

Carnegie Mellon University

OPTIMIZE!

Database Query Optimization

Query Plan Transformations

SPRING 2025 » SPECIAL TOPICS IN DATABASES » PROF. ANDY PAVLO

ADMINISTRIVIA

Paper Reviews resume this Wednesday Feb 5th

Project #1 is due Friday Feb 28th

UPCOMING DATABASE TALKS

Convex (DB Seminar)

→ Monday Feb 10th @ 4:30pm ET

→ Zoom



The Germans (DB Seminar)

→ Monday Feb 17th @ 4:30pm ET

→ Zoom



Pinot (DB Seminar)

→ Monday Feb 24th @ 4:30pm ET

→ Zoom



LAST CLASS

We discussed the key ideas of the Cascades optimizer architecture:

- Task-based transformation scheduling
- Deferred Expansion of Logical Expressions
- Promise-based Guidance
- Memo Table

OBSERVATION

We now have a conceptual understanding of the high-level architecture of a query optimizer.

The quality of the plans that an optimizer generates is mostly based on three factors:

- Search Algorithm
- Cost Model
- Transformations

TRANSFORMATIONS

Changing query plan into a new form that is semantically equivalent / logically correct.

- Need to ensure new query plan produces the same result as the original no matter the inputs.
- Exploit relational algebra equivalencies in the context of query and database contents (logical + physical).

TRANSFORMATIONS

Changing query plan into a new form that is semantically equivalent / logically correct.

- Need to ensure new query plan produces the same result as the original no matter the inputs.
- Exploit relational algebra equivalencies in the context of query and database contents (logical + physical).

The goal of each transformation is to:

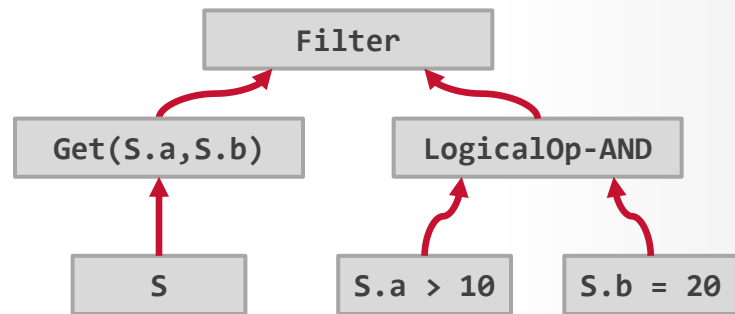
- Lower query execution cost.
- Unlock additional transformations.

NOTA BENE

We will follow the [EQO Book](#) style and represent predicates as separate logical operators.

In a real-world implementation, physical operators contain the predicates as internal attributes.

```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```



TODAY'S AGENDA

Access Path

Inner Joins

Outer Joins

Group-By

Star / Snowflake Queries

ACCESS PATH TRANSFORMATION

An optimizer chooses the access method(s) for those relations that minimizes the cost of retrieving a query's requested data from base relations.

Cost of access method depends on several factors:

- Selectivity of predicate
- Sort order of the table / index
- Data accoutrements (e.g., **INCLUDE**, zone maps)
- Compression / encoding

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to improve the likelihood of index-only queries.

DBMS stores these columns in the leaf nodes and are not part of the search key.

→ Include columns cannot be used to guarantee uniqueness.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      INCLUDE (c)
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to improve the likelihood of index-only queries.

DBMS stores these columns in the leaf nodes and are not part of the search key.

→ Include columns cannot be used to guarantee uniqueness.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      INCLUDE (c);
```

```
SELECT b FROM foo
WHERE a = 123
      AND c = 'WuTang';
```

INDEX INCLUDE COLUMNS

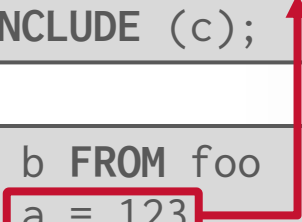
Embed additional columns in indexes to improve the likelihood of index-only queries.

DBMS stores these columns in the leaf nodes and are not part of the search key.

→ Include columns cannot be used to guarantee uniqueness.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      INCLUDE (c);
```

```
SELECT b FROM foo
WHERE a = 123
AND c = 'WuTang';
```



INDEX INCLUDE COLUMNS

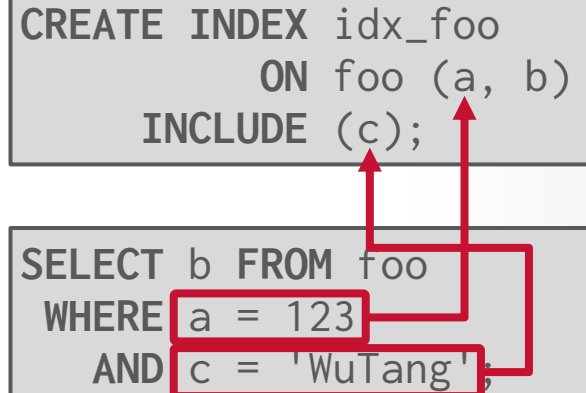
Embed additional columns in indexes to improve the likelihood of index-only queries.

DBMS stores these columns in the leaf nodes and are not part of the search key.

→ Include columns cannot be used to guarantee uniqueness.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      INCLUDE (c);
```

```
SELECT b FROM foo
WHERE a = 123
AND c = 'WuTang';
```

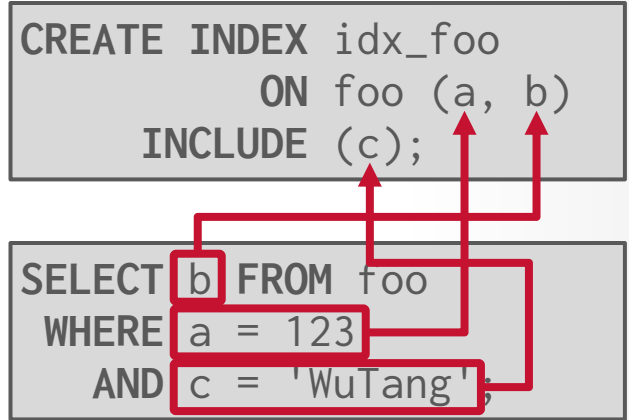


INDEX INCLUDE COLUMNS

Embed additional columns in indexes to improve the likelihood of index-only queries.

DBMS stores these columns in the leaf nodes and are not part of the search key.

→ Include columns cannot be used to guarantee uniqueness.



ACCESS METHODS

Index Seek:

→ Retrieve tuple(s) for a predicate on index.

Index Key Lookup:

→ Return true/false if key exists in index.

Index Scan:

→ Range scan on index.

Table Scan:

→ Sequential scan on table heap.

SINGLE ACCESS METHOD

Generate multiple alternatives for retrieving data from a base relation for a given expression.

Available alternatives depend on query, database logical schema, and DBMS implementation.

→ Example: A rule determines whether an index qualifies based on a query's predicates (e.g., partial indexes).

Table Scan is always the fallback option.

→ Often worst choice in row stores but it is sometimes the only choice in column stores.

SINGLE ACCESS METHOD

□ Logical Op

■ Physical Op

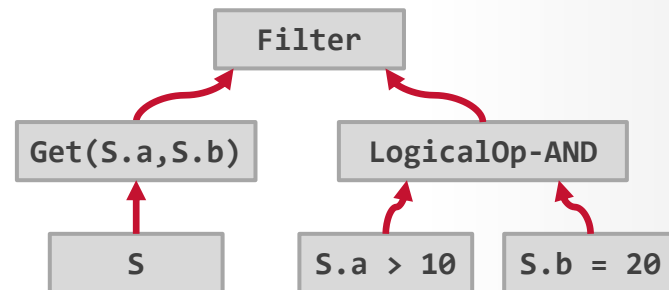
■ Enforcer

```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a_cb ON S(a)
INCLUDE (c,b);
```

```
CREATE INDEX idx_b_ca ON S(b)
INCLUDE (c,a);
```

```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```



SINGLE ACCESS METHOD

□ Logical Op

■ Physical Op

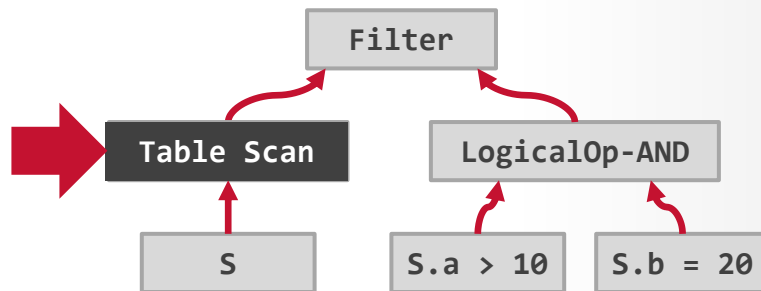
■ Enforcer

```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a_cb ON S(a)
INCLUDE (c,b);
```

```
CREATE INDEX idx_b_ca ON S(b)
INCLUDE (c,a);
```

```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```



SINGLE ACCESS METHOD

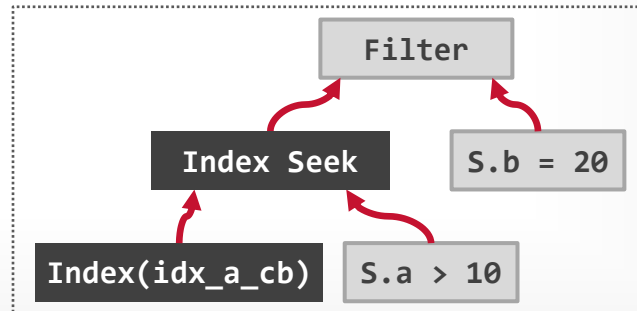
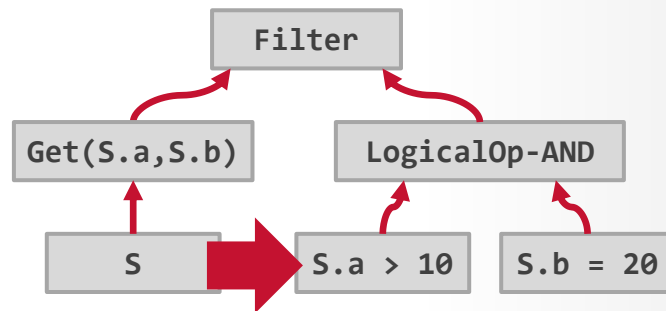
- Logical Op
- Physical Op
- Enforcer

```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a_cb ON S(a)
INCLUDE (c,b);
```

```
CREATE INDEX idx_b_ca ON S(b)
INCLUDE (c,a);
```

```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```



SINGLE ACCESS METHOD

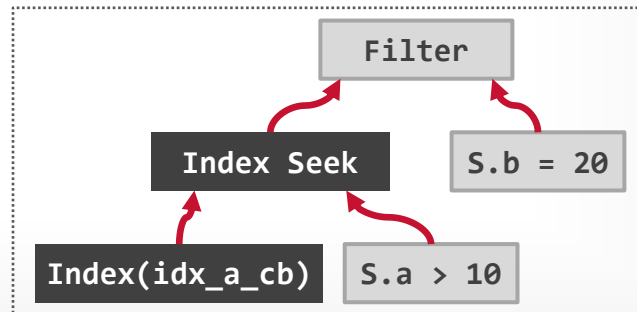
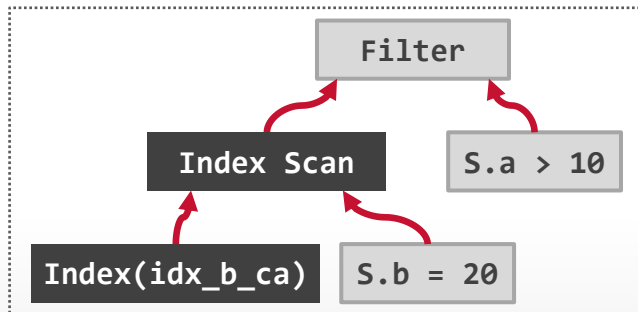
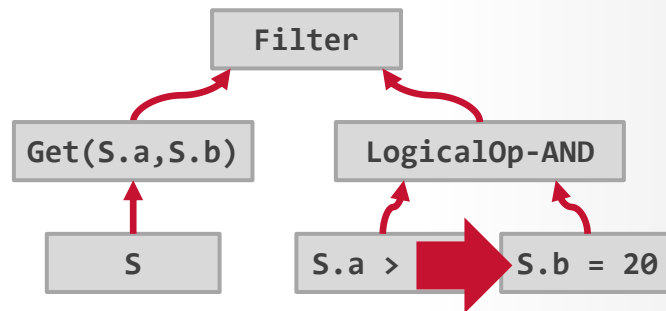
- Logical Op
- Physical Op
- Enforcer

```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a_cb ON S(a)
INCLUDE (c,b);
```

```
CREATE INDEX idx_b_ca ON S(b)
INCLUDE (c,a);
```

```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```



MULTIPLE ACCESS METHODS

If there are multiple access methods available for a query, the optimizer does not have to pick only one:
→ May be a combination of multiple indexes (common) and/or table scan (rare).

Represent multiple access methods as a logical self-join on the target relation.

Convert this self-join into a regular join or use a special multi-index scans physical operator
→ Example: [PostgreSQL Bitmap Scan](#)

MULTIPLE ACCESS METHODS

☐ *Logical Op*

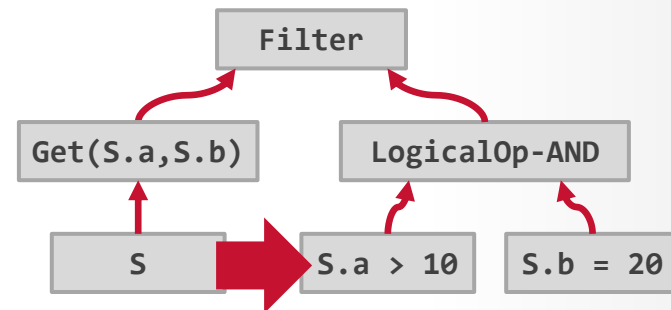
■ *Physical Op*

■ *Enforcer*

```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a ON S(a);
```

```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```

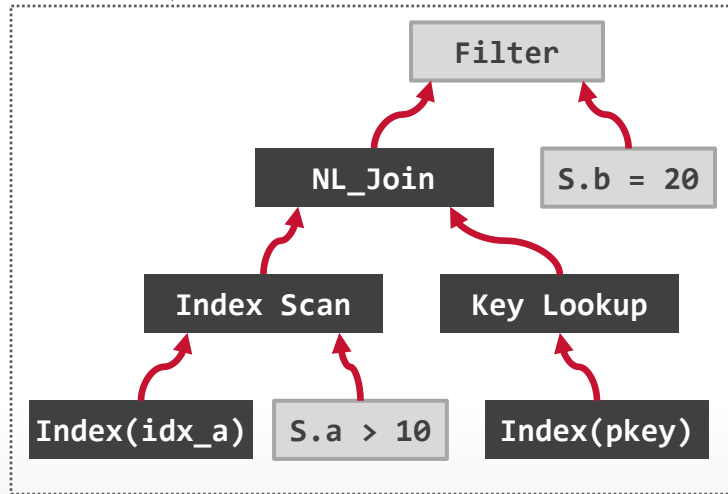


MULTIPLE ACCESS METHODS

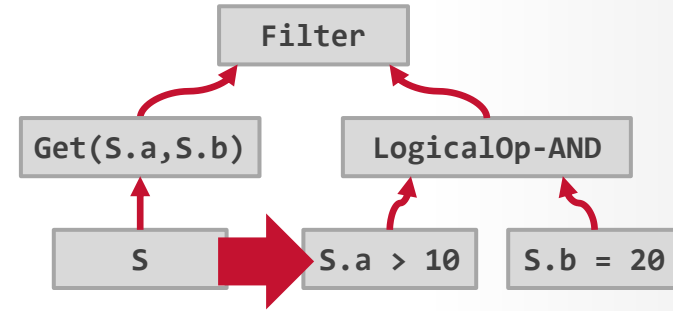
- Logical Op
- Physical Op
- Enforcer

```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a ON S(a);
```



```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```

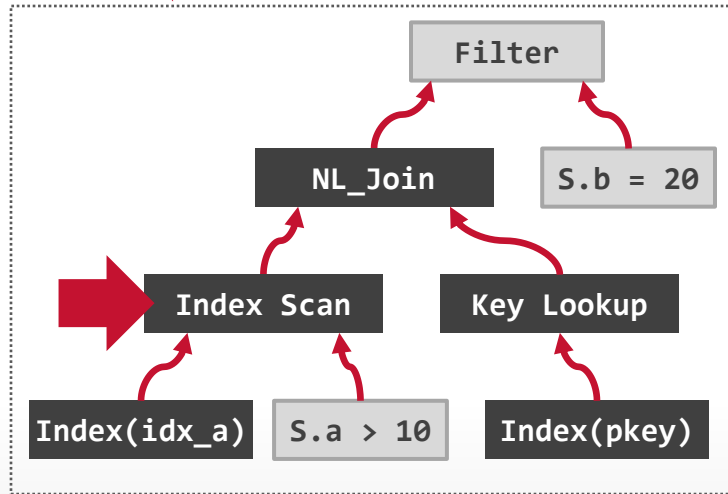


MULTIPLE ACCESS METHODS

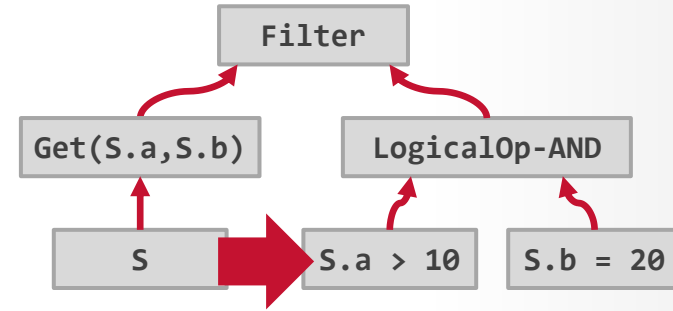
- Logical Op
- Physical Op
- Enforcer

```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a ON S(a);
```



```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```



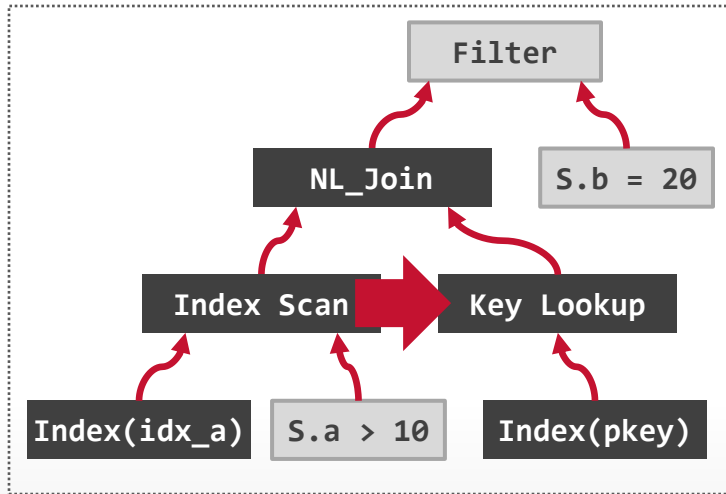
MULTIPLE ACCESS METHODS

- Logical Op
- Physical Op
- Enforcer

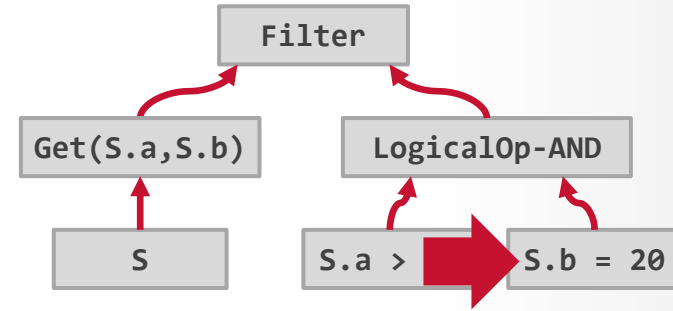
```
CREATE TABLE S (
  id INT PRIMARY KEY,
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL );
```

```
CREATE INDEX idx_a ON S(a);
```

Physical Data Structure?
Selectivity of Predicate?



```
SELECT S.a, S.b FROM S
WHERE S.a > 10
AND S.b = 20;
```



INNER JOIN

Generate different join orderings for the query.

→ Inner equi-joins are the most common, but optimizer needs to handle corner cases (e.g., anti-joins).

The commutativity and associativity rules will generate duplicate expressions.

→ Cascades' memo table ensures that the search algorithm does not waste time exploring those duplicates.

Join Enumeration via Transformations:

→ Space Complexity: $O(3^N)$

→ Computational Complexity: $O(4^N)$

INNER JOIN

Generate different join orderings for the query.

→ Inner equi-joins are the
needs to handle corner

The commutativity and
generate duplicate exp

→ Cascades' memo table
does not waste time ex

Rel	Linear join trees		Bushy join trees	
	Op	Duplicates	Op	Duplicates
2	1	1	2	1
3	6	10	12	10
4	22	47	50	71
5	65	166	180	416
6	171	517	602	2157
7	420	1422	1932	10326

Figure 5: Number of duplicates generated during the exploration of a MEMO-structure.

Join Enumeration via

→ Space Complexity: $O(3^N)$

→ Computational Complexity: $O(4^N)$

JOIN ENUMERATION GUIDANCE

Apply rules in a specific order and maintain a summary about derivation history of each operator to determine which rules to disable.

Leverage information about the behavior of rules to avoid unnecessary transformations.



JOIN ENUMERATION GUIDANCE

R1: Commutativity

→ $X \bowtie_0 Y \triangleright Y \bowtie_1 X$

→ Disable **R1**, **R2**, **R3**, **R4** on new operator \bowtie_1 .

R2: Right Associativity

→ $(X \bowtie_0 Y) \bowtie_1 Z \triangleright X \bowtie_2 (Y \bowtie_3 Z)$

→ Disable **R2**, **R3**, **R4** on new operator \bowtie_2 .

R3: Left Associativity

→ $X \bowtie_0 (Y \bowtie_1 Z) \triangleright (X \bowtie_2 Y) \bowtie_3 Z$

→ Disable rules **R2**, **R3**, **R4** on new operator \bowtie_3 .

R4: Exchange

→ $(W \bowtie_0 X) \bowtie_1 (Y \bowtie_2 Z) \triangleright (W \bowtie_3 Y) \bowtie_4 (X \bowtie_5 Z)$

→ Disable all rules **R1**, **R2**, **R3**, **R4** on \bowtie_4 .

JOIN ENUMERATION GUIDANCE

R1: Commutativity

→ $X \bowtie_0 Y \triangleright Y \bowtie_1 X$

→ Disable **R1, R2, R3, R4** on new operator \bowtie_1 .



R2: Right Associativity

→ $(X \bowtie_0 Y) \bowtie_1 Z \triangleright X \bowtie_2 (Y \bowtie_3 Z)$

→ Disable **R2, R3, R4** on new operator \bowtie_2 .



R3: Left Associativity

→ $X \bowtie_0 (Y \bowtie_1 Z) \triangleright (X \bowtie_2 Y) \bowtie_3 Z$

→ Disable rules **R2, R3, R4** on new operator \bowtie_3 .



R4: Exchange

→ $(W \bowtie_0 X) \bowtie_1 (Y \bowtie_2 Z) \triangleright (W \bowtie_3 Y) \bowtie_4 (X \bowtie_5 Z)$

→ Disable all rules **R1, R2, R3, R4** on \bowtie_4 .

JOIN ENUMERATION GUIDANCE

R1: Commutativity

→ $X \bowtie_0 Y \triangleright Y \bowtie_1 X$

→ Disable **R1, R2, R3, R4** on new operator \bowtie_1 .

R2: Right Associativity

→ $(X \bowtie_0 Y) \bowtie_1 Z \triangleright X \bowtie_2 (Y \bowtie_3 Z)$

→ Disable **R2, R3, R4** on new operator \bowtie_2 .

R3: Left Associativity

→ $X \bowtie_0 (Y \bowtie_1 Z) \triangleright (X \bowtie_2 Y) \bowtie_3 Z$

→ Disable rules **R2, R3, R4** on new operator \bowtie_3 .

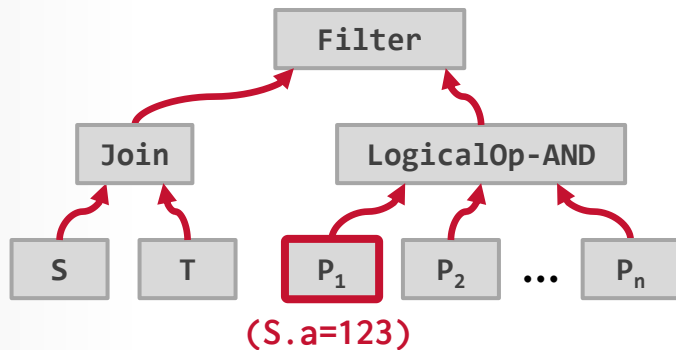
R4: Exchange

→ $(W \bowtie_0 X) \bowtie_1 (Y \bowtie_2 Z) \triangleright (W \bowtie_3 Y) \bowtie_4 (X \bowtie_5 Z)$

→ Disable all rules **R1, R2, R3, R4** on \bowtie_4 .

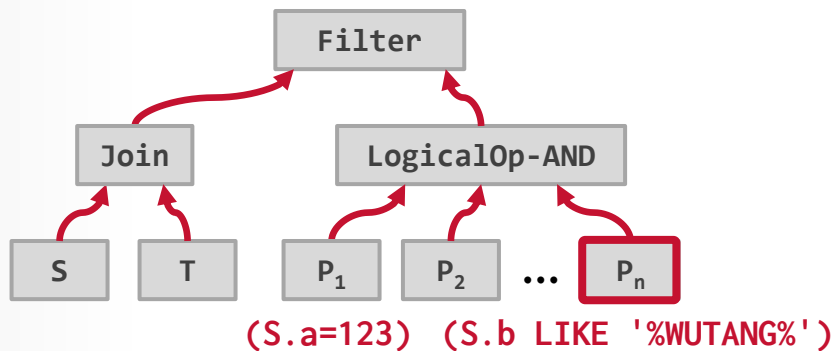
PREDICATE PUSHDOWN / PULLUP

A predicate that does not reference attributes in a join result can be moved to occur before or after a join based on its selectivity and computational cost.



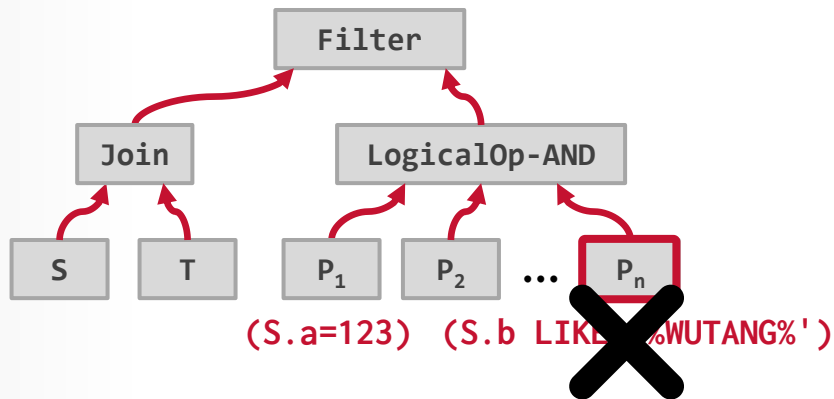
PREDICATE PUSHDOWN / PULLUP

A predicate that does not reference attributes in a join result can be moved to occur before or after a join based on its selectivity and computational cost.



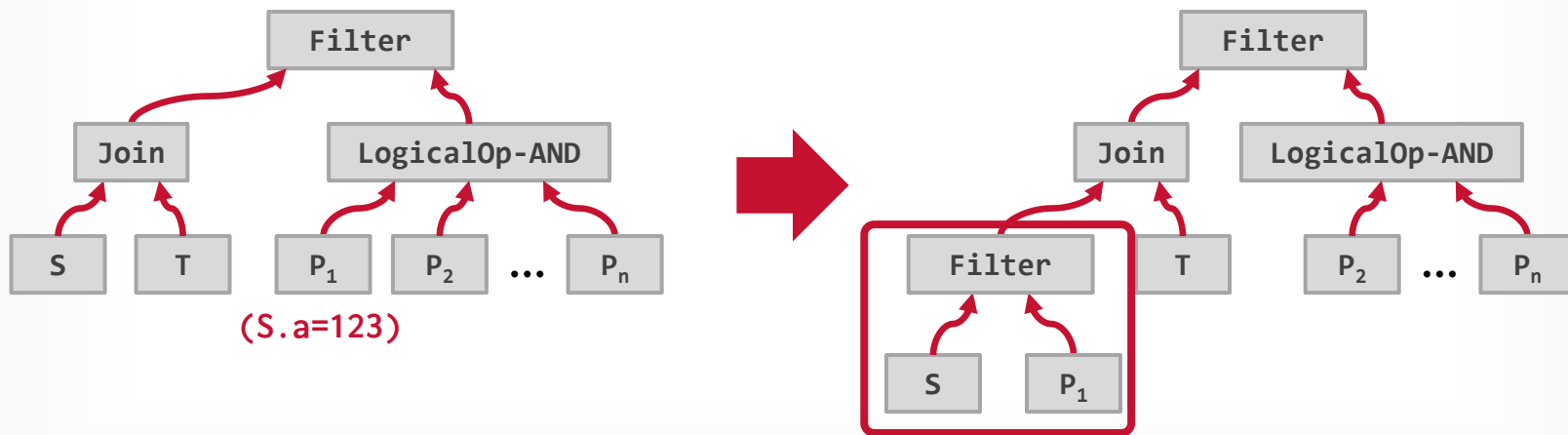
PREDICATE PUSHDOWN / PULLUP

A predicate that does not reference attributes in a join result can be moved to occur before or after a join based on its selectivity and computational cost.



PREDICATE PUSHDOWN / PULLUP

A predicate that does not reference attributes in a join result can be moved to occur before or after a join based on its selectivity and computational cost.



PHYSICAL OPERATOR TRANSFORMATION

The selection of a physical join operator depends on the join predicates and the layout of data.

- Hash Joins: Can only be used for equi-joins.
- Merge Join: Input data must be sorted on join key(s).
- Nested Loop Join: Fallback option.

The optimizer also selects runtime operator parameters during this process.

- Example: Hash table size

OUTER JOIN

Like inner joins, different orderings for outer joins have wildly different costs.

But unlike inner joins, transformation rules do not always hold for outer joins.

→ Different orderings retain different non-matching tuples.

The goal is to preserve the same non-matching tuples and NULL values for attributes.

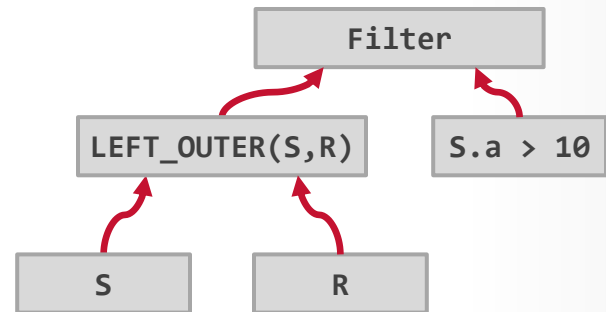
REDUNDANCY RULE

Some outerjoins can safely be replaced by inner joins if the query contains a null-rejecting predicate

→ Any predicate that filters NULL values.

Converting an outer join to an inner join unlocks additional optimizations.

```
SELECT S.a
FROM S LEFT OUTER JOIN R
ON S.id = R.id
WHERE S.a > 10;
```



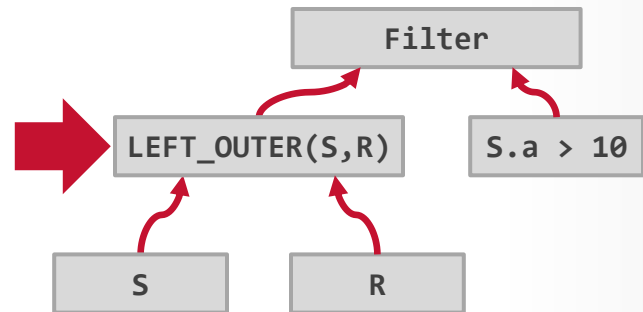
REDUNDANCY RULE

Some outerjoins can safely be replaced by inner joins if the query contains a null-rejecting predicate

→ Any predicate that filters NULL values.

Converting an outer join to an inner join unlocks additional optimizations.

```
SELECT S.a
FROM S LEFT OUTER JOIN R
ON S.id = R.id
WHERE S.a > 10;
```



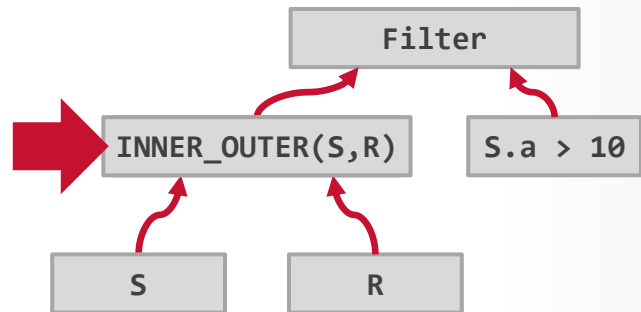
REDUNDANCY RULE

Some outerjoins can safely be replaced by inner joins if the query contains a null-rejecting predicate

→ Any predicate that filters NULL values.

Converting an outer join to an inner join unlocks additional optimizations.

```
SELECT S.a
FROM S LEFT OUTER JOIN R
ON S.id = R.id
WHERE S.a > 10;
```



GROUP-BY

When a query computes an aggregation on the result of joining two or more tables, it may be better to perform the aggregation first before the join and then join the tables on the aggregation result.

Two Cases:

- **Complete Group-By Pushdown**
- **Partial Group-By Pushdown**

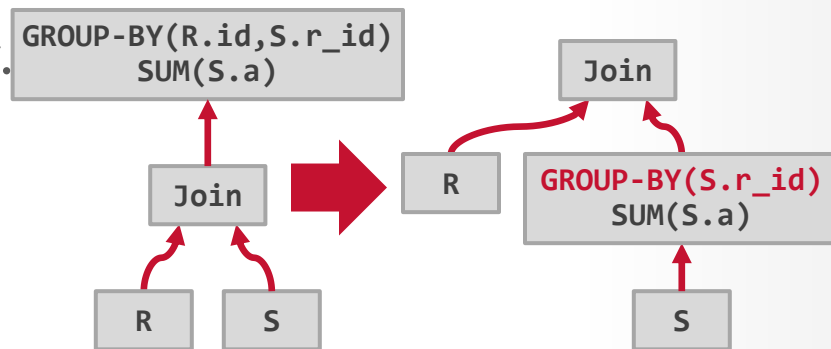
COMPLETE GROUP-BY PUSHDOWN

All aggregate functions in a group-by operator only use columns in **S**.

The primary key of **R** is a subset of the columns referenced in a group-by.

The columns referenced in a pushed down group-by operator is the union of columns in original group-by and equi-join columns of **S** in **R ⋈ S**.

```
SELECT SUM(R.a)
FROM R JOIN S
ON R.id = S.r_id
GROUP BY R.id, S.r_id;
```



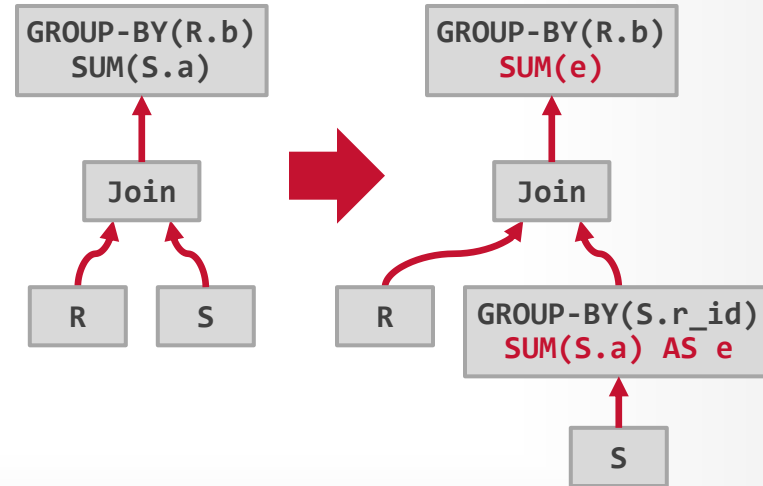
PARTIAL GROUP-BY PUSHDOWN

It may not always be possible to move an aggregation below a join.

Instead, the optimizer can create a new group-by operator that computes a portion of the aggregation.

→ Partial aggregation reduces the cardinality of the input relation to the join.

```
SELECT SUM(S.a)
FROM R JOIN S
ON R.id = S.r_id
GROUP BY R.b;
```



OBSERVATION

The optimizer can detect whether a query is targeting a database with a common design pattern and invoke transformations that push a query plan into an ideal form.

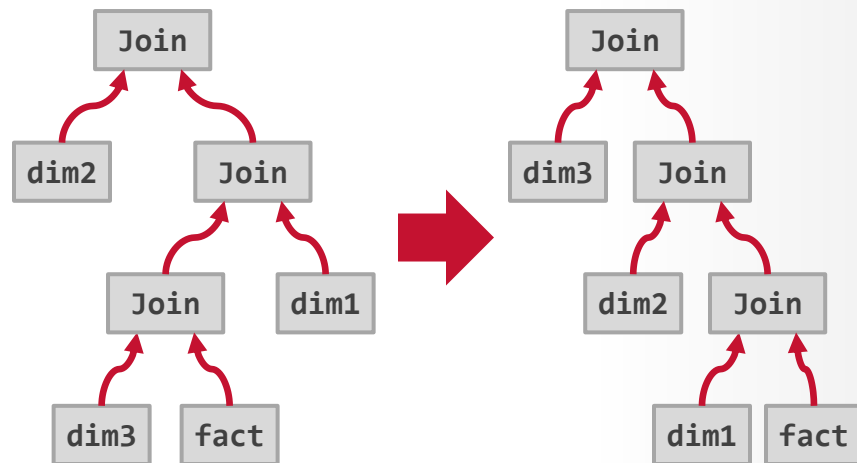
We saw this before with sargable queries where the optimizer can immediately select the best index.

STAR / SNOWFLAKE QUERIES

If a query joins a fact table with multiple dimension tables, then transform it to a left/right-deep join tree and order dimension tables from most to least selective.

Avoid wasting time exploring bushy plans or alternative join orderings for dimension tables.

```
SELECT * FROM fact AS F
  JOIN dim1 ON F.d1 = dim1.id
  JOIN dim2 ON F.d2 = dim2.id
  JOIN dim3 ON F.d3 = dim3.id;
```



PARTING THOUGHTS

It takes years to create a comprehensive library of transformation rules to handle all query plan shapes.

→ Even the best systems miss many opportunities.

→ Search for "\$DBMS bad query plan"

One project could be creating an open-source repository of transformation rules.

We did not discuss nested subquery decorrelation because it is very hard and Microsoft's approach (as described in the book) is incomplete.

NEXT CLASS

Switching back to bottom-up optimization to discuss join enumeration alternatives.

This will be your first (of many) encounter with a paper from the Germans this semester...