Query Optimization

Iztok Savnik, FAMNIT

November, 2025.

SOURCES

- Course:
  - Carnegie Mellon University (CMU)
  - Advanced Database Systems (Spring 2024)
  - Prof. Andy Pavlo
  - Transparencies
- Papers:
  - Conferences VLDB, SIGMOD, CIDR, etc.
  - Journals ACM TODS, IS, VLDBJ, etc.

QUERY OPTIMIZATION

For a given query, find a correct physical execution plan for that query with the lowest "cost".

This is the part of a DBMS that is the hardest to implement well (proven to be NP-Complete).

No optimizer truly produces the "optimal" plan

- Use estimation techniques to guess real plan cost.
- Use heuristics to limit the search space.

LOGICAL VS. PHYSICAL PLANS

The optimizer generates a **mapping** of a logical algebra expression to the optimal equivalent physical algebra expression.

Physical operators define a specific execution strategy using an access path.

- They can depend on the physical format of the data that they process (i.e., sorting, compression).
- Not always a 1:1 mapping from logical to physical.

COST ESTIMATION

Generate an estimate of the cost of executing a plan for the current state of the database.

- Interactions with other work in DBMS
- Size of intermediate results
- Choices of algorithms, access methods
- Resource utilization (CPU, I/O, network)
- Data properties (skew, order, placement)

TODAY'S AGENDA

Heuristics

Heuristics + Cost-based Search

Stratified Search

Unified Search

Randomized Search

HEURISTIC-BASED OPTIMIZATION

Define static rules that transform logical operators to a physical plan without a cost model.

- Perform most restrictive selection early
- Perform all selections before joins
- Predicate/Limit/Projection pushdowns
- Join ordering based on simple rules or cardinality estimates

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

LOGICAL QUERY OPTIMIZATION

Split Conjunctive Predicates

Predicate Pushdown

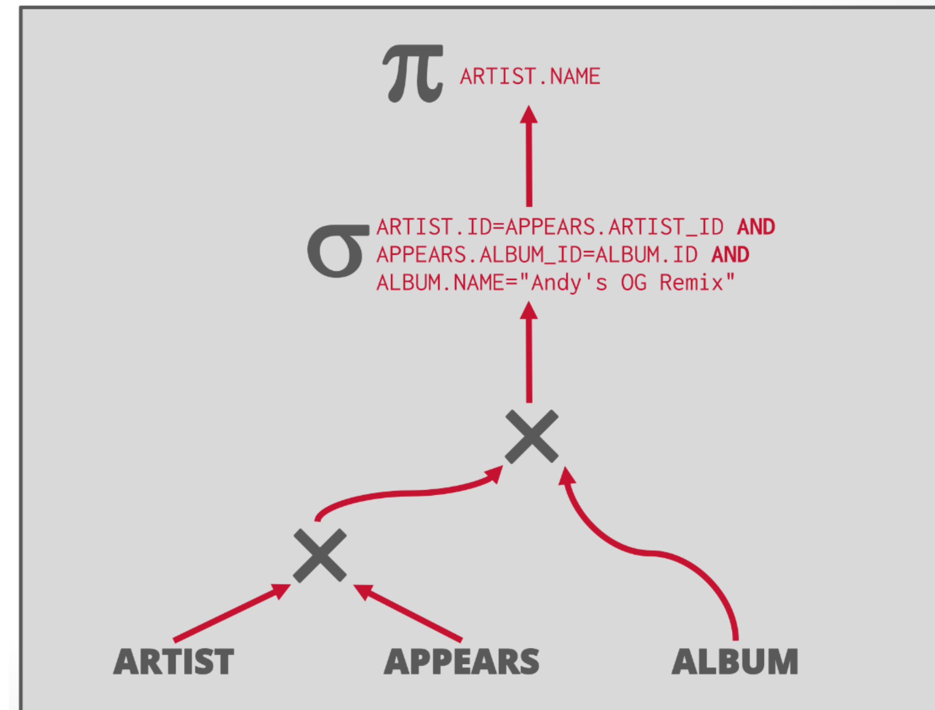
Replace Cartesian Products with Joins

Projection Pushdown

SPLIT CONJUNCTIVE PREDICATES

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

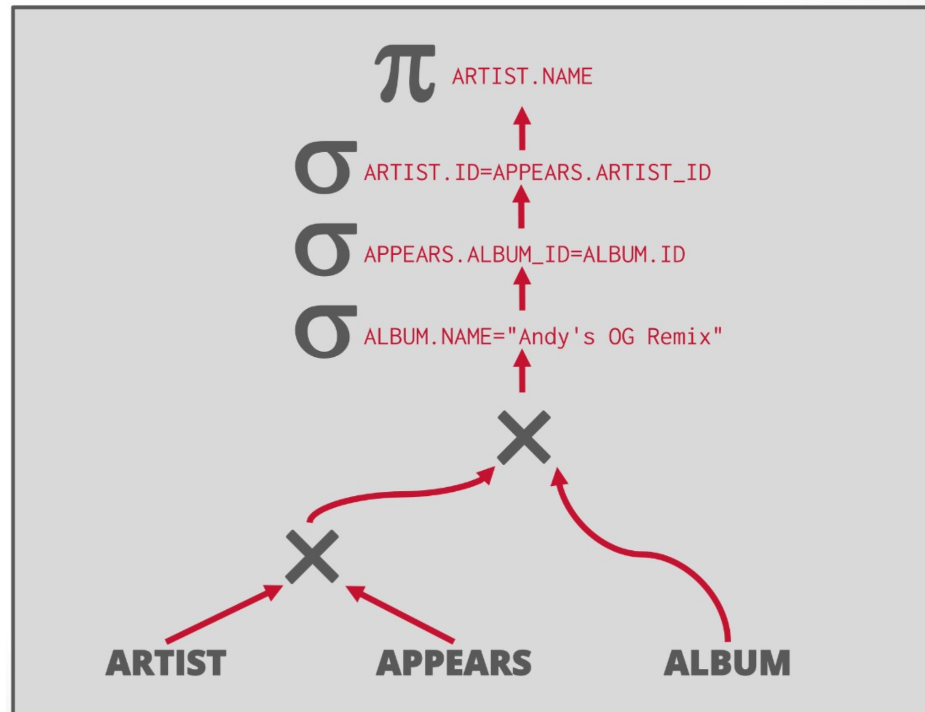
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



SPLIT CONJUNCTIVE PREDICATES

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

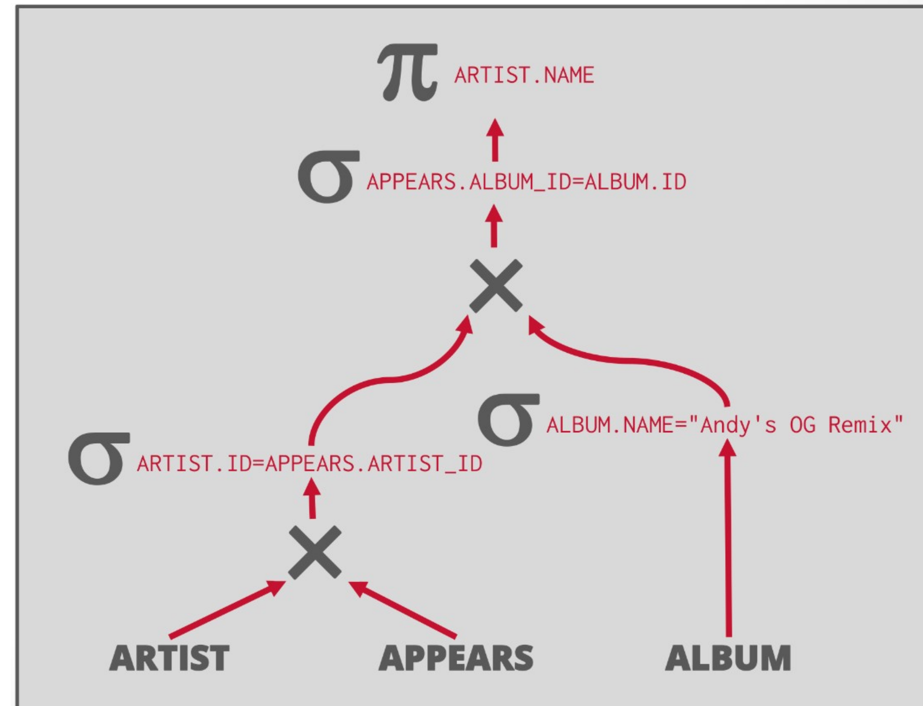
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



PREDICATE PUSHDOWN

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

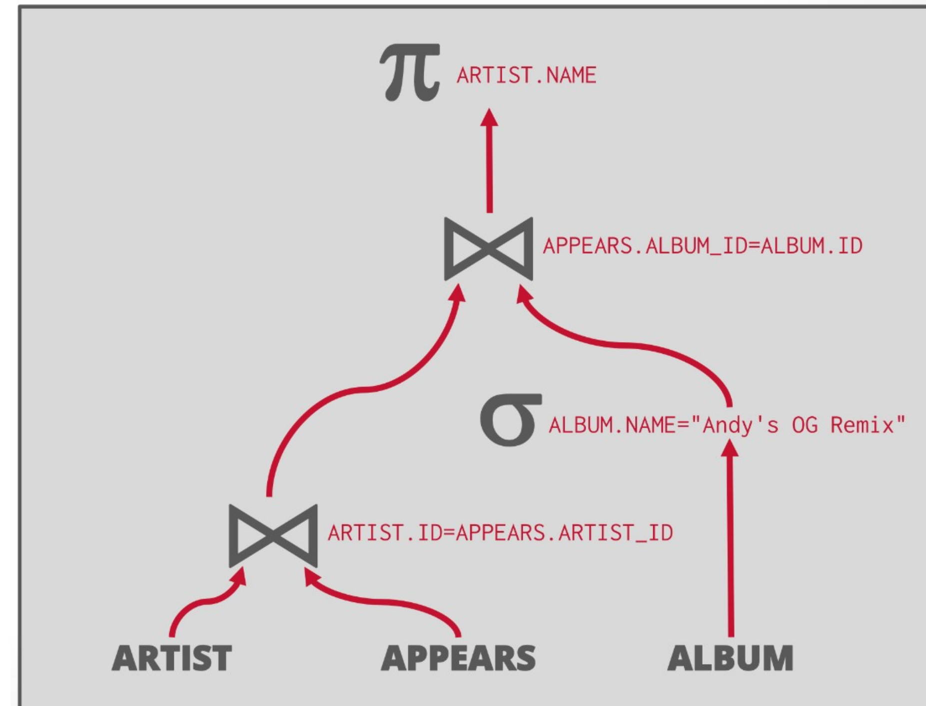
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



REPLACE CARTESIAN PRODUCTS

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

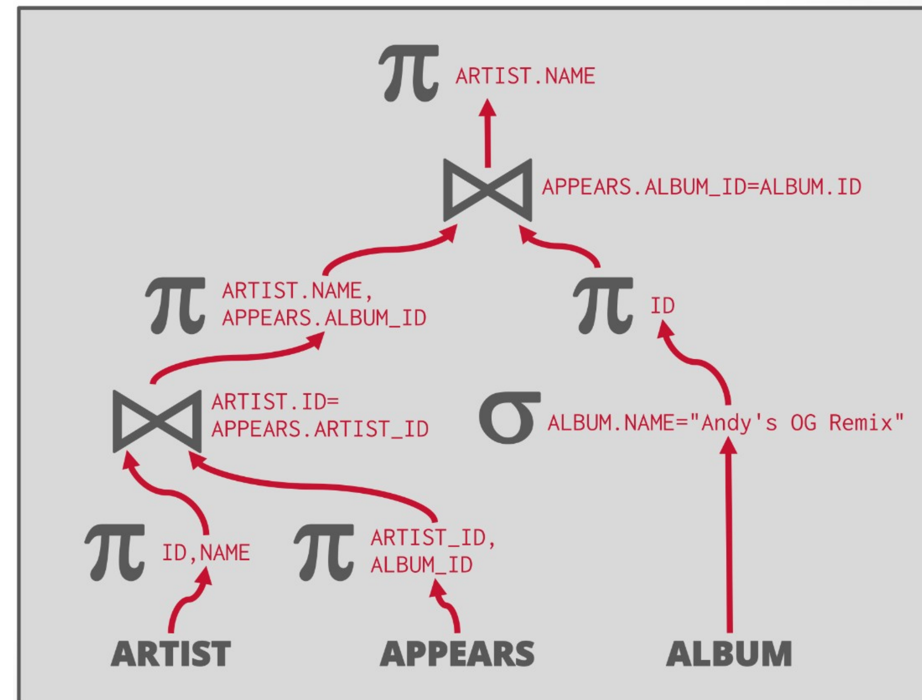
Replace all Cartesian Products with inner joins using the join predicates.



PROJECTION PUSHDOWN

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

Eliminate redundant attributes before pipeline breakers to reduce materialization cost.



INGRES OPTIMIZER

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

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

Query #1

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

Query #2

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, TEMP1
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
 ORDER BY APPEARS.ID
```

Step #1: Decompose into single-value queries

INGRES OPTIMIZER

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

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

Step #1: Decompose into single-value queries

Query #1

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

Query #2

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, TEMP1
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
 ORDER BY APPEARS.ID
```

INGRES OPTIMIZER

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

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

Step #1: Decompose into single-value queries

Query #1

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

Query #3

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

Query #4

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

INGRES OPTIMIZER

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

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

ALBUM_ID
9999

Query #3

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

Query #4

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

Step #1: Decompose into single-value queries

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

INGRES OPTIMIZER

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

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

ALBUM_ID
9999

```
SELECT APPEARS.ARTIST_ID
  FROM APPEARS
 WHERE APPEARS.ALBUM_ID=9999
 ORDER BY APPEARS.ARTIST_ID
```

Step #1: Decompose into single-value queries

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

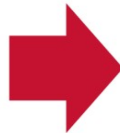
Query #4

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

INGRES OPTIMIZER

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

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



ALBUM_ID
9999

ARTIST_ID
123
456

Step #1: Decompose into single-value queries

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

Query #4

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

INGRES OPTIMIZER

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

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

ALBUM_ID
9999

ARTIST_ID
123
456

```
SELECT ARTIST.NAME
  FROM ARTIST
 WHERE ARTIST.ARTIST_ID=123
```

```
SELECT ARTIST.NAME
  FROM ARTIST
 WHERE ARTIST.ARTIST_ID=456
```

Step #1: Decompose into single-value queries

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

INGRES OPTIMIZER

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

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

ALBUM_ID
9999

ARTIST_ID
123
456

NAME
O.D.B.

NAME
DJ Premier

Step #1: Decompose into single-value queries

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

HEURISTIC-BASED OPTIMIZATION

Advantages:

- Easy to implement and debug.
- Works reasonably well and is fast for simple queries.

Disadvantages:

- Relies on magic constants that predict the efficacy of a planning decision.
- Nearly impossible to generate good plans when operators have complex inter-dependencies.

HEURISTICS + COST-BASED SEARCH

First use static rules to perform initial logical → logical optimizations.

Then enumerate plans using logical → physical transformations to find best plan according to a cost model.

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

PHYSICAL QUERY OPTIMIZATION

Transform a query plan's logical operators into physical operators.

- Add more execution information
- Select indexes / access paths
- Choose operator implementations
- Choose when to materialize (i.e., temp tables).

This stage must support cost model estimates.

PLAN ENUMERATION

Approach #1: Generative / Bottom-Up

- Start with nothing and then iteratively assemble and add building blocks to generate a query plan.
- Examples: System R, Starburst

Approach #2: Transformation / Top-Down

- Start with the outcome that the query wants, and then transform it to equivalent alternative sub-plans to find the optimal plan that gets to that goal.
- Examples: Volcano, Cascades

SYSTEM R OPTIMIZER

Break query up into blocks and generate the logical operators for each block.

For each logical operator, generate a set of physical operators that implement it.

- All combinations of join algorithms and access paths

Then iteratively construct a "left-deep" join tree that minimizes the estimated amount of work to execute the plan.

SYSTEM R OPTIMIZER

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

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

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

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

SYSTEM R OPTIMIZER

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

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

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

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

Step #2: Enumerate all possible join orderings for tables

ARTIST ⋈ APPEARS ⋈ ALBUM
APPEARS ⋈ ALBUM ⋈ ARTIST
ALBUM ⋈ APPEARS ⋈ ARTIST
APPEARS ⋈ ARTIST ⋈ ALBUM
ARTIST × ALBUM ⋈ APPEARS
ALBUM × ARTIST ⋈ APPEARS
⋮ ⋮ ⋮

SYSTEM R OPTIMIZER

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

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

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

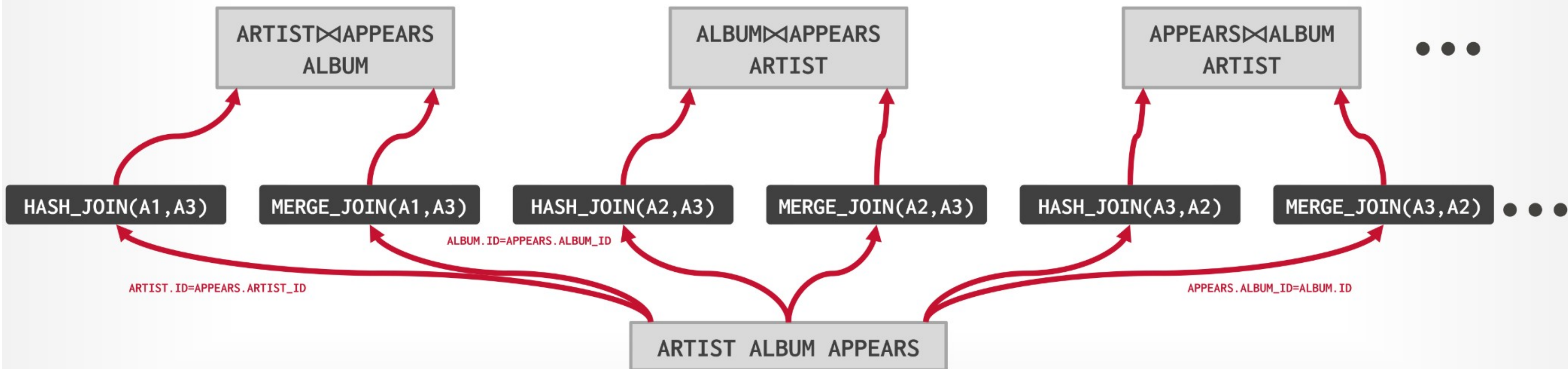
Step #2: Enumerate all possible join orderings for tables

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

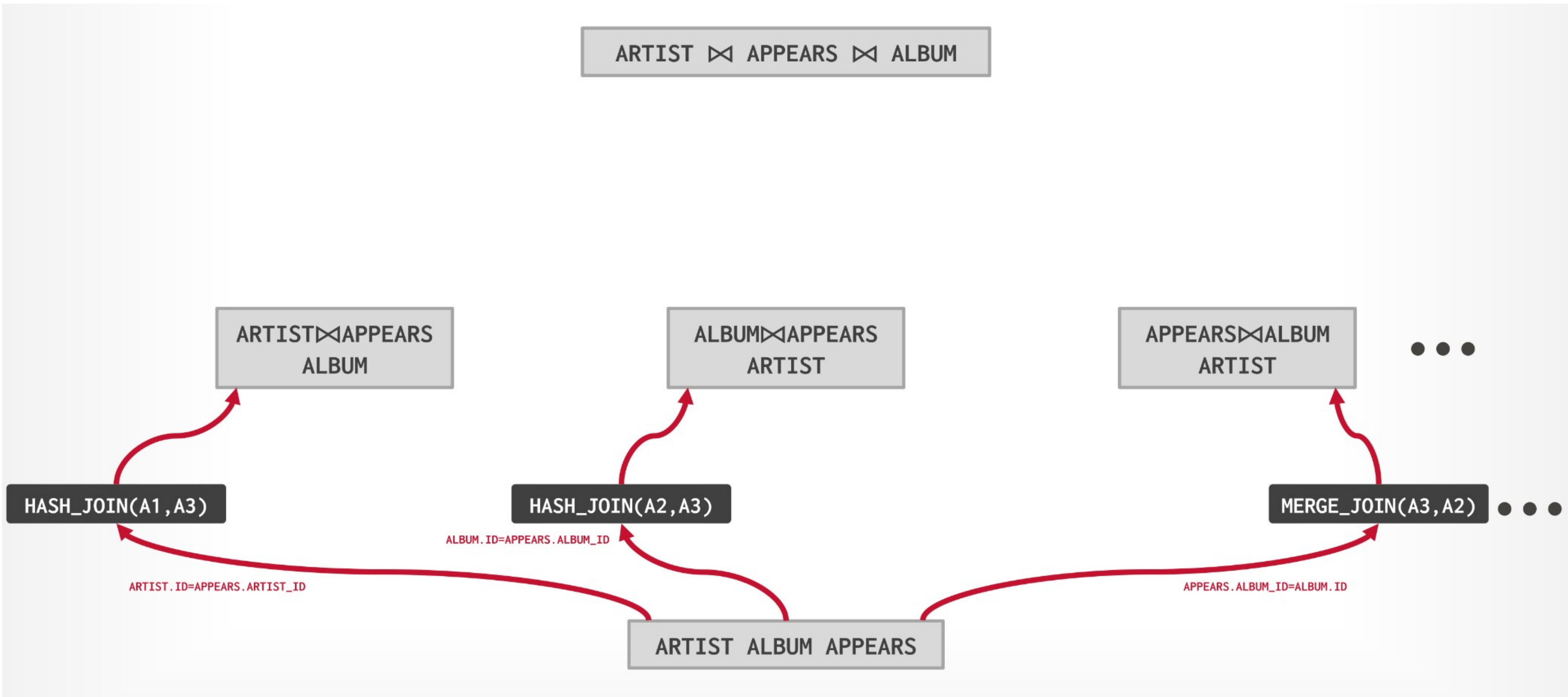
ARTIST: Sequential Scan
APPEARS: Sequential Scan
ALBUM: Index Look-up on **NAME**

ARTIST ⋈ APPEARS ⋈ ALBUM
APPEARS ⋈ ALBUM ⋈ ARTIST
ALBUM ⋈ APPEARS ⋈ ARTIST
APPEARS ⋈ ARTIST ⋈ ALBUM
ARTIST × ALBUM ⋈ APPEARS
ALBUM × ARTIST ⋈ APPEARS
⋮ ⋮ ⋮

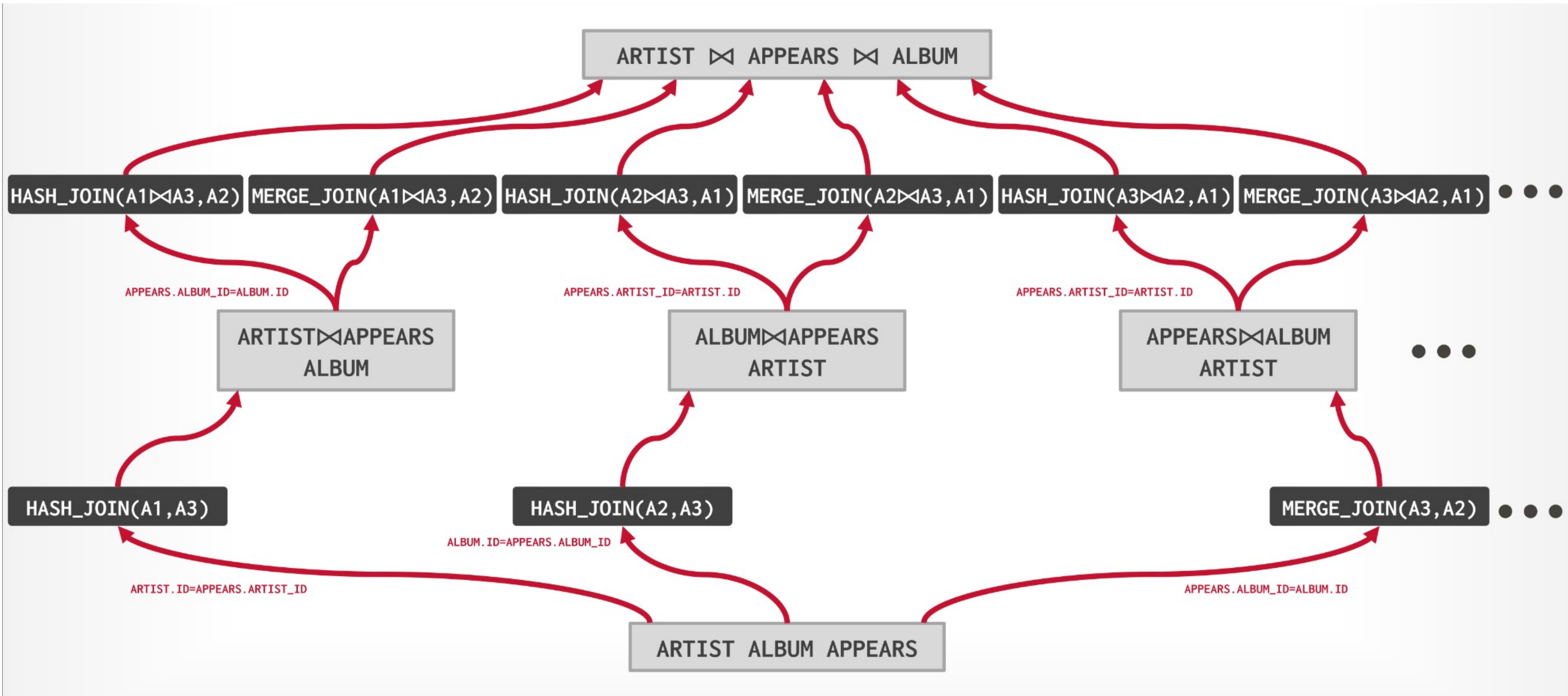
SYSTEM R OPTIMIZER



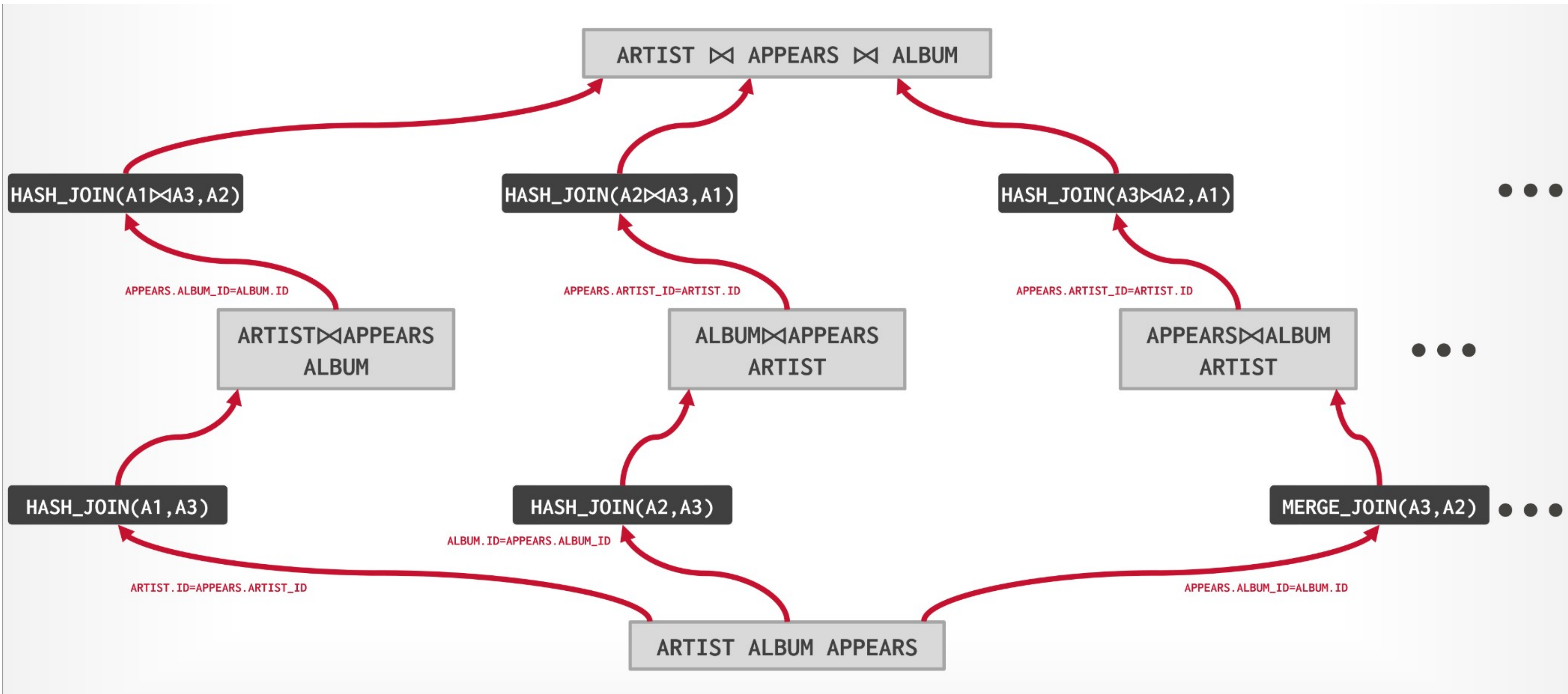
SYSTEM R OPTIMIZER



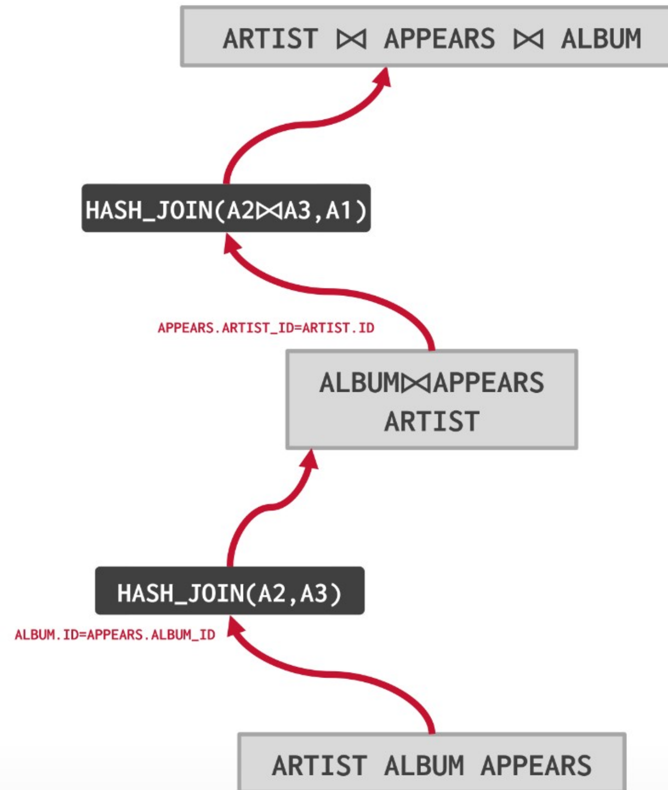
SYSTEM R OPTIMIZER



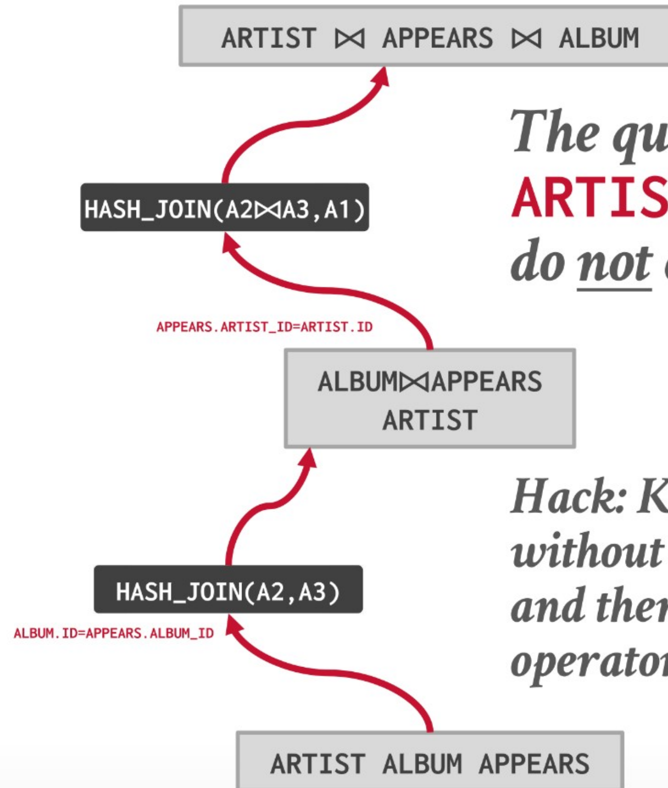
SYSTEM R OPTIMIZER



SYSTEM R OPTIMIZER



SYSTEM R OPTIMIZER



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

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

SEARCH TERMINATION

Approach #1: Wall-clock Time

– Stop after the optimizer runs for some length of time.

Approach #2: Cost Threshold

– Stop when the optimizer finds a plan that has a lower cost than some threshold.

Approach #3: Exhaustion

– Stop when there are no more enumerations of the target plan. Usually done per sub-plan/group.

Approach #4: Transformation Count

– Stop after a certain number of transformations have been considered.

HEURISTICS + COST-BASED SEARCH

Advantages:

- Usually finds a reasonable plan without having to perform an exhaustive search.

Disadvantages:

- All the same problems as the heuristic-only approach.
- Left-deep join trees are not always optimal.
- Must take in consideration the physical properties of data in the cost model (e.g., sort order).

OBSERVATION

Writing query transformation rules in a procedural language is hard and error-prone.

- No easy way to verify that the rules are correct without running a lot of fuzz tests.
- Generation of physical operators per logical operator is decoupled from deeper semantics about query.

A better approach is to use a declarative DSL to write the transformation rules and then have the optimizer enforce them during planning.

OPTIMIZER GENERATORS

Choice #1: **Stratified Search**

- Planning is done in multiple stages (heuristics then cost-based search).
- Examples: Starburst, CockroachDB

Choice #2: **Unified Search**

- Perform query planning all at once.
- Examples: Cascades, OPT++, SQL Server

STRATIFIED SEARCH

First rewrite the logical query plan using transformation rules.

- The engine checks whether the transformation is allowed before it can be applied.
- Cost is never considered in this step.

Then perform a cost-based search to map the logical plan to a physical plan.

STARBURST OPTIMIZER

Better implementation of the System R optimizer that uses declarative rules.

Stage #1: Query Rewrite

- Compute a SQL-block-level, relational calculus-like representation of queries.

Stage #2: Plan Optimization

- Execute a System R-style (bottoms-up) dynamic programming phase once query rewrite has completed.

Example: Latest version of IBM DB2

STARBURST OPTIMIZER

Advantages:

- Works well in practice with fast performance.

Disadvantages:

- Difficult to assign priorities to transformations
- Some transformations are difficult to assess without computing multiple cost estimations.
- Rules maintenance is a huge pain because they are written in IBM's Query Graph Model (QGM) DSL.

UNIFIED SEARCH

Unify the notion of both logical → logical and logical → physical transformations.

- No need for separate stages because everything is transformations.

This approach generates many transformations, so it makes heavy use of memoization to reduce redundant work.

VOLCANO OPTIMIZER

General purpose cost-based query optimizer, based on equivalence rules on algebras.

- Easily add new operations and equivalence rules.
- Treats physical properties of data as first-class entities during planning.
- **Top-down approach** (backward chaining) using branch- and-bound search.

Example: Academic prototypes

TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

ARTIST ⋈ APPEARS ⋈ ALBUM
ORDER-BY(ARTIST.ID)

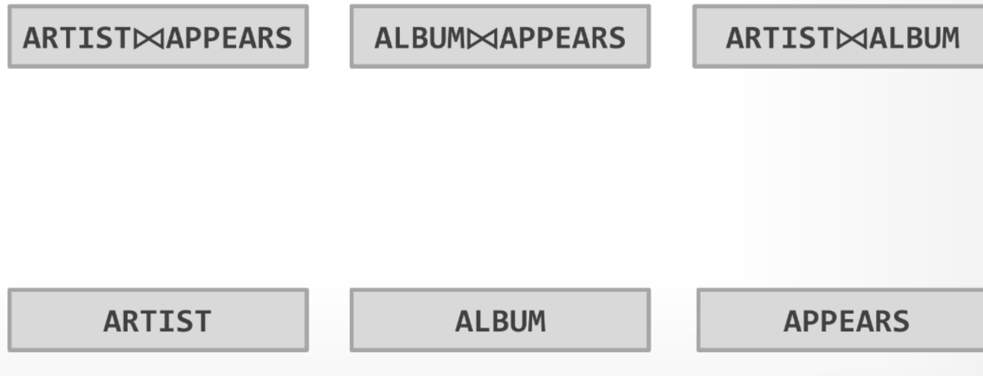
TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical: JOIN(A,B) to JOIN(B,A)
- Logical → Physical: JOIN(A,B) to HASH_JOIN(A,B)

ARTIST ⋈ APPEARS ⋈ ALBUM
ORDER-BY(ARTIST.ID)

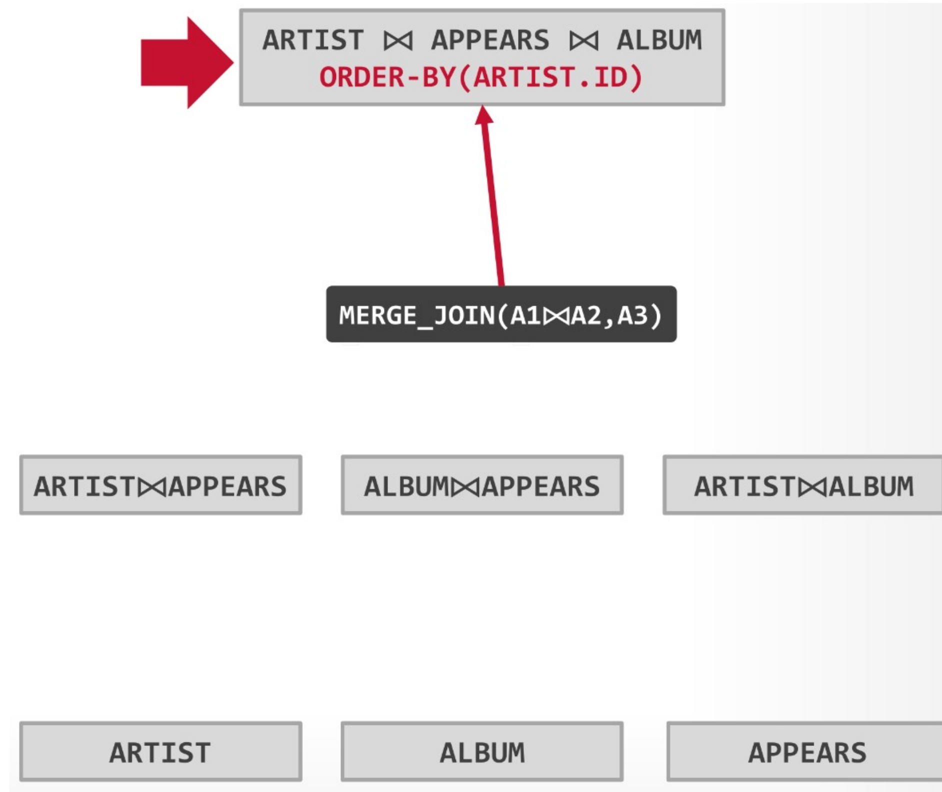


TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical→Logical:
  JOIN(A,B) to JOIN(B,A)
- Logical→Physical:
  JOIN(A,B) to HASH_JOIN(A,B)

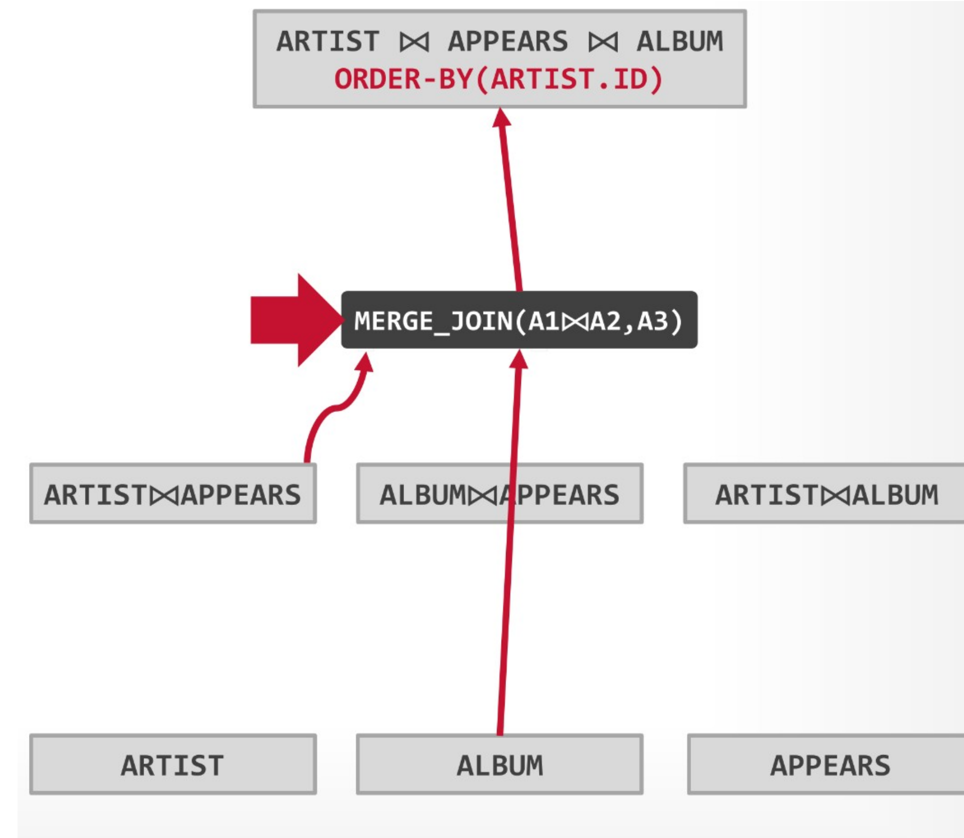


TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical:
  JOIN(A,B) to JOIN(B,A)
- Logical → Physical:
  JOIN(A,B) to HASH_JOIN(A,B)

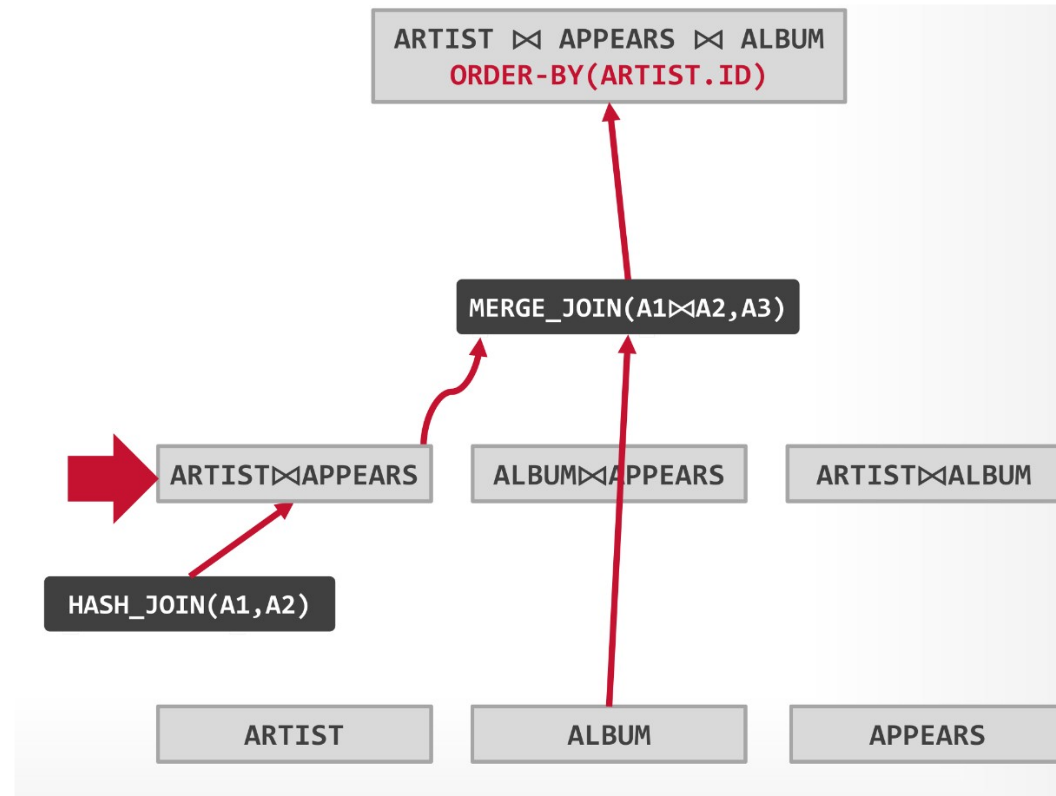


TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical:
  JOIN(A,B) to JOIN(B,A)
- Logical → Physical:
  JOIN(A,B) to HASH_JOIN(A,B)

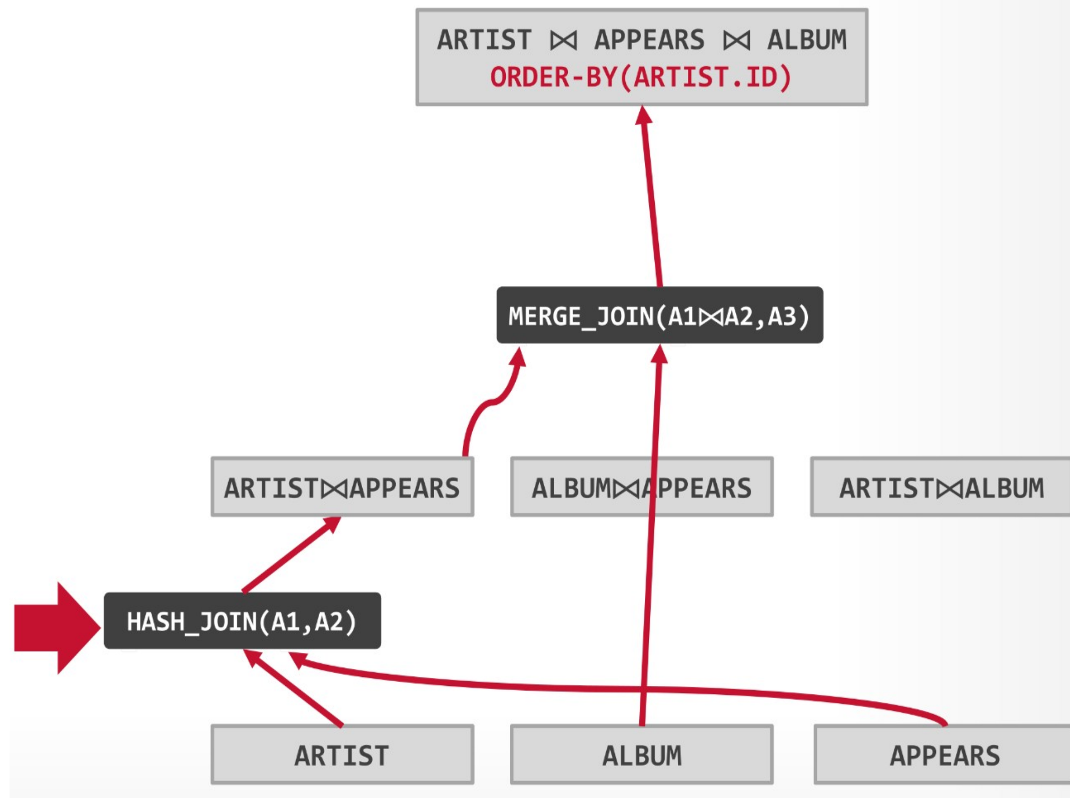


TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical: JOIN(A,B) to JOIN(B,A)
- Logical → Physical: JOIN(A,B) to HASH_JOIN(A,B)

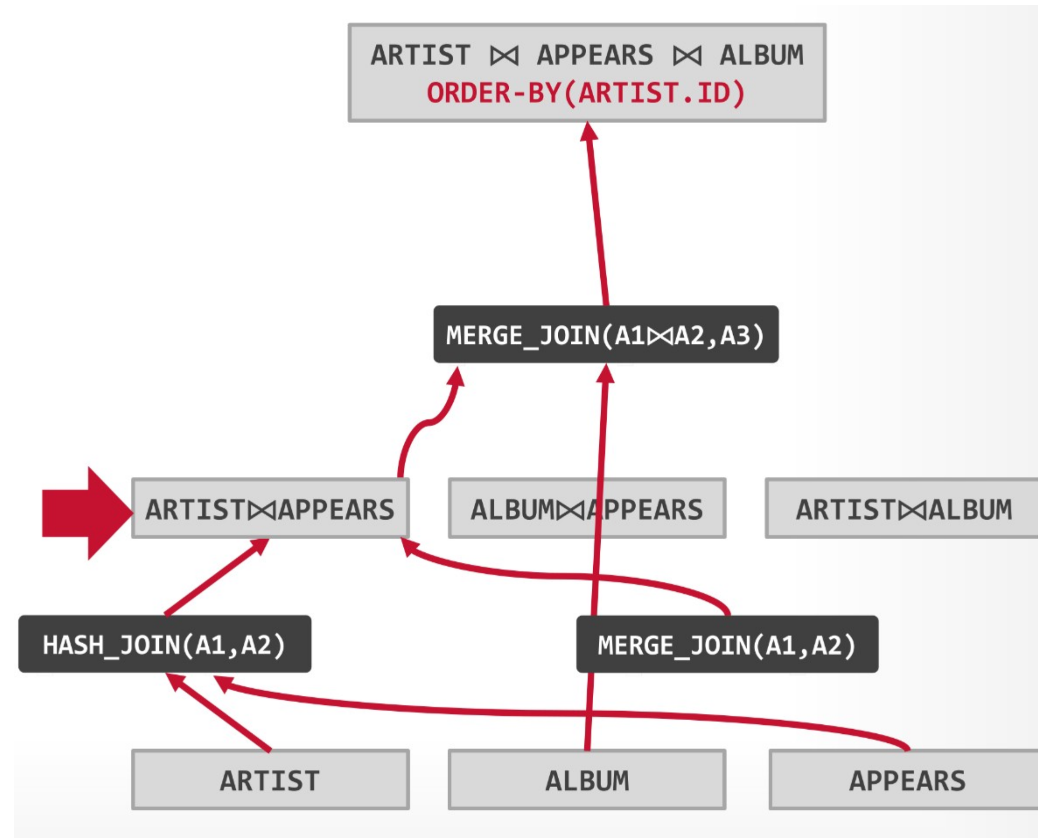


TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical: JOIN(A,B) to JOIN(B,A)
- Logical → Physical: JOIN(A,B) to HASH_JOIN(A,B)

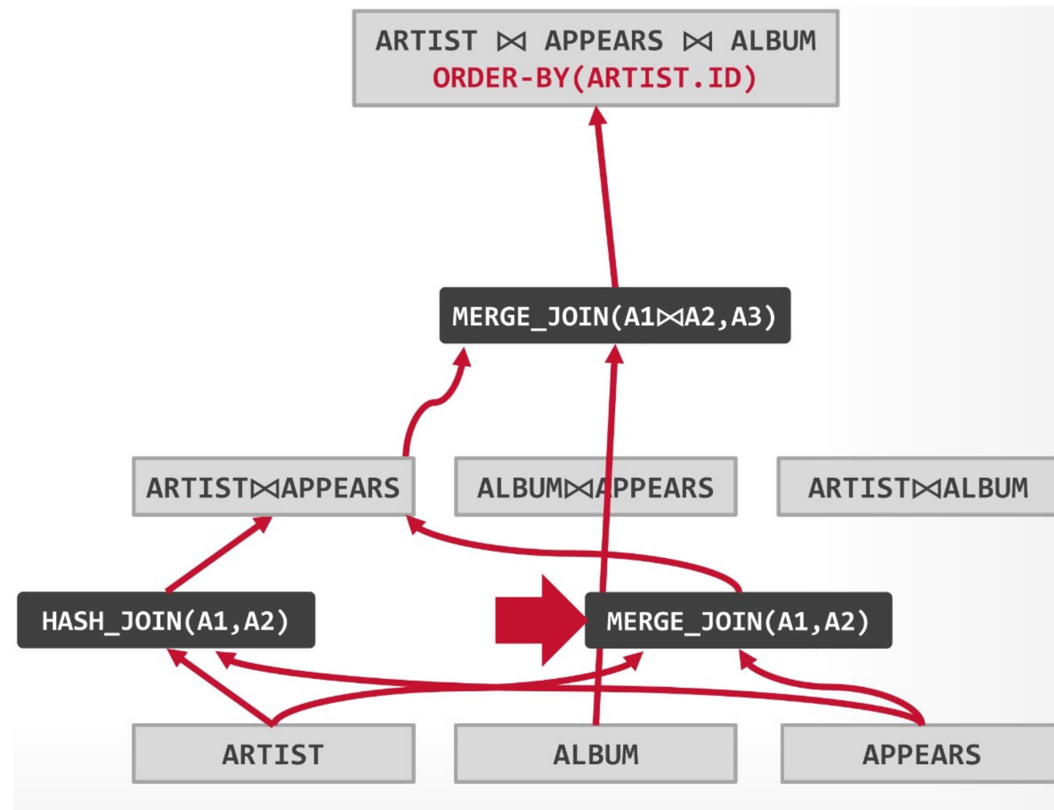


TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical: JOIN(A,B) to JOIN(B,A)
- Logical → Physical: JOIN(A,B) to HASH_JOIN(A,B)



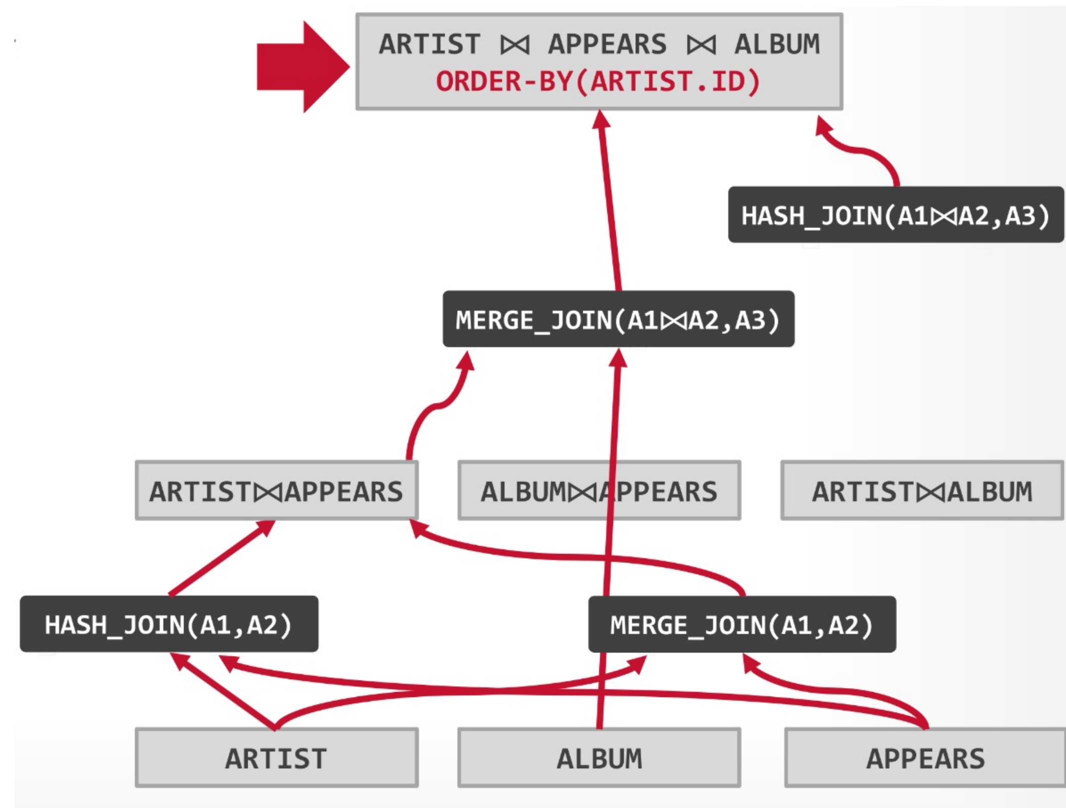
TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical:
  JOIN(A,B) to JOIN(B,A)
- Logical → Physical:
  JOIN(A,B) to HASH_JOIN(A,B)

Can create "enforcer" rules that require input to have certain properties.



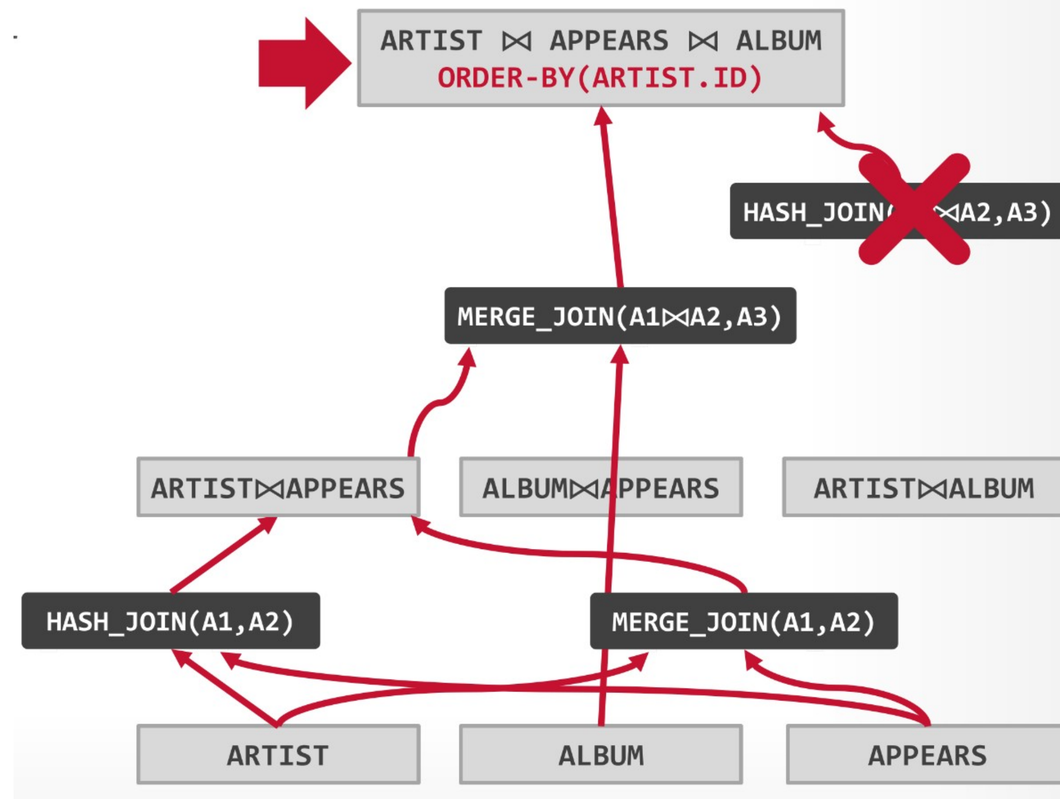
TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical: JOIN(A,B) to JOIN(B,A)
- Logical → Physical: JOIN(A,B) to HASH_JOIN(A,B)

Can create "enforcer" rules that require input to have certain properties.



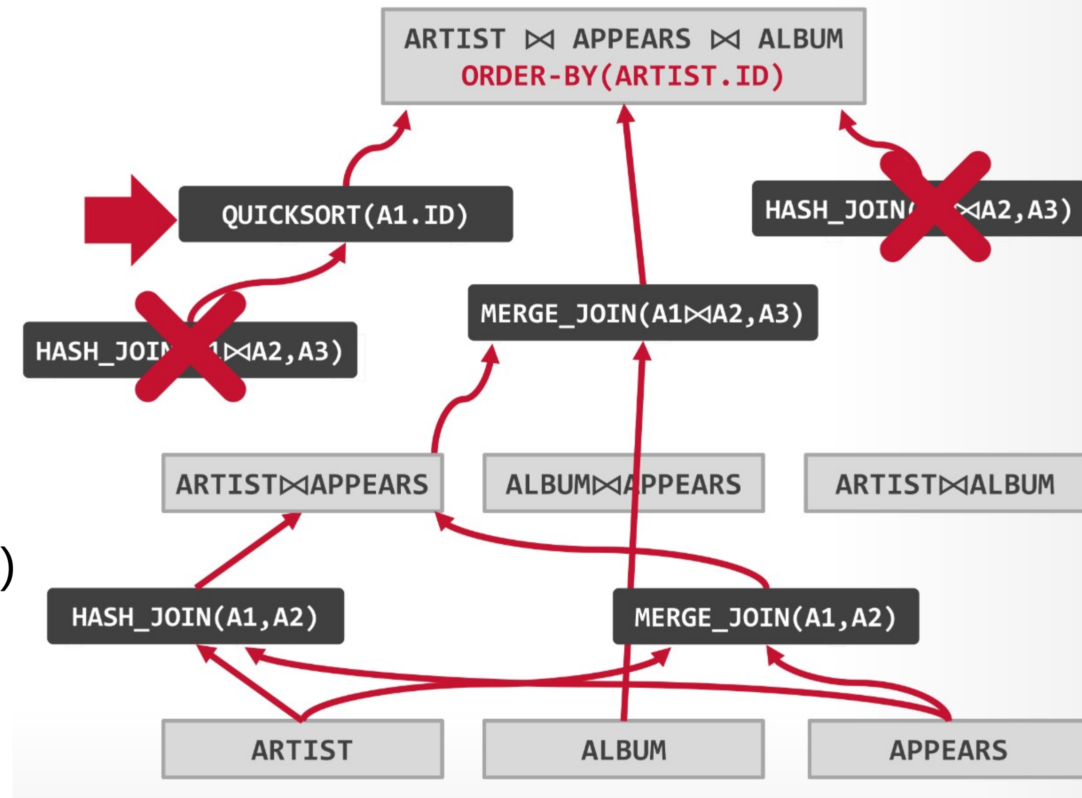
TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

- Logical → Logical: JOIN(A,B) to JOIN(B,A)
- Logical → Physical: JOIN(A,B) to HASH_JOIN(A,B)

Can create "enforcer" rules that require input to have certain properties.



VOLCANO OPTIMIZER

Advantages:

- Use declarative rules to generate transformations.
- Better extensibility with an efficient search engine. Reduce redundant estimations using memoization.

Disadvantages:

- All equivalence classes are completely expanded to generate all possible logical operators before the optimization search.
- Not easy to modify predicates.

CASCADES OPTIMIZER

Object-oriented implementation of the previous Volcano query optimizer.

- Top-down approach (backward chaining) using branchand-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.

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

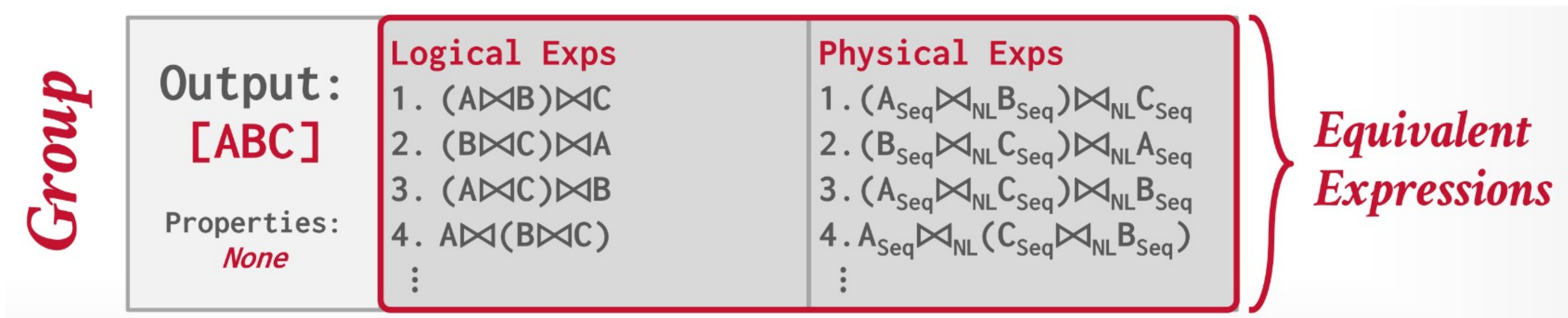
Group

Output: [ABC]	Logical Exps	Physical Exps
Properties: *None*	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.



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] Properties: *None*	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.

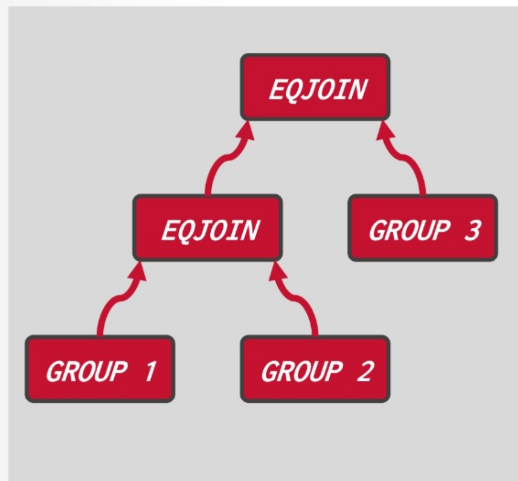
- Transformation Rule: Logical to Logical
- Implementation Rule: Logical to Physical

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

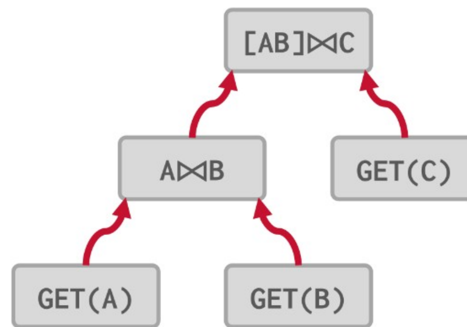
Pattern



Group

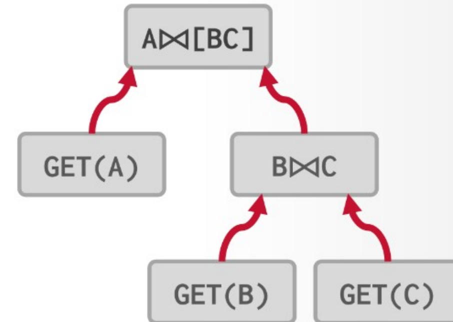
Logical Expr

Physical Expr

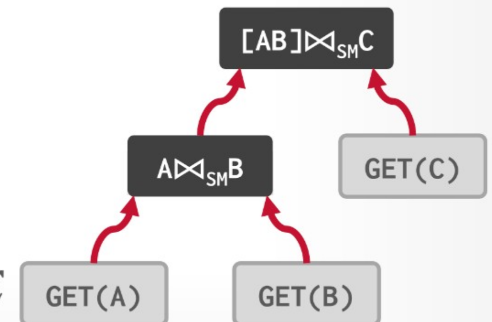


Matching Plan

Transformation Rule
Rotate Left-to-Right



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

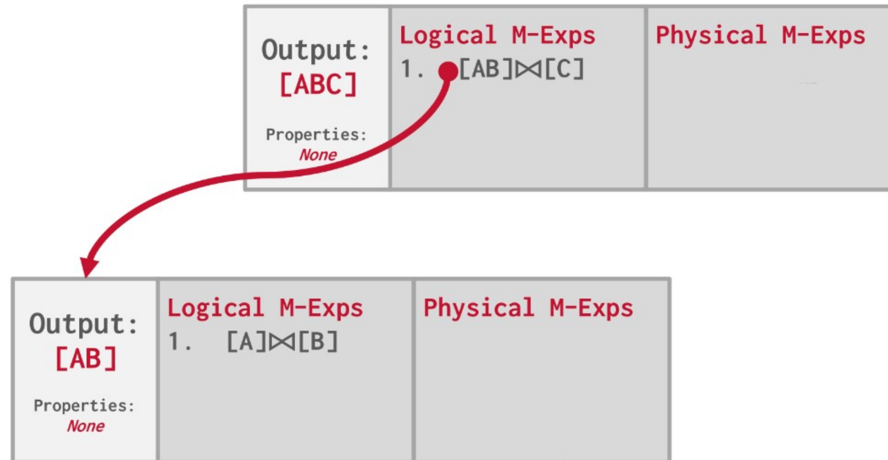
CASCADES: MEMO TABLE

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

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

CASCADES: MEMO TABLE

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



CASCADES: MEMO TABLE

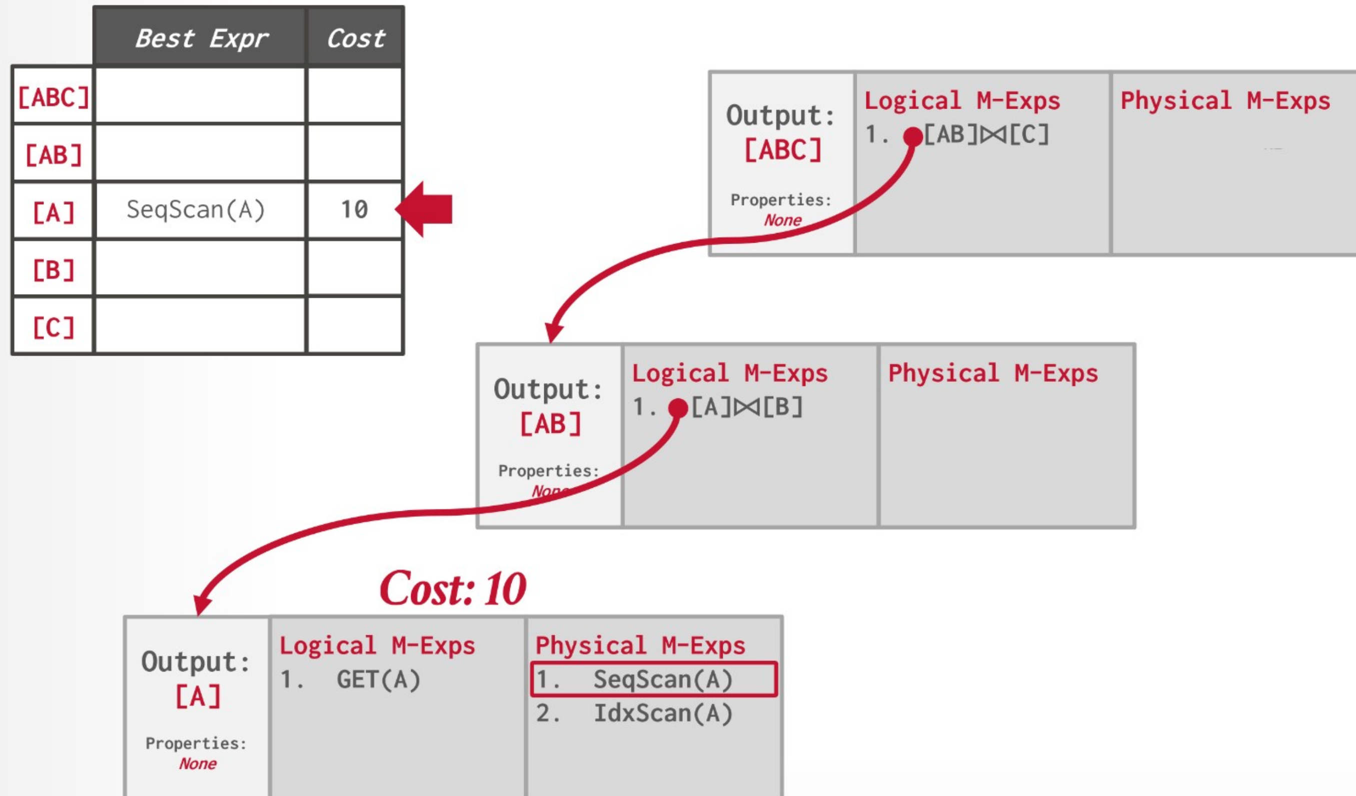
	Best Expr	Cost
[ABC]		
[AB]		
[A]		
[B]		
[C]		

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

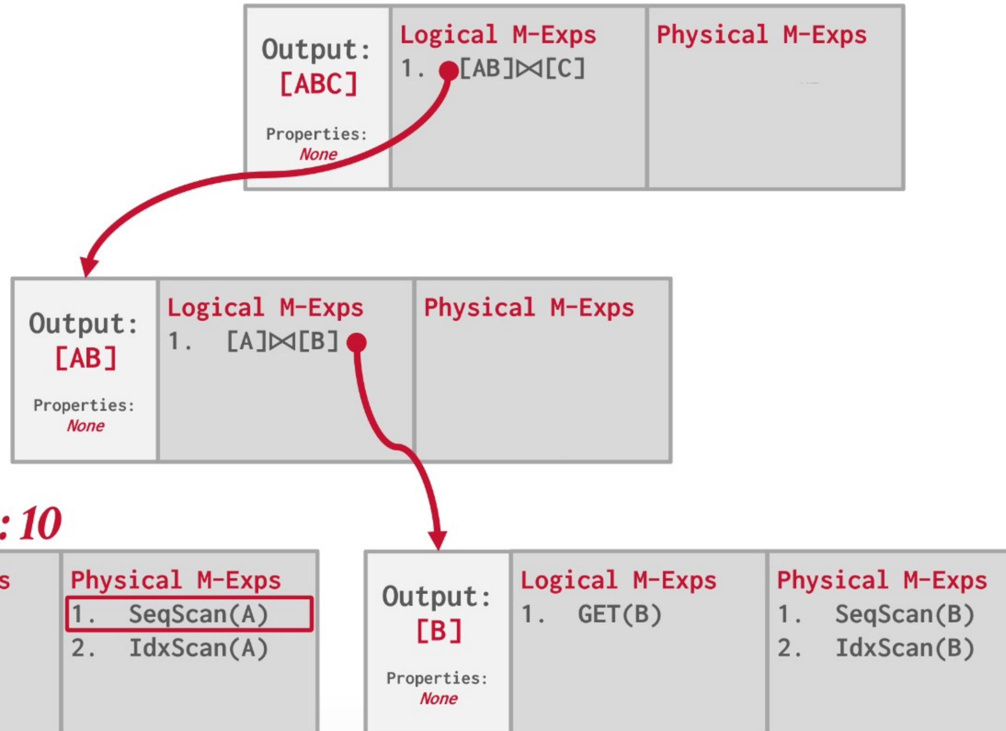
Output: [AB] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. [A]⋈[B]	

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

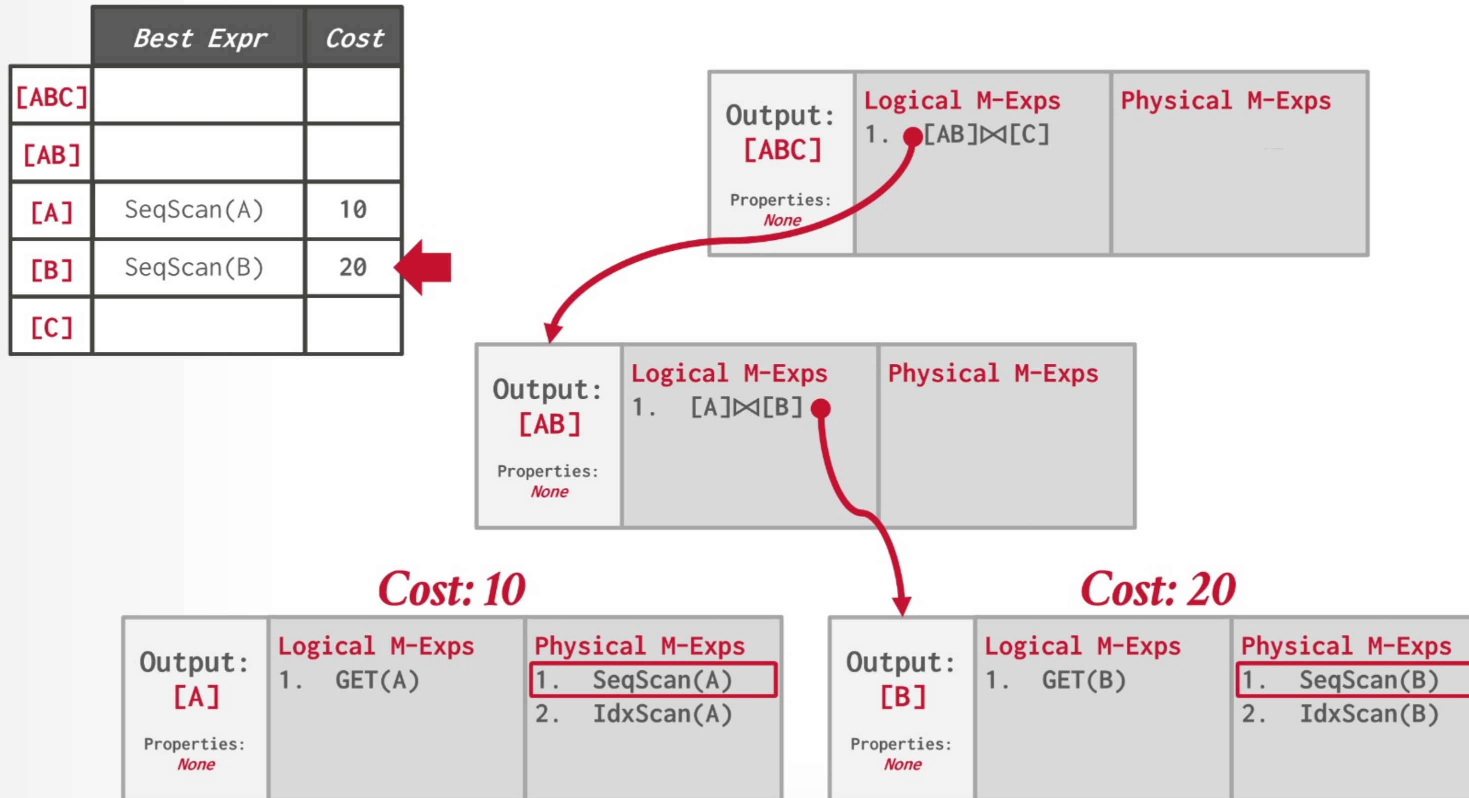
CASCADES: MEMO TABLE



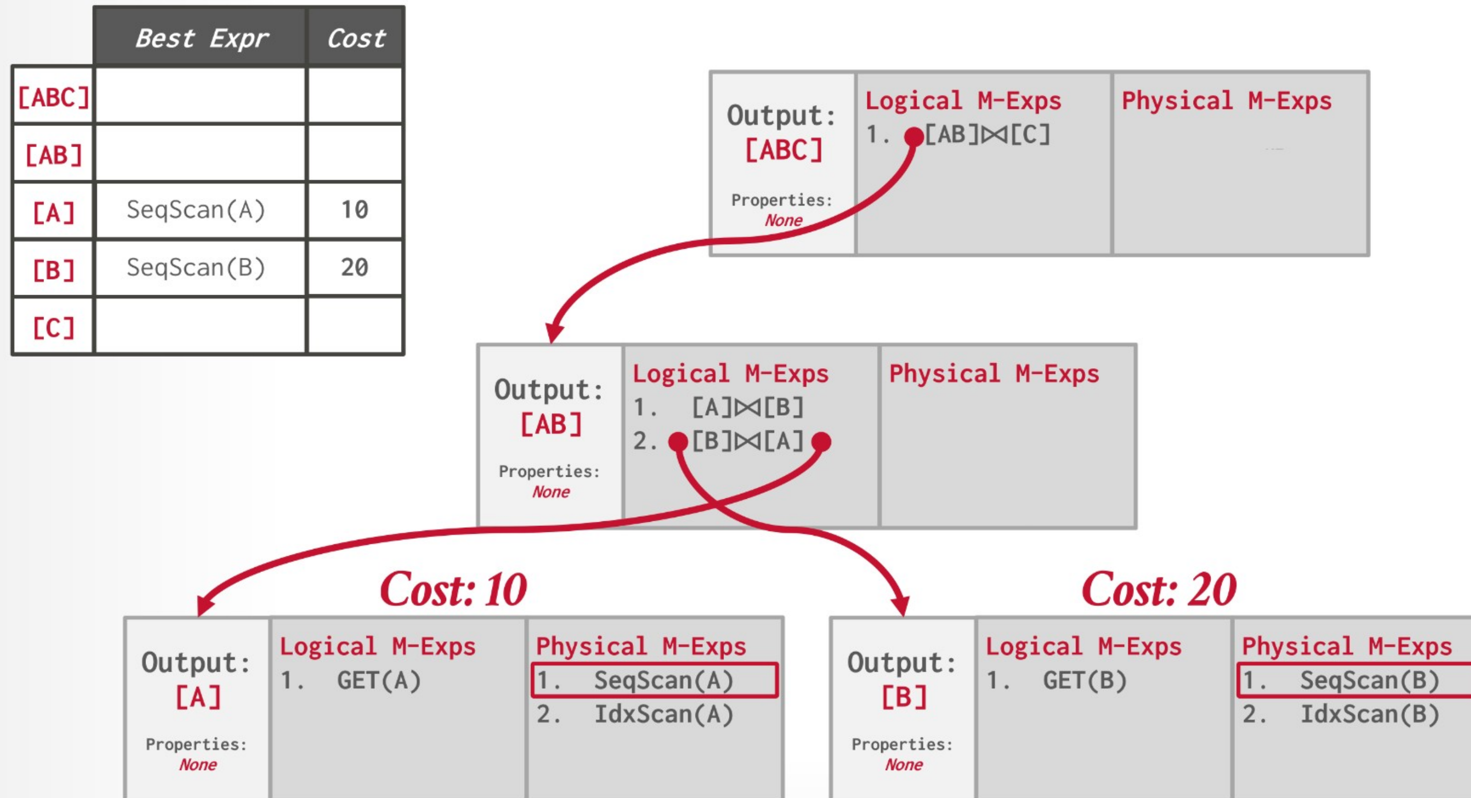
CASCADES: MEMO TABLE



CASCADES: MEMO TABLE

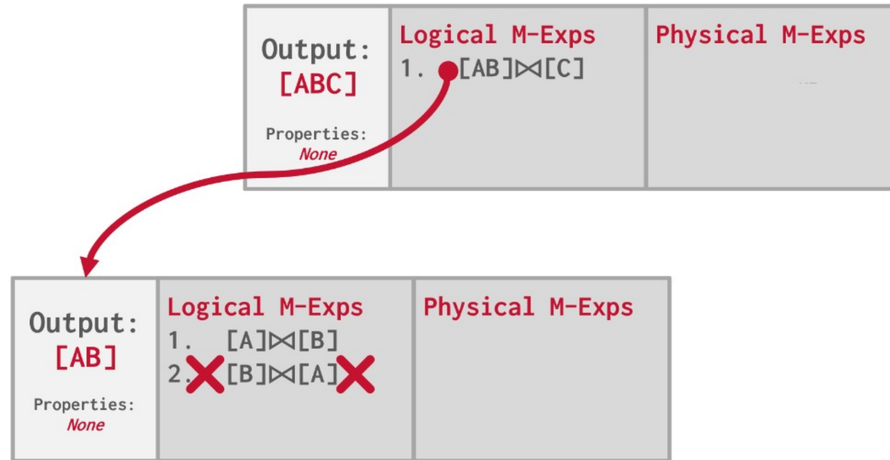


CASCADES: MEMO TABLE



CASCADES: MEMO TABLE

	Best Expr	Cost
[ABC]		
[AB]		
[A]	SeqScan(A)	10
[B]	SeqScan(B)	20
[C]		



Cost: 10

Output: [A]
Properties: None

Logical M-Exps
1. GET(A)

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

Cost: 20

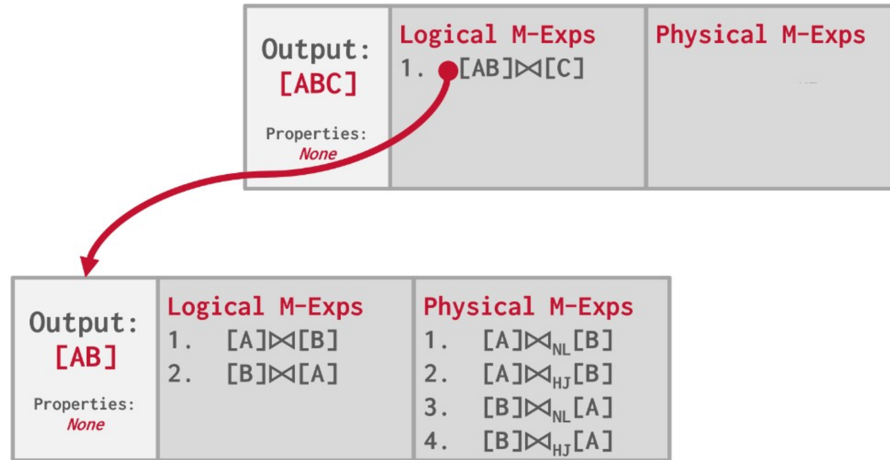
Output: [B]
Properties: None

Logical M-Exps
1. GET(B)

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

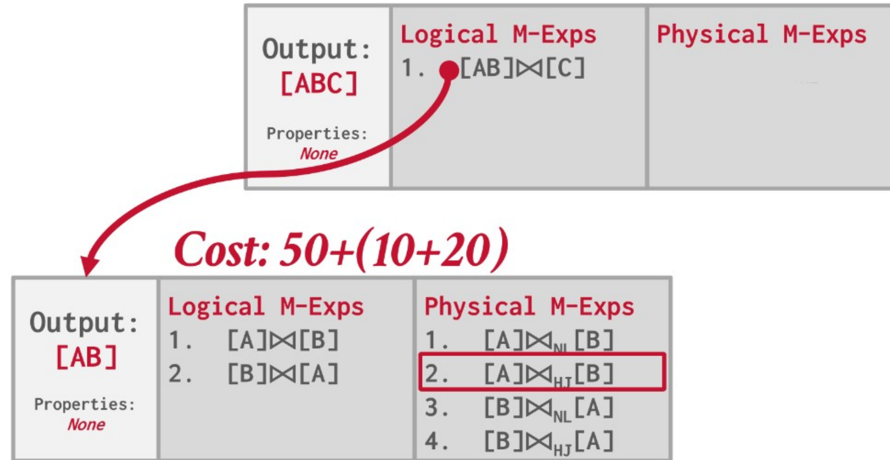
CASCADES: MEMO TABLE

	Best Expr	Cost
[ABC]		
[AB]		
[A]	SeqScan(A)	10
[B]	SeqScan(B)	20
[C]		



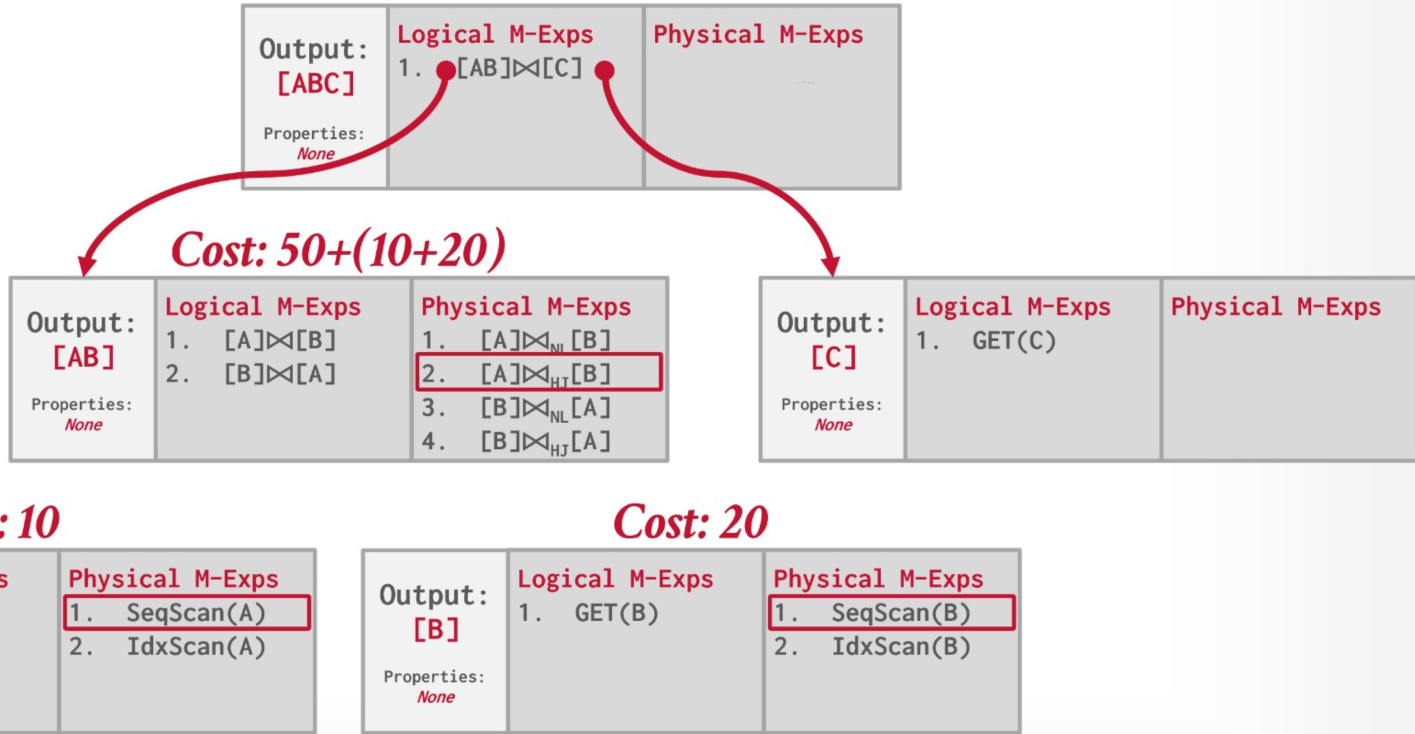
CASCADES: MEMO TABLE

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



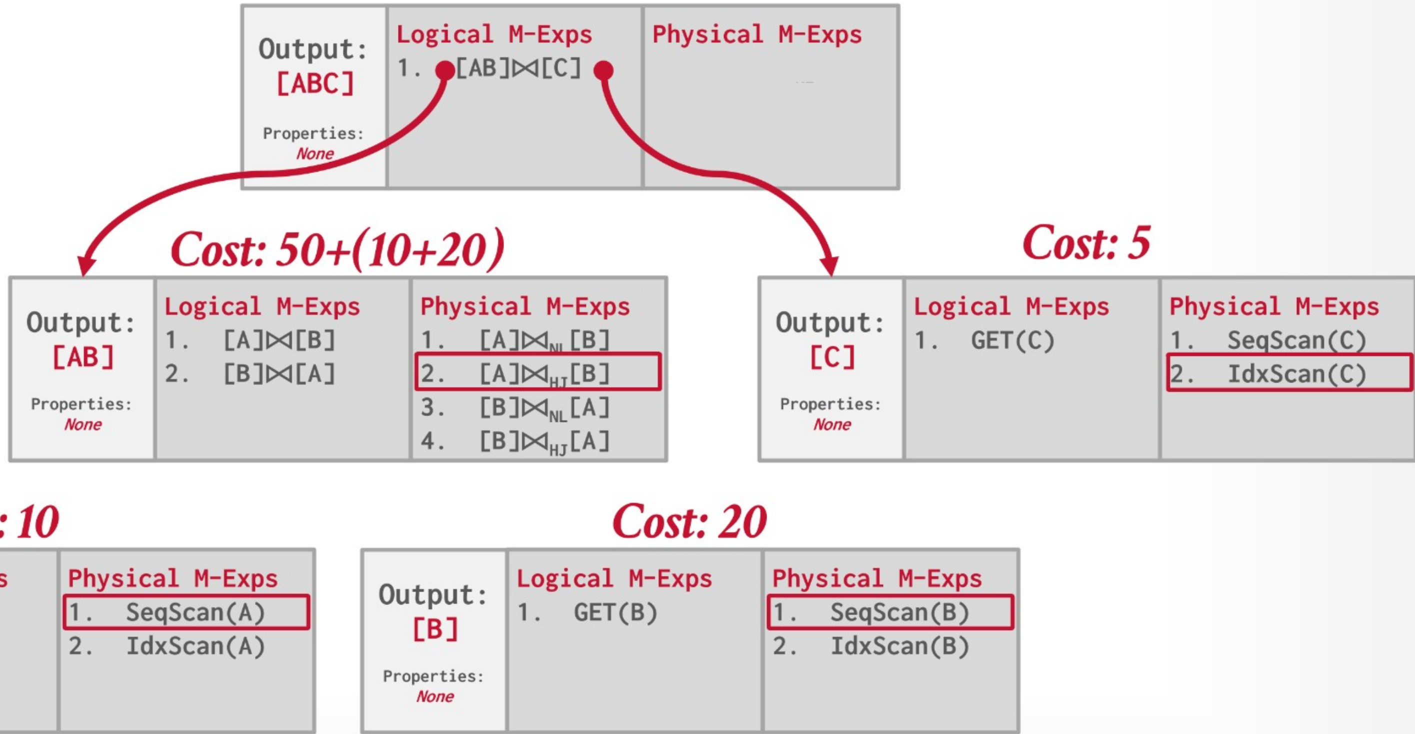
CASCADES: MEMO TABLE

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



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



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 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]
Properties: *None*

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

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

Cost: 5

Output: [C]
Properties: *None*

Logical M-Exps
1. GET(C)

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

Cost: 10

Output: [A]
Properties: *None*

Logical M-Exps
1. GET(A)

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

Cost: 20

Output: [B]
Properties: *None*

Logical M-Exps
1. GET(B)

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

CASCADES: MEMO TABLE

	Best Expr	Cost
[ABC]	([A]⋈$_{HJ}$[B])⋈$_{HJ}$[C]	125
[AB]	[A]⋈$_{HJ}$[B]	80
[A]	SeqScan(A)	10
[B]	SeqScan(B)	20
[C]	IdxScan(C)	5

Cost: 40+(80+5)

Output: [ABC]	Logical M-Exps	Physical M-Exps
Properties: *None*	1. [AB]⋈[C]	1. [AB]⋈$_{NL}$C
	2. [BC]⋈[A]	2. [BC]⋈$_{NL}$A
	3. [AC]⋈[B]	3. [AC]⋈$_{NL}$B
	4. [B]⋈[AC]	⋮

Cost: 50+(10+20)

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

Cost: 10

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

Cost: 20

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

CASCADES IMPLEMENTATIONS

Standalone:

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

Integrated:

- Microsoft SQL Server (1990s)
- Tandem NonStop SQL (1990s)
- CockroachDB (2010s)

RANDOMIZED ALGORITHMS

Perform a random walk over a solution space of all possible (valid) plans for a query.

Continue searching until a cost threshold is reached or the optimizer runs for a length of time.

Examples: Postgres’ genetic algorithm.

SIMULATED ANNEALING

Start with a query plan that is generated using the heuristic-only approach.

Compute random permutations of operators (e.g., swap the join order of two tables):

- Always accept a change that reduces cost.
- Only accept a change that increases cost with some probability.
- Reject any change that violates correctness (e.g., sort ordering).

POSTGRES GENETIC OPTIMIZER

More complicated queries use a genetic algorithm that selects join orderings (GEQO).

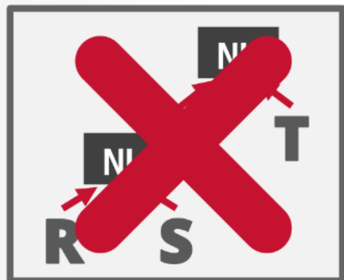
At the beginning of each round, generate different variants of the query plan.

Select the plans that have the lowest cost and permute them with other plans. Repeat.

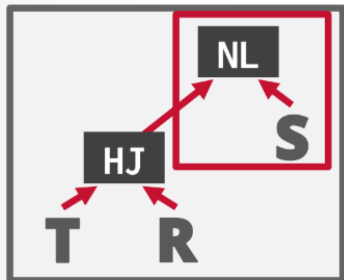
- The mutator function only generates valid plans.

POSTGRES GENETIC OPTIMIZER

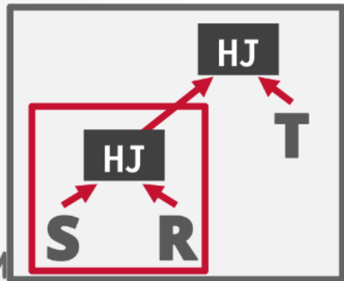
1st Generation



Cost: 300

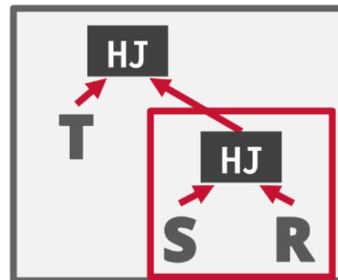


Cost: 200



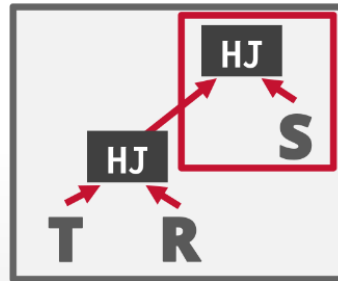
Cost: 100

2nd Generation



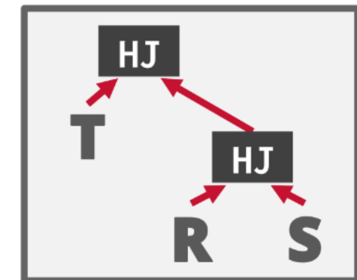
Cost: 80

Cost: 200

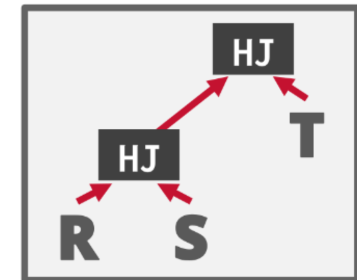


Cost: 110

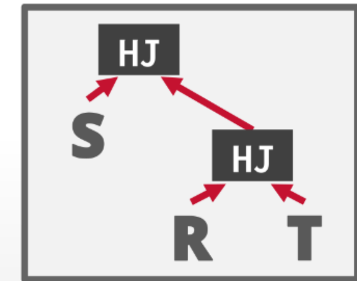
3rd Generation



Cost: 90



Cost: 160



Cost: 120

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.

PARTING THOUGHTS

Most new DBMSs implement a variant of Ingres or System R query optimizator.

Query optimization is hard.

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