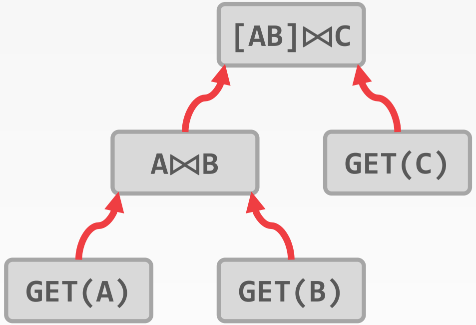


- *Group*
- *Logical Expr*
- *Physical Expr*

CASCADES – RULES

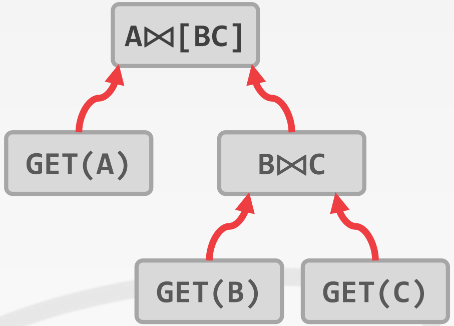
Pattern



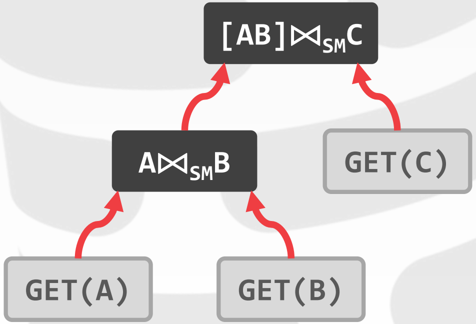


Matching Plan

Transformation Rule
Rotate Left-to-Right



Implementation Rule
EQJOIN→SORTMERGE



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

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.

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

CASCADES – MEMO TABLE

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

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

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

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

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

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

CASCADES – MEMO TABLE

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

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

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

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

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

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

CASCADES – MEMO TABLE

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

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

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

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

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

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

CASCADES – MEMO TABLE

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

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

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

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

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

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

CASCADES – MEMO TABLE

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

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

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

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

Cost: 10

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

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

CASCADES – MEMO TABLE

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

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

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

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

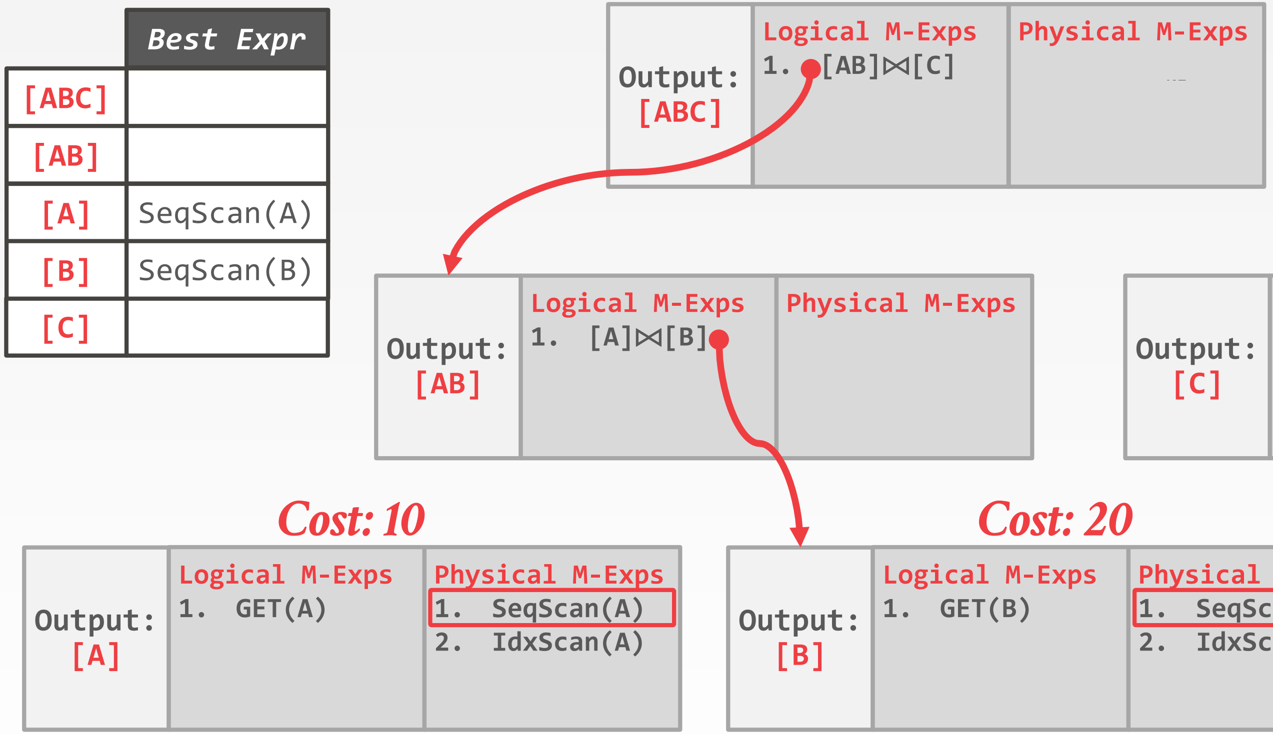
Cost: 10

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

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

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]	

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

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

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

Cost: 10

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

Cost: 20

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

CASCADES – MEMO TABLE

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

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

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

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

Cost: 10

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

Cost: 20

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

Output: [ABC]	Logical M-Exps	Physical M-Exps
	1. $[AB]\bowtie[C]$	

Cost: 50+(10+20)

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

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

Cost: 10

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

Cost: 20

Output: [B]	Logical M-Exps	Physical M-Exps
	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)

Output: [ABC]	Logical M-Exps	Physical M-Exps
	1. [AB]$\bowtie$[C]	

Cost: 50+(10+20)

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

Cost: 5

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

Cost: 10

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

Cost: 20

Output: [B]	Logical M-Exps	Physical M-Exps
	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)

Output	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)

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

Cost: 5

Output	Logical M-Exps	Physical M-Exps
Output: [C]	1. GET(C)	1. SeqScan(C) 2. $\boxed{\text{IdxScan(C)}}$

Cost: 10

Output	Logical M-Exps	Physical M-Exps
Output: [A]	1. GET(A)	1. $\boxed{\text{SeqScan(A)}}$ 2. IdxScan(A)

Cost: 20

Output	Logical M-Exps	Physical M-Exps
Output: [B]	1. GET(B)	1. $\boxed{\text{SeqScan(B)}}$ 2. IdxScan(B)

CASCADES – MEMO TABLE

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

Output: [ABC]

Logical M-Exps
1. $[AB]\bowtie[C]$
2. $[BC]\bowtie[A]$
3. $[AC]\bowtie[B]$
4. $[B]\bowtie[AC]$

Physical M-Exps
1. $[AB]\bowtie_{NL}C$
2. $[BC]\bowtie_{NL}A$
3. $[AC]\bowtie_{NL}B$
⋮

Cost: 50+(10+20)

Output: [AB]

Logical M-Exps
1. $[A]\bowtie[B]$

Physical M-Exps
1. $[A]\bowtie_{NL}[B]$

Cost: 5

Output: [C]

Logical M-Exps
1. GET(C)

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

Output: [BC]

Logical M-Exps
1. $[B]\bowtie[C]$

Physical M-Exps

Output: [AC]

Logical M-Exps
1. $[A]\bowtie[C]$
2. $[C]\bowtie[A]$

Physical M-Exps

Cost: 10

Output: [A]

Logical M-Exps
1. GET(A)

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

Output: [B]

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

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:

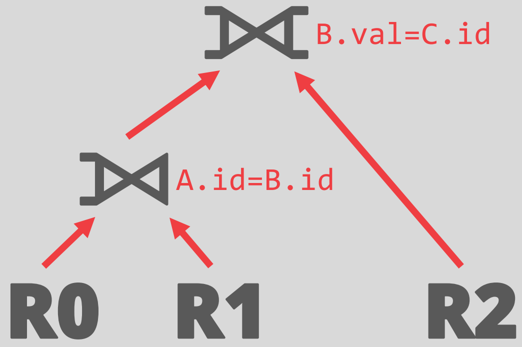
→ Equi/Inner Joins
→ Simple join predicates that reference only two tables.
→ No cross products

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);
```

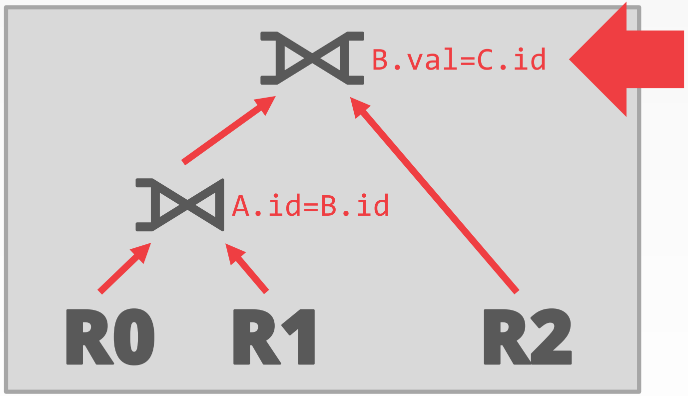


No valid reordering is possible.

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);
```



Source: Pit Fender

No valid reordering is possible.

The **A⟕B** operator is not commutative with **B⟗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

ON THE CORRECT AND COMPLETE ENUMERATION OF THE CORE SEARCH SPACE
SIGMOD 2013

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

SIGMOD 2008

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

VLDB 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.

PREDICATE MIGRATION: OPTIMIZING QUERIES WITH EXPENSIVE PREDICATES
SIGMOD 1993

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: A MODULAR QUERY OPTIMIZER ARCHITECTURE FOR BIG DATA
SIGMOD 2014

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.

APACHE CALCITE: A FOUNDATIONAL FRAMEWORK FOR OPTIMIZED QUERY PROCESSING OVER HETEROGENEOUS DATA SOURCES
SIGMOD 2018

MEMSQL OPTIMIZER

Rewriter

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

Enumerator

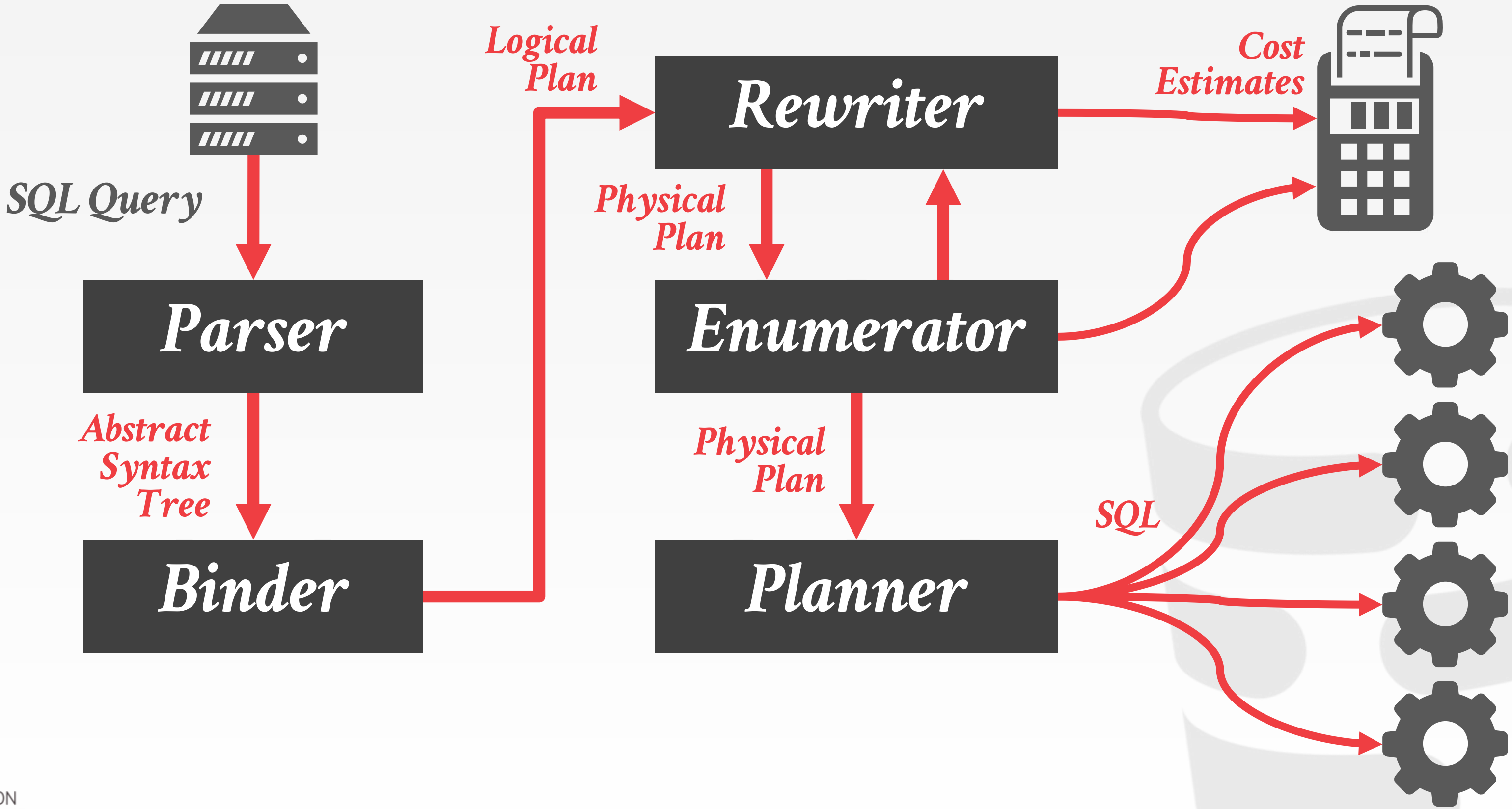
→ Logical-to-physical transformations.
→ Mostly join ordering.

Planner

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

THE MEMSQL QUERY OPTIMIZER
VLDB 2017

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