15-721 ADVANCED DATABASE SYSTEMS

Lecture #15 – Optimizer Implementation (Part II)

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

Cascades
Orca
Project #3 Topics

QUERY OPTIMIZATION

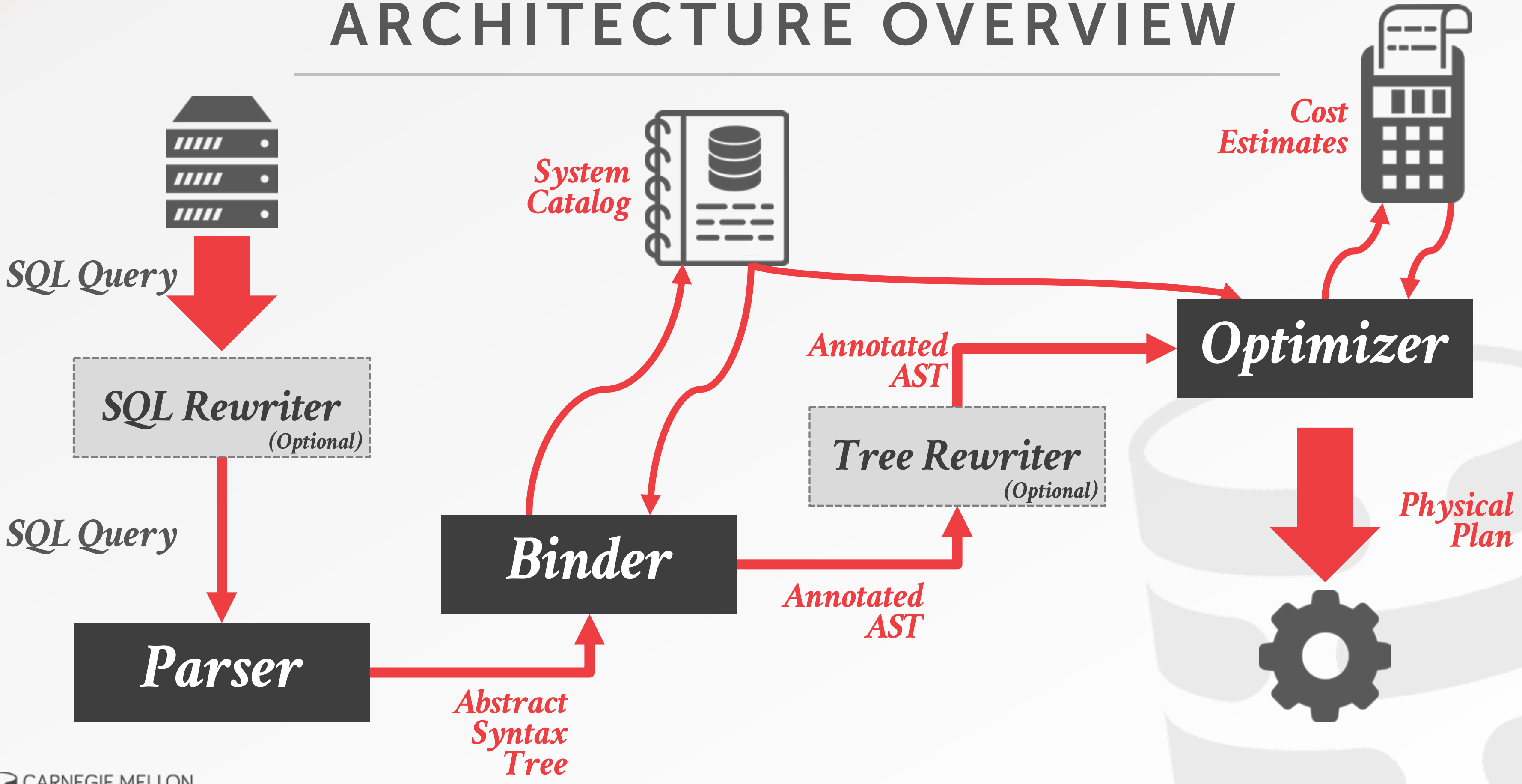
For a given query, find a **correct** execution plan that has the lowest "cost".

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.

ARCHITECTURE OVERVIEW



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

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.

POSTGRES OPTIMIZER

Imposes a rigid workflow for query optimization:
→ First stage performs initial rewriting with heuristics
→ It then executes a cost-based search to find optimal join ordering.
→ Everything else is treated as an "add-on".
→ Then recursively descends into sub-queries.

Difficult to modify or extend because the ordering has to be preserved.

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.

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.

Graefe

Example: Academic prototypes

THE VOLCANO OPTIMIZER GENERATOR: EXTENSIBILITY AND EFFICIENT SEARCH
ICDE 1993

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

EXAMPLE DATABASE

```
CREATE TABLE ARTIST (
  ID INT PRIMARY KEY,
  NAME VARCHAR(32)
);
```

```
CREATE TABLE ALBUM (
  ID INT PRIMARY KEY,
  NAME VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE APPEARS (
  ARTIST_ID INT
   ↪REFERENCES ARTIST(ID),
  ALBUM_ID INT
   ↪REFERENCES ALBUM(ID),
  PRIMARY KEY
   ↪(ARTIST_ID, ALBUM_ID)
);
```

VOLCANO 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
```

VOLCANO OPTIMIZER

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

```
ARTIST⨝APPEARS⨝ALBUM
ORDER-BY(ARTIST.ID)
```

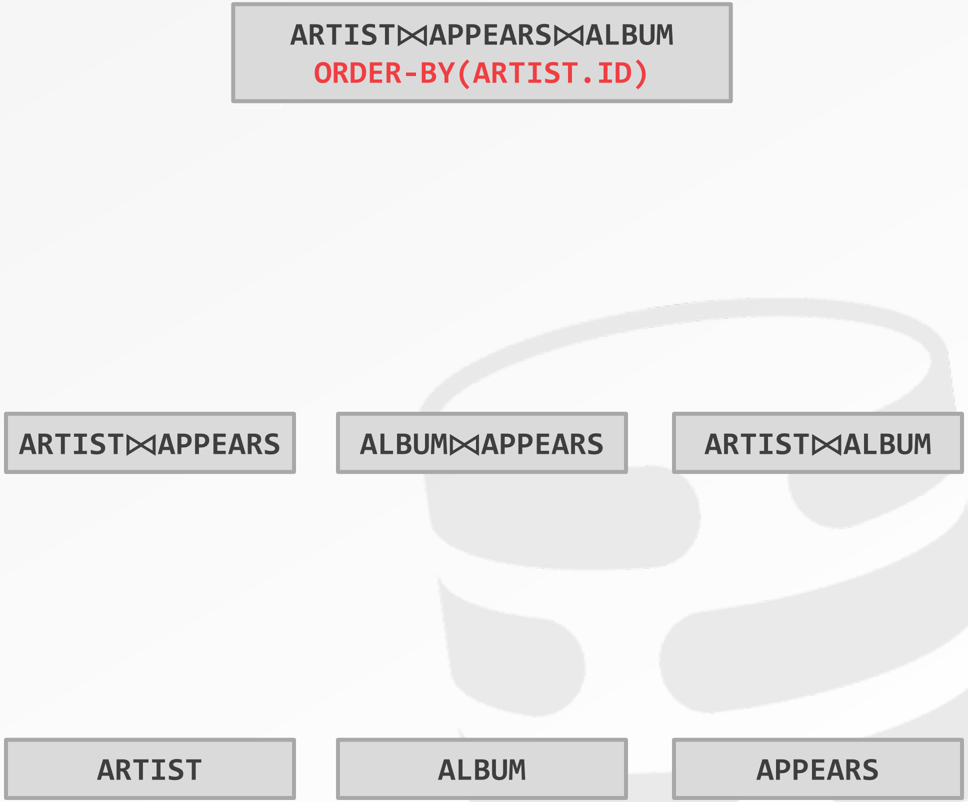
VOLCANO OPTIMIZER

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)



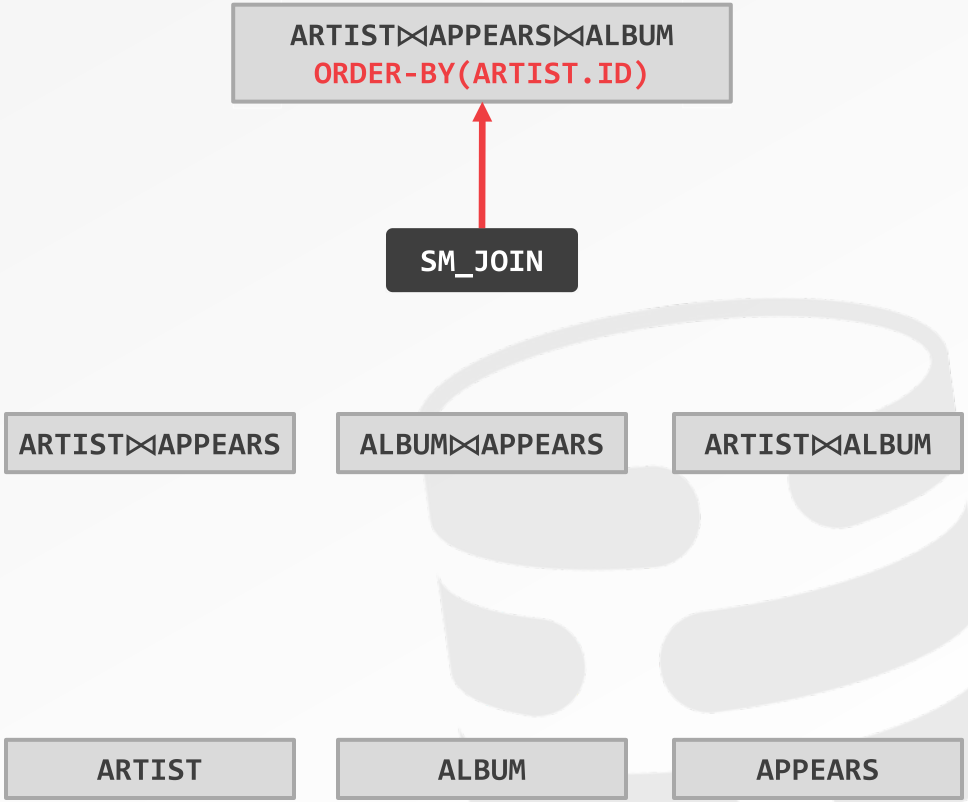
VOLCANO OPTIMIZER

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)

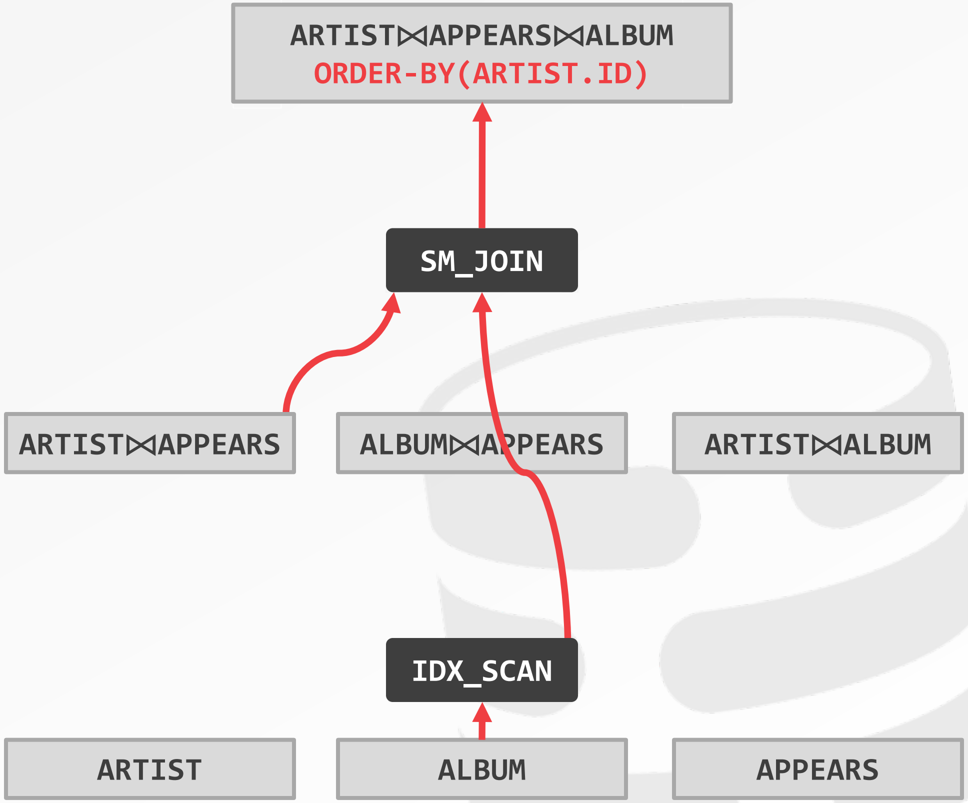


VOLCANO OPTIMIZER

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)



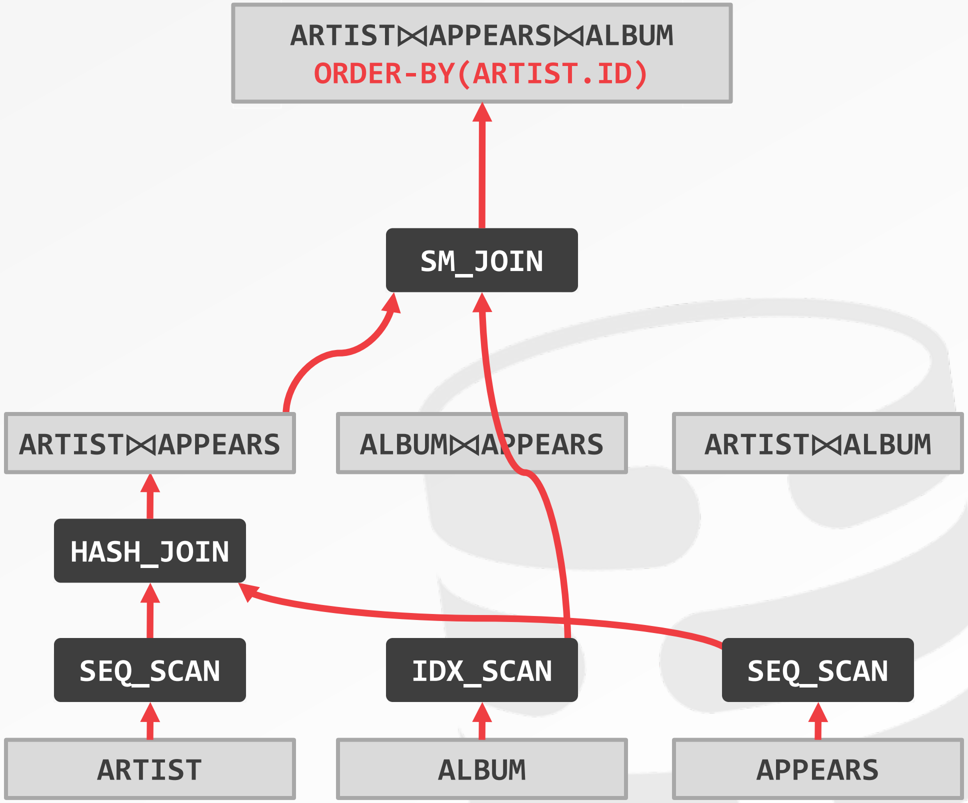
VOLCANO OPTIMIZER

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)



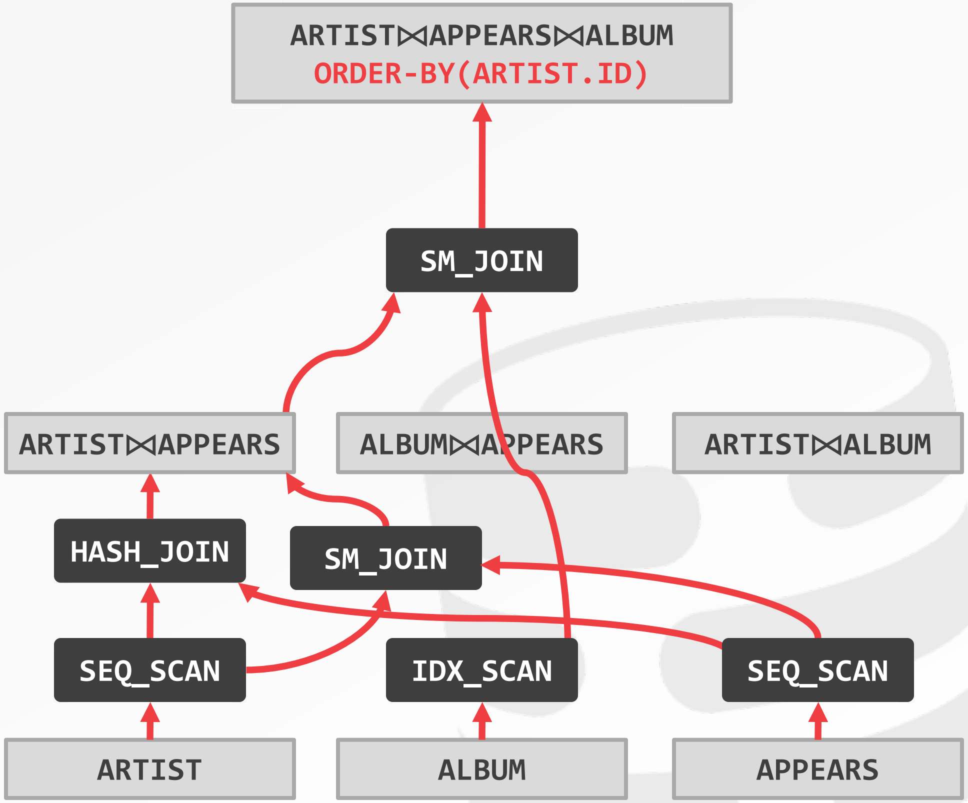
VOLCANO OPTIMIZER

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)



VOLCANO OPTIMIZER

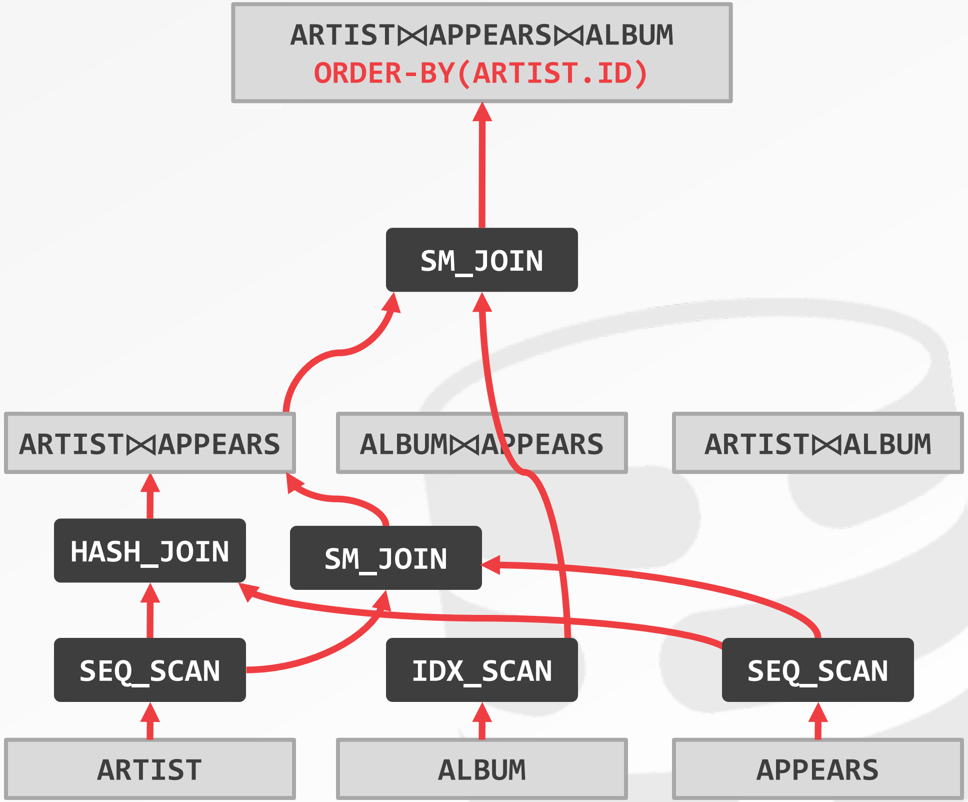
Start with a logical plan of what we want the query to be.

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

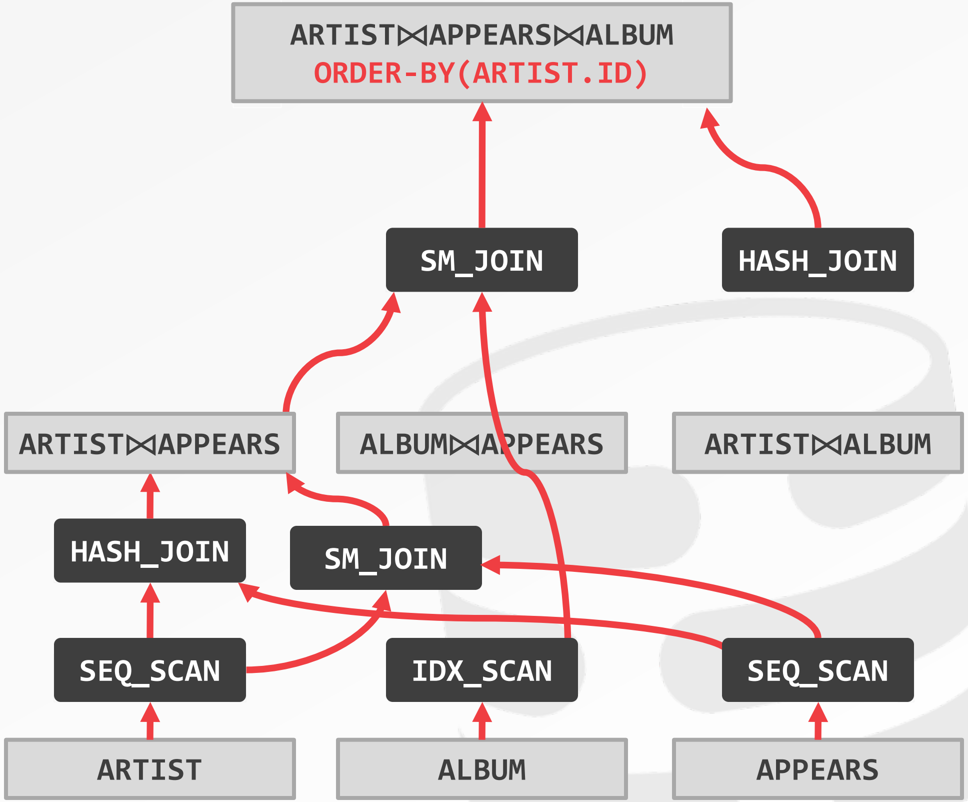
Start with a logical plan of what we want the query to be.

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

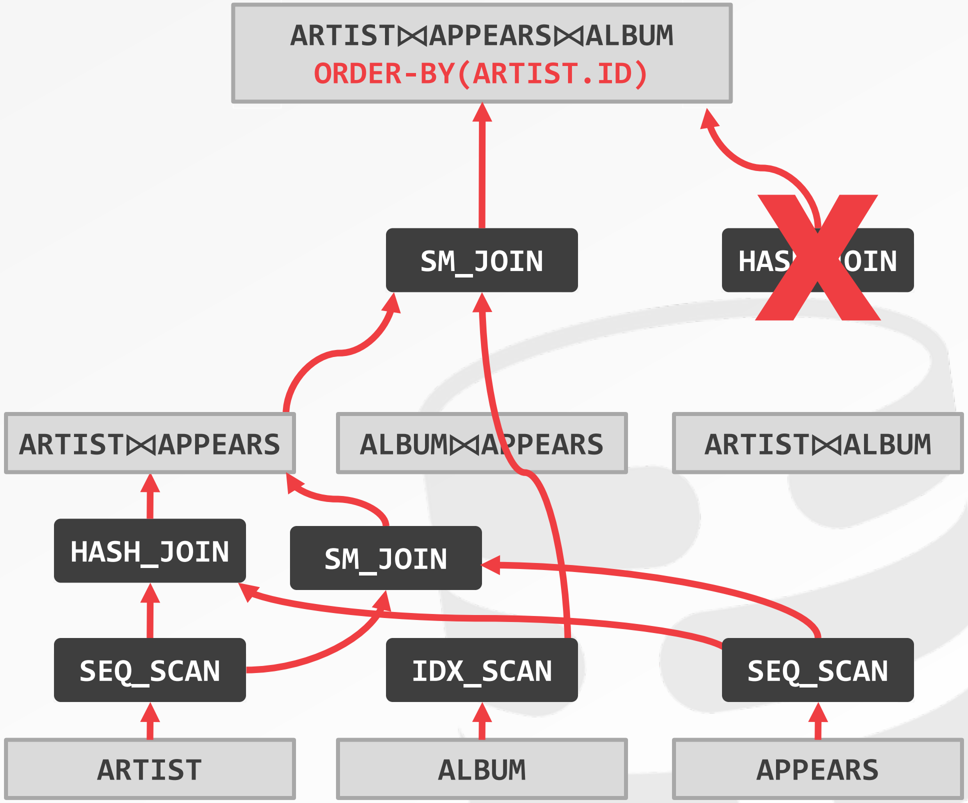
Start with a logical plan of what we want the query to be.

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.



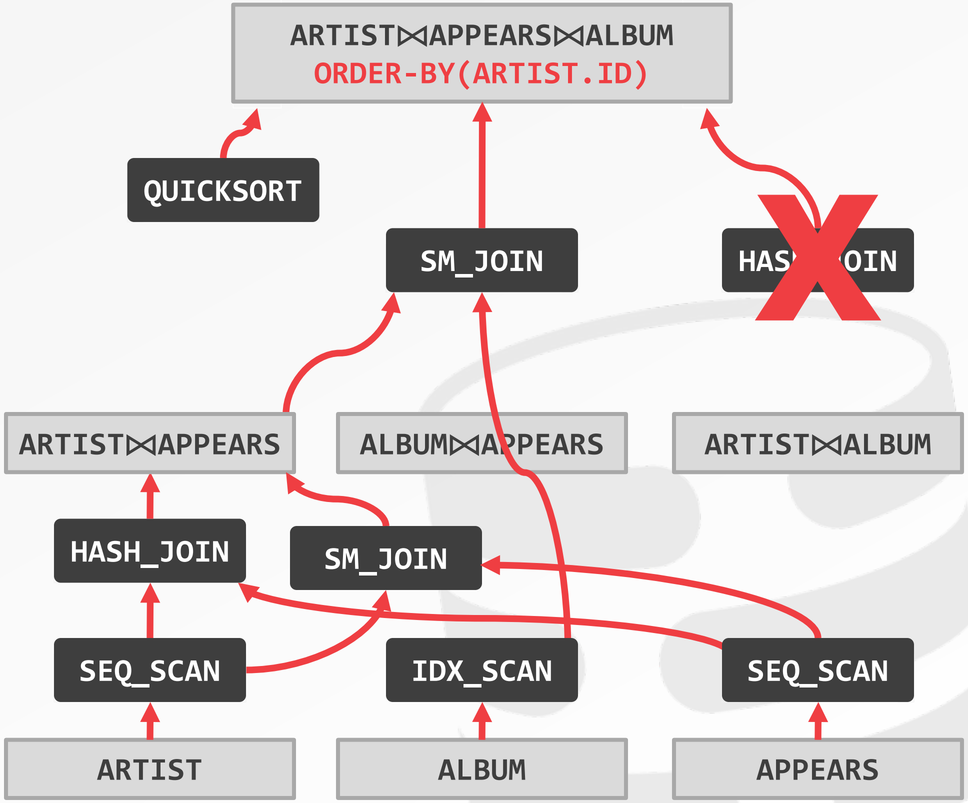
VOLCANO OPTIMIZER

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

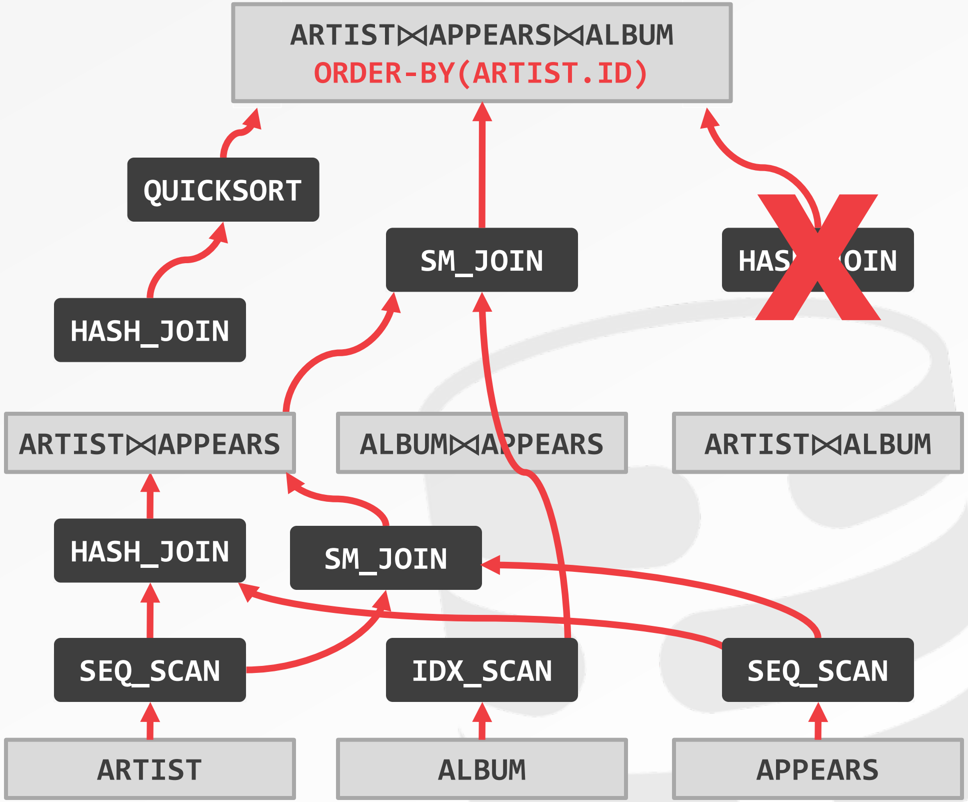
Start with a logical plan of what we want the query to be.

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

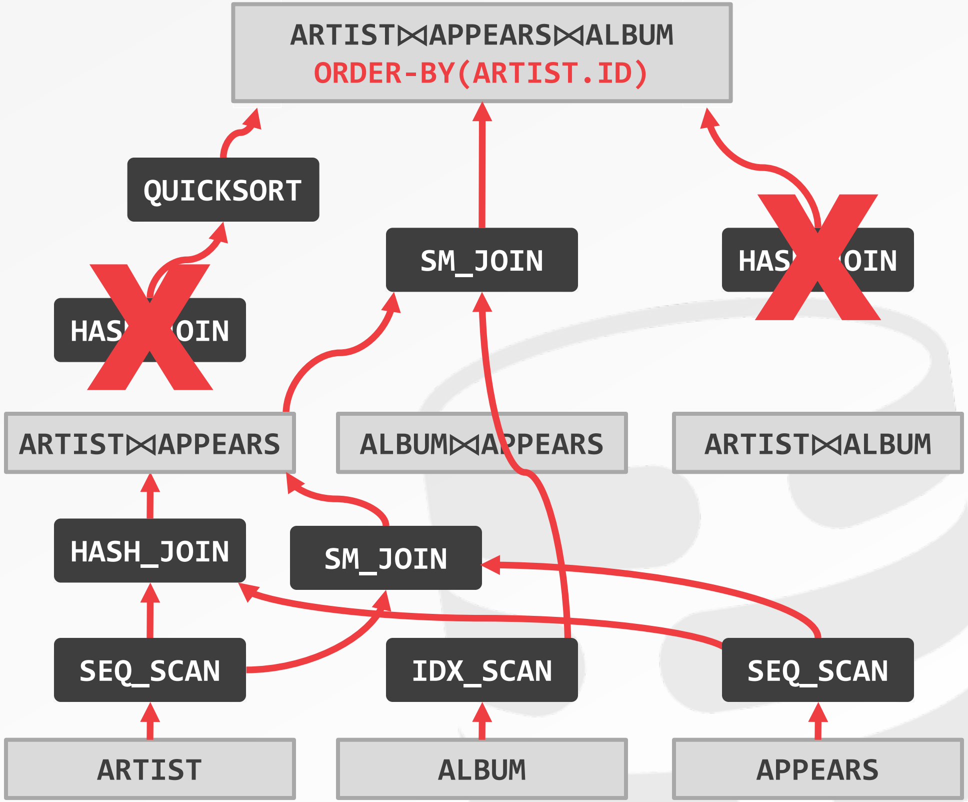
Start with a logical plan of what we want the query to be.

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

The optimizer needs to enumerate all possible transformations without repeating.

Go from logical to physical plan as fast as possible, then try alternative plans.

→ Use a top-down rules engine that performs branch-and-bound pruning.
→ Use memoization to cache equivalent operators.

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 Volcano query optimizer.

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

Graefe

Example: SQL Server, Greenplum's Orca, Apache Calcite, and Peloton (WIP).

THE CASCADES FRAMEWORK FOR QUERY OPTIMIZATION
IEEE Data Engineering Bulletin 1995

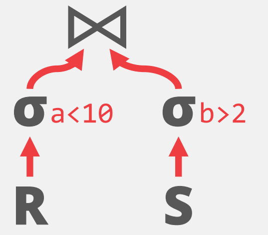
MEMO TABLE

Stores all previously explored alternatives in a compact graph structure.

Equivalent operator trees and their corresponding plans are stored together in groups.

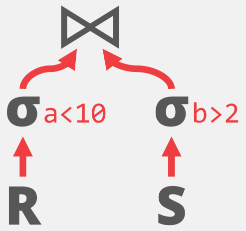
Provides memoization, duplicate detection, and property + cost management.

MEMO TABLE




```
1.  SELECT(a<10,b>2)
2.  JOIN(x=y,  ,  )
3.  ...
```

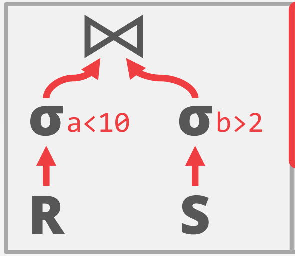
MEMO TABLE



1. SELECT(a<10,b>2)
2. JOIN(x=y, ,)
3. ...

Group

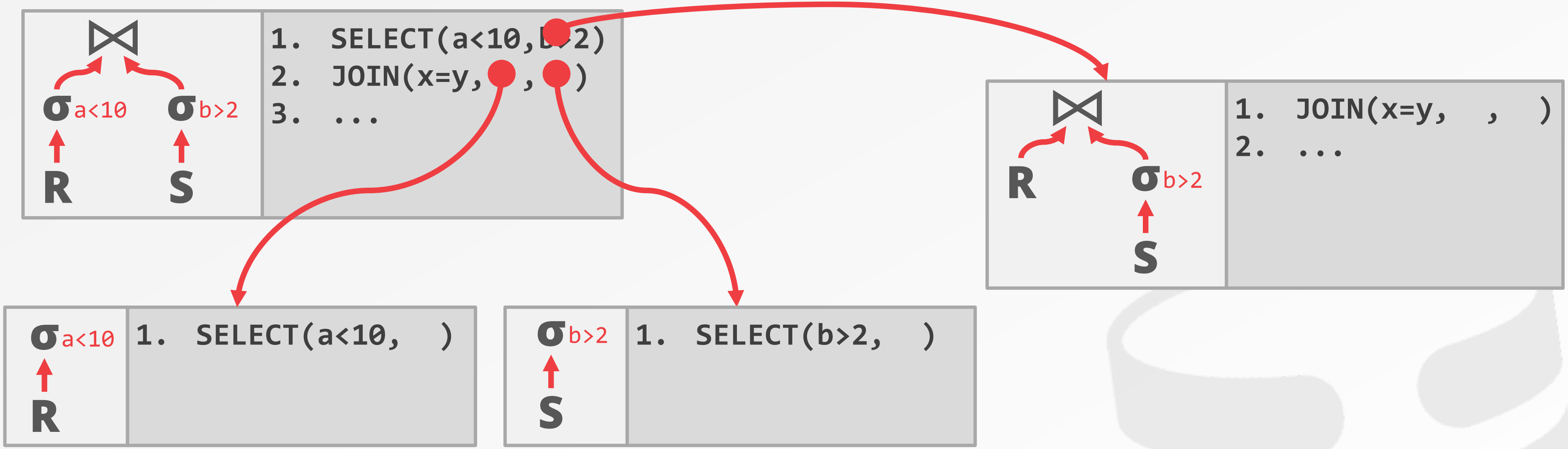
MEMO TABLE




```
1.  SELECT(a<10,b>2)
2.  JOIN(x=y,  ,  )
3.  ...
```

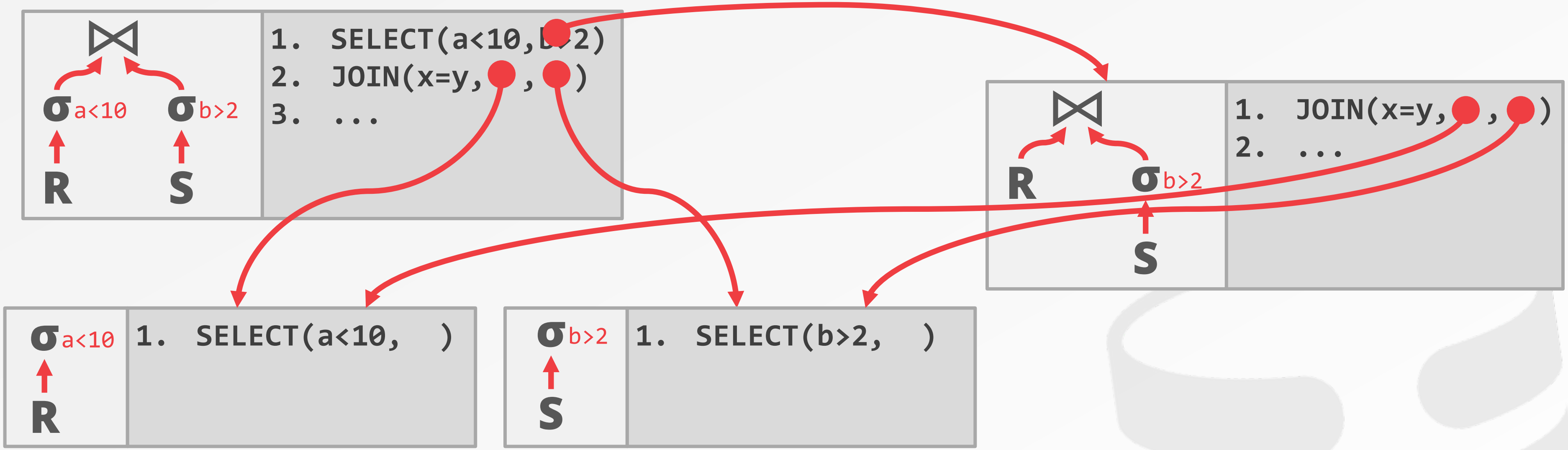
Expressions

MEMO TABLE



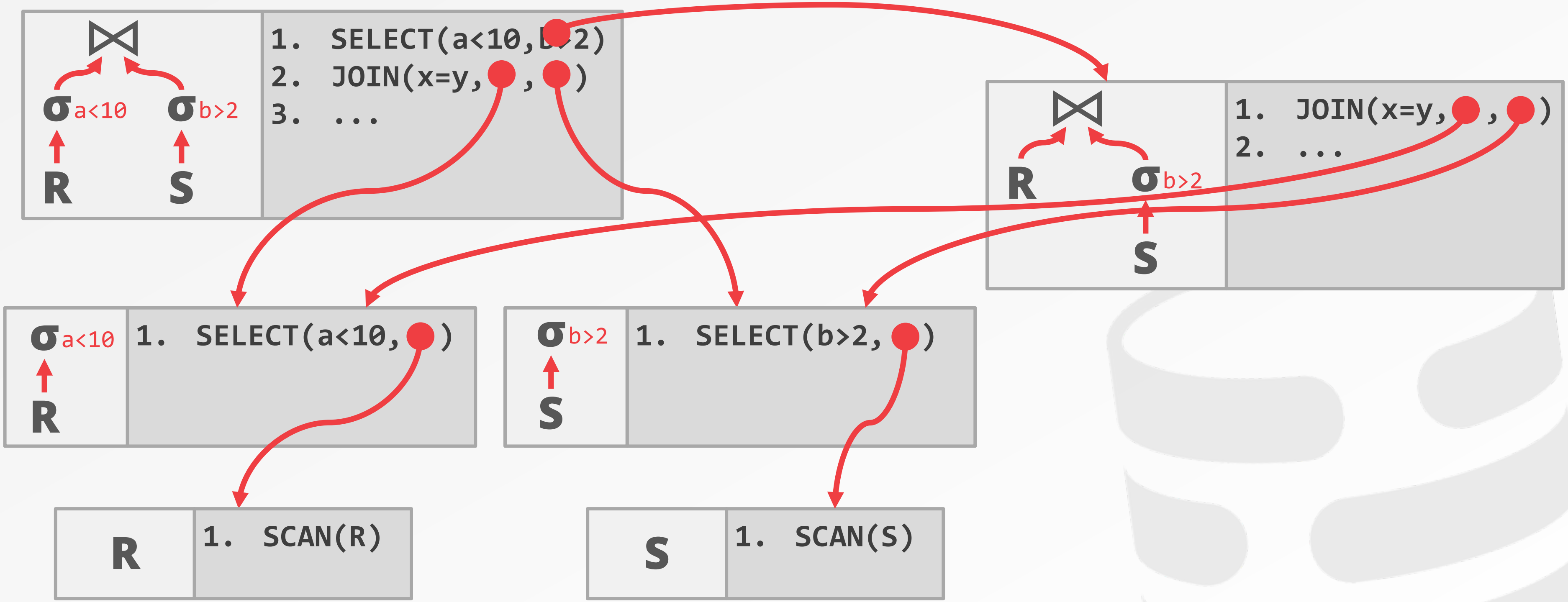
⋈
σ a<10
σ b>2
R
S
1. SELECT(a<10,b>2)
2. JOIN(x=y, ,)
3. ...
⋈
R
σ b>2
S
1. JOIN(x=y, ,)
2. ...
σ a<10
R
1. SELECT(a<10,)
σ b>2
S
1. SELECT(b>2,)

MEMO TABLE



σa<10
σb>2
R
S
1. SELECT(a<10,b>2)
2. JOIN(x=y, ,)
3. ...
R
σb>2
S
1. JOIN(x=y, ,)
2. ...
σa<10
R
1. SELECT(a<10,)
σb>2
S
1. SELECT(b>2,)

MEMO TABLE


⋈
σ a<10
σ b>2
R
S
1. SELECT(a<10,b>2)
2. JOIN(x=y, ,)
3. ...
⋈
R
σ b>2
S
1. JOIN(x=y, ,)
2. ...
σ a<10
R
1. SELECT(a<10,)
σ b>2
S
1. SELECT(b>2,)
R
1. SCAN(R)
S
1. SCAN(S)

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.

→ We will see a better way to evaluate predicates when we talk about query compilation.

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.

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.

PARTING THOUGHTS

"Query optimization is not rocket science. When you flunk out of query optimization, we make you go build rockets." – *David DeWitt*

The research literature suggests that there is no difference in quality between bottom-up vs. top-down search strategies.

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

PROJECT #3

Group project to implement some substantial component or feature in a DBMS.

Projects should incorporate topics discussed in this course as well as from your own interests.

Each group must pick a project that is unique from their classmates.

PROJECT #3

Project deliverables:
→ Proposal
→ Project Update
→ Code Review
→ Final Presentation
→ Code Drop

PROJECT #3 – PROPOSAL

Five minute presentation to the class that discusses the high-level topic.

Each proposal must discuss:
→ What files you will need to modify.
→ How you will test whether your implementation is correct.
→ What workloads you will use for your project.

PROJECT #3 – STATUS UPDATE

Five minute presentation to update the class about the current status of your project.

Each presentation should include:
→ Current development status.
→ Whether your plan has changed and why.
→ Anything that surprised you during coding.

PROJECT #3 – CODE REVIEW

Each group will be paired with another group and provide feedback on their code.

There will be two separate code review rounds.

Grading will be based on participation.

PROJECT #3 – FINAL PRESENTATION

10 minute presentation on the final status of your project during the scheduled final exam.

You'll want to include any performance measurements or benchmarking numbers for your implementation.

Demos are always hot too…

PROJECT #3 – CODE DROP

A project is **not** considered complete until:

→ The code can merge into the master branch without any conflicts.
→ All comments from code review are addressed.
→ The project includes test cases that correctly verify that implementation is correct.
→ The group provides documentation in both the source code and in separate Markdown files.

We will select the merge order randomly.

PROJECT TOPICS

Query Optimizer
System Catalogs
Mat. Views & Triggers
Constraints
User-defined Functions
LLVM Transactions

Partial Compilation
Multi-Threaded Queries
Database Compression
Alternative Protocols
Statistics + Sampling
Alternative Storage

PROJECT TOPICS

- Query Optimizer
- System Catalogs
- Mat. Views & Triggers
- Constraints
- User-defined Functions
- LLVM Transactions
- Partial Compilation
- Multi-Threaded Queries
- Database Compression
- Alternative Protocols
- Statistics + Sampling
- Alternative Storage

MATERIALIZED VIEWS & TRIGGERS

A materialized view is like a view that is updated whenever its underlying table is updated. Triggers are used to invoke changes.

Project: Implement support for incremental updates to mat. views and triggers in Peloton.

→ Will need to leverage Postgres' catalog infrastructure and then populate new data structures.

CONSTRAINTS

Constraints are important feature in DBMSs to ensure database integrity.

Project: Implement support for enforcing integrity constraints in Peloton.

→ Will want to start with simple constraints first.
→ Final goal will be to implement foreign key constraints.

USER-DEFINED FUNCTIONS

UDFs allow the developer to implement complex logic for evaluating tuples.

Project: Implement support for UDFs in Peloton for all query types.
→ Will need to extend Peloton's expression sub-system.
→ Can borrow code for Postgres API support.

LLVM TRANSACTIONS

We are porting over our new LLVM-based execution engine to the master branch. Currently only supports **SELECT** statements.

Project: Add support for generating compiled query plans for **INSERT**, **UPDATE**, and **DELETE**.

→ Will need to add support for MemSQL-style parameterization and plan caching.

→ Need to make sure that queries check tuple visibility.

MULTI-THREADED QUERIES

Peloton currently only uses a single worker thread per txn/query.

Project: Implement support for intra-query parallelism with multiple threads.

→ Will need to implement this to work with the new LLVM execution engine.
→ **Bonus:** Add support for NUMA-aware data placement. Will need to update internal catalog.

DATABASE COMPRESSION

Compression enables the DBMS to use less space to store data and potentially process less data per query.

Project: Implement different compression schemes for table storage.

→ Delta encoding, Dictionaries
→ Will need to implement new query operators that can operate directly on this data.
→ **Bonus:** Implement the ability to automatically determine what scheme to use per tile.

ALTERNATIVE PROTOCOLS

Add support for different ways to connect to Peloton and ingest/query data.

Examples: Kafka, Memcached

Project: Implement these APIs directly inside of Peloton and enable it to read/write directly to in-memory data.

→ Need to overhaul the client connection handling code.
→ GET/PUT can be implemented as a single-query txn with prepared statements.

STATISTICS + SAMPLING

We currently do not maintain any statistics about the database. This is needed for our new query optimizer.

Project: Implement a mechanism for collecting statistics about tables for the query optimizer.
→ Can choose lazy or eager sampling.
→ Add this data to the catalog.
→ **Bonus:** Implement a new cost model.

ALTERNATIVE STORAGE

Peloton stores all data in its in-memory storage engine. We want to extend it to support external sources (e.g., Foreign Data Wrappers).

Project: Implement a new storage API that can support external storage managers.

→ Start with an embedded DBMS (LevelDB, RocksDB).
→ Will need to update the catalog to know what's available.
→ **Bonus:** Implement anti-caching / data movement.

TESTING

We plan to expand Peloton's SQL-based regression test suite to check that your project does not break high-level functionalities.

Every group is going to need to implement their own unit tests for their code.

COMPUTING RESOURCES

Use the same machines as your other projects.
→ Dual-socket Xeon E5-2620 (6 cores / 12 threads)
→ 128 GB DDR4

Let me know if you think you need special hardware.

OLTP-BENCH

We already have a full-featured benchmarking framework that you can use for your projects.

It includes 15 ready to execute workloads
→ OLTP: TPC-C, TATP, YCSB, Wikipedia
→ OLAP: CH-Benchmark, TPC-H

http://oltpbenchmark.com/

PROJECT #3 PROPOSALS

Each group will make a **5** minute presentation about their project topic proposal to the class on **Tuesday March 21st**.

I am able during Spring Break for additional discussion and clarification of the project idea.

NEXT CLASS

Cost Models

Working in a large code base

Extra Credit Assignment