Lecture 21: Design Decisions + Search Strategies

Recap

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

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)
- We will discuss this more next week. . .

Today's Agenda

- Design Decisions
- Optimization Search Strategies
- Optimizer Generators

Design Decisions

Design Decisions

- Optimization Granularity
- Optimization Timing
- Prepared Statements
- Plan Stability
- Search Termination
- Search Strategy – Important

Optimization Granularity

- **Choice 1: Single Query**
  - Much smaller search space.
  - DBMS (usually) does not reuse results across queries.
  - To account for resource contention, the cost model must consider what is currently running.
- **Choice 2: Multiple Queries**
  - More efficient if there are many similar queries.
  - Search space is much larger.
  - Useful for data / intermediate result sharing.

Optimization Timing

- **Choice 1: Static Optimization**
  - Select the best plan prior to execution.
  - Plan quality is dependent on cost model accuracy.
  - Can amortize over executions with **prepared statements**.
- **Choice 2: Dynamic Optimization**
  - Select operator plans on-the-fly as queries execute.
  - Will have re-optimize for multiple executions.
  - Difficult to implement/debug (non-deterministic)
- **Choice 3: Adaptive Optimization**
  - Compile using a static algorithm.
  - If the estimate errors > threshold, change or re-optimize.

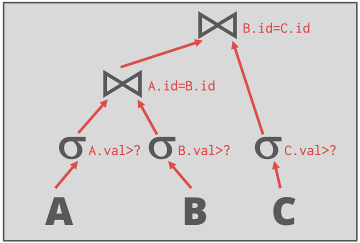
Prepared Statements

```
SELECT A.id, B.val
  FROM A, B, C
 WHERE A.id = B.id
   AND B.id = C.id
   AND A.val > 100
   AND B.val > 99
   AND C.val > 5000
```

Prepared Statements

```
PREPARE myQuery(int, int, int) AS
  SELECT A.id, B.val
    FROM A, B, C
   WHERE A.id = B.id
     AND B.id = C.id
     AND A.val > ?
     AND B.val > ?
     AND C.val > ?
```

```
EXECUTE myQuery(100, 99, 5000);
```



What should be the join order for **A**, **B**, and **C**?

Prepared Statements

- **Choice 1: Reuse Last Plan**
  - Use the plan generated for the previous invocation.
- **Choice 2: Re-Optimize**
  - Rerun optimizer each time the query is invoked.
  - Tricky to reuse existing plan as starting point.
- **Choice 3: Multiple Plans**
  - Generate multiple plans for different values of the parameters (e.g., buckets).
- **Choice 4: Average Plan**
  - Choose the average value for a parameter and use that for all invocations.

Plan Stability

- **Choice 1: Hints**
  - Allow the DBA to provide hints to the optimizer.
- **Choice 2: Fixed Optimizer Versions**
  - Set the optimizer version number and migrate queries one-by-one to the new optimizer.
- **Choice 3: Backwards-Compatible Plans**
  - Save query plan from old version and provide it to the new DBMS.

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 (*e.g.*, search depth in MySQL's optimizer).
- **Approach 3: Exhaustion**
  - Stop when there are no more enumerations of the target plan. Usually done per group.

Optimization Search Strategies

Optimization Search Strategies

- Heuristics
- Heuristics + Cost-based Join Order Search
- Randomized Algorithms
- Stratified Search
- Unified Search

Heuristic-Based Optimization

- Define static rules that transform logical operators to a physical plan.
  - Perform most restrictive selection early
  - Perform all selections before joins
  - Predicate/Limit/Projection pushdowns
  - Join ordering based on cardinality
- **Examples:** INGRES and Oracle (until mid 1990s).
- Reference

Example Database

```
CREATE TABLE APPEARS (
  ARTIST_ID INT
   REFERENCES ARTIST(ID),
  ALBUM_ID INT
   REFERENCES ALBUM(ID),
  PRIMARY KEY
   (ARTIST_ID, ALBUM_ID)
);
CREATE TABLE ARTIST (
  ID INT PRIMARY KEY,
  NAME VARCHAR(32)
);
CREATE TABLE ALBUM (
  ID INT PRIMARY KEY,
  NAME VARCHAR(32) UNIQUE
);
```

Ingres Optimizer

Retrieve the names of people that appear on Andy's mixtape

```
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"
```

Q1

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

Q2

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

Step #1: Decompose into single-value queries

Ingres Optimizer

Retrieve the names of people that appear on Andy's mixtape

```
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"
```

Step #1: Decompose into single-value queries

Q1

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

Q3

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

Q4

```
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

```
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"
```

ALBUM_ID
9999

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

Step #1: Decompose into single-value queries

Step #2: Substitute the values from Q1→Q3→Q4

Q4

```
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

```
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"
```

ALBUM_ID
9999

ARTIST_ID
123
456

Step #1: Decompose into single-value queries

Step #2: Substitute the values from Q1→Q3→Q4

Q4

```
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

```
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"
```

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 Q1→Q3→Q4

Ingres Optimizer

Retrieve the names of people that appear on Andy's mixtape

```
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"
```

ALBUM_ID
9999

ARTIST_ID
123
456

NAME
Mozart

NAME
Beethoven

Step #1: Decompose into single-value queries

Step #2: Substitute the values from Q1→Q3→Q4

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 Join Search

- Use static rules to perform initial optimization.
- Then use **dynamic programming** to determine the best join order for tables.
  - First cost-based query optimizer
  - **Bottom-up planning** (forward chaining) using a divide-and-conquer search method
- **Examples:** System R, early IBM DB2, most open-source DBMSs.
- Reference

Pat Selinger

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

```
\item SELECT ARTIST.NAME
\item   FROM ARTIST, APPEARS, ALBUM
\item  WHERE ARTIST.ID=APPEARS.ARTIST_ID
\item    AND APPEARS.ALBUM_ID=ALBUM.ID
\item    AND ALBUM.NAME= “Andy's OG Remix”
\item  ORDER BY ARTIST.ID      --- Ordered based on the artist id.
```

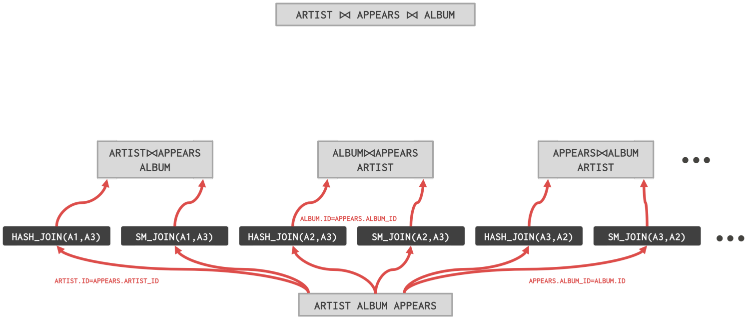
- Step 1: Choose the best access paths to each table
- Step 2: Enumerate all possible join orderings for tables
- Step 3: Determine the join ordering with the lowest cost

System R Optimizer

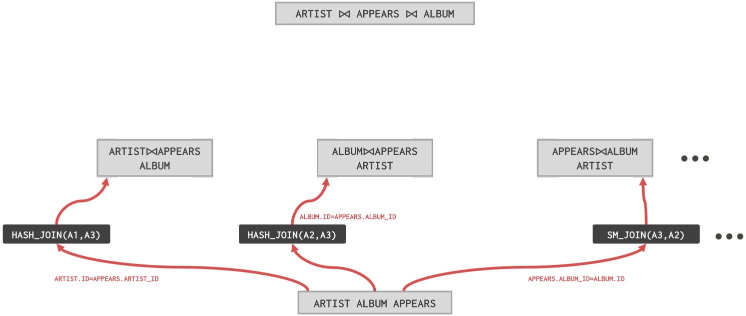
ARTIST:	Sequential Scan
APPEARS:	Sequential Scan
ALBUM:	Index Look-up on NAME

- ARTIST $\bowtie$ APPEARS $\bowtie$ ALBUM
- APPEARS $\bowtie$ ALBUM $\bowtie$ ARTIST
- ALBUM $\bowtie$ APPEARS $\bowtie$ ARTIST
- APPEARS $\bowtie$ ARTIST $\bowtie$ ALBUM
- ARTIST $\times$ ALBUM $\bowtie$ APPEARS
- ALBUM $\times$ ARTIST $\bowtie$ APPEARS
- ...
- ...

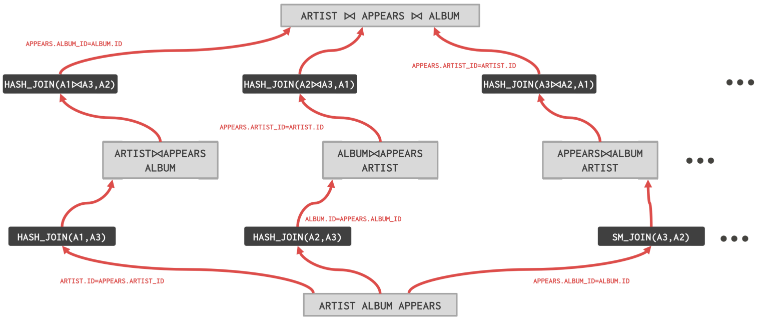
System R Optimizer



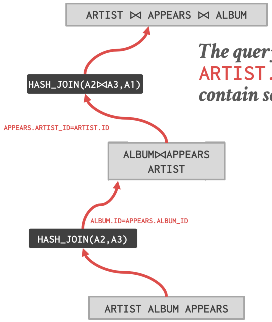
System R Optimizer



System R Optimizer



System R Optimizer



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

Top-down vs. Bottom-up

- **Top-down Optimization**
  - Start with the outcome that you want, 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, Hyper

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.
  - Asumptions about inputs are baked into the code (not elegant).
- Difficult to modify or extend because the ordering must be preserved.

Heuristics + Cost-based Join 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).

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.

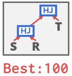
Simulated Annealing

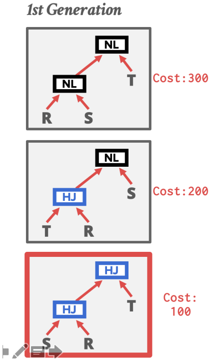
- Start with a query plan that is generated using the heuristic-only approach.
- Compute random permutations of operators (e.g., swap the join order of two tables)
  - Always accept a change that reduces cost
  - Only accept a change that increases cost with some probability.
  - Reject any change that violates correctness (e.g., sort ordering)
- Reference

Postgres Genetic Optimizer

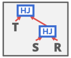
- More complicated queries use a **genetic algorithm** that selects join orderings (GEQO).
- At the beginning of each round, generate different variants of the query plan.
- Select the plans that have the lowest cost and permute them with other plans. Repeat.
  - The mutator function only generates valid plans.
- Postgres Documentation

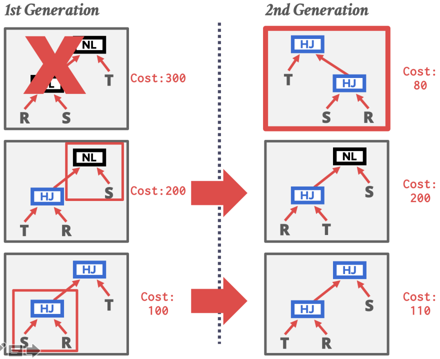
Postgres Optimizer



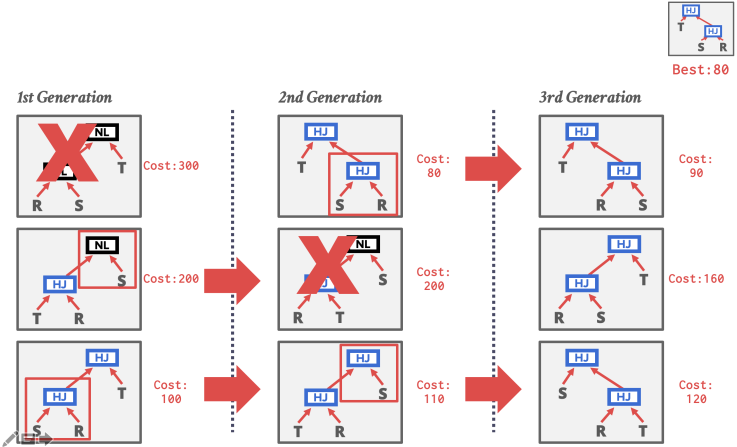


Postgres Optimizer





Postgres 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.
  - Must still implement correctness rules.

Optimizer Generators

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

- Framework to allow a DBMS implementer to write the **declarative rules** for optimizing queries.
  - Separate the **search strategy** from the data model.
  - Separate the **transformation rules** and logical operators from **physical rules** and physical operators.
- Implementation can be independent of the optimizer's search strategy.
- **Examples:** Starburst, Exodus, Volcano, Cascades, OPT++

Optimizer Generators

- Use a rule engine that allows transformations to modify the query plan operators.
- The physical properties of data is embedded with the operators themselves.
- **Choice 1: Stratified Search**
  - Planning is done in multiple stages
- **Choice 2: Unified Search**
  - Perform query planning all at once.

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 dynamic programming phase once query rewrite has completed.
- **Example:** Latest version of IBM DB2
- Reference

Guy Lohman

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.

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
- Reference

Goetz Graefe

Volcano Optimizer

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

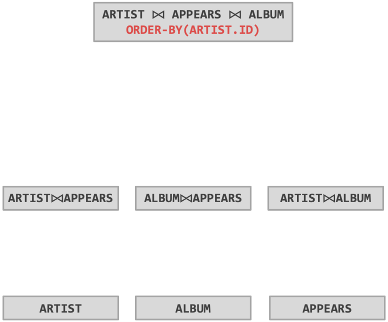
```
ARTIST ⋈ APPEARS ⋈ ALBUM
ORDER-BY(ARTIST.ID)
```

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)

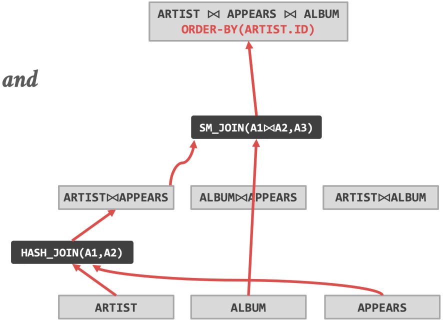


Volcano Optimizer

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)

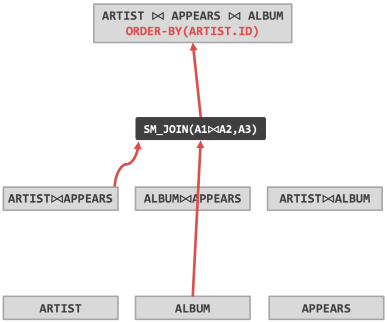


Volcano Optimizer

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)



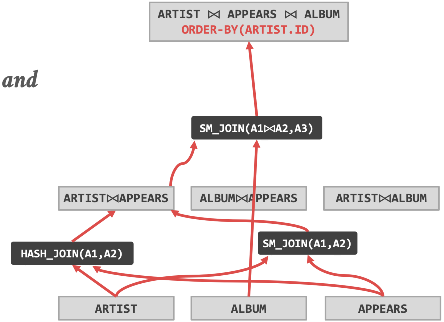
Volcano Optimizer

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)



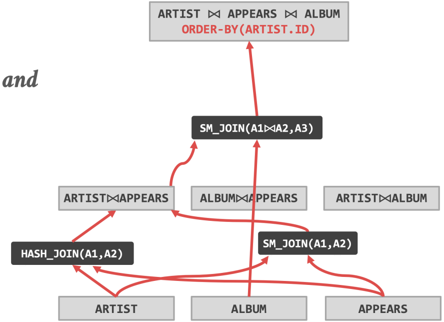
Volcano Optimizer

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)

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



Volcano Optimizer

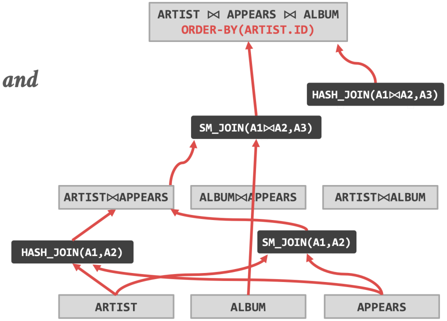
Start with a logical plan of what we want the query to be.

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.



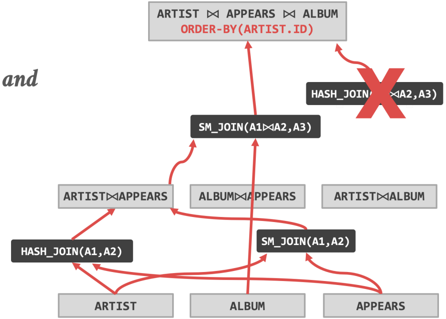
Volcano Optimizer

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.



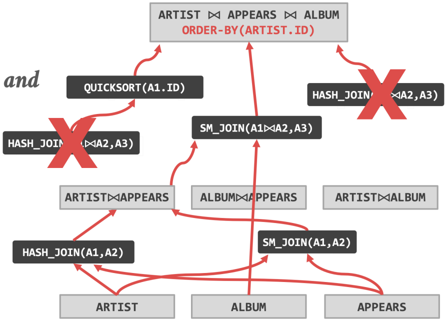
Volcano Optimizer

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

- **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.

Conclusion

Parting Thoughts

- Design decisions
  - Optimization Granularity
  - Optimization Timing
  - Prepared Statements
  - Plan Stability
  - Search Termination
  - Search Strategy – Important
- Query optimization is **non-trivial**
- This difficulty is why NoSQL systems didn't implement optimizers (at first).

Next Class

- Cascades