Lecture 22: Cascades Framework

Recap

Optimization Search 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.

Unified Search

- Unify the notion of both logical→logical and logical→physical transformations.
  - No need for separate stages because everything is transformations.
- This approach generates a lot more transformations so it makes heavy use of memoization to reduce redundant work.

Top-Down vs. Bottom-Up

- **Top-down Optimization**
  - ▶ Start with the final outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal.
  - ▶ Example: Volcano, Cascades
- **Bottom-up Optimization**
  - ▶ Start with nothing and then build up the plan to get to the final outcome that you want.
  - ▶ Examples: System R, Starburst

Today's Agenda

- Logical Query Optimization
- Physical Query Optimization
- Cascades Optimizer
- Dynamic Programming
- Case Studies

Logical Query Optimization

Logical Query Optimization

- Transform a logical plan into an equivalent logical plan using pattern matching rules.
- The goal is to increase the likelihood of enumerating the optimal plan in the search.
- Cannot compare plans because there is no cost model but can "direct" a transformation to a preferred side.

Logical Query Optimization

- Split Conjunctive Predicates
- Predicate Pushdown
- Replace Cartesian Products with Joins
- Projection Pushdown
- Reference

Split Conjunctive Predicates

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

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

Split Conjunctive Predicates

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```
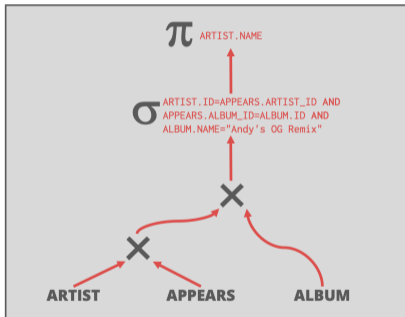
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.

Predicate Pushdown

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```
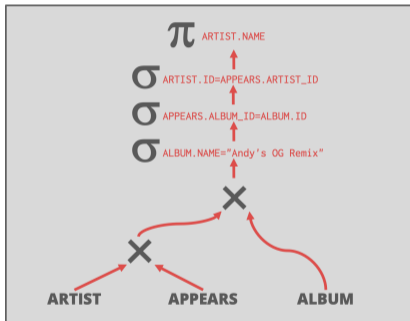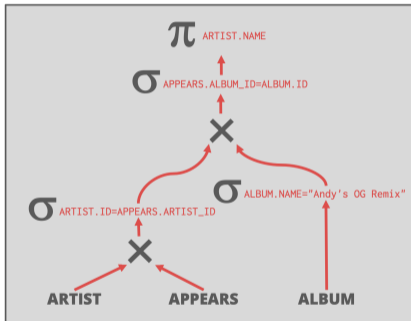
Move the predicate to the lowest point in the plan after Cartesian products.

Predicate Pushdown

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```
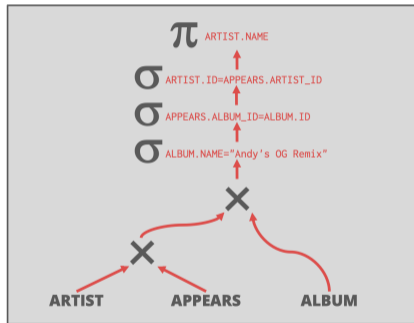
Move the predicate to the lowest
point in the plan after Cartesian
products.

Replace Cartesian Products with Joins

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```
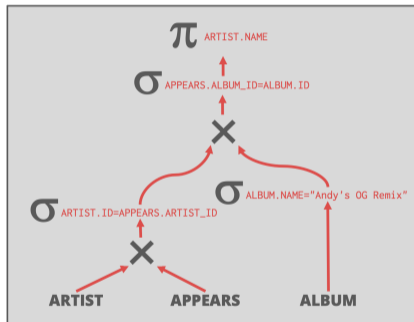
Replace all Cartesian Products with inner joins using the join predicates.

Replace Cartesian Products with Joins

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```
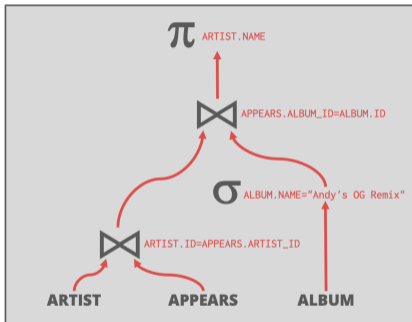
Replace all Cartesian Products with inner joins using the join predicates.

Projection Pushdown

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```
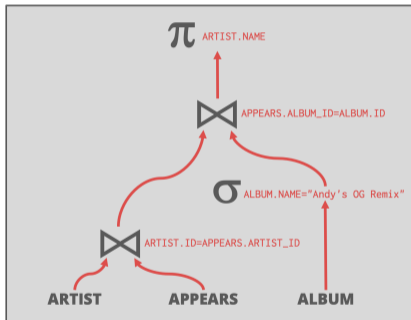
Eliminate redundant attributes before pipeline breakers to reduce materialization cost.
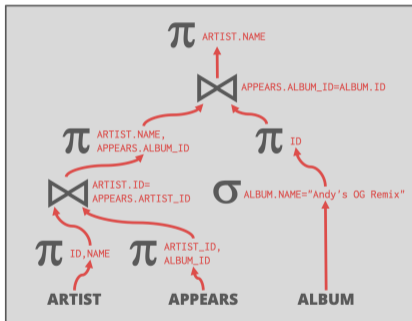
Projection Pushdown

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

Eliminate redundant attributes
before pipeline breakers to
reduce materialization cost.

Physical Query Optimization

Physical Query Optimization

- Transform a query plan's logical operators into physical operators.
  - ▶ Add more execution information
  - ▶ Select indexes / access paths
  - ▶ Choose operator implementations
  - ▶ Choose when to materialize (*i.e.*, temp tables).
- This stage must support **<u>cost model estimates</u>**.

Observation

- All the queries we have looked at so far have had the following properties:
  - ▶ Equi/Inner Joins
  - ▶ Simple join predicates that reference only two tables.
  - ▶ No cross products
- Real-world queries are much more complex:
  - ▶ Outer Joins
  - ▶ Semi-joins
  - ▶ Anti-joins

Reordering: Limitations

- No valid reordering is possible.
- The A⋈B operator is not commutative with B⋈C.
  - ▶ The DBMS does not know the value of B.val (may be NULL) until after computing the join with A.
- Reference

```
SELECT * FROM
  A LEFT OUTER JOIN B
  ON A.id = B.id
  FULL OUTER JOIN C
  ON B.val = C.id;
```

Plan Enumeration

- **Approach 1: Transformation**
  - ▶ Modify some part of an existing query plan to transform it into an alternative plan that is equivalent.
- **Approach 2: Generative**
  - ▶ Assemble building blocks to generate a query plan (similar to dynamic programming).
- Reference

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.
- Reference

Cascades Optimizer

- Object-oriented implementation of the Volcano query optimizer.
- Materialize transformations on the fly (rather than pre-generate them all at once).
- Unlike Volcano, restricts the set of transformations to constrain the search space.
- Supports simplistic expression re-writing through a direct mapping function rather than an **exhaustive search**.

Cascades Optimizer

Cascades Optimizer: Design Decisions

- Optimization tasks as data structures.
- Rules to place **property enforcers** (*e.g.*, sorting order).
- Ordering of transformations by priority. Dynamically adjust ordering as we traverse the search tree.
- Predicates are first class citizens (same as logical/physical operators).
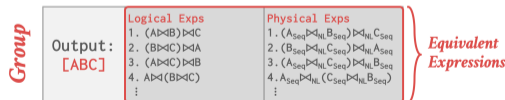
Cascades – Expressions

- An **expression** is an **operator** with zero or more input expressions.
- **Logical Expression**: $(A \bowtie B) \bowtie C$
- **Physical Expression**: $(A_{Seq} \bowtie_{HJ} B_{Seq}) \bowtie_{NL} C_{Idx}$

```sql
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON C.id = A.id;
```

Cascades – Groups

- A **group** is a set of **logically equivalent** logical and physical expressions that produce the same output.
  - ▶ All logical forms of an expression.
  - ▶ All physical expressions that can be derived from selecting the allowable physical operators for the corresponding logical forms.

Cascades – Multi-Expression

- Instead of explicitly instantiating all possible expressions in a group, the optimizer implicitly represents redundant expressions in a group as a **multi-expression**.
  - ▶ This reduces the number of transformations, storage overhead, and repeated cost estimations.
 - ▶ We can make decisions about whether to traverse [AB] first vs. [C] first.

Cascades – Rules

- A **rule** is a transformation of an expression to a logically equivalent expression.
    - ▶ **Transformation Rule:** Logical to Logical
    - ▶ **Implementation Rule:** Logical to Physical
- Each rule is represented as a pair of attributes:
    - ▶ **Pattern:** Defines the structure of the logical expression that can be applied to the rule.
    - ▶ **Substitute:** Defines the structure of the result after applying the rule.
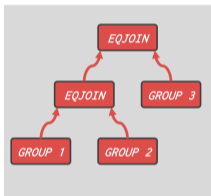
Cascades – Rules

Pattern
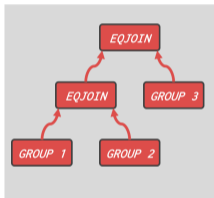
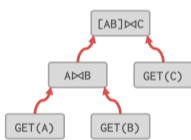- ■ *Group*
- ▢ *Logical Expr*
- ■ *Physical Expr*

Matching Plan

Cascades – Rules

Pattern
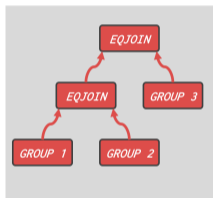
Transformation Rule
Rotate Left-to-Right

Matching Plan

■ *Group*
□ *Logical Expr*
■ *Physical Expr*

Cascades – Rules

Pattern

Transformation Rule
Rotate Left-to-Right

Matching Plan

Implementation Rule
EQJOIN→SORT MERGE

■ *Group*
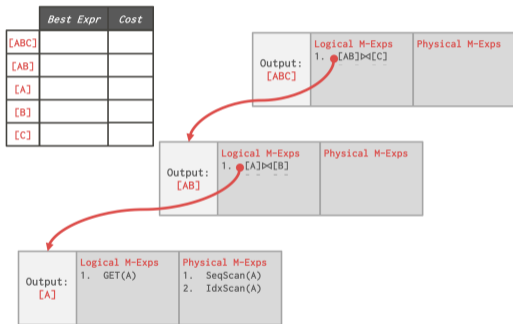□ *Logical Expr*
■ *Physical Expr*
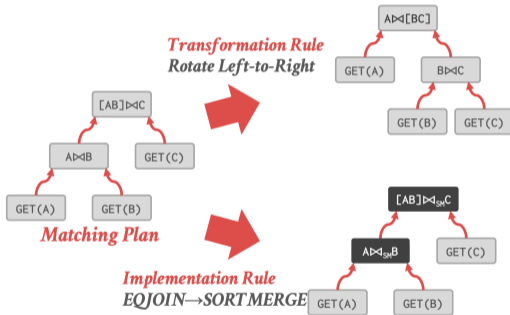
Cascades – Memo Table

- Stores all previously explored alternatives in a compact graph structure / hash table.
- Equivalent operator trees and their corresponding plans are stored together in groups.
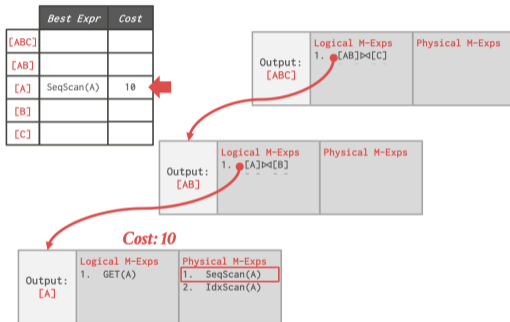- Provides memoization, duplicate detection, and property + cost management.

Principle of Optimality

- Every sub-plan of an optimal plan is itself optimal.
- This allows the optimizer to restrict the search space to a smaller set of expressions.
  - ▶ The optimizer never has to consider a plan containing sub-plan P1 that has a greater cost than equivalent plan P2 with the same physical properties.
  - ▶ Reference

Cascades – Memo Table

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

Output: [ABC]

Logical M-Exps
1. [AB]⋈[C]

Physical M-Exps

Output: [AB]

Logical M-Exps
1. [A]⋈[B]

Physical M-Exps

Output: [A]

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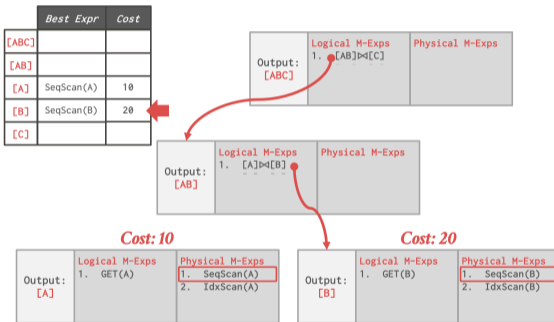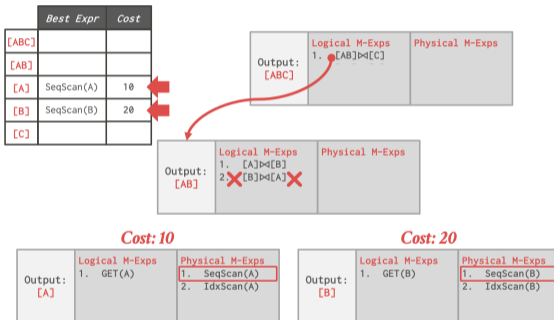1. GET(A)
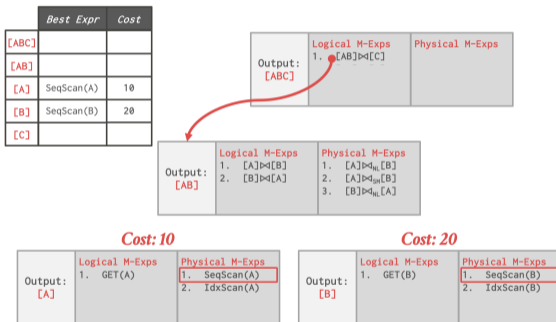
Physical M-Exps
1. SeqScan(A)
2. IdxScan(A)

Cascades – Memo Table

Cascades – Memo Table
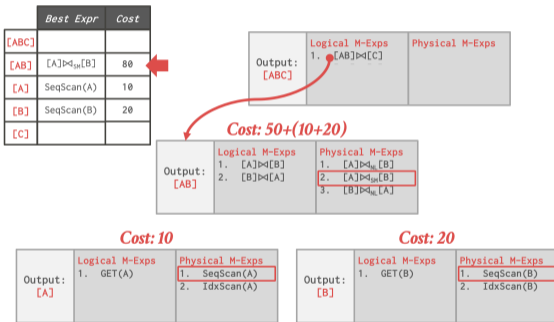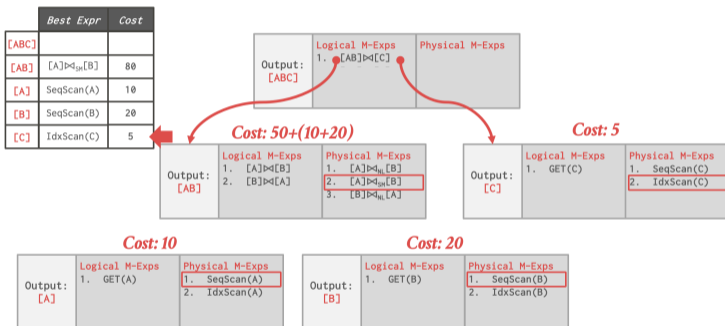
Cascades – Memo Table

Cascades – Memo Table

Cascades – Memo Table

Cascades – Memo Table

Cascades – Memo Table

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

Output: [ABC]

Logical M-Exps	Physical M-Exps
1. [AB]⋈[C]	1. [AB]⋈$_{NL}$C
2. [BC]⋈[A]	2. [BC]⋈$_{NL}$A
3. [AC]⋈[B]	3. [AC]⋈$_{NL}$B
4. [B]⋈[AC]	⋮

Cost: 50+(10+20)

Output: [AB]

Logical M-Exps	Physical M-Exps
1. [A]⋈[B]	1. [A]⋈$_{NL}$[B]
2. [B]⋈[A]	2. [A]⋈$_{pn}$[B]
	3. [B]⋈$_{NL}$[A]

Cost: 5

Output: [C]

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

Cost: 10

Output: [A]

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

Cost: 20

Output: [B]

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

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.

Case Studies

Cascades: Implementations

- **Approach 1: Standalone Optimizer Generator**
  - ▶ Wisconsin OPT++ (1990s)
  - ▶ Portland State Columbia (1990s)
  - ▶ Pivotal Orca (2010s)
  - ▶ Apache Calcite (2010s)
- **Approach 2: Integrated**
  - ▶ Microsoft SQL Server (1990s)
  - ▶ Tandem NonStop SQL (1990s)
  - ▶ Clustrix (2000s)
  - ▶ CMU Peloton (2010s – RIP)

Pivotal Orca

- Standalone Cascades (Optimization-as-a-service).
- Reference
  - 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 – 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.

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.
- Reference
- Originally part of LucidDB.

MemSQL Optimizer

- **Rewriter**
  - Logical-to-logical transformations with access to the cost-model.
- **Enumerator**
  - Logical-to-physical transformations.
  - Mostly join ordering.
- **Planner**
  - Convert physical plans back to SQL.
  - Contains MemSQL-specific commands for moving data.
- Reference

Conclusion

Parting Thoughts

- Cascades
  - ▶ Optimization tasks as data structures.
  - ▶ Rules to place **property enforcers** (*e.g.*, sorting order).
  - ▶ Ordering of transformations by priority.
  - ▶ Predicates are first class citizens (same as logical/physical operators).
- All of this relies on a good **cost model**.
- A good cost model needs good statistics.

Next Class

- Non-Traditional Query Optimization Techniques