ADVANCED DATABASE SYSTEMS

Query Optimizer Implementation Part 2

14

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

LAST CLASS

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

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

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

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

Choice #5: Randomized Search
→ Academics in the 1980s, current Postgres

STRATIFIED SEARCH

First rewrite the logical query plan using transformation rules.

→ The engine checks whether the transformation is allowed before it can be applied.

→ Cost is never considered in this step.

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

UNIFIED SEARCH

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

→ No need for separate stages because everything is transformations.

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

TOP-DOWN VS. BOTTOM-UP

Top-down Optimization

→ Start with the outcome that the query wants, and then work down the tree to find the optimal plan that gets you to that goal.

→ **Examples**: Volcano, Cascades

Bottom-up Optimization

→ Start with nothing and then build up the plan to get to the outcome that you want.

→ **Examples**: System R, Starburst

TODAY'S AGENDA

Unified Search
Randomized Search
Real-World Implementations
Unnesting Subqueries

CASCADES OPTIMIZER

Graefe

Object-oriented implementation of the previous Volcano query optimizer.

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

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

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.

Predicates as logical/physical operators.
→ Use same pattern/rule engine for expressions.

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 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. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$
Properties: *None*	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] Properties: *None*	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] Properties: *None*	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

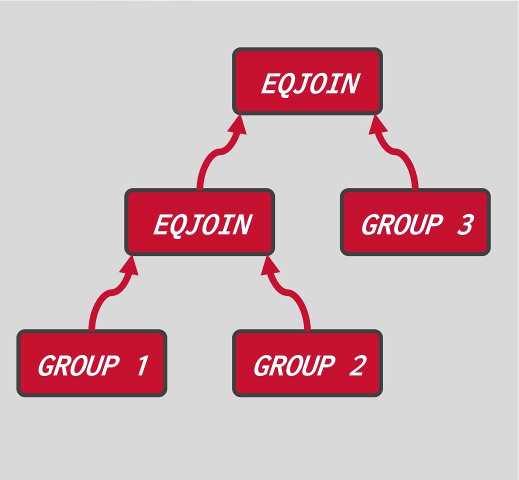
Each rule is represented as a pair of attributes:

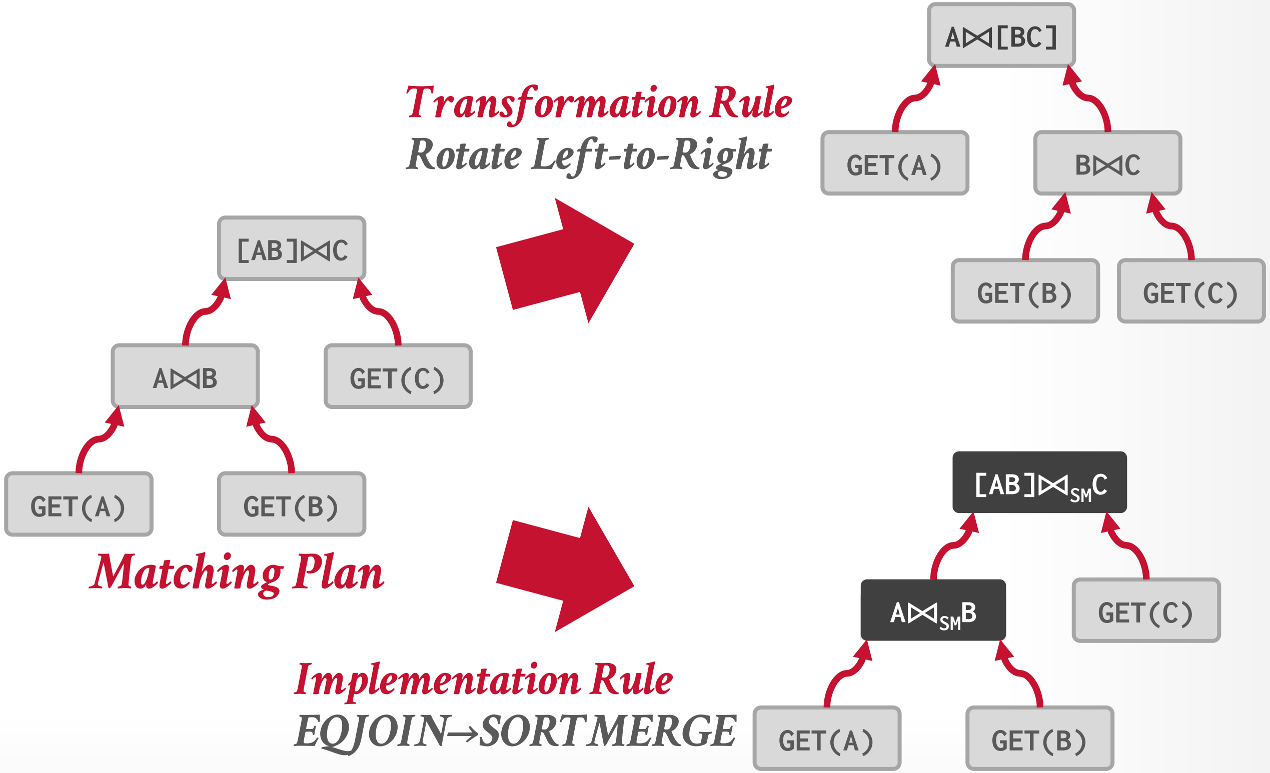
→ **Pattern**: Defines the structure of the logical expression that can be applied to the rule.

→ **Substitute**: Defines the structure of the result after applying the rule.

CASCADES: RULES

Pattern





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

CASCADES: 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.

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

CASCADES: MEMO TABLE

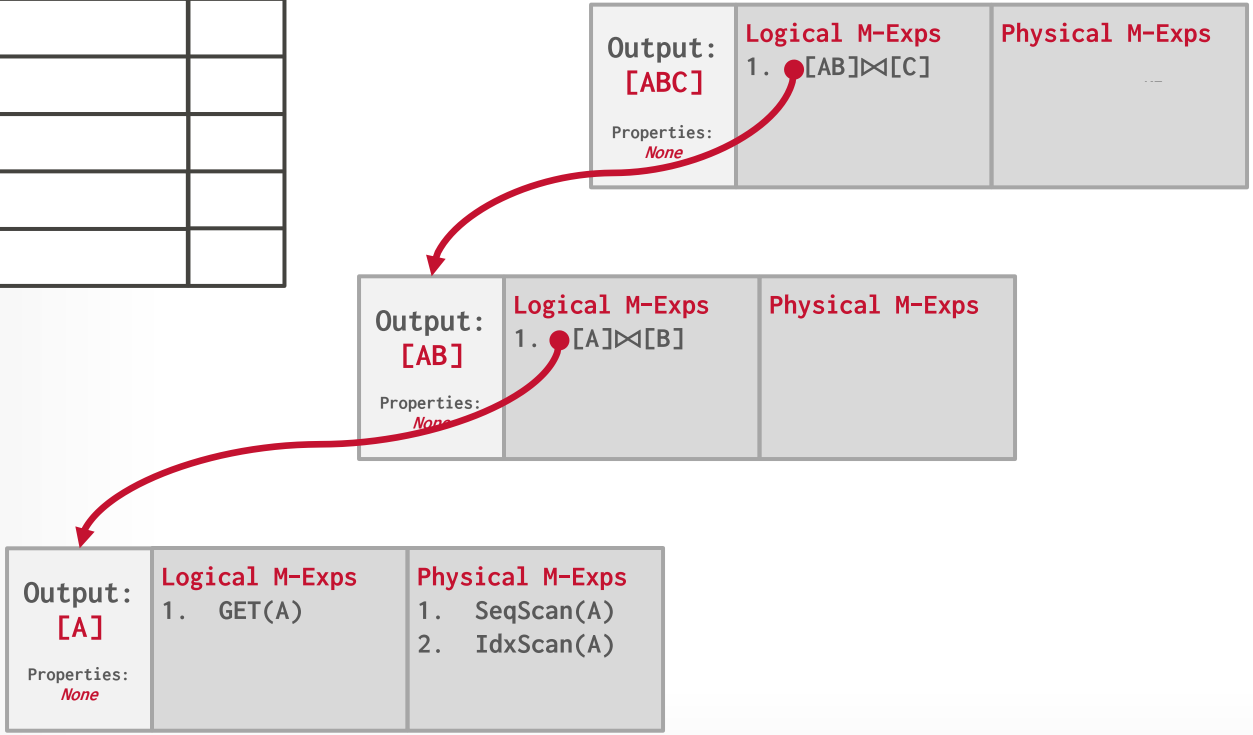
	Best Expr	*Cost*
[ABC]		
[AB]		
[A]		
[B]		
[C]		

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

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

CASCADES: MEMO TABLE

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



CASCADES: MEMO TABLE

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

Output: [ABC]
Properties: *None*

Logical M-Exps
1. [AB]⋈[C]

Physical M-Exps

Output: [AB]
Properties: *None*

Logical M-Exps
1. [A]⋈[B]

Physical M-Exps

Cost: 10

Output: [A]
Properties: *None*

Logical M-Exps
1. GET(A)

Physical M-Exps
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 M-Exps
1. [AB]⋈[C]

Physical M-Exps

Output: [AB]
Properties: *None*

Logical M-Exps
1. [A]⋈[B]

Physical M-Exps

Cost: 10

Output: [A]
Properties: *None*

Logical M-Exps
1. GET(A)

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

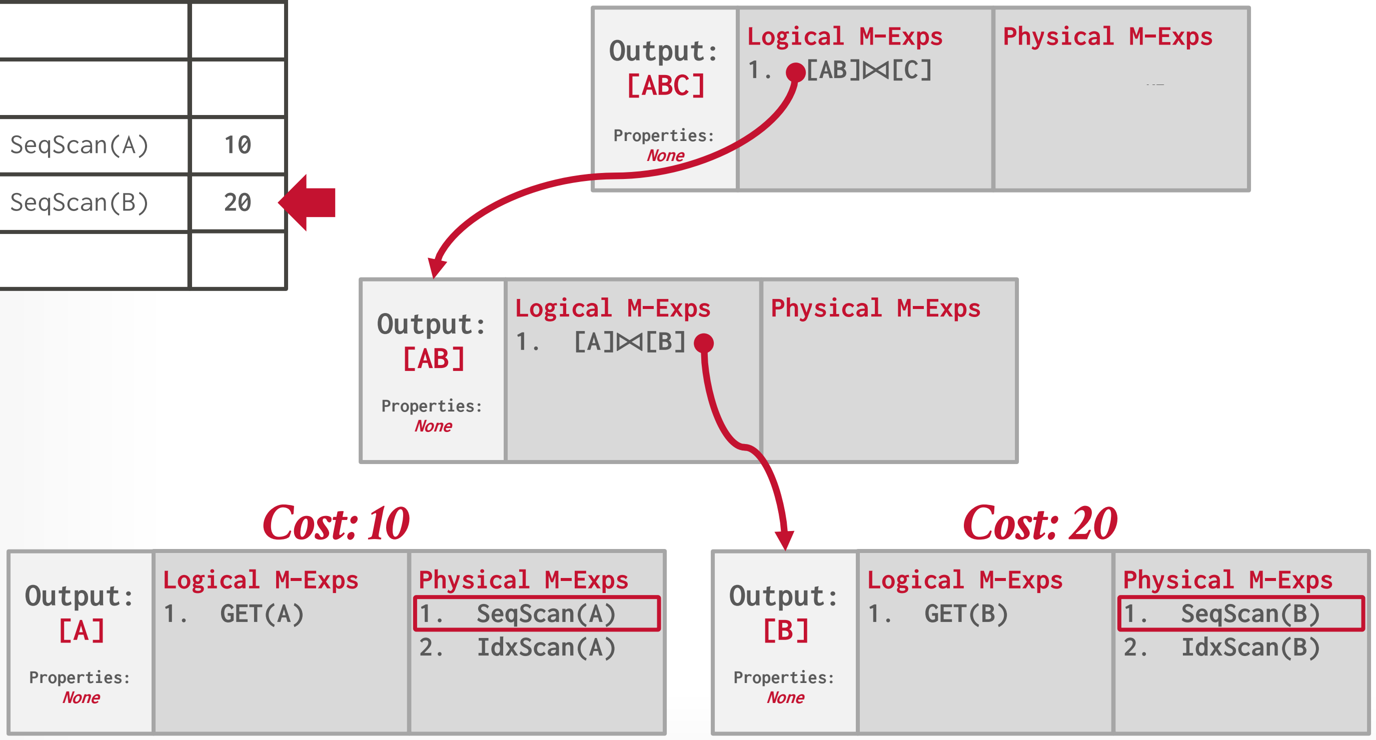
Output: [B]
Properties: *None*

Logical M-Exps
1. GET(B)

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

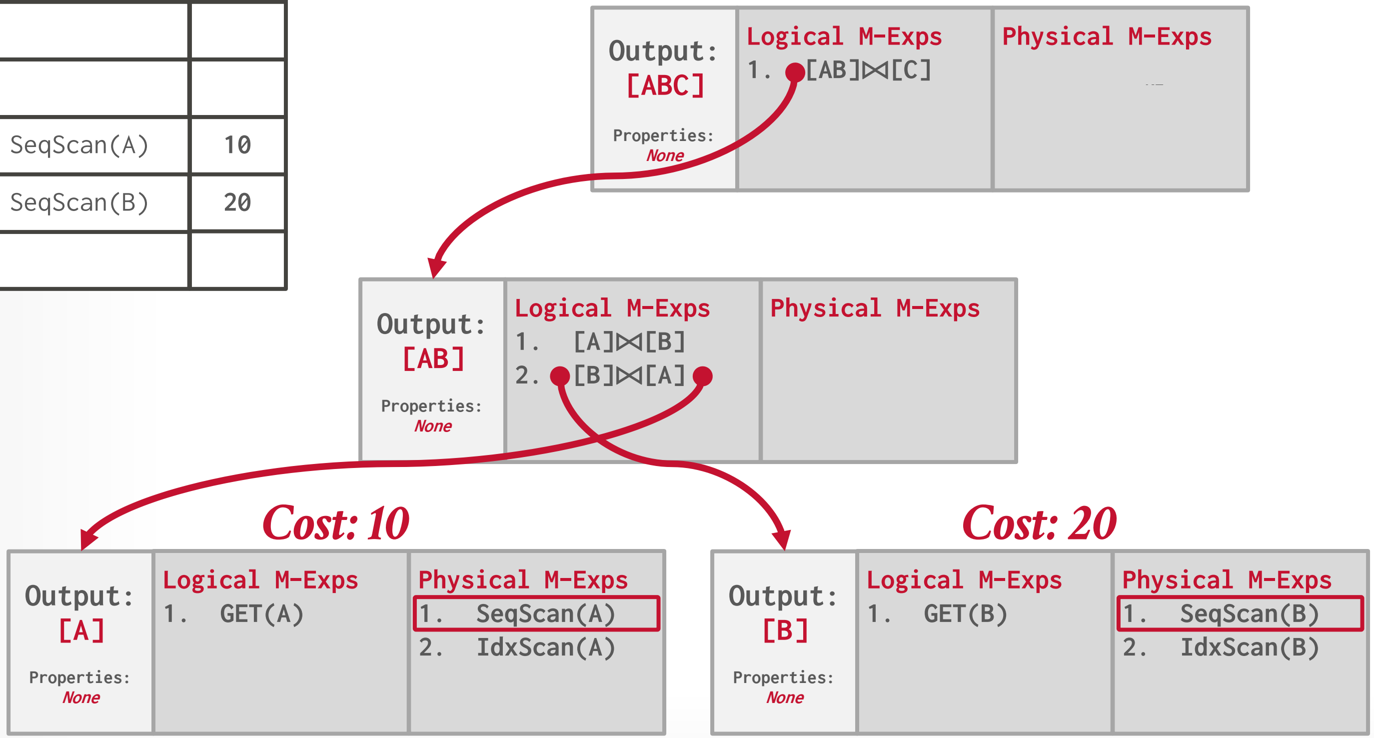
CASCADES: MEMO TABLE

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



CASCADES: MEMO TABLE

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



CASCADES: MEMO TABLE

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

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

Output: [AB]	Logical M-Exps	Physical M-Exps
Properties: *None*	1. [A]⋈[B] 2. ~~[B]⋈[A]~~	

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: MEMO TABLE

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

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

Output: [AB] Properties: *None*	Logical M-Exps	Physical M-Exps
	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: 10

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

Cost: 20

Output: [B] Properties: *None*	Logical M-Exps	Physical M-Exps
	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]		

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

Cost: 50+(10+20)

Output: [AB] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. $[A]\bowtie[B]$	1. $[A]\bowtie_{NL}[B]$
	2. $[B]\bowtie[A]$	2. $\boxed{[A]\bowtie_{HJ}[B]}$
		3. $[B]\bowtie_{NL}[A]$
		4. $[B]\bowtie_{HJ}[A]$

Cost: 10

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

Cost: 20

Output: [B] Properties: *None*	Logical M-Exps	Physical M-Exps
	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]		

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

Cost: 50+(10+20)

Output: [AB] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. $[A]\bowtie[B]$	1. $[A]\bowtie_{NL}[B]$
	2. $[B]\bowtie[A]$	2. $\boxed{[A]\bowtie_{HJ}[B]}$
		3. $[B]\bowtie_{NL}[A]$
		4. $[B]\bowtie_{HJ}[A]$

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

Cost: 10

Output: [A] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. GET(A)	1. $\boxed{\text{SeqScan(A)}}$
		2. IdxScan(A)

Cost: 20

Output: [B] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. GET(B)	1. $\boxed{\text{SeqScan(B)}}$
		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]	Logical M-Exps	Physical M-Exps
Properties: *None*	1. $[AB]\bowtie[C]$	

Cost: 50+(10+20)

Output: [AB]	Logical M-Exps	Physical M-Exps
Properties: *None*	1. $[A]\bowtie[B]$ 2. $[B]\bowtie[A]$	1. $[A]\bowtie_{NL}[B]$ $\boxed{\text{2. } [A]\bowtie_{HJ}[B]}$ 3. $[B]\bowtie_{NL}[A]$ 4. $[B]\bowtie_{HJ}[A]$

Cost: 5

Output: [C]	Logical M-Exps	Physical M-Exps
Properties: *None*	1. GET(C)	1. SeqScan(C) $\boxed{\text{2. IdxScan(C)}}$

Cost: 10

Output: [A]	Logical M-Exps	Physical M-Exps
Properties: *None*	1. GET(A)	$\boxed{\text{1. SeqScan(A)}}$ 2. IdxScan(A)

Cost: 20

Output: [B]	Logical M-Exps	Physical M-Exps
Properties: *None*	1. GET(B)	$\boxed{\text{1. SeqScan(B)}}$ 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 M-Exps	Physical M-Exps
	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 M-Exps	Physical M-Exps
	1. $[A]\bowtie[B]$	1. $[A]\bowtie_{NL}[B]$
	2. $[B]\bowtie[A]$	2. $\boxed{[A]\bowtie_{HJ}[B]}$
		3. $[B]\bowtie_{NL}[A]$
		4. $[B]\bowtie_{HJ}[A]$

Cost: 5

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

Cost: 10

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

Cost: 20

Output: [B] Properties: *None*	Logical M-Exps	Physical M-Exps
	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 M-Exps	Physical M-Exps
	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 M-Exps	Physical M-Exps
	1. $[A]\bowtie[B]$	1. $[A]\bowtie_{NL}[B]$
	2. $[B]\bowtie[A]$	$\boxed{\text{2. } [A]\bowtie_{HJ}[B]}$
		3. $[B]\bowtie_{NL}[A]$
		4. $[B]\bowtie_{HJ}[A]$

Cost: 5

Output: [C] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. GET(C)	1. SeqScan(C)
		$\boxed{\text{2. IdxScan(C)}}$

Cost: 10

Output: [A] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. GET(A)	$\boxed{\text{1. SeqScan(A)}}$
		2. IdxScan(A)

Cost: 20

Output: [B] Properties: *None*	Logical M-Exps	Physical M-Exps
	1. GET(B)	$\boxed{\text{1. SeqScan(B)}}$
		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).

QUERY OPTIMIZATION BY SIMULATED ANNEALING
SIGMOD 1987

POSTGRES GENETIC OPTIMIZER

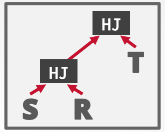
More complicated queries use a **genetic algorithm** that selects join orderings (GEQO).

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.

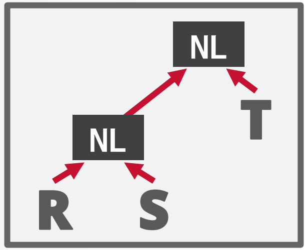
Source: Postgres Documentation

POSTGRES GENETIC OPTIMIZER

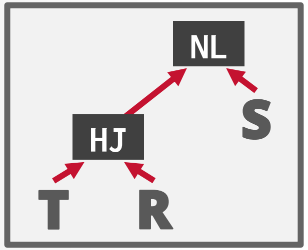


Best:100

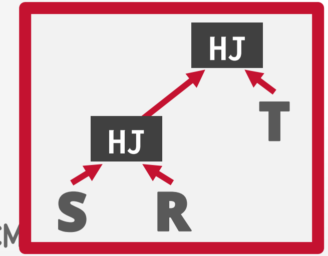
1st Generation



Cost: 300

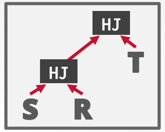


Cost: 200



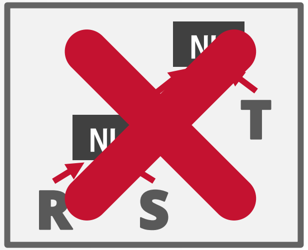
Cost: 100

POSTGRES GENETIC OPTIMIZER

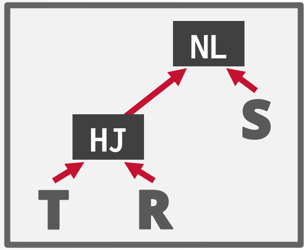


Best:100

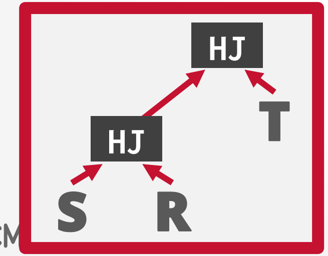
1st Generation



Cost:
300

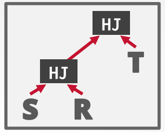


Cost:
200



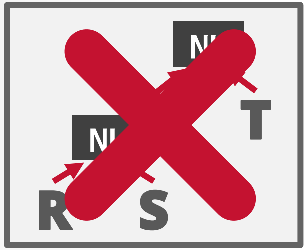
Cost:
100

POSTGRES GENETIC OPTIMIZER

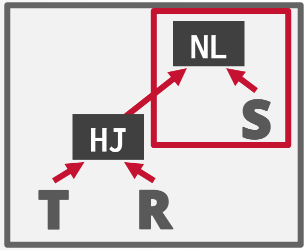


Best:100

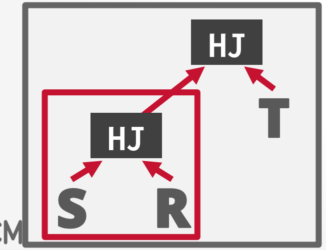
1st Generation



Cost:
300

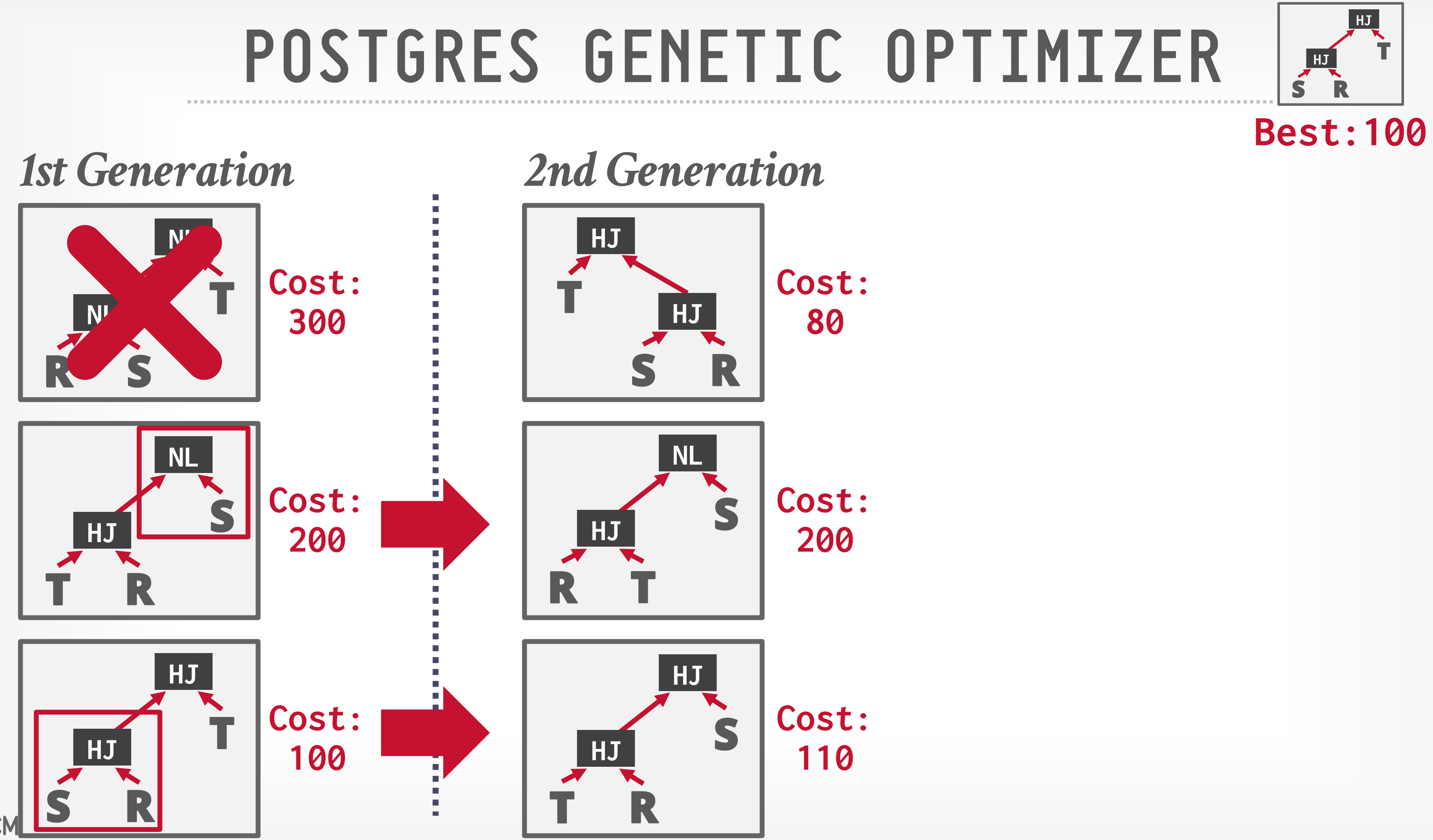


Cost:
200

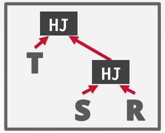


Cost:
100

POSTGRES GENETIC OPTIMIZER

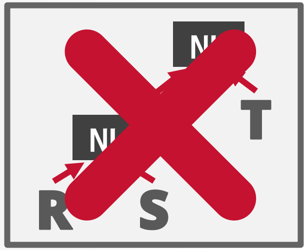


POSTGRES GENETIC OPTIMIZER

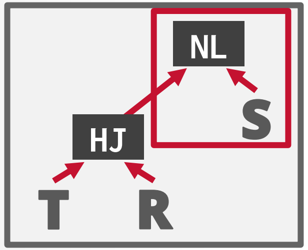


Best:80

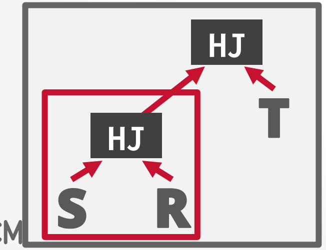
1st Generation



Cost: 300

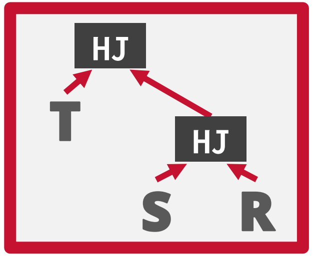


Cost: 200

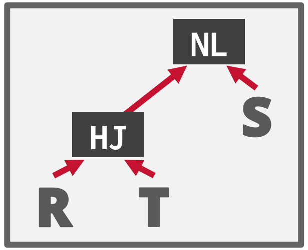


Cost: 100

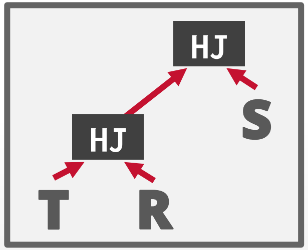
2nd Generation



Cost: 80

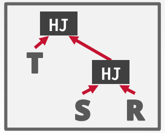


Cost: 200



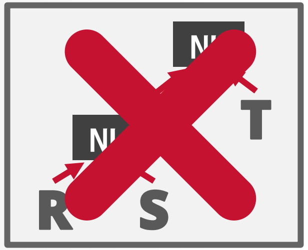
Cost: 110

POSTGRES GENETIC OPTIMIZER

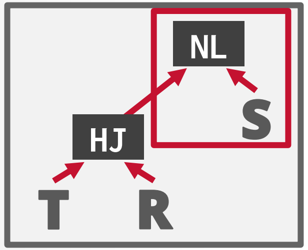


Best:80

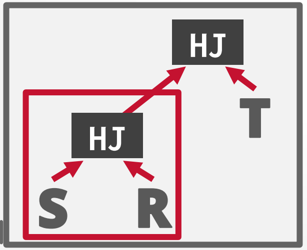
1st Generation



Cost: 300

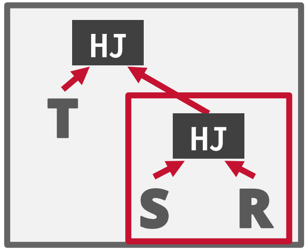


Cost: 200

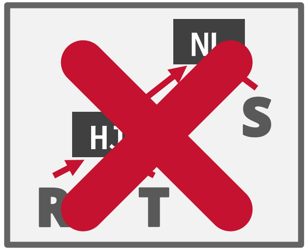


Cost: 100

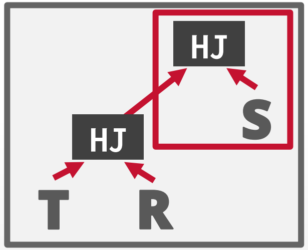
2nd Generation



Cost: 80

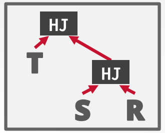


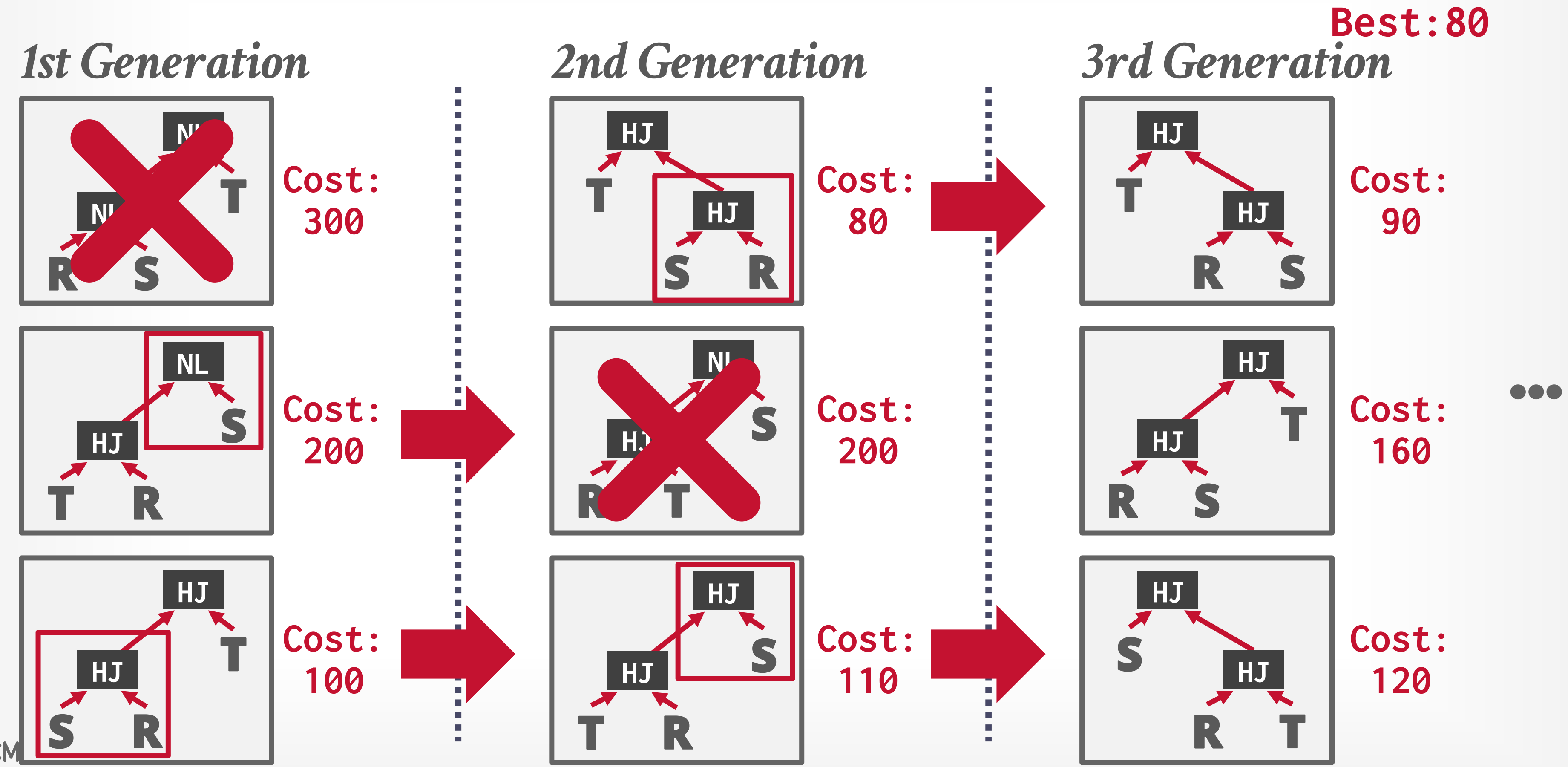
Cost: 200



Cost: 110

POSTGRES GENETIC OPTIMIZER





RANDOMIZED ALGORITHMS

Advantages:

→ Jumping around the search space randomly allows the optimizer to get out of local minimums.

→ Low memory overhead (if no history is kept).

Disadvantages:

→ Difficult to determine why the DBMS may have chosen a plan.

→ Must do extra work to ensure that query plans are deterministic.

→ Still must implement correctness rules.

RANDOMIZED ALGORITHMS

Advantages:

→ Jumping arour
optimizer to g

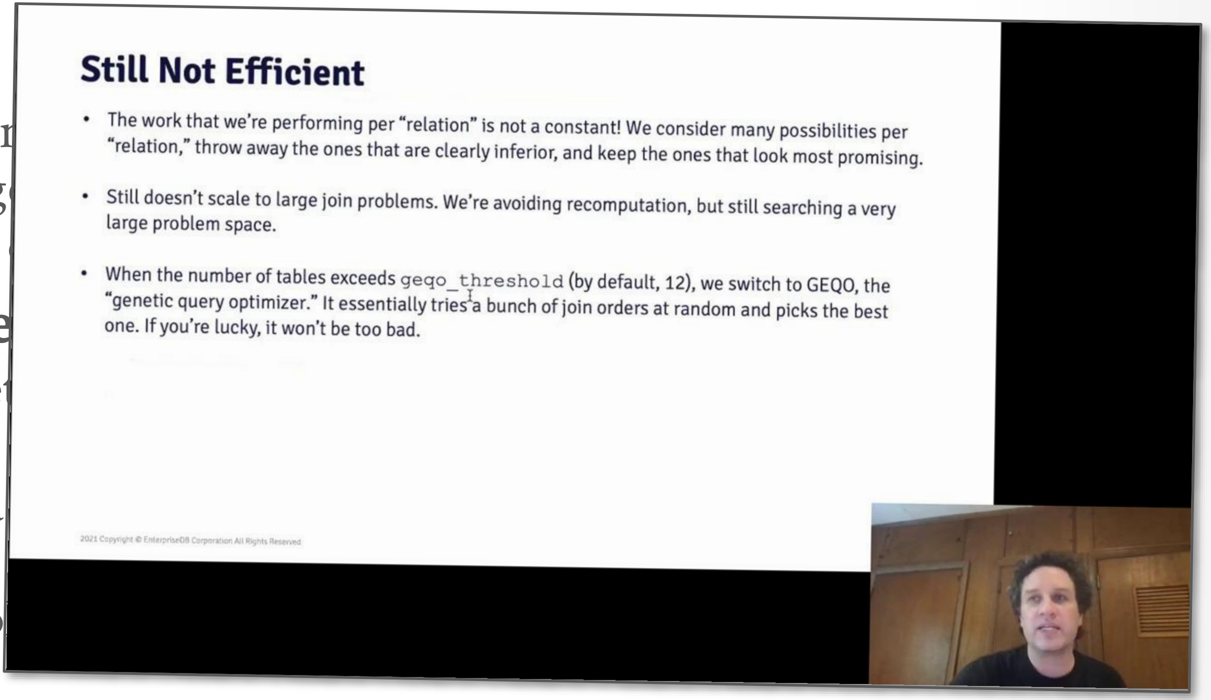
→ Low memory

Disadvantage

→ Difficult to de
plan.

→ Must do extra
deterministic.

→ Still must imp



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

REAL-WORLD IMPLEMENTATIONS

- Microsoft SQL Server
- Apache Calcite
- Greenplum Orca
- CockroachDB
- SingleStore
- Snowflake

Cascades

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.

MICROSOFT SQL SERVER

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

Cost-based Search Initialization

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

Engine-Specific Transformations

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

Tree-to-Tree Transformations

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

Multi-Stage Cost-Based Search

Source: Nico Bruno + Cesar Galindo-Legaria

MICROSOFT SQL SERVER

Optimization #1: Timeouts are based on the number of transformations not wallclock time.

→ Ensures that overloaded systems do not generate different plans than under normal operations.

Optimization #2: Pre-populate the Memo Table with potentially useful join orderings.

→ Heuristics that consider relationships between tables.

→ Syntactic appearance in query.

APACHE CALCITE

Standalone extensible query optimization framework for data processing systems.

→ Support for pluggable query languages, cost models, and rules.

→ Does not distinguish between logical and physical operators. Physical properties are provided as annotations.

Originally part of LucidDB.

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

GREENPLUM ORCA

Standalone Cascades implementation in C++.

→ Originally written for Greenplum.

→ Extended to support HAWQ.

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

Supports multi-threaded search.

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

GREENPLUM 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 for further debugging.

Issue #2: Optimizer Accuracy

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

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.

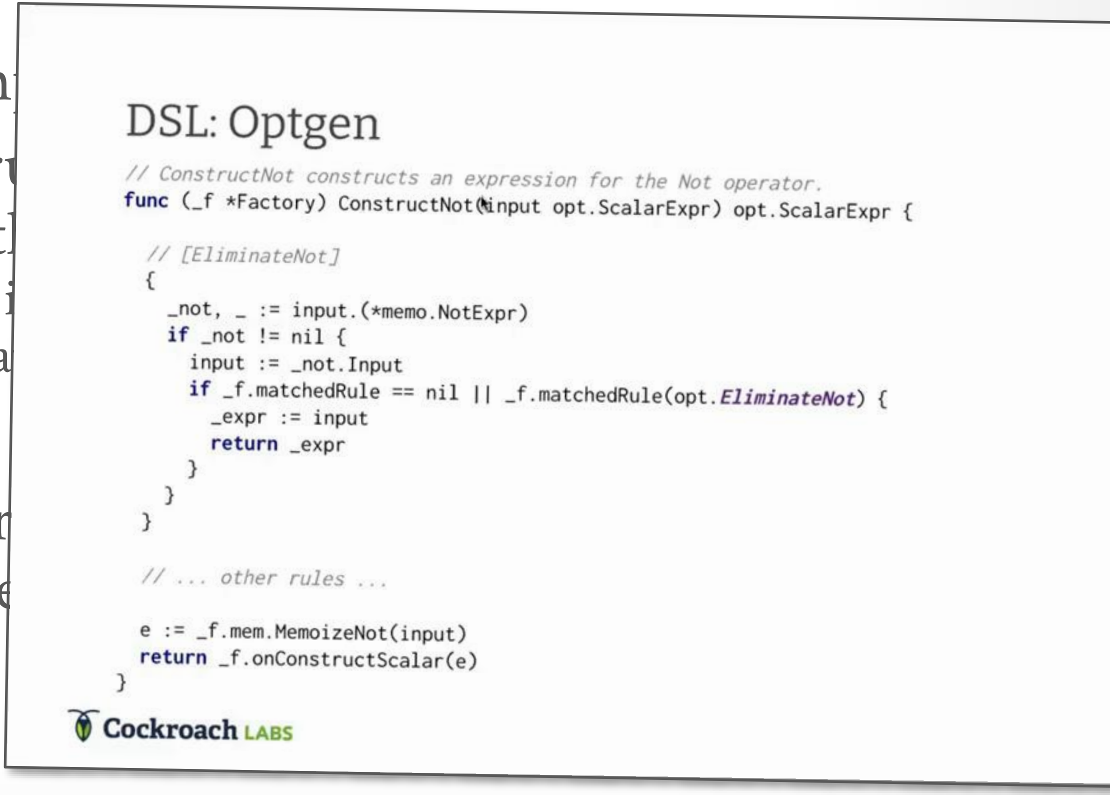
Source: Rebecca Taft

COCKROACHDB

Custom Cascades im

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

Also considers scalar
transformations toge



Source: Rebecca Taft

SUBQUERIES

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

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

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

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

SUBQUERIES

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

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

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

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

Key Distinction: Uncorrelated vs. Correlated

UNCORRELATED SUBQUERY

An uncorrelated subquery does reference any attributes from the (calling) outer query.

```
SELECT name
  FROM students
 WHERE score =
  (SELECT MAX(score) FROM students);
```

The DBMS logically executes it once and reuse the result for all tuples in the outer query.

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci'

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci'

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci'   MAX(s2.score)=90

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major
GZA	CompSci

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci' MAX(s2.score)=90

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major
GZA	CompSci

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci' MAX(s2.score)=90

s1.major='CompSci'

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major
GZA	CompSci

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci'   MAX(s2.score)=90

s1.major='CompSci'

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major
GZA	CompSci

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci'   MAX(s2.score)=90

s1.major='CompSci'   MAX(s2.score)=90

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major
GZA	CompSci

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci' MAX(s2.score)=90

s1.major='CompSci' MAX(s2.score)=90

s1.major='Streets'

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major
GZA	CompSci

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

```
s1.major='CompSci'      MAX(s2.score)=90
s1.major='CompSci'      MAX(s2.score)=90
s1.major='Streets'
```

CORRELATED SUBQUERY

A correlated subquery refers to one or more attributes from outside of the subquery (i.e., the outer query).

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

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

name	major
GZA	CompSci
ODB	Streets

name	major	score
GZA	CompSci	90
RZA	CompSci	80
ODB	Streets	100

s1.major='CompSci' MAX(s2.score)=90

s1.major='CompSci' MAX(s2.score)=90

s1.major='Streets' MAX(s2.score)=100

HEURISTIC REWRITING

Almost every DBMS that supports uses heuristics that identify specific query plan patterns to decorrelate nested subqueries.

The goal is to move the subquery up a level so that the DBMS can execute it as a join.

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

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

ORTHOGONAL OPTIMIZATION OF SUBQUERIES AND AGGREGATION
SIGMOD 2001

HEURISTIC REWRITING

Almost every DBMS that supp
u
qu
ne

Th
lev
as

$$R\ \mathcal{A}^{\otimes}\ E \;=\; R \otimes_{\text{true}} E, \quad \text{if no parameters in } E \text{ resolved from } R \qquad (1)$$

$$R\ \mathcal{A}^{\otimes}\ (\sigma_p E) \;=\; R \otimes_p E, \quad \text{if no parameters in } E \text{ resolved from } R \qquad (2)$$

$$R\ \mathcal{A}^{\times}\ (\sigma_p E) \;=\; \sigma_p(R\ \mathcal{A}^{\times}\ E) \qquad (3)$$

$$R\ \mathcal{A}^{\times}\ (\pi_v E) \;=\; \pi_{v \cup \text{columns}(R)}(R\ \mathcal{A}^{\times}\ E) \qquad (4)$$

$$R\ \mathcal{A}^{\times}\ (E_1 \cup E_2) \;=\; (R\ \mathcal{A}^{\times}\ E_1) \cup (R\ \mathcal{A}^{\times}\ E_2) \qquad (5)$$

$$R\ \mathcal{A}^{\times}\ (E_1 - E_2) \;=\; (R\ \mathcal{A}^{\times}\ E_1) - (R\ \mathcal{A}^{\times}\ E_2) \qquad (6)$$

$$R\ \mathcal{A}^{\times}\ (E_1 \times E_2) \;=\; (R\ \mathcal{A}^{\times}\ E_1) \bowtie_{R.key} (R\ \mathcal{A}^{\times}\ E_2) \qquad (7)$$

$$R\ \mathcal{A}^{\times}\ (\mathcal{G}_{A,F} E) \;=\; \mathcal{G}_{A \cup \text{columns}(R),F}(R\ \mathcal{A}^{\times}\ E) \qquad (8)$$

$$R\ \mathcal{A}^{\times}\ (\mathcal{G}^1_F E) \;=\; \mathcal{G}_{\text{columns}(R),F'}(R\ \mathcal{A}^{\text{LOJ}}\ E) \qquad (9)$$

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

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

ORTHOGONAL OPTIMIZATION OF SUBQUERIES AND AGGREGATION
SIGMOD 2001

HEURISTIC RE…

Almost every DBMS that supp…

u…

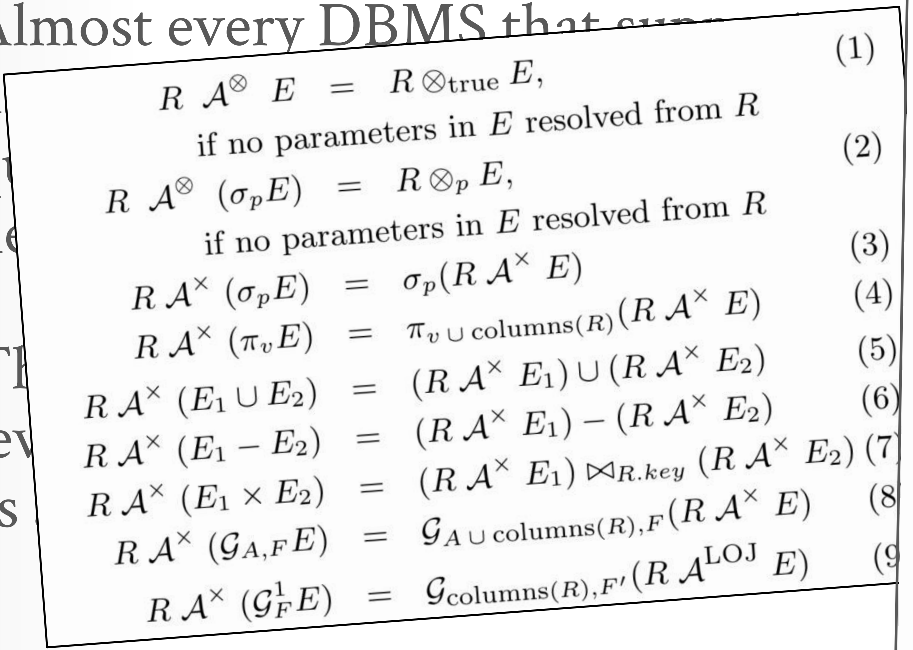
qu…

ne…

Th…

lev…

as …



$$R\ \mathcal{A}^{\otimes}\ E \;=\; R \otimes_{\text{true}} E, \quad \text{if no parameters in } E \text{ resolved from } R \tag{1}$$

$$R\ \mathcal{A}^{\otimes}\ (\sigma_p E) \;=\; R \otimes_p E, \quad \text{if no parameters in } E \text{ resolved from } R \tag{2}$$

$$R\ \mathcal{A}^{\times}\ (\sigma_p E) \;=\; \sigma_p(R\ \mathcal{A}^{\times}\ E) \tag{3}$$

$$R\ \mathcal{A}^{\times}\ (\pi_v E) \;=\; \pi_{v \cup \text{columns}(R)}(R\ \mathcal{A}^{\times}\ E) \tag{4}$$

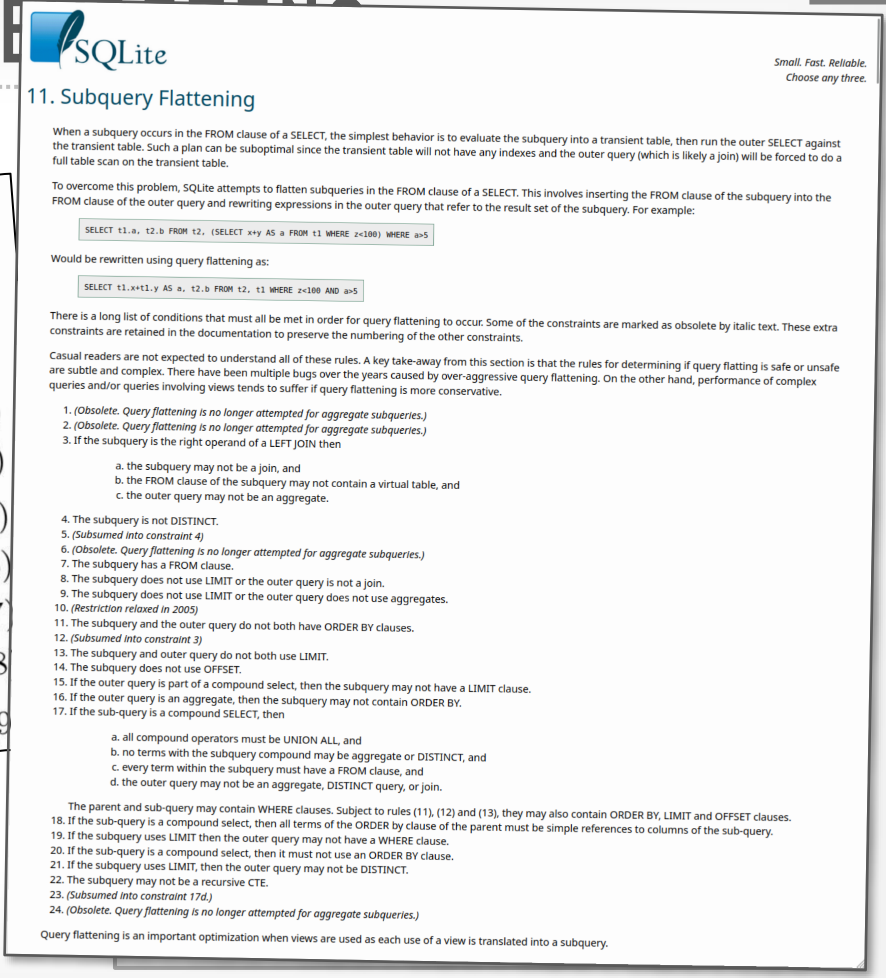
$$R\ \mathcal{A}^{\times}\ (E_1 \cup E_2) \;=\; (R\ \mathcal{A}^{\times}\ E_1) \cup (R\ \mathcal{A}^{\times}\ E_2) \tag{5}$$

$$R\ \mathcal{A}^{\times}\ (E_1 - E_2) \;=\; (R\ \mathcal{A}^{\times}\ E_1) - (R\ \mathcal{A}^{\times}\ E_2) \tag{6}$$

$$R\ \mathcal{A}^{\times}\ (E_1 \times E_2) \;=\; (R\ \mathcal{A}^{\times}\ E_1) \bowtie_{R.key} (R\ \mathcal{A}^{\times}\ E_2) \tag{7}$$

$$R\ \mathcal{A}^{\times}\ (\mathcal{G}_{A,F} E) \;=\; \mathcal{G}_{A \cup \text{columns}(R),F}(R\ \mathcal{A}^{\times}\ E) \tag{8}$$

$$R\ \mathcal{A}^{\times}\ (\mathcal{G}^1_F E) \;=\; \mathcal{G}_{\text{columns}(R),F'}(R\ \mathcal{A}^{\text{LOJ}}\ E) \tag{9}$$



SQLite

Small. Fast. Reliable.
Choose any three.

11. Subquery Flattening

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

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

```
SELECT t1.a, t2.b FROM t2, (SELECT x+y AS a FROM t1 WHERE z<100) WHERE a>5
```

Would be rewritten using query flattening as:

```
SELECT t1.x+t1.y AS a, t2.b FROM t2, t1 WHERE z<100 AND a>5
```

There is a long list of conditions that must all be met in order for query flattening to occur. Some of the constraints are marked as obsolete by italic text. These extra constraints are retained in the documentation to preserve the numbering of the other constraints.

Casual readers are not expected to understand all of these rules. A key take-away from this section is that the rules for determining if query flatting is safe or unsafe are subtle and complex. There have been multiple bugs over the years caused by over-aggressive query flattening. On the other hand, performance of complex queries and/or queries involving views tends to suffer if query flattening is more conservative.

1. *(Obsolete. Query flattening is no longer attempted for aggregate subqueries.)*
2. *(Obsolete. Query flattening is no longer attempted for aggregate subqueries.)*
3. If the subquery is the right operand of a LEFT JOIN then
    a. the subquery may not be a join, and
    b. the FROM clause of the subquery may not contain a virtual table, and
    c. the outer query may not be an aggregate.
4. The subquery is not DISTINCT.
5. *(Subsumed into constraint 4)*
6. *(Obsolete. Query flattening is no longer attempted for aggregate subqueries.)*
7. The subquery has a FROM clause.
8. The subquery does not use LIMIT or the outer query is not a join.
9. The subquery does not use LIMIT or the outer query does not use aggregates.
10. *(Restriction relaxed in 2005)*
11. The subquery and the outer query do not both have ORDER BY clauses.
12. *(Subsumed into constraint 3)*
13. The subquery and outer query do not both use LIMIT.
14. The subquery does not use OFFSET.
15. If the outer query is part of a compound select, then the subquery may not have a LIMIT clause.
16. If the outer query is an aggregate, then the subquery may not contain ORDER BY.
17. If the sub-query is a compound SELECT, then
    a. all compound operators must be UNION ALL, and
    b. no terms with the subquery compound may be aggregate or DISTINCT, and
    c. every term within the subquery must have a FROM clause, and
    d. the outer query may not be an aggregate, DISTINCT query, or join.

    The parent and sub-query may contain WHERE clauses. Subject to rules (11), (12) and (13), they may also contain ORDER BY, LIMIT and OFFSET clauses.
18. If the sub-query is a compound select, then all terms of the ORDER by clause of the parent must be simple references to columns of the sub-query.
19. If the subquery uses LIMIT then the outer query may not have a WHERE clause.
20. If the sub-query is a compound select, then it must not use an ORDER BY clause.
21. If the subquery uses LIMIT, then the outer query may not be DISTINCT.
22. The subquery may not be a recursive CTE.
23. *(Subsumed into constraint 17d.)*
24. *(Obsolete. Query flattening is no longer attempted for aggregate subqueries.)*

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

ORTHOGONAL OPTIMIZATION OF SUBQUERIES AND AGGREGATION
SIGMOD 2001

HEURISTIC REWRITING

Advantages:

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

Disadvantages:

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

Source: Mayank Baranwal

GERMAN-STYLE UNNESTING SUBQUERIES

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

The optimizer then converts dependent joins to regular joins.

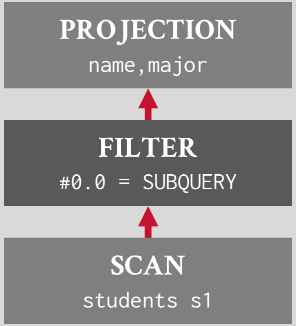
→ Some queries switch from a **$O(n^2)$** nested-loop join to a **$O(n)$** hash join.

UNNESTING ARBITRARY QUERIES
BTW 2015

FLATTENING CORRELATED QUERIES

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

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

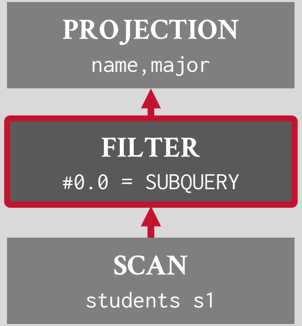


Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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

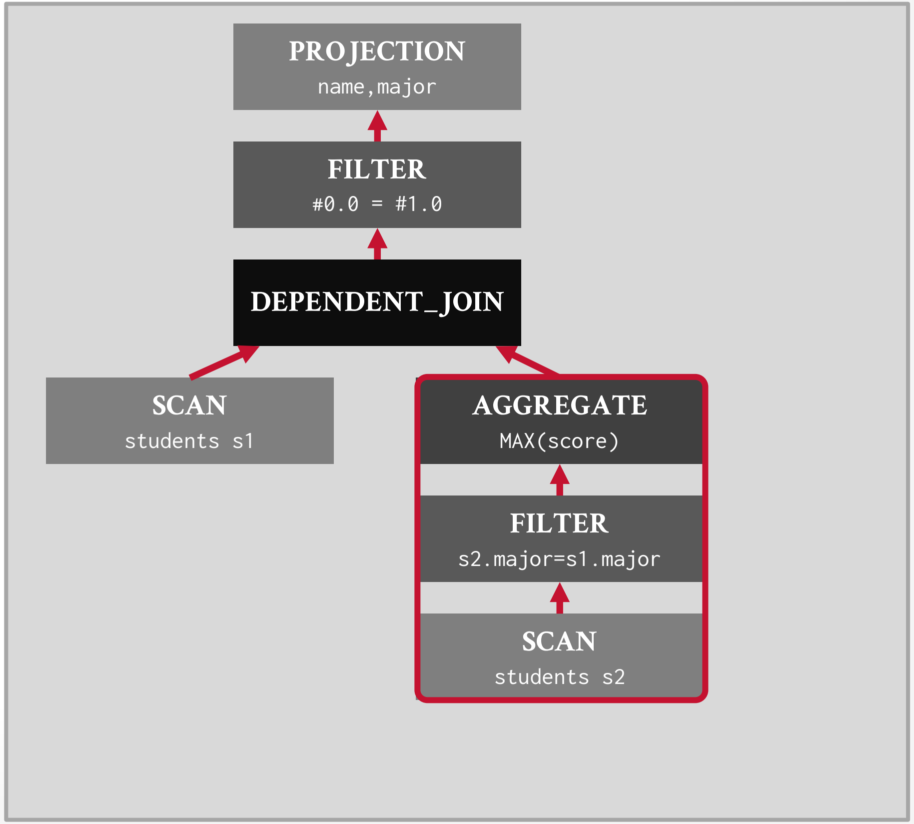


Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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



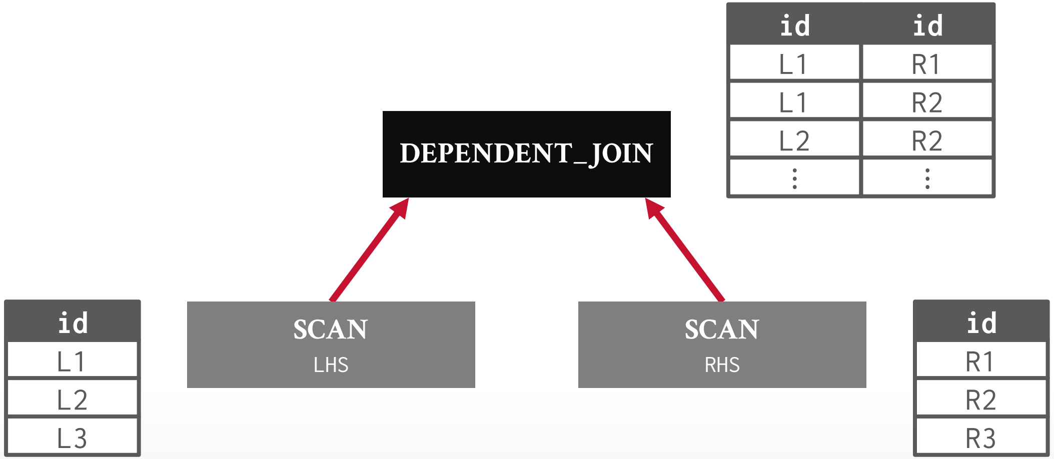
Source: Mark Raasveldt

DEPENDENT JOIN

New **dependent join** relational algebra operator that denotes a correlated subquery.

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

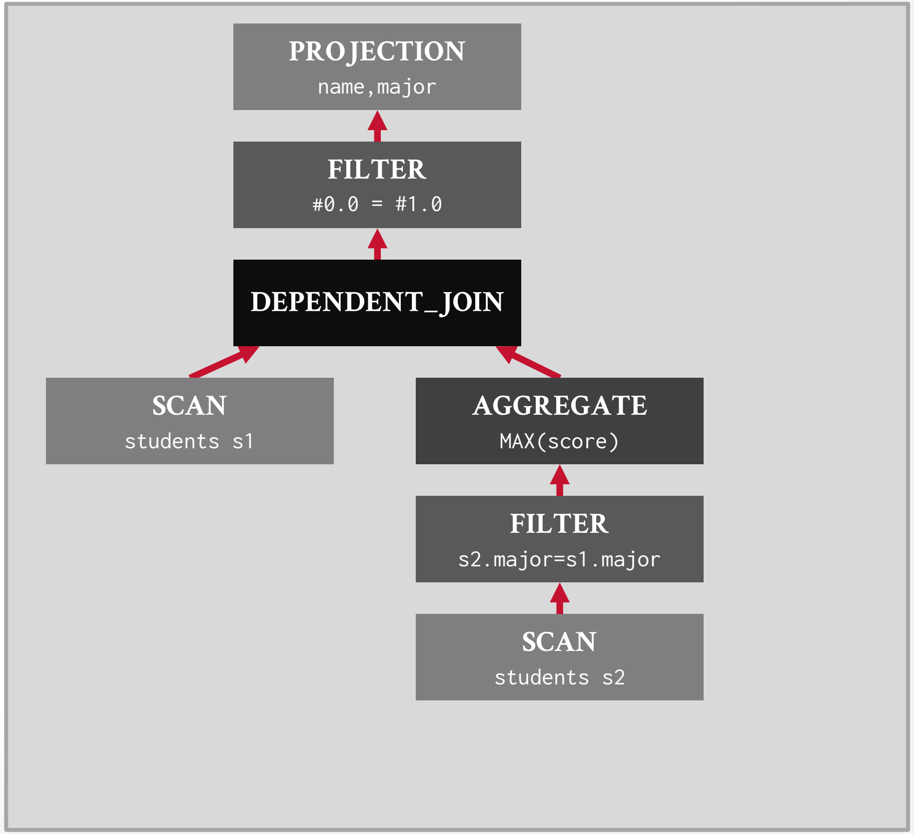
→ The operator combine results from every execution and return them as its output.



Source: Mayank Baranwal

FLATTENING CORRELATED QUERIES

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



Push dependent join down into the RHS of the plan.

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

→ Duplicate Elimination Scan

Source: Mark Raasveldt

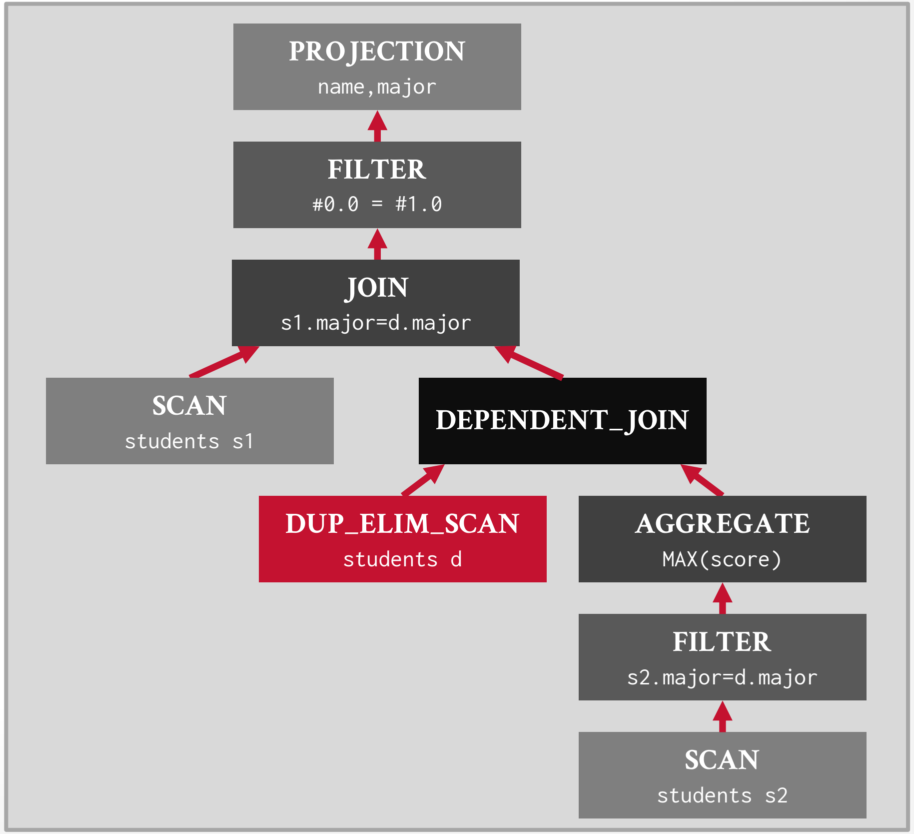
FLATTENING CORRELATED QUERIES

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

Push dependent join down into the RHS of the plan.

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

→ Duplicate Elimination Scan

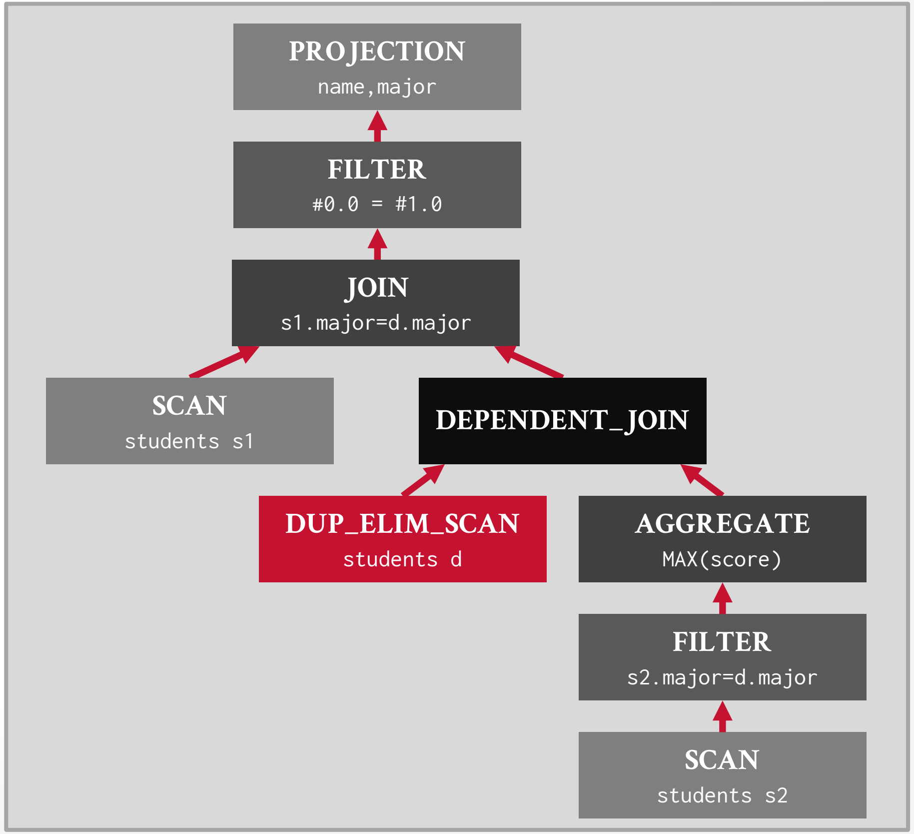


Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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

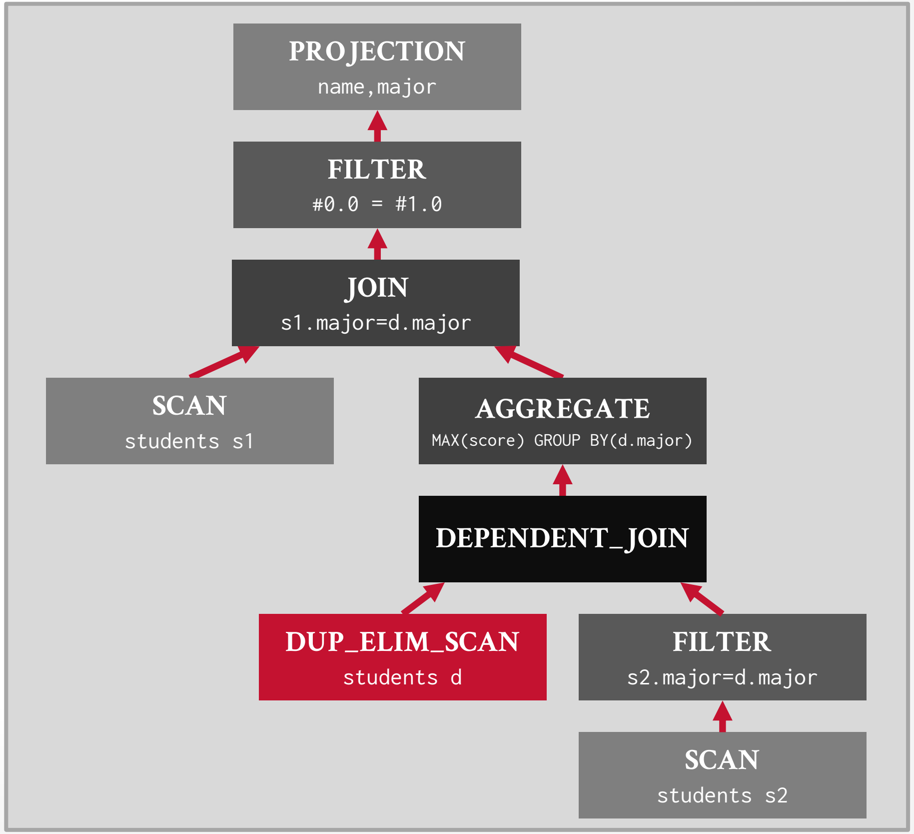


Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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

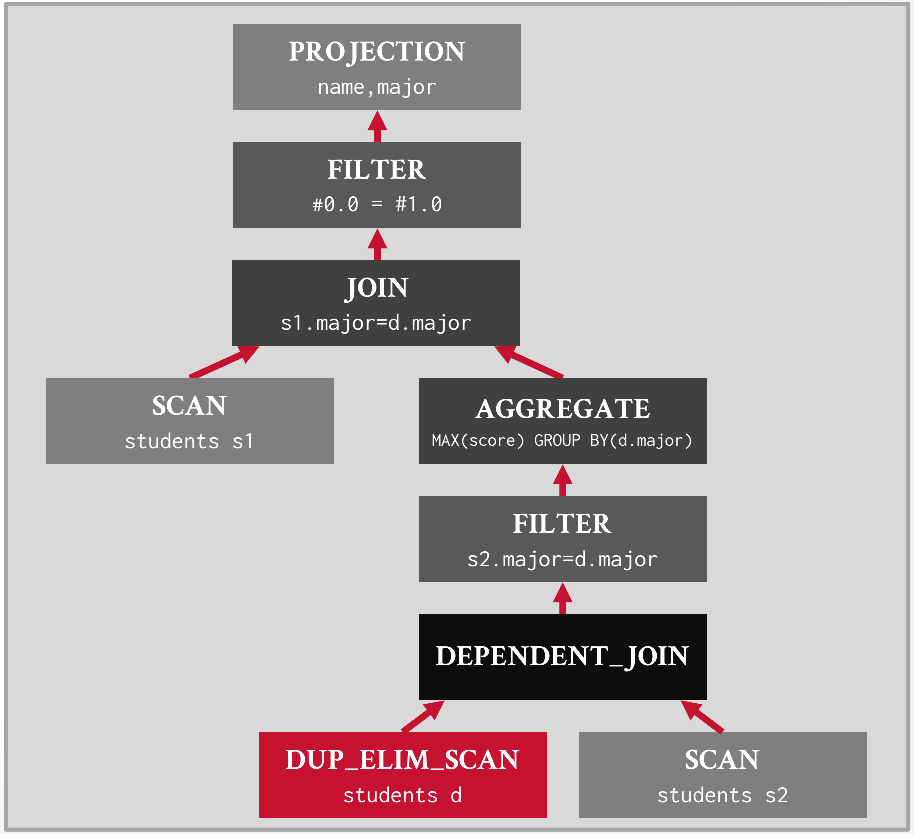


Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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

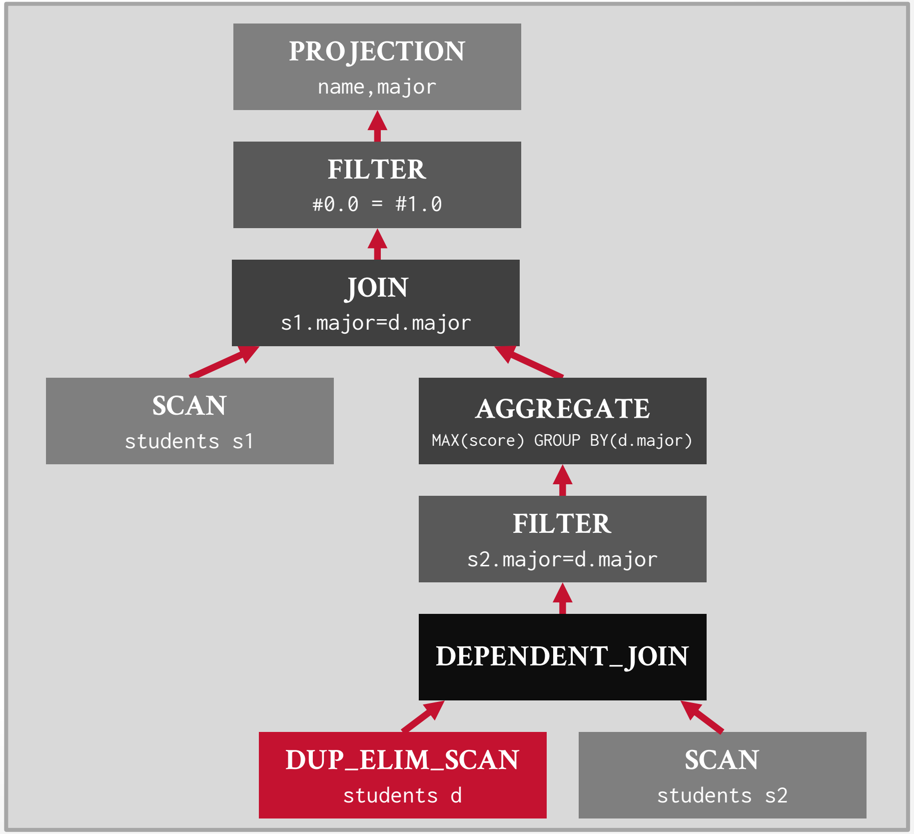


Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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

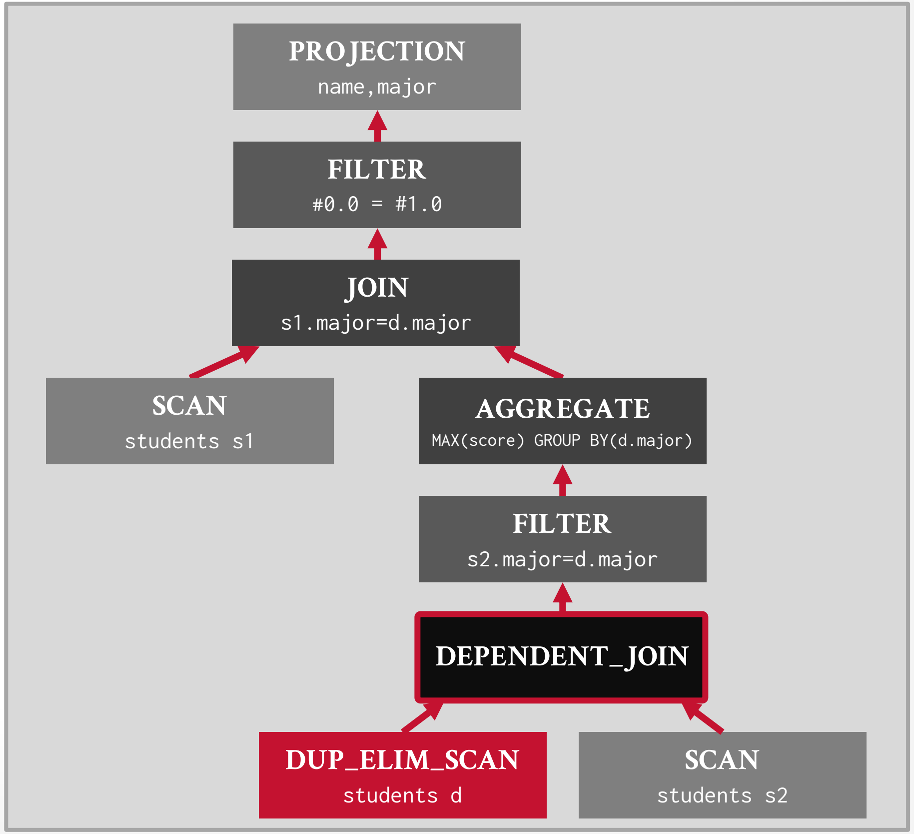


Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

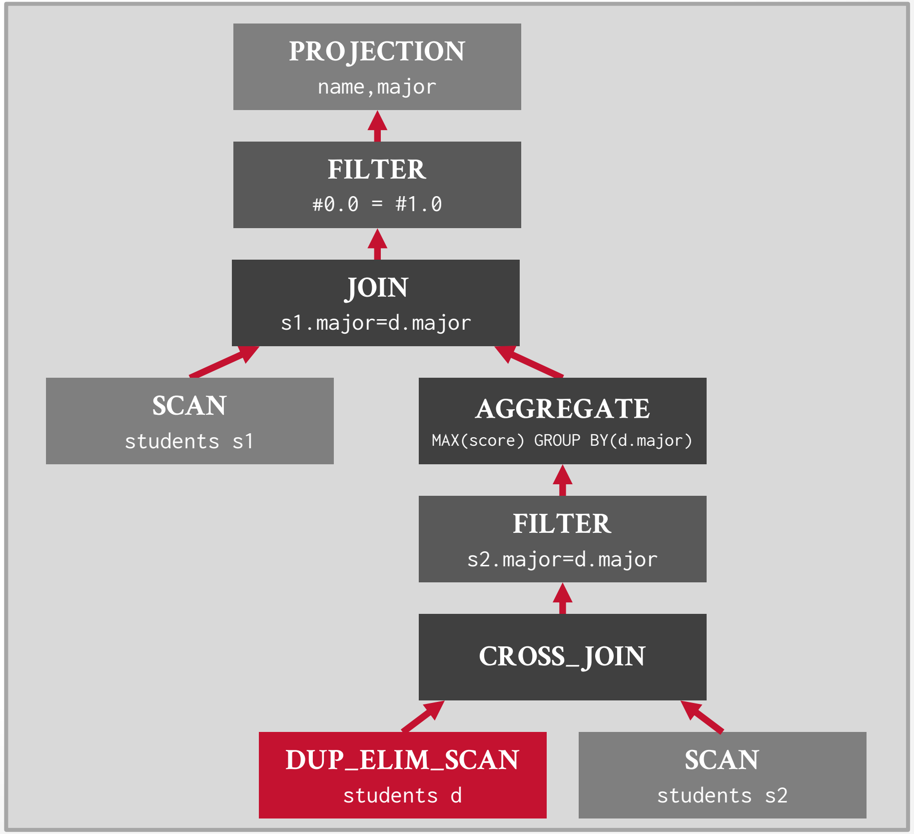
Convert the **dependent join** operator into a **cross product**.



FLATTENING CORRELATED QUERIES

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

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



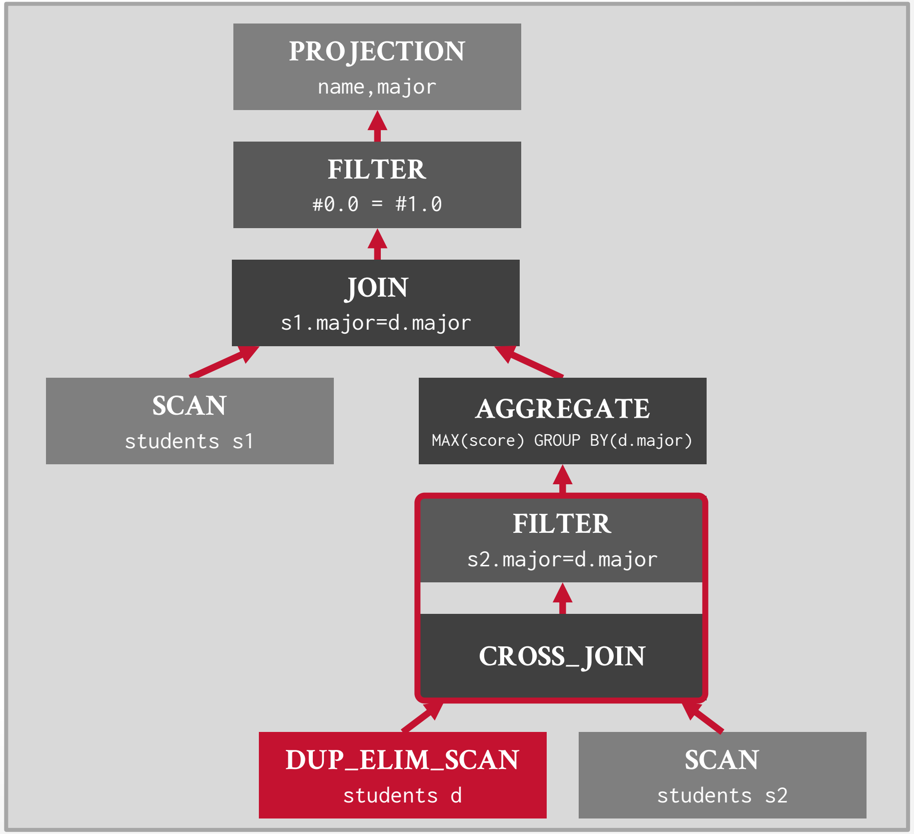
Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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

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



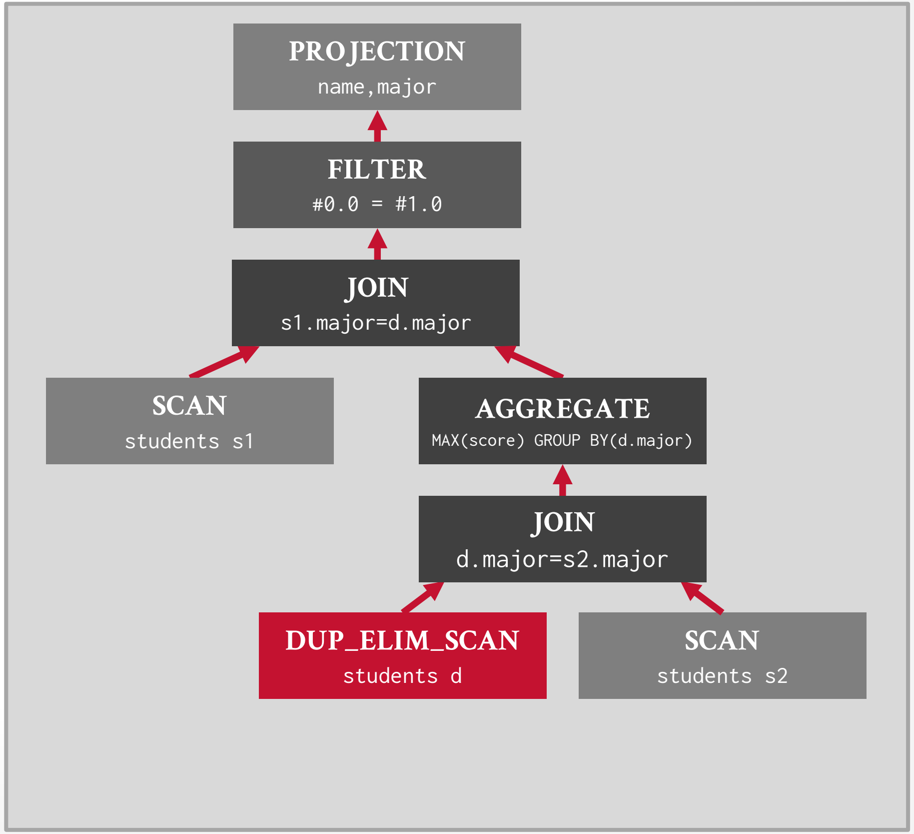
Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

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

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

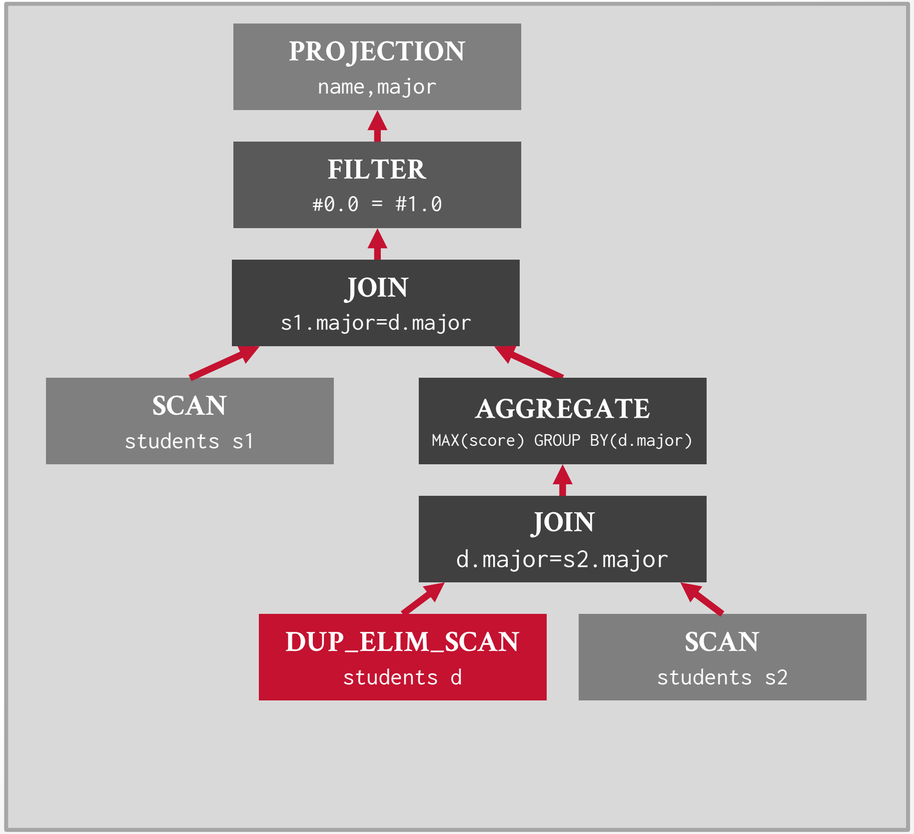


FLATTENING CORRELATED QUERIES

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

Remove duplicate elimination scan entirely.

Remove the filter above the new join.



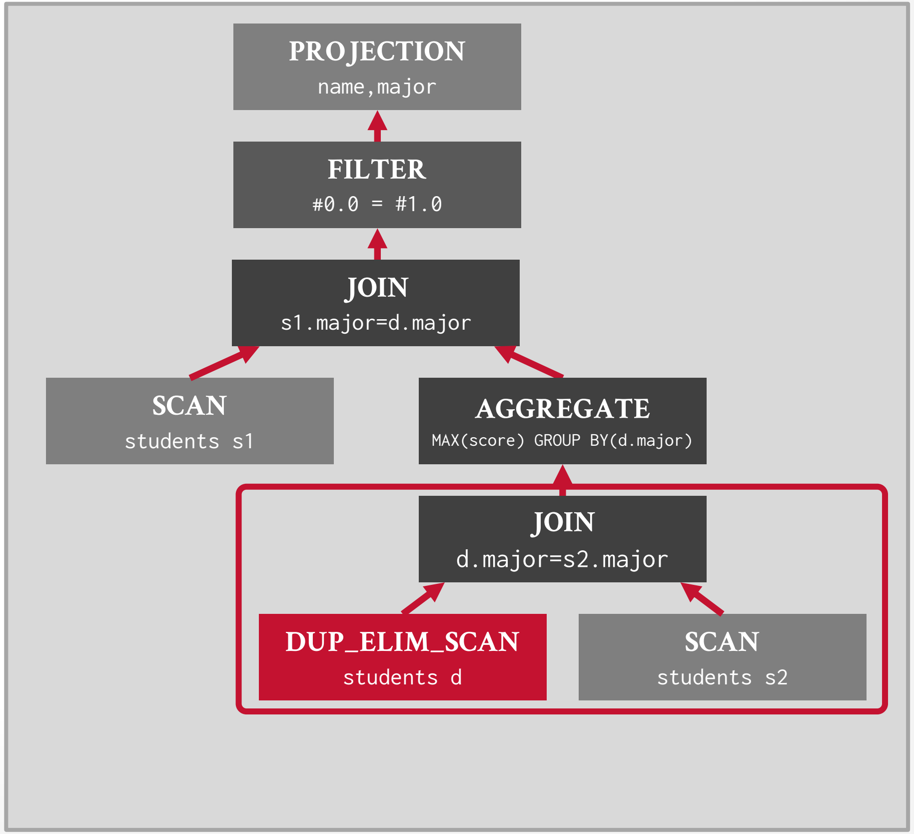
Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

Remove duplicate elimination scan entirely.

Remove the filter above the new join.



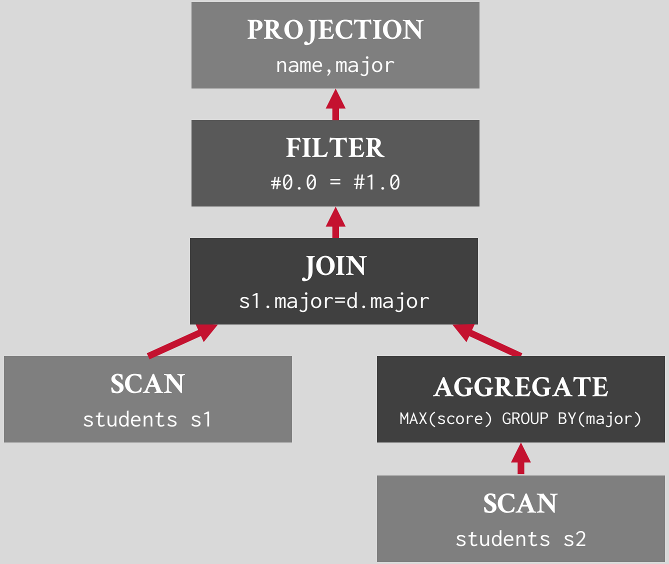
Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

Remove duplicate elimination scan entirely.

Remove the filter above the new join.



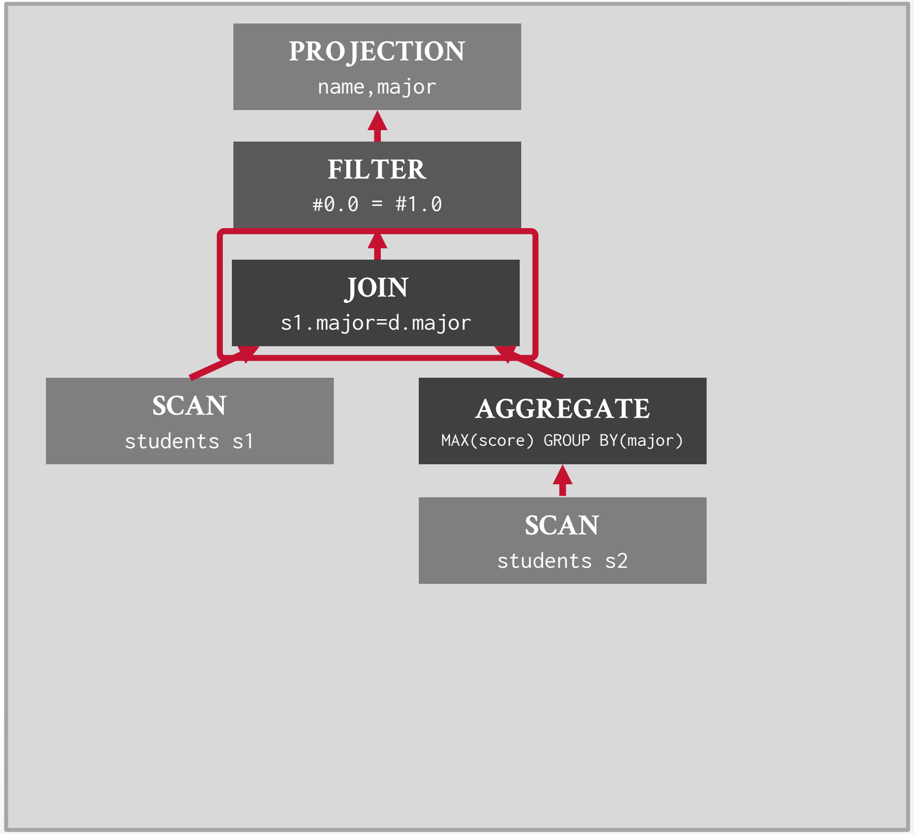
Source: Mark Raasveldt

FLATTENING CORRELATED QUERIES

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

Remove duplicate elimination scan entirely.

Remove the filter above the new join.

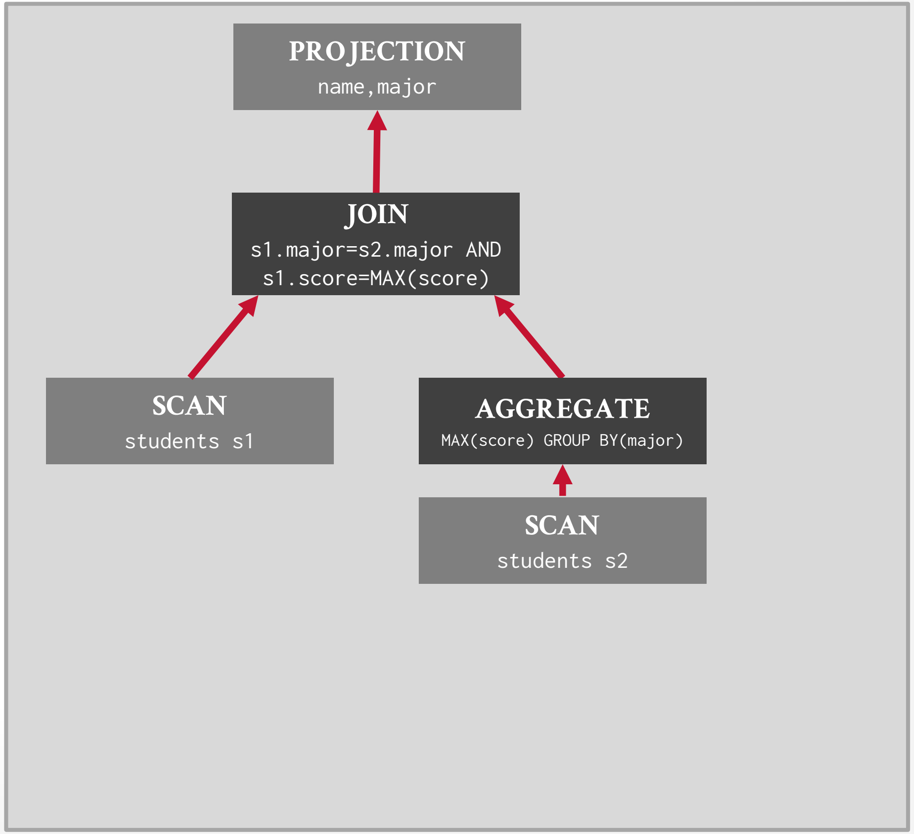


FLATTENING CORRELATED QUERIES

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

Remove duplicate elimination scan entirely.

Remove the filter above the new join.



Source: Mark Raasveldt

PARTING THOUGHTS

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

All the optimizer strategies we discussed assume that the optimizer has one shot at choosing a plan.

But what happens if the DBMS discovers that the cost estimates don't match reality when it starts processing data?

PARTING THOUGHTS

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

All the optimizer strategies we discussed assume that the optimizer has one shot at choosing a plan.

But what happens if the DBMS discovers that the cost estimates don't match reality when it starts processing data?

NEXT CLASS

Adaptive Query Optimization