



香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen



Ack: Prof. Jignesh Patel @ CMU
Prof. Andy Pavlo @CMU

CSC3170

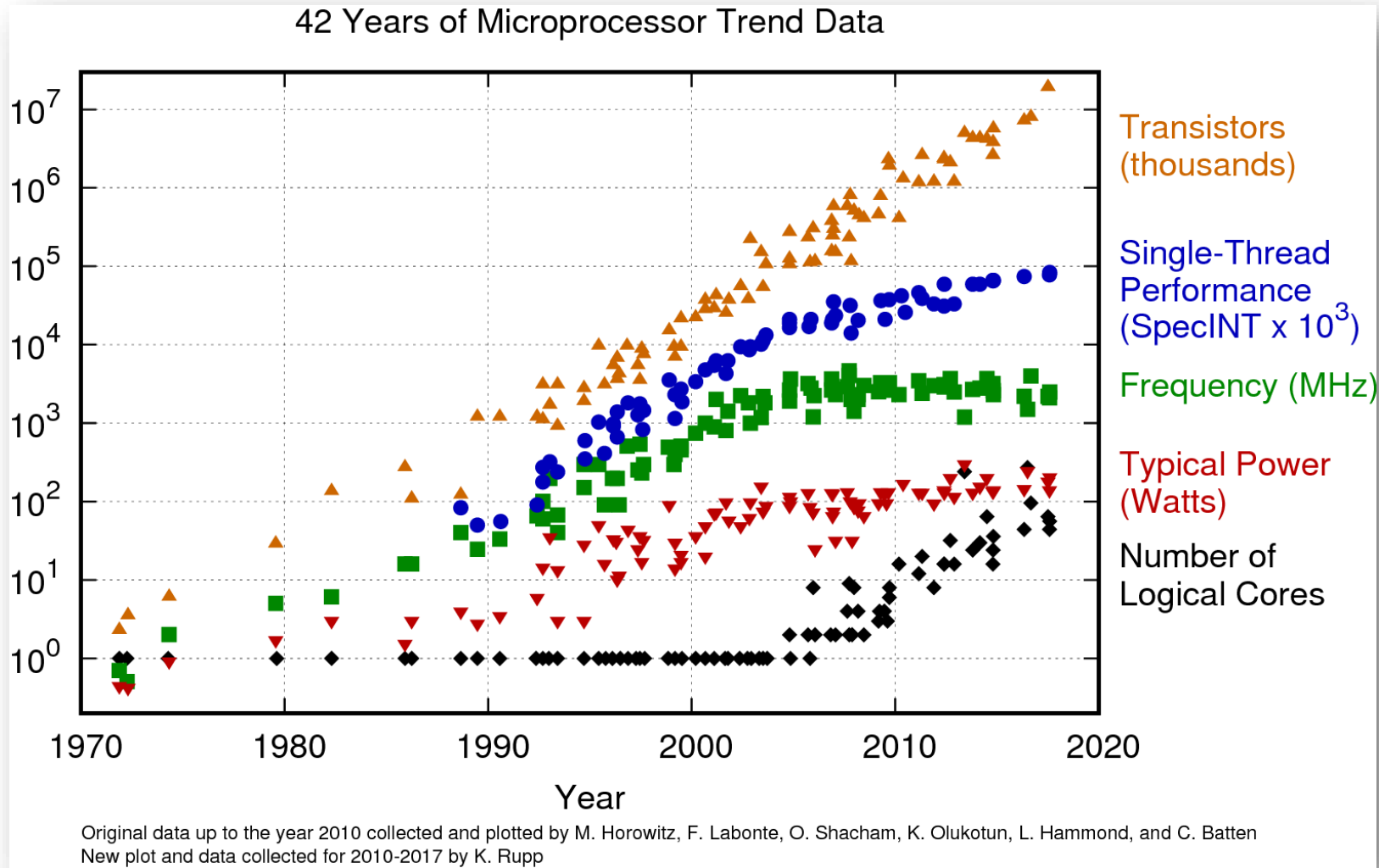
12: Query Execution *part a*

Chenhao Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen

This Lecture

- Processing Models
- Access Methods
- Modification Queries
- Expression Evaluation

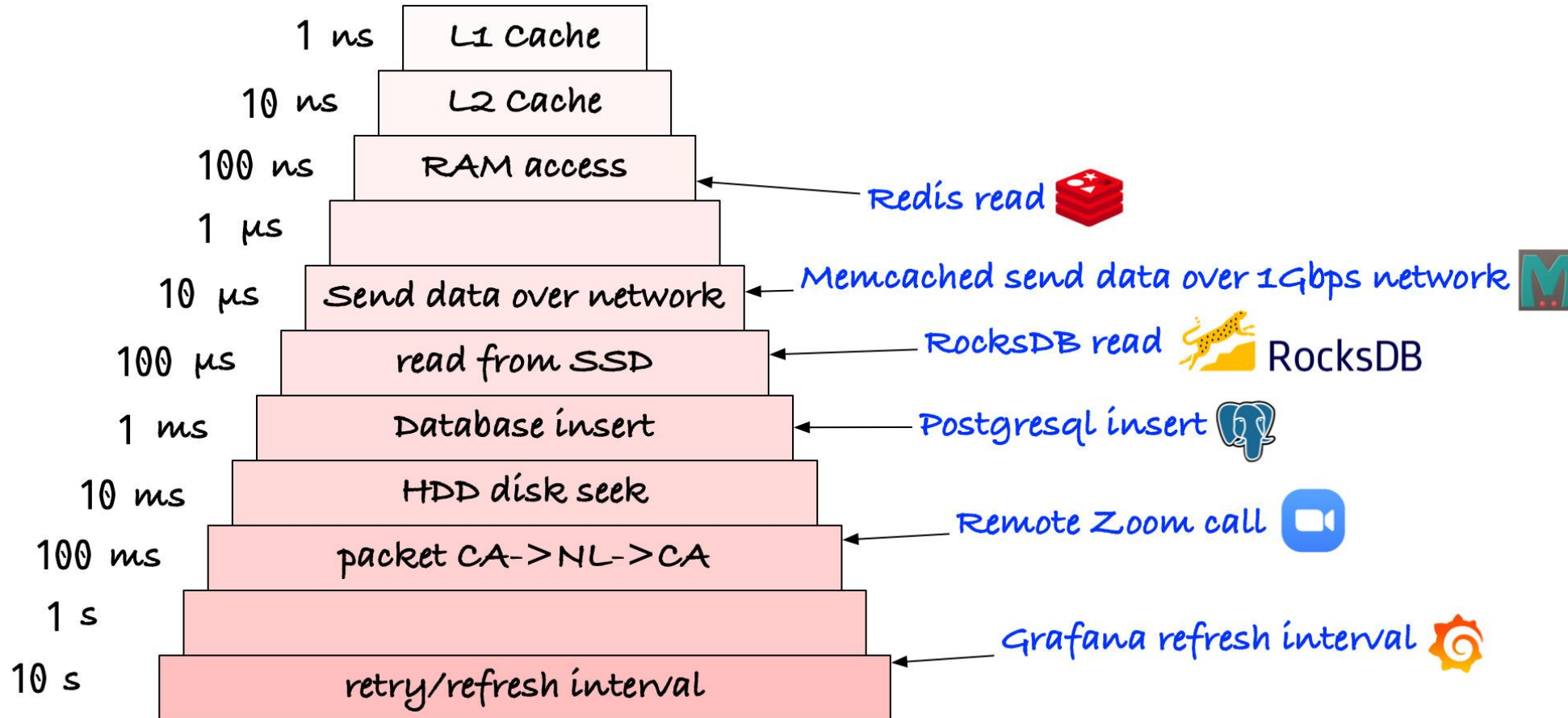
Hardware trends



- Transistor growth continues.
- The question is how to use this hardware for higher application performance.
- Individual cores are not becoming faster, but there are more cores.
- Every processor is now a “parallel” data machine, and the degree of parallelism is increasing.

<https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>

Latency Numbers You Should Know



Every programmer must know these numbers.



Jeff Dean

<https://blog.bytebytego.com/p/ep22-latency-numbers-you-should-know>

Processing Model

Processing Model

- A DBMS's **processing model** defines how the system executes a query plan.
 - Different trade-offs for different workloads.
- **Approach #1: Iterator Model**
- **Approach #2: Materialization Model**
- **Approach #3: Vectorized / Batch Model**

Iterator Model

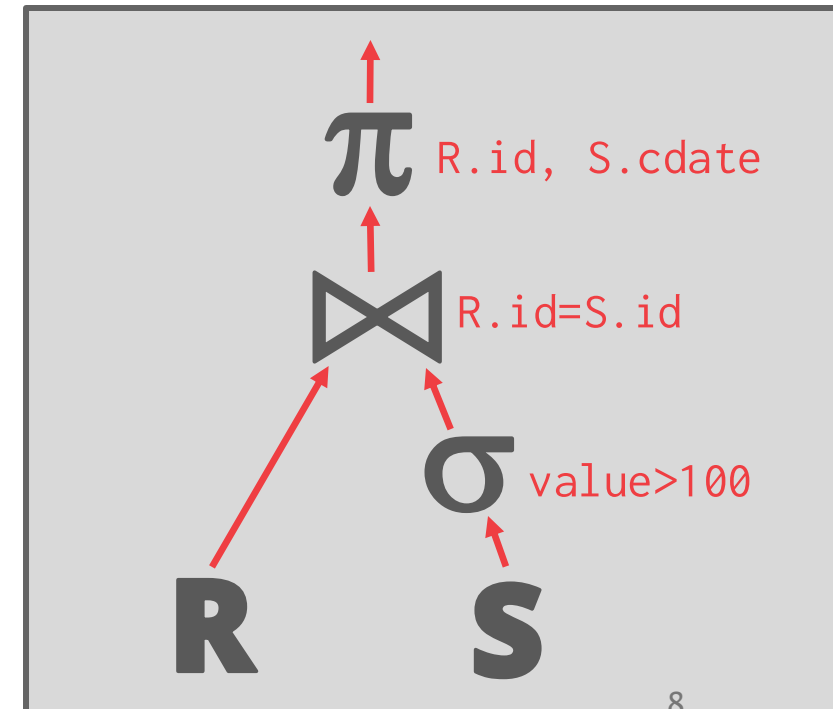
- Each query plan **operator** implements a **Next()** function.
 - On each invocation, the operator returns either a single tuple or a eof marker if there are no more tuples.
 - The operator implements a loop that calls **Next()** on its children to retrieve their tuples and then process them.

Each operator implementation also has **Open()** and **Close()** functions. Analogous to constructors and destructors, but for operators.

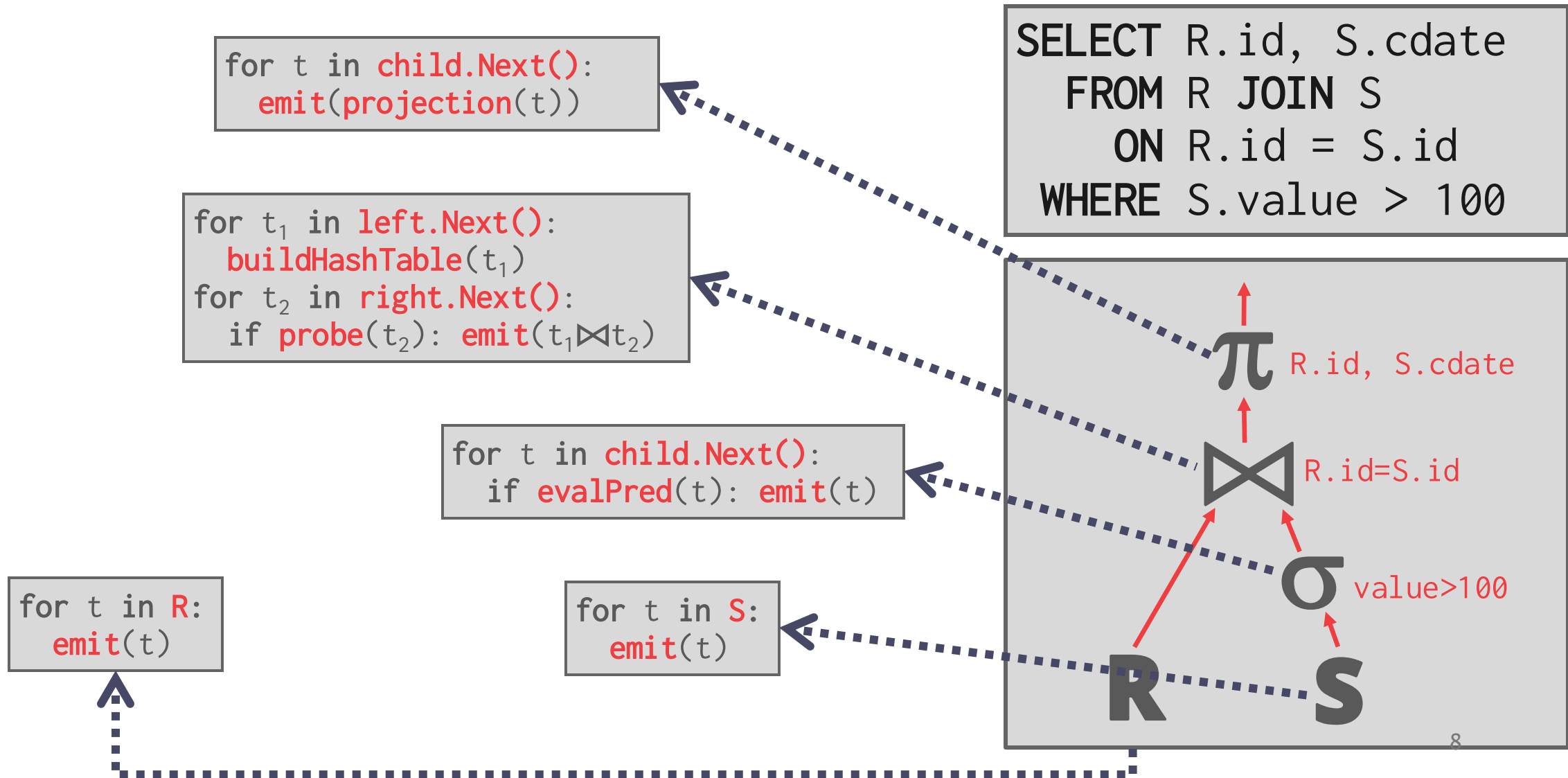
- Also called the **Volcano** or the **Pipeline** Model.

Iterator Model

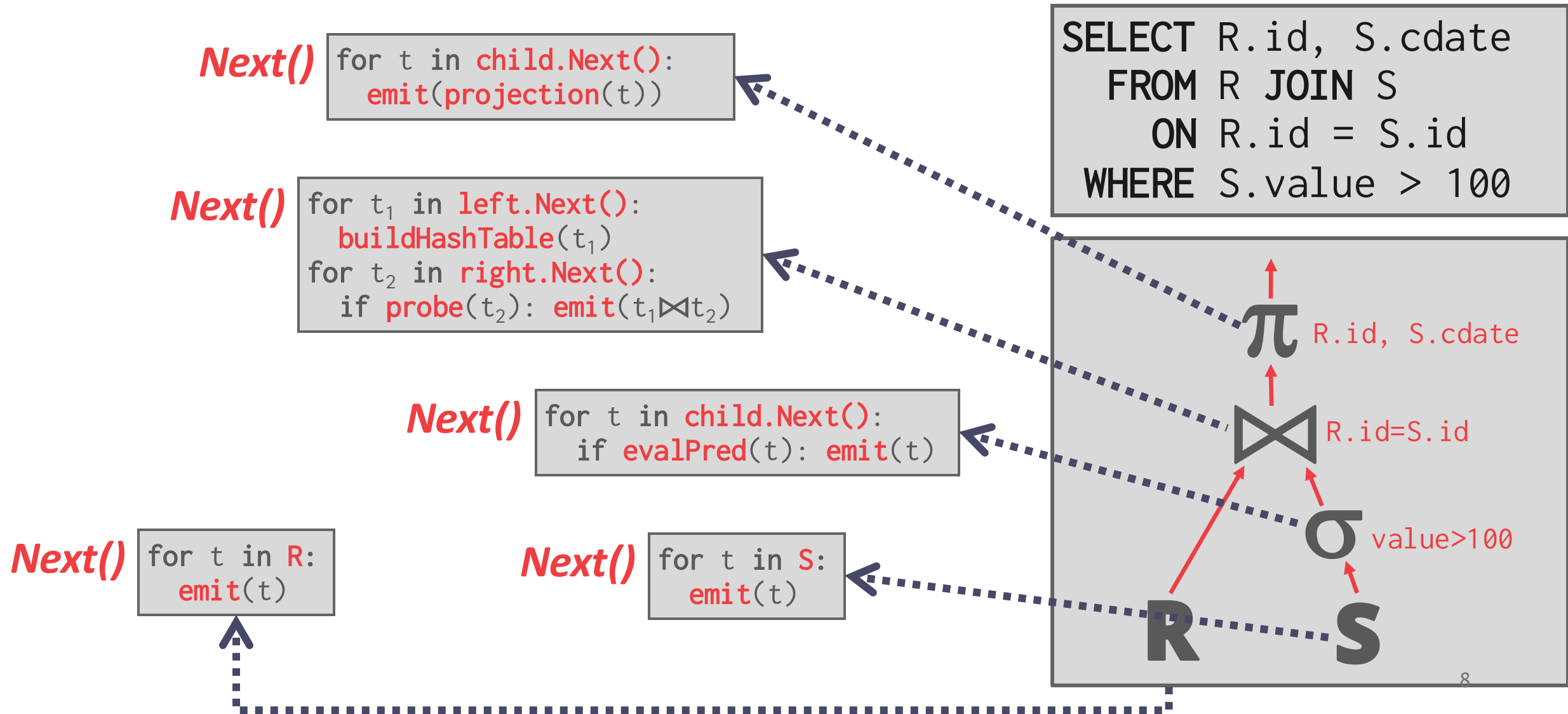
```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



Iterator Model



Iterator Model



Iterator Model

```
for t in child.Next():
    emit(projection(t))
```

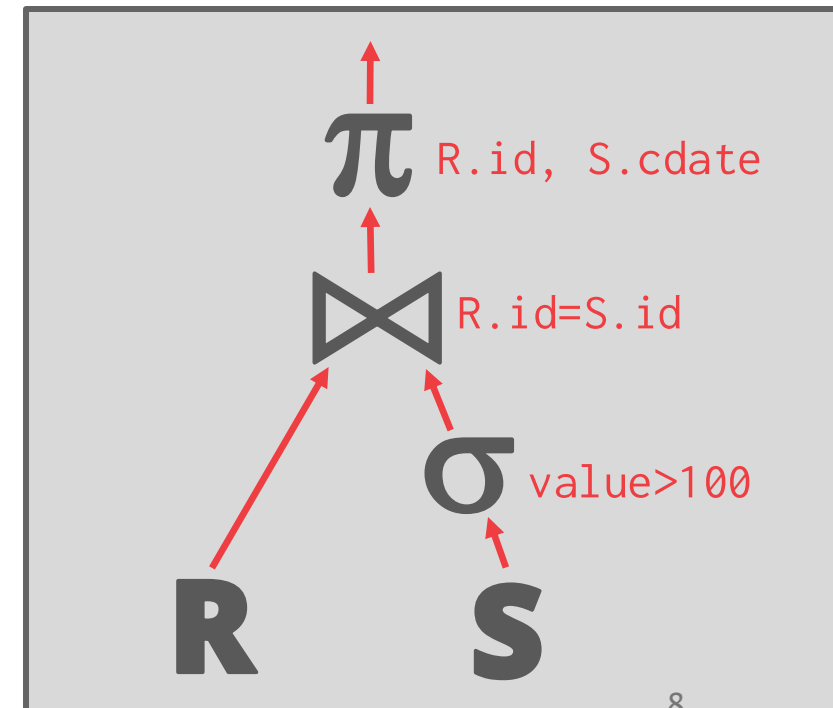
```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1 ⋈ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

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



Iterator Model

1

```
for t in child.Next():
    emit(projection(t))
```

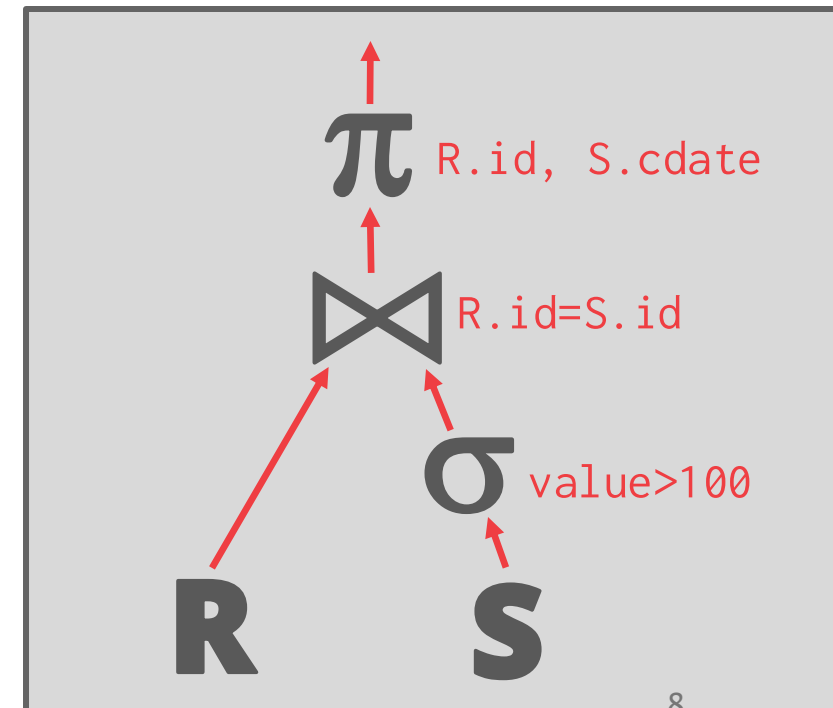
```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1 ⋈ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

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



Iterator Model

1

```
for t in child.Next():
    emit(projection(t))
```

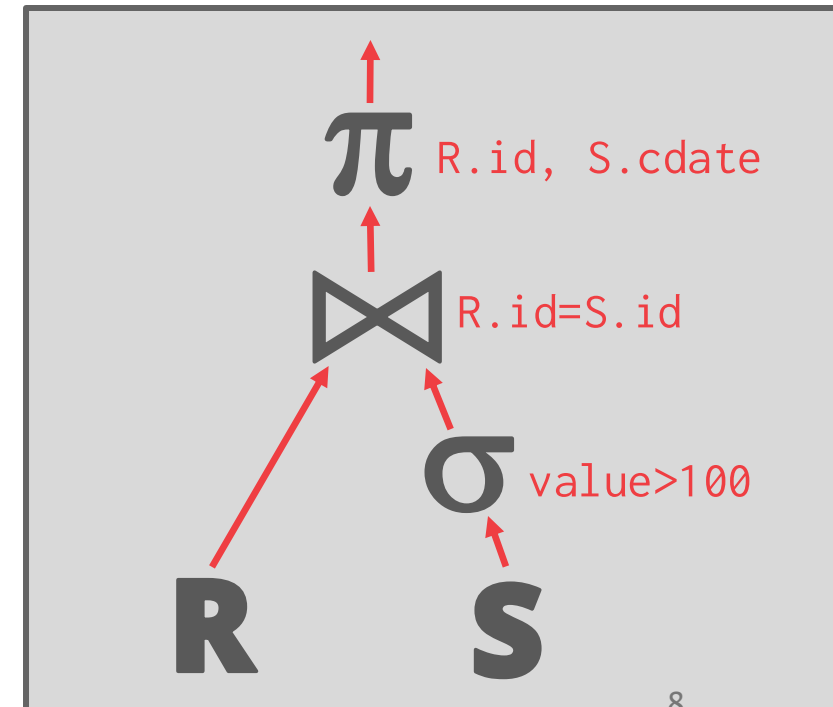
```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1 ⋈ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

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



Iterator Model

1

```
for t in child.Next():
    emit(projection(t))
```

2

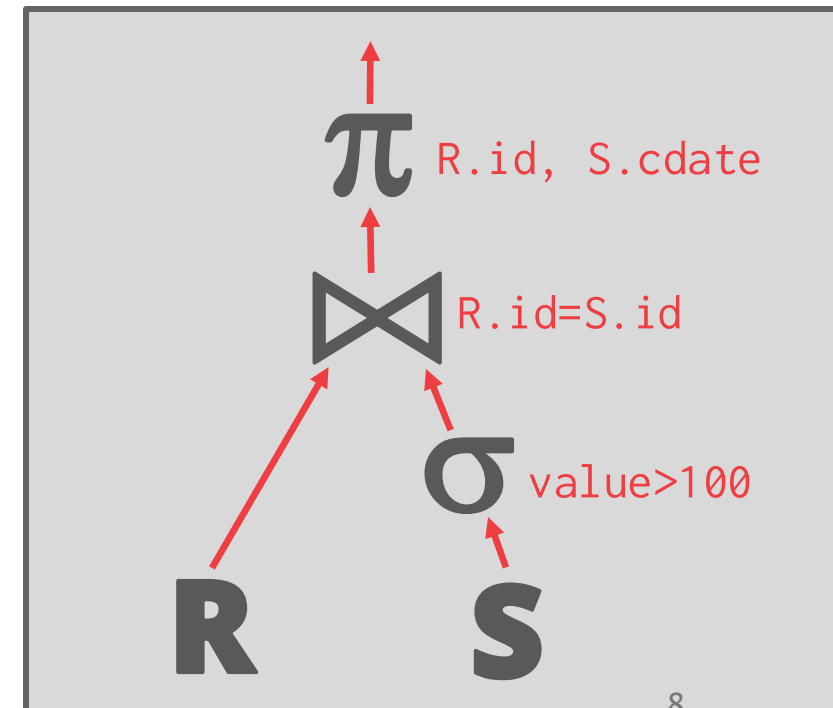
```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1 ⋈ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

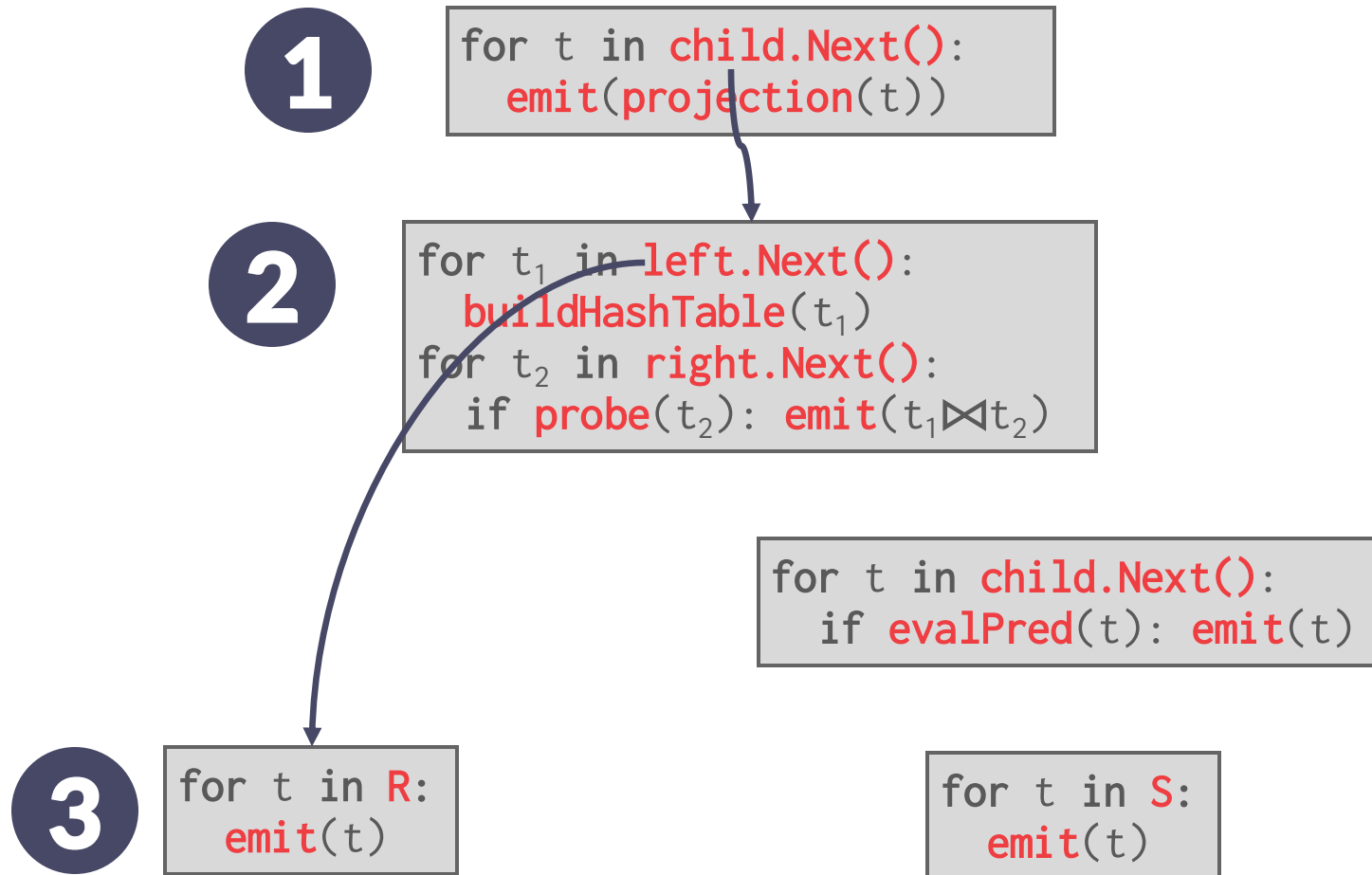
```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

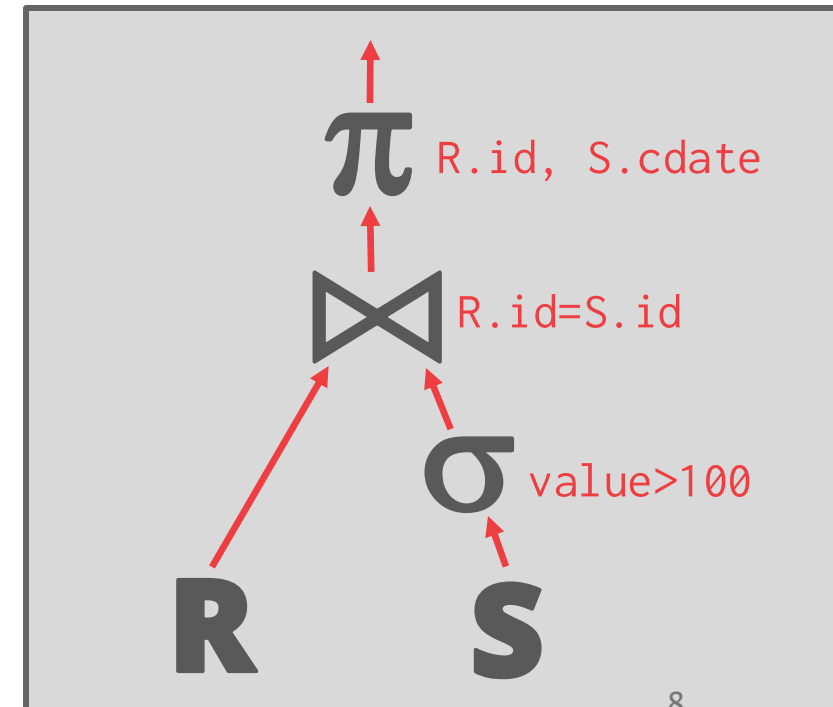
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



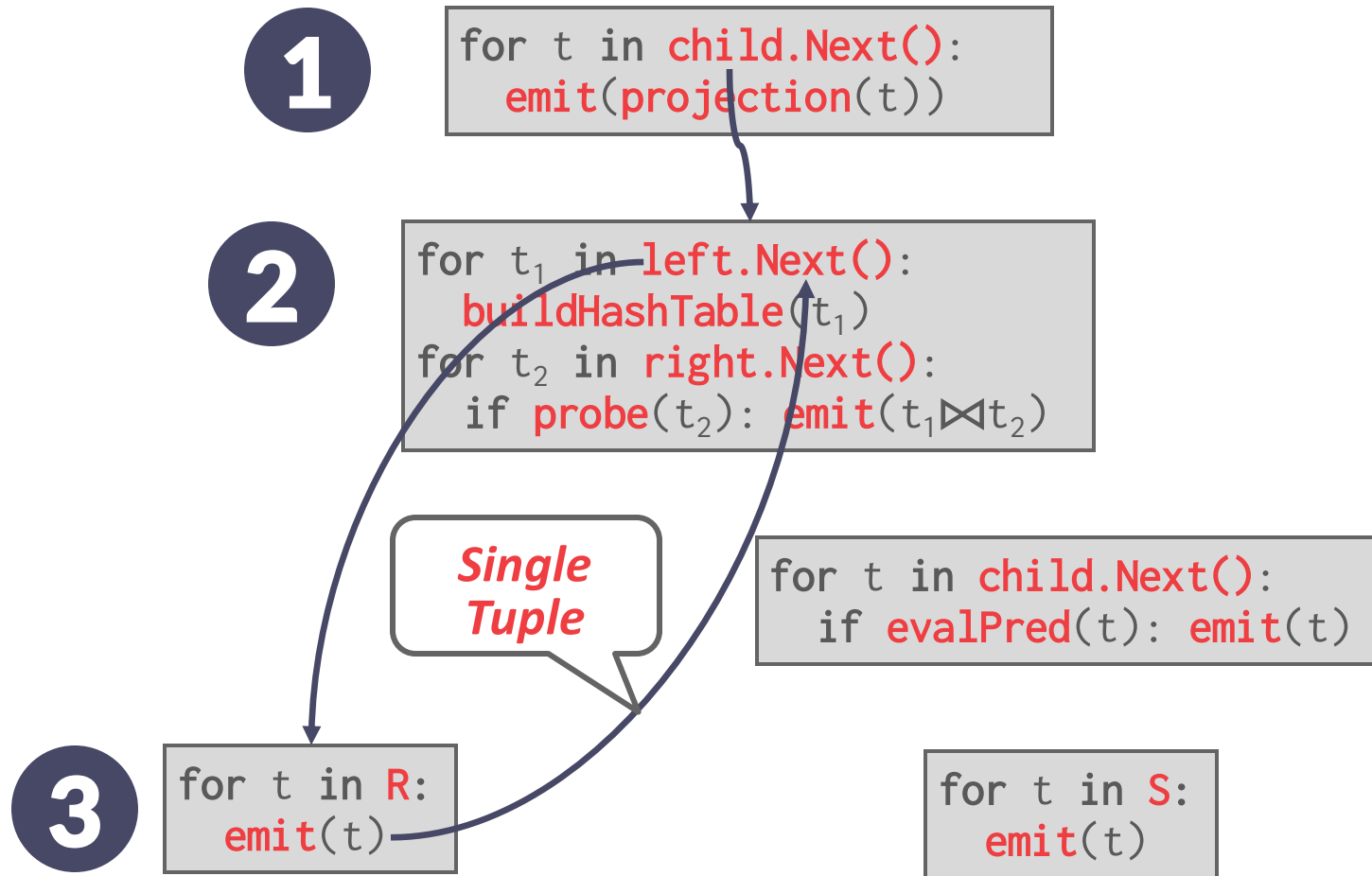
Iterator Model



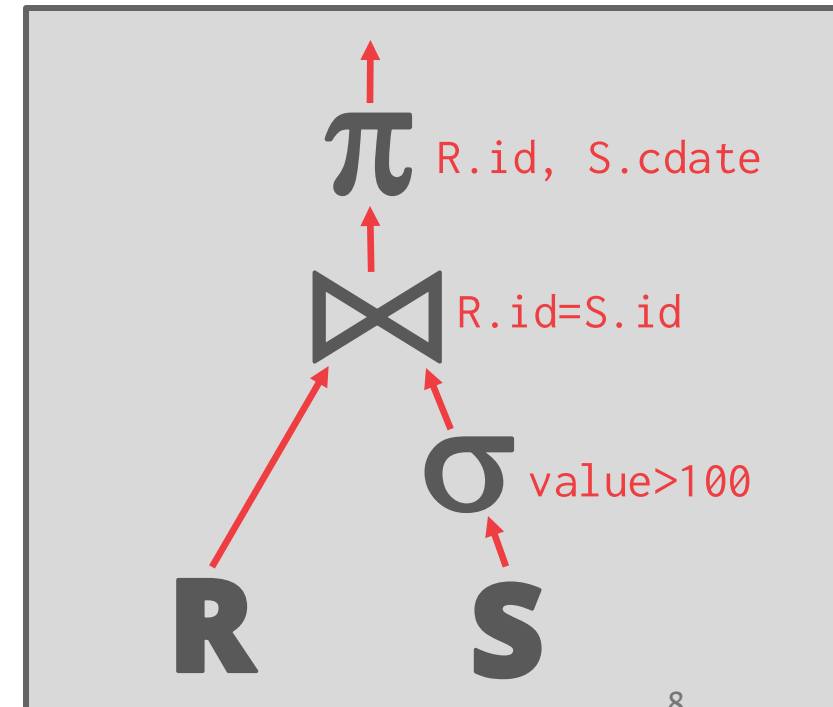
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Iterator Model



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



Iterator Model

1

```
for t in child.Next():
    emit(projection(t))
```

2

```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1 ⋈ t2)
```

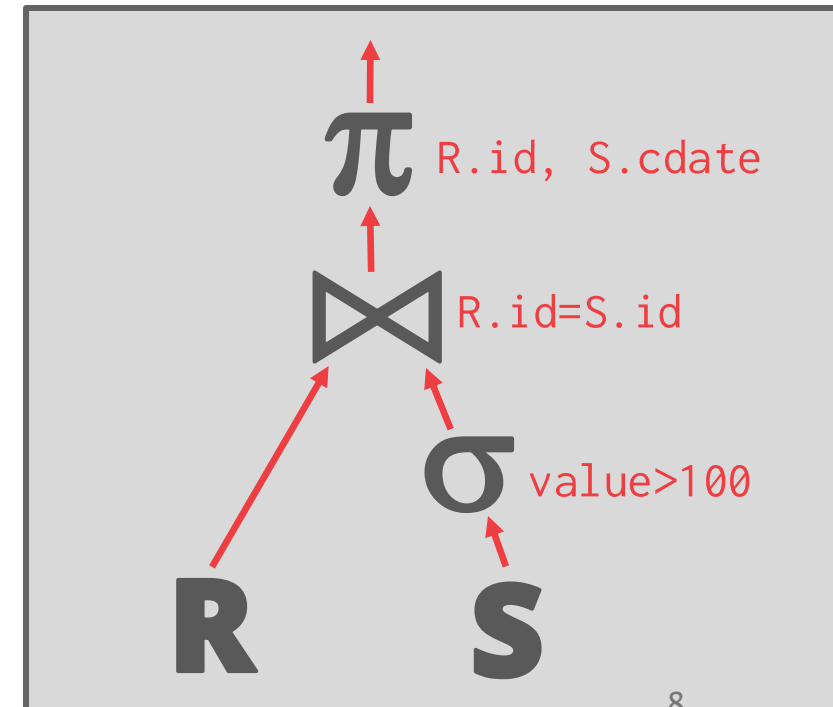
```
for t in child.Next():
    if evalPred(t): emit(t)
```

3

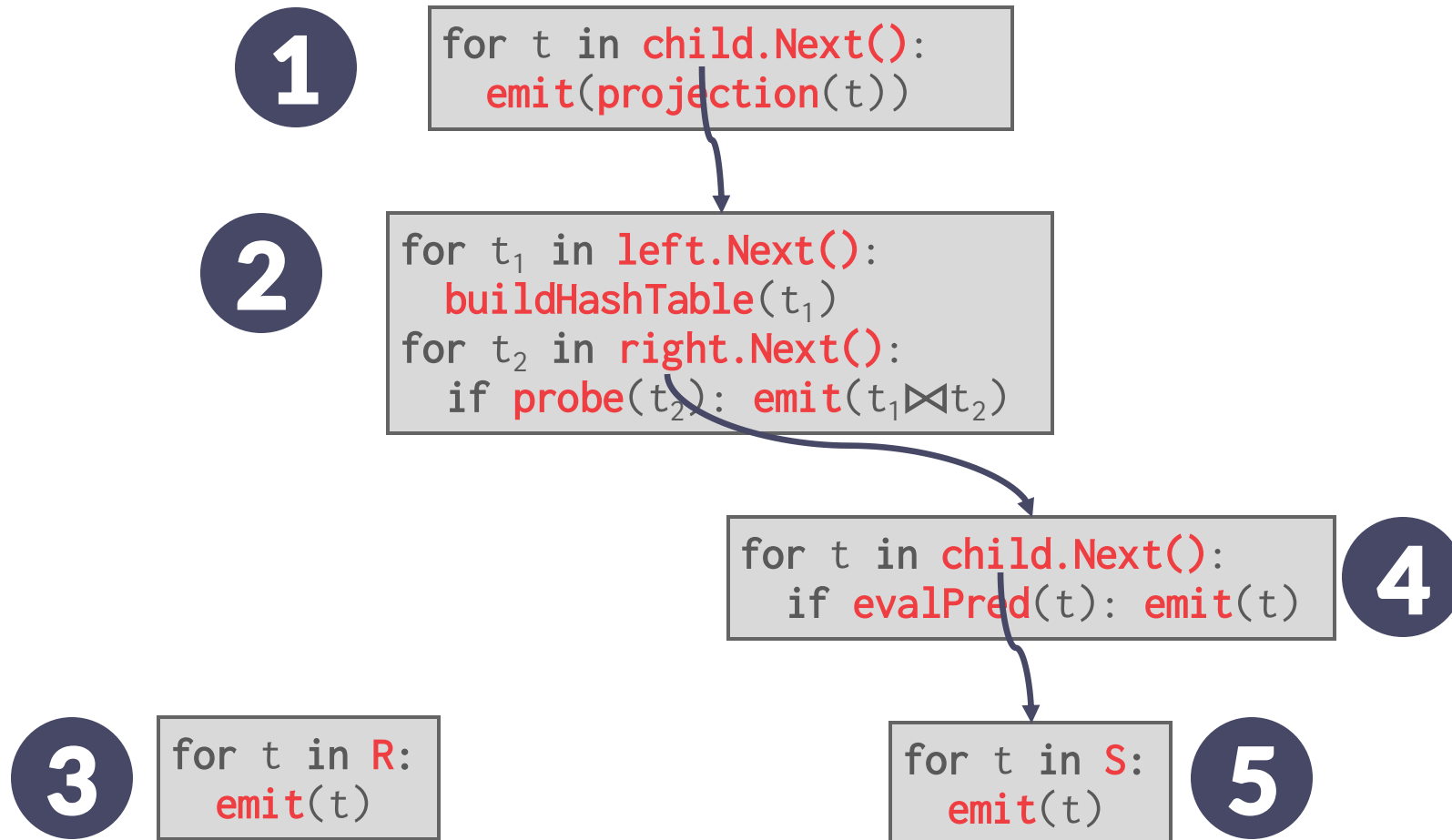
```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

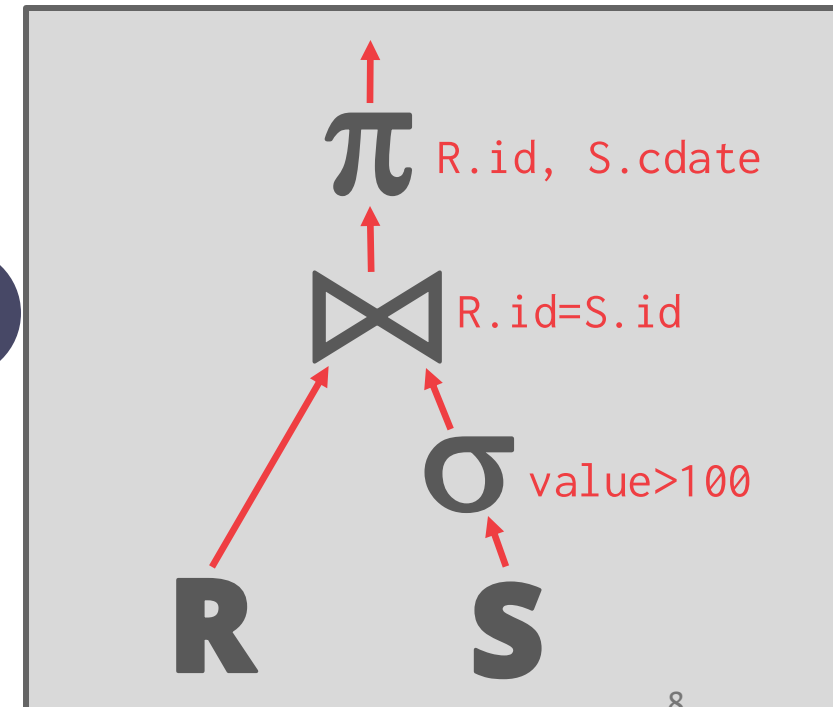
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



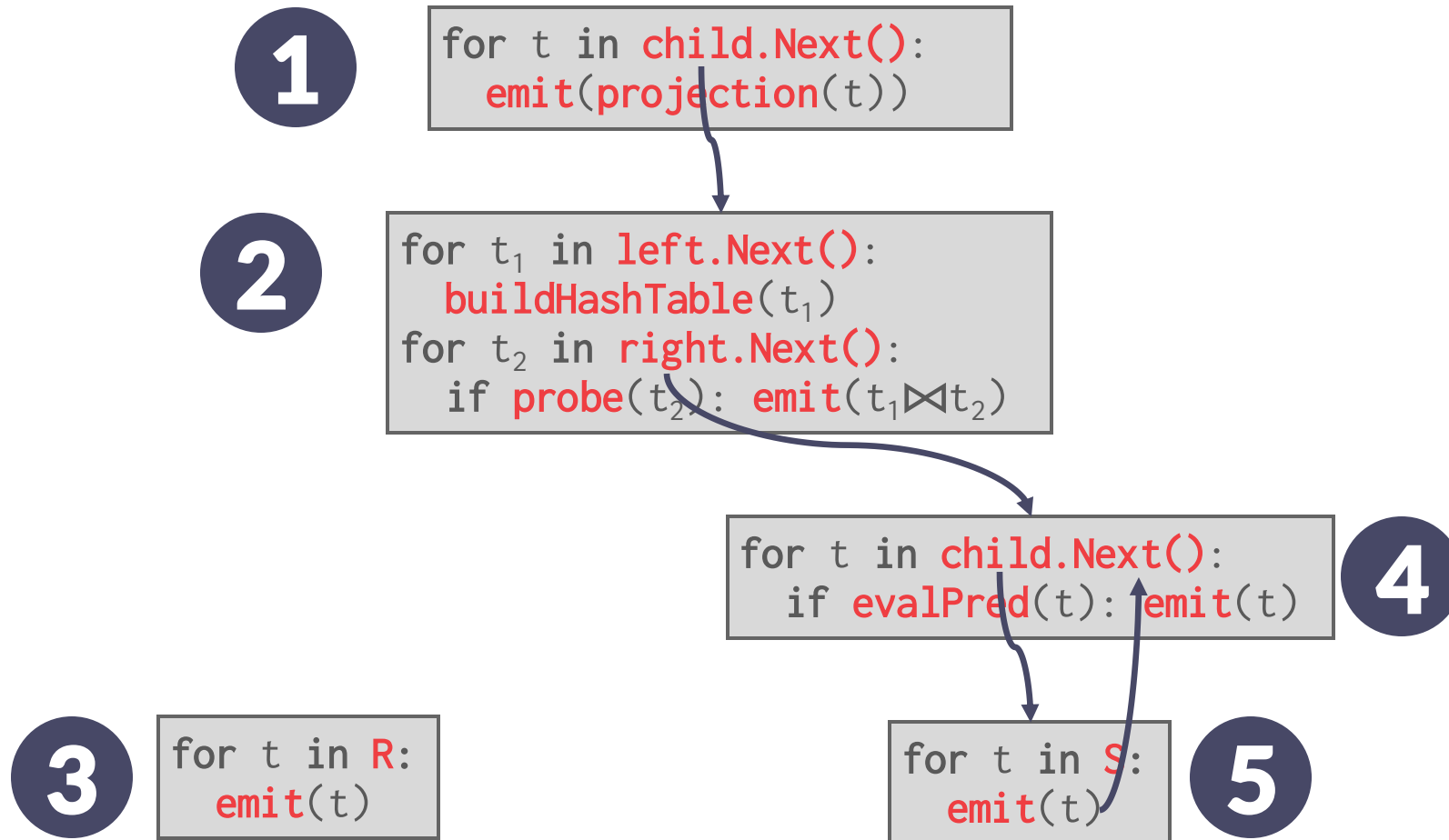
Iterator Model



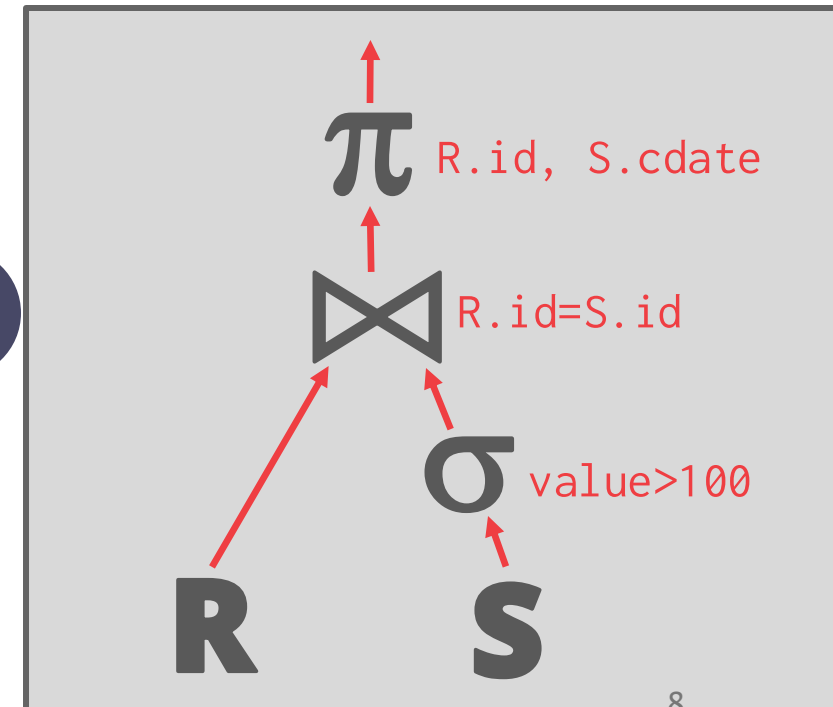
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



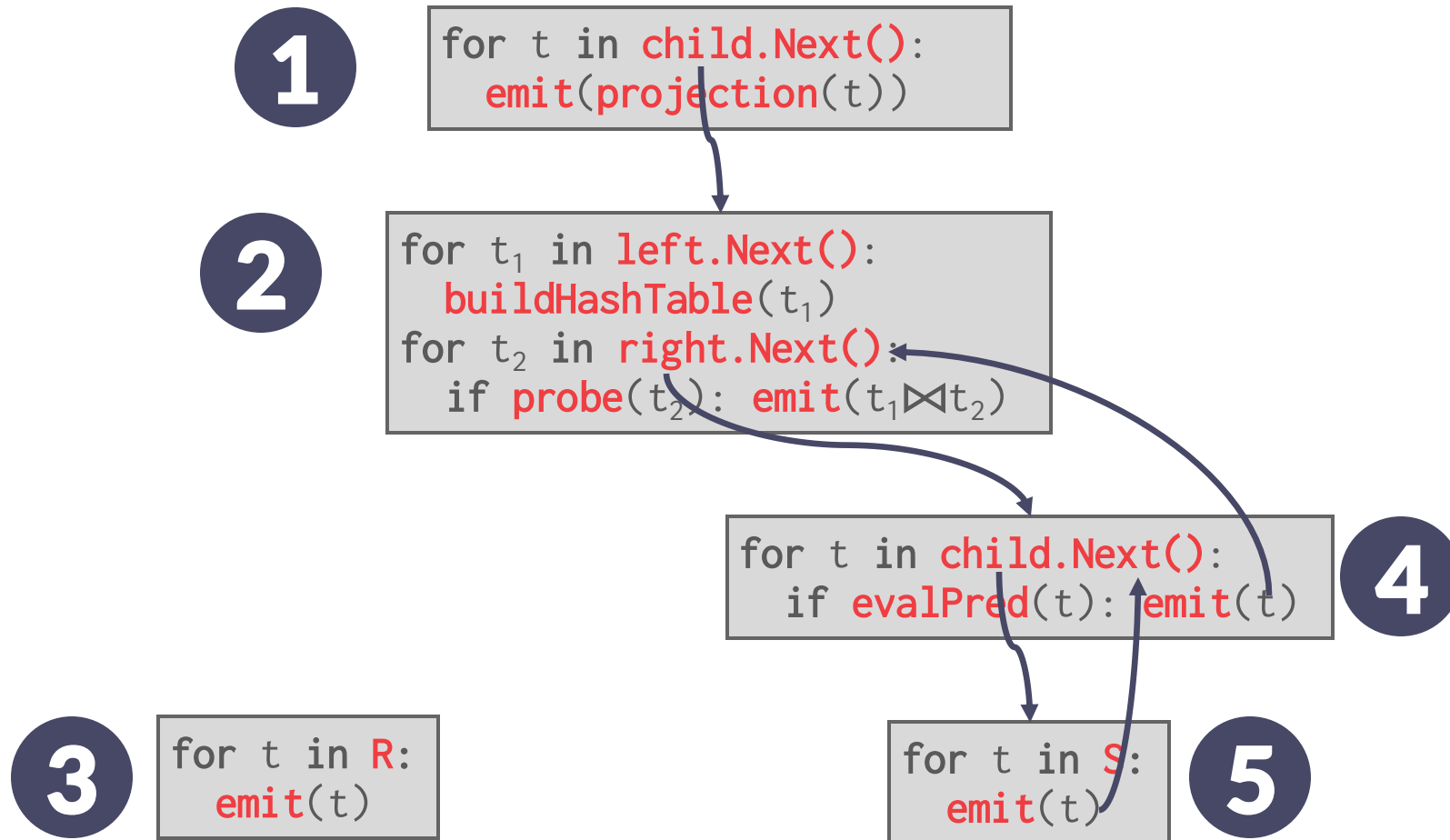
Iterator Model



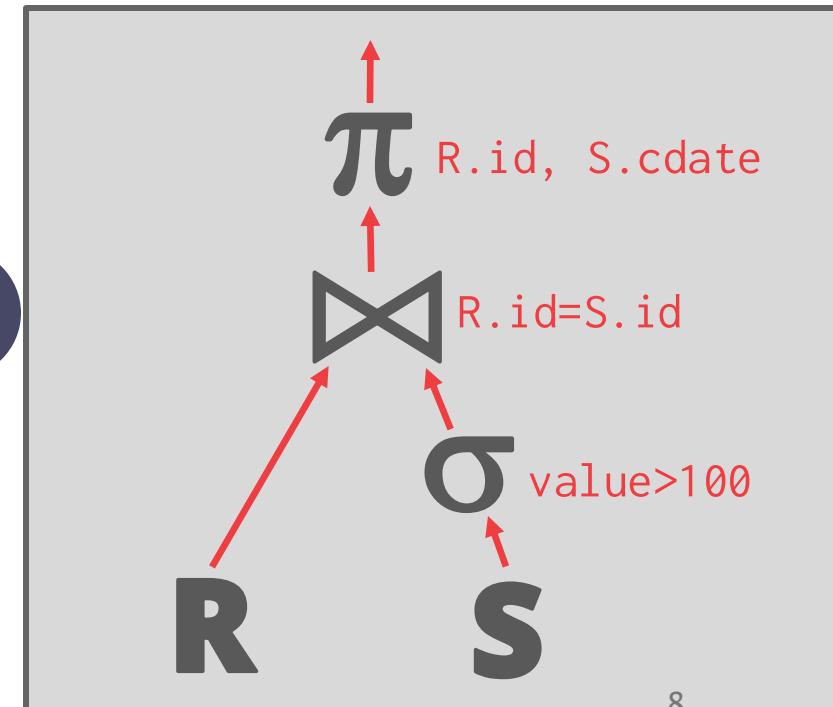
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



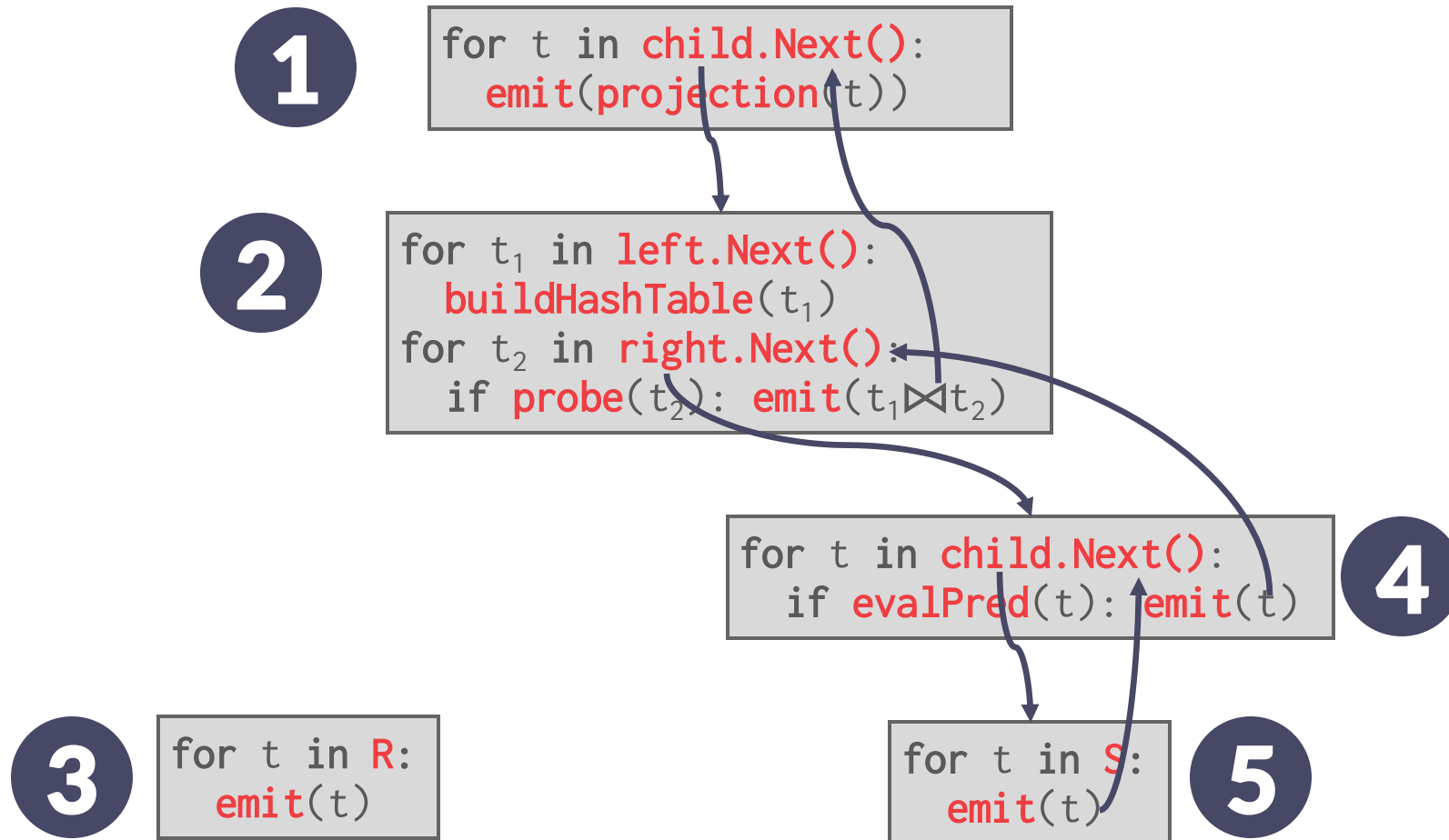
Iterator Model



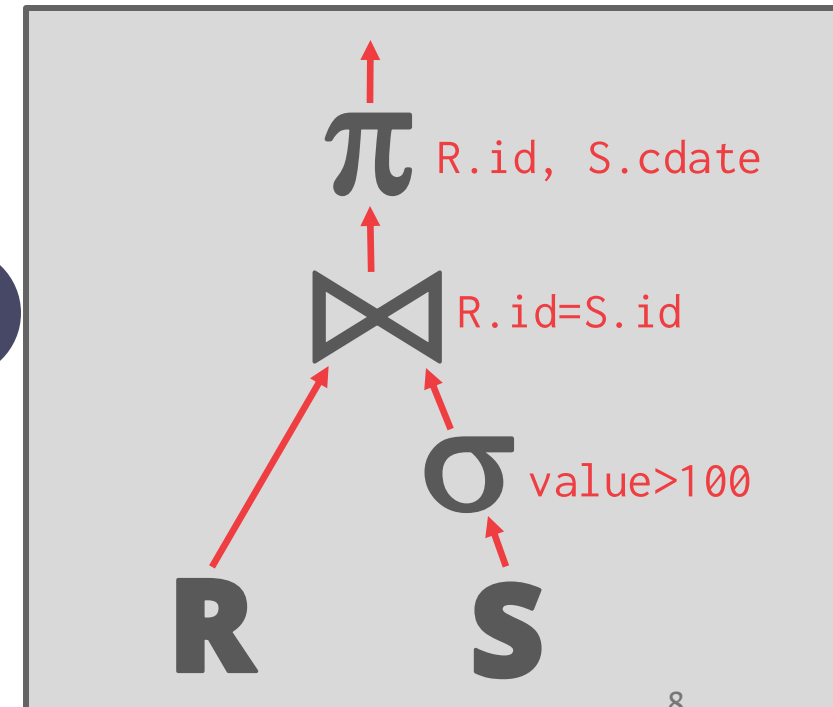
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Iterator Model



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



Iterator Model

- This is used in most DBMSs today. Allows for tuple **pipelining**.
- Many operators must block until their children emit all their tuples.
 - Joins, Aggregates, Subqueries, Order By
- Output control works easily with this approach.



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

```
out = [ ]
for t in child.Output():
    out.add(projection(t))
return out
```

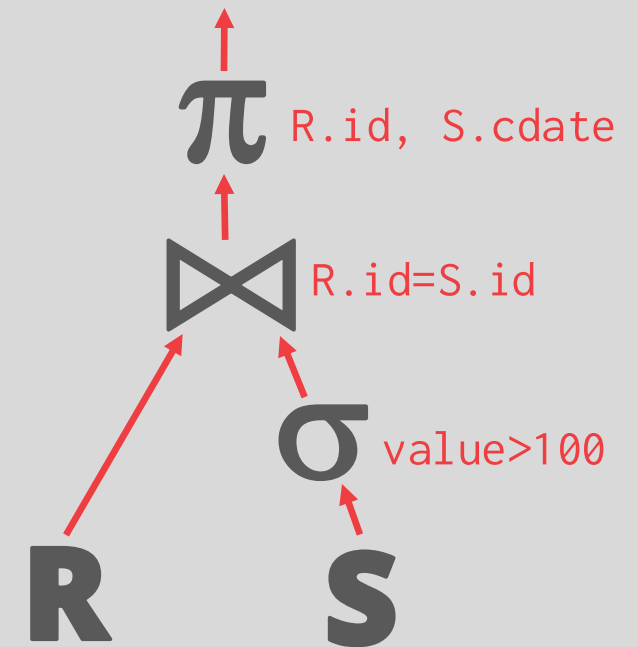
```
out = [ ]
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1 ⋈ t2)
return out
```

```
out = [ ]
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in R:
    out.add(t)
return out
```

```
out = [ ]
for t in S:
    out.add(t)
return out
```

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



Materialization Model

1

```
out = [ ]
for t in child.Output():
    out.add(projection(t))
return out
```

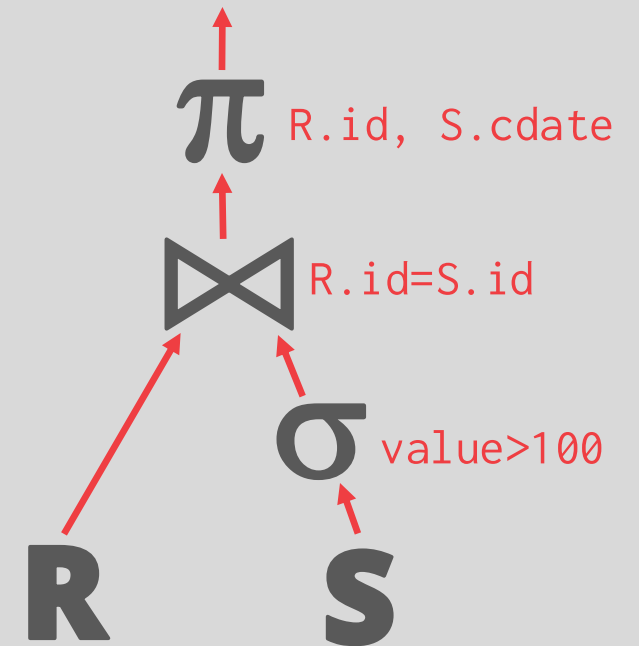
```
out = [ ]
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1 ⋈ t2)
return out
```

```
out = [ ]
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in R:
    out.add(t)
return out
```

```
out = [ ]
for t in S:
    out.add(t)
return out
```

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



Materialization Model

1

```
out = [ ]
for t in child.Output():
    out.add(projection(t))
return out
```

2

```
out = [ ]
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1 ⋈ t2)
return out
```

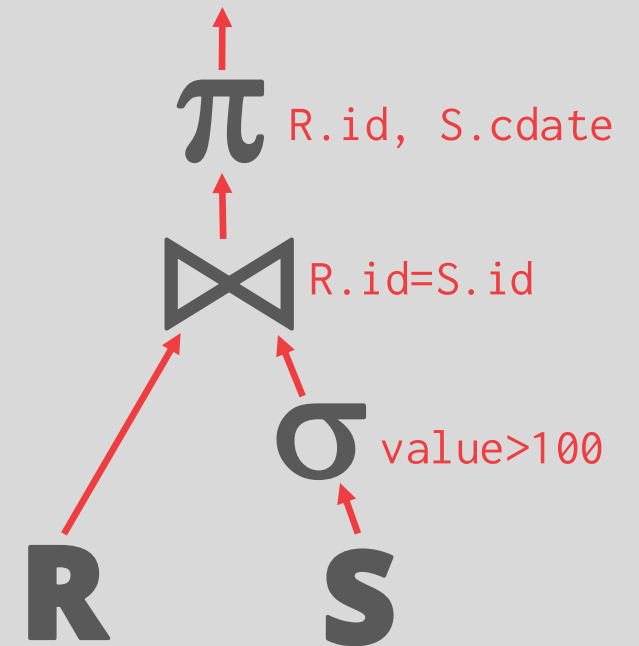
3

```
out = [ ]
for t in R:
    out.add(t)
return out
```

```
out = [ ]
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in S:
    out.add(t)
return out
```

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



Materialization Model

1

```
out = [ ]
for t in child.Output():
    out.add(projection(t))
return out
```

2

```
out = [ ]
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1 ⋈ t2)
return out
```

3

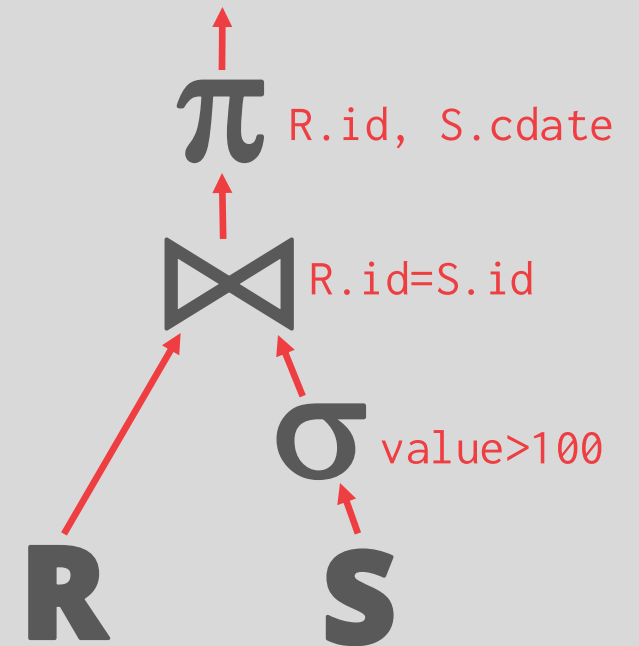
```
out = [ ]
for t in R:
    out.add(t)
return out
```

All Tuples

```
out = [ ]
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in S:
    out.add(t)
return out
```

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



Materialization Model

1

```
out = [ ]
for t in child.Output():
    out.add(projection(t))
return out
```

2

```
out = [ ]
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1 ⋈ t2)
return out
```

3

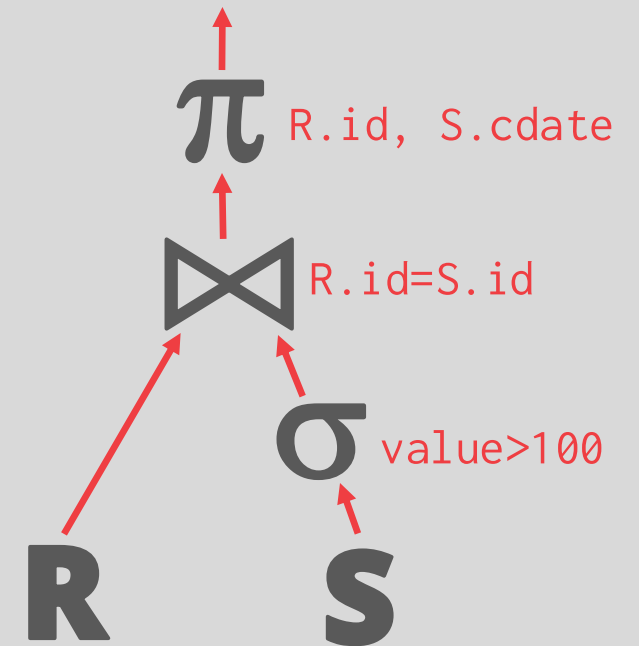
```
out = [ ]
for t in R:
    out.add(t)
return out
```

4

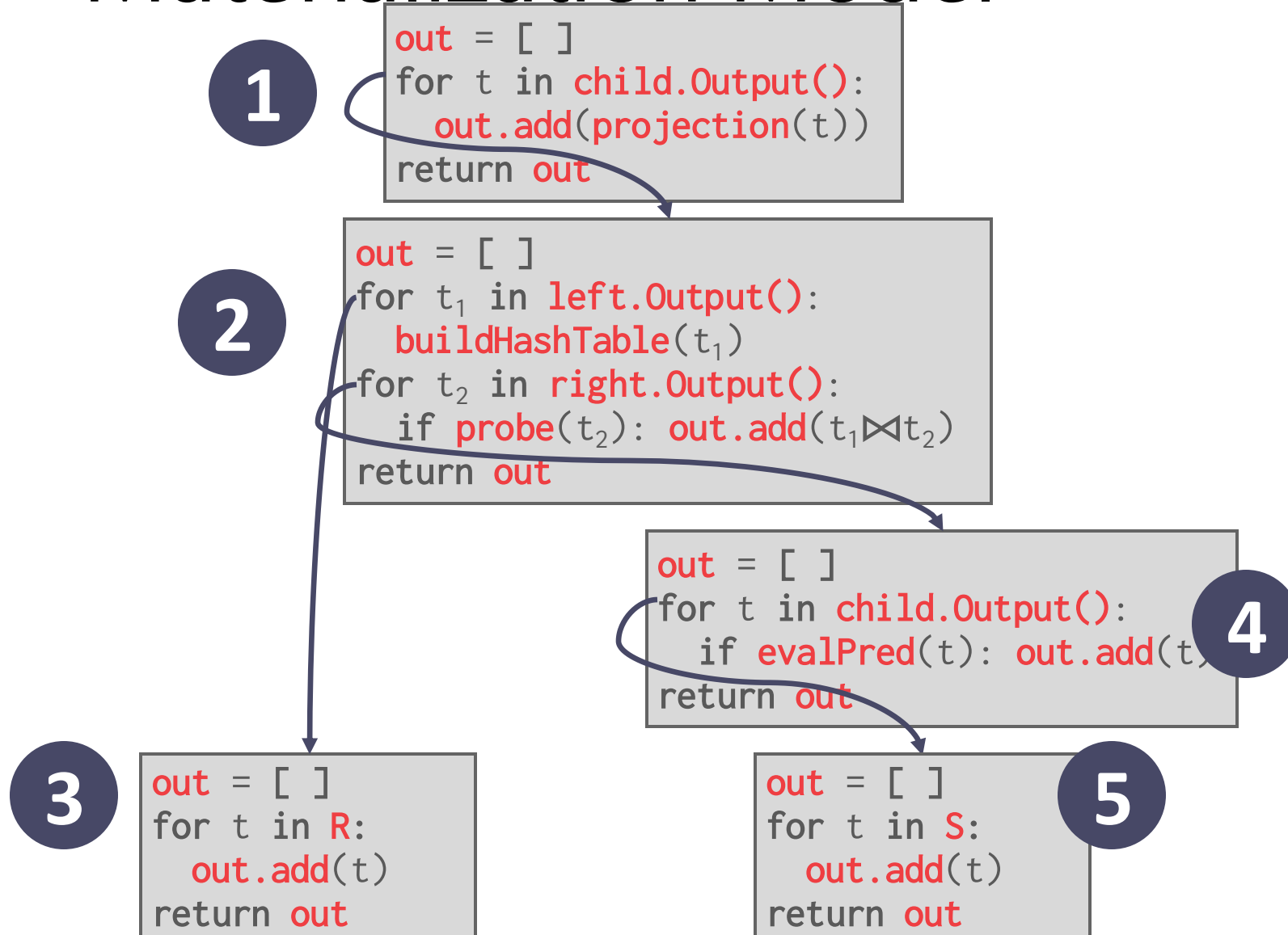
```
out = [ ]
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in S:
    out.add(t)
return out
```

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

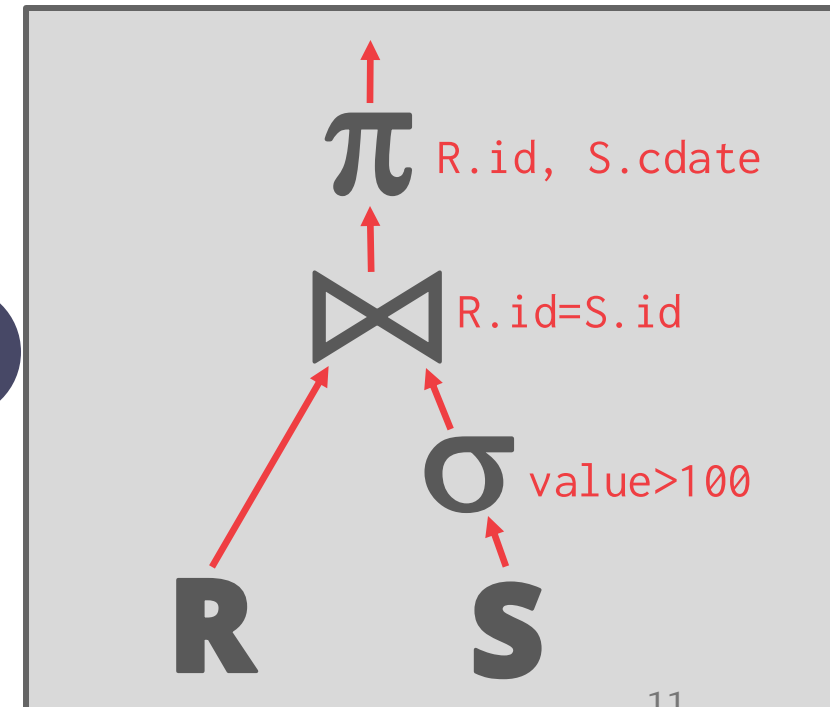


Materialization Model

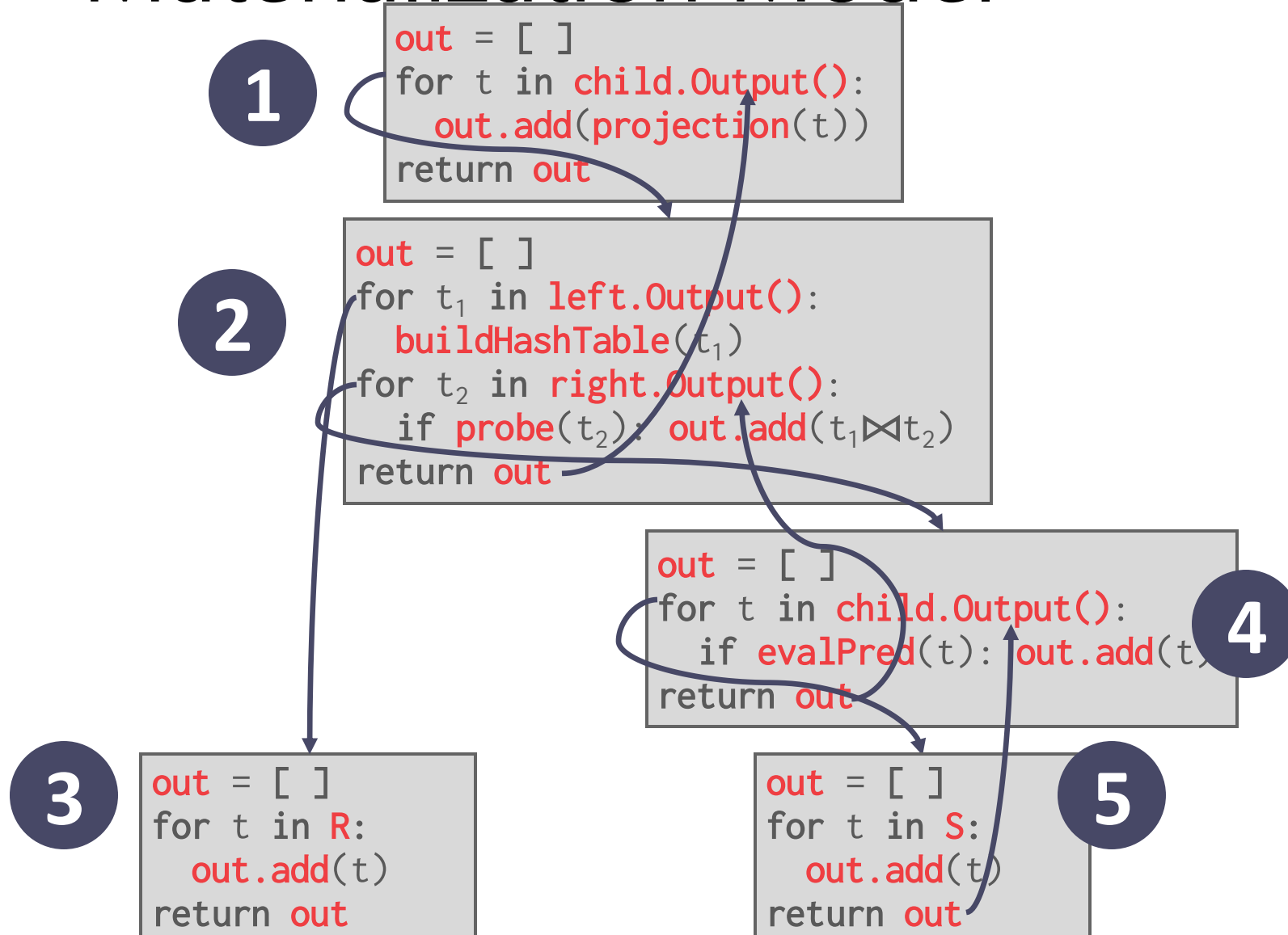


```

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



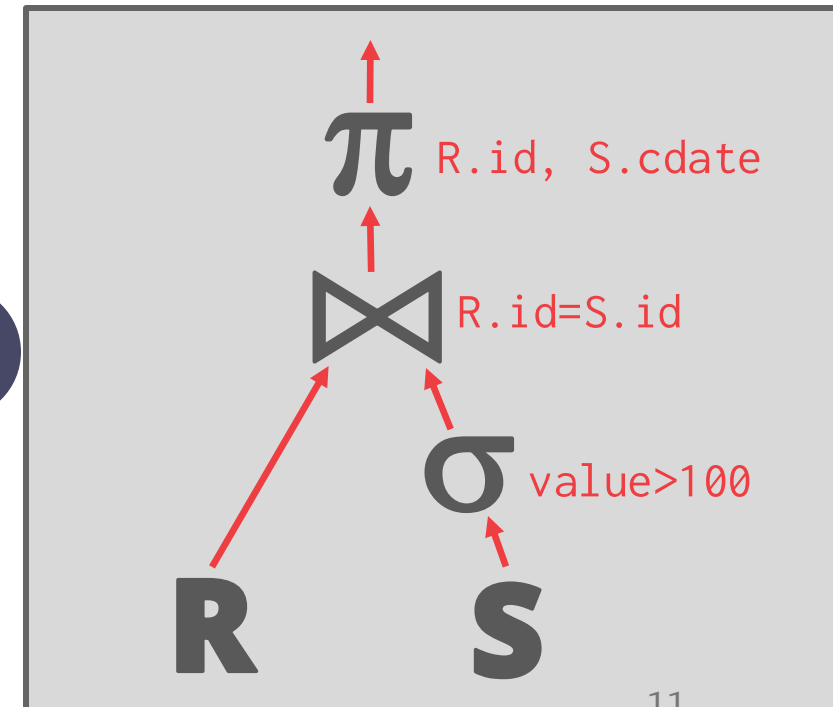
Materialization Model



```

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 good for OLAP queries with large intermediate results.

 **VOLTDB** **RAVENDB** **monetdb** **+ CrateDB**

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.

Vectorization Model

```
out = [ ]
for t in child.Next():
    out.add(projection(t))
    if |out|>n: emit(out)
```

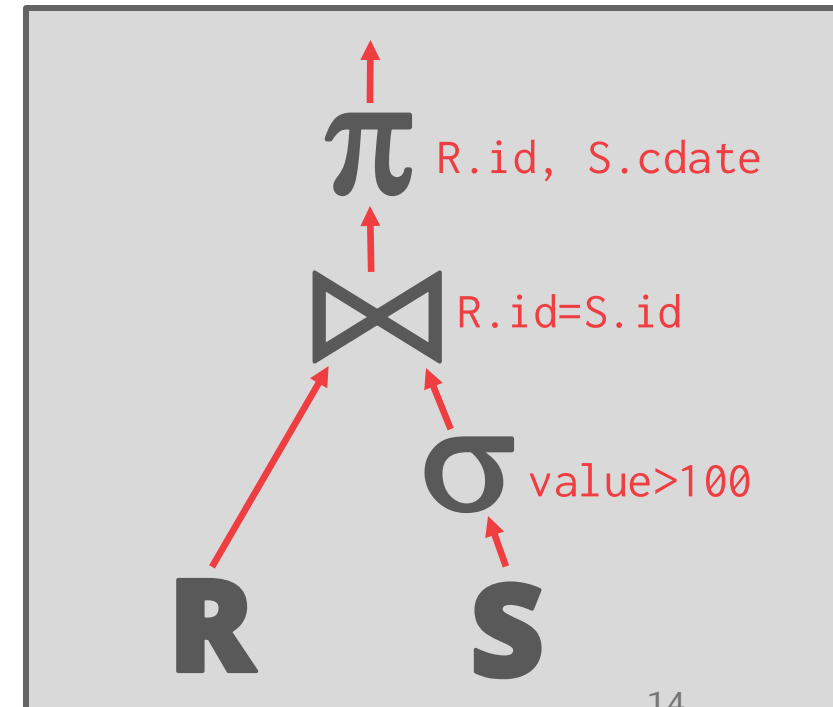
```
out = [ ]
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): out.add(t1 ⋈ t2)
    if |out|>n: emit(out)
```

```
out = [ ]
for t in child.Next():
    if evalPred(t): out.add(t)
    if |out|>n: emit(out)
```

```
out = [ ]
for t in R:
    out.add(t)
    if |out|>n: emit(out)
```

```
out = [ ]
for t in S:
    out.add(t)
    if |out|>n: emit(out)
```

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



Vectorization Model

1

```

out = [ ]
for t in child.Next():
    out.add(projection(t))
    if |out| > n: emit(out)
    
```

2

```

out = [ ]
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): out.add(t1 ⋈ t2)
    if |out| > n: emit(out)
    
```

```

out = [ ]
for t in child.Next():
    if evalPred(t): out.add(t)
    if |out| > n: emit(out)
    
```

3

```

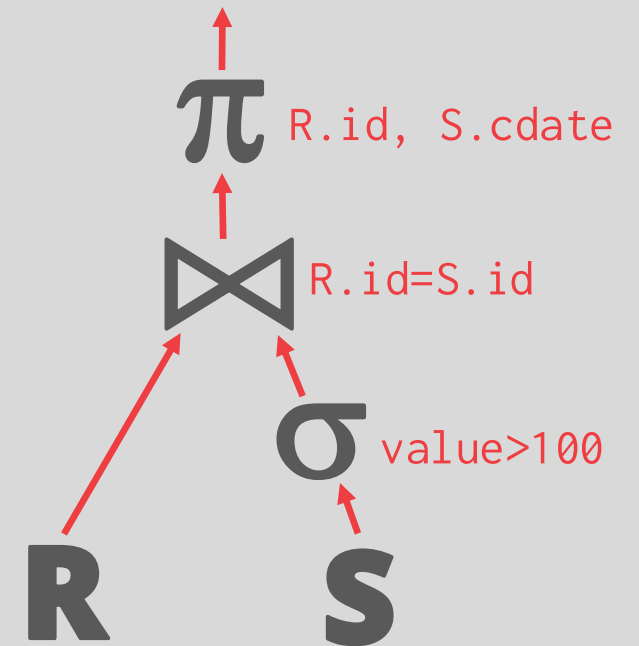
out = [ ]
for t in R:
    out.add(t)
    if |out| > n: emit(out)
    
```

```

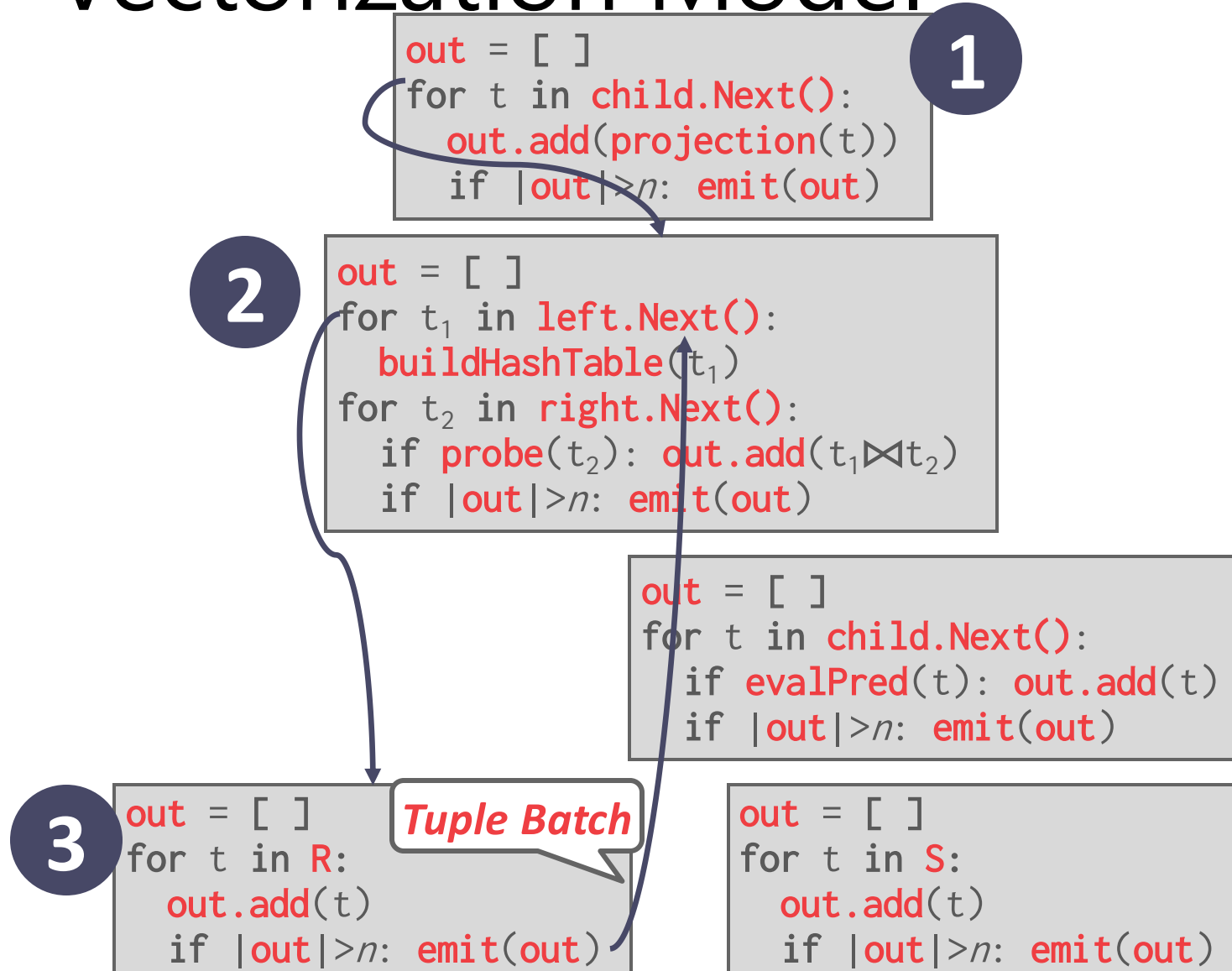
out = [ ]
for t in S:
    out.add(t)
    if |out| > n: emit(out)
    
```

```

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

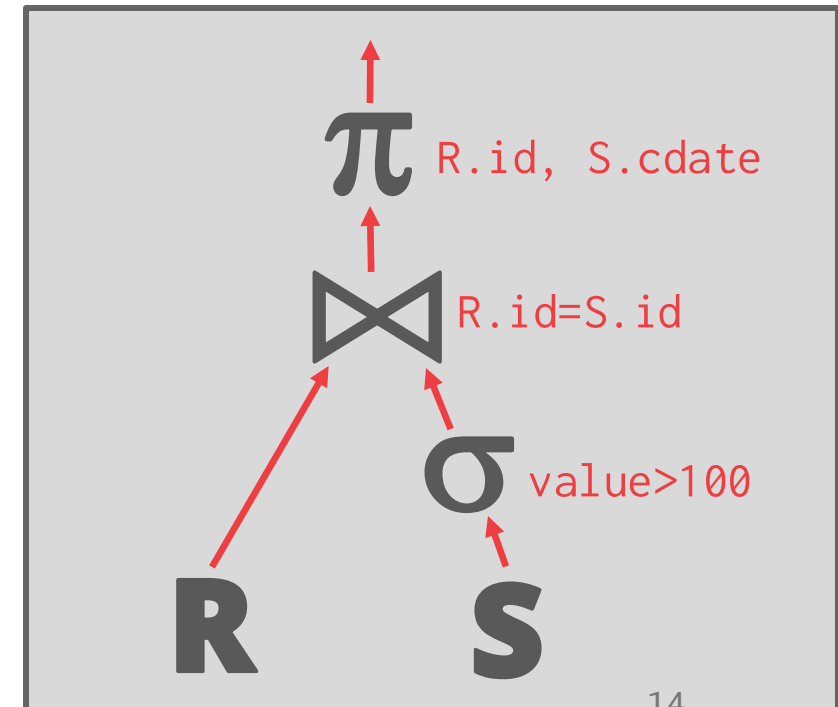


Vectorization Model

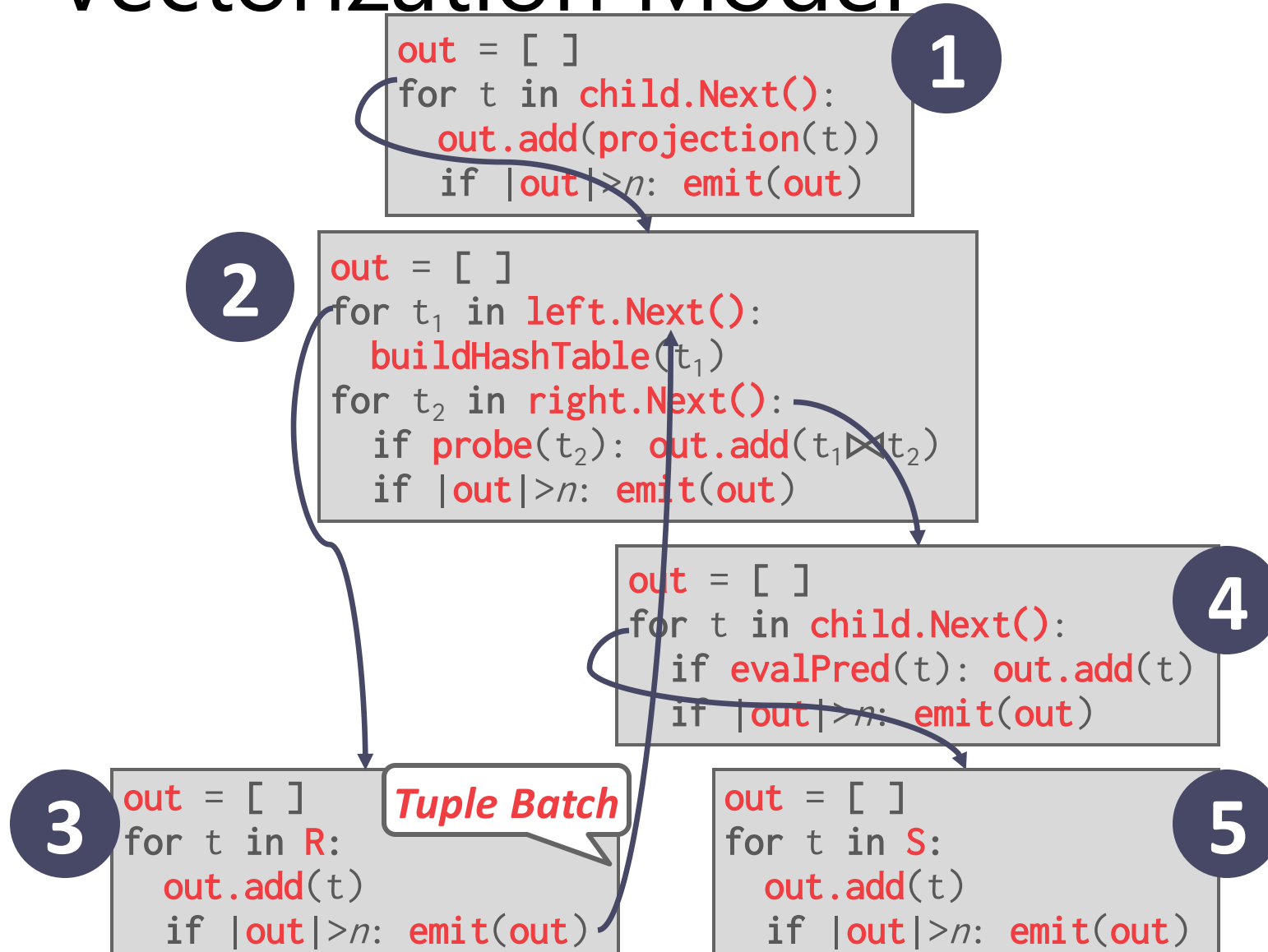


```

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

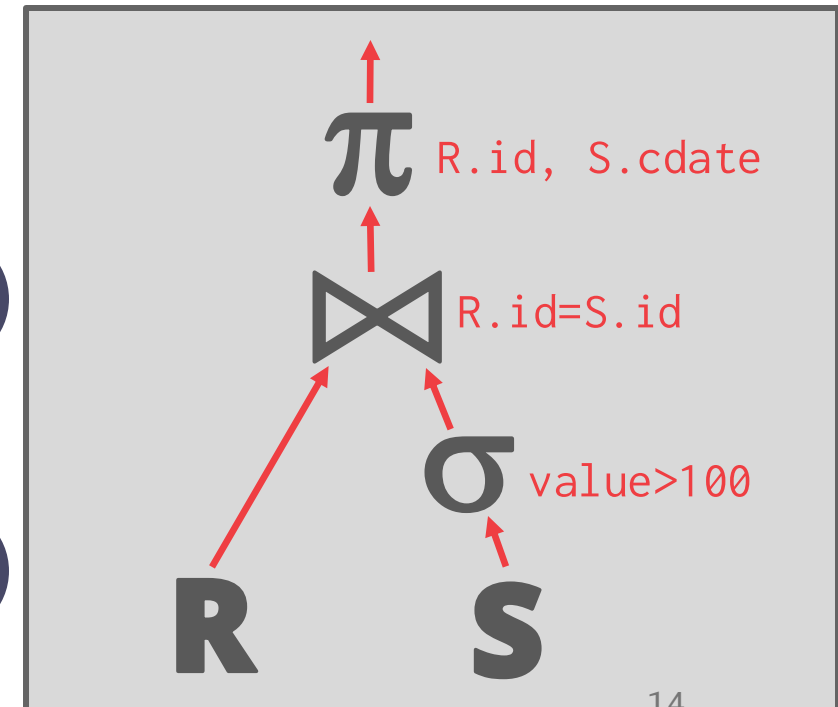


Vectorization Model



```

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 for operators to more easily use vectorized (SIMD) instructions to process batches of tuples.



Plan Processing Direction

- **Approach #1: Top-to-Bottom**

- Start with the root and “pull” data up from its children.
- Tuples are always passed with function calls.

- **Approach #2: Bottom-to-Top**

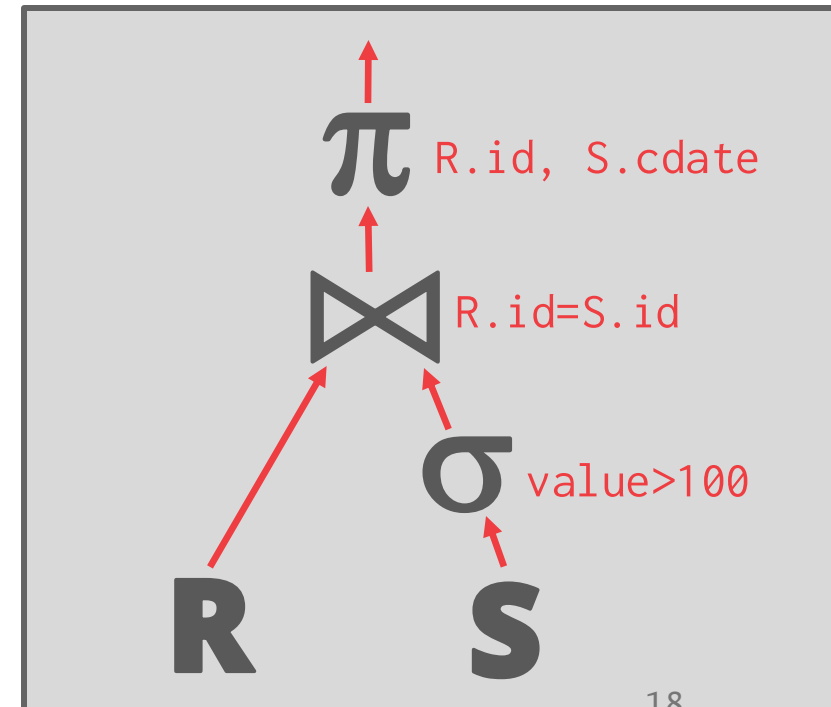
- Start with leaf nodes and push data to their parents.
- Allows for tighter control of caches/registers in pipelines.
- More amenable to dynamic query re-optimization.

Access Methods

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.

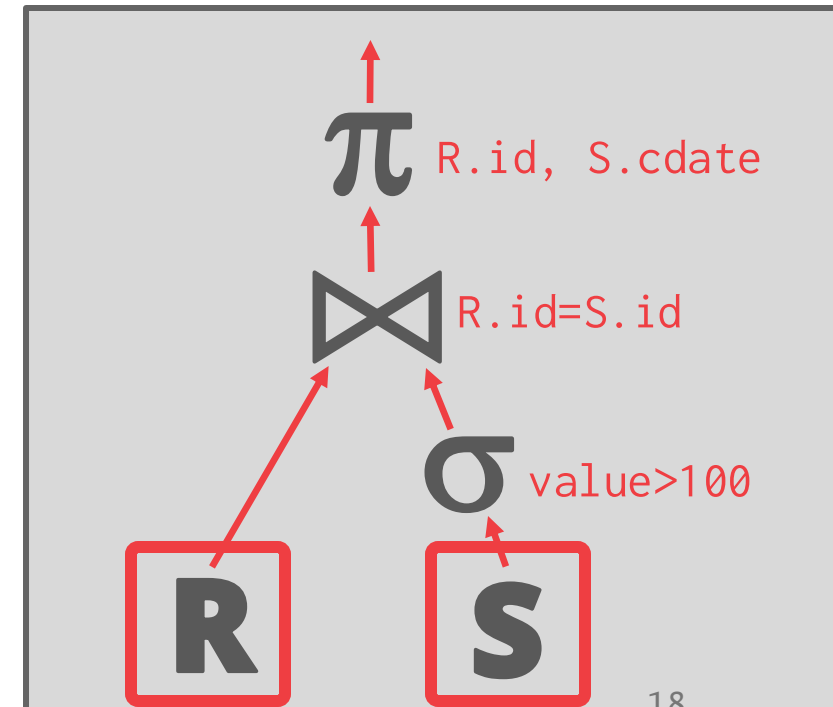
```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



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.

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



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 / slot it examined.

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

Sequential Scan: Optimizations

- This is almost always the worst thing that the DBMS can do to execute a query, but it may be the only choice available.
- Sequential Scan Optimizations:
 - Prefetching
 - Buffer Pool Bypass
 - Parallelization
 - Heap Clustering
 - Late Materialization
 - Data Skipping

Sequential Scan: Optimizations

- This is almost always the worst thing that the DBMS can do to execute a query, but it may be the only choice available.
- Sequential Scan Optimizations:
 - Prefetching
 - Buffer Pool Bypass
 - Parallelization
 - Heap Clustering
 - Late Materialization
 - Data Skipping

Data Skipping

- **Approach #1: Approximate Queries (Lossy)**

- Execute queries on a sampled subset of the entire table to produce approximate results.
- Examples: [BlinkDB](#), [Redshift](#), [ComputeDB](#), [XDB](#), [Oracle](#), [Snowflake](#), [Google BigQuery](#), [DataBricks](#)

- **Approach #2: 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.
- Examples: [Oracle](#), Vertica, SingleStore, [Netezza](#), Snowflake, Google BigQuery

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.

Original Data

val
100
200
300
400
400

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.

Original Data

val
100
200
300
400
400



Zone Map

type	val
<i>MIN</i>	100
<i>MAX</i>	400
<i>AVG</i>	280
<i>SUM</i>	1400
<i>COUNT</i>	5

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

Zone Maps

- Pre-computed aggregate values in a page. first to decide whether page.

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

Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing

Guido Moerkotte
moer@pi3.informatik.uni-mannheim.de

Lehrstuhl für praktische Informatik III, Universität Mannheim, Germany

Abstract

Small Materialized Aggregates (SMAs for short) are considered a highly flexible and versatile alternative for materialized data cubes. The basic idea is to compute many aggregate values for small to medium-sized buckets of tuples. These aggregates are then used to speed up query processing. We present the general idea and present an application of SMAs to the TPC-D benchmark. We show that exploiting SMAs for TPC-D Query 1 results in a speed up of two orders of magnitude. Then, we investigate the problem of query processing in the presence of SMAs. Last, we briefly discuss some further tuning possibilities for SMAs.

1 Introduction

Among the predominant demands put on data warehouse management systems (DWMSs) is performance, i.e., the highly efficient evaluation of complex analytical queries. A very successful means to speed up query processing is the exploitation of index structures. Several index structures have been applied to data warehouse management systems (for an overview see [2, 17]). Among them are traditional index structures [1, 3, 6], bitmaps [15], and R-tree-like structures [9].

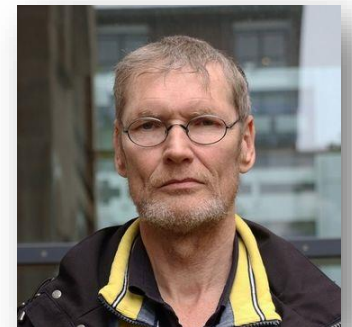
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

Since most of the queries against data warehouses incorporate grouping and aggregation, it seems to be a good idea to materialize according views. The most popular of these approaches is the materialized data cube where for a set of dimensions, for all their possible grouping combinations, the aggregates of interest are materialized. Then, query processing against a data cube boils down to a very efficient lookup. Since the complete data cube is very space consuming [5, 18], strategies have been developed for materializing only those parts of a data cube that pay off most in query processing [10]. Another approach—based on [14]—is to hierarchically organize the aggregates [12]. But still the storage consumption can be very high, even for a simple grouping possibility, if the number of dimensions and/or their cardinality grows. On the user side, the data cube operator has been proposed to allow for easier query formulation [8]. But since we deal with performance here, we will throughout the rest of the paper use the term *data cube* to refer to a *materialized data cube* used to speed up query processing.

Besides high storage consumption, the biggest disadvantage of the data cube is its inflexibility. Each data cube implies a fixed number of queries that can be answered with it. As soon as for example an additional selection condition occurs in the query, the data cube might not be applicable any more. Furthermore, for queries not foreseen by the data cube designer, the data cube is useless. This argument applies also to alternative structures like the one presented in [12]. This inflexibility—together with the extraordinary space consumption—maybe the reason why, to the knowledge of the author, data cubes have never been applied to the standard data warehouse benchmark TPC-D [19]. (cf. Section 2.4 for space requirements of a data cube applied to TPC-D data) Our goal was to design an index structure that allows for efficient support of complex queries against high volumes of data as exemplified by the TPC-D benchmark.

The main problem encountered is that some queries



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

- 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

Future Lecture

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'
```

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.

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 that the DBMS can use for a query:
 - 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](#)

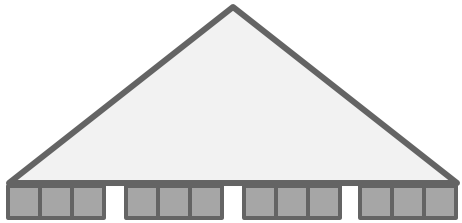
Multi-Index Scan

- With an index on **age** and an index on **dept**:
 - We can retrieve the Record IDs satisfying **age<30** using the first,
 - Then retrieve the Record IDs satisfying **dept='CS'** using the second,
 - Take their intersection
 - Retrieve records and check **country='US'**.

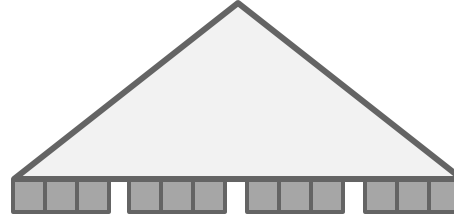
```
SELECT * FROM students
WHERE age < 30
      AND dept = 'CS'
      AND country = 'US'
```


Multi-Index Scan

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



age < 30



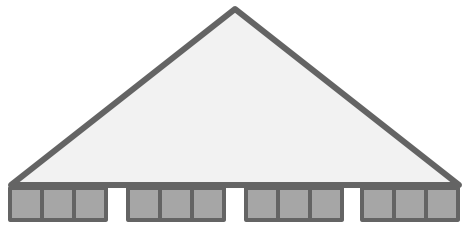
dept = 'CS'

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

Multi-Index Scan

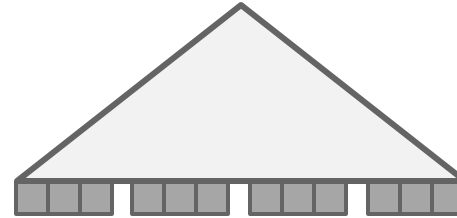
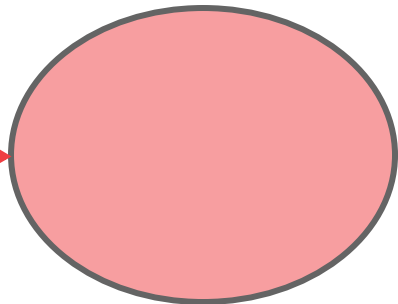
- Set intersection can be done efficiently with bitmaps or hash tables.

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



age < 30

record ids

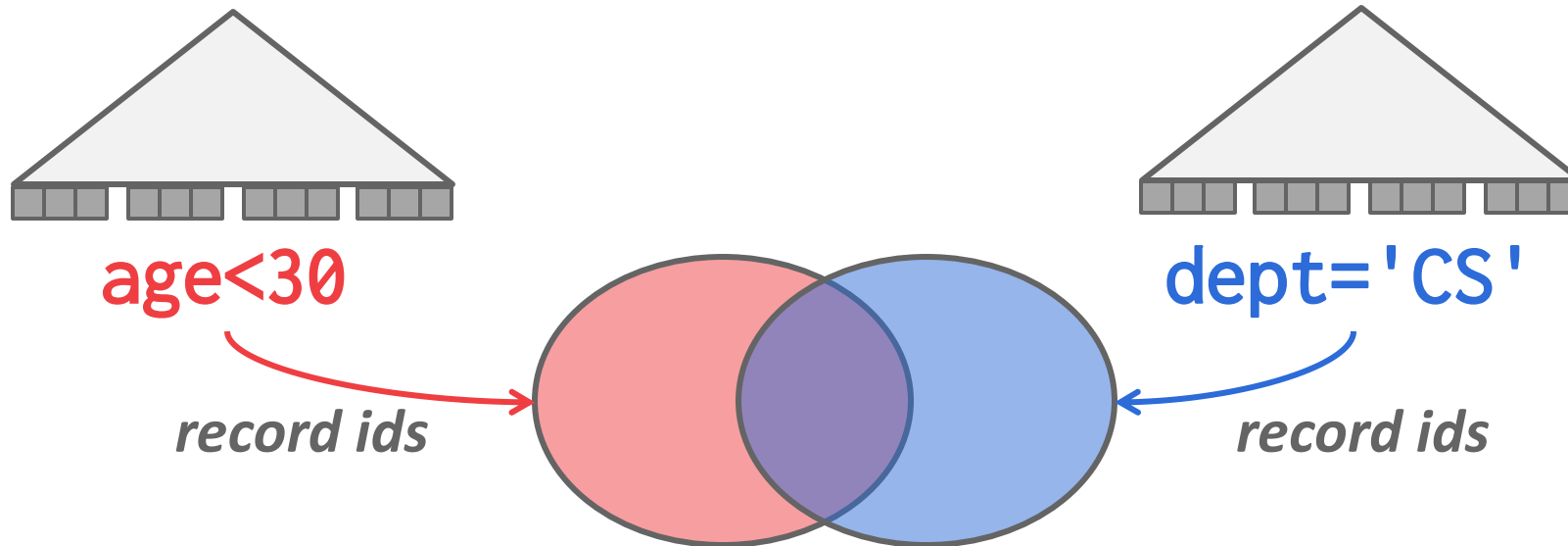


dept = 'CS'

Multi-Index Scan

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

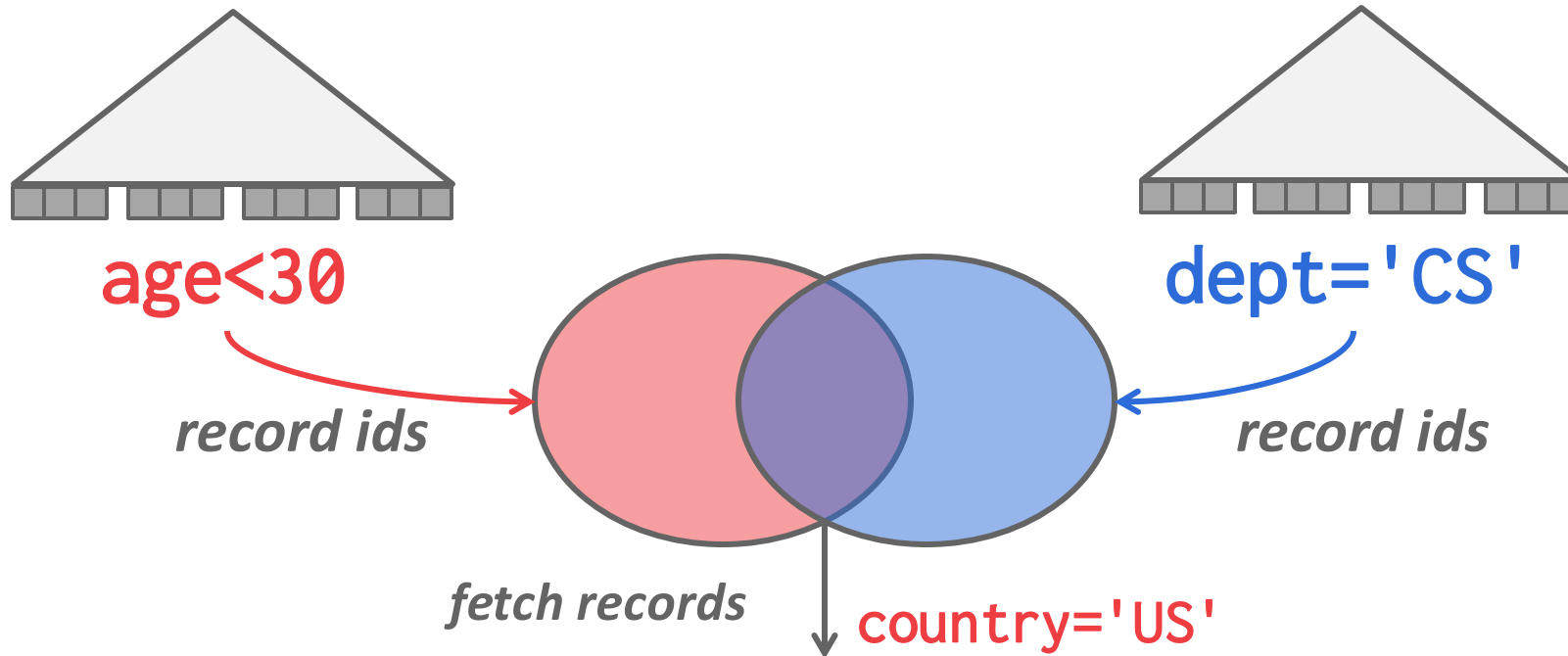
```
SELECT * FROM students
WHERE age < 30
      AND dept = 'CS'
      AND country = 'US'
```



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'
```



Modifications

Modification Queries

- Operators that modify the database (**INSERT**, **UPDATE**, **DELETE**) are responsible for modifying the target table and its indexes.
 - Constraint checks can either happen immediately inside of operator or deferred until later in query/transaction.
- The output of these operators can either be Record Ids or tuple data (i.e., **RETURNING**).

Modification Queries

- **UPDATE/DELETE:**
 - Child operators pass Record IDs for target tuples.
 - Must keep track of previously seen tuples.
- **INSERT:**
 - **Choice #1:** Materialize tuples inside of the operator.
 - **Choice #2:** Operator inserts any tuple passed in from child operators.

Update Query Problem

```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
    SET salary = salary + 100  
WHERE salary < 1100
```


Update Query Problem

```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```



```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
    SET salary = salary + 100  
WHERE salary < 1100
```

Update Query Problem

```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

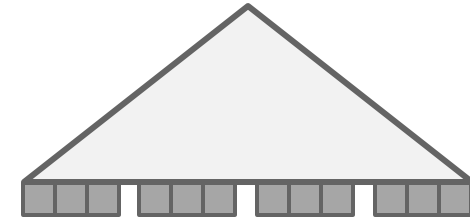


```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
    SET salary = salary + 100  
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

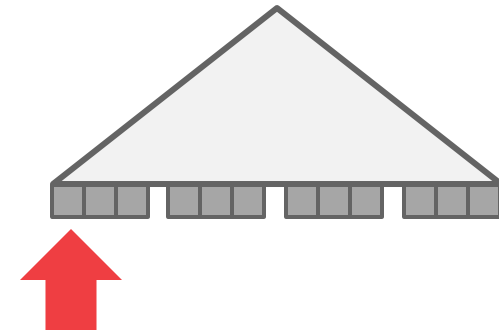


```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
SET salary = salary + 100  
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

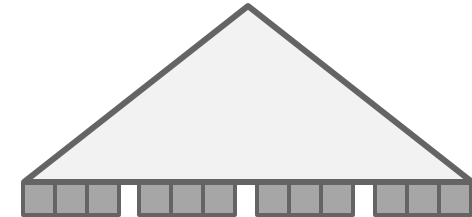
→

```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
SET salary = salary + 100  
WHERE salary < 1100
```

Index(people.salary)



(999, Andy)

Update Query Problem

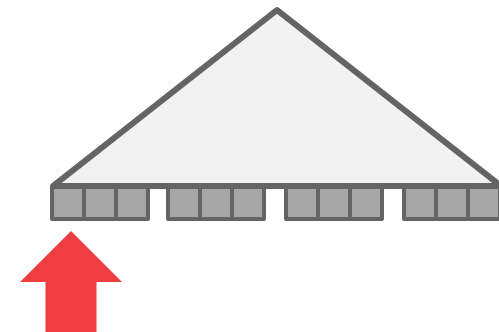
```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
SET salary = salary + 100  
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

```
for t in child.Next():           (999, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)
```

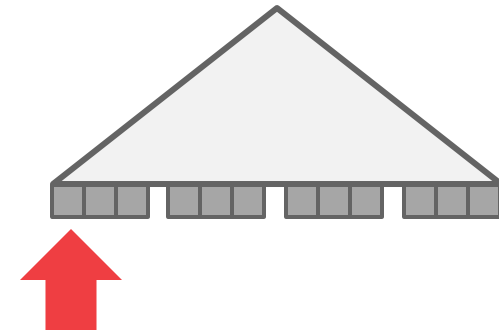
→

```
for t in Indexpeople:
    if t.salary < 1100:
        emit(t)
```

```
CREATE INDEX idx_salary
ON people (salary);
```

```
UPDATE people
SET salary = salary + 100
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

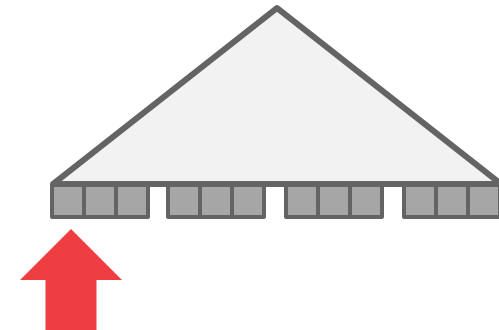
```
for t in child.Next():           (999, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:
    if t.salary < 1100:
        emit(t)
```

```
CREATE INDEX idx_salary
ON people (salary);
```

```
UPDATE people
SET salary = salary + 100
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

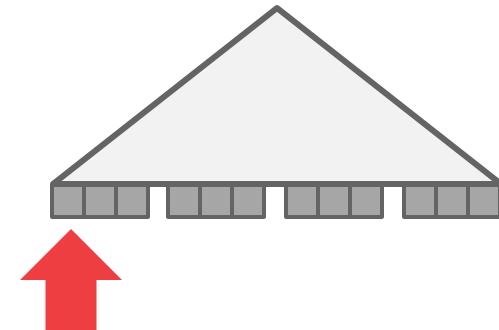
```
for t in child.Next():           (1099, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:
    if t.salary < 1100:
        emit(t)
```

```
CREATE INDEX idx_salary
ON people (salary);
```

```
UPDATE people
SET salary = salary + 100
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

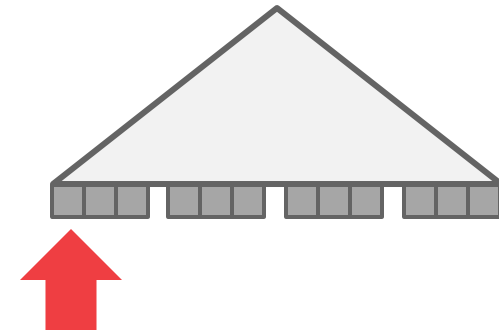
```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
SET salary = salary + 100  
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

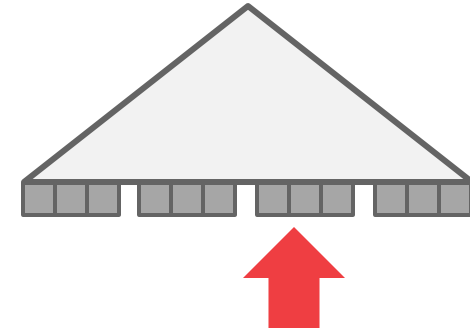


```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
SET salary = salary + 100  
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem

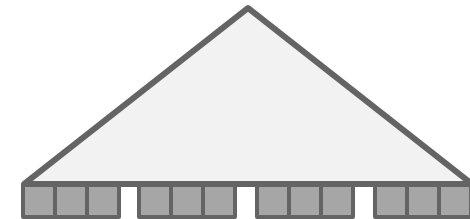
```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
    SET salary = salary + 100  
    WHERE salary < 1100
```

Index(people.salary)



(1099, Andy)

Update Query Problem

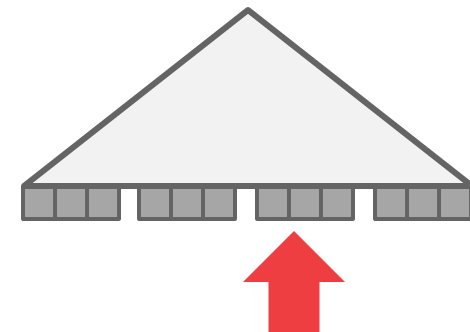
```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

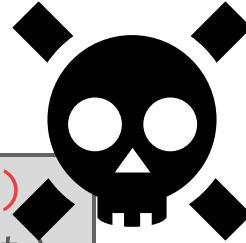
```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
SET salary = salary + 100  
WHERE salary < 1100
```

Index(people.salary)



Update Query Problem



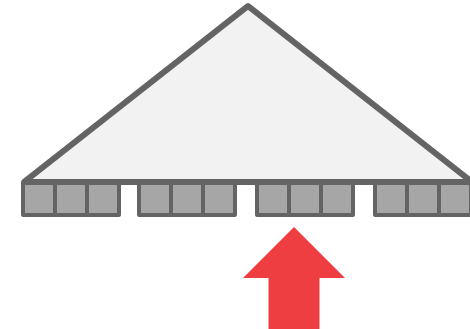
```
CREATE INDEX idx_salary  
ON people (salary);
```

```
UPDATE people  
SET salary = salary + 100  
WHERE salary < 1100
```

```
for t in child.Next():  
    removeFromIndex(idx_salary, t.salary, t)  
    updateTuple(t.salary = t.salary + 100)  
    insertIntoIndex(idx_salary, t.salary, t)
```

```
for t in Indexpeople:  
    if t.salary < 1100:  
        emit(t)
```

Index(people.salary)



Halloween Problem

- Anomaly where an update operation changes the physical location of a tuple, which causes a scan operator to visit the tuple multiple times.
 - Can occur on clustered tables or index scans.
- First discovered by IBM researchers while working on System R on Halloween day in 1976.
- **Solution:** Track modified record ids per query.

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

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


Expression Evaluation

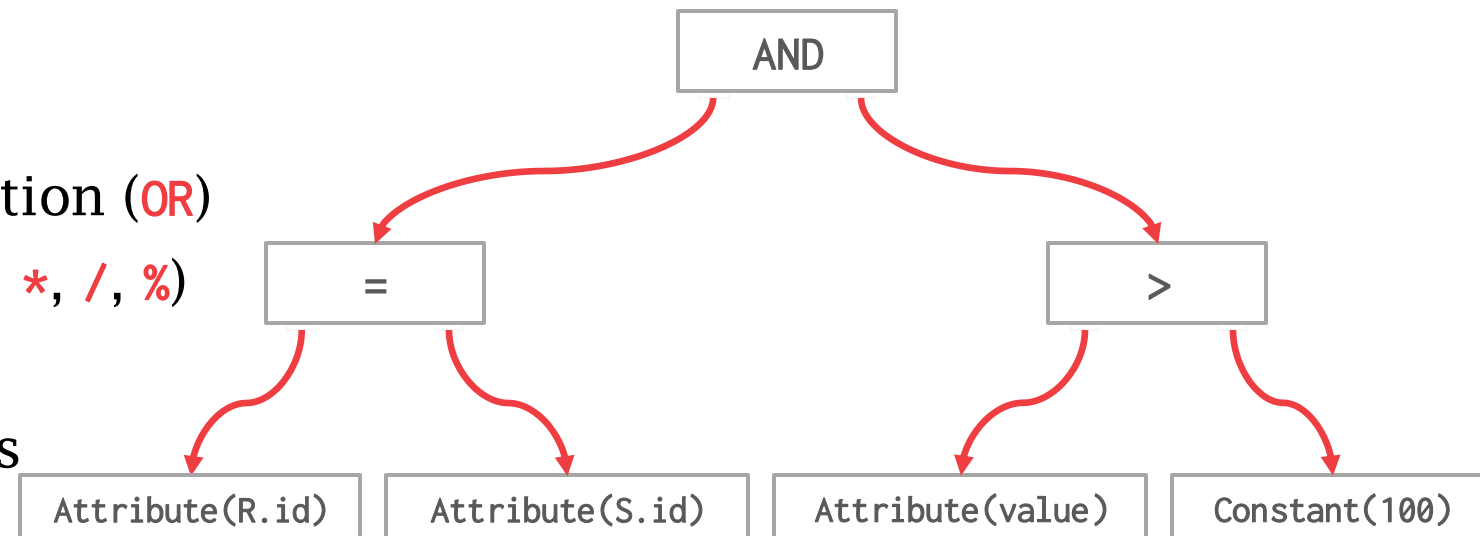
- The DBMS represents a **WHERE** clause as an **expression tree**.
- The nodes in the tree represent different expression types:
 - Comparisons (**=**, **<**, **>**, **!=**)
 - Conjunction (**AND**), Disjunction (**OR**)
 - Arithmetic Operators (**+**, **-**, *****, **/**, **%**)
 - Constant Values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

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

Expression Evaluation

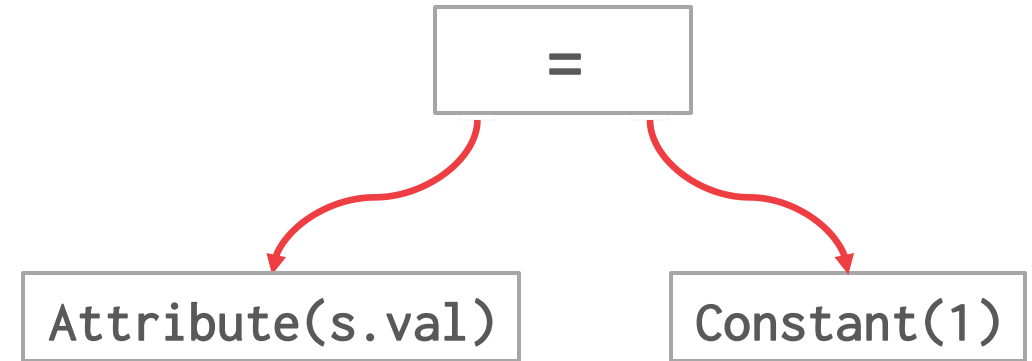
- The DBMS represents a **WHERE** clause as an **expression tree**.
- The nodes in the tree represent different expression types:
 - Comparisons (**=**, **<**, **>**, **!=**)
 - Conjunction (**AND**), Disjunction (**OR**)
 - Arithmetic Operators (**+**, **-**, *****, **/**, **%**)
 - Constant Values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

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



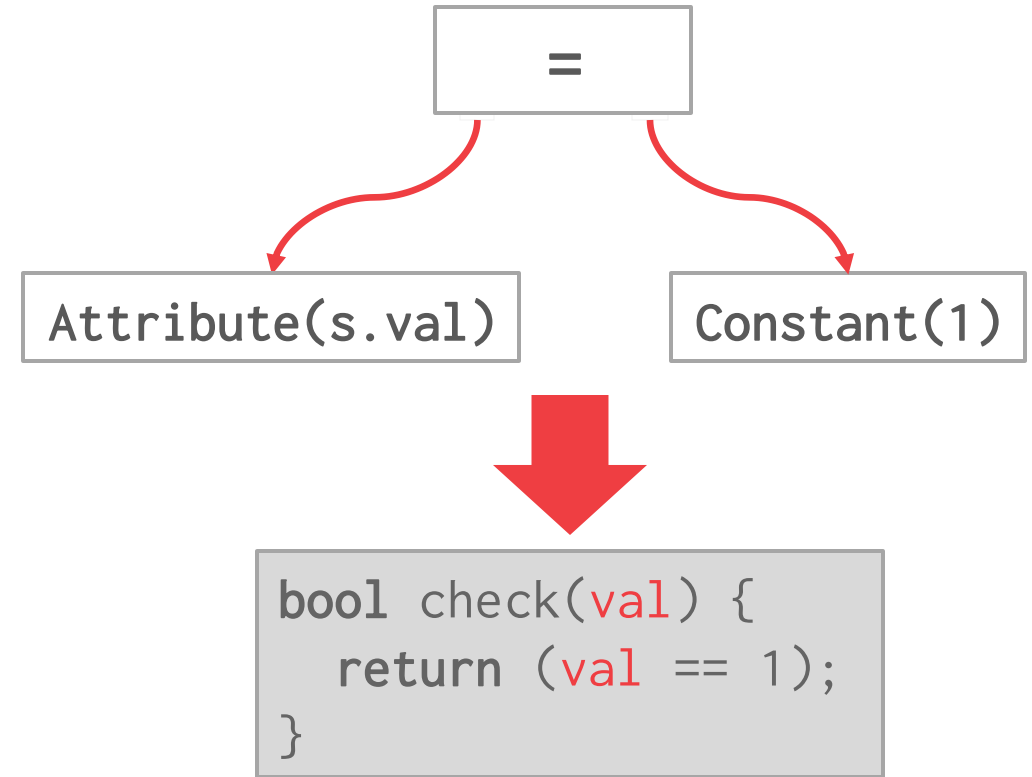
Expression Evaluation

- Evaluating predicates in this manner is slow.
 - The DBMS traverses the tree and for each node that it visits, it must figure out what the operator needs to do.
- Consider this predicate:
WHERE S.val=1
- A better approach is to just evaluate the expression directly.
 - Think JIT compilation



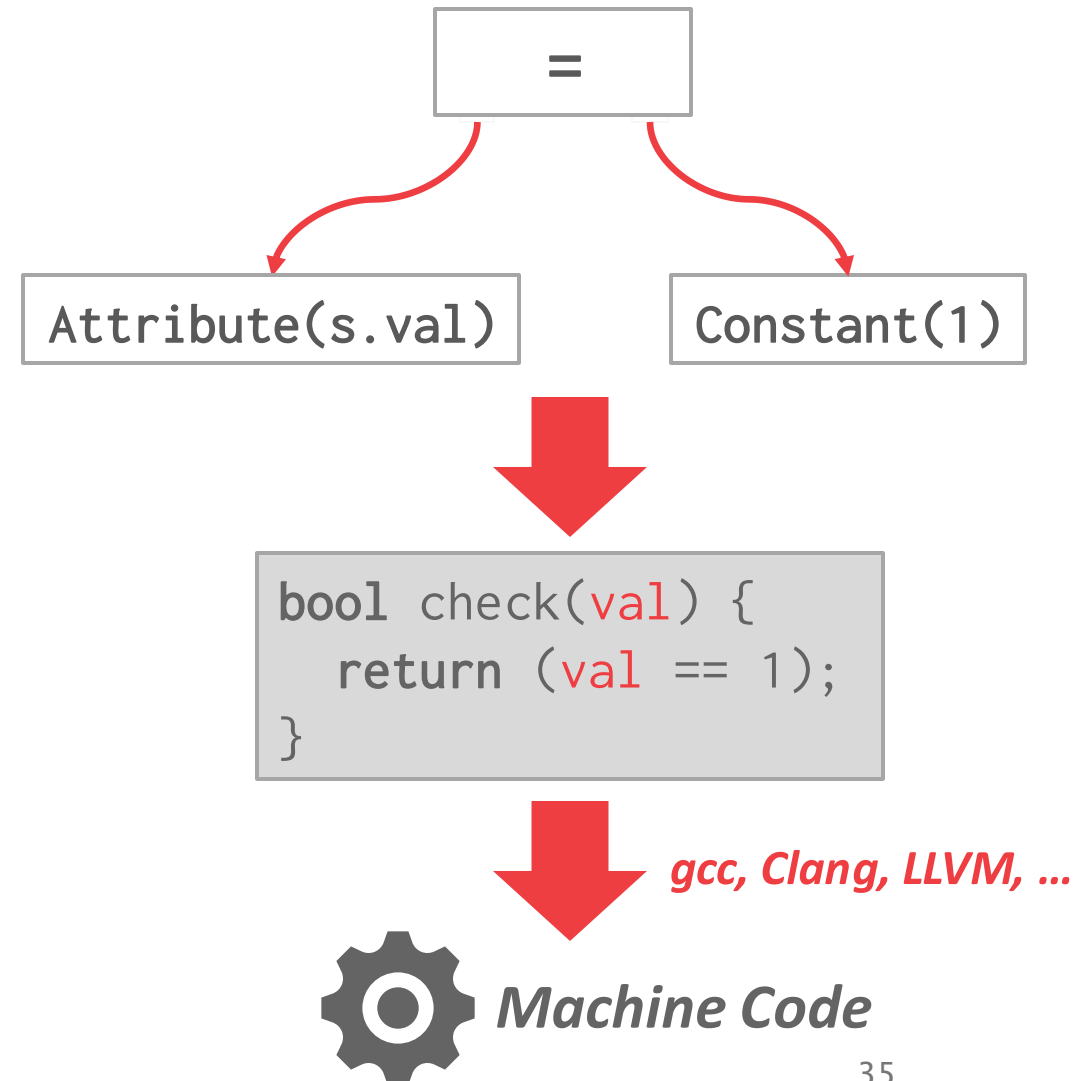
Expression Evaluation

- Evaluating predicates in this manner is slow.
 - The DBMS traverses the tree and for each node that it visits, it must figure out what the operator needs to do.
- Consider this predicate:
WHERE S.val=1
- A better approach is to just evaluate the expression directly.
 - Think JIT compilation



Expression Evaluation

- Evaluating predicates in this manner is slow.
 - The DBMS traverses the tree and for each node that it visits, it must figure out what the operator needs to do.
- Consider this predicate:
WHERE S.val=1
- A better approach is to just evaluate the expression directly.
 - Think JIT compilation



Expression Evaluation

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

```
EXECUTE xxx(991)
```

Expression Evaluation

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

```
EXECUTE xxx(991)
```

Expression Evaluation

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

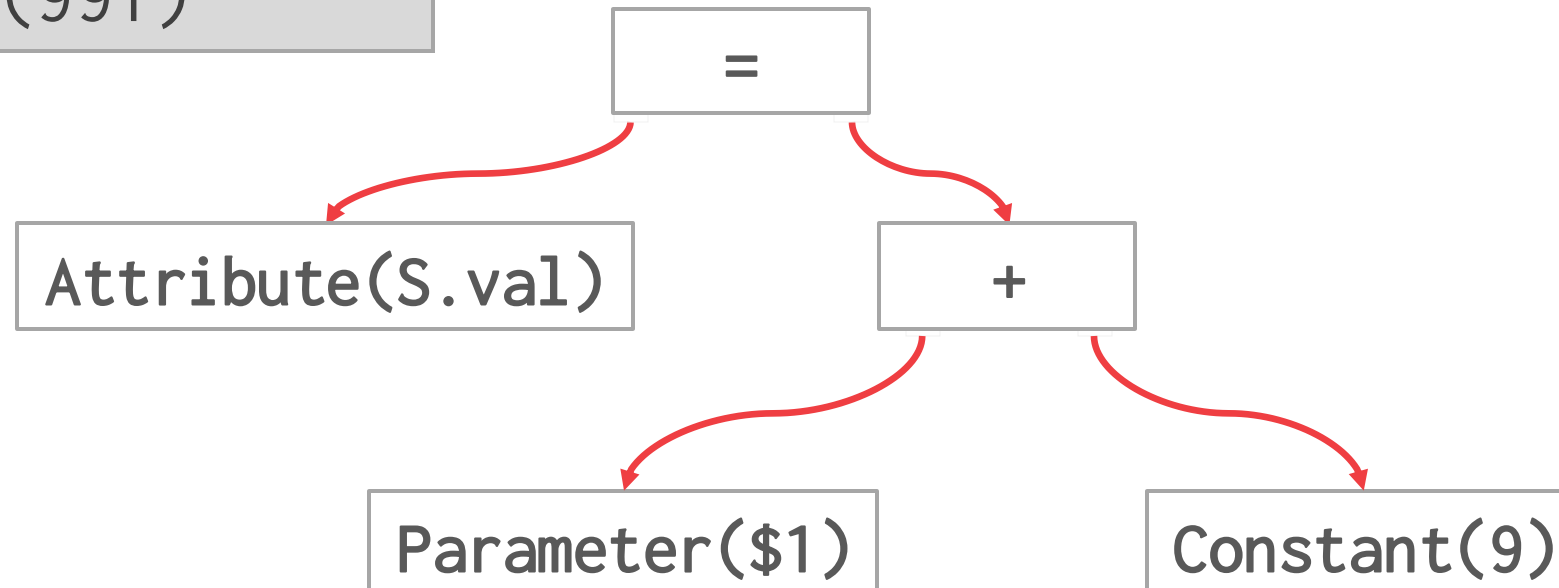
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

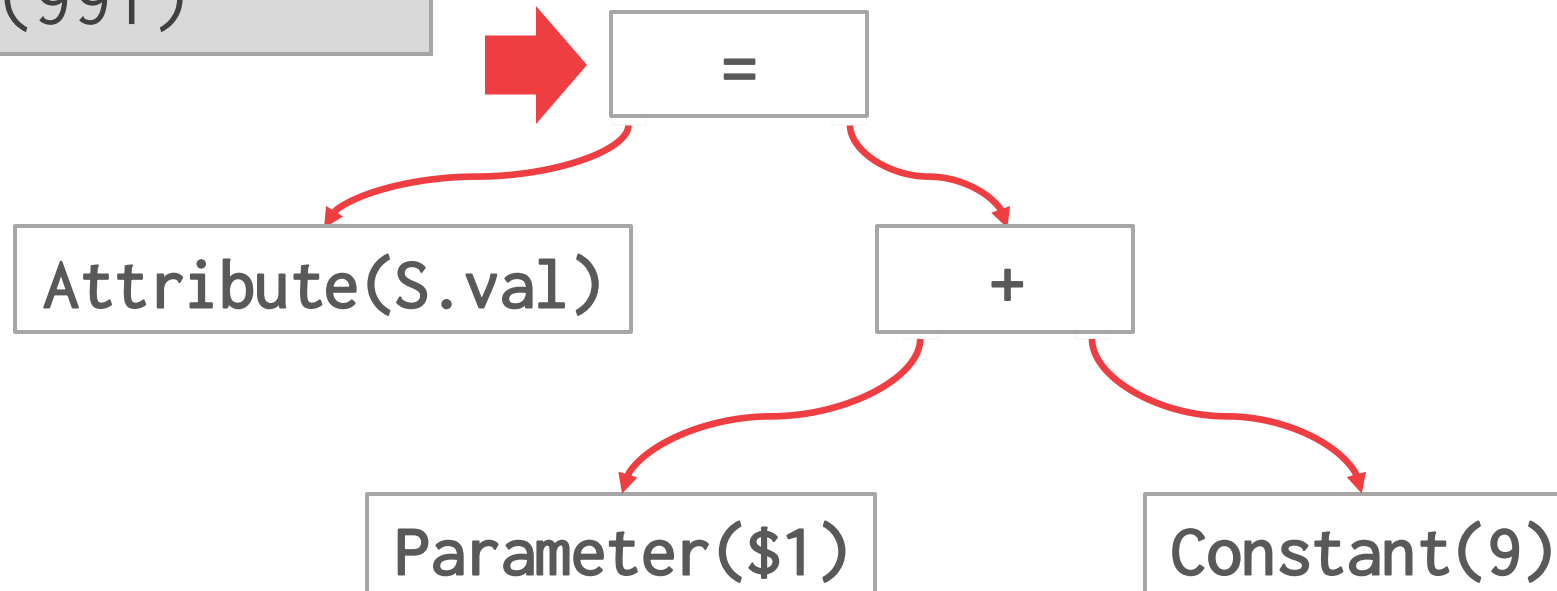
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

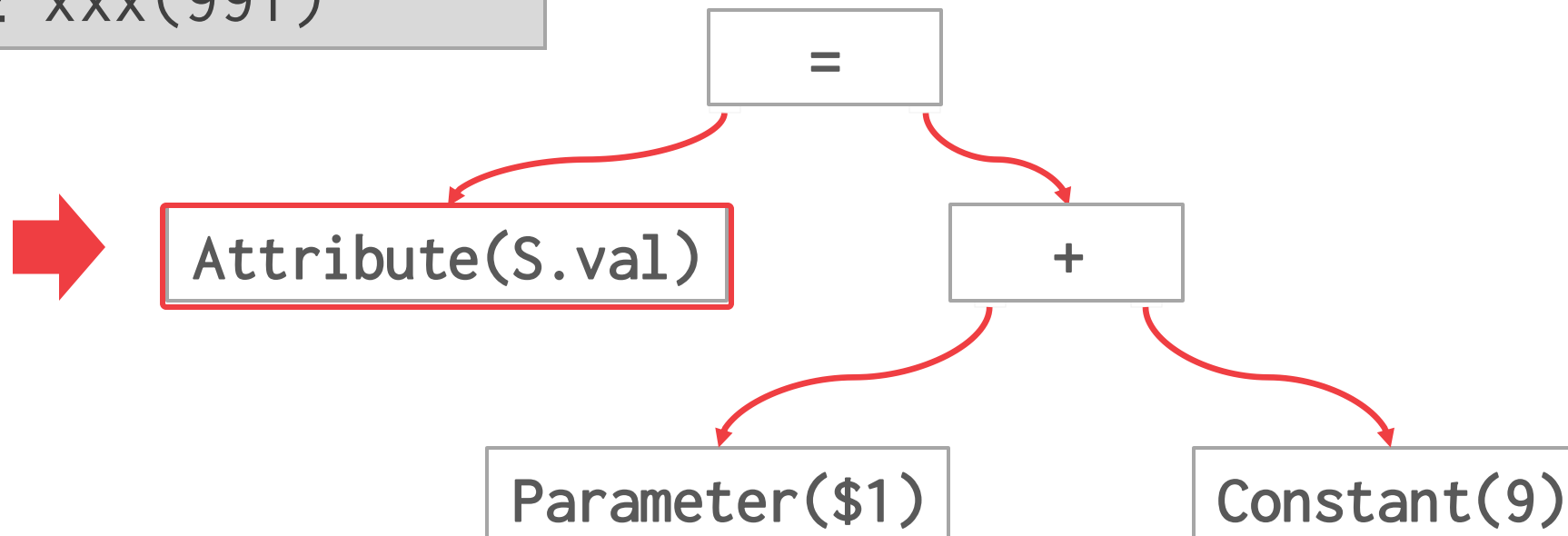
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

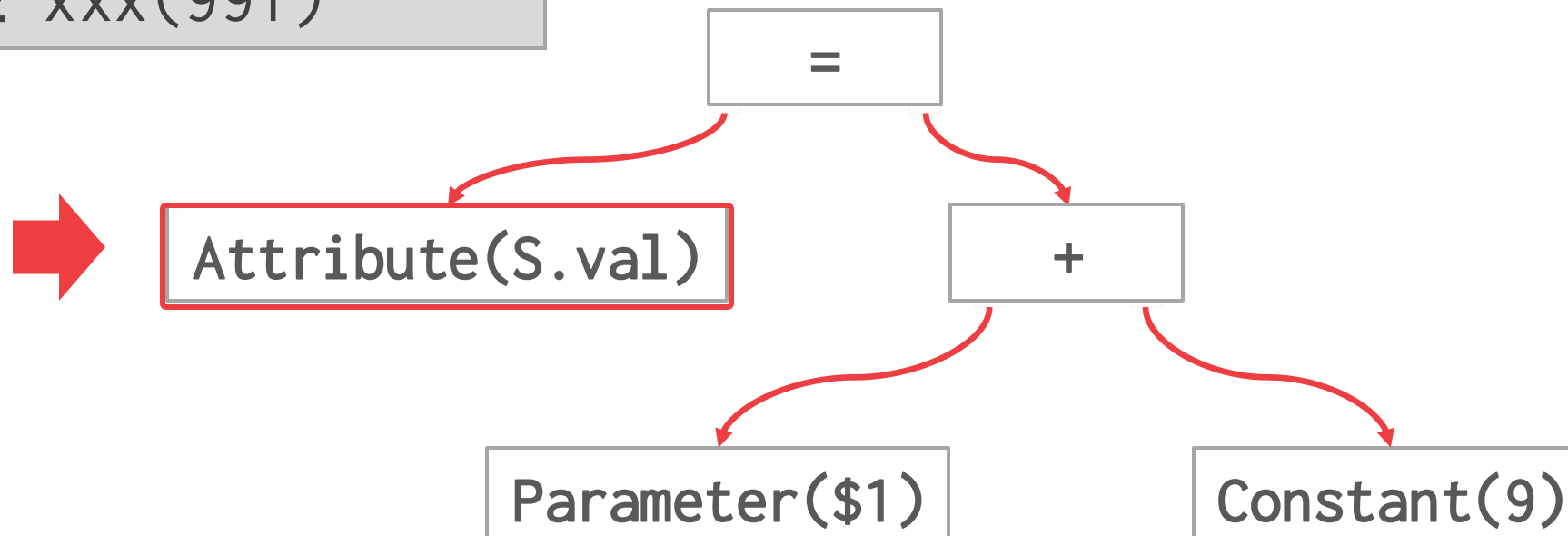
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

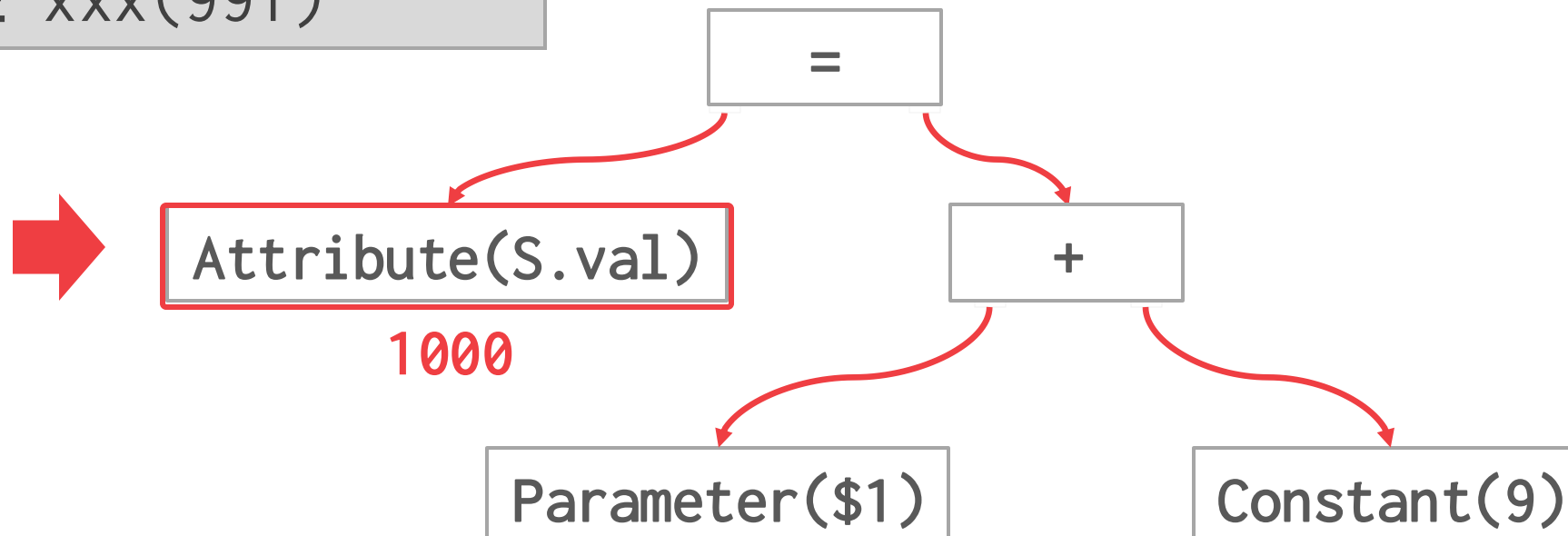
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

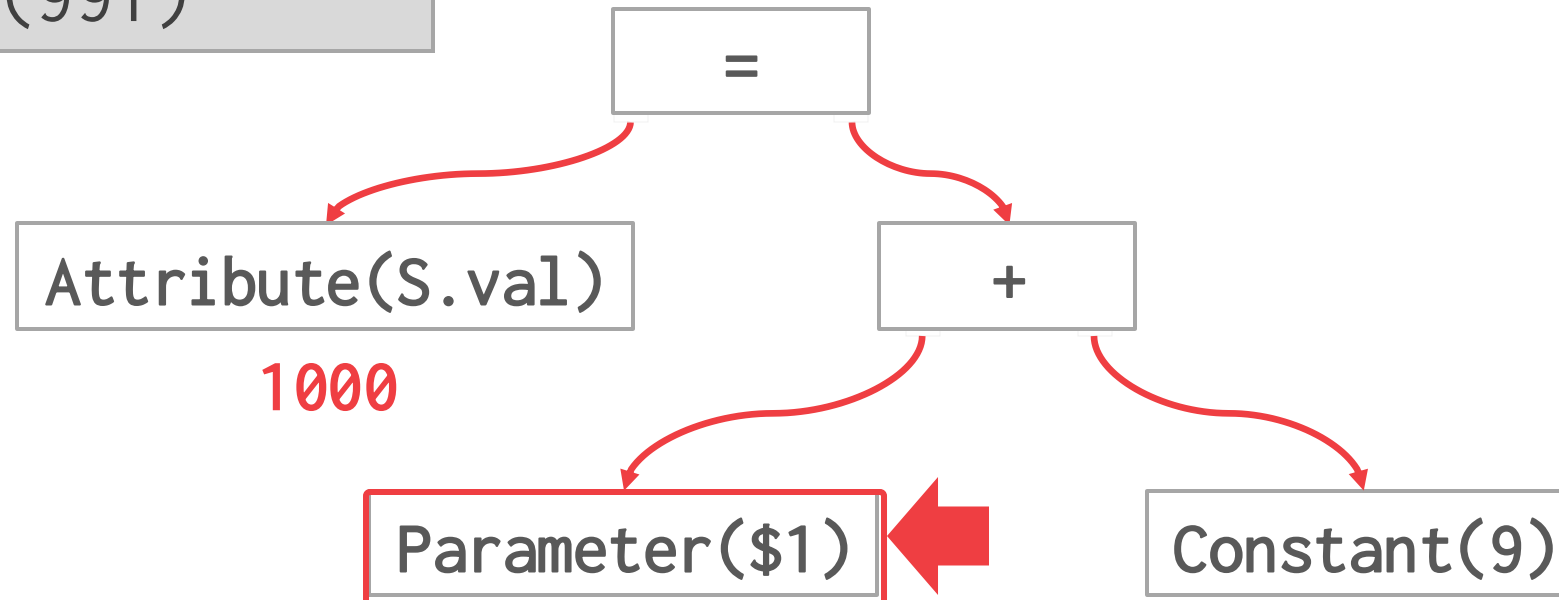
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

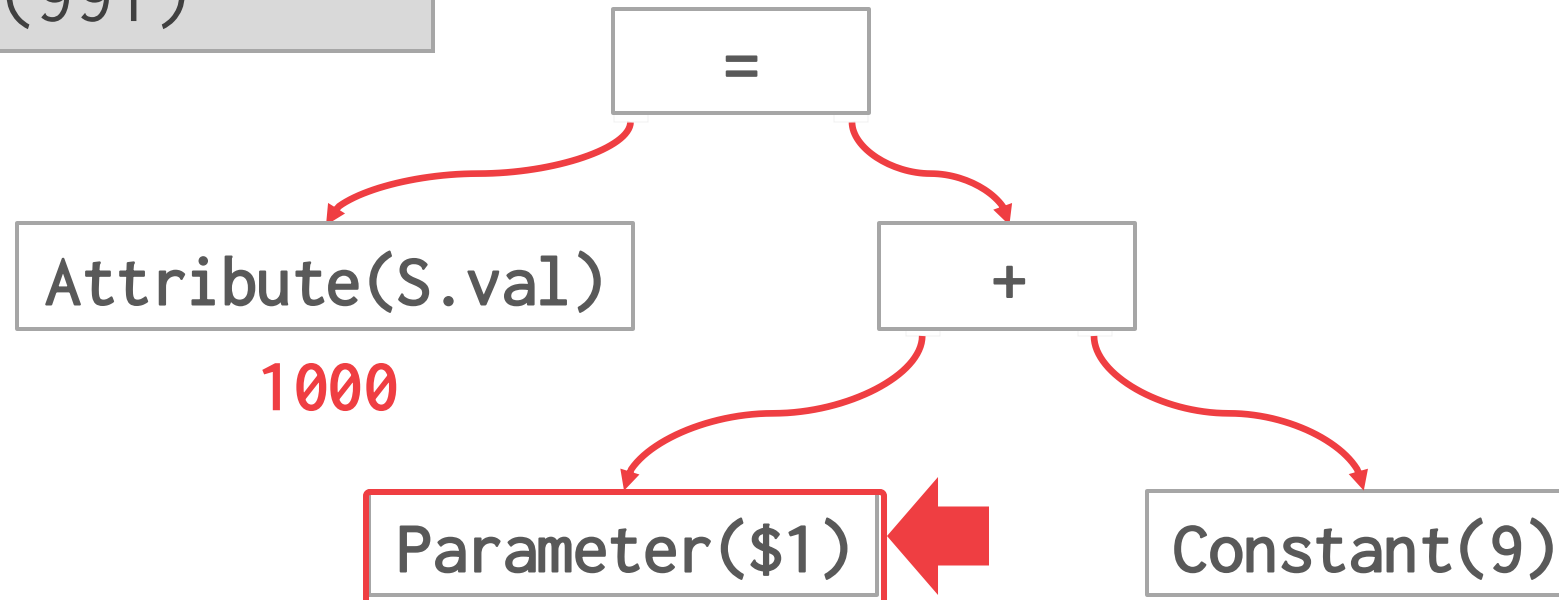
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

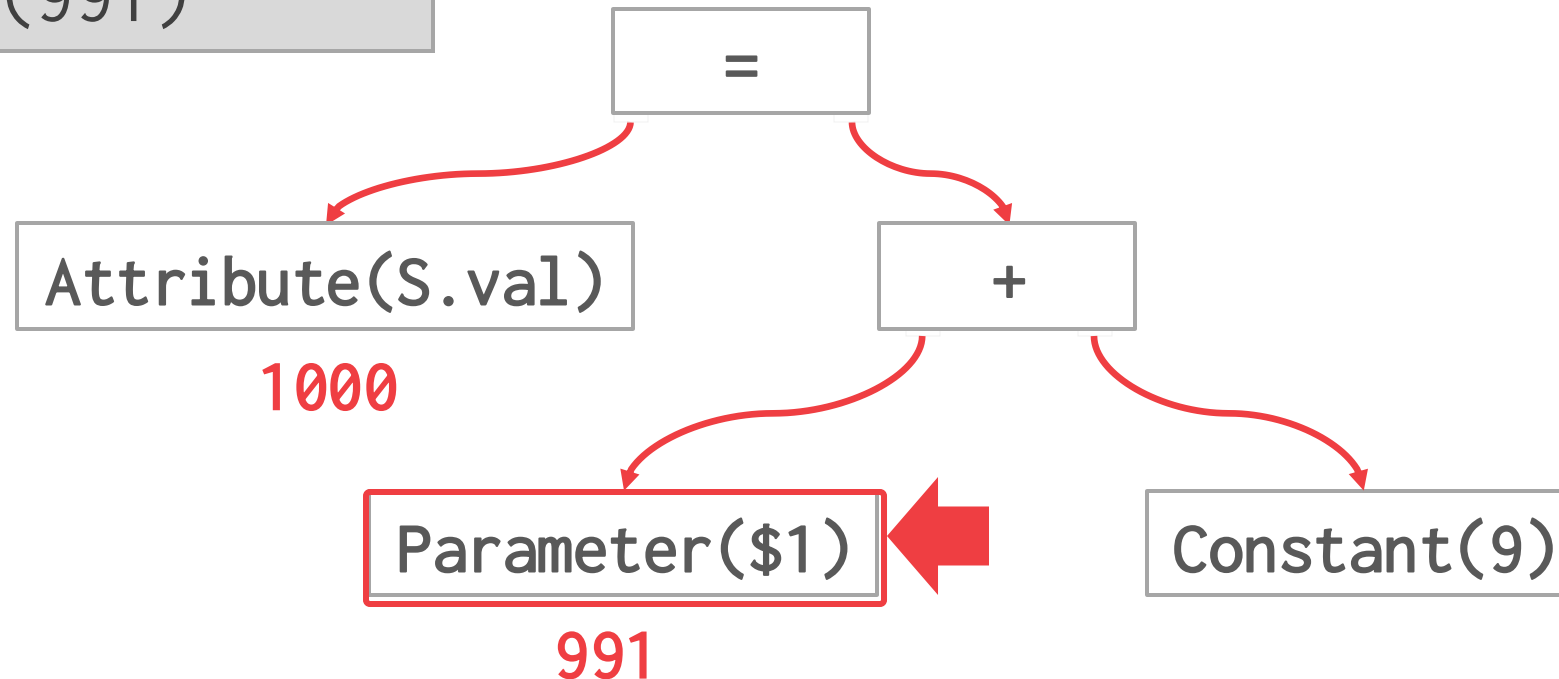
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

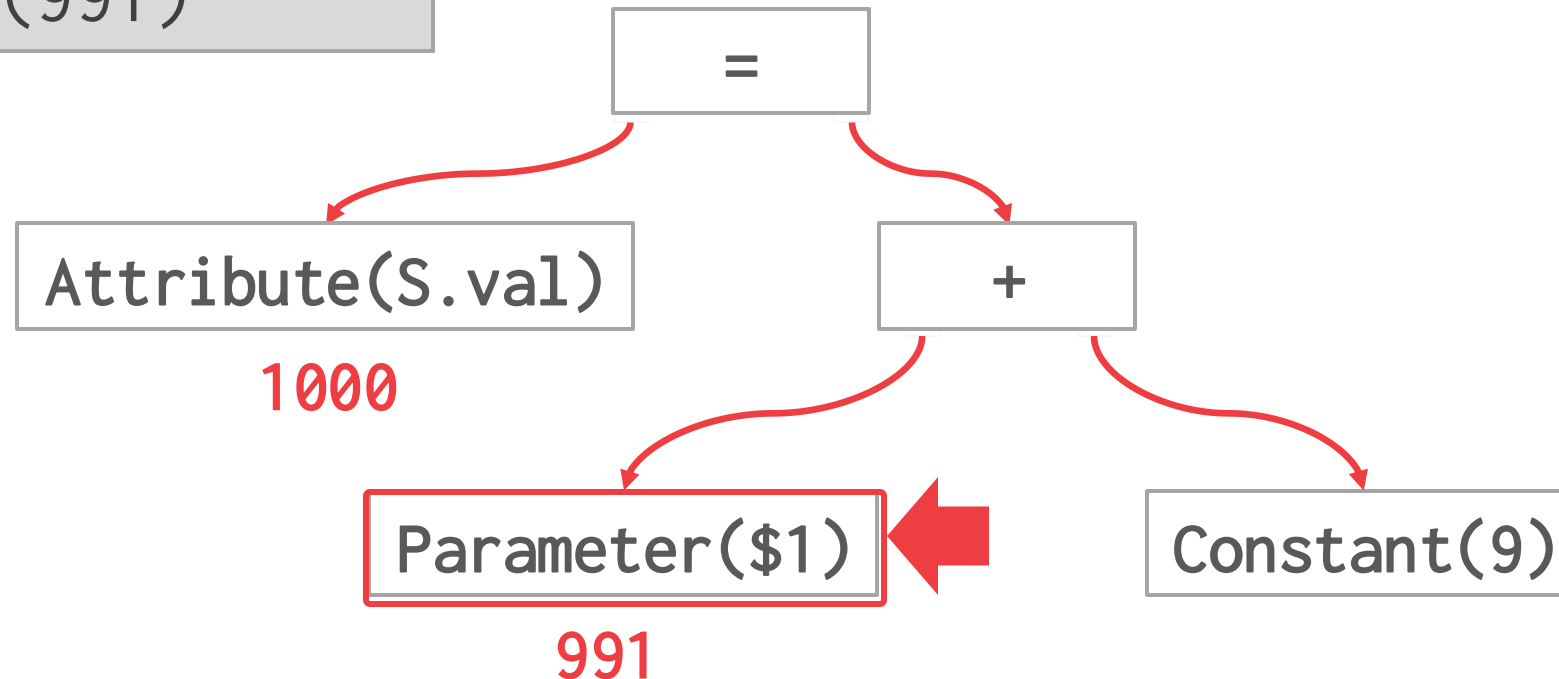
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

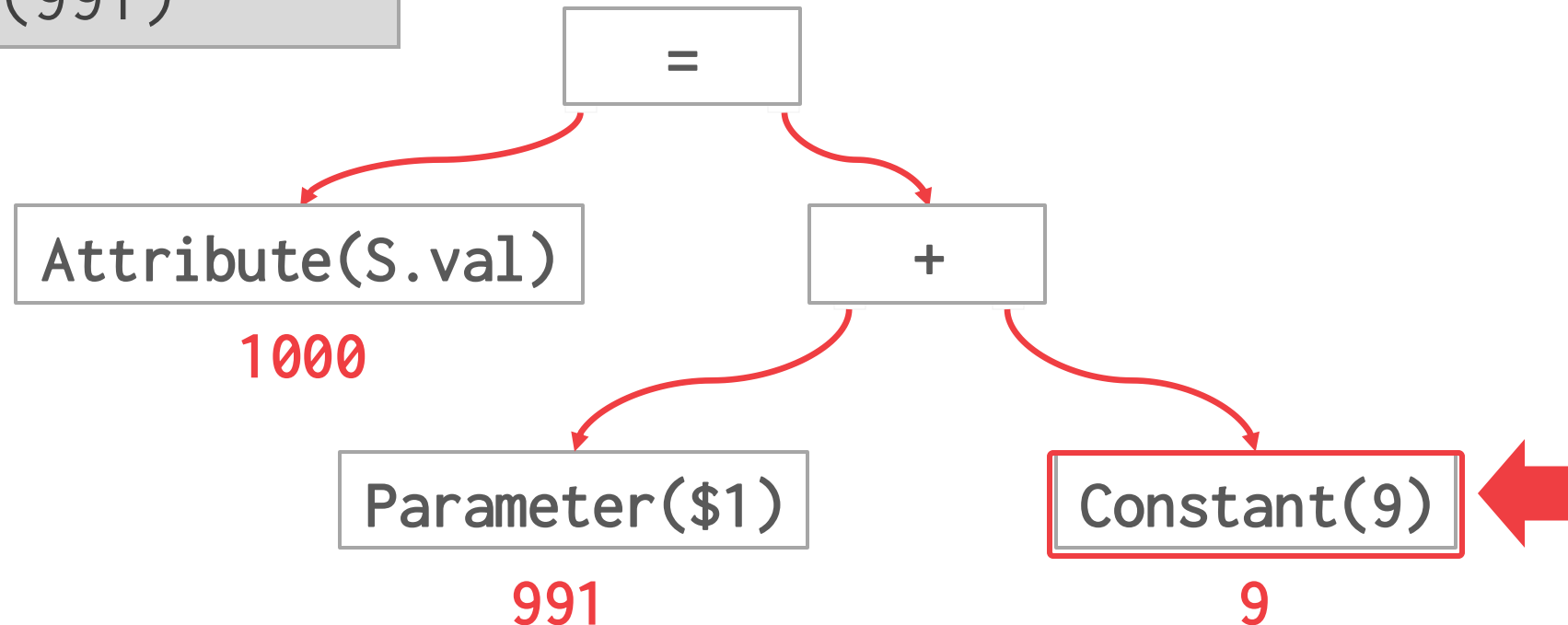
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

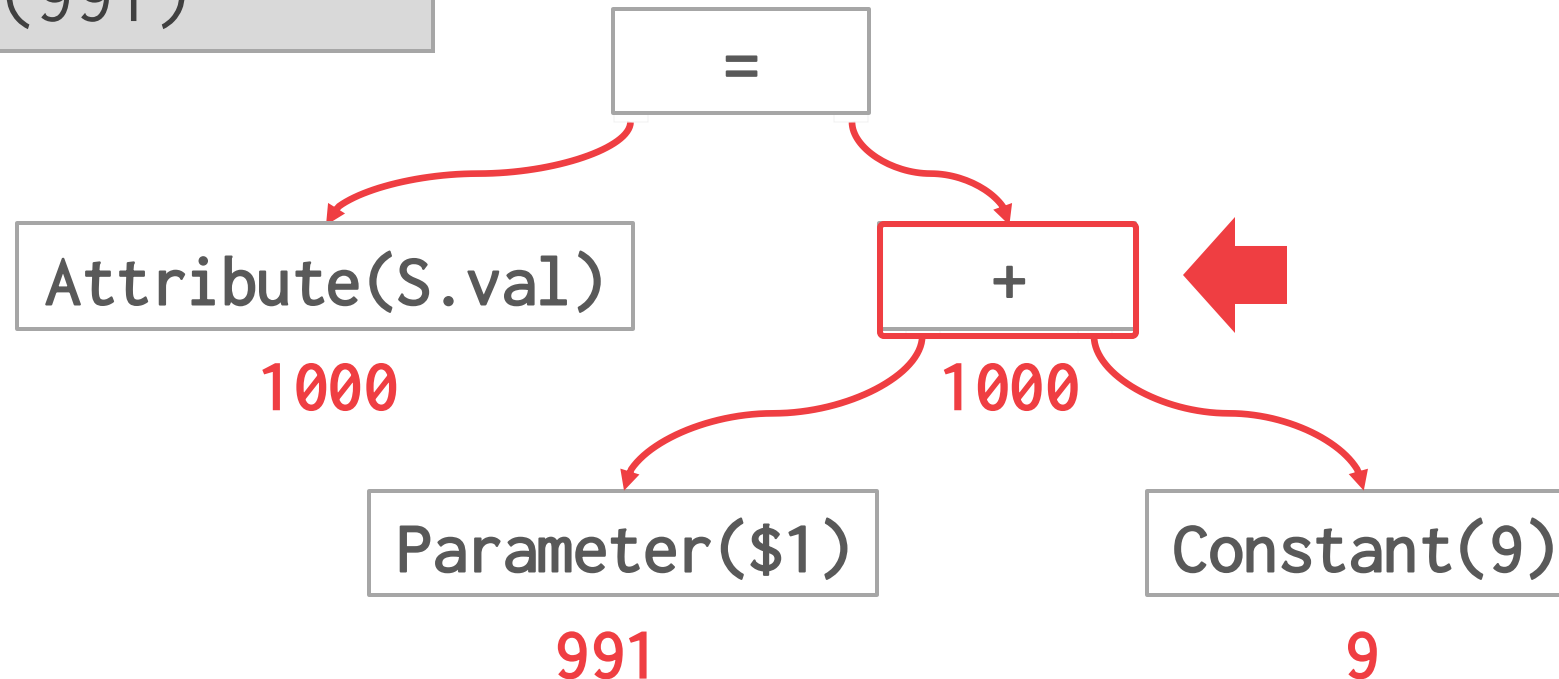
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Expression Evaluation

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

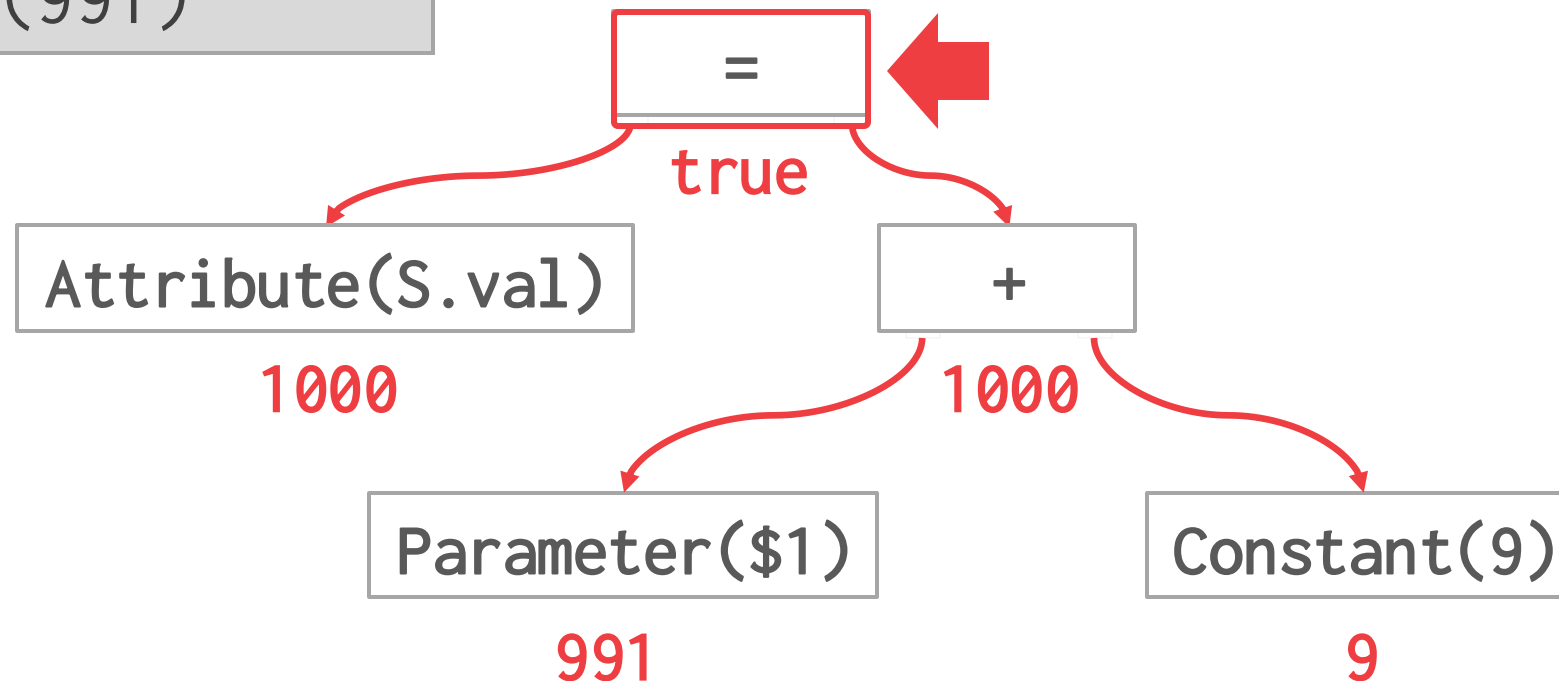
```
EXECUTE xxx(991)
```

Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→(int:id, int:val)



Conclusion

- The same query plan can be executed in multiple different ways.
- (Most) DBMSs will want to use index scans as much as possible.
- Expression trees are flexible but slow.
JIT compilation can (sometimes) speed them up.

Next Lecture

- Parallel Query Execution