

**CSAI 302**

# **Advanced Database**

## **Systems**

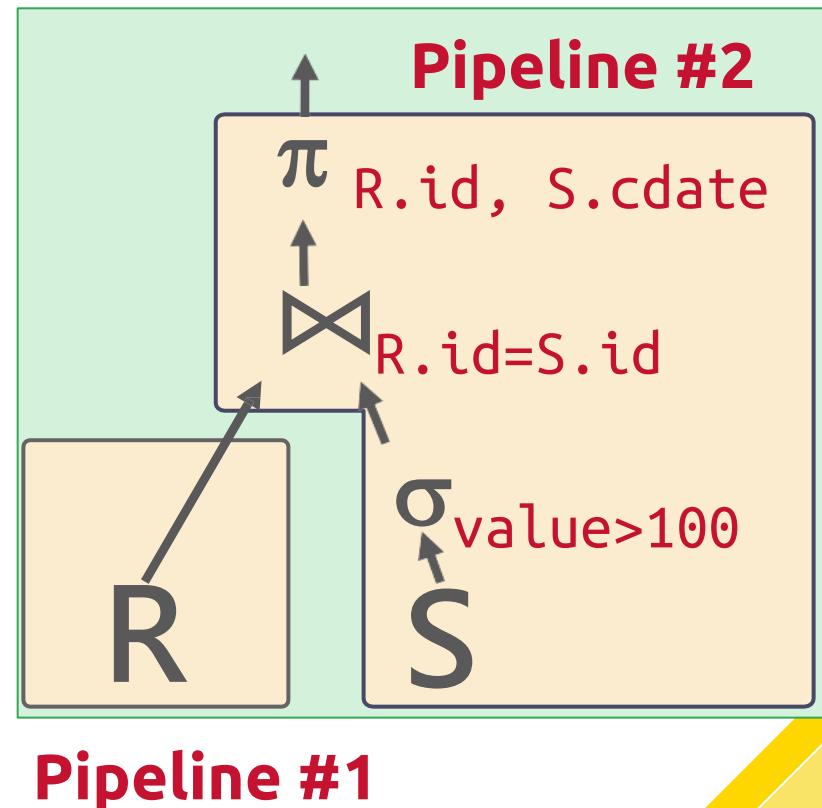
**Lec 05**

**Query Processing  
& Optimization [1]**

# QUERY EXECUTION

- ◆ A query plan is a DAG of **operators**.
- ◆ A **pipeline** is a sequence of operators where tuples continuously flow between them without intermediate storage.
- ◆ A **pipeline breaker** is an operator that cannot finish until all its children emit all their tuples.
  - ★ Joins (Build Side), Subqueries, Order By

```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



# Query Execution

# Processing Model

- ◆ A DBMS's processing model defines how the system executes a query plan and moves data from one operator to the next.
  - ★ Different trade-offs for workloads (OLTP vs. OLAP).
- ◆ Each processing model has two types of execution paths:
  - ★ Control Flow
    - How the DBMS invokes an operator.
  - ★ Data Flow
    - How an operator sends its results.
- ◆ The output of an operator can be either whole tuples (NSM) or subsets of columns (DSM).

# Processing Model

## Iterator Model

- Most Common

## Vectorized / Batch Model

- Common

## Materialization Model

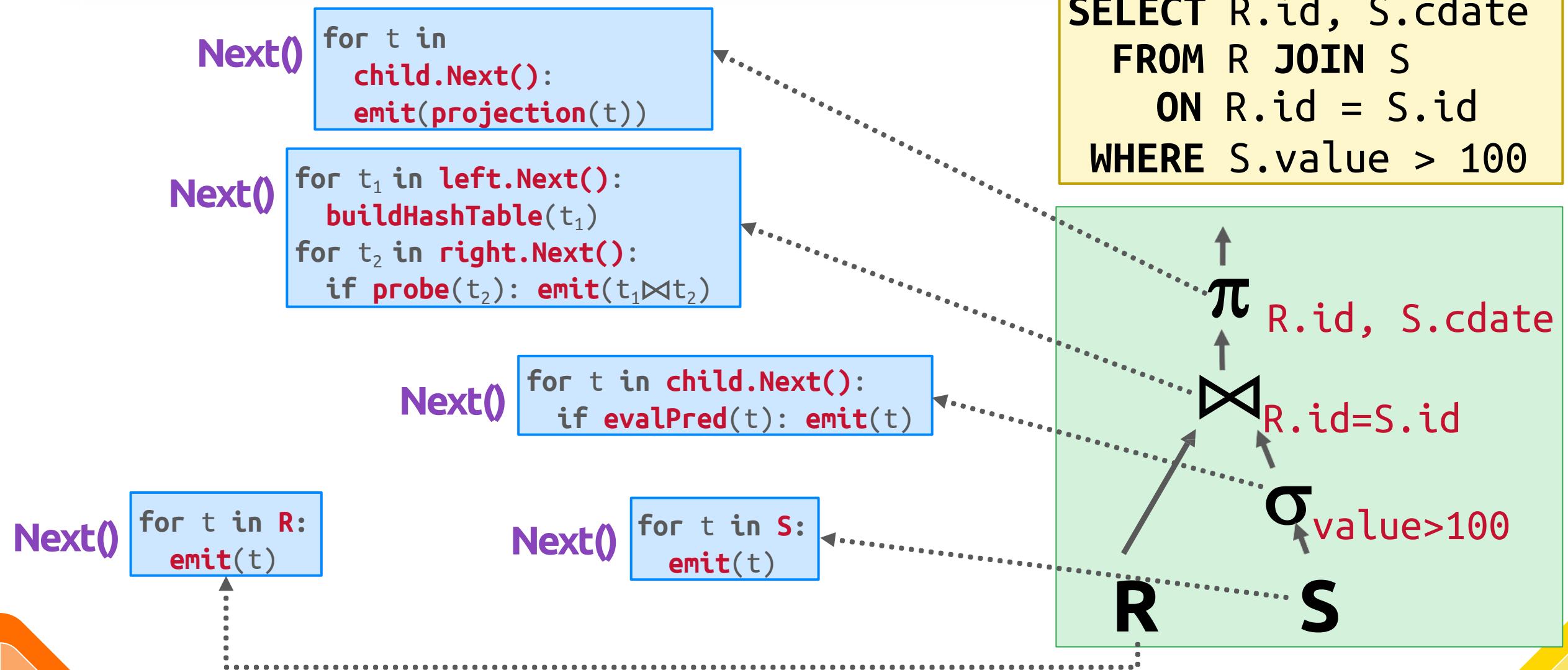
- Rare

# Iterator Model

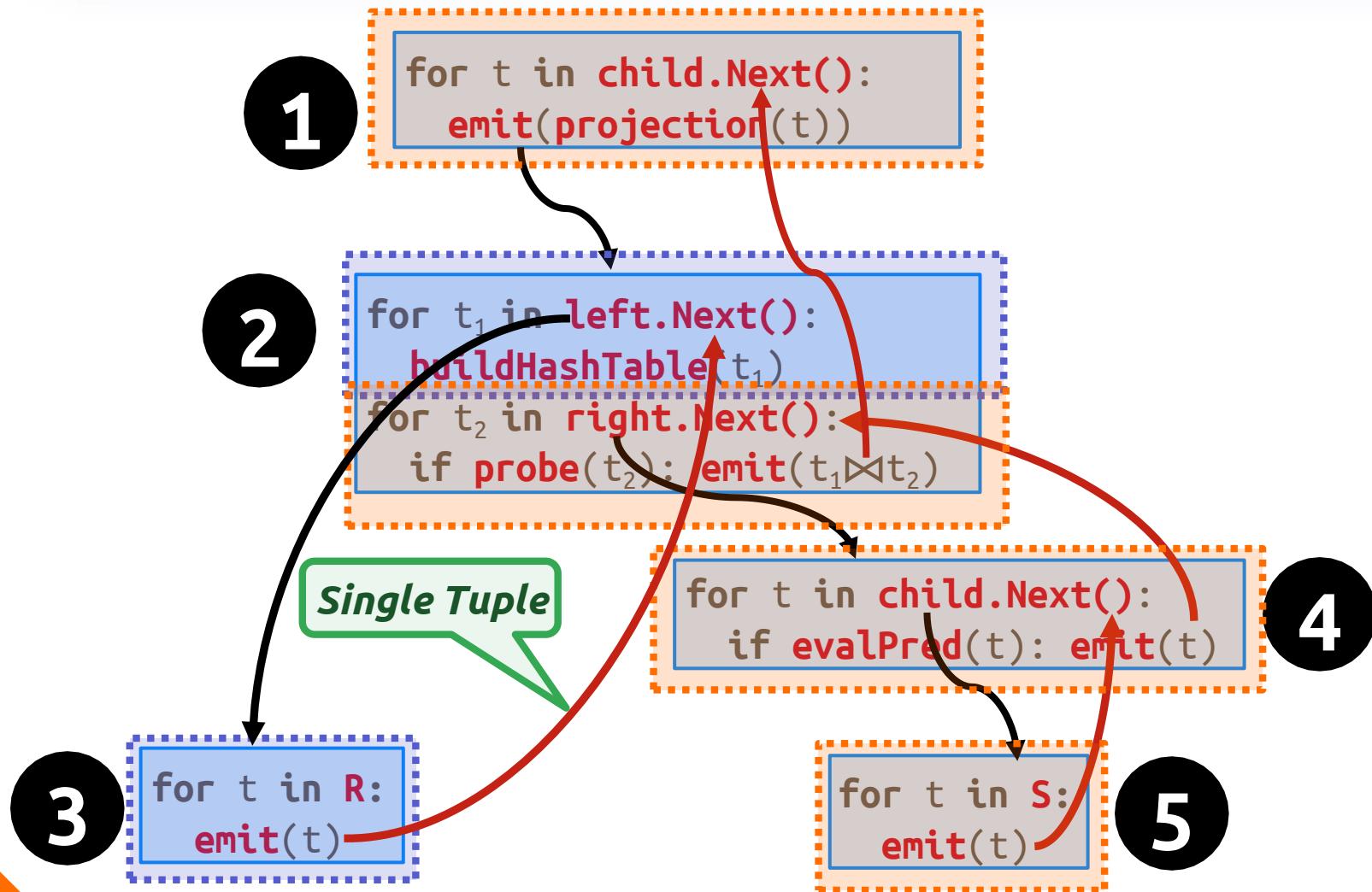
- ◆ Each query plan operator implements a **Next()** function.
  - ★ On each invocation, the operator returns either a single tuple or a EOF marker if there are no more tuples.
  - ★ The operator implements a loop that calls **Next()** on its children to retrieve their tuples and then process them.
- ◆ Each operator implementation also has **Open()** and **Close()** functions.

Also called **Volcano** or **Pipeline** Model.

# Iterator model

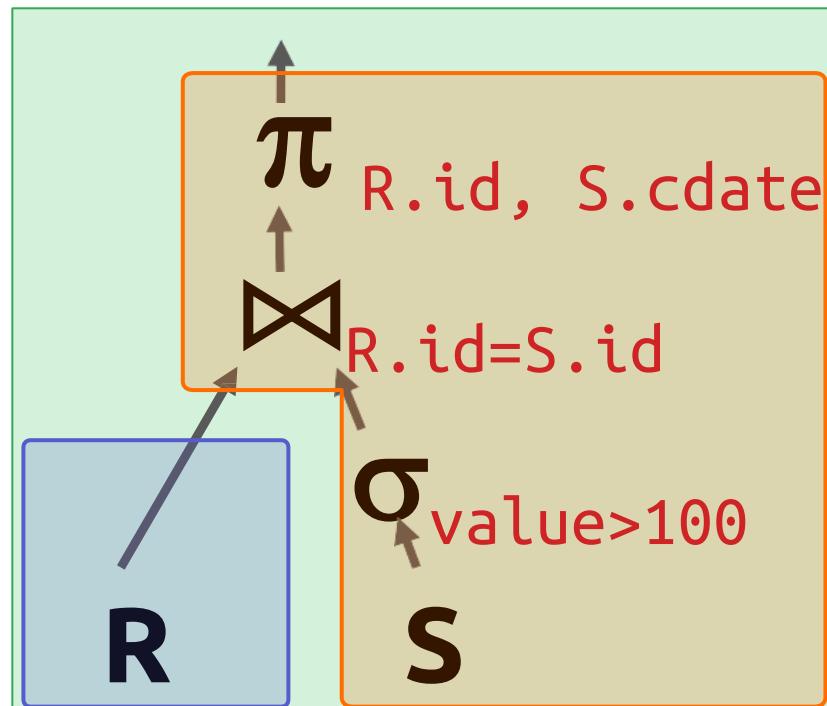


# Iterator model



→ Control Flow → Data Flow

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



# Iterator model

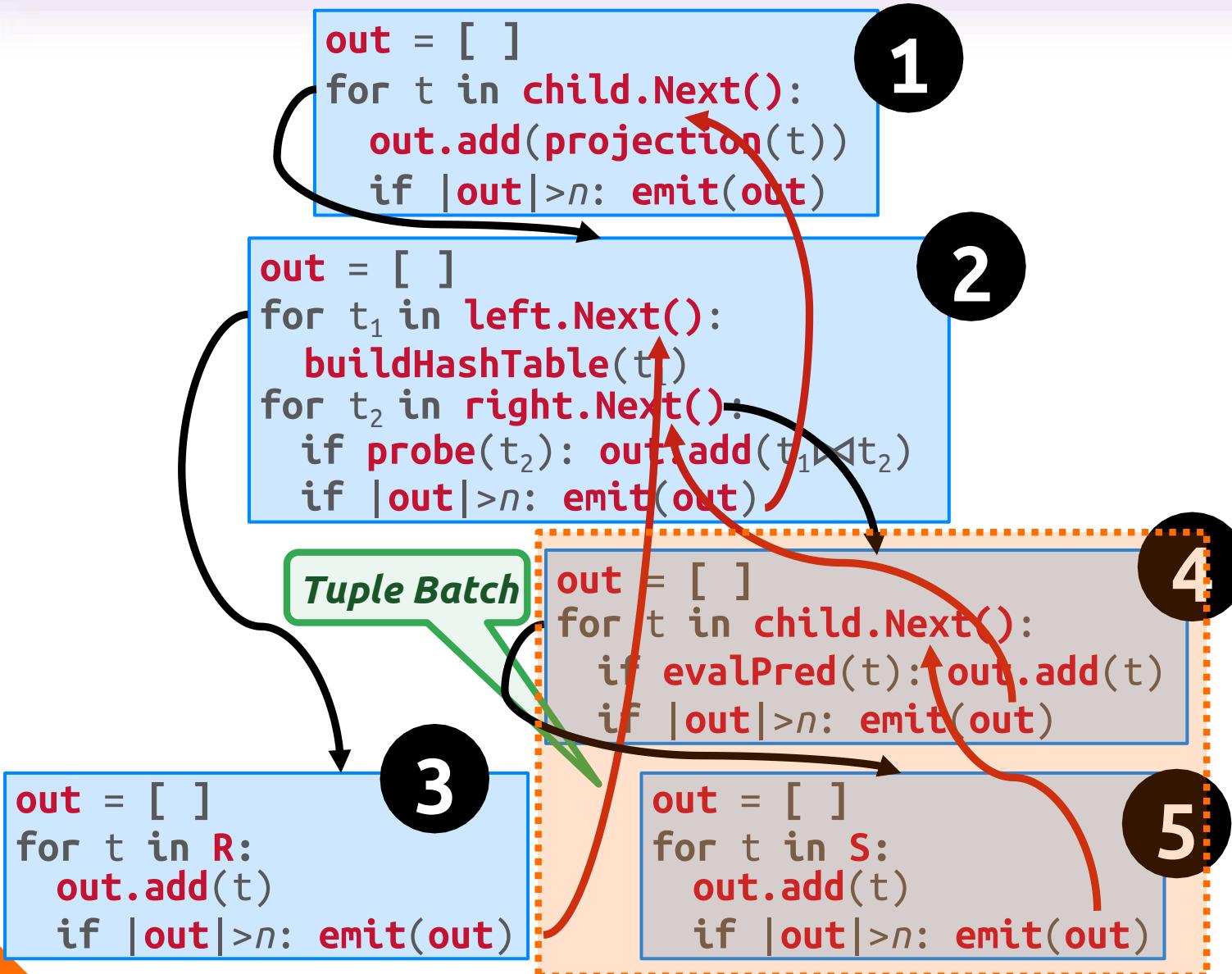
- ◆ The Iterator model is used in almost every DBMS.
  - ★ Easy to implement / debug.
  - ★ Output control works easily with this approach.
- ◆ Allows for pipelining where the DBMS tries to process each tuple through as many operators as possible before retrieving the next tuple.



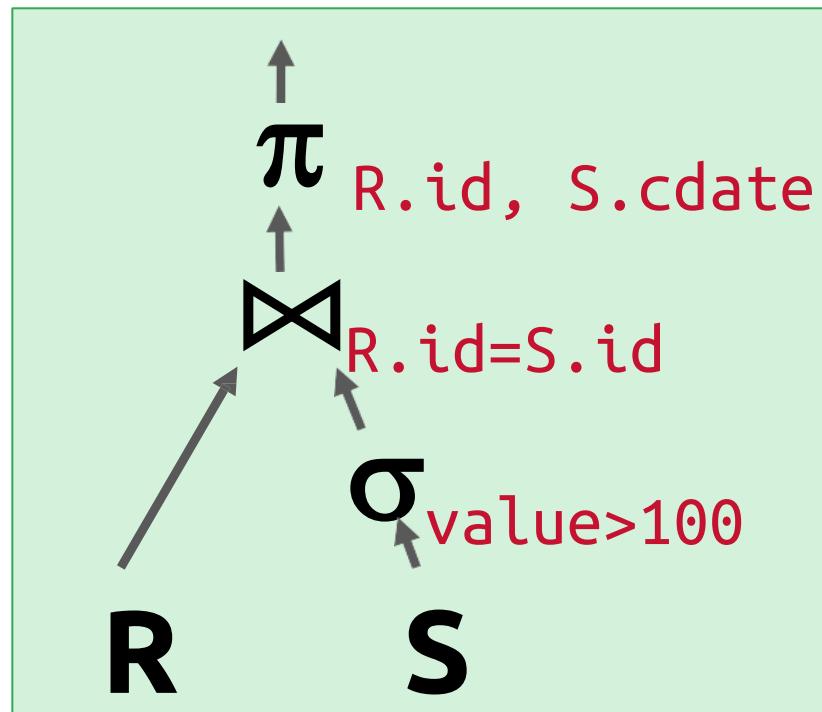
# Vectorization model

- ◆ Like the Iterator Model where each operator implements a **Next()** function, but...
- ◆ Each operator emits a **batch** of tuples instead of a single tuple.
  - ★ The operator's internal loop processes multiple tuples at a time.
  - ★ The size of the batch can vary based on hardware or query properties.
  - ★ Each batch will contain one or more columns each their own null bitmaps.

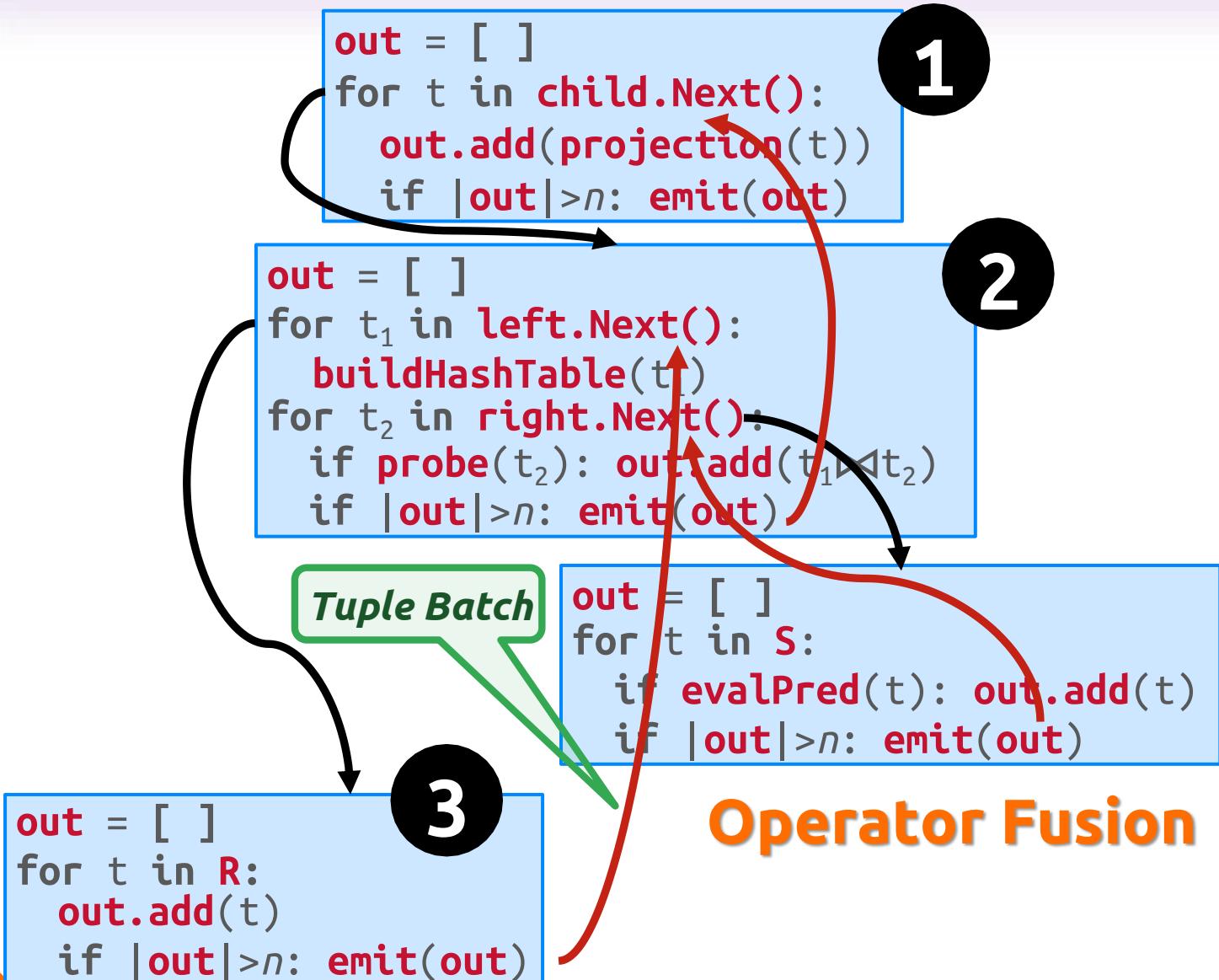
# Vectorization model



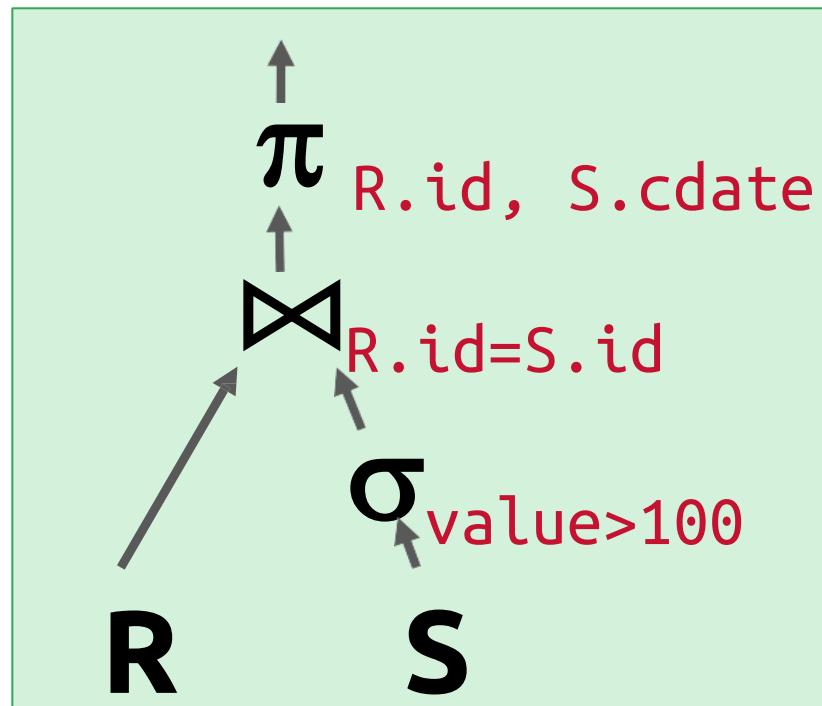
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



# Vectorization model



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



# Vectorization model

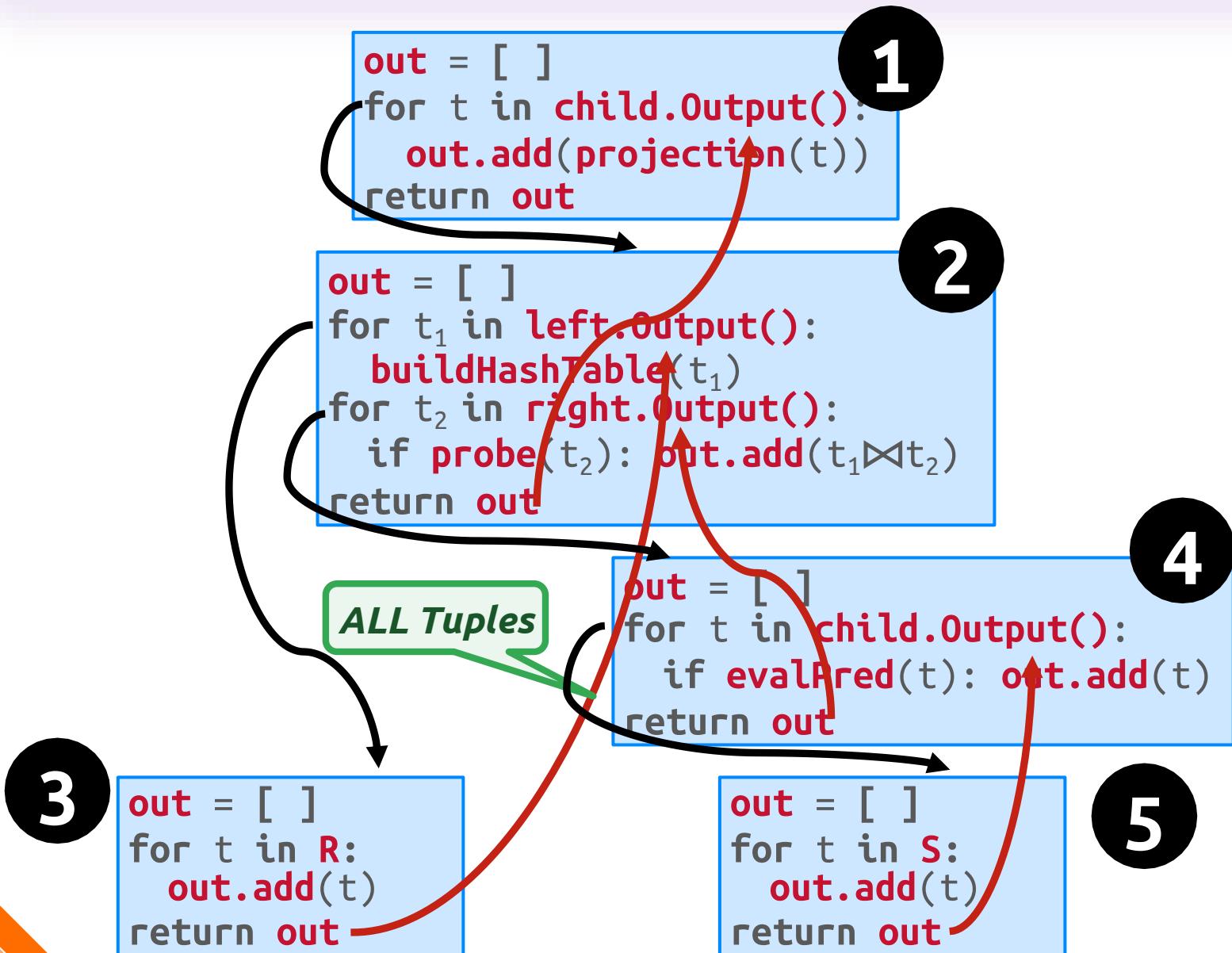
- ◆ Ideal for OLAP queries because it greatly reduces the number of invocations per operator.
- ◆ Allows an out-of-order CPU to efficiently execute operators over batches of tuples.
  - ★ Operators perform work in tight for-loops over arrays, which compilers know how to optimize / vectorize.
  - ★ No data or control dependencies.
  - ★ Hot instruction cache.



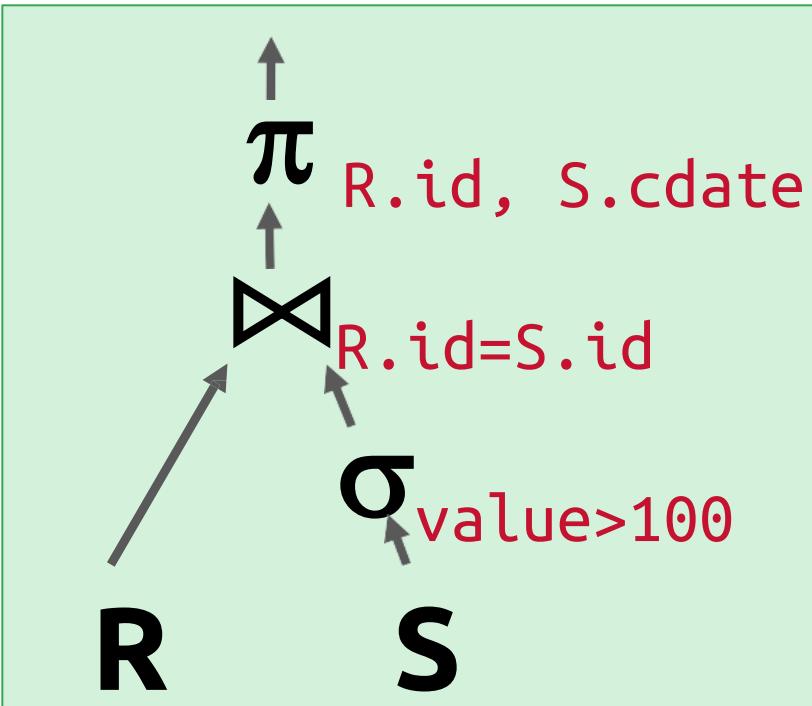
# Materialization model

- ◆ Each operator processes its input all at once and then emits its output all at once.
  - ★ The operator "materializes" its output as a single result.
  - ★ The DBMS can push down hints (e.g., **LIMIT**) to avoid scanning too many tuples.
  - ★ Can send either a materialized row or a single column.
- ◆ The output can be either whole tuples (NSM) or subsets of columns (DSM).

# Materialization model



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



# Materialization model

- ◆ Better for OLTP workloads because queries only access a small number of tuples at a time.
  - ★ → Lower execution / coordination overhead.
  - ★ → Fewer function calls.
- ◆ Not ideal for OLAP queries with large intermediate results because DBMS must allocate buffers.



# **Plan Processing**

# Plan Processing Direction

## ◆ Top-to-Bottom (Pull)

- ★ Start with the root and "pull" data up from its children.
- ★ Tuples are always passed between operators using function calls (unless it's a pipeline breaker).

## ◆ Bottom-to-Top (Push)

- ★ Start with leaf nodes and "push" data to their parents.
- ★ Can "fuse" operators together within a for-loop to minimize intermediate result staging.



# Access methods

- ◆ An access method is the way that the DBMS accesses the data stored in a table.
  - ★ Not defined in relational algebra.
- ◆ Three basic approaches:
  - ★ Sequential Scan.
  - ★ Index Scan (many variants).
  - ★ Multi-Index Scan.

# SEQUENTIAL SCAN

- ◆ For each page in the table:
  - ★ Retrieve it from the buffer pool.
  - ★ Iterate over each tuple and check whether to include it.
- ◆ The DBMS maintains an internal cursor that tracks the last page examined.

```
for page in table.pages:  
    for t in page.tuples:  
        if evalPred(t):  
            // Do Something!
```

# Sequential scan: Optimizations

Data Encoding /  
Compression

Prefetching /  
Scan Sharing /  
Buffer Bypass

Task  
Parallelization /  
Multi-threading

Clustering /  
Sorting

Late  
Materialization

Materialized  
Views / Result  
Caching

Data Skipping

Data  
Parallelization /  
Vectorization

Code  
Specialization /  
Compilation

# Data skipping

- ◆ Approximate Queries (Lossy)
  - ★ Execute queries on a ***sampled subset*** of the entire table to produce approximate results.
- ◆ Zone Maps (Lossless)
  - ★ Pre-compute ***columnar aggregations per page*** that allow the DBMS to check whether queries need to access it.
  - ★ Trade-off between page size vs. filter efficacy.

# Zone maps

- ◆ Pre-computed aggregates for the attribute values in a page.
- ◆ DBMS checks the zone map first to decide whether it wants to access the page.

```
SELECT * FROM table  
WHERE val > 600
```

Original Data

val
100
200
300
400
400



Zone Map

type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

# Index scan

- ◆ The DBMS picks an index to find the tuples that the query needs.
- ◆ Which index to use depends on:
  - ★ What attributes the index contains
  - ★ What attributes the query references
  - ★ The attribute's value domains
  - ★ Predicate composition
  - ★ Whether the index has unique or non-unique keys

# Index scan

- ◆ Suppose that we have a single table with 100 tuples and two indexes:

- ★ Index #1: **age**

- ★ Index #2: **dept**

```
SELECT * FROM students  
WHERE age < 30  
    AND dept = 'CS'  
    AND country = 'US'
```

## *Scenario #1*

There are 99 people under the age of 30 but only 2 people *in the CS department.*

## *Scenario #2*

There are 99 people in the CS department but only 2 people *under the age of 30.*

# Multi-index scan

- ◆ If there are multiple indexes available for a query, the DBMS does not have to pick only one:
  - ★ Compute sets of Record IDs using each matching index.
  - ★ Combine these sets based on the query's predicates (union vs. intersect).
  - ★ Retrieve the records and apply any remaining predicates.
- ◆ Examples:
  - ★ DB2 Multi-Index Scan
  - ★ PostgreSQL Bitmap Scan
  - ★ MySQL Index Merge

# Multi-index scan

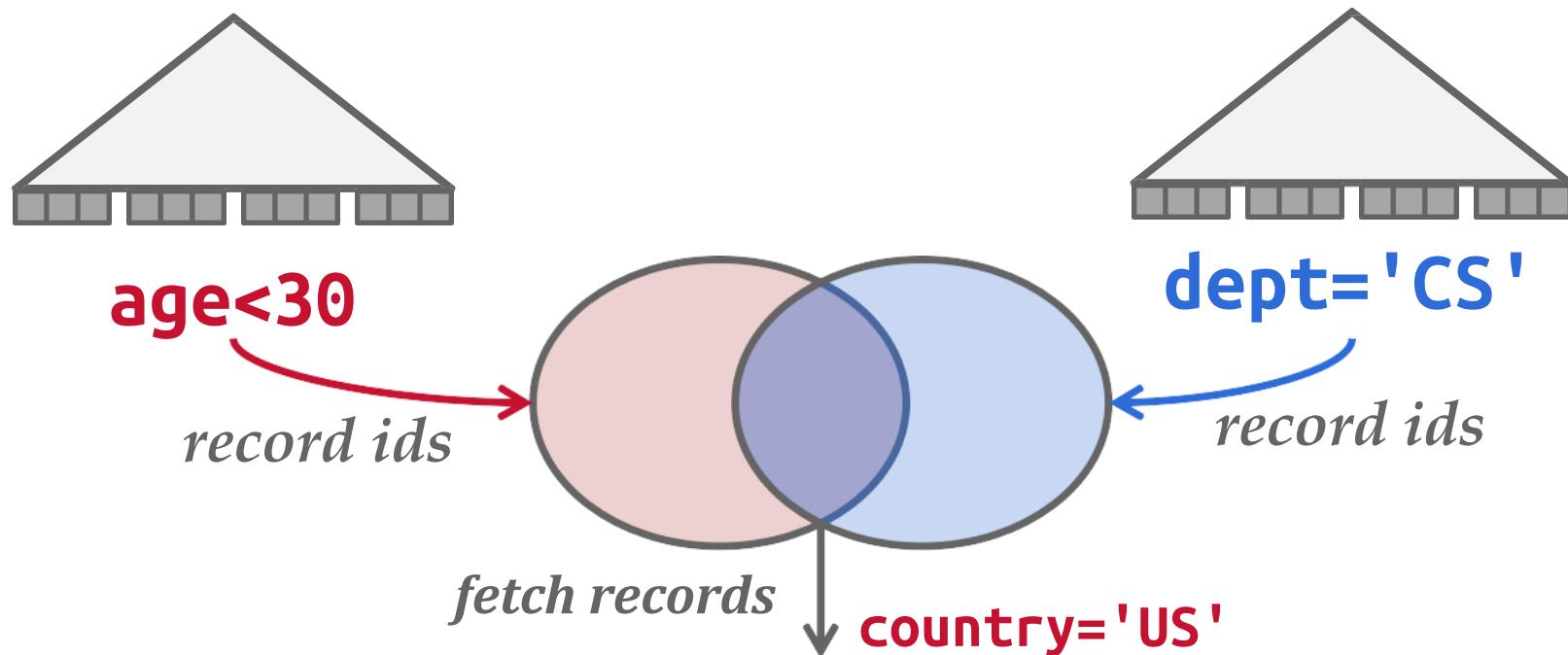
- Given the following query on a database with an index #1 on **age** and an index #2 on **dept**:
  - Retrieve the Record IDs satisfying **age<30** using index #1.
  - Retrieve the Record IDs satisfying **dept='CS'** using index #2.
  - Take their intersection.
  - Retrieve records and check **country='US'**.

```
SELECT * FROM students  
WHERE age < 30  
      AND dept = 'CS'  
      AND country = 'US'
```

# Multi-index scan

Set intersection can be done efficiently with bitmaps or hash tables.

```
SELECT * FROM students  
WHERE age < 30  
    AND dept = 'CS'  
    AND country = 'US'
```



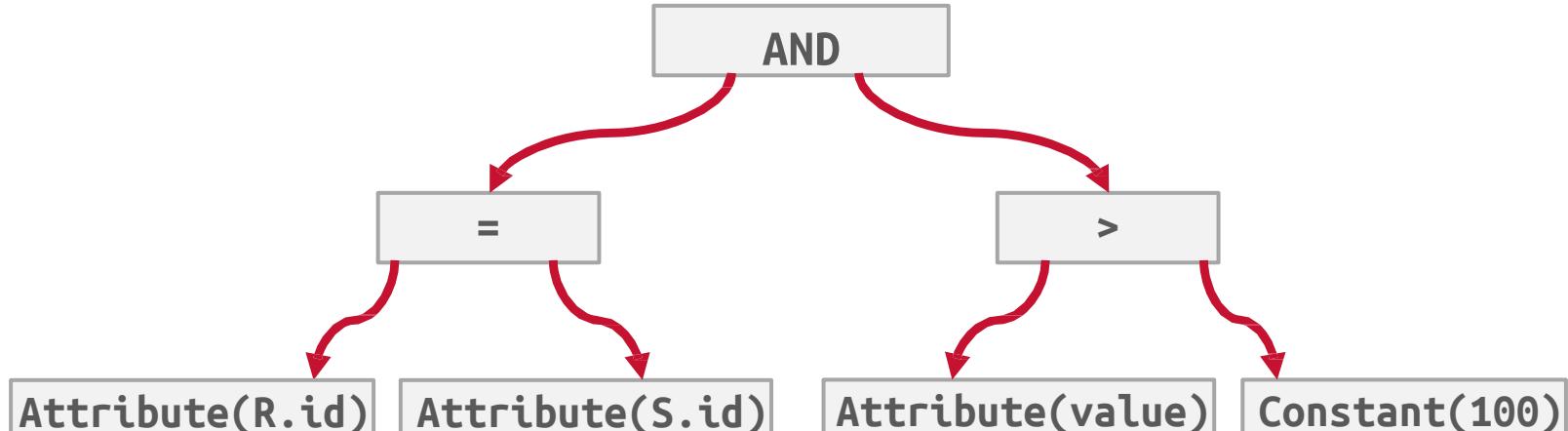
# **Expression evaluation**

# Expression evaluation

- ◆ The DBMS represents a WHERE clause as an expression tree.
- ◆ The nodes in the tree represent different expression types:
  - ★ Comparisons ( $=, <, >, !=$ )
  - ★ Conjunction (AND), Disjunction (OR)
  - ★ Arithmetic Operators ( $+, -, *, /, \%$ )
  - ★ Constant Values
  - ★ Tuple Attribute References
  - ★ Functions

# Expression evaluation

```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100;
```

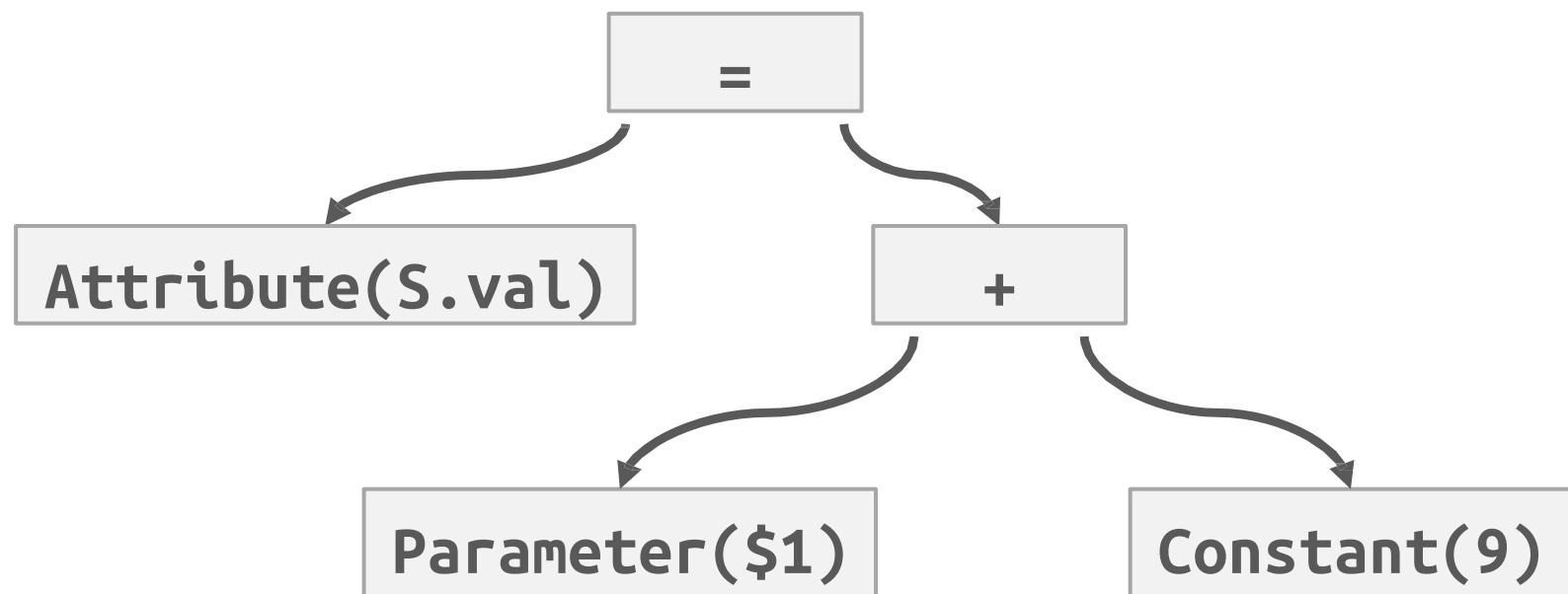


# Expression evaluation

```
PREPARE xxx AS  
SELECT * FROM S  
WHERE S.val = $1 + 9  
  
EXECUTE xxx(991)
```

## Execution Context

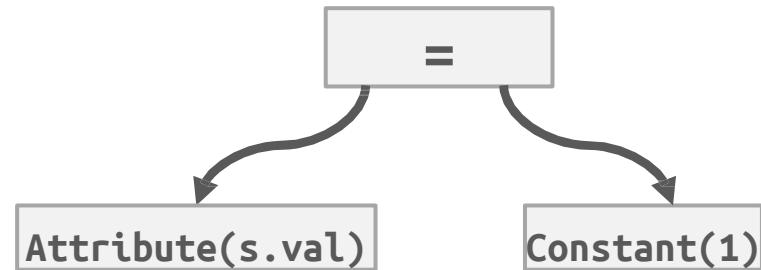
Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:991)	S→(int:id, int:val)



# Expression evaluation

- ◆ Evaluating predicates by traversing a tree is terrible for the CPU.
  - ★ The DBMS traverses the tree and for each node that it visits, it must figure out what the operator needs to do.
- ◆ A better approach is to evaluate the expression directly.
- ◆ An even better approach is to **vectorize** it evaluate a batch of tuples at the same time...

```
SELECT * WHERE s.val = 1;
```



```
bool check(val) {  
    return (val == 1);  
}
```



*gcc, Clang, LLVM,*

*Machine Code*

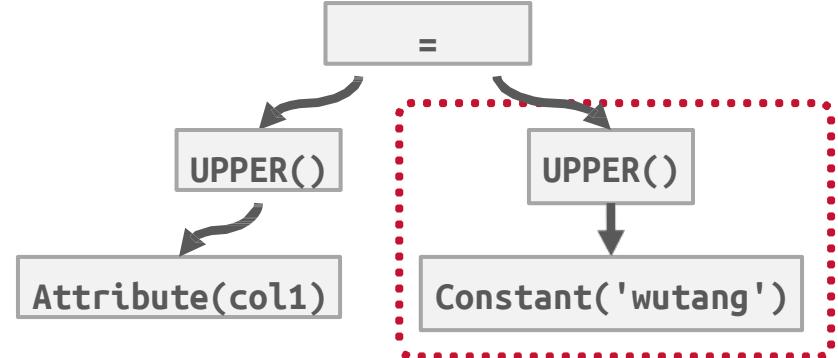
# **EXPRESSION EVALUATION: OPTIMIZATIONS**

- ◆ Constant Folding:
  - ◆ → Identify redundant / unnecessary operations that are wasteful.
  - ◆ → Compute a sub-expression on a constant value once and reuse result per tuple.
- ◆ Common Sub-Expr. Elimination: Identify repeated sub-expressions that can be shared across expression tree.
  - ◆ Compute once and then reuse result.

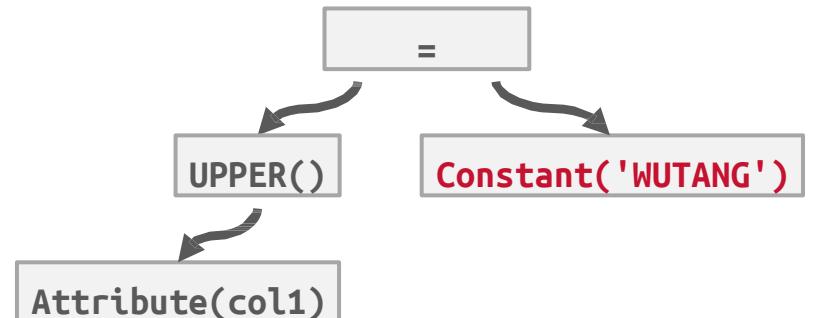
# Constant Folding

- ◆ Identify redundant / unnecessary operations that are wasteful.
- ◆ Compute a sub-expression on a constant
- ◆ value once and reuse result per tuple.

```
WHERE UPPER(col1) = UPPER('wutang');
```



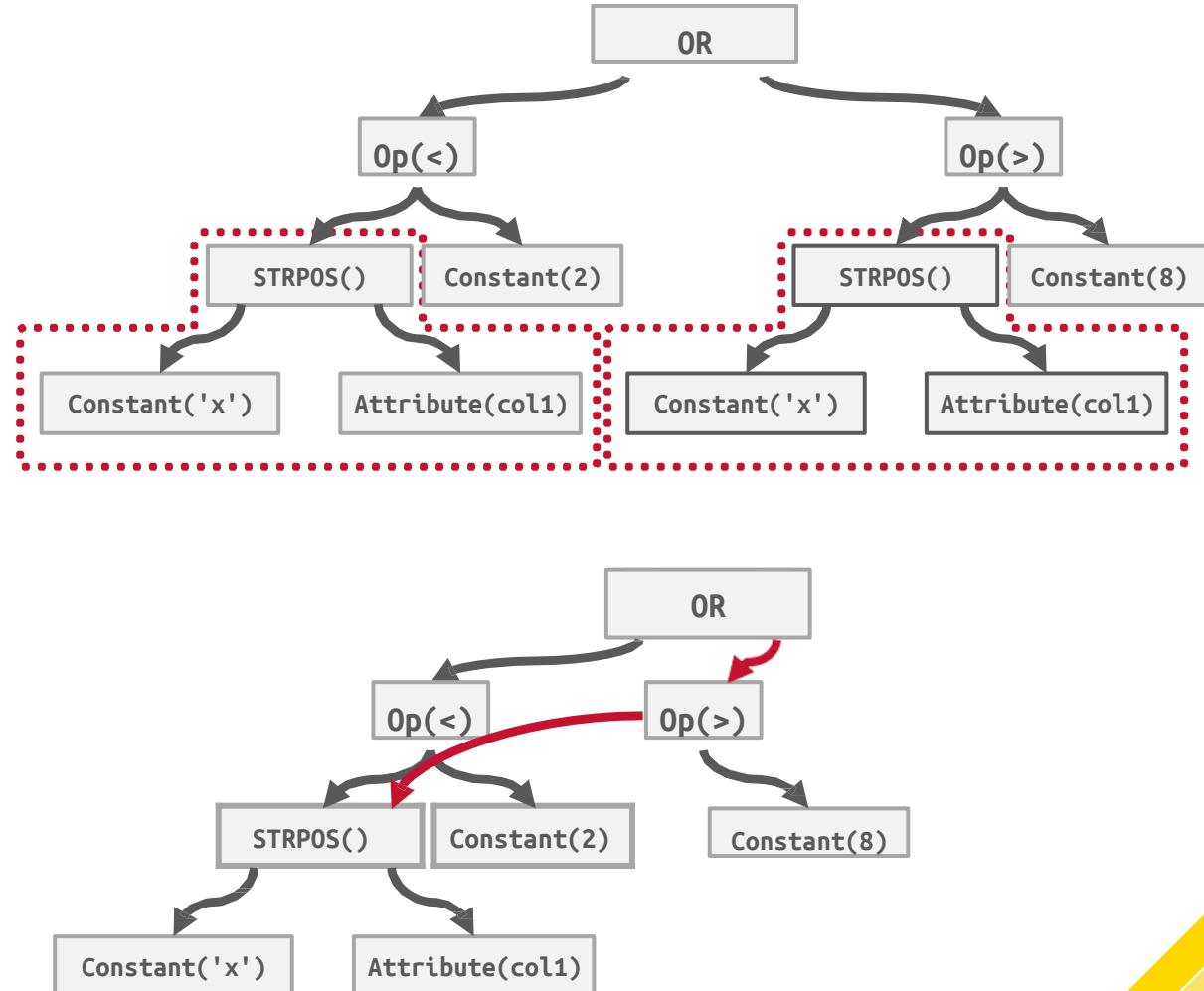
```
WHERE UPPER(col1) = UPPER('wutang');
```



# Common Sub-Expr. Elimination

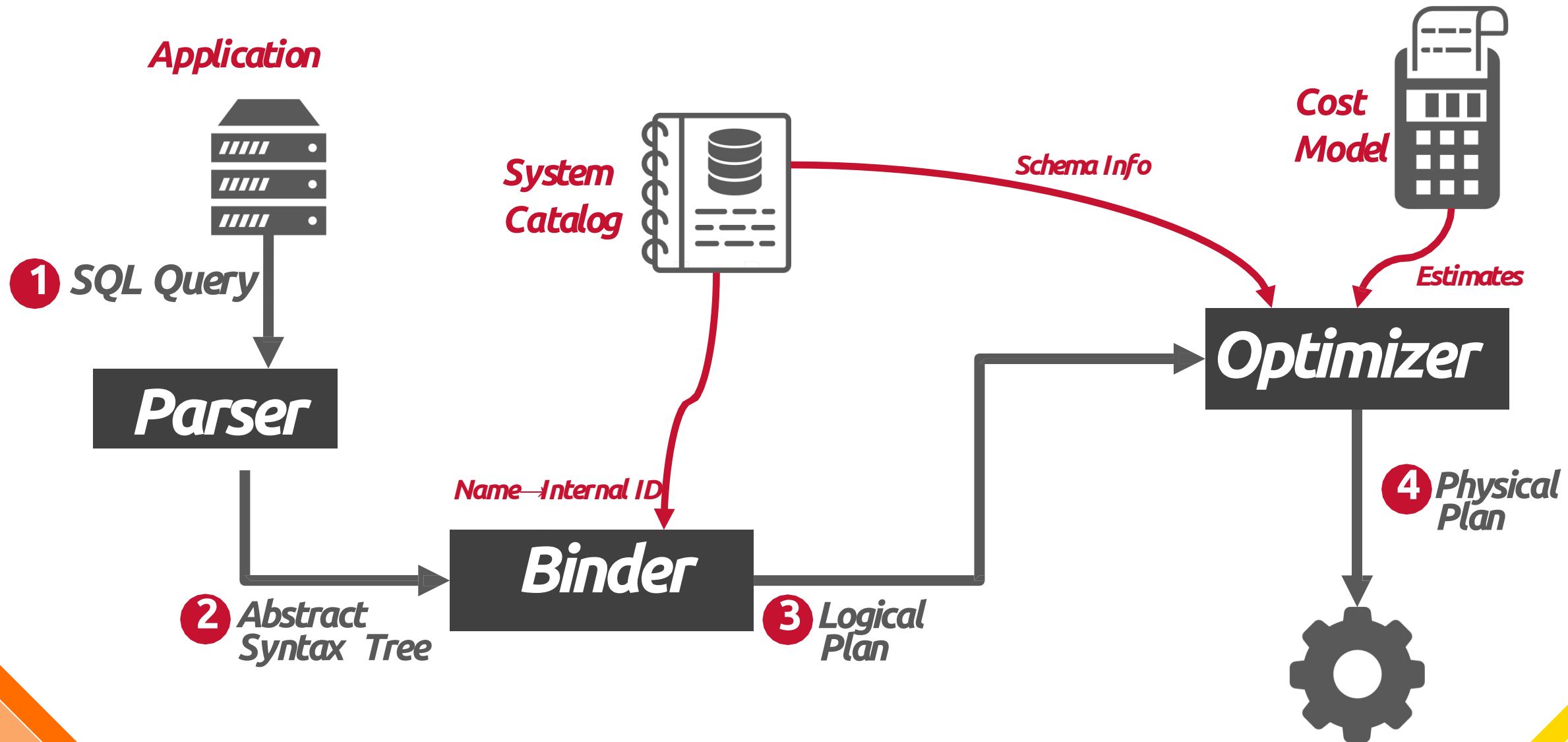
```
WHERE STRPOS('x', col1) < 2  
OR  STRPOS('x', col1) > 8
```

- ◆ Identify repeated sub-expressions that can be shared across expression tree.
- ◆ Compute once and then reuse result.



# Query Optimization

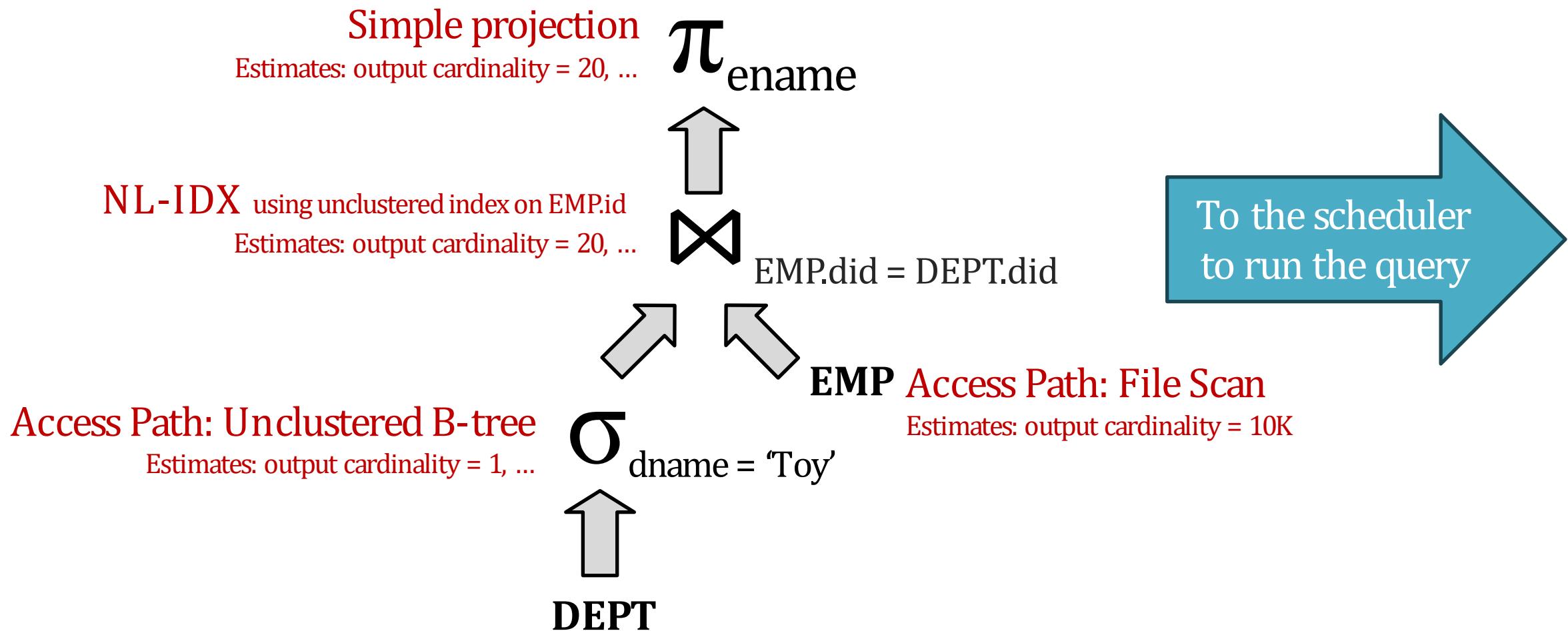
# Optimization architecture



# Logical vs. Physical plans

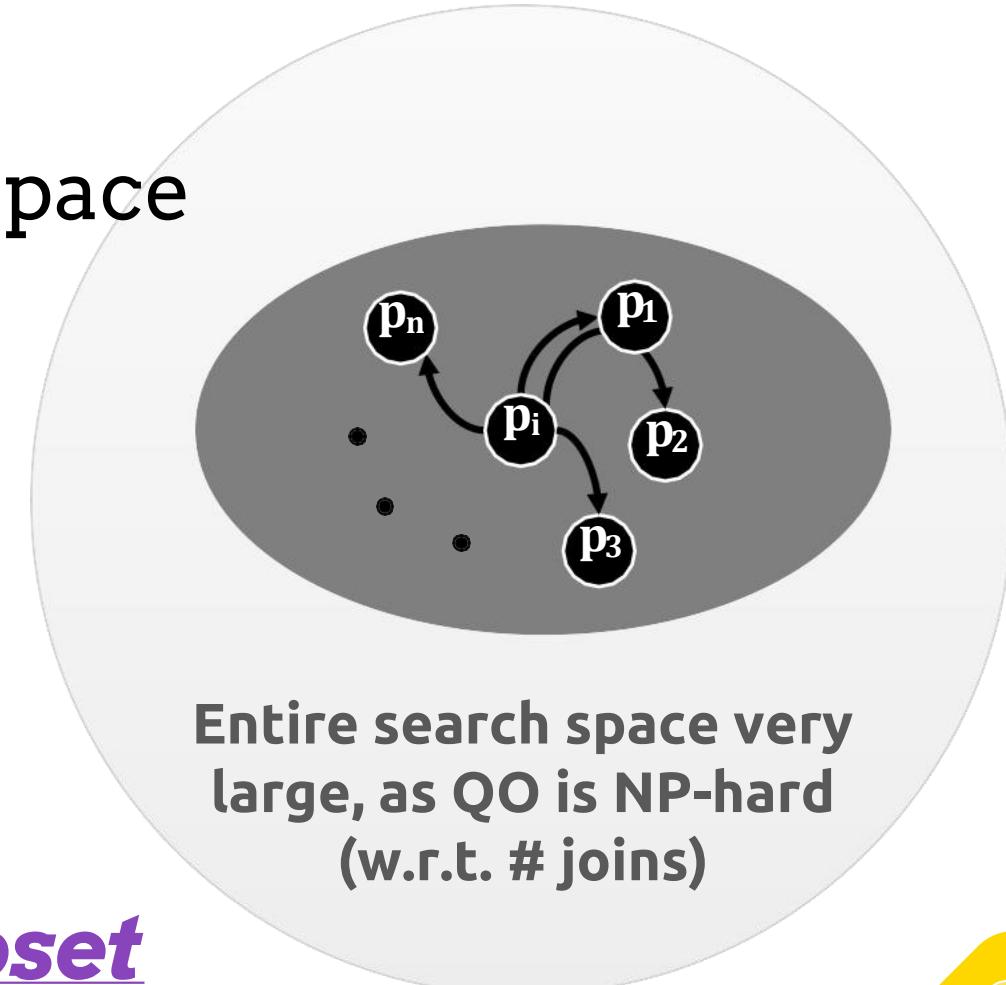
- ◆ The optimizer generates a mapping of a **logical** algebra expression to the optimal equivalent physical algebra expression.
- ◆ **Physical** operators define a specific execution strategy using an access path.
  - ★ They can depend on the physical format of the data that they process (i.e., sorting, compression).
  - ★ Not always a 1:1 mapping from logical to physical.

# Annotated RA Tree = The Physical Plan



# Query optimization (QO)

- ◆ Identify candidate equivalent trees (logical).
  - ★ It is an NP-hard problem, so the space is large.
- ◆ For each candidate, find the execution plan (physical).
  - ★ Estimate the cost of each plan.
- ◆ Choose the best (physical) plan.
- ◆ ***Practically: Choose from a subset of all possible plans.***



# **QUERY OPTIMIZATION**

**Heuristics /  
Rules**

**Cost-based  
Search**

# Heuristics / Rules

- ◆ Rewrite the query to remove (guessed) inefficiencies.
- ◆ Examples:
  - ★ always do selections first or push down projections as early as possible.
- ◆ These techniques may **need** to **examine catalog**, but they **do not need** to **examine data**.

# Logical plan 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.
  - ★ Many equivalence rules for relational algebra!
- ◆ Cannot compare plans because there is no cost model but can "direct" a transformation to a preferred side.

# **Heuristic Algebraic optimization Algorithm**

# Algorithm Outline

- ◆ **Using rule 1,** *break up any select operations* with conjunctive conditions into a cascade of select operations.
- ◆ **Using rules 2, 4, 6, and 10** concerning the commutativity of select with other operations, *move each select operation as far down the query tree* as is permitted by the attributes involved in the select condition.
- ◆ **Using rule 9** concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the *most restrictive select operations are executed first* in the query tree representation.

# Algorithm Outline

- ◆ Using Rule 12, **combine a Cartesian product operation with a subsequent select operation** in the tree into a join operation.
- ◆ Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, **break down and move lists of projection attributes down the tree as far as possible** by creating new project operations as needed.
- ◆ Identify subtrees that represent groups of operations that can be executed by a single algorithm.

# EXAMPLE

## ◆ Heuristic Optimization of Query Trees:

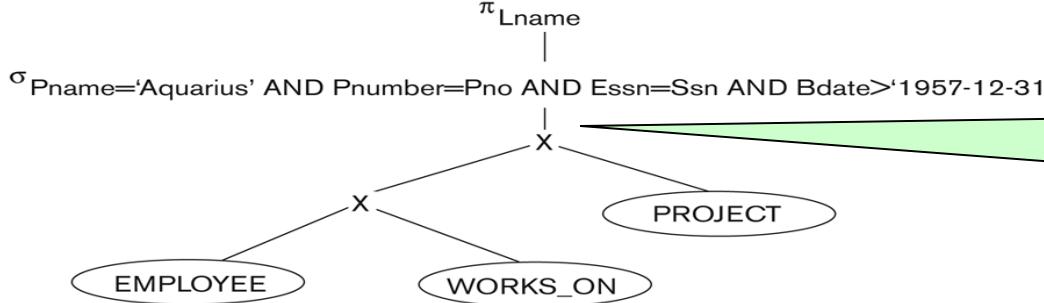
- ★ The same query could correspond to many different relational algebra expressions — and hence many different query trees.
- ★ The task of heuristic optimization of query trees is to find a ***final query tree*** that is **Efficient to Execute**.

## ◆ Example:

★ **SELECT** LNAME  
★ **FROM** EMPLOYEE, WORKS\_ON, PROJECT  
★ **WHERE** PNAME = 'AQUARIUS' **AND** PNMUBER=PNO  
    **AND** ESSN=SSN **AND** BDATE > '1957-12-31';

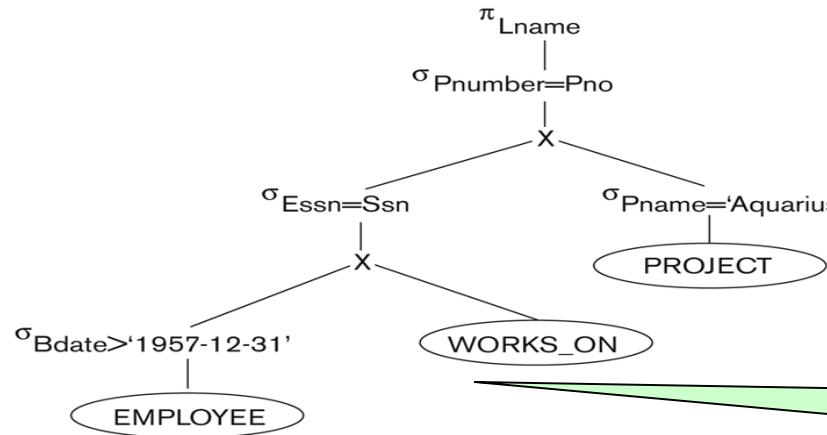
# Using Heuristics in Query Optimization

(a)



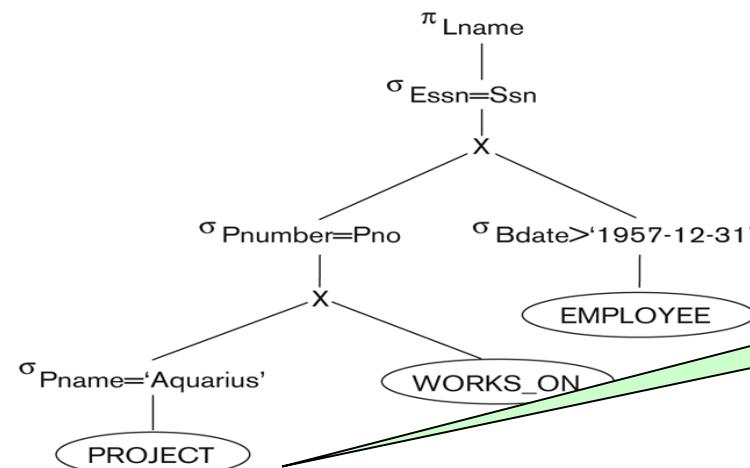
Cartesian Product  
→ **Avoid it!!!**

(b)



Apply Select to reduce number of tuples

(c)



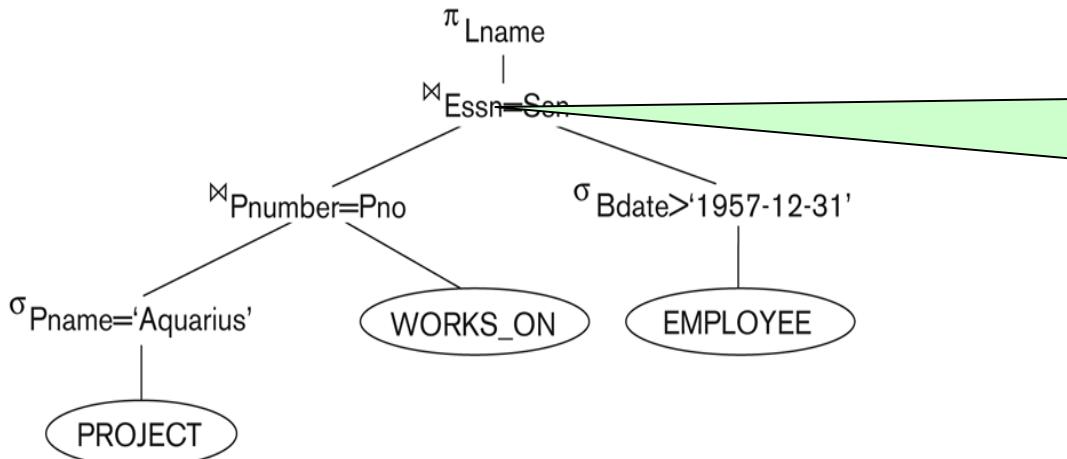
Move Project down since it is more selective

Figure

Heuristic rules in converting a query tree during heuristic optimization.

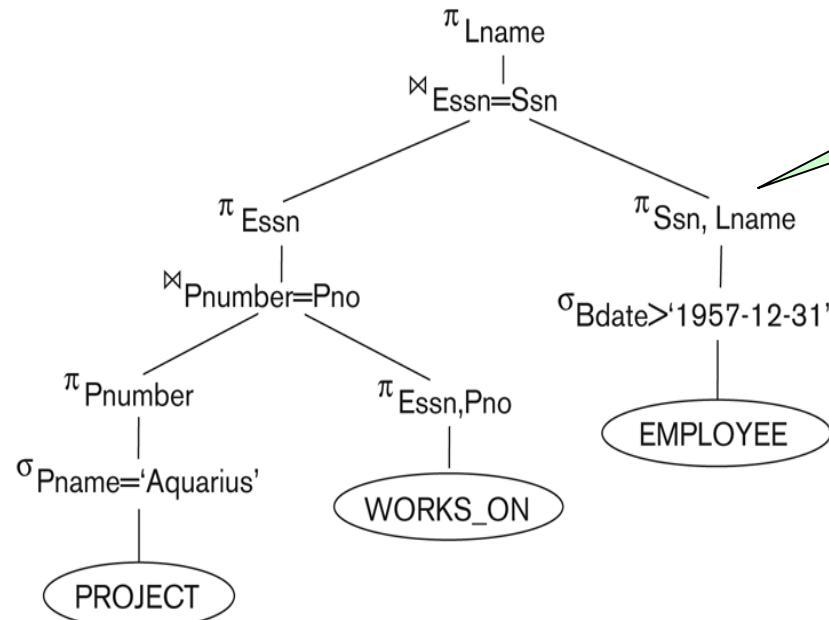
- Initial (canonical) query tree for SQL query Q.
- Moving SELECT operations down the query tree.
- Applying the more restrictive SELECT operation first.
- Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
- Moving PROJECT operations down the query tree.

(d)



Replace Cart. Product  
followed by Join Condition  
 $\rightarrow \bowtie$  JOIN

(e)



Move Project  
operations down the  
query tree

**Figure 15.5**

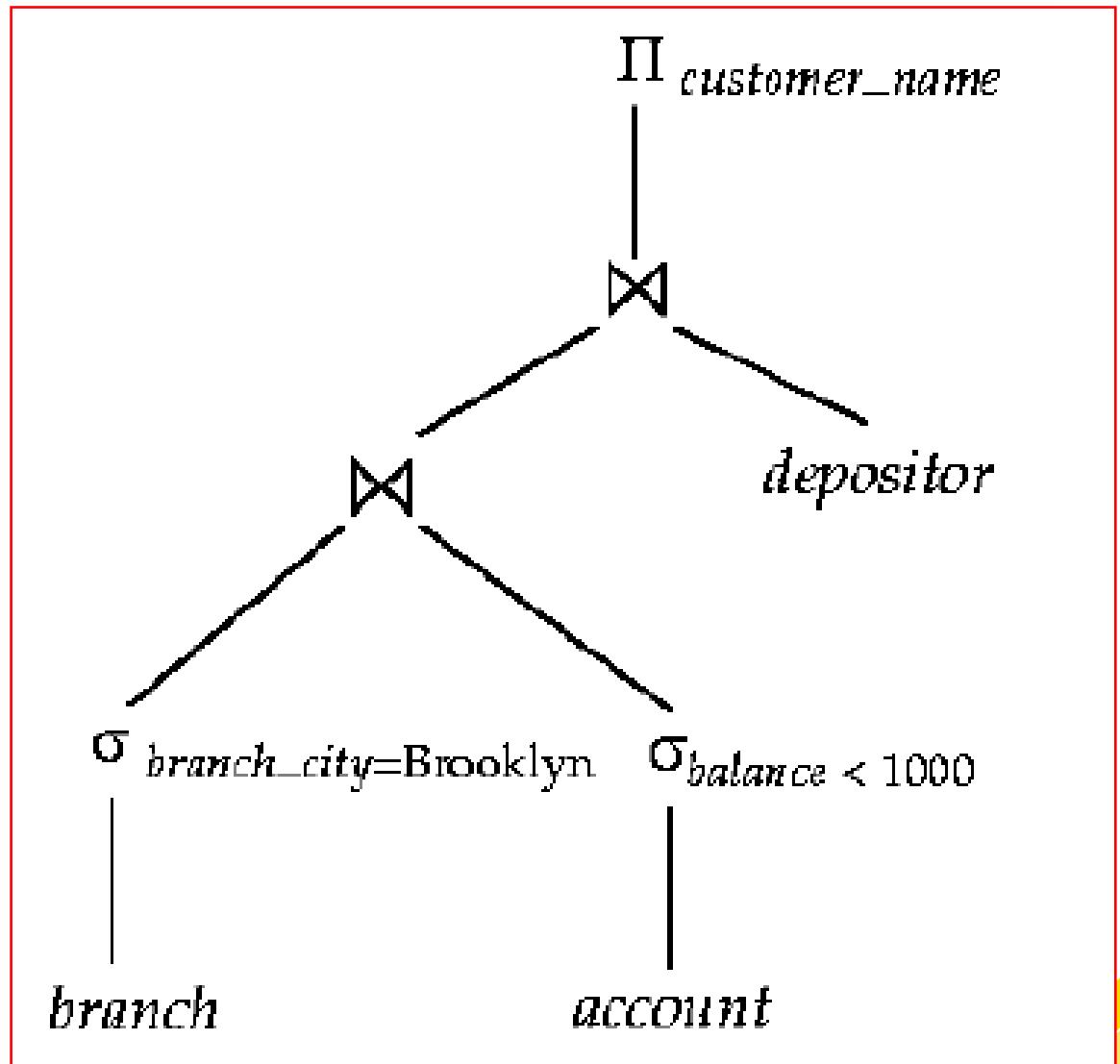
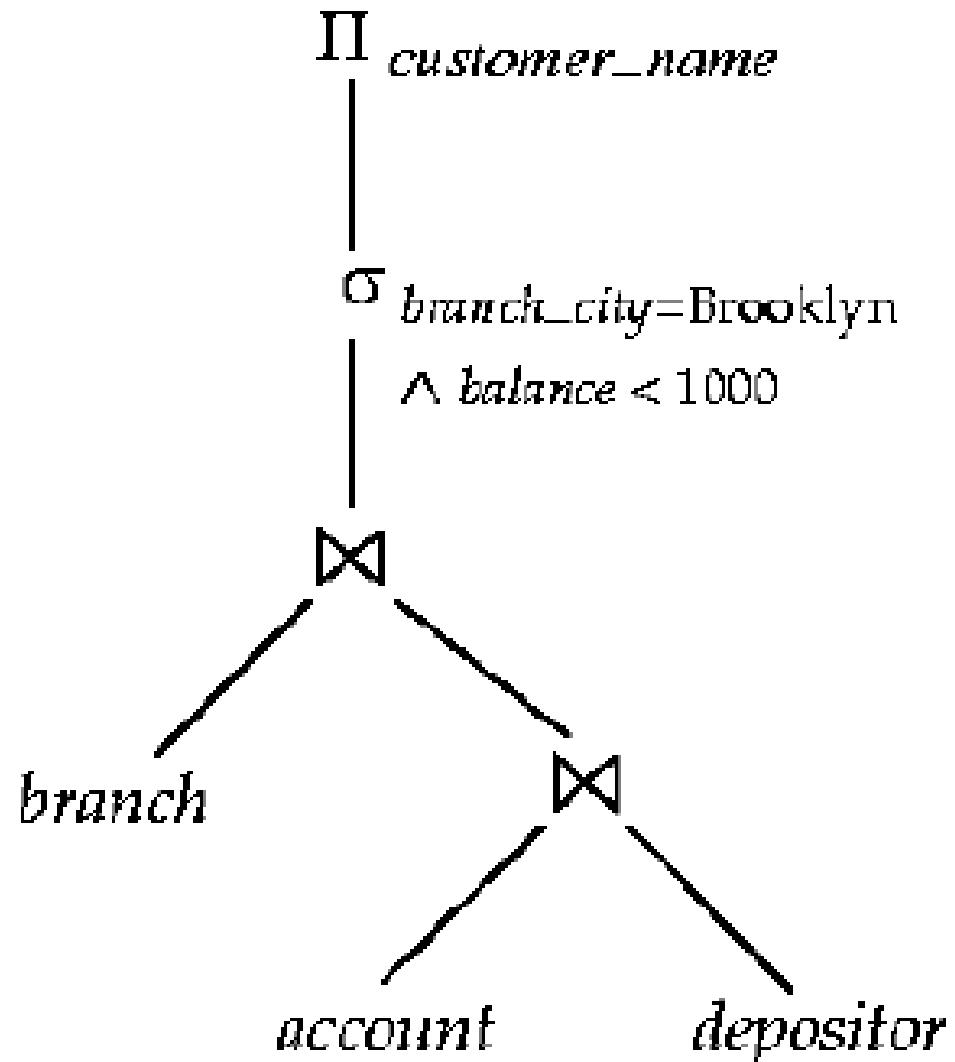
Steps in converting a query tree during heuristic optimization.

- Initial (canonical) query tree for SQL query Q.
- Moving SELECT operations down the query tree.
- Applying the more restrictive SELECT operation first.
- Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
- Moving PROJECT operations down the query tree.

# Summary of Heuristics

- ◆ The main heuristic is to apply first the operations that reduce the size of intermediate results.
  - ★ Perform select operations as early as possible to reduce the number of tuples
  - ★ Perform project operations as early as possible to reduce the number of attributes.
  - ★ Move select & project operations down the tree
- ◆ The select and join operations that are most restrictive should be executed before other similar operations.
  - ★ Reorder the leaf nodes of the tree to have most restrictive operations far down

# Example



# Query optimization example

- Sailors (sid, sname, rating, age)
- Boats(bid, bname, color)
- Reserves(sid, bid, day, rname)
- Query:

```
SELECT S.sid, S.sname, S.age  
FROM      Sailors S, Boats B, Reserves R  
WHERE      B.bid = R.rid AND B.bid = R.bid AND  
          B.color = "Red" AND S.age < 30;
```

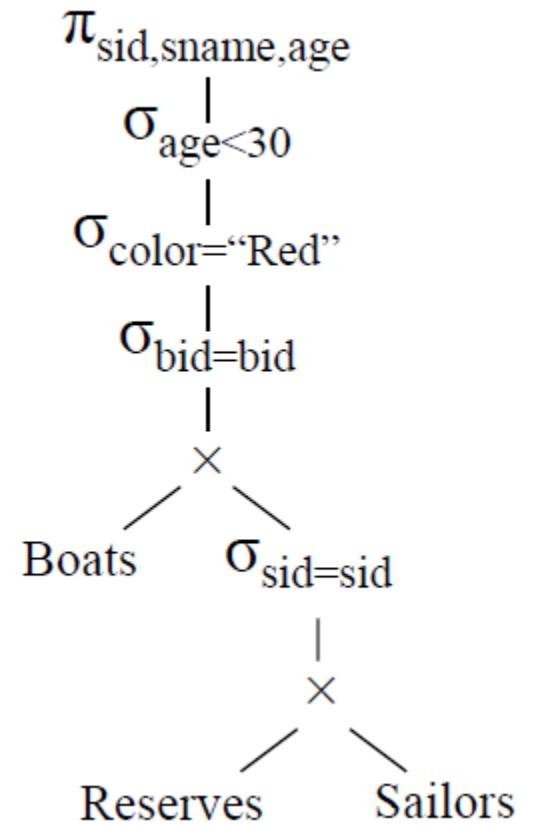
- ◆ **SELECT S.sid, S.name, S.age**
  - ◆ **FROM Sailors S, Boats B, Reservers R**
  - ◆ **WHERE S.sid=R.sid AND B.bid=R.bid AND B.color=“Red” AND S.age>30**
- 
- **Sailors (sid, sname, rating, age)**
  - **Boats(bid, bname, color)**
  - **Reserves(sid, bid, day, rname)**

# Query optimization example

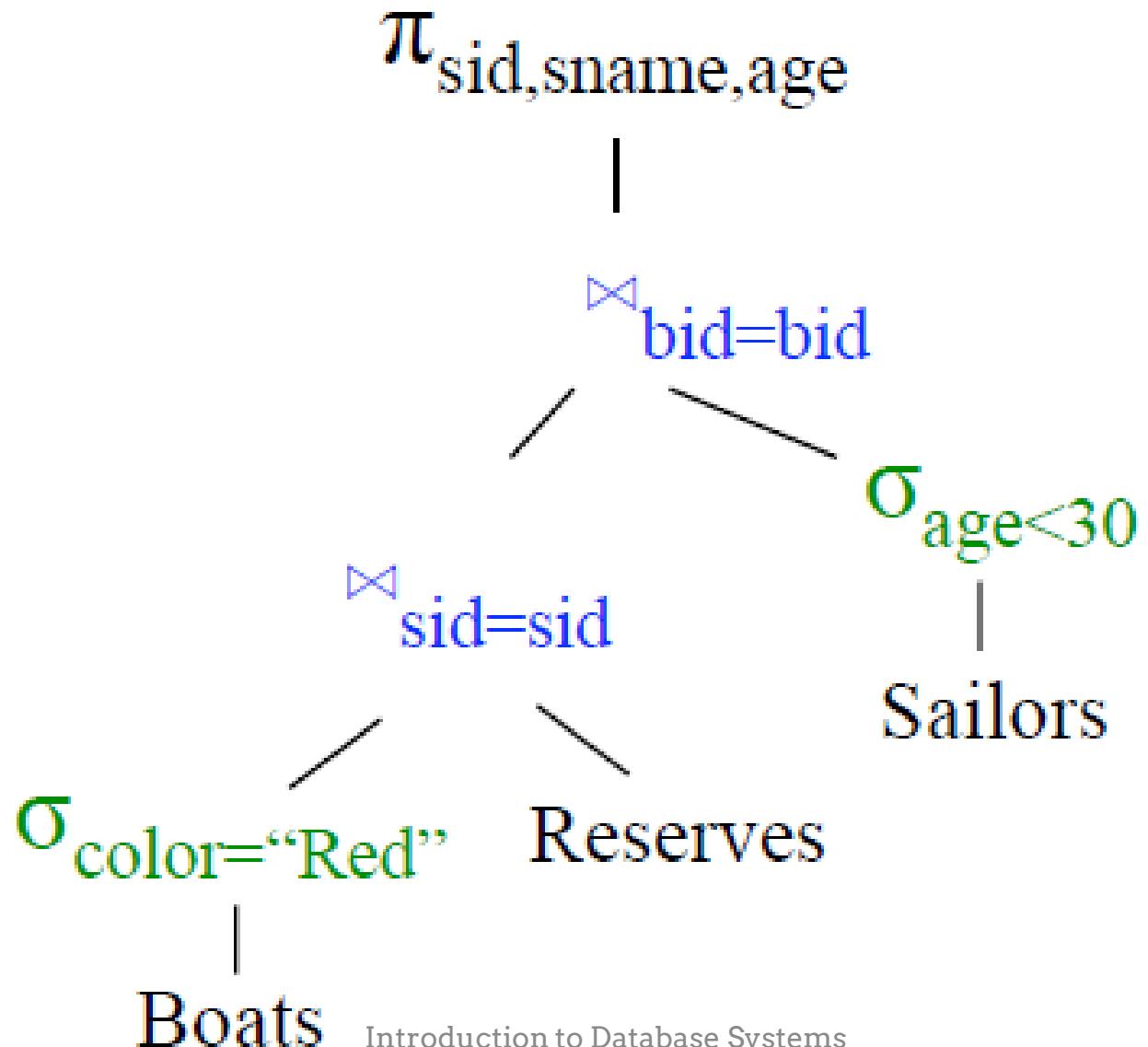
- Step 1: translate SQL query to algebra query

$$\pi_{\text{sid, sname, age}}(\sigma_{\text{age} < 30}(\sigma_{\text{color} = \text{"Red"}}(\sigma_{\text{bid} = \text{bid}}(\text{B} \times \sigma_{\text{sid} = \text{sid}}(\text{S} \times \text{R})))))$$

- Step 2: generate initial query tree
- Step 3: apply heuristic rules and Generate optimized tree



# Query optimization example





THANK  
YOU ☺

