12 | Query Execution – Part I

Intro to Database Systems
15-445/15-645
Fall 2019

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #3 is due Wed Oct 9th @ 11:59pm

Mid-Term Exam is Wed Oct 16th @ 12:00pm

Project #2 is due Sun Oct 20th @ 11:59pm

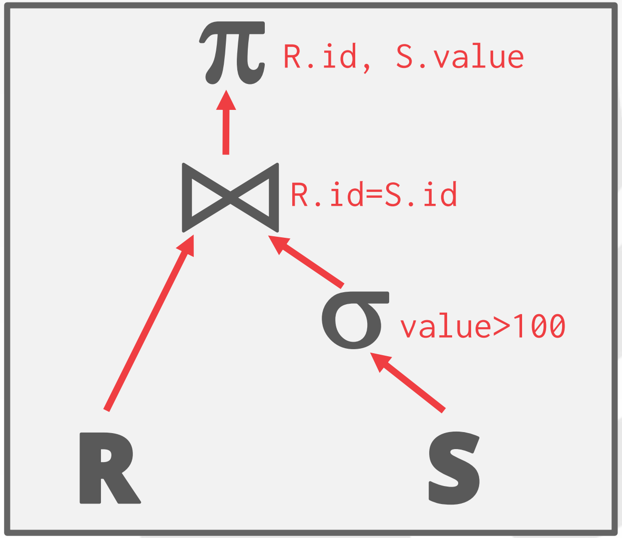
QUERY PLAN

The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.

The output of the root node is the result of the query.

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



TODAY'S AGENDA

Processing Models
Access Methods
Expression Evaluation

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 null 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.

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

ITERATOR MODEL

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

Next()

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

Next()

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

Next()

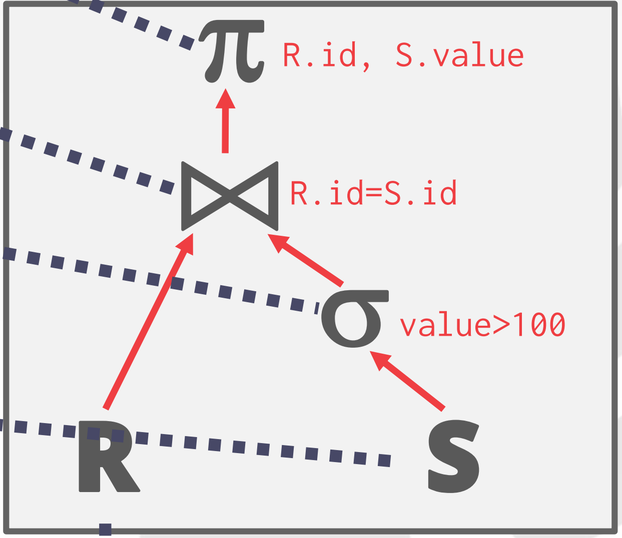
```
for t in child.Next():
  if evalPred(t): emit(t)
```

Next()

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

Next()

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



ITERATOR MODEL

1

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

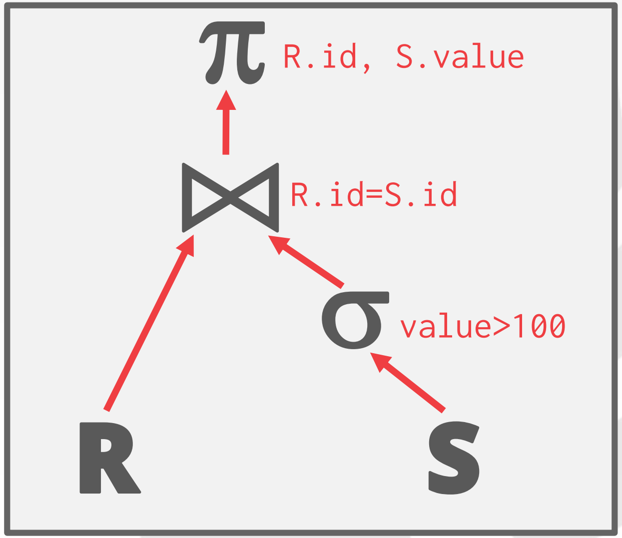
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
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

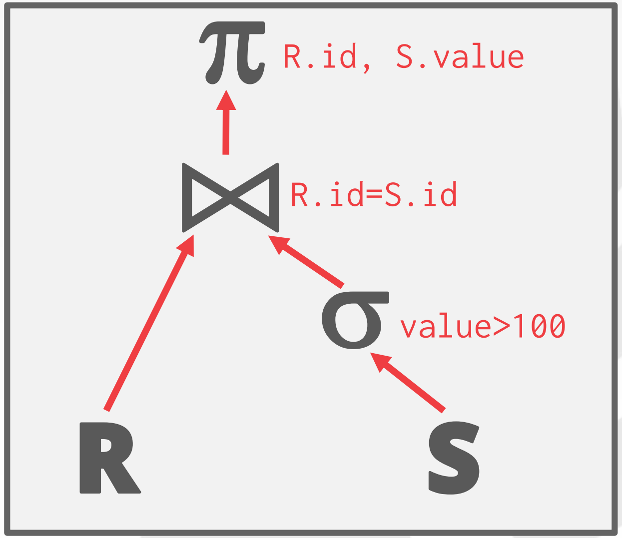
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

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

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

```sql
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)
```

Single Tuple

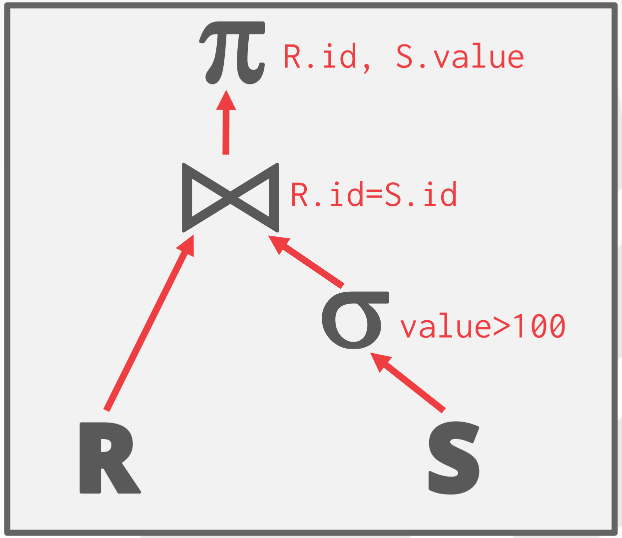
```
for t in child.Next():
  if evalPred(t): emit(t)
```

3

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

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

```sql
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))
```

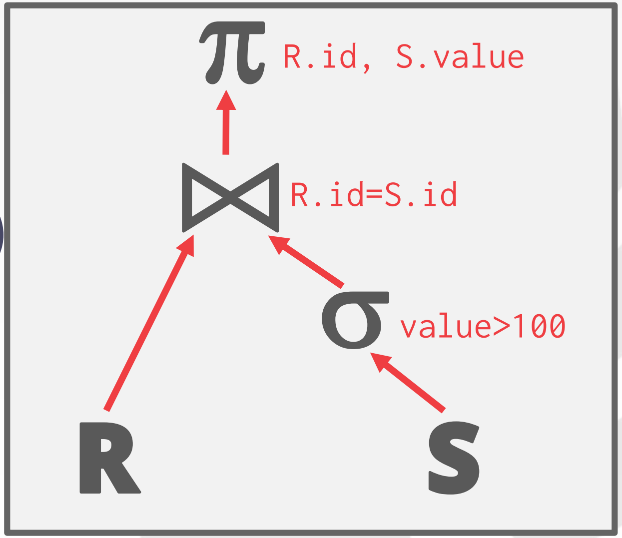
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

4
```
for t in child.Next():
  if evalPred(t): emit(t)
```

3
```
for t in R:
  emit(t)
```

5
```
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
```

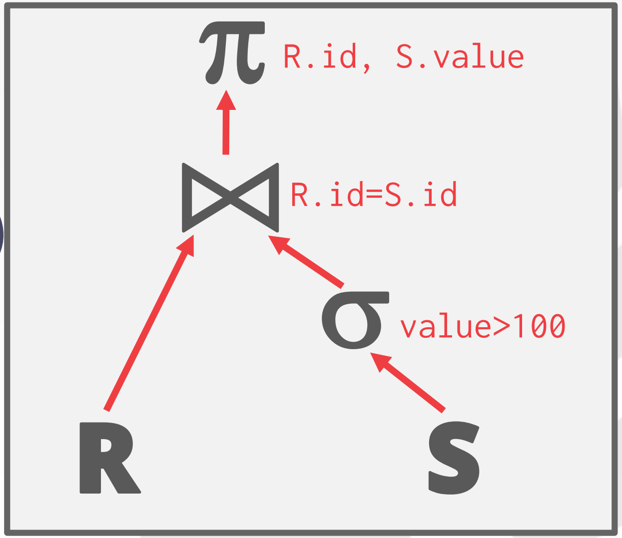
1
```
for t in child.Next():
  emit(projection(t))
```

2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

3
```
for t in R:
  emit(t)
```

4
```
for t in child.Next():
  if evalPred(t): emit(t)
```

5
```
for t in S:
  emit(t)
```



ITERATOR MODEL

This is used in almost every DBMS. Allows for tuple **pipelining**.

Some operators have to block until their children emit all of their tuples.
→ Joins, 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 into 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

1

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

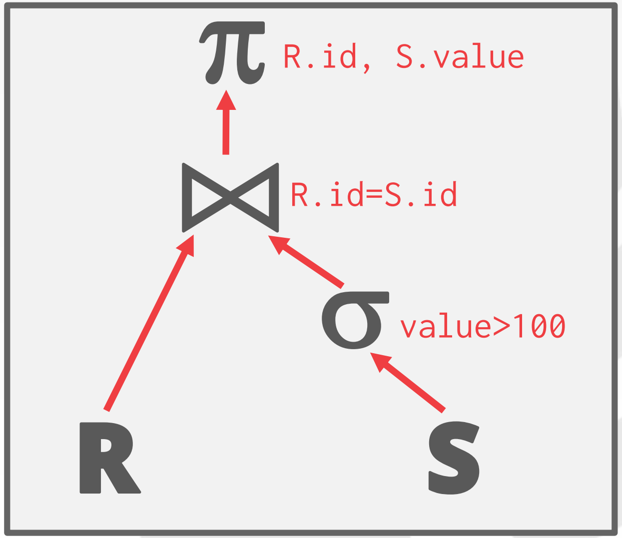
```
out = [ ]
for t₁ in left.Output():
  buildHashTable(t₁)
for t₂ in right.Output():
  if probe(t₂): out.add(t₁⋈t₂)
return out
```

```
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
```

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

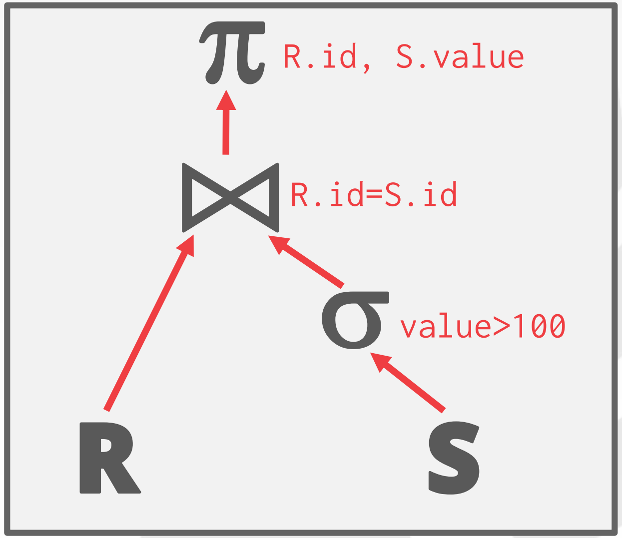
All Tuples

3

```
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
```

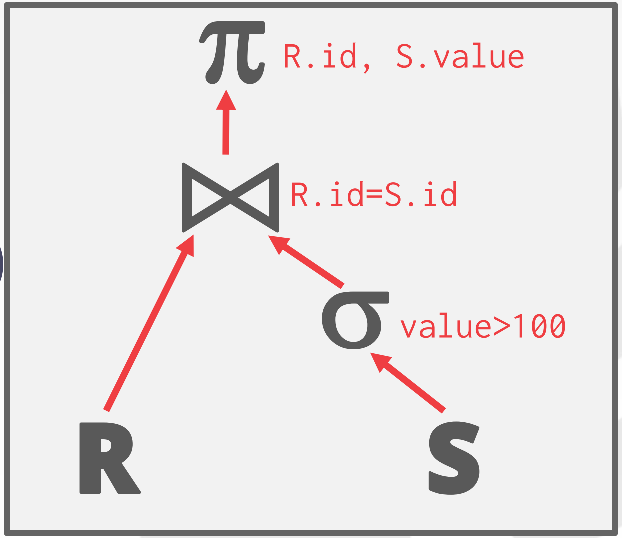
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

3
```
out = [ ]
for t in R:
  out.add(t)
return out
```

4
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

5
```
out = [ ]
for t in S:
  out.add(t)
return out
```

```sql
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
```

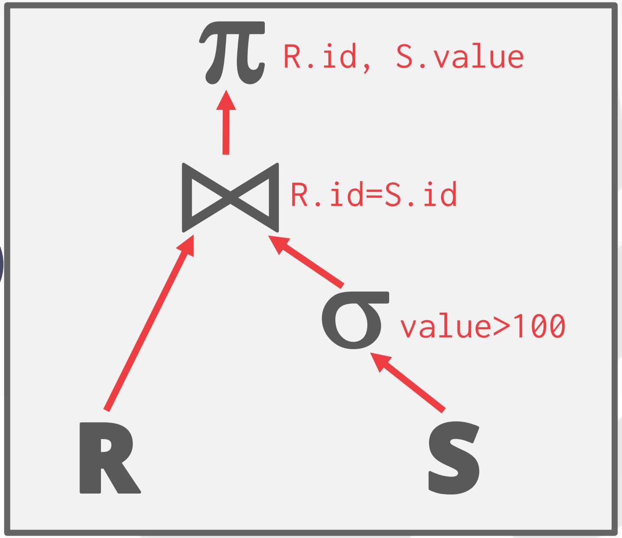
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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
```

5
```
out = [ ]
for t in S:
  out.add(t)
return out
```

```sql
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.

VECTORIZATION MODEL

Like the Iterator Model where each operator implements a **Next** function in this model.

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

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)
```

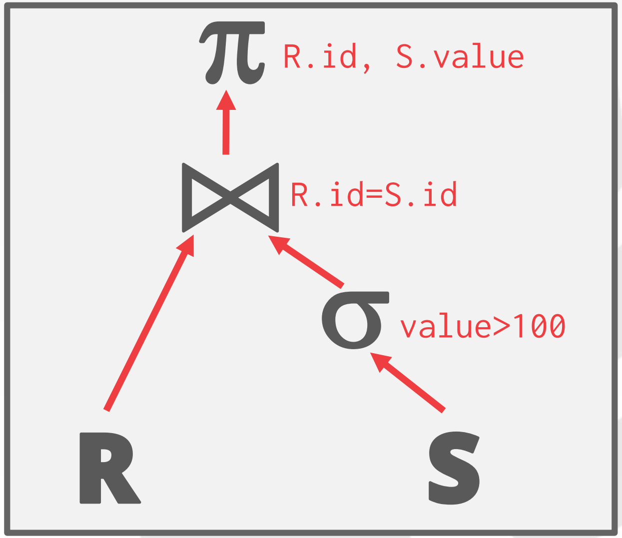
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

Tuple Batch

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

```sql
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)
```

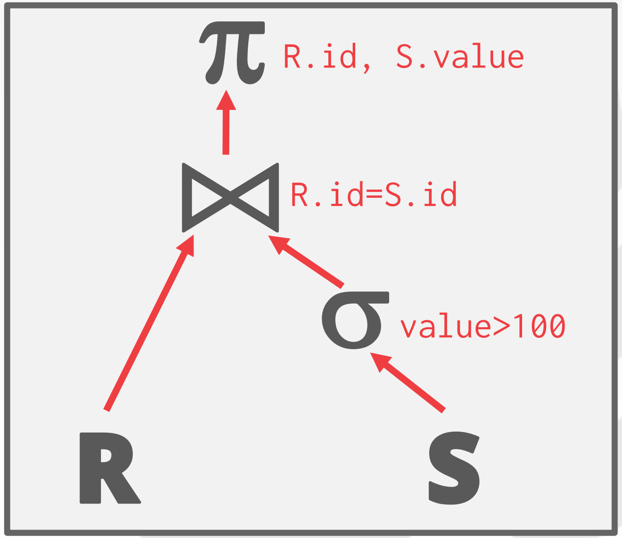
4
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

Tuple Batch

5
```
out = [ ]
for t in S:
  out.add(t)
  if |out|>n: emit(out)
```

```sql
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 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

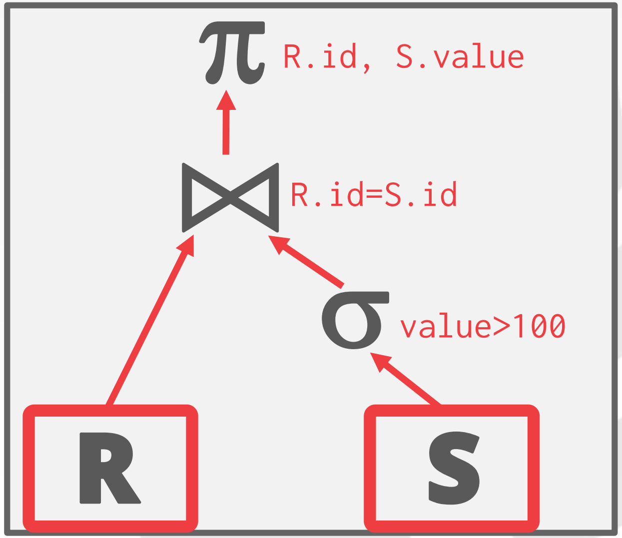
→ Start with leaf nodes and push data to their parents.
→ Allows for tighter control of caches/registers in pipelines.

ACCESS METHODS

An **access method** is a way that the DBMS can access the data stored in a table.
→ Not defined in relational algebra.

Three basic approaches:
→ Sequential Scan
→ Index Scan
→ Multi-Index / "Bitmap" 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.

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

The DBMS maintains an internal **cursor** that tracks the last page / slot it examined.

SEQUENTIAL SCAN: OPTIMIZATIONS

This is almost always the worst thing that the DBMS can do to execute a query.

Sequential Scan Optimizations:
→ Prefetching
→ Buffer Pool Bypass
→ Parallelization
→ Zone Maps
→ Late Materialization
→ Heap Clustering

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

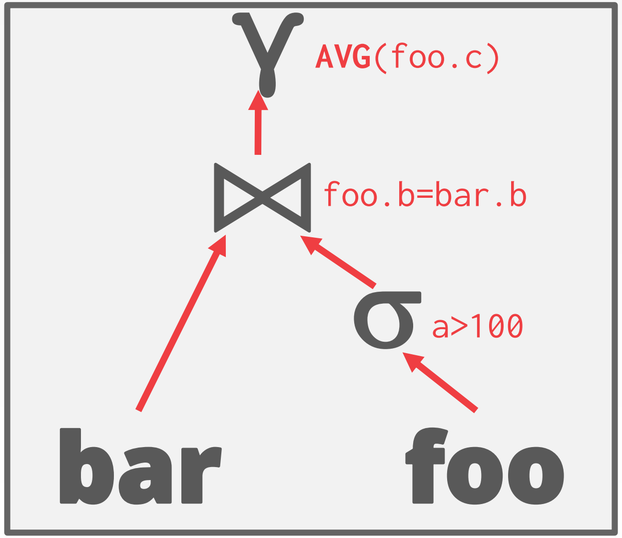
val
100
200
300
400
400

Zone Map

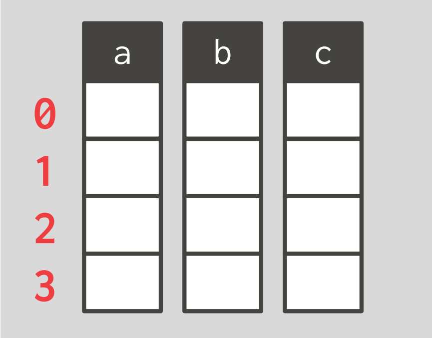
type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

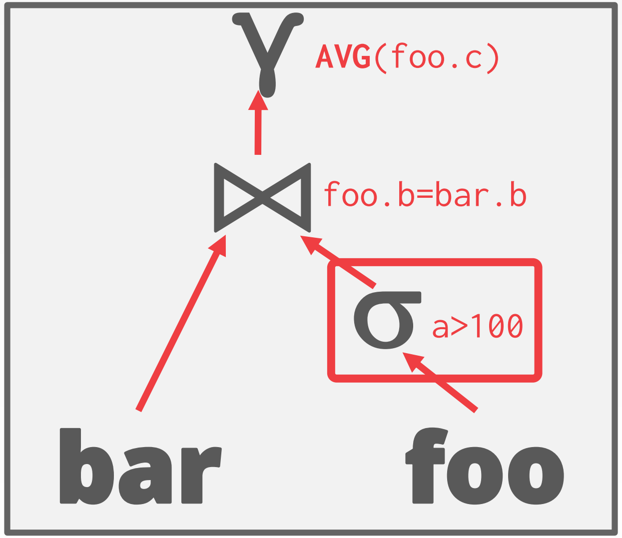



```
SELECT AVG(foo.c)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE foo.a > 100
```



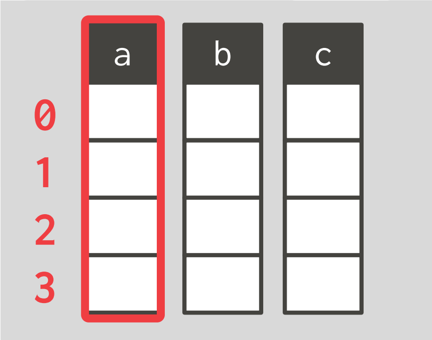
LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.



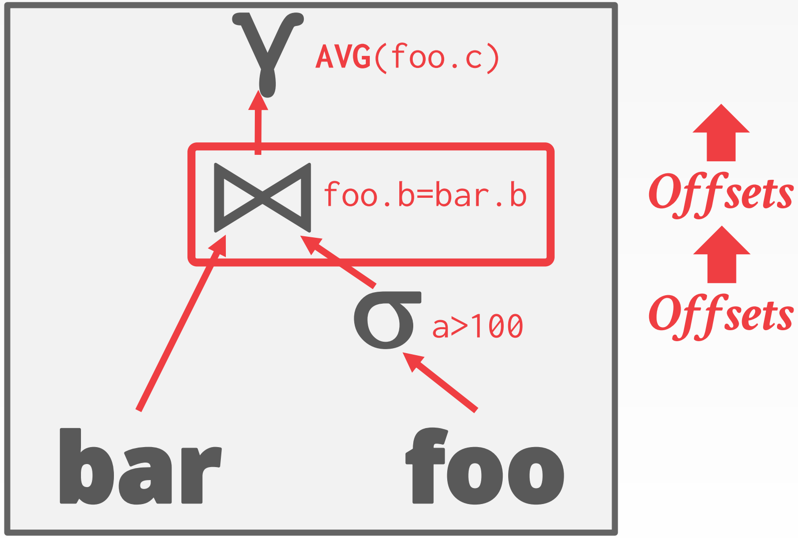
Offsets

```
SELECT AVG(foo.c)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE foo.a > 100
```

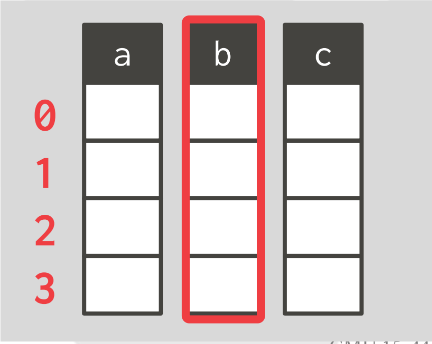


LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

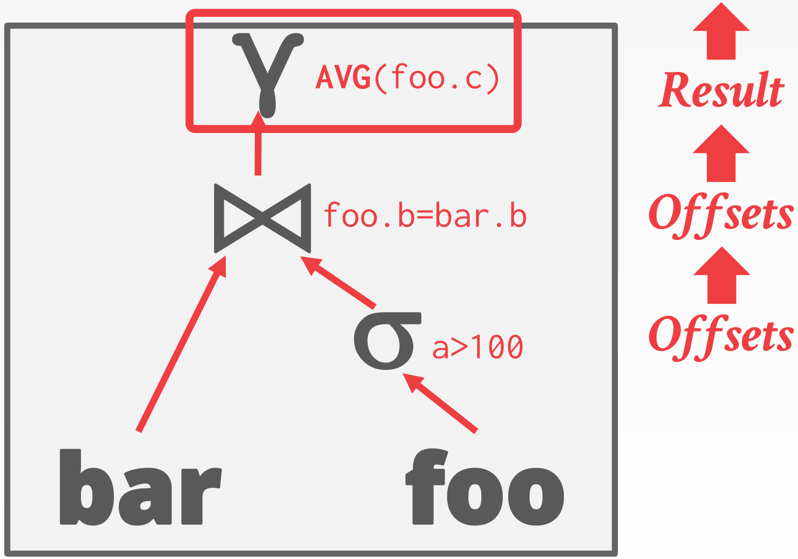



```
SELECT AVG(foo.c)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE foo.a > 100
```

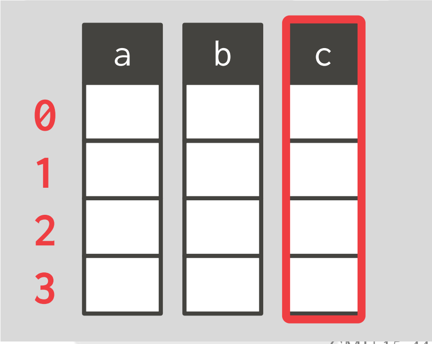


LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.



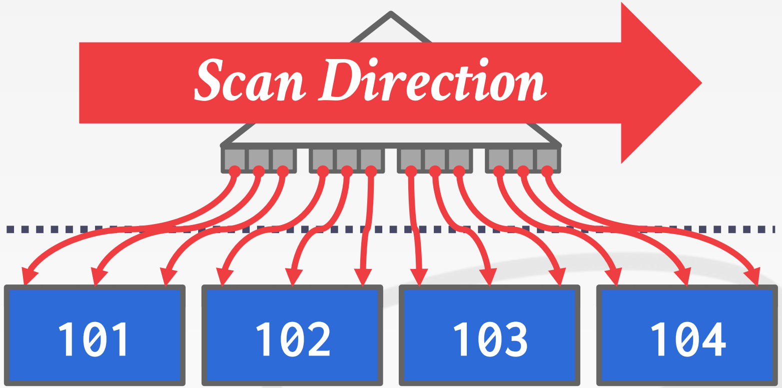

```
SELECT AVG(foo.c)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE foo.a > 100
```



HEAP CLUSTERING

Tuples are sorted in the heap's pages using the order specified by a clustering index.

If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.



INDEX SCAN

The DBMS picks an index to find the tuples that the query needs.

Lecture 14

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

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

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

→ Index #1: **age**
→ Index #2: **dept**

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.

Postgres calls this Bitmap Scan.

MULTI-INDEX SCAN

With an index on **`age`** and an index on **`dept`**,

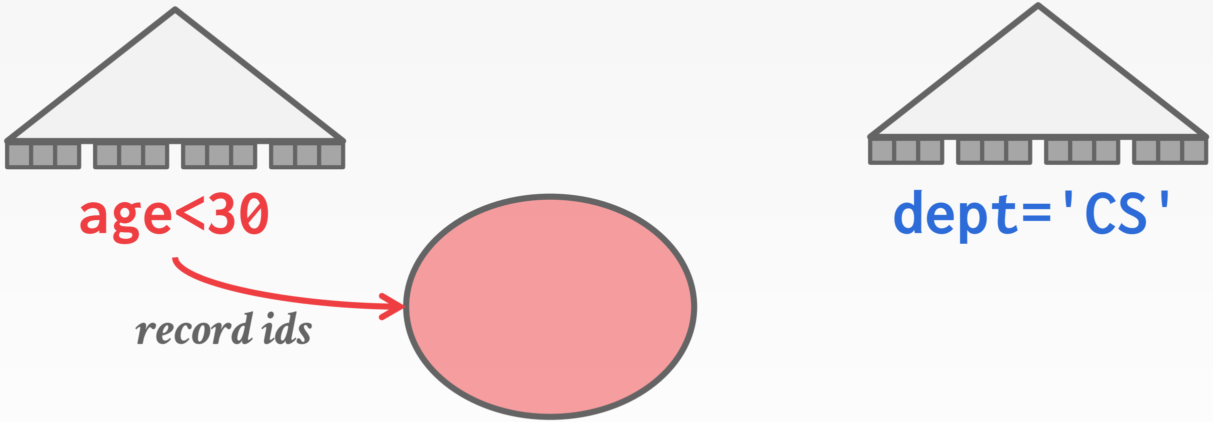
- → We can retrieve the record ids satisfying **`age<30`** using the first,
- → Then retrieve the record ids satisfying **`dept='CS'`** using the second,
- → Take their intersection
- → Retrieve records and check **`country='US'`**.

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

MULTI-INDEX SCAN

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

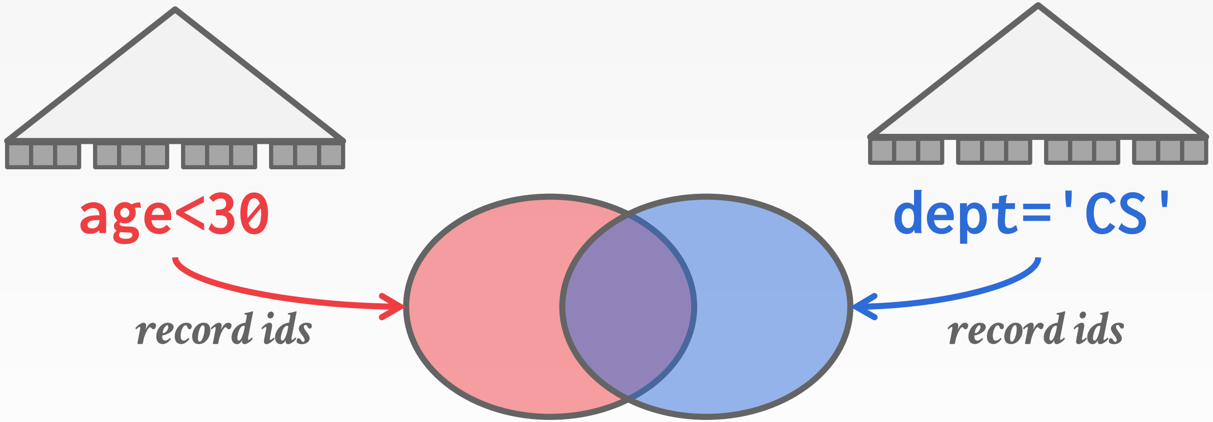
Set intersection can be done with bitmaps, hash tables, or Bloom filters.



MULTI-INDEX SCAN

Set intersection can be done with bitmaps, hash tables, or Bloom filters.

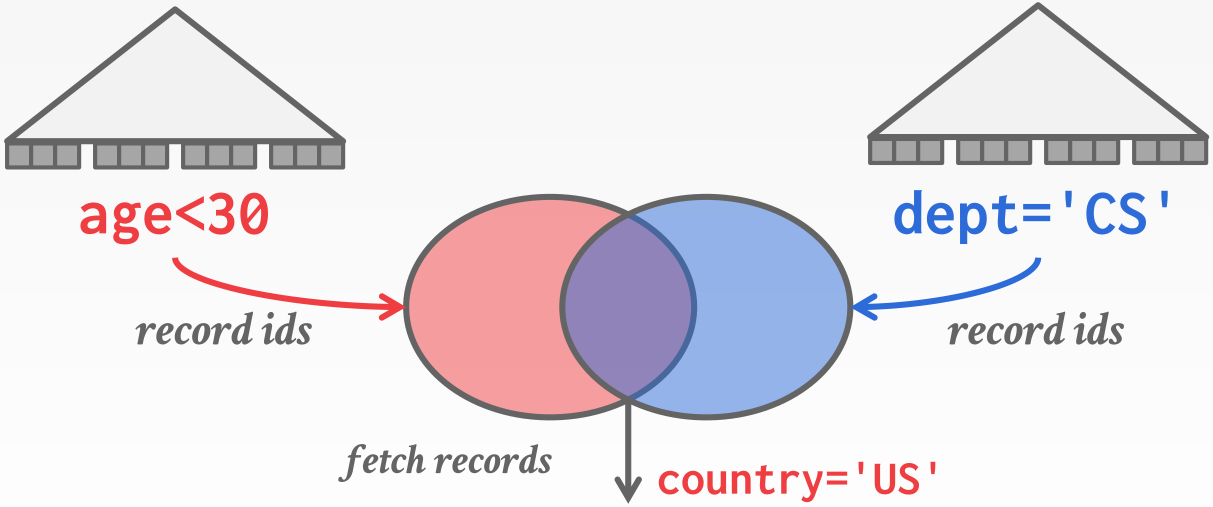
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



MULTI-INDEX SCAN

Set intersection can be done with bitmaps, hash tables, or Bloom filters.

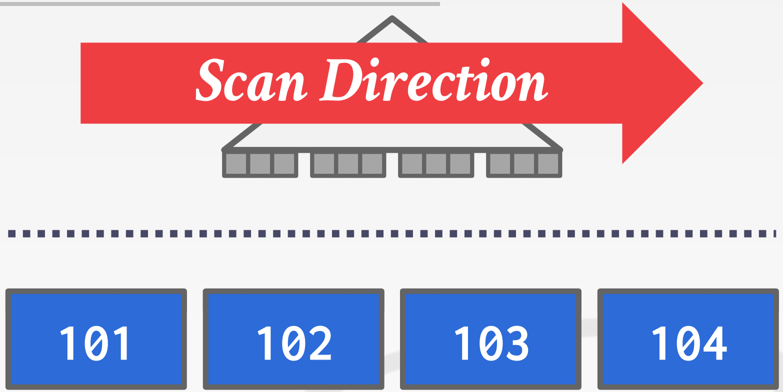
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

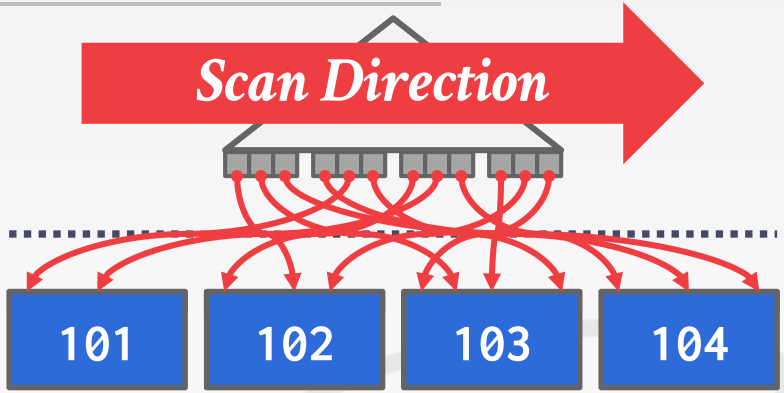
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

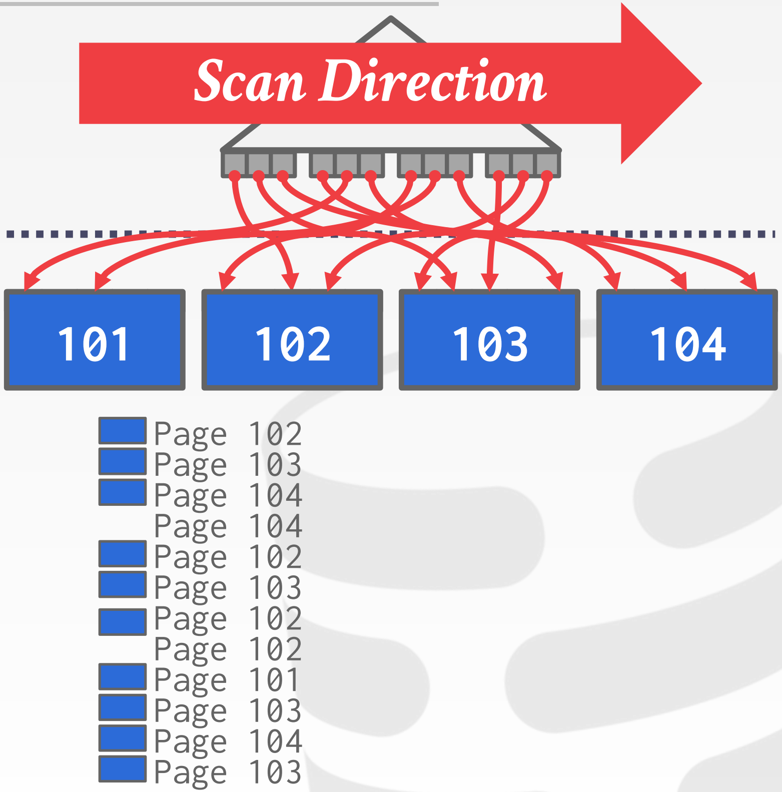
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

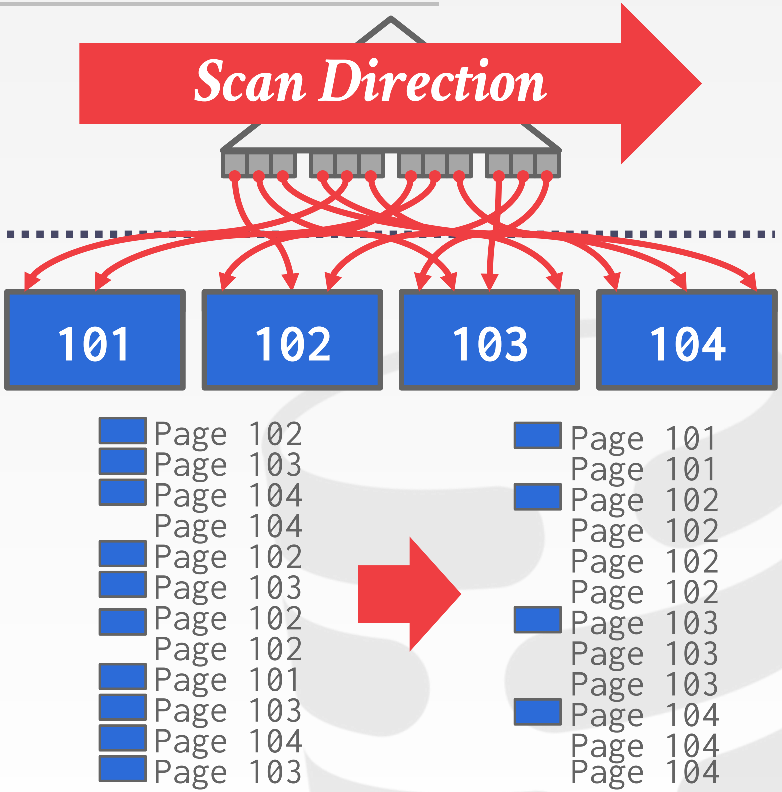
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



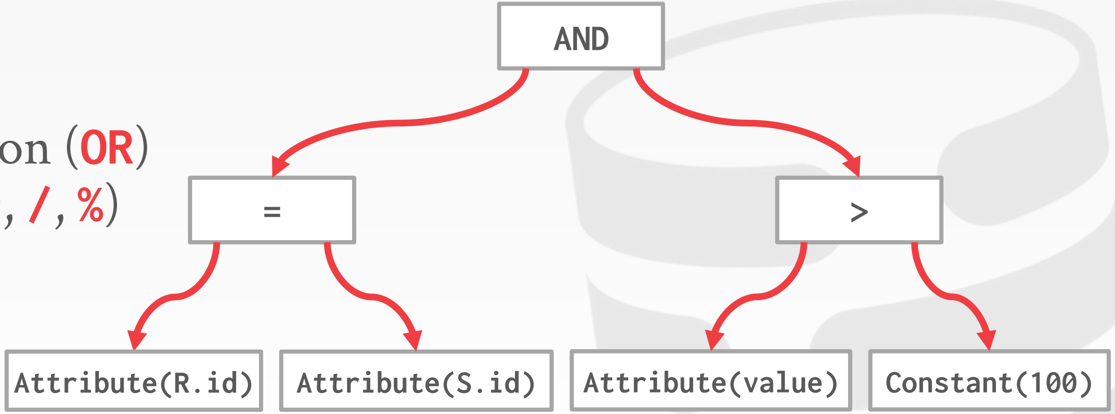
EXPRESSION EVALUATION

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

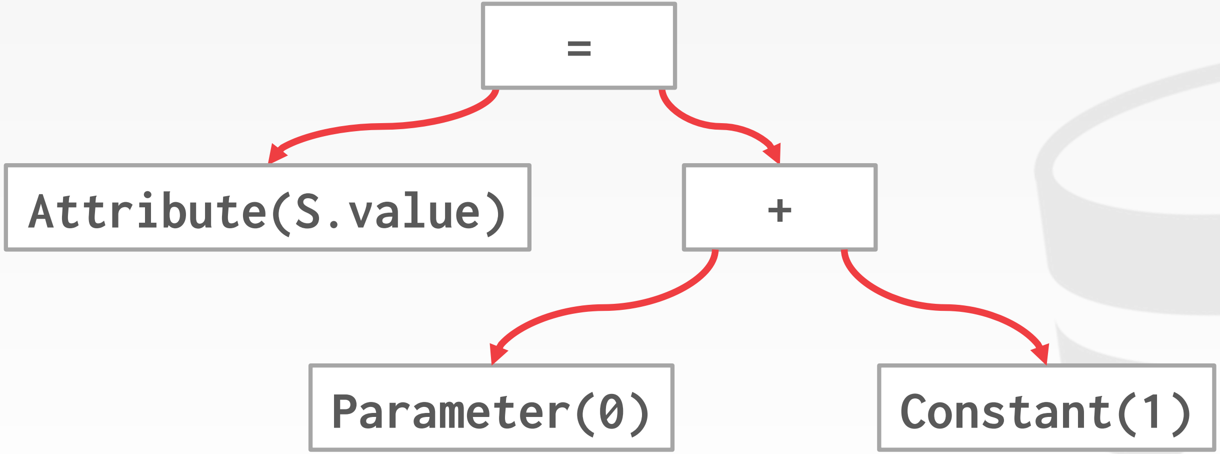
```
SELECT * FROM S
 WHERE B.value = ? + 1
```

EXPRESSION EVALUATION

```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	S→(int:id, int:value)

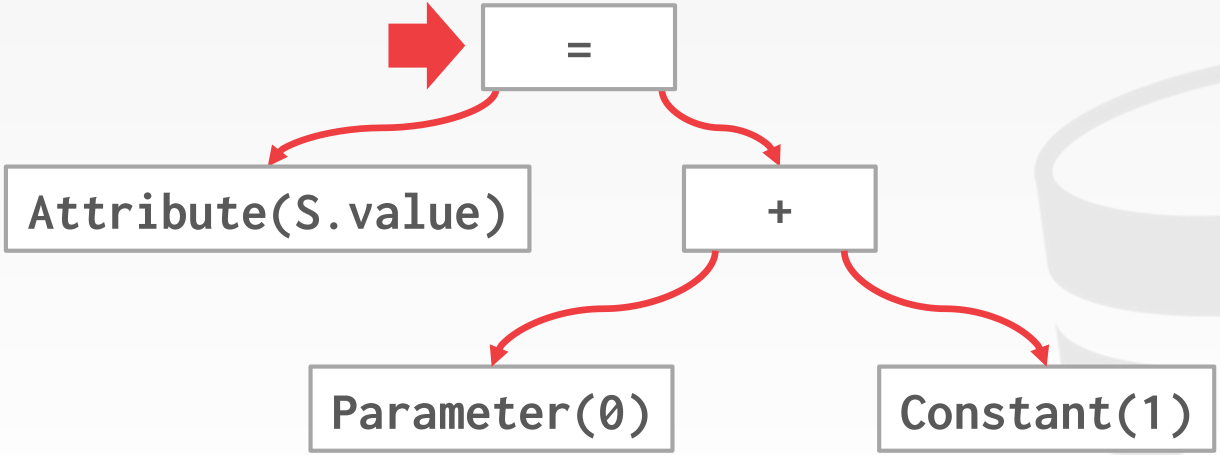


EXPRESSION EVALUATION

```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	S→(int:id, int:value)

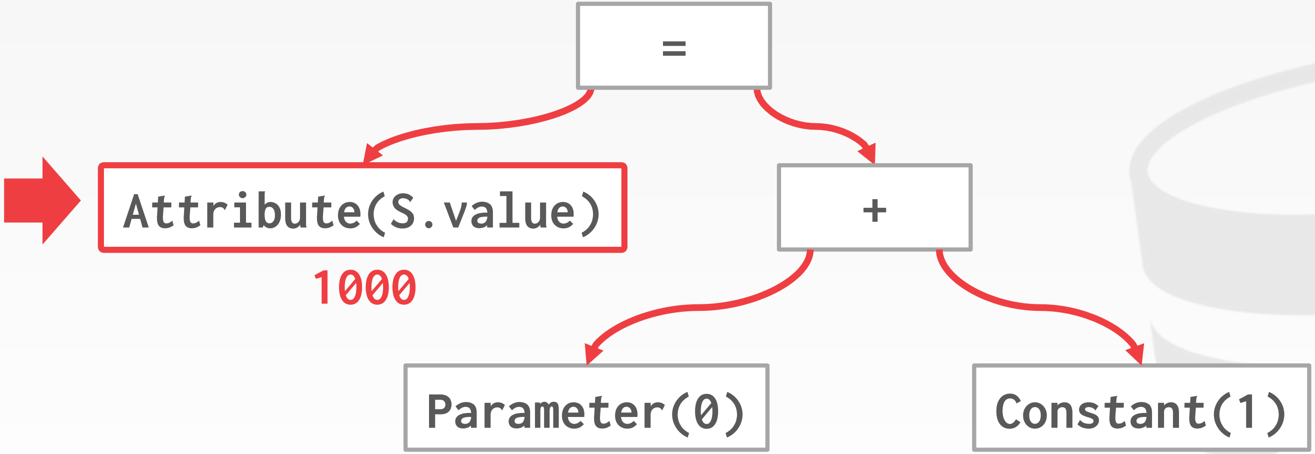


EXPRESSION EVALUATION

```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	S→(int:id, int:value)

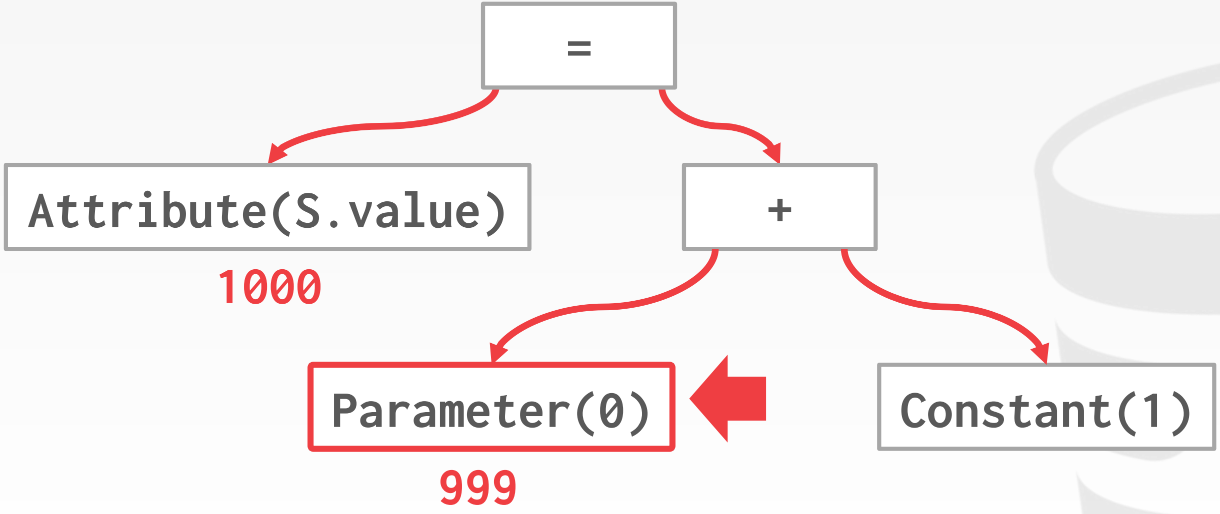


EXPRESSION EVALUATION

```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	S→(int:id, int:value)

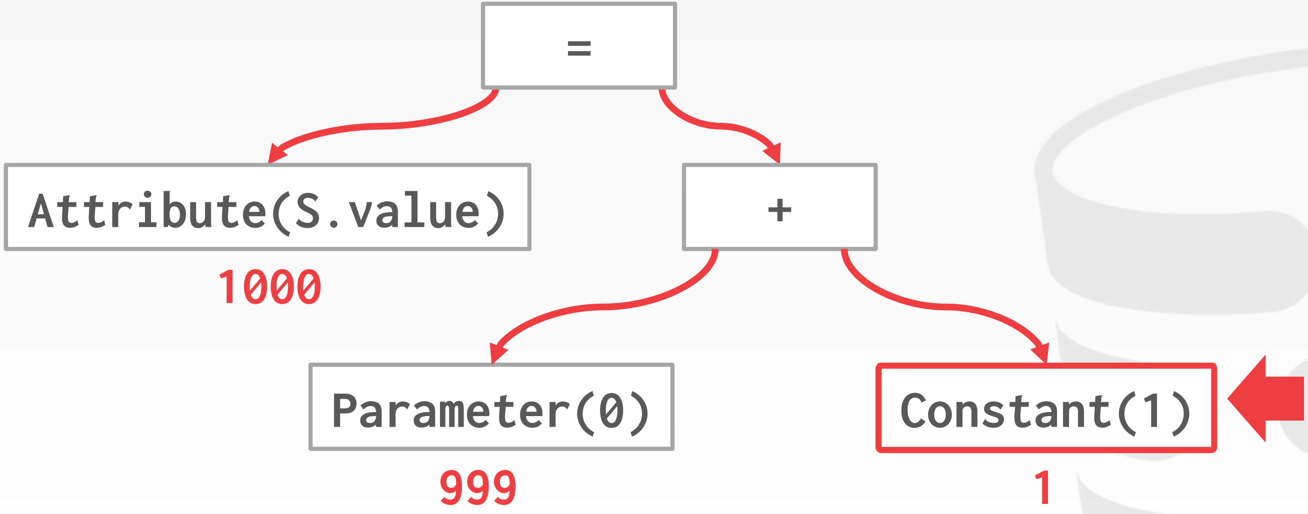


EXPRESSION EVALUATION

```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	S→(int:id, int:value)

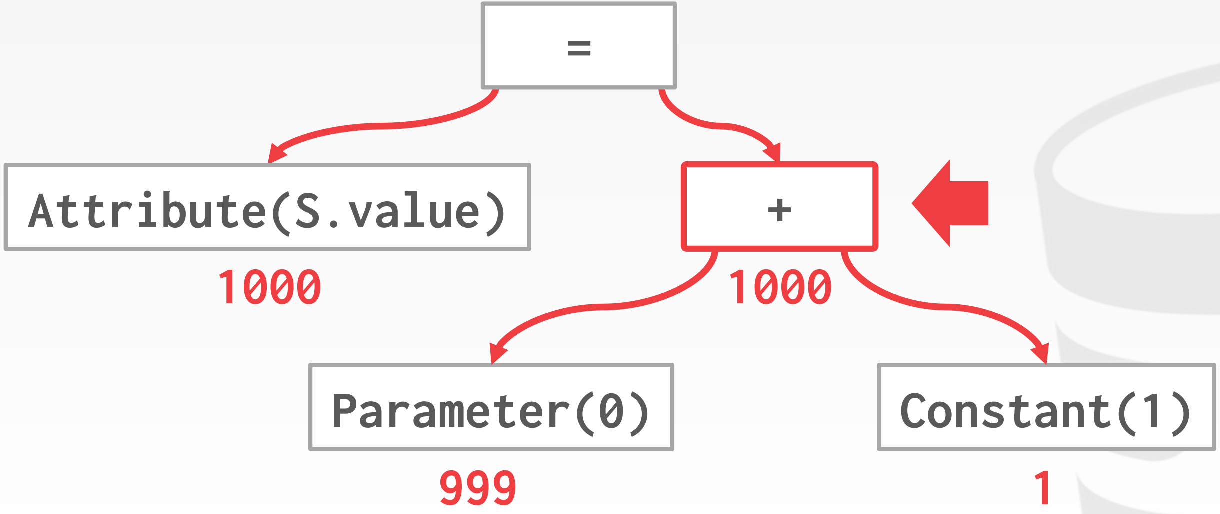


EXPRESSION EVALUATION

```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	S→(int:id, int:value)

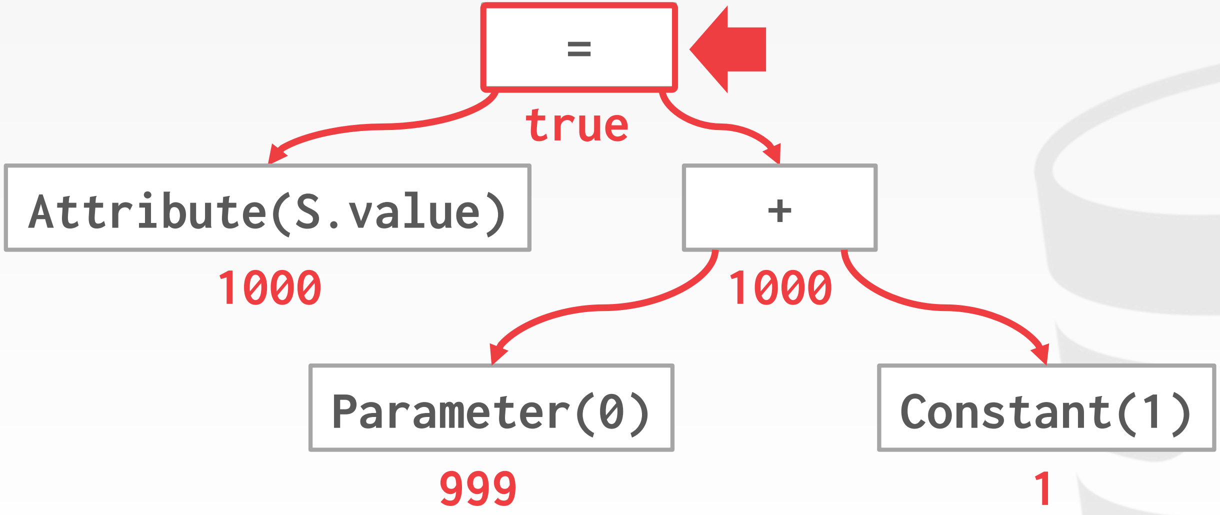


EXPRESSION EVALUATION

```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	S→(int:id, int:value)

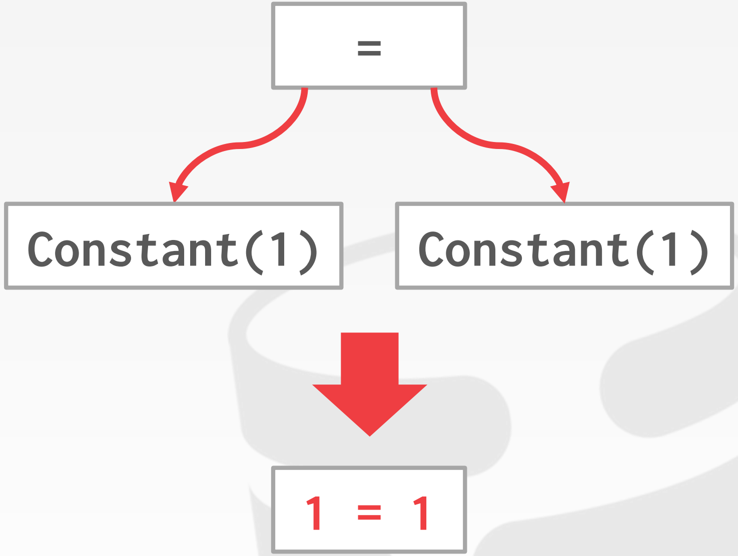


EXPRESSION EVALUATION

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 the predicate "WHERE 1=1"

A better approach is to just evaluate the expression directly.
→ Think JIT compilation



CONCLUSION

The same query plan be executed in multiple ways.

(Most) DBMSs will want to use an index scan as much as possible.

Expression trees are flexible but slow.

NEXT CLASS

Parallel Query Execution