Carnegie Mellon University

14 Query Planning – Part I

Intro to Database Systems
15-445/15-645
Fall 2020

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Mid-Term Exam is Wed Oct 21st

→ Session #1: 9:00am ET

→ Session #2: 3:20pm ET

→ See mid-term exam guide for more info.

Project #2 is due Sun Oct 25th @ 11:59pm

UPCOMING DATABASE TALKS

FoundationDB Testing
→ Monday Oct 19th @ 5pm ET

Datometry
→ Monday Oct 26th @ 5pm ET

MySQL Query Optimizer
→ Monday Nov 2nd @ 5pm ET

QUERY OPTIMIZATION

Remember that SQL is declarative.

→ User tells the DBMS what answer they want, not how to get the answer.

There can be a big difference in performance based on plan is used:

→ See last week: 1.3 hours vs. 0.45 seconds

IBM SYSTEM R

First implementation of a query optimizer from the 1970s.

→ People argued that the DBMS could never choose a query plan better than what a human could write.

Many concepts and design decisions from the **System R** optimizer are still used today.

QUERY OPTIMIZATION

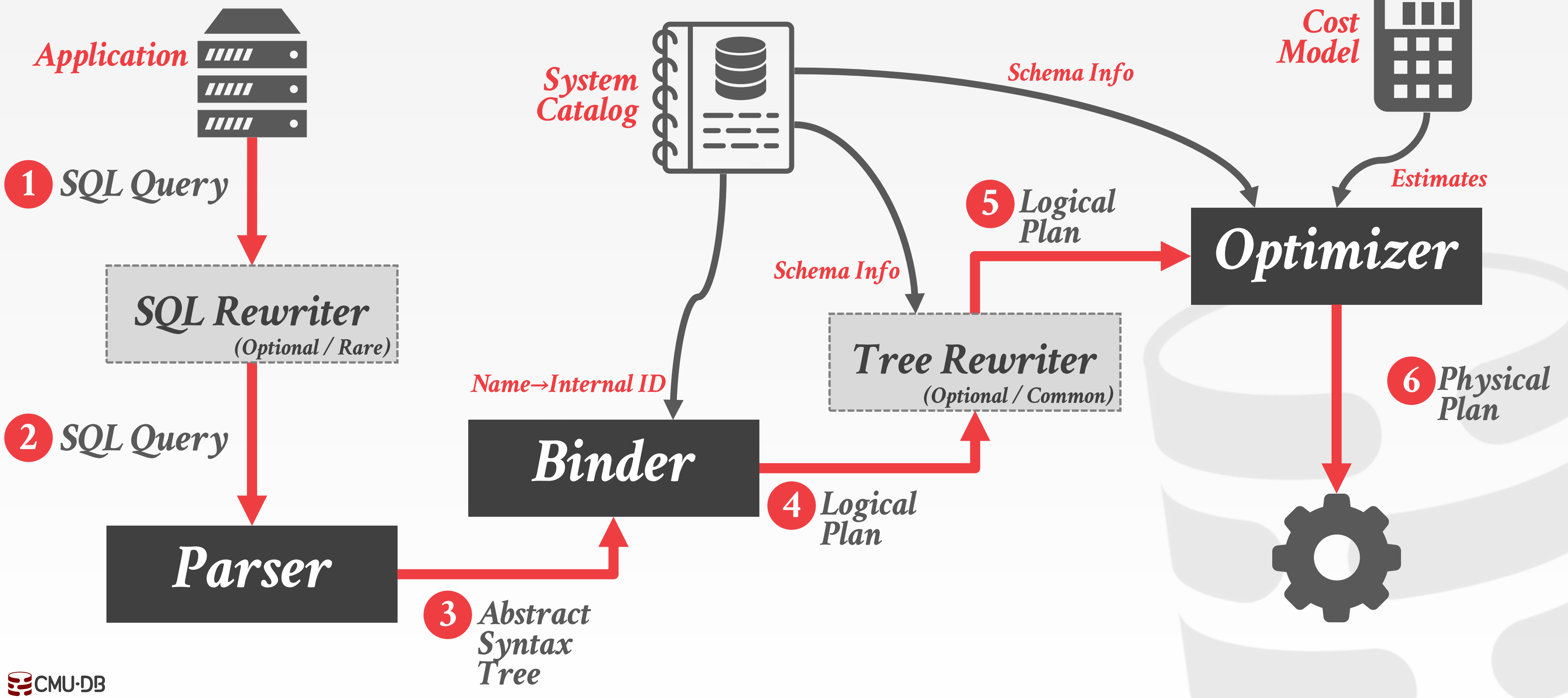
Heuristics / Rules

→ Rewrite the query to remove stupid / inefficient things.
→ These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search

→ Use a model to estimate the cost of executing a plan.
→ Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.

ARCHITECTURE OVERVIEW



LOGICAL VS. PHYSICAL PLANS

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

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

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

QUERY OPTIMIZATION IS NP-HARD

This is the hardest part of building a DBMS.

If you are good at this, you will get paid $$$.

People are starting to look at employing ML to improve the accuracy and efficacy of optimizers.

→ IBM DB2 tried this with LEO in the early 2000s…

TODAY'S AGENDA

Relational Algebra Equivalences

Logical Query Optimization

Nested Queries

Expression Rewriting

RELATIONAL ALGEBRA EQUIVALENCES

Two relational algebra expressions are equivalent if they generate the same set of tuples.

The DBMS can identify better query plans without a cost model.

This is often called query rewriting.

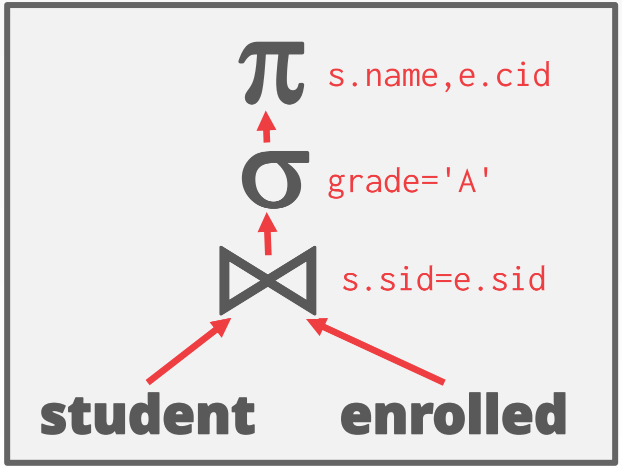
PREDICATE PUSHDOWN

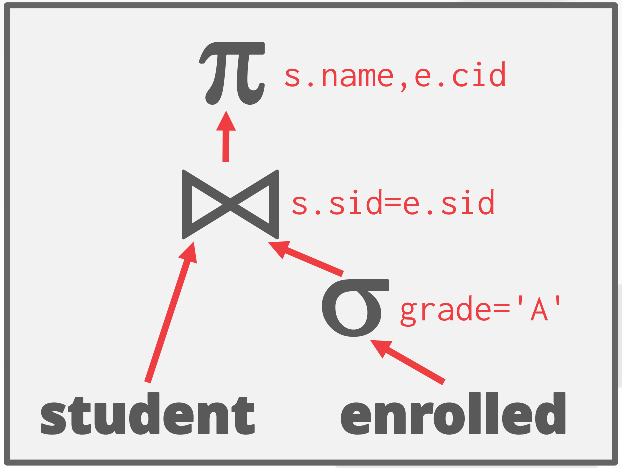
```sql
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

$$\pi_{\text{name, cid}}(\sigma_{\text{grade='A'}}(\text{student} \bowtie \text{enrolled}))$$

PREDICATE PUSHDOWN

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```





RELATIONAL ALGEBRA EQUIVALENCES

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

$$\pi_{\text{name, cid}}(\sigma_{\text{grade='A'}}(\text{student} \bowtie \text{enrolled}))$$

$$=$$

$$\pi_{\text{name, cid}}(\text{student} \bowtie (\sigma_{\text{grade='A'}}(\text{enrolled})))$$

RELATIONAL ALGEBRA EQUIVALENCES

Selections:

→ Perform filters as early as possible.
→ Break a complex predicate, and push down

$$\sigma_{p1 \wedge p2 \wedge \ldots pn}(\mathbf{R}) = \sigma_{p1}(\sigma_{p2}(\ldots\sigma_{pn}(\mathbf{R})))$$

Simplify a complex predicate

→ `(X=Y AND Y=3) → X=3 AND Y=3`

RELATIONAL ALGEBRA EQUIVALENCES

Joins:

→ Commutative, associative

$R \bowtie S = S \bowtie R$

$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

The number of different join orderings for an n-way join is a **Catalan Number** ($\approx 4^n$)

→ Exhaustive enumeration will be too slow.

RELATIONAL ALGEBRA EQUIVALENCES

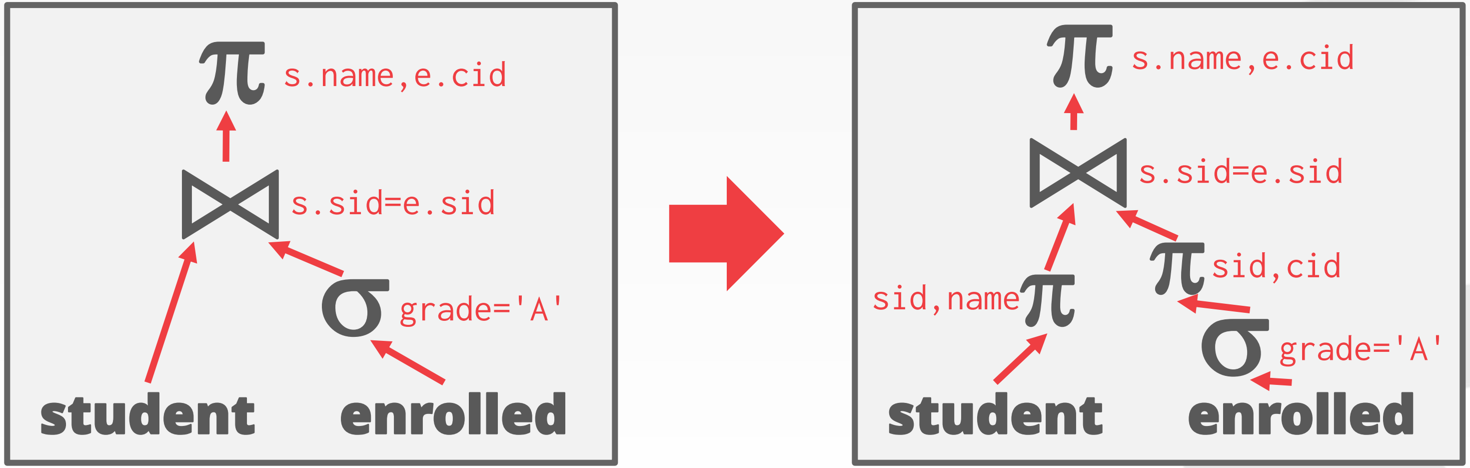
Projections:

- → Perform them early to create smaller tuples and reduce intermediate results (if duplicates are eliminated)
- → Project out all attributes except the ones requested or required (e.g., joining keys)

This is not important for a column store…

PROJECTION PUSHDOWN

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```



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

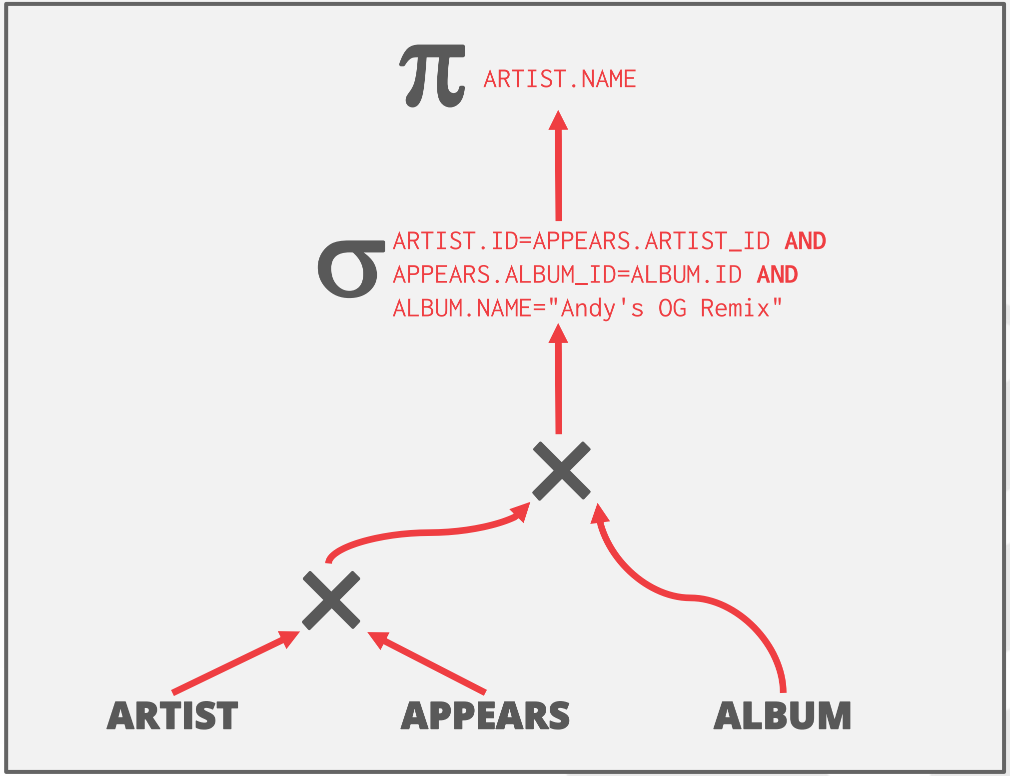
Projection Pushdown

Source: Thomas Neumann

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"
```

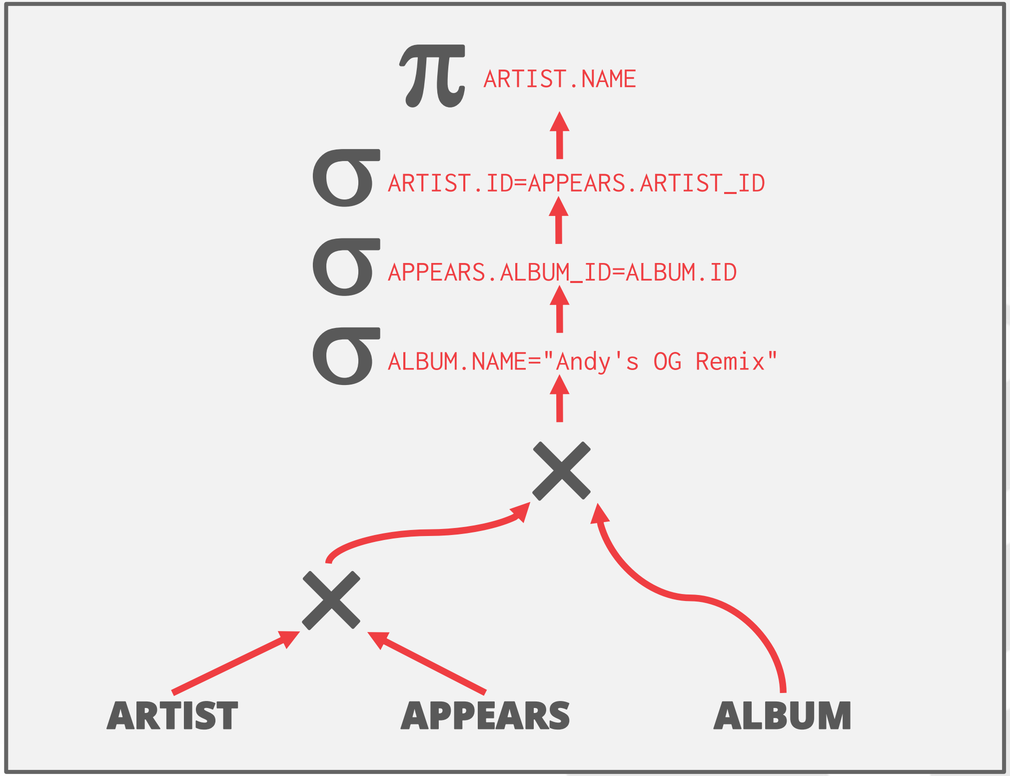
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



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"
```

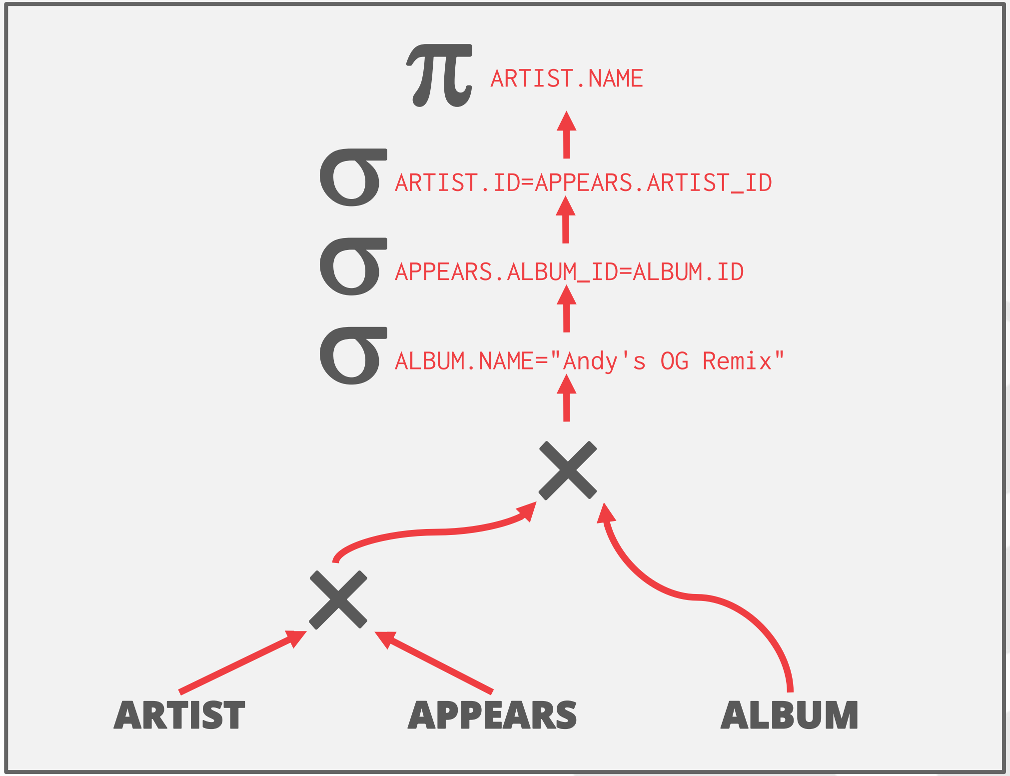
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



PREDICATE PUSHDOWN

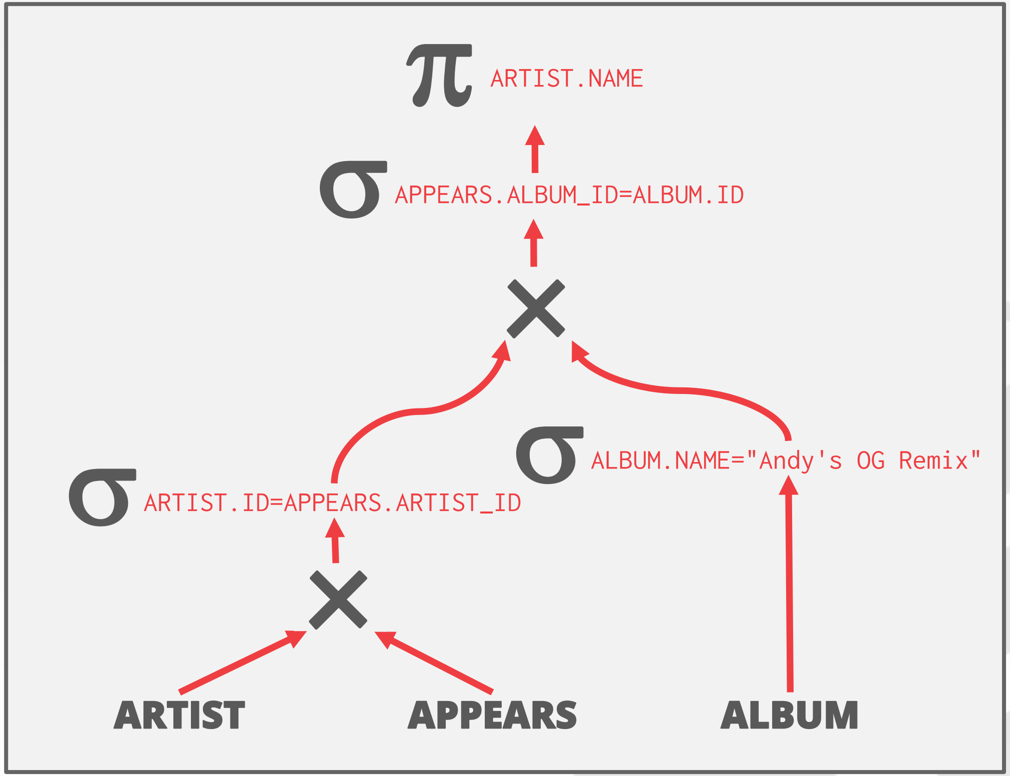
```sql
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

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"
```

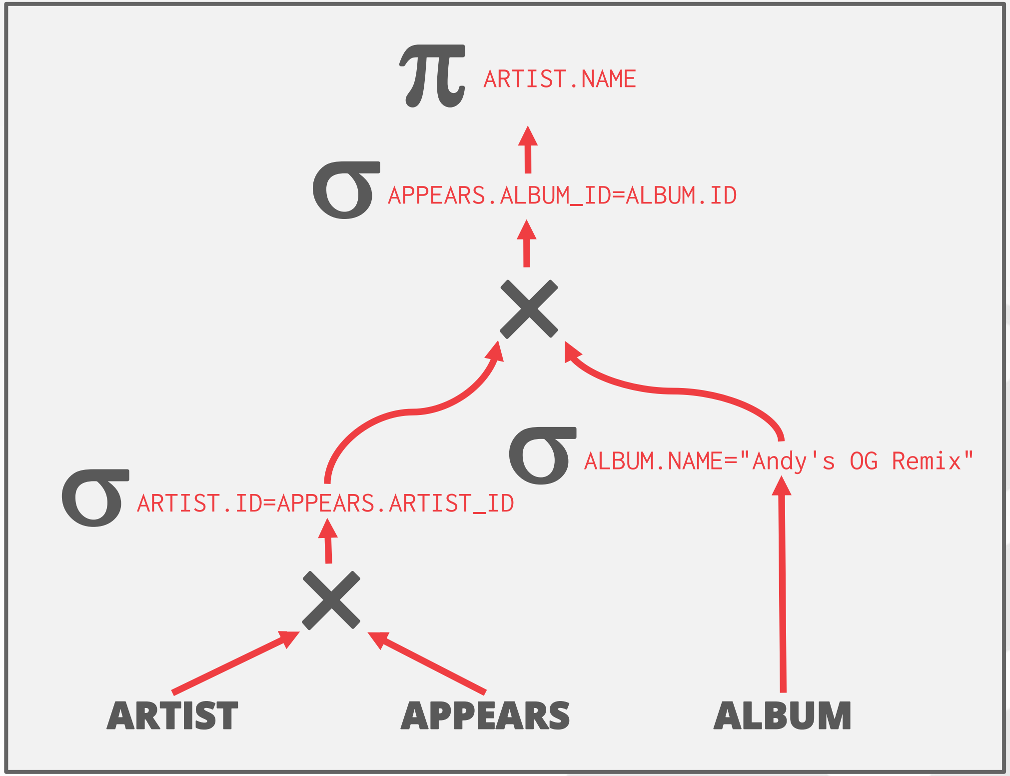


Move the predicate to the lowest point in the plan after Cartesian products.

REPLACE CARTESIAN PRODUCTS

```
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"
```

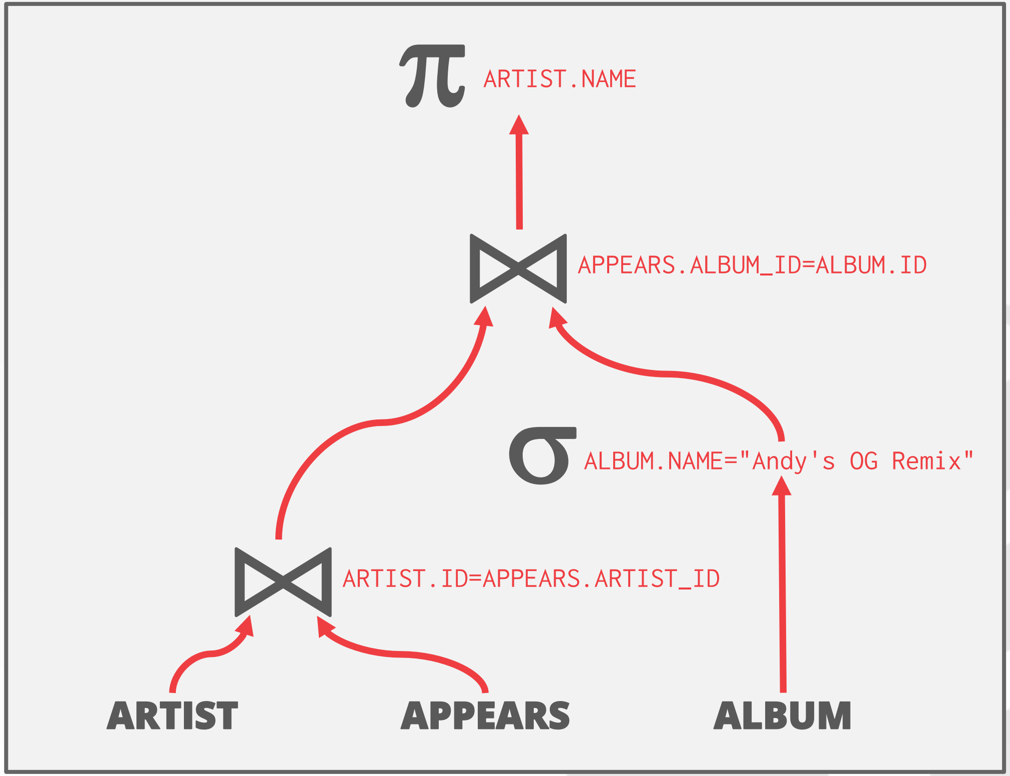
Replace all Cartesian Products with inner joins using the join predicates.



REPLACE CARTESIAN PRODUCTS

```
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"
```

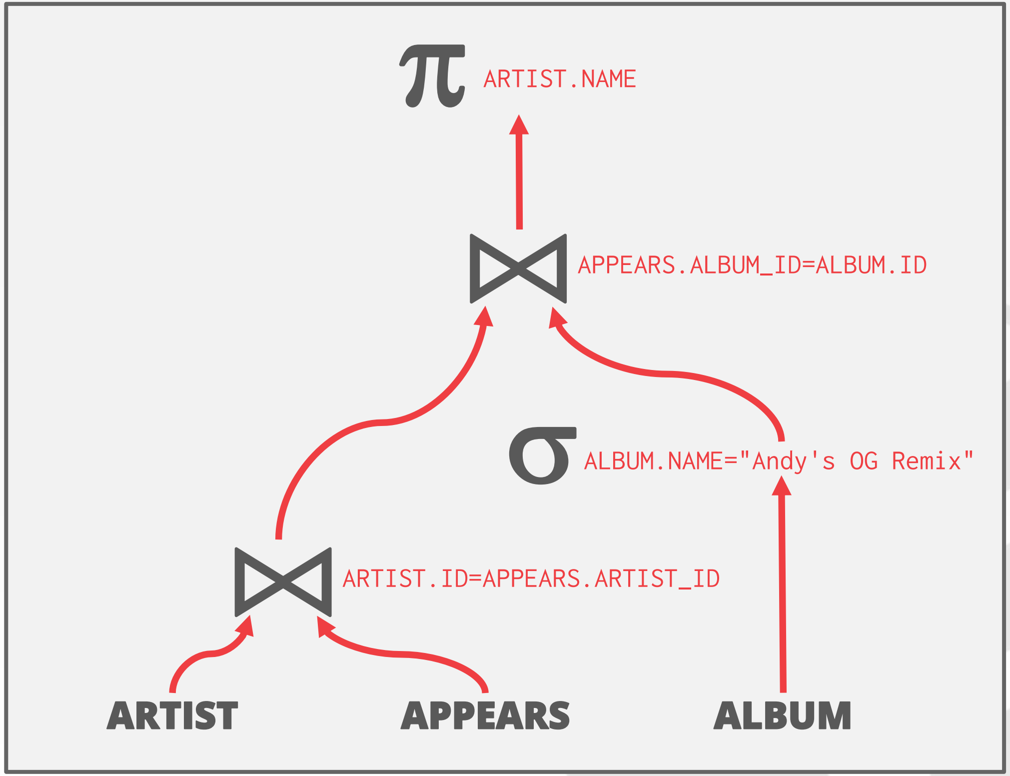
Replace all Cartesian Products with inner joins using the join predicates.



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"
```

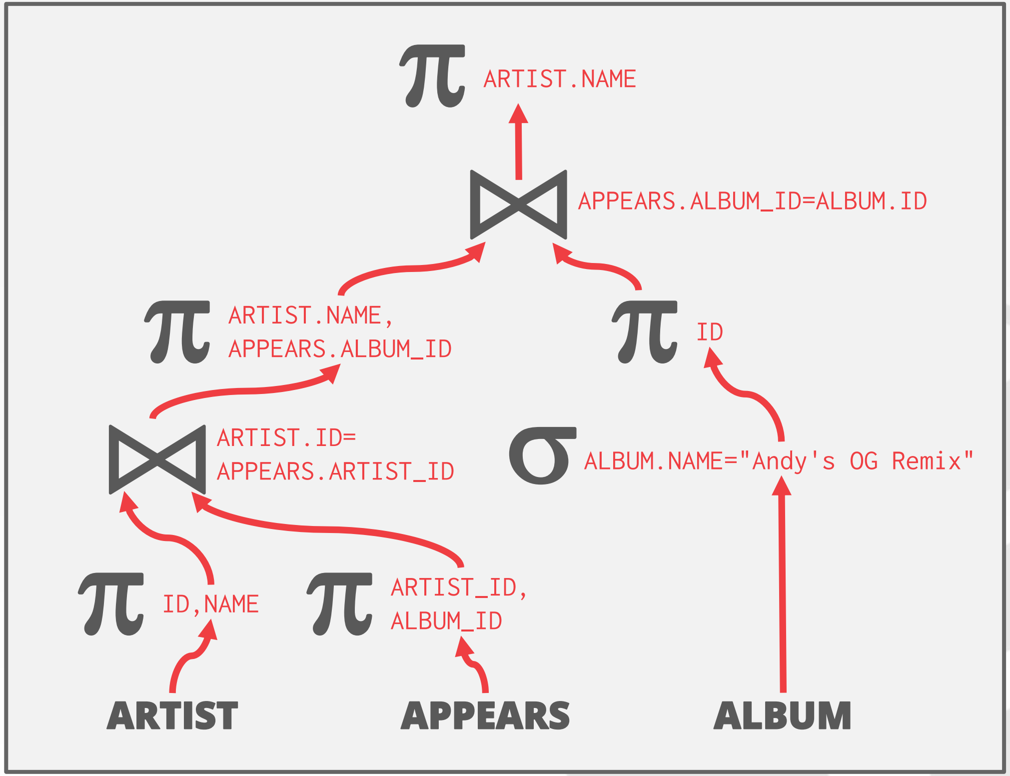
Eliminate redundant attributes before pipeline breakers to reduce materialization cost.



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.



NESTED SUB-QUERIES

The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

Two Approaches:

- → Rewrite to de-correlate and/or flatten them
- → Decompose nested query and store result to temporary table

NESTED SUB-QUERIES: REWRITE

```
SELECT name FROM sailors AS S
 WHERE EXISTS (
    SELECT * FROM reserves AS R
     WHERE S.sid = R.sid
       AND R.day = '2018-10-15'
 )
```

```
SELECT name
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = '2018-10-15'
```

NESTED SUB-QUERIES: DECOMPOSE

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                      FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

DECOMPOSING QUERIES

For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.

Sub-queries are written to a temporary table that are discarded after the query finishes.

DECOMPOSING QUERIES

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                     FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                     FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating =  ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Outer Block

EXPRESSION REWRITING

An optimizer transforms a query's expressions (e.g., **WHERE** clause predicates) into the optimal/minimal set of expressions.

Implemented using if/then/else clauses or a pattern-matching rule engine.

→ Search for expressions that match a pattern.
→ When a match is found, rewrite the expression.
→ Halt if there are no more rules that match.

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
SELECT * FROM A WHERE 1 = 1;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
SELECT * FROM A;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
SELECT * FROM A;
```

Join Elimination

```
SELECT A1.*
  FROM A AS A1 JOIN A AS A2
    ON A1.id = A2.id;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
SELECT * FROM A;
```

Join Elimination

```
SELECT * FROM A;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A AS A1
 WHERE EXISTS(SELECT val FROM A AS A2
               WHERE A1.id = A2.id);
```

Source: Lukas Eder

```sql
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```sql
SELECT * FROM A;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A;
```

Merging Predicates

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150;
```

Source: Lukas Eder

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A;
```

Merging Predicates

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 150;
```

Source: Lukas Eder

CONCLUSION

We can use static rules and heuristics to optimize a query plan without needing to understand the contents of the database.

NEXT CLASS

MID-TERM EXAM!