Carnegie Mellon University

Query Processing

Lecture #10

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #2 – Checkpoint #1 is due Monday October 9th @ 11:59pm

Mid-term Exam is on Wednesday October 17th (in class)

UPCOMING DATABASE EVENTS

SQream DB Tech Talk

→ Thursday Oct 4th @ 12:00pm

→ CIC 4th Floor

treated as a tree

QUERY PLAN

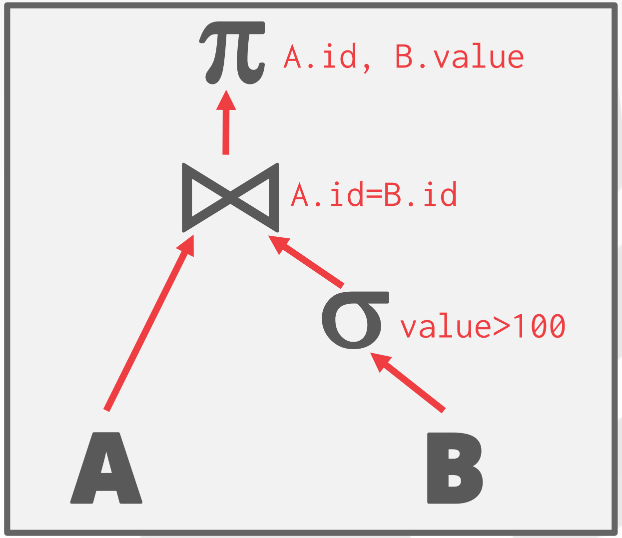
The operators are arranged in a tree. Data flows from the leaves toward the root.

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

given a sql, convert it into query plan comprised of relational operators

equivalent as relational algebra

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.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.

Three approaches:
→ Iterator Model
→ Materialization Model OLTP query
→ Vectorized / Batch Model OLAP query

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.

Top-down plan processing.

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

ITERATOR MODEL

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

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

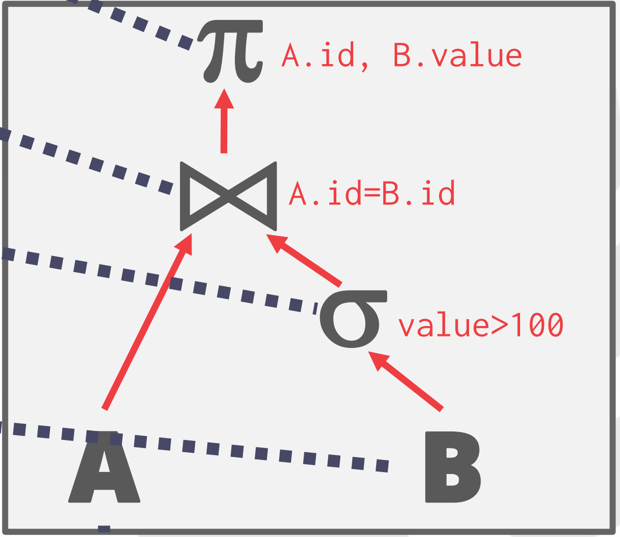
```
for t₁ in left.Next():
  buildHashTable(t₁)
for t₂ in right.Next():
  if probe(t₂): emit(t₁⋈t₂)
```

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

can be calculated in parallel meaning try to work on a single tuple for as long as you can before it is swapped out from memory which in practice is to keep popping up from leave to root as far as possible (usually stopped when meet a join operator)

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

```
for t in B:
  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)
```

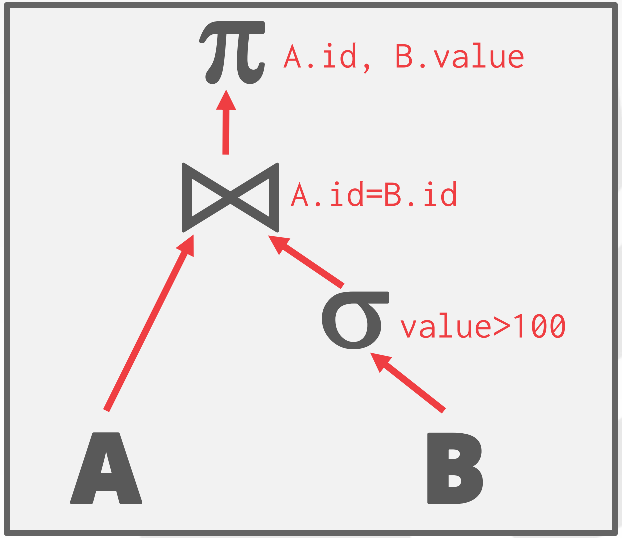
need keep invoking next(),
but is useful when sql has LIMIT

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

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

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

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.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)
```

pipeline breaker: join

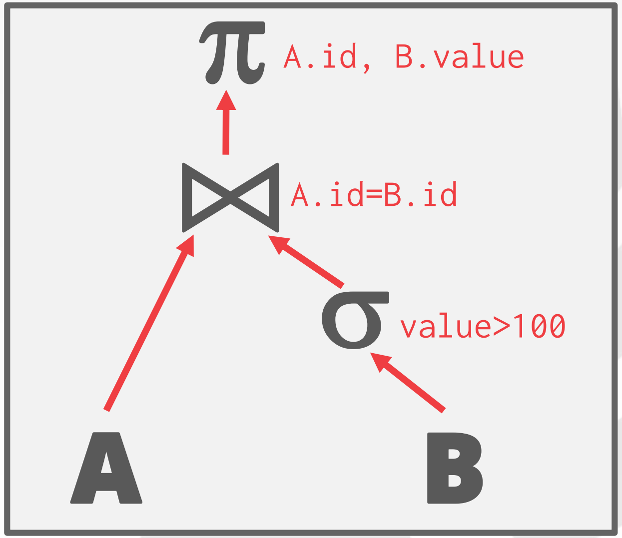
```
for t in child.Next():
  if evalPred(t): emit(t)
```

3

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

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

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL

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

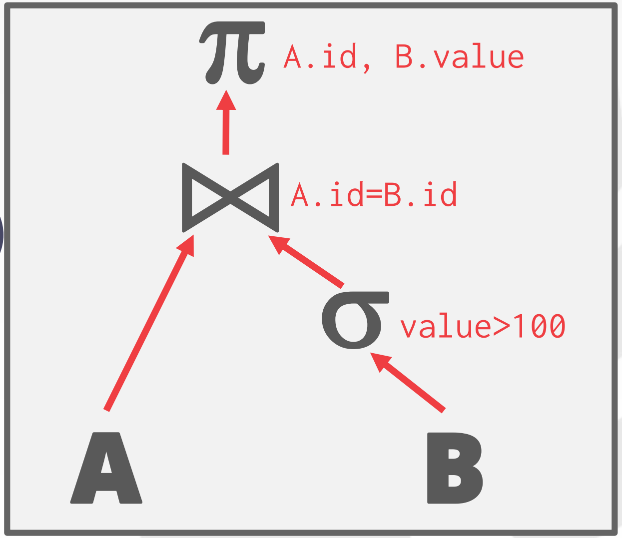
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

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

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

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL

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

Some operators will block until children emit all of their tuples.
→ Joins, Subqueries, Order By

Output control works easily with this approach.
→ Limit

SQLite
cloudera IMPALA
IBM DB2
NUODB
ORACLE
MySQL
Microsoft SQL Server
PostgreSQL
VERTICA
Greenplum

MATERIALIZATION MODEL

Each operator processes its input all at once and then emits its output all at once.

→ The operator "materializes" it output as a single result.
→ The DBMS can push down hints into to avoid scanning too many tuples.

Bottom-up plan processing.

MATERIALIZATION MODEL

```
out = { }
for t in child.Output():
  out.add(projection(t))
```

```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
```

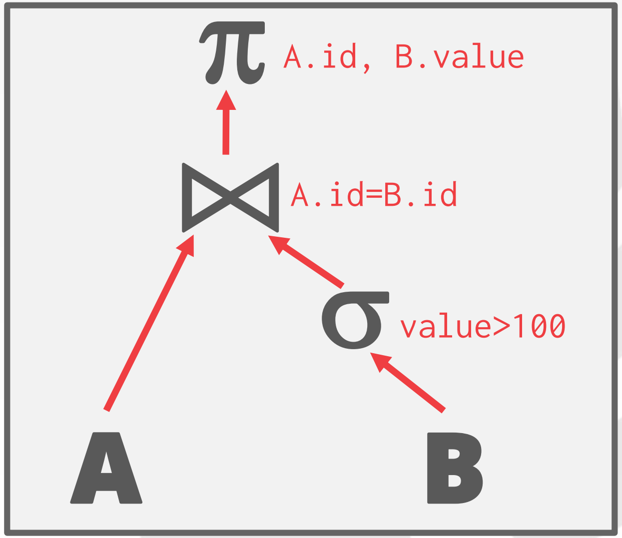
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
```

1

```
out = { }
for t in A:
  out.add(t)
```

```
out = { }
for t in B:
  out.add(t)
```

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



MATERIALIZATION MODEL

better for transaction query
to not touch too much data

```
out = { }
for t in child.Output():
  out.add(projection(t))
```

```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
```

output all tuples this operator satisfies

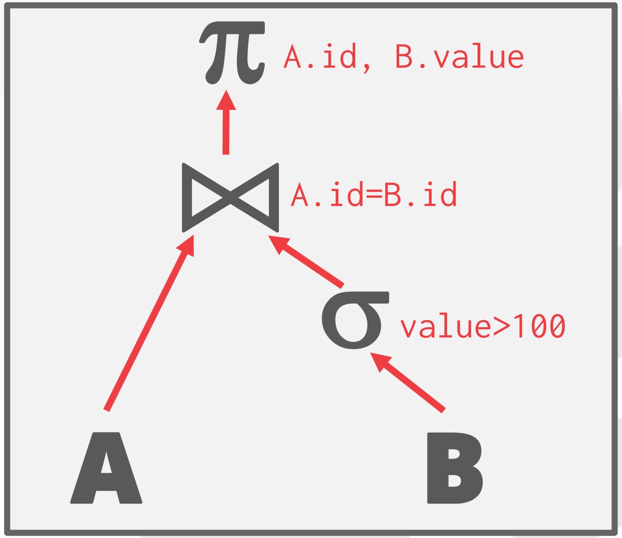
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
```

1

```
out = { }
for t in A:
  out.add(t)
```

```
out = { }
for t in B:
  out.add(t)
```

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



MATERIALIZATION MODEL

```
out = { }
for t in child.Output():
  out.add(projection(t))
```

```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
```

3

```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
```

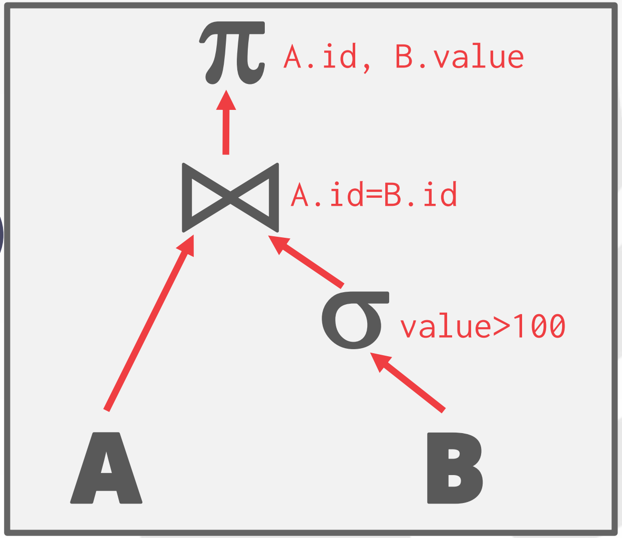
1

```
out = { }
for t in A:
  out.add(t)
```

2

```
out = { }
for t in B:
  out.add(t)
```

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



MATERIALIZATION MODEL

5
```
out = { }
for t in child.Output():
  out.add(projection(t))
```

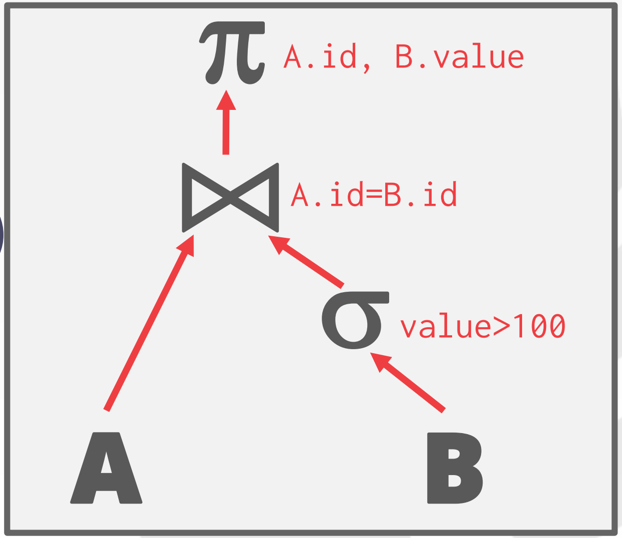
4
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
```

3
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
```

1
```
out = { }
for t in A:
  out.add(t)
```

2
```
out = { }
for t in B:
  out.add(t)
```

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



MATERIALIZATION MODEL

Better for OLTP workloads because queries typically only access a small number of tuples at a time.

→ Lower execution / coordination overhead.

Not good for OLAP queries with large intermediate results.

Materialization model is like bottom-up

VECTORIZATION MODEL

Like Iterator Model, each operator implements a **next** function.

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.Output():
  out.add(projection(t))
  if |out|>n: emit(out)
```

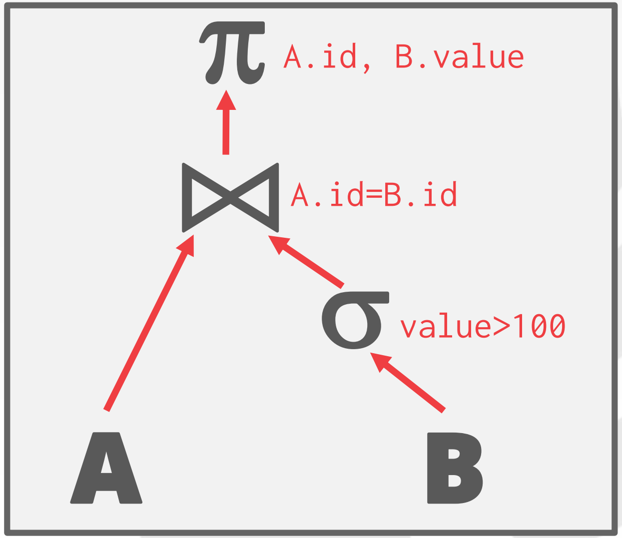
2
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

3
```
out = { }
for t in A:
  out.add(t)
  if |out|>n: emit(out)
```

```
out = { }
for t in B:
  out.add(t)
  if |out|>n: emit(out)
```

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



VECTORIZATION MODEL

1
```
out = { }
for t in child.Output():
  out.add(projection(t))
  if |out|>n: emit(out)
```

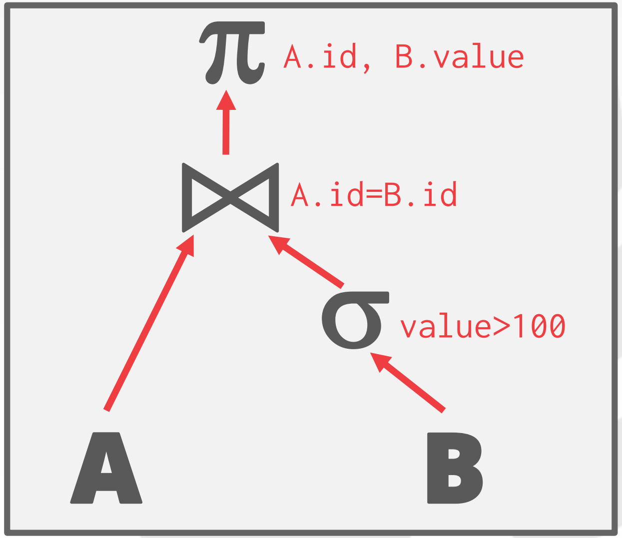
2
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

4
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

3
```
out = { }
for t in A:
  out.add(t)
  if |out|>n: emit(out)
```

5
```
out = { }
for t in B:
  out.add(t)
  if |out|>n: emit(out)
```

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



VECTORIZATION MODEL

Ideal for disk dbms

Ideal for OLAP queries
→ Greatly reduces the number of invocations per operator.
→ Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.

PROCESSING MODELS SUMMARY

Iterator / Volcano

→ Direction: Top-Down
→ Emits: Single Tuple
→ Target: General Purpose

Vectorized

→ Direction: Top-Down
→ Emits: Tuple Batch
→ Target: OLAP

Materialization

→ Direction: Bottom-Up
→ Emits: Entire Tuple Set
→ Target: OLTP

ACCESS METHODS

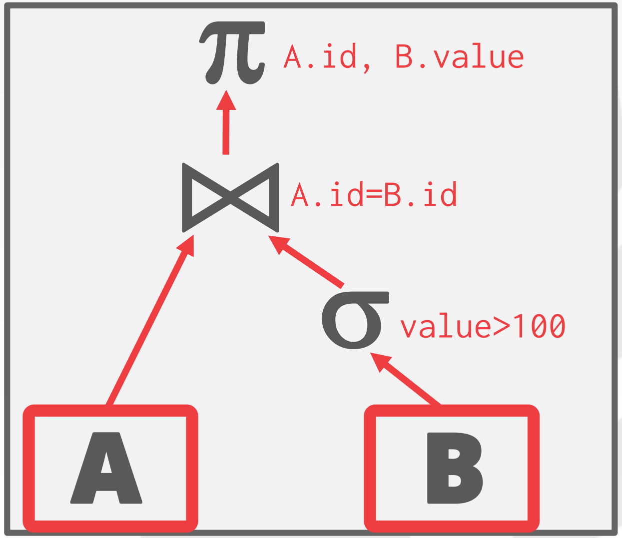
An **access method** is a way that the DBMS can access the data stored in a table.

→ Not defined in relational algebra.

Three basic approaches:

→ Sequential Scan
→ Index Scan
→ Multi-Index / "Bitmap" Scan

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.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
→ Parallelization
→ Buffer Pool Bypass
→ 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.

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 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

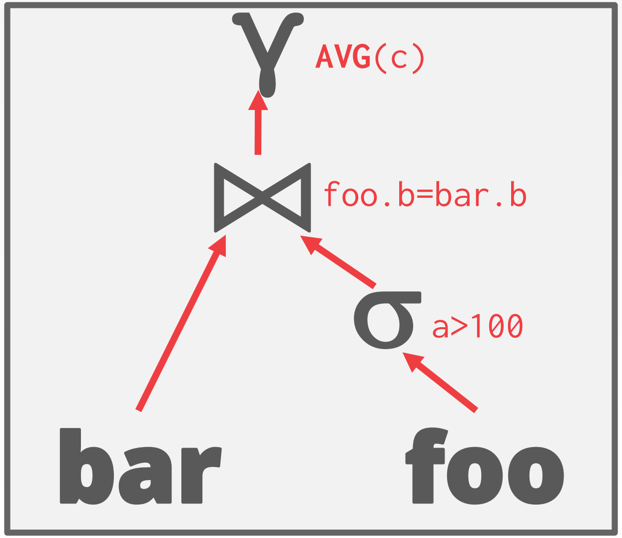
val
100
200
300
400
400

Zone Map

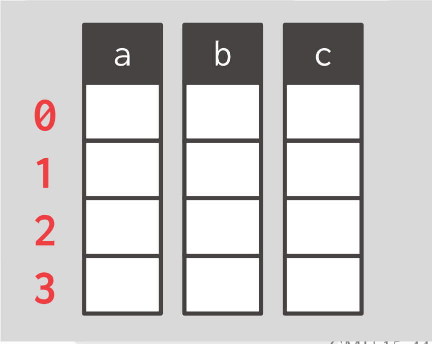
type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

LATE MATERIALIZATION

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

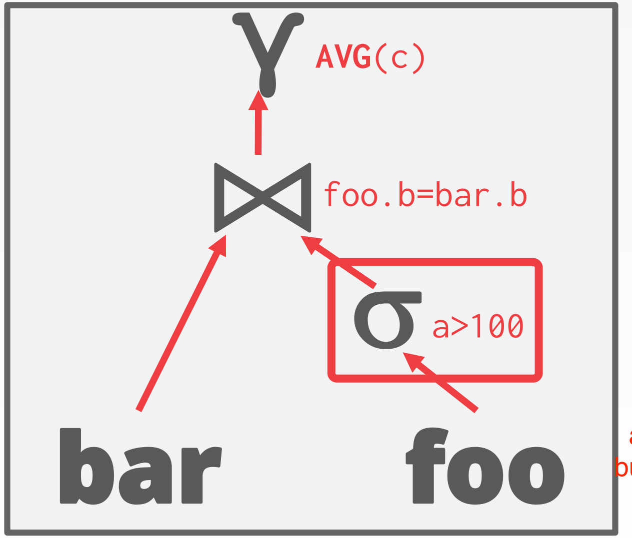



```
SELECT AVG(C)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE a > 100
```



LATE MATERIALIZATION

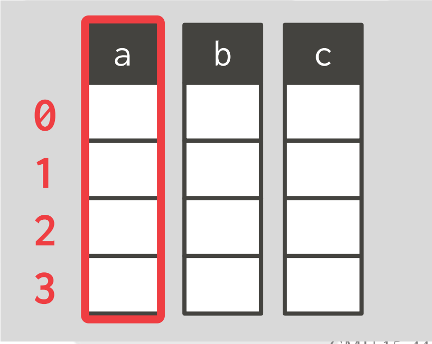
DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.



Offsets

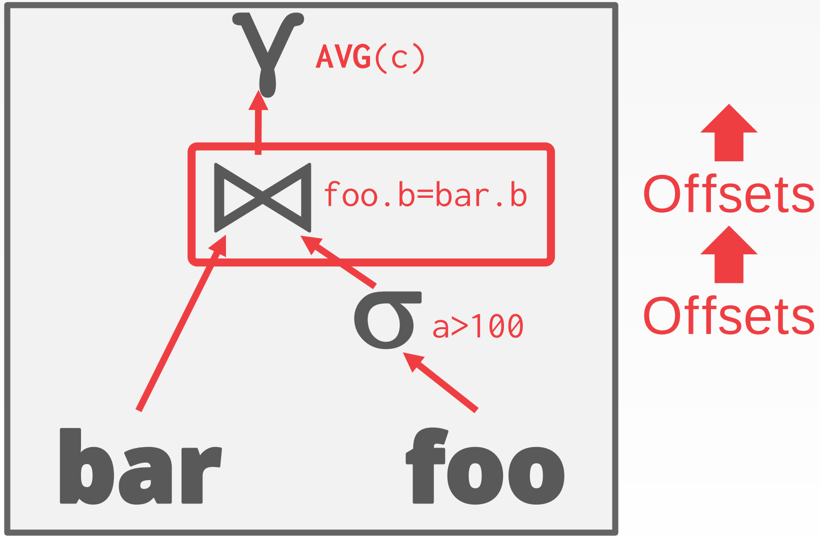
a will be no longer used, so only shove up offsets of a and throw away tuples in the buffer pool to relieve io presure

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

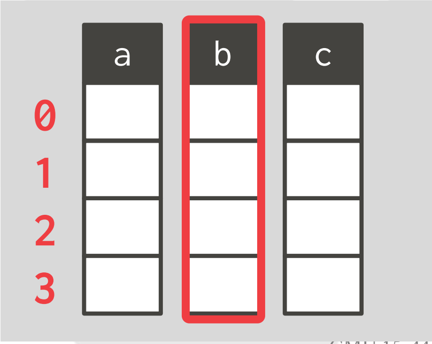


LATE MATERIALIZATION

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

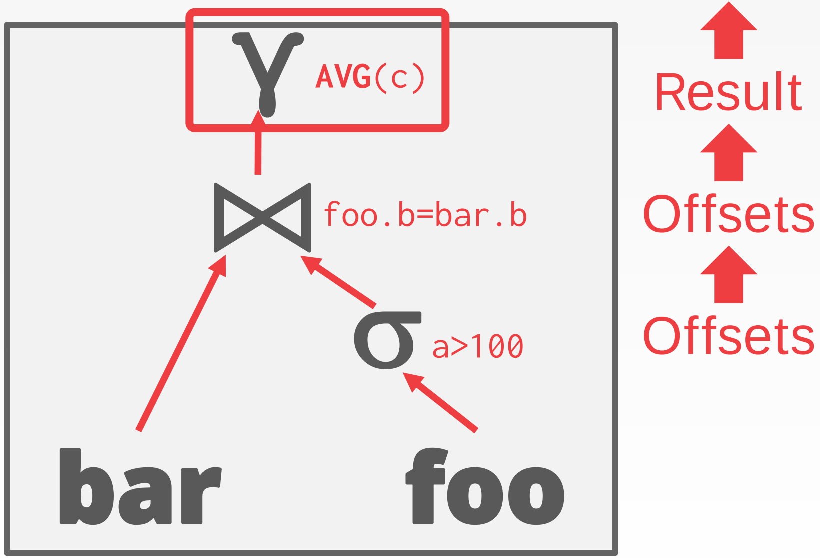



```sql
SELECT AVG(C)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE a > 100
```

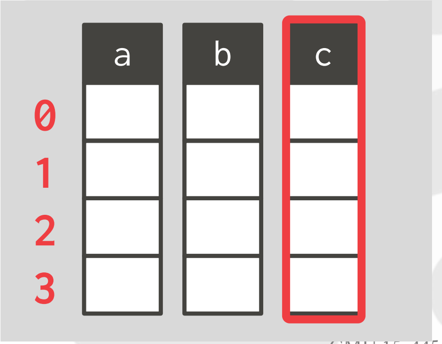


LATE MATERIALIZATION

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



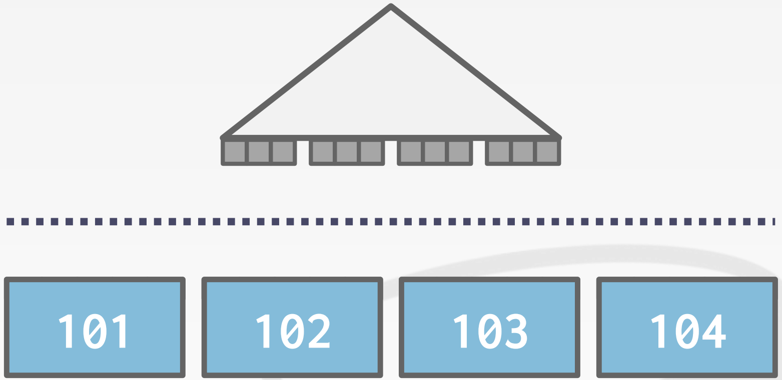

```
SELECT AVG(C)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE a > 100
```



HEAP CLUSTERING

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

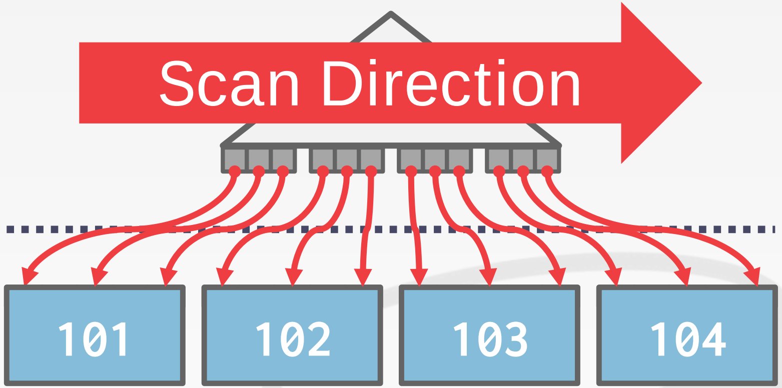
If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.



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.



Indexes created by DBMS are stored in the disk and every time when DBMS starts running, indexes will be loaded from disk to memory and help increase the query speed

INDEX SCAN

when fetching tuples (rows) from disk, the minimum granularity is page which contains several tuples including target tuple, and then do random access on that page, so index is b+ tree data structure which gives efficient way to get target page(address) on the leave nodes

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

Lecture 17

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 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

```
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 terms.

Postgres calls this Bitmap Scan

MULTI-INDEX SCAN

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

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,

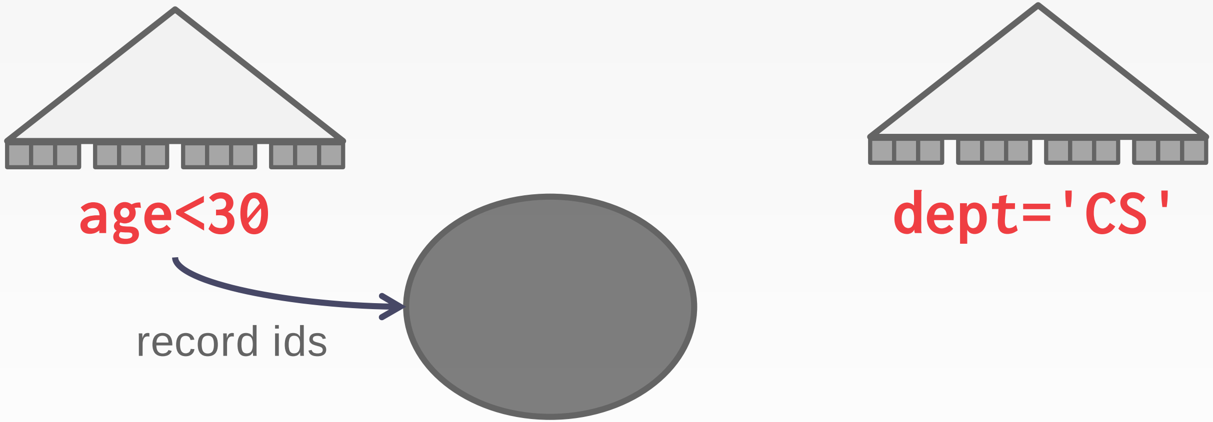
→ Take their intersection

→ Retrieve records and check **country='US'**.

MULTI-INDEX SCAN

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

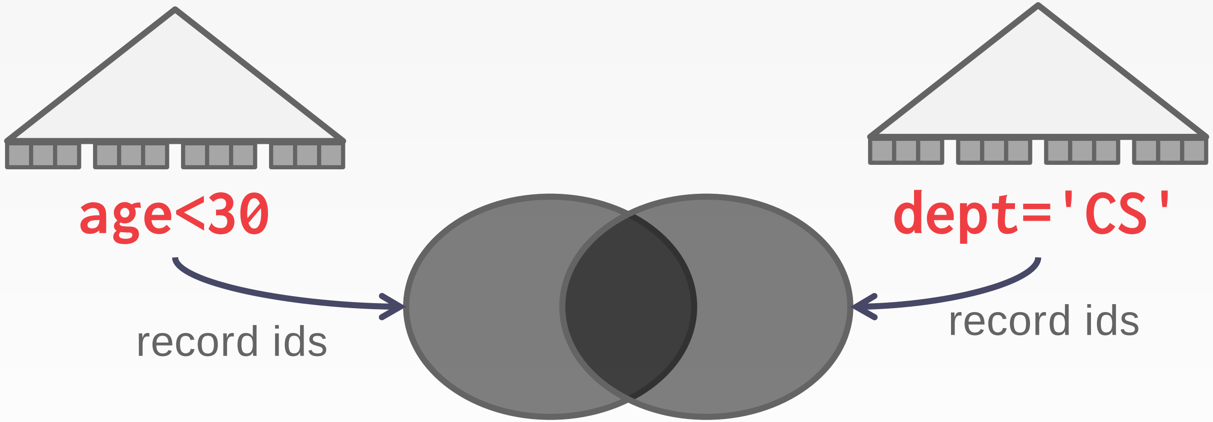
Set intersection can be done with bitmaps, hash tables, or Bloom filters.



MULTI-INDEX SCAN

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

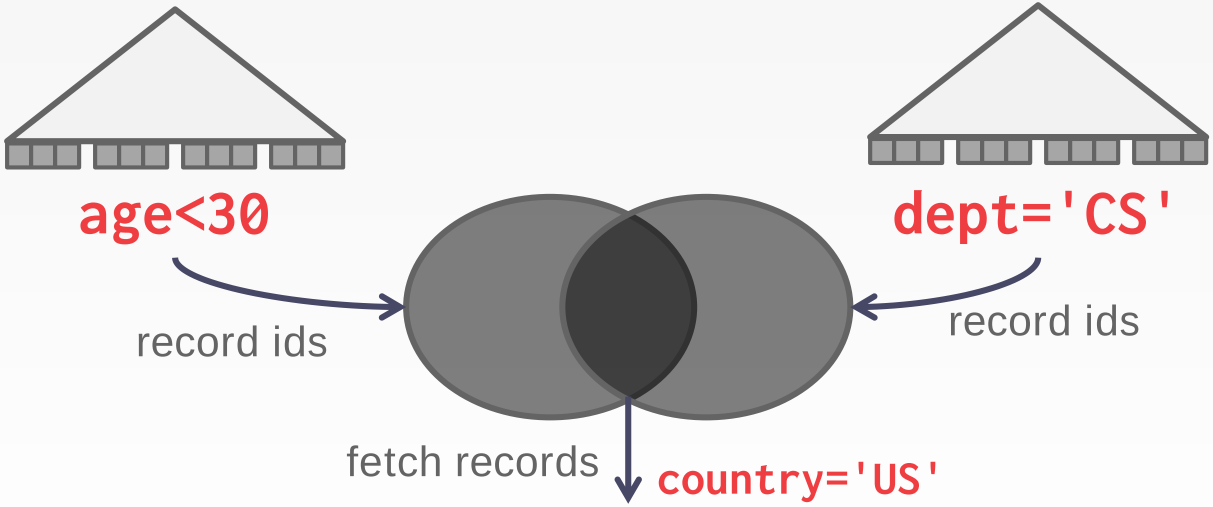
Set intersection can be done with bitmaps, hash tables, or Bloom filters.



MULTI-INDEX SCAN

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

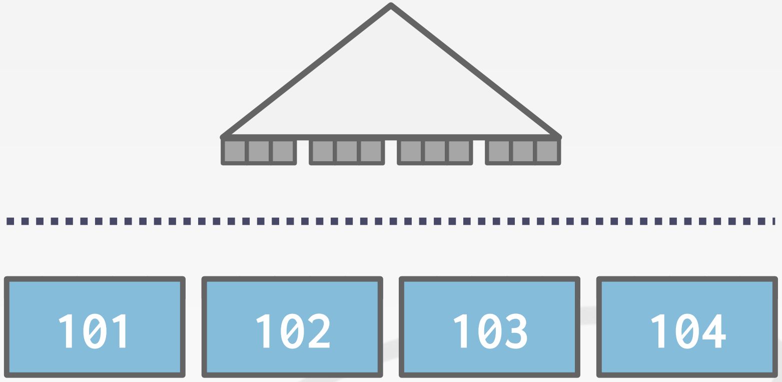
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



INDEX SCAN PAGE SORTING

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

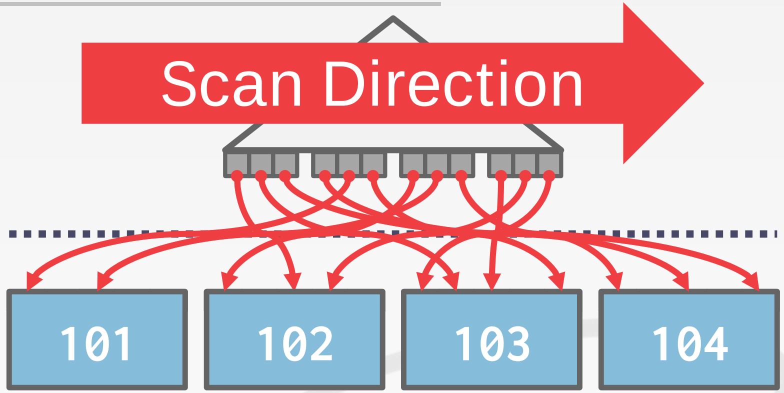
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



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.

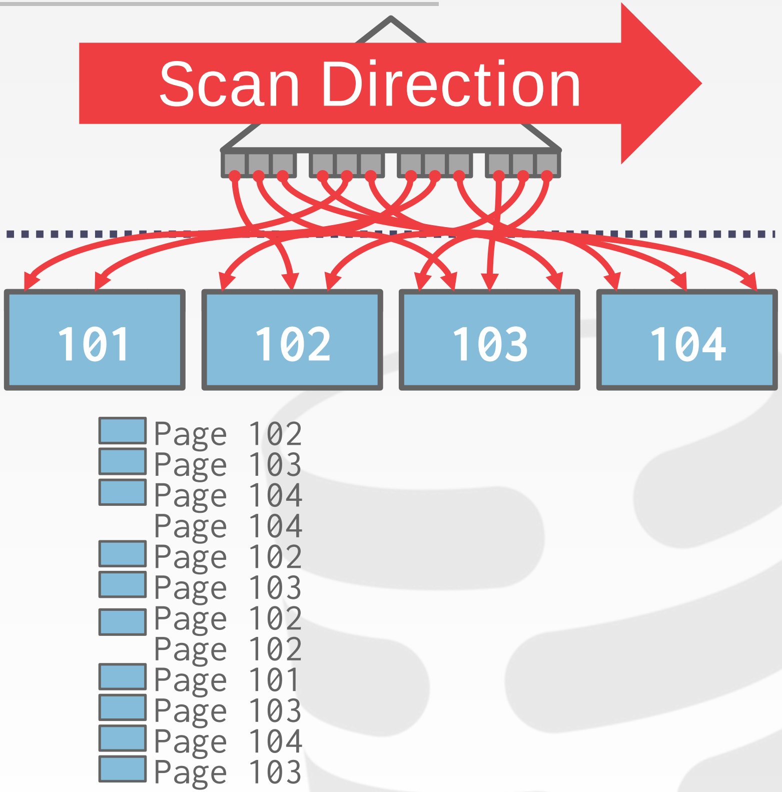


where clauses could be complex and we need to fetch multiple pages via index scan, instead of each time go through index and get one page, we could first invoke index scan to get all tuples we need in where clause, then sort those pages by its id and never go back scan pages we already scanned

INDEX SCAN PAGE SORTING

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

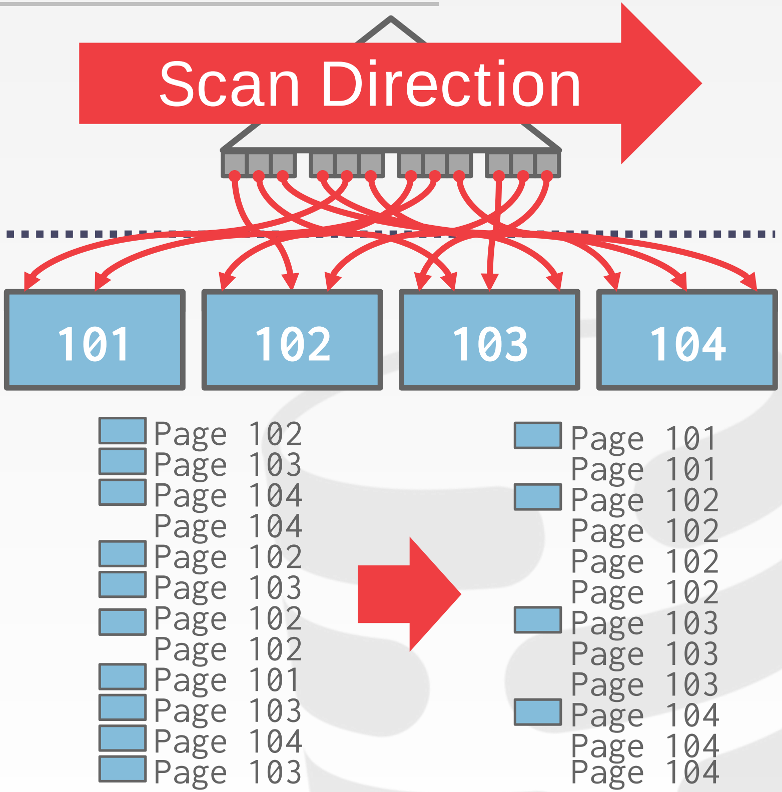
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



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.



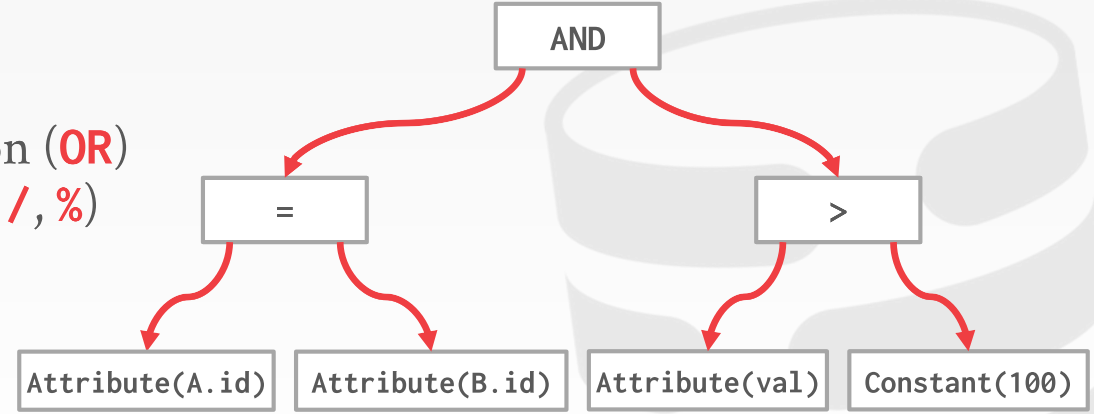
EXPRESSION EVALUATION

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 A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.val > 100
```

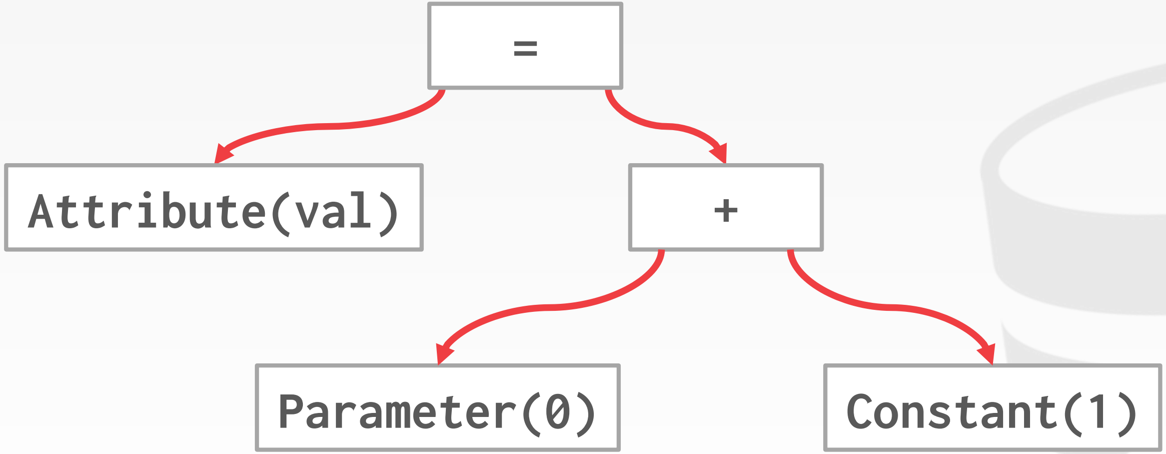


EXPRESSION EVALUATION

```
SELECT * FROM B
 WHERE B.val = ? + 1
```

Execution Context

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

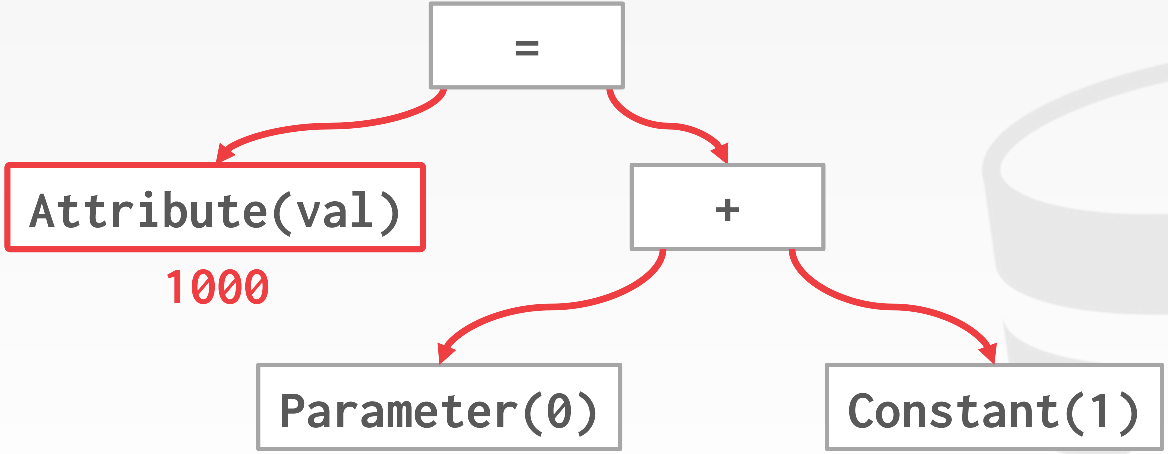


EXPRESSION EVALUATION

```
SELECT * FROM B
 WHERE B.val = ? + 1
```

Execution Context

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

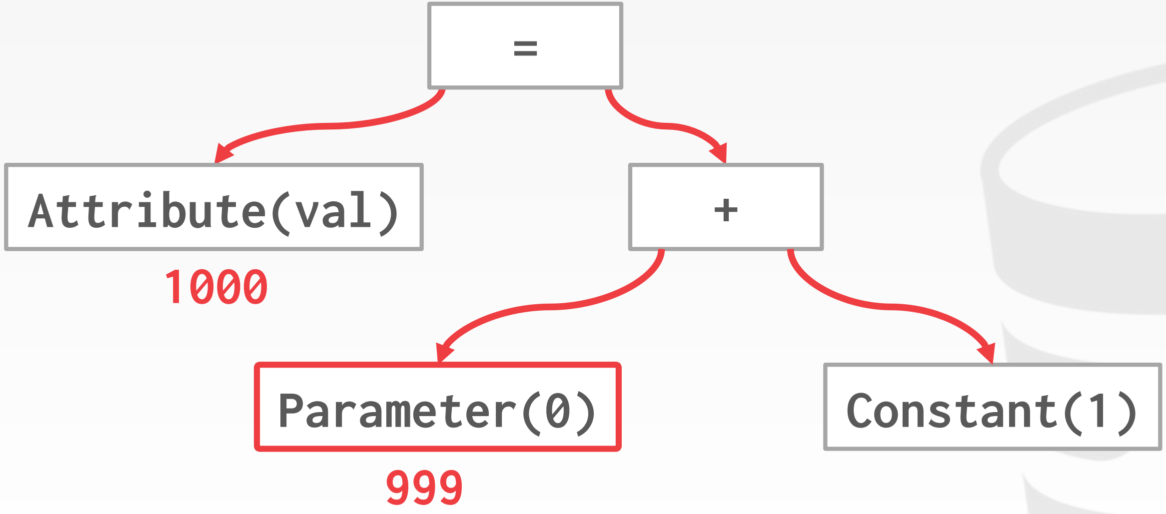


EXPRESSION EVALUATION

```
SELECT * FROM B
 WHERE B.val = ? + 1
```

Execution Context

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

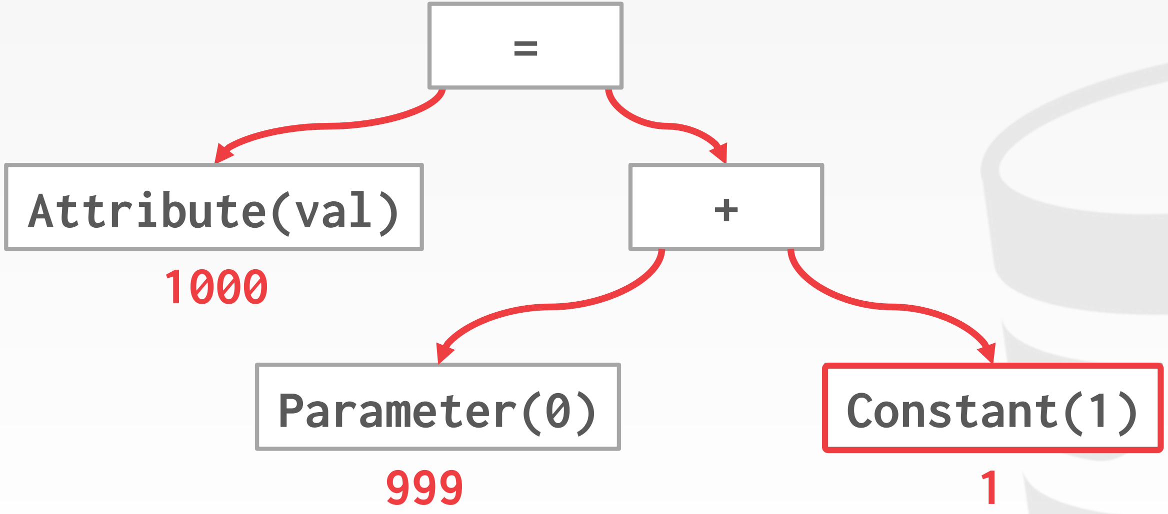


EXPRESSION EVALUATION

```
SELECT * FROM B
 WHERE B.val = ? + 1
```

Execution Context

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



EXPRESSION EVALUATION

```
SELECT * FROM B
 WHERE B.val = ? + 1
```

Execution Context

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