University of Michigan

20 Query Planning – Part I

Database Management Systems
EEGS 484
Fall 2024

Lin Ma
Computer Science and Engineering Division

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

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 operations.

→ These techniques may need to examine catalog, but they do not need to examine data.

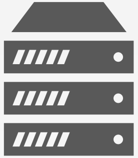
Cost-based Search

→ Use a model to estimate the cost of executing a plan.

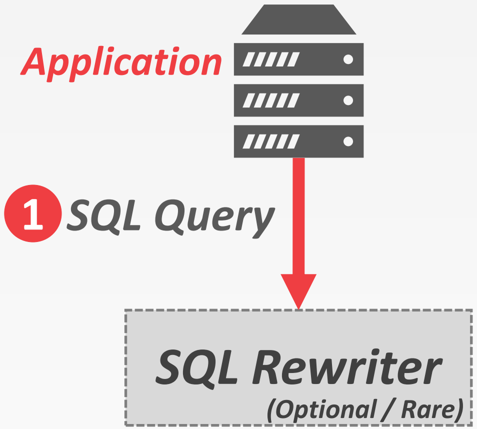
→ Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.

ARCHITECTURE OVERVIEW

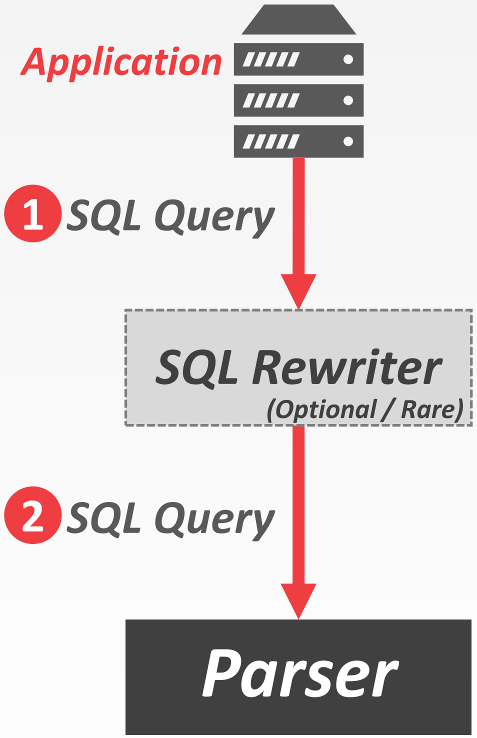
Application



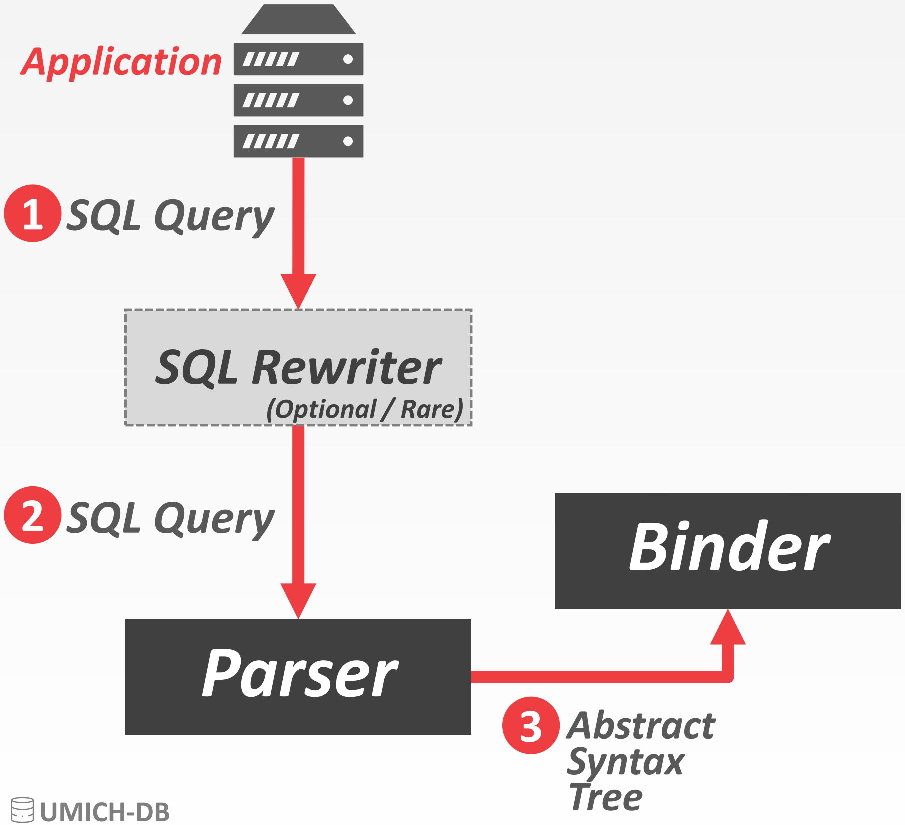
ARCHITECTURE OVERVIEW



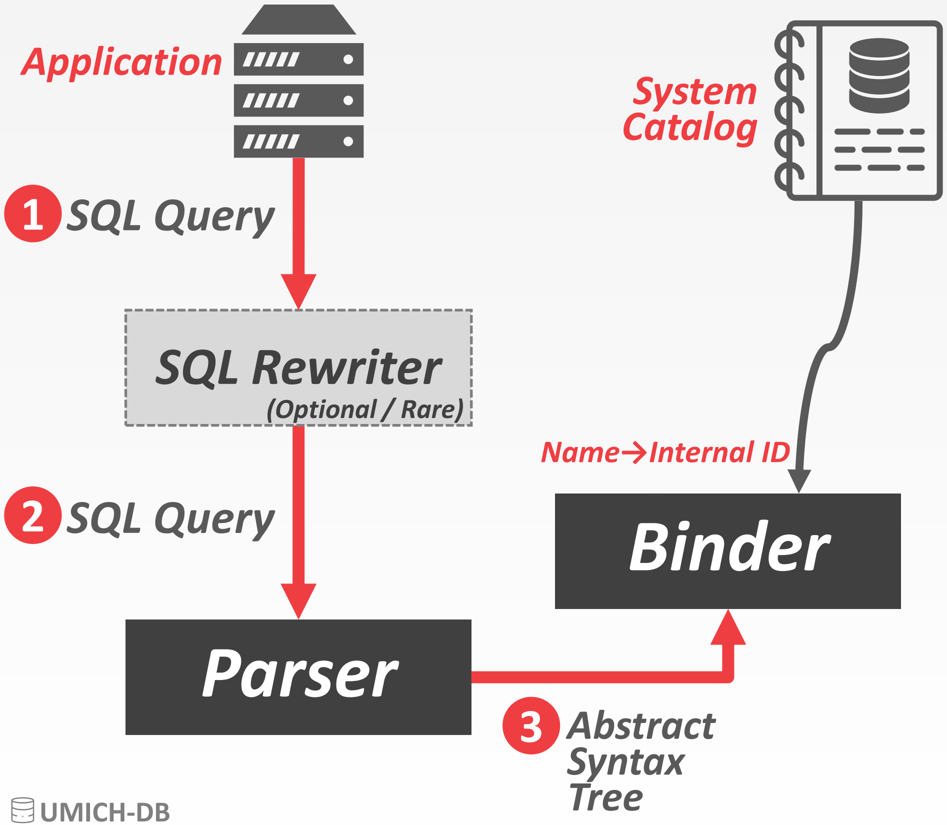
ARCHITECTURE OVERVIEW



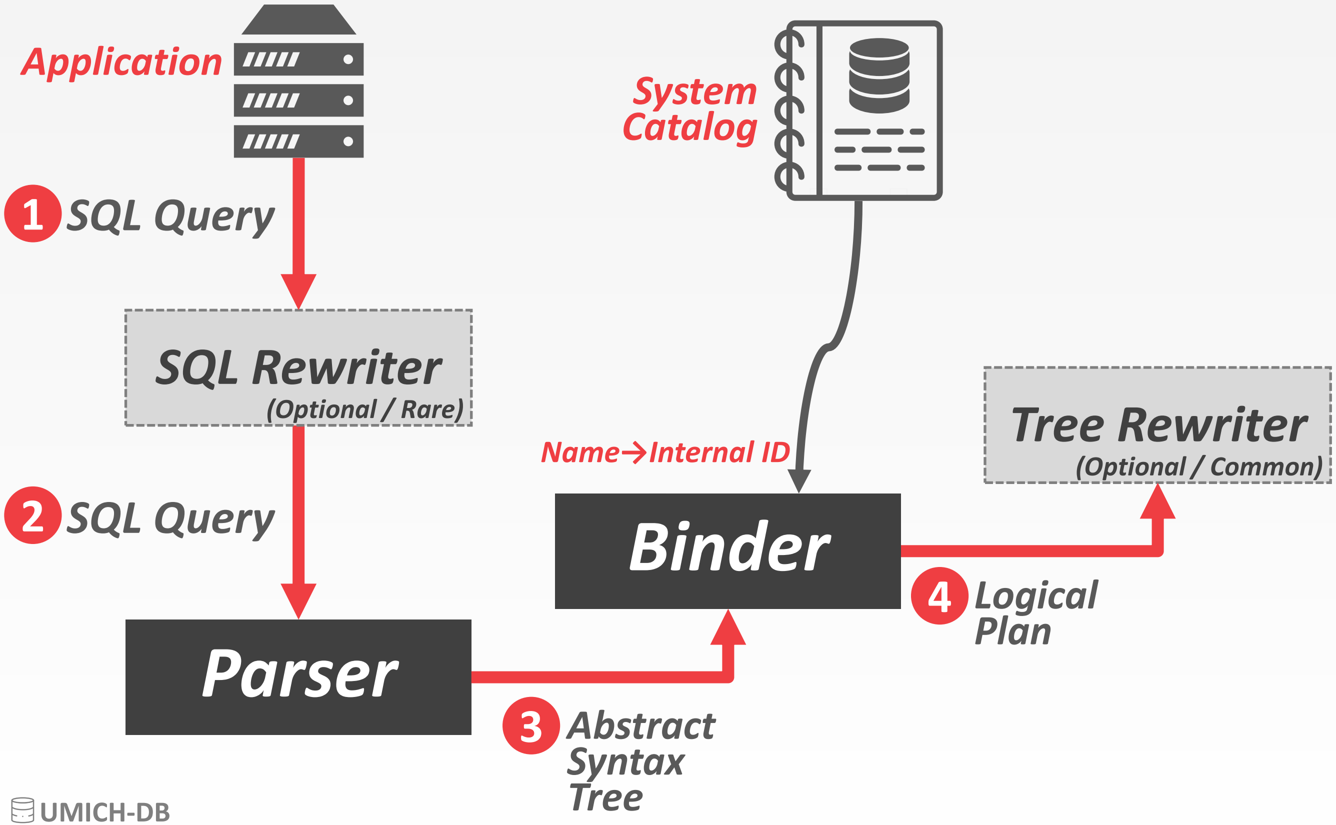
ARCHITECTURE OVERVIEW



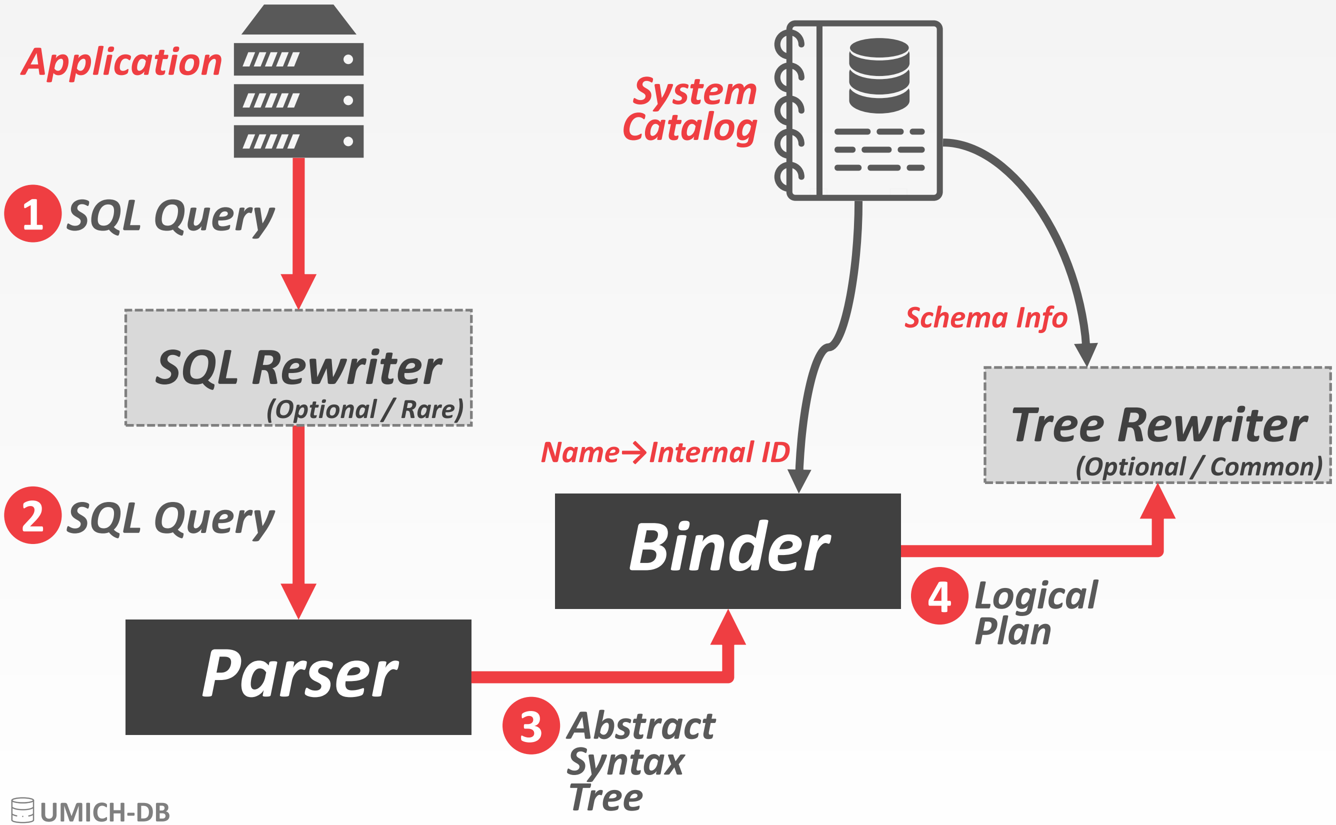
ARCHITECTURE OVERVIEW



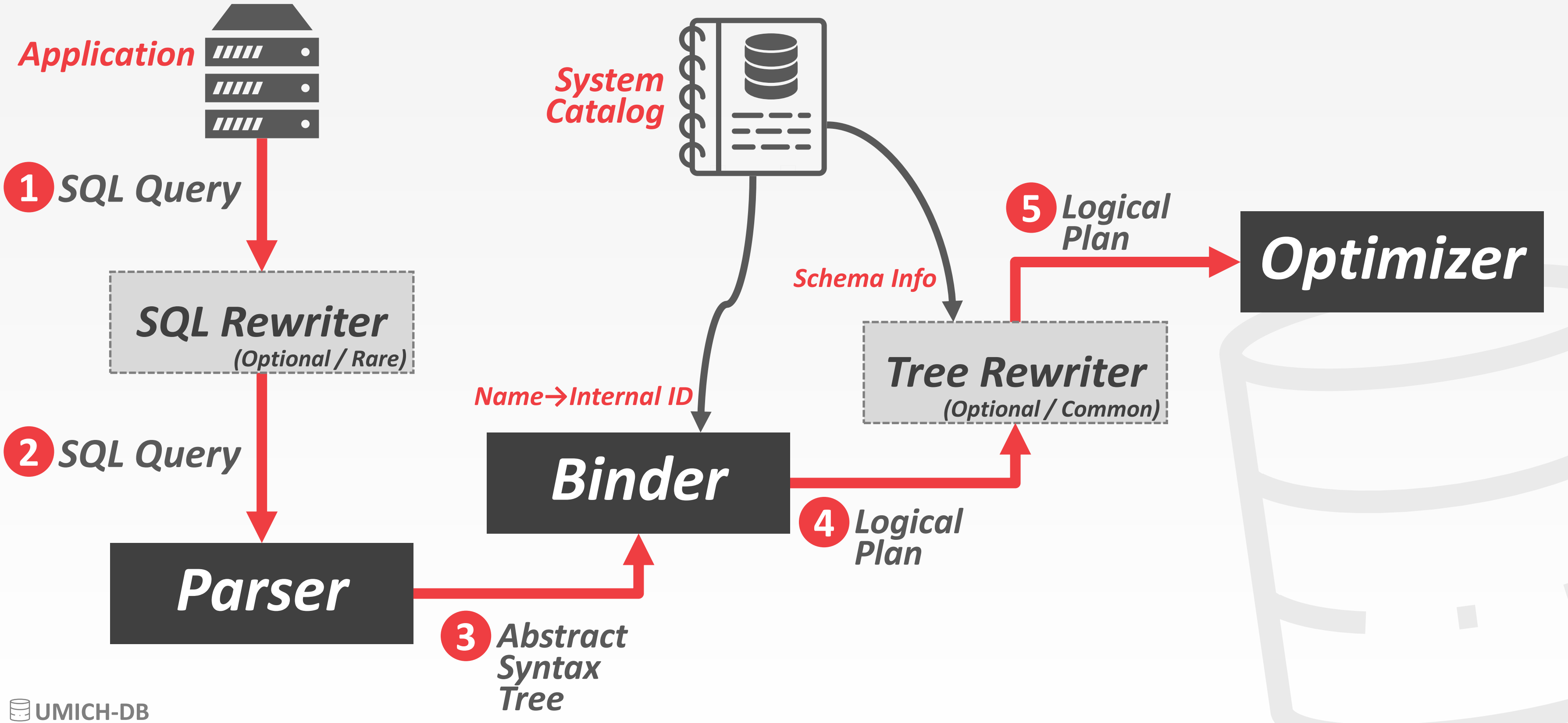
ARCHITECTURE OVERVIEW



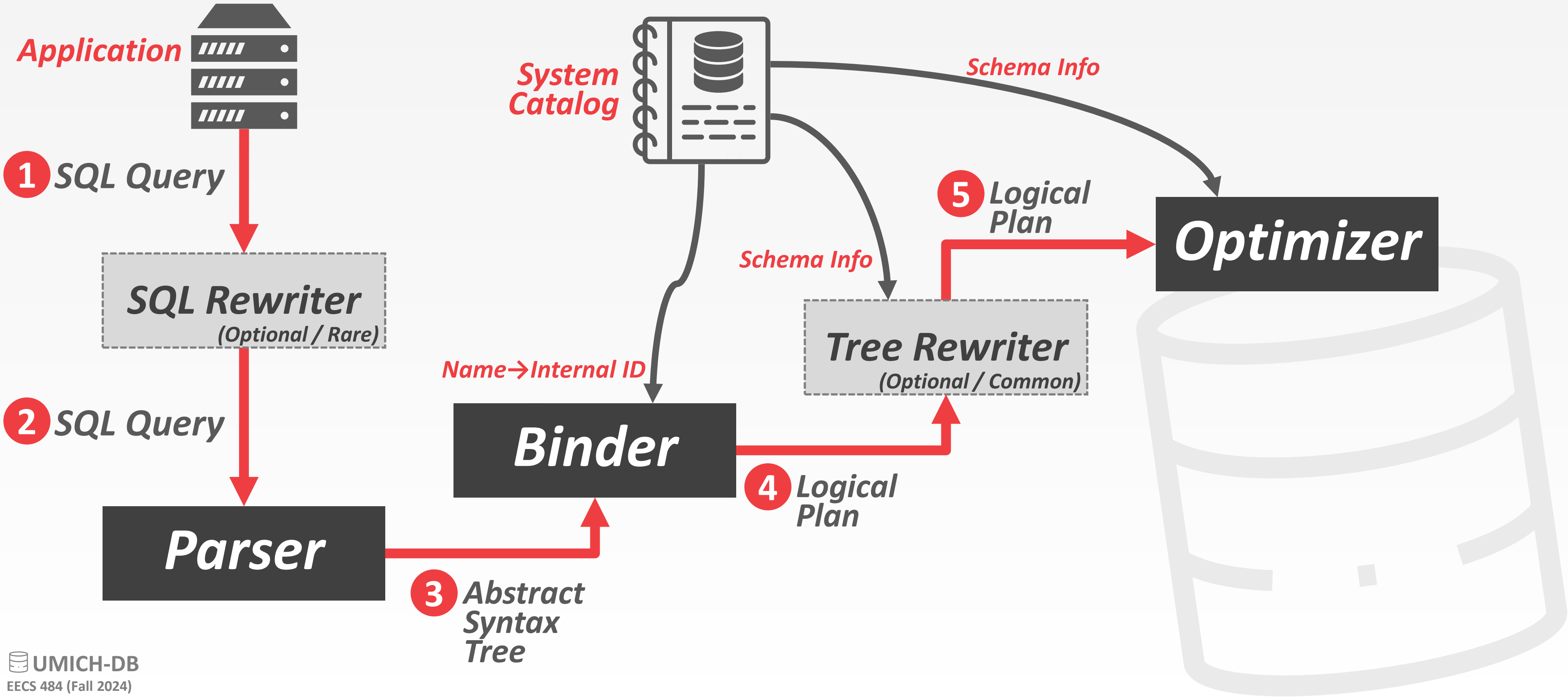
ARCHITECTURE OVERVIEW



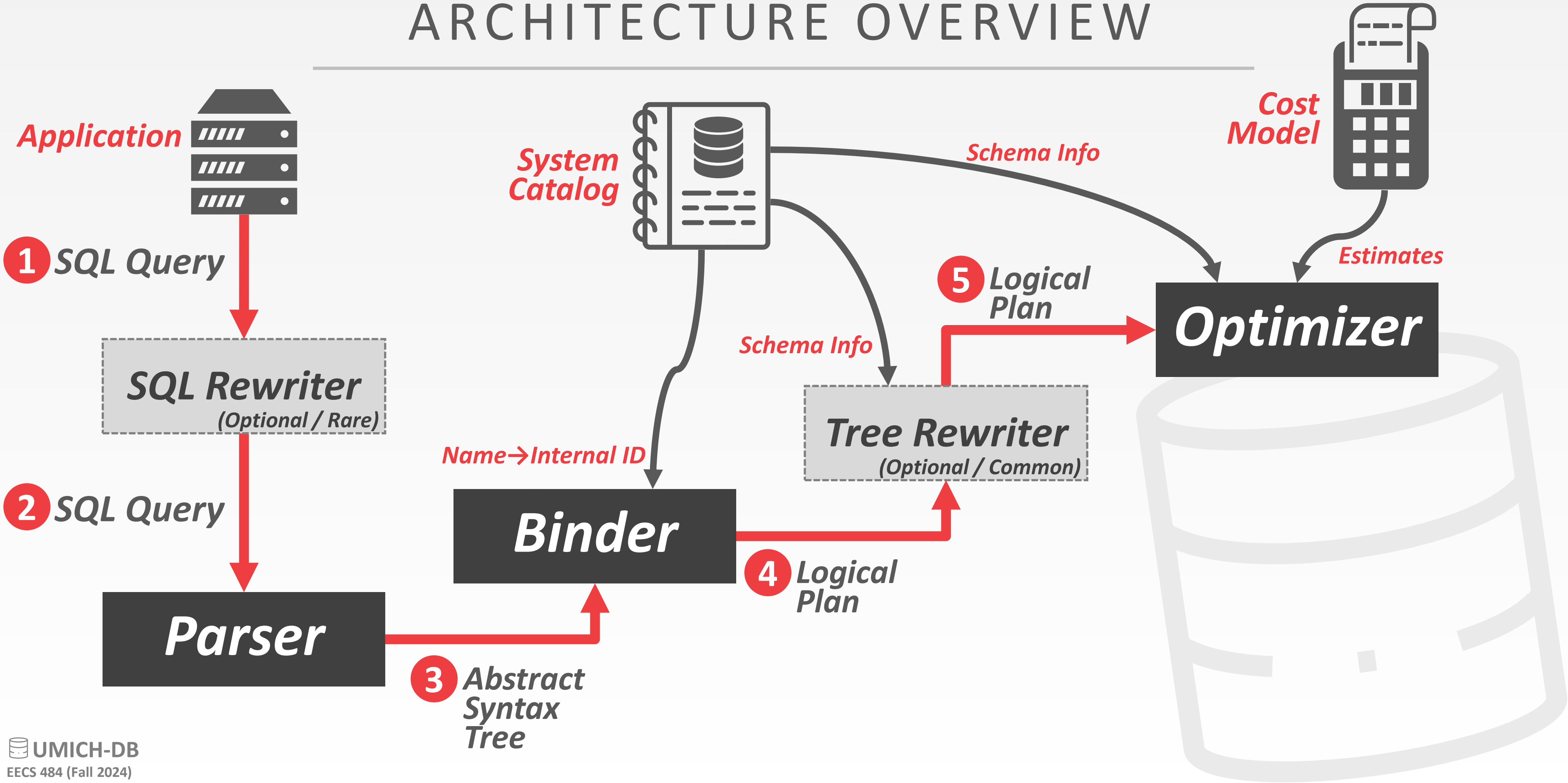
ARCHITECTURE OVERVIEW



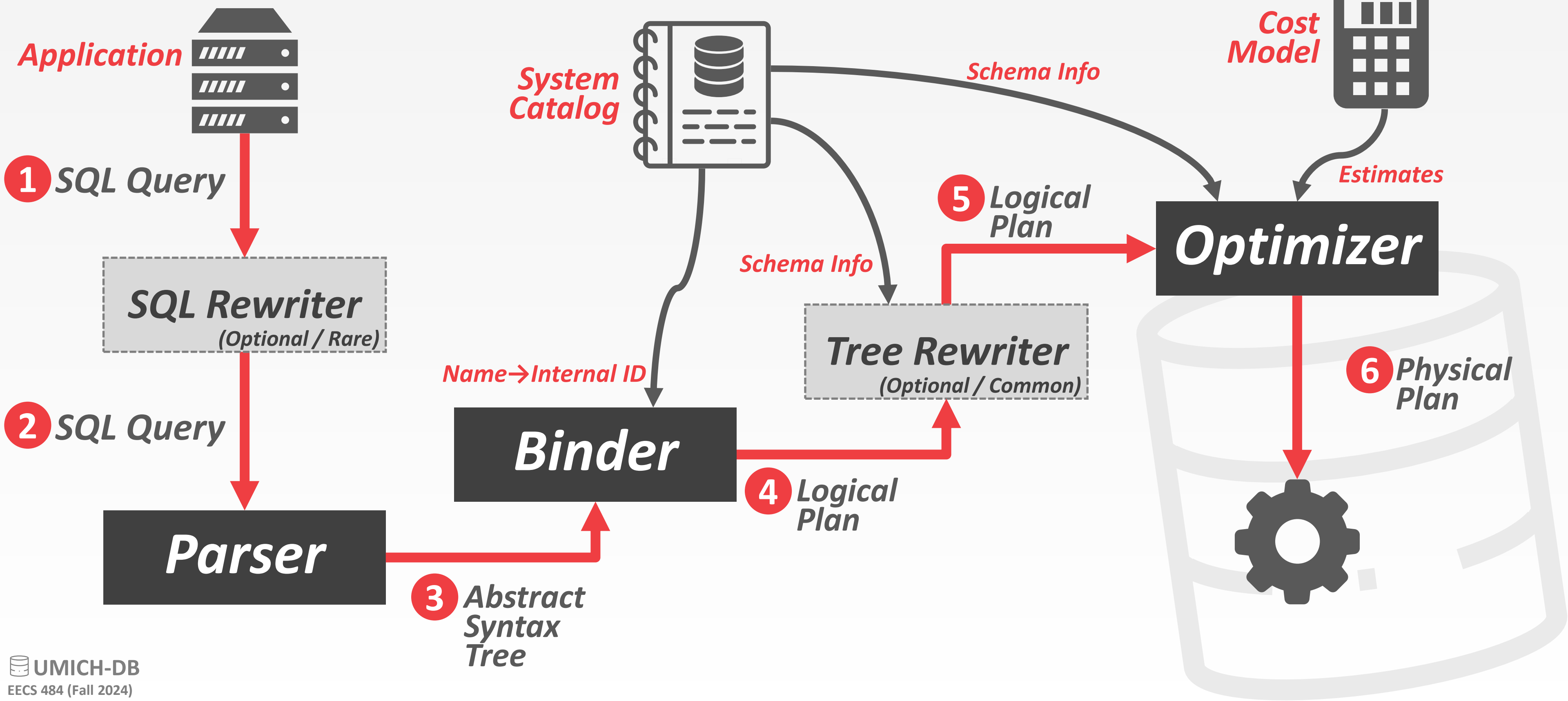
ARCHITECTURE OVERVIEW



ARCHITECTURE OVERVIEW



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.

TODAY'S AGENDA

Relational Algebra Equivalences

Logical Query Optimization

Nested Queries

Expression Rewriting

Cost Model

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.

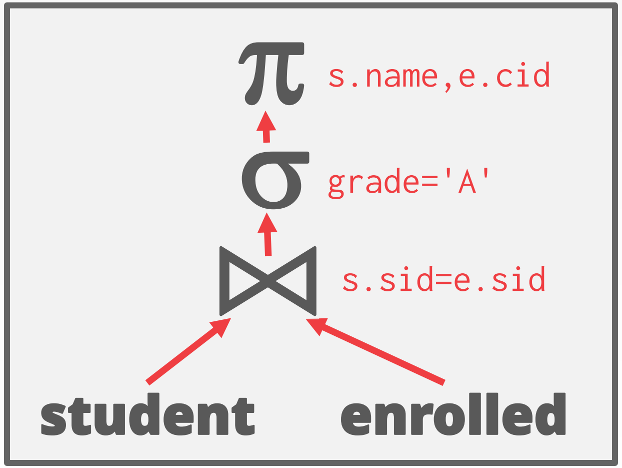
PREDICATE PUSHDOWN

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

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

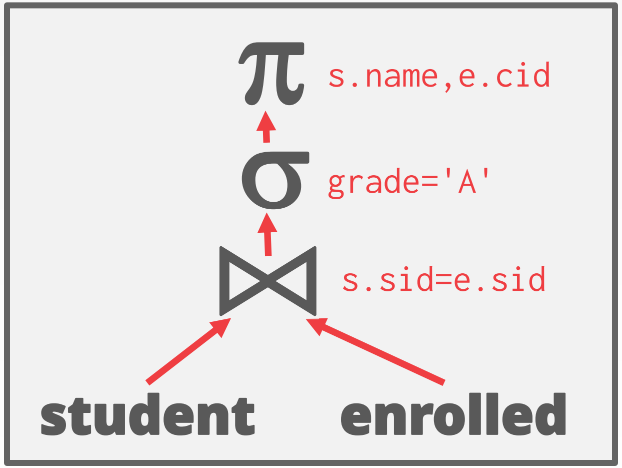
PREDICATE PUSHDOWN

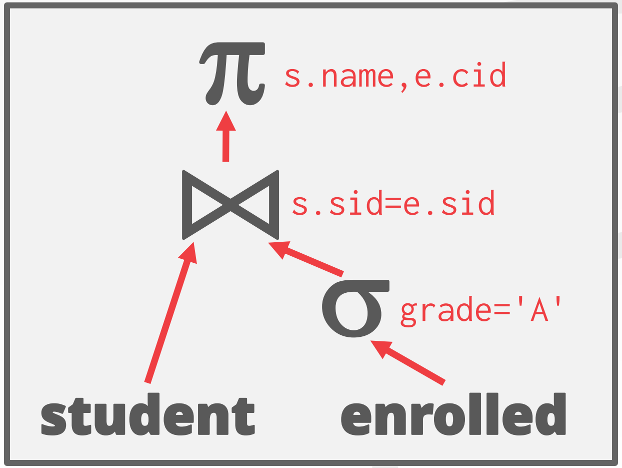
```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```



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

→ (A.X=B.Y AND B.Y=3) → (A.X=3) AND (B.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.

PROJECTION PUSHDOWN

Projections:

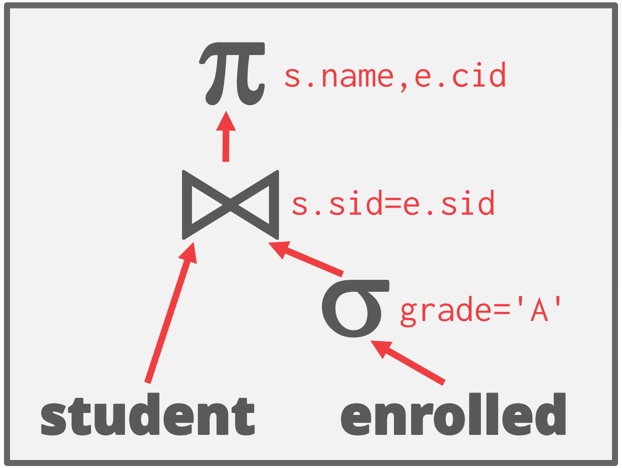
→ Perform them early to create smaller tuples and reduce intermediate results (if duplicates are eliminated)

→ Project out all attributes except the ones requested or required (e.g., joining keys)

This is not important for a column store…

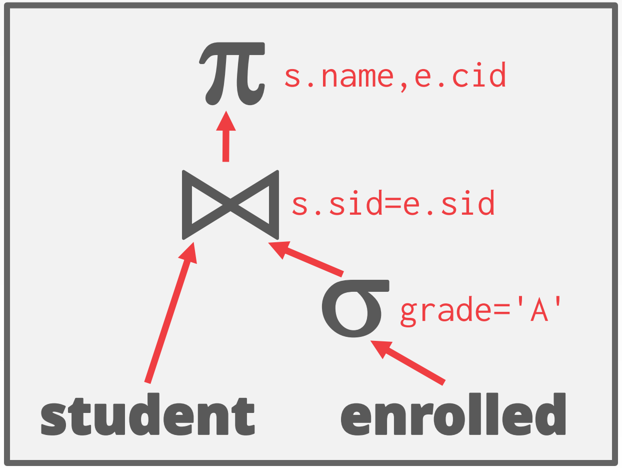
PROJECTION PUSHDOWN

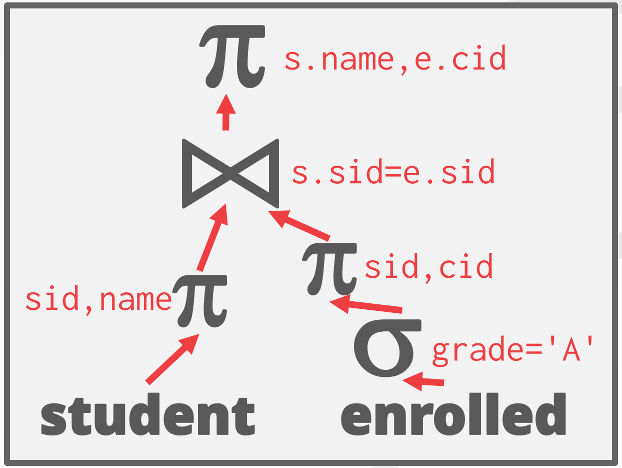
```sql
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```



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

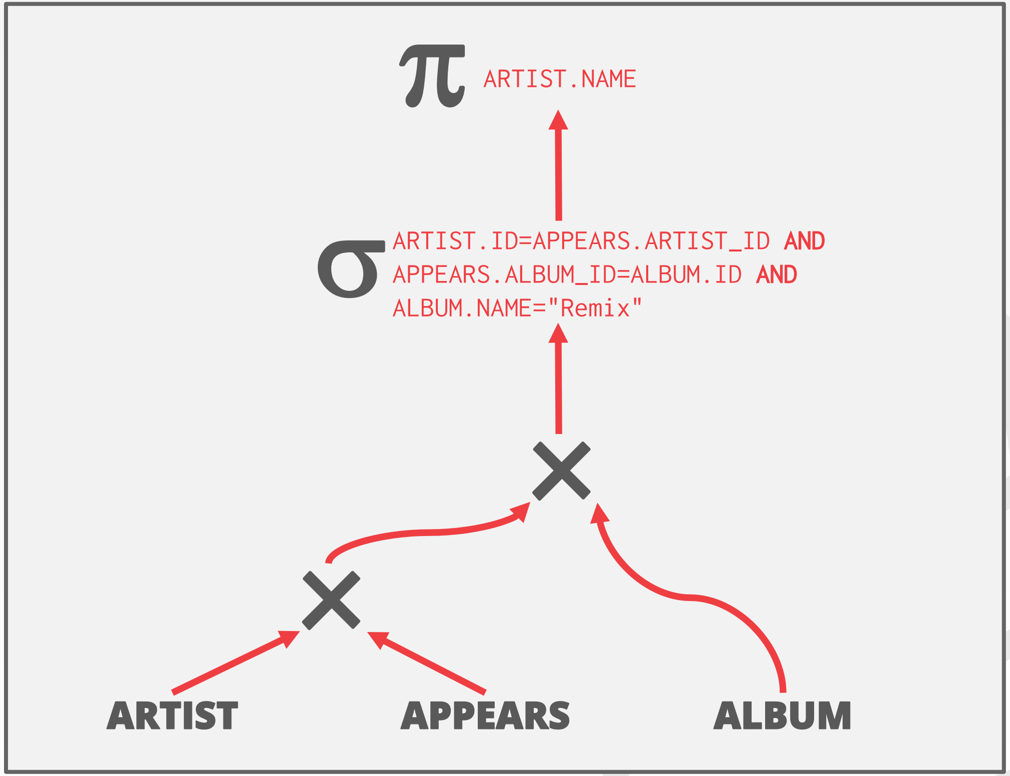
Projection Pushdown

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="Remix"
```

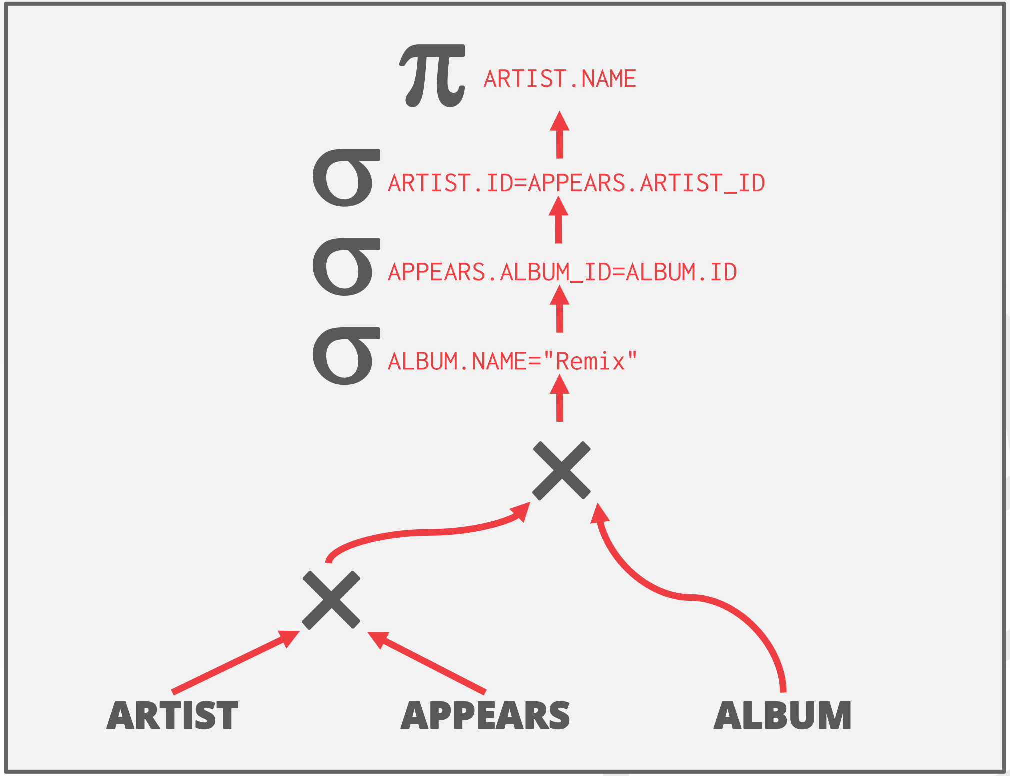
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



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="Remix"
```

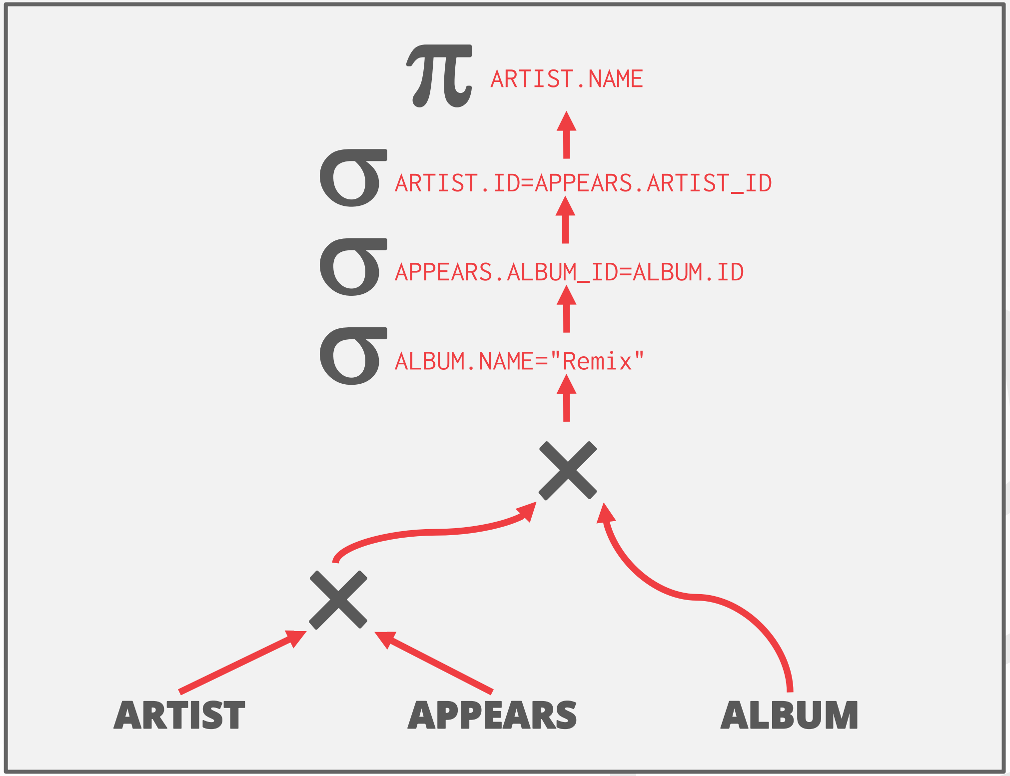
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



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"
```

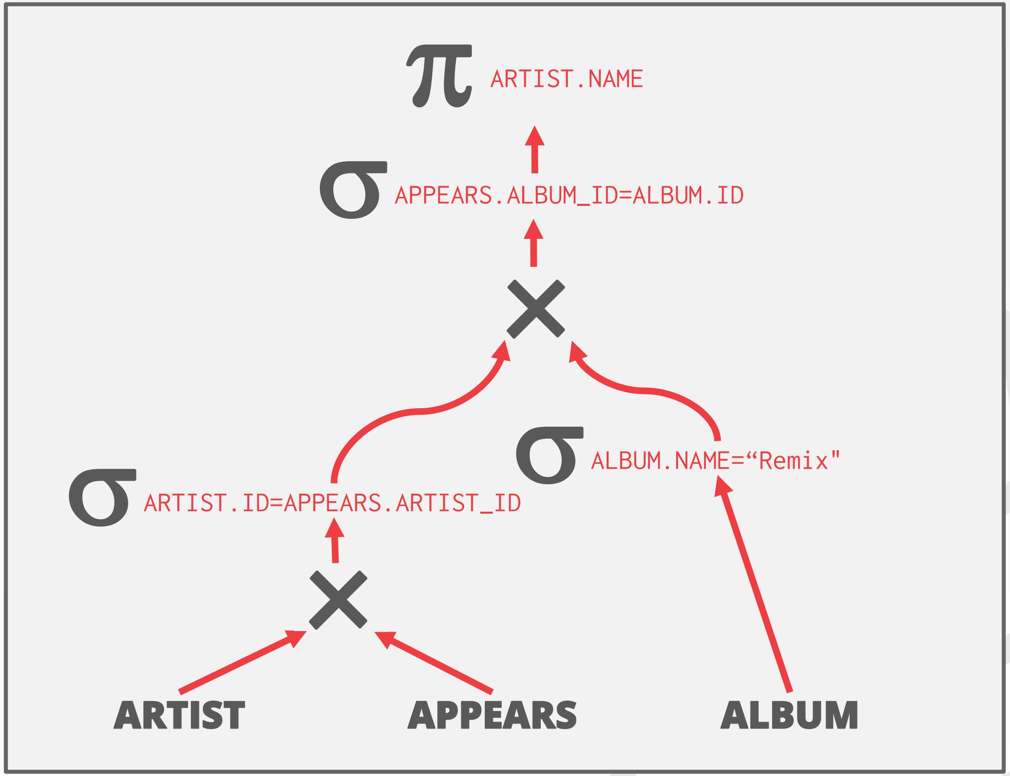
Move the predicate to the lowest applicable point in the plan.



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"
```

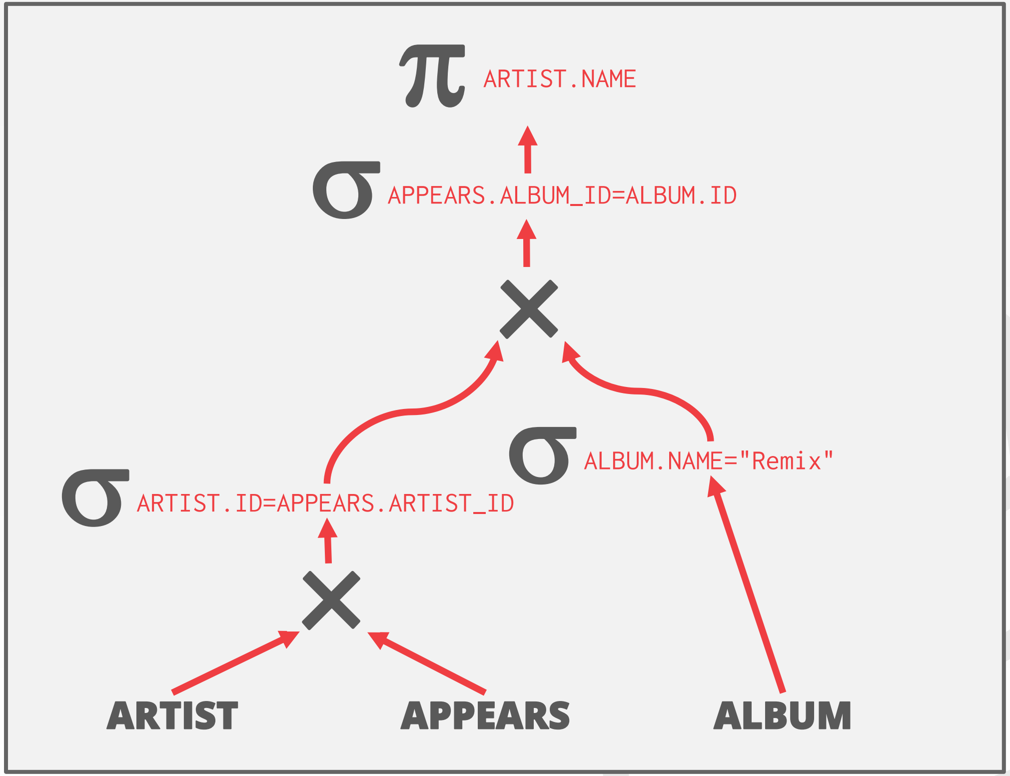
Move the predicate to the lowest applicable point in the plan.



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"
```

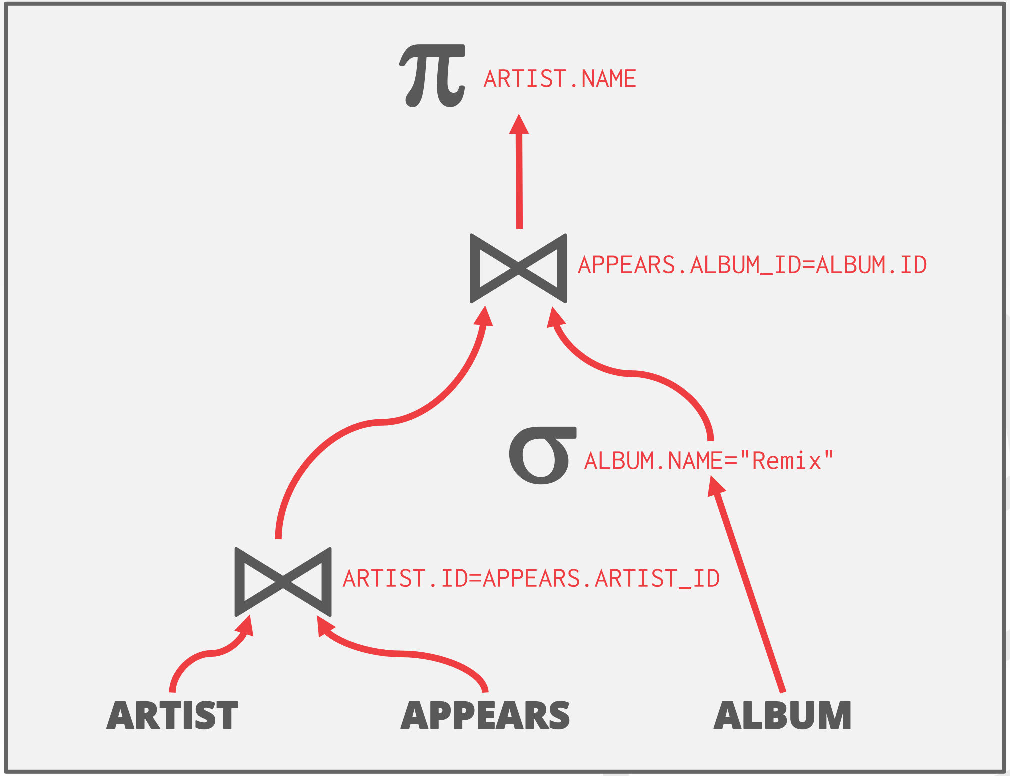
Replace all Cartesian Products with inner joins using the join predicates.



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"
```

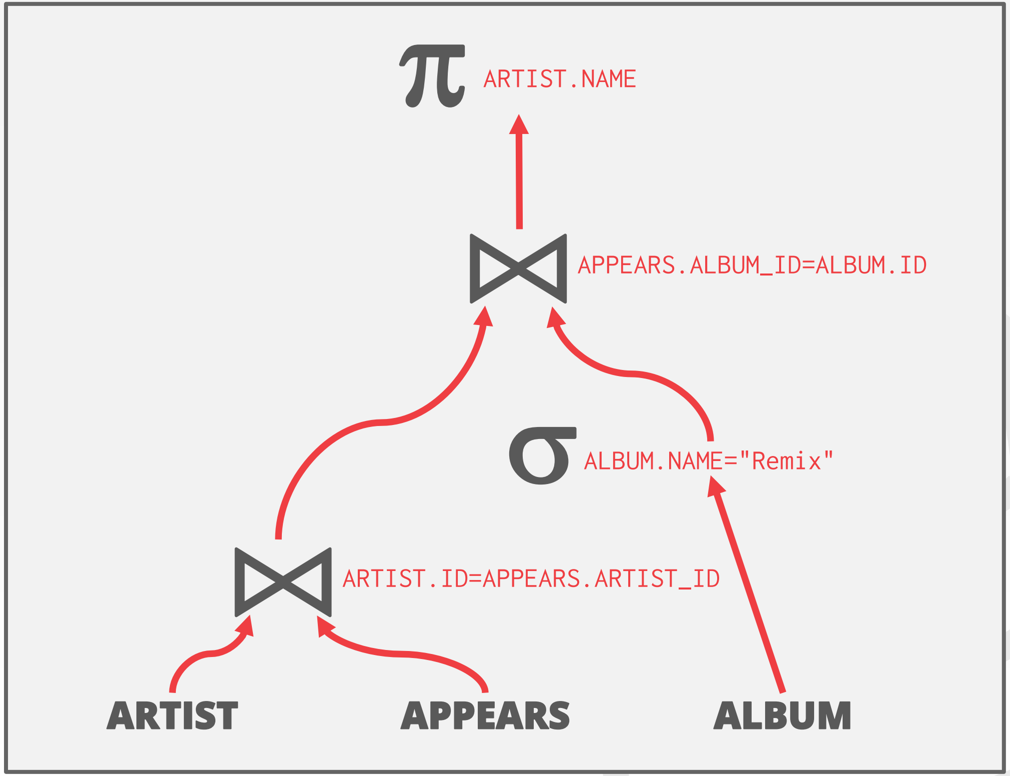
Replace all Cartesian Products with inner joins using the join predicates.



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"
```

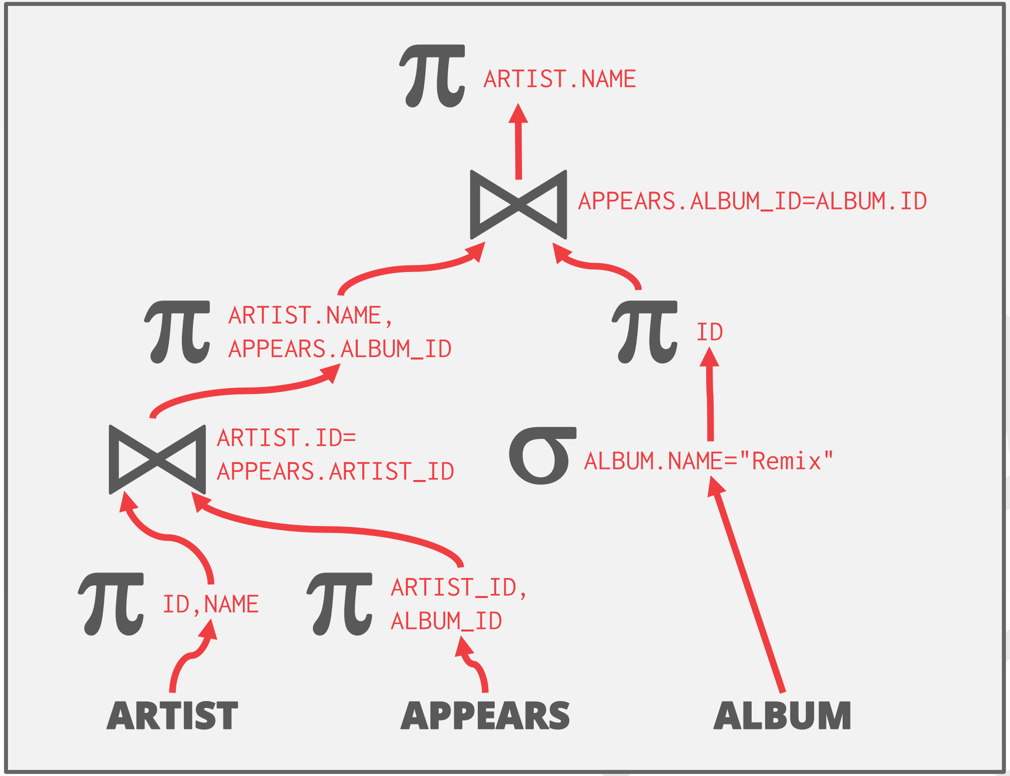
Eliminate redundant attributes to reduce materialization cost.



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

NESTED SUB-QUERIES: REWRITE

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

NESTED SUB-QUERIES: REWRITE

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

```
SELECT DISTINCT(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.

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
```

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;
```

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;
```

Source: Lukas Eder

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

MORE EXAMPLES

Impossible / Unnecessary Predicates

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

```sql
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 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

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

MORE EXAMPLES

Impossible / Unnecessary Predicates

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

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

Join Elimination

```sql
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

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

MORE EXAMPLES

Join Elimination with Sub-Query

```sql
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

Join Elimination with Sub-Query

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

Source: Lukas Eder

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

MORE EXAMPLES

Join Elimination with Sub-Query

```
SELECT * FROM A;
```

Source: Lukas Eder

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

MORE EXAMPLES

Join Elimination with Sub-Query

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

Merging Predicates

```sql
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

Join Elimination with Sub-Query

```
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

Join Elimination with Sub-Query

```
SELECT * FROM A;
```

Merging Predicates

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

Source: Lukas Eder

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.
→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

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.

→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

COST-BASED QUERY PLANNING

Generate an estimate of the cost of executing a particular query plan for the current state of the database.

→ Estimates are only meaningful internally.

This is independent of the plan enumeration step that we will talk about next class.

COST MODEL COMPONENTS

Choice #1: Physical Costs

→ Predict CPU cycles, I/O time, cache misses, RAM consumption, pre-fetching, etc…

→ Depends heavily on hardware.

Choice #2: Logical Costs

→ Estimate result sizes per operator.

→ Complexity of the operator algorithm implementation.

→ # sequential I/Os, # random I/Os, # arithmetics.

DISK-BASED DBMS COST MODEL

The number of disk accesses will always dominate the execution time of a query.

→ CPU costs are negligible.
→ Should consider sequential vs. random I/O.

This is easier to model if the DBMS has full control over buffer management.

→ We will know the replacement strategy, pinning, and assume exclusive access to disk.

POSTGRES COST MODEL

Uses a combination of CPU and I/O costs that are weighted by “magic” constant factors.

Default settings are for a disk-resident database without a lot of memory:

→ Processing a tuple in memory is **400x** faster than reading a tuple from disk.
→ Sequential I/O is **4x** faster than random I/O.

19.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

> **Note:** Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost` (`floating point`)

19.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

> **Note:** Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost` (`floating point`)

IBM DB2 COST MODEL

Database characteristics in system catalogs

Hardware environment (microbenchmarks)

Storage device characteristics (microbenchmarks)

Communications bandwidth (distributed only)

Memory resources (buffer pools, sort heaps)

Concurrency Environment

→ Average number of users

→ Isolation level / blocking

→ Number of available locks

Source: Guy Lohman

CONCLUSION

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

We use cost model to help perform more advanced query optimizations

NEXT CLASS

Statistics and plan enumeration