DATA ANALYTICS USING DEEP LEARNING

GT 8803 // FALL 2019 // JOY ARULRAJ

LECTURE #08: QUERY EXECUTION

Georgia Tech

CREATING THE NEXT®

ADMINISTRIVIA

- Reminders
  - Sign up for discussion slots on **Thursday**
  - Proposal presentations on next Wednesday
  - 2 page document + 5 minute presentation
  - Assignment 1 due on Wednesday

LAST CLASS

- Storage models: NSM, DSM, and FSM

FLEXIBLE STORAGE MODEL

ID	University	Enrollment	City
1	Georgia Tech	15000	Atlanta
2	Wisconsin	30000	Madison
3	Carnegie Mellon	6000	Pittsburgh
4	UC Berkeley	30000	Berkeley

LAST CLASS

- Compression
  - Zone maps
  - Dictionary encoding

Original Data

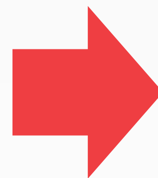
val
100
200
300
400
400

LAST CLASS

- Compression
  - Zone maps
  - Dictionary encoding

Original Data

val
100
200
300
400
400



Zone Map

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

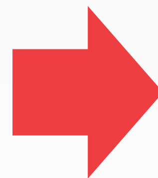
LAST CLASS

- Compression
  - Zone maps
  - Dictionary encoding

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

Original Data

val
100
200
300
400
400

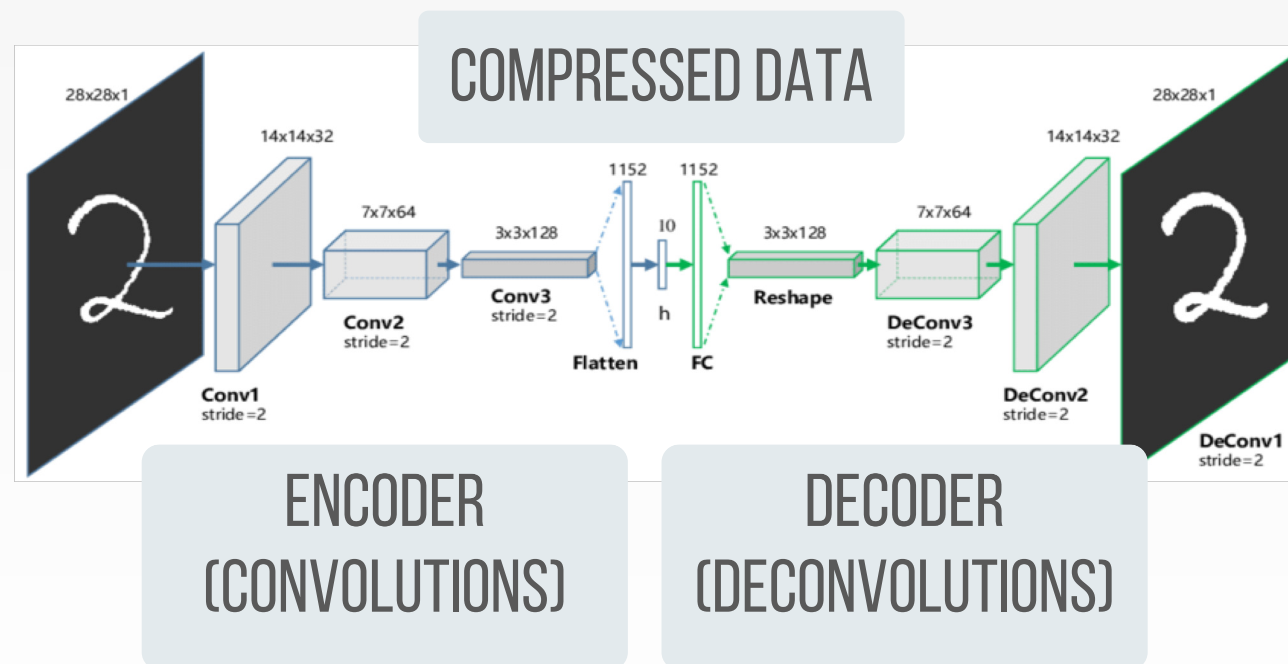


Zone Map

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

LAST CLASS

- Visual Storage Engine
  - Convolutional Auto Encoder

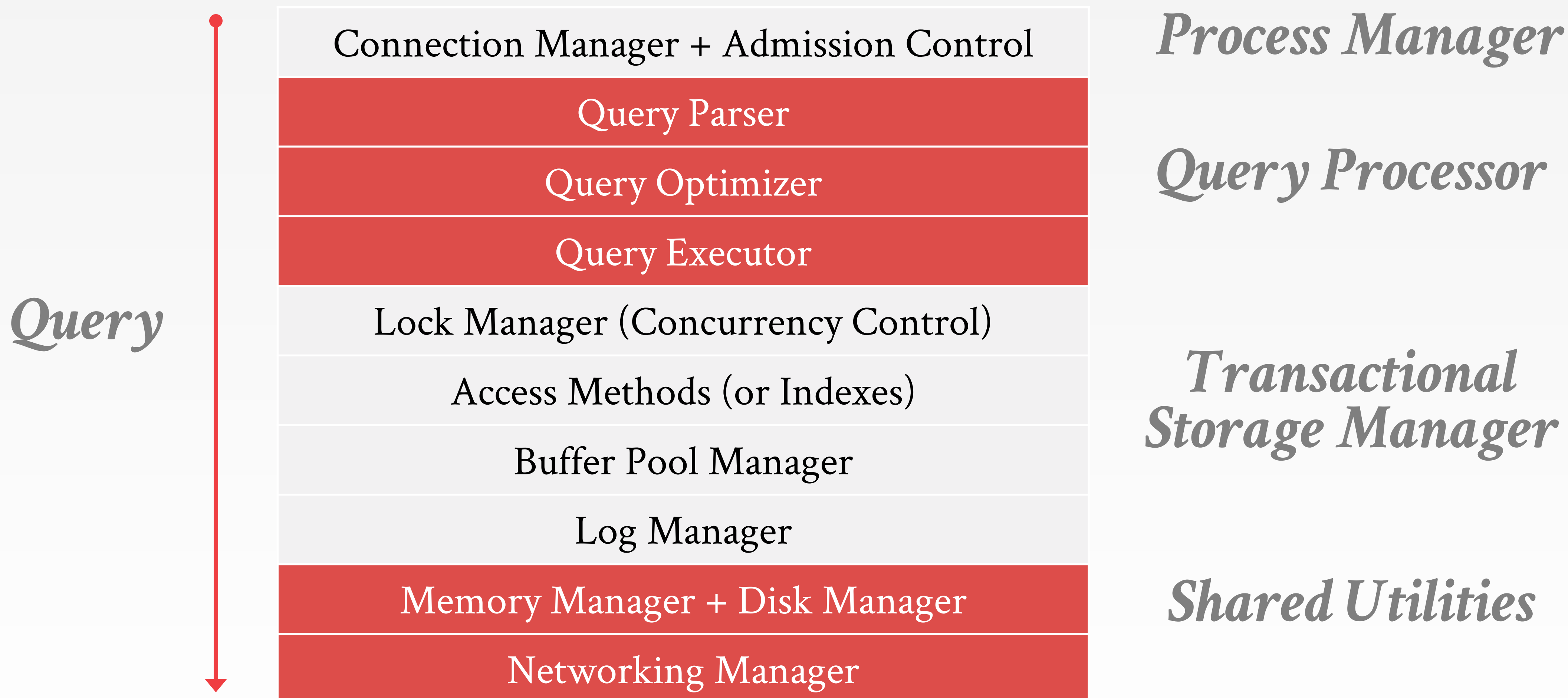


TODAY'S AGENDA

- Query Processing Models
- Access Methods
- Expression Evaluation
- Visual Query Execution Engine

QUERY PROCESSING MODELS

ANATOMY OF A DATABASE SYSTEM

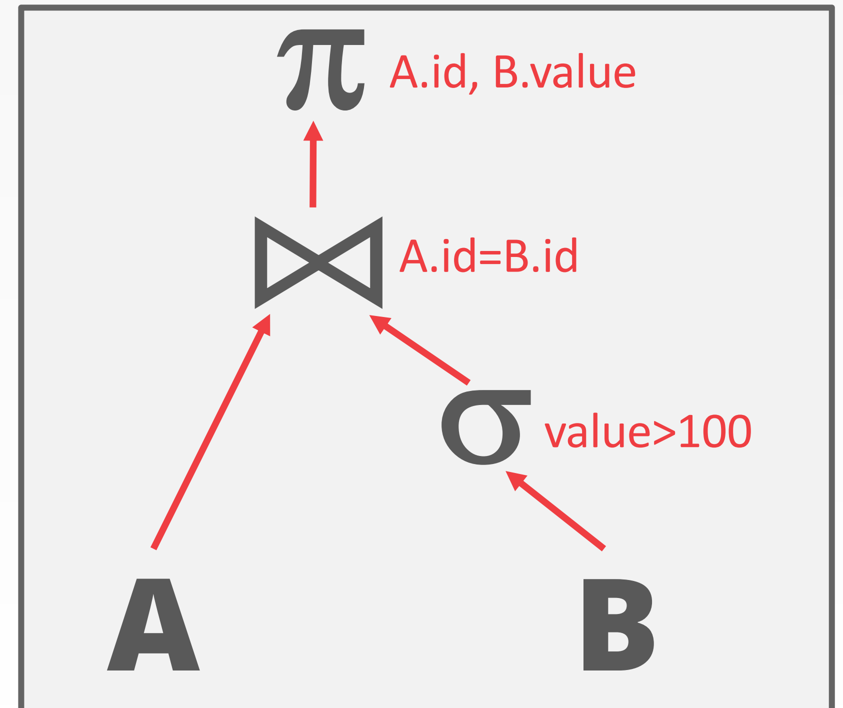


Source: Anatomy of a Database System

QUERY PLAN

- The operators are arranged in a tree.
  - Data flows from the leaves toward the root.
  - Output of the root node is the result of the query.

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



QUERY PROCESSING MODELS

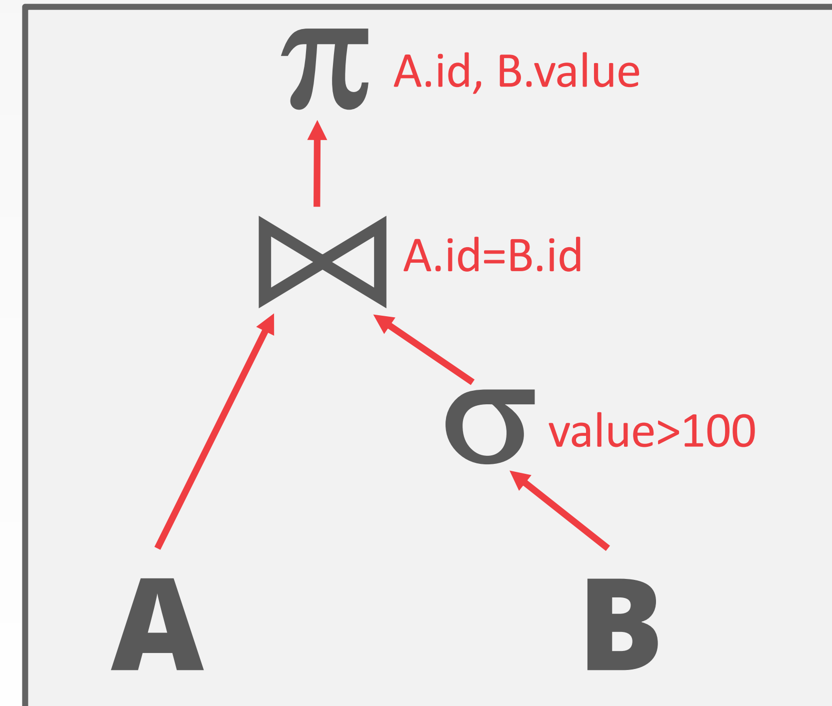
- A DBMS's **processing model** defines how the system executes a query plan.
  - Different trade-offs for different workloads.
  - Top-down or Bottom-up execution
  - Determines what data is sent between operators
- Three approaches:
  - Tuple-at-a-time Model
  - Operator-at-a-time Model
  - Vector-at-a-time Model

#1: TUPLE-AT-A-TIME MODEL

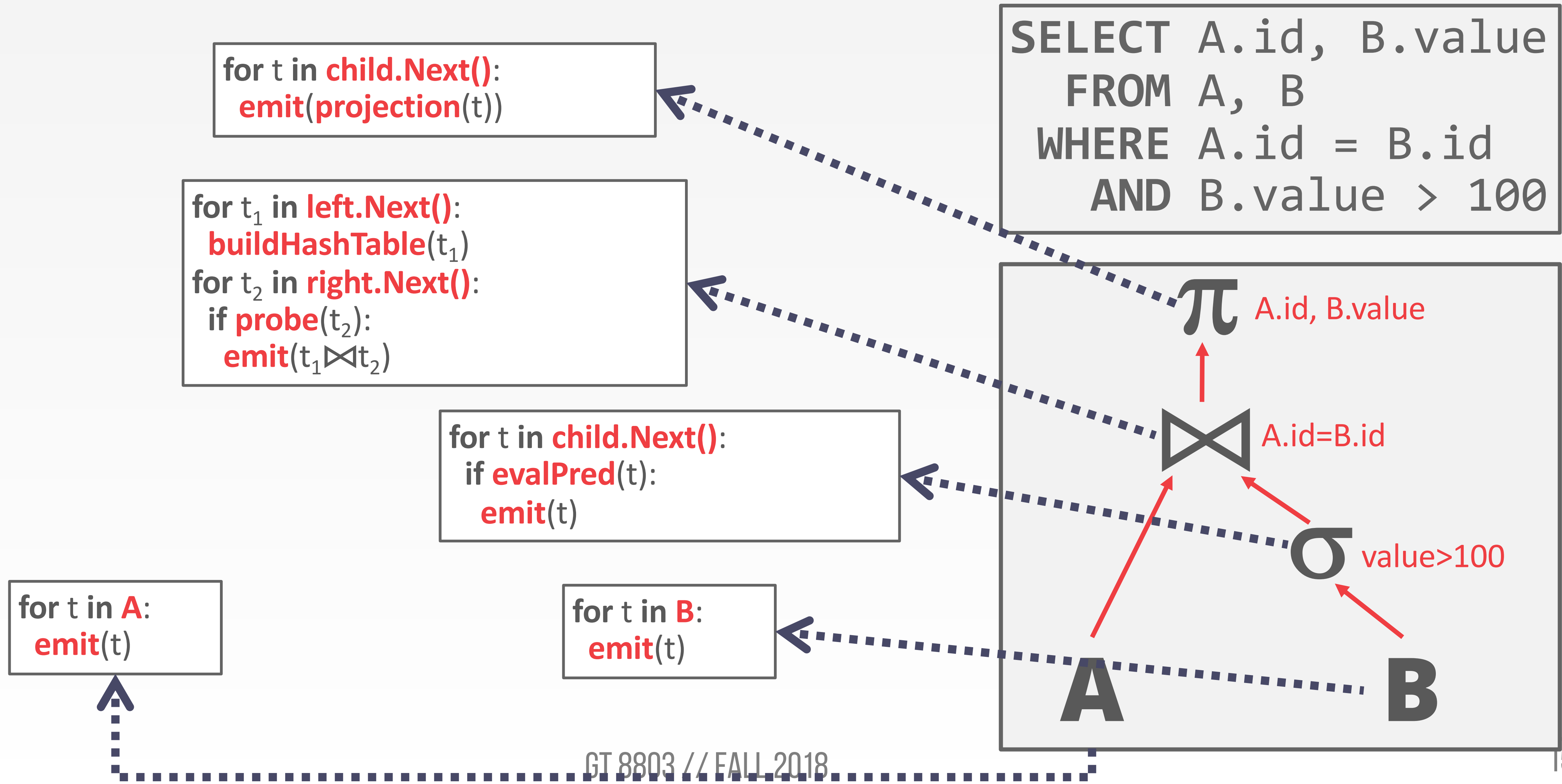
- Each query plan operator implements a **next** function.
  - On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples.
  - The operator implements a loop that calls next on its children to retrieve their tuples and then process them.
- Top-down plan processing.
- Also called **Iterator** / **Volcano** / **Pipeline** model.

#1: TUPLE-AT-A-TIME MODEL

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



#1: TUPLE-AT-A-TIME MODEL



#1: TUPLE-AT-A-TIME MODEL

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

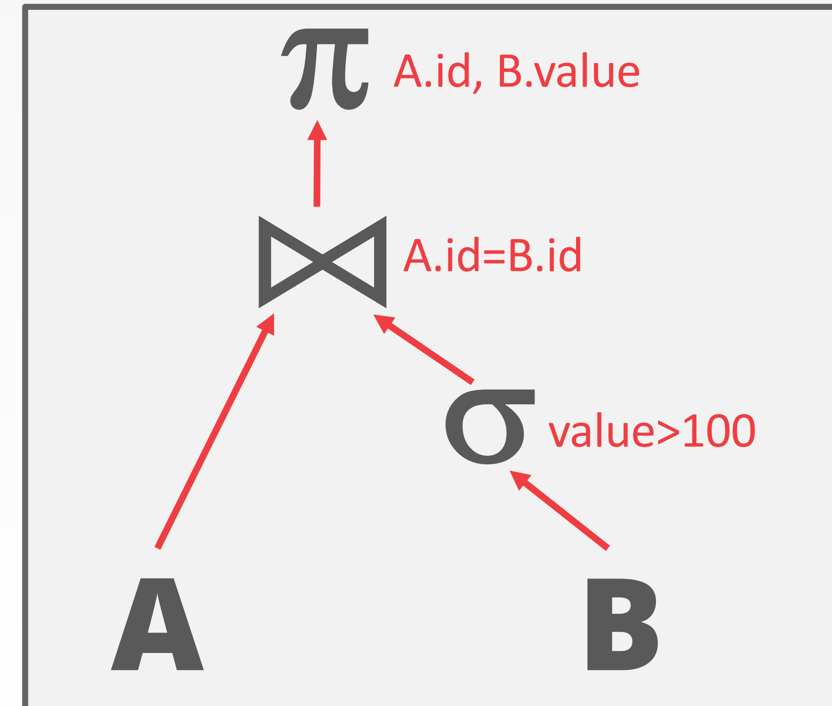
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2):
    emit(t1⋈t2)
```

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

```
for t in 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
```



Georgia Tech

#1: TUPLE-AT-A-TIME MODEL

1

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

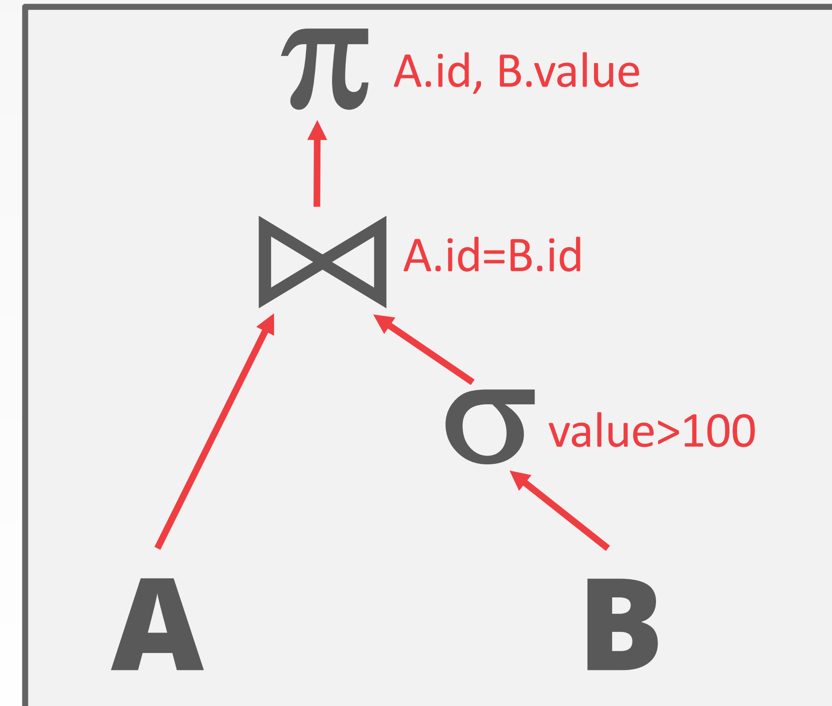
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2):
    emit(t1⋈t2)
```

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

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

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

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



#1: TUPLE-AT-A-TIME MODEL

1

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

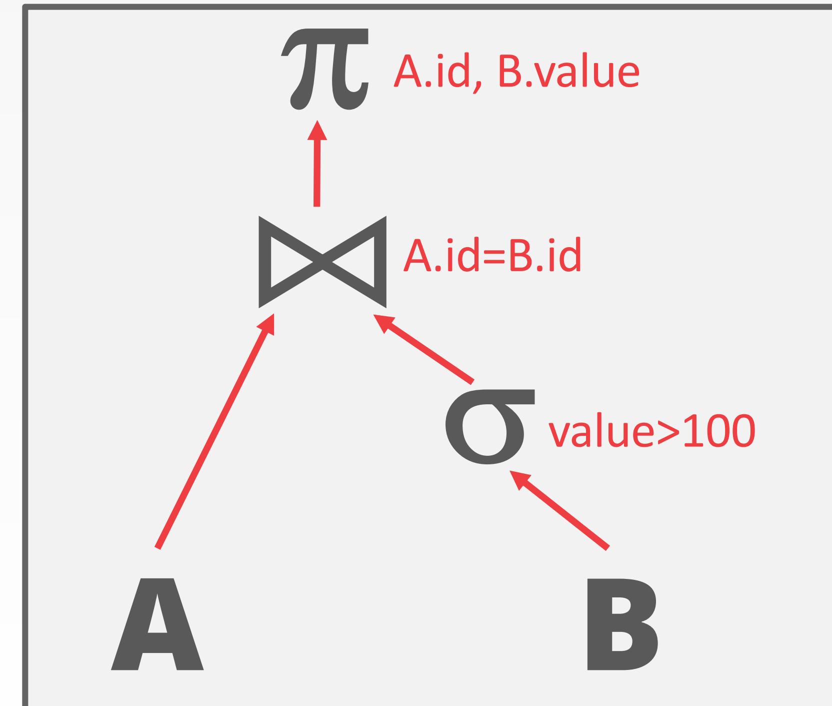
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2):
    emit(t1⋈t2)
```

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

```
for t in 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
```



#1: TUPLE-AT-A-TIME MODEL

1

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

2

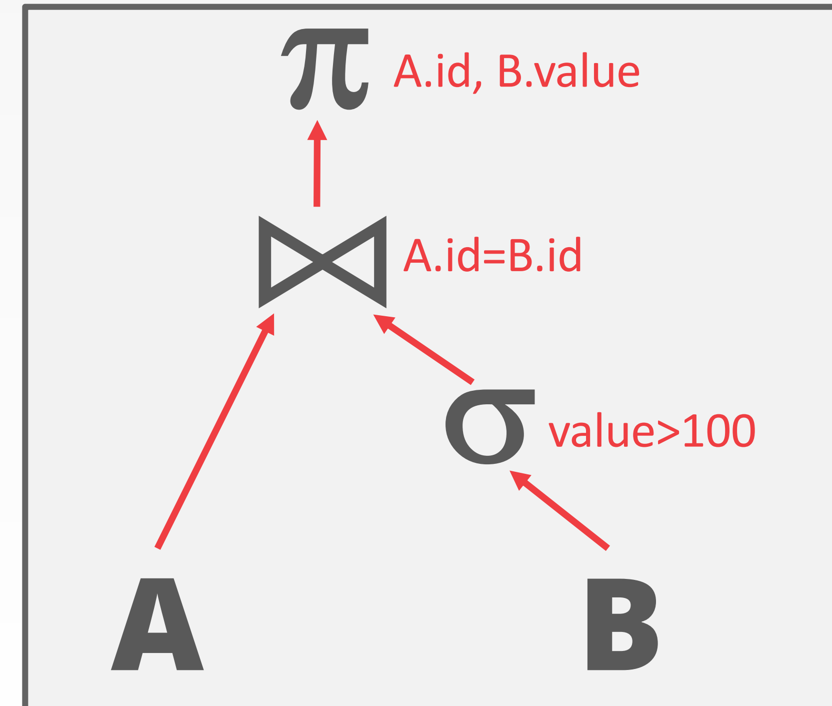
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2):
    emit(t1⋈t2)
```

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

```
for t in 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
```



#1: TUPLE-AT-A-TIME 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)
```

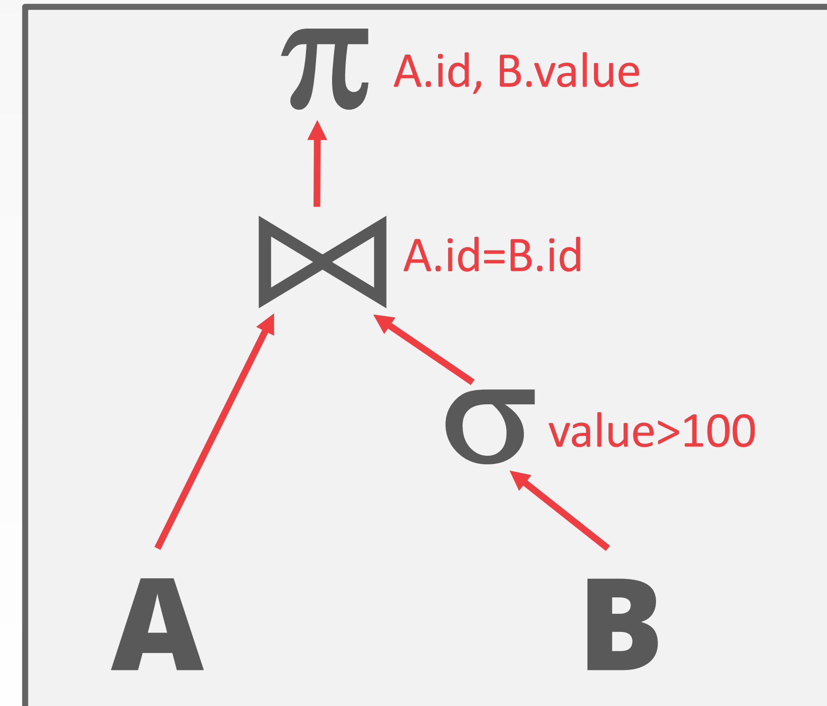
```
for t in child.Next():
  if evalPred(t):
    emit(t)
```

3

```
for t in 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
```



#1: TUPLE-AT-A-TIME 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)
```

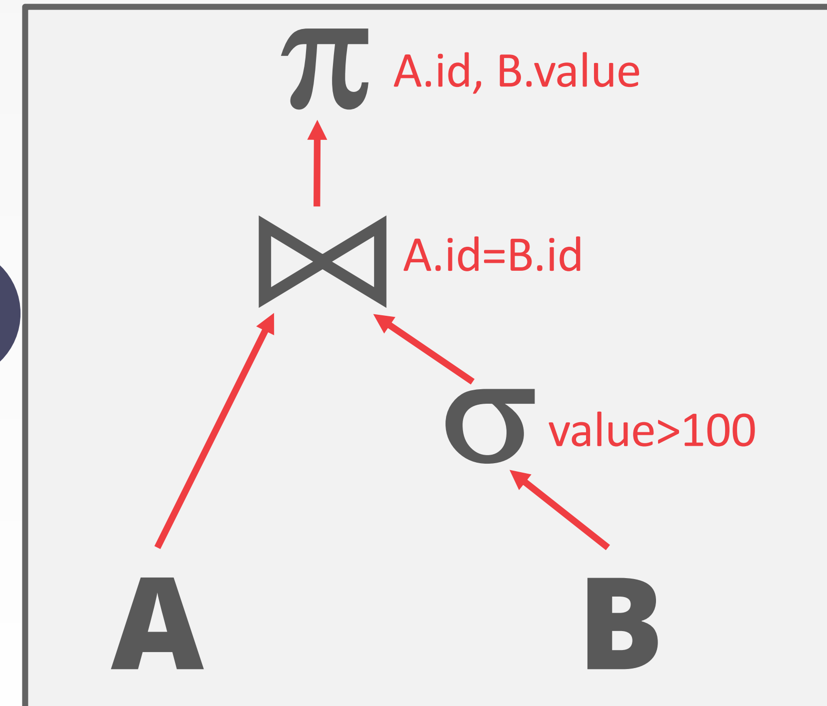
3

```
for t in A:
  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
```



#1: TUPLE-AT-A-TIME 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
  - Known as pipeline breakers
- Output control works easily with this approach.
  - Limit

#2: OPERATOR-AT-A-TIME 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.

#2: OPERATOR-AT-A-TIME MODEL

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

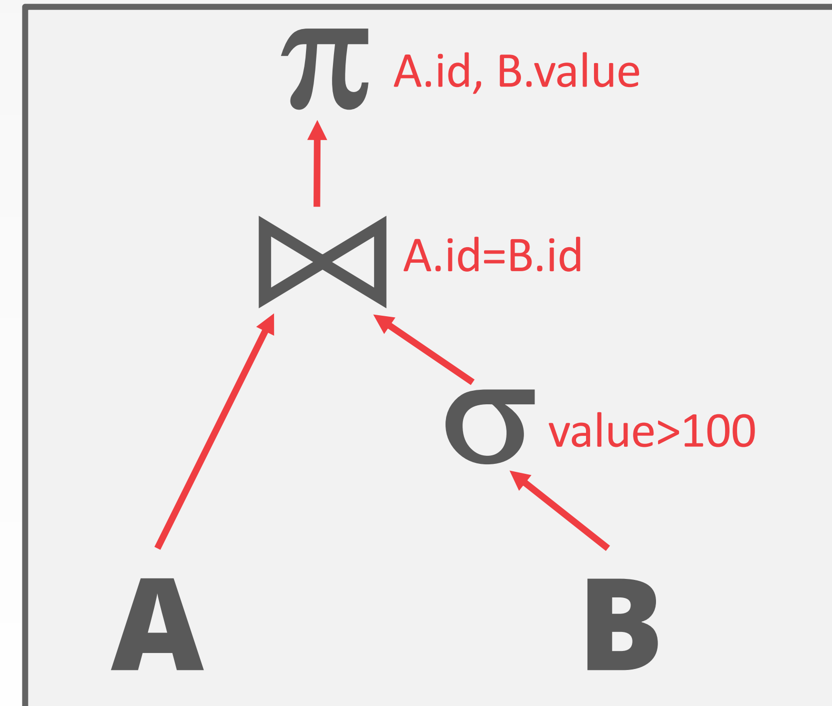
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
```

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

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

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

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



#2: OPERATOR-AT-A-TIME 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)
```

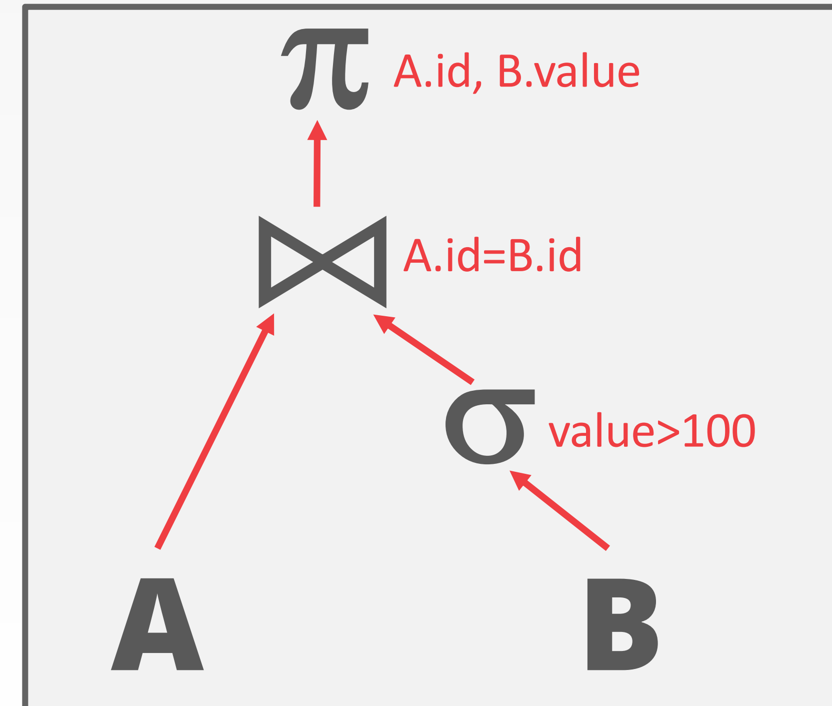
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
```

1

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

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

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



#2: OPERATOR-AT-A-TIME 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)
```

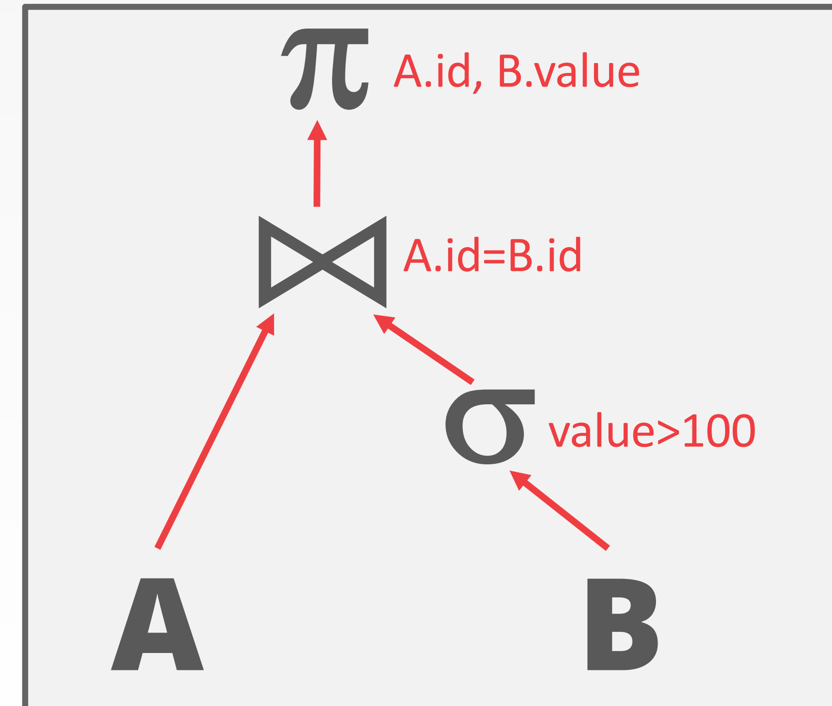
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
```

1

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

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

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



#2: OPERATOR-AT-A-TIME 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)
```

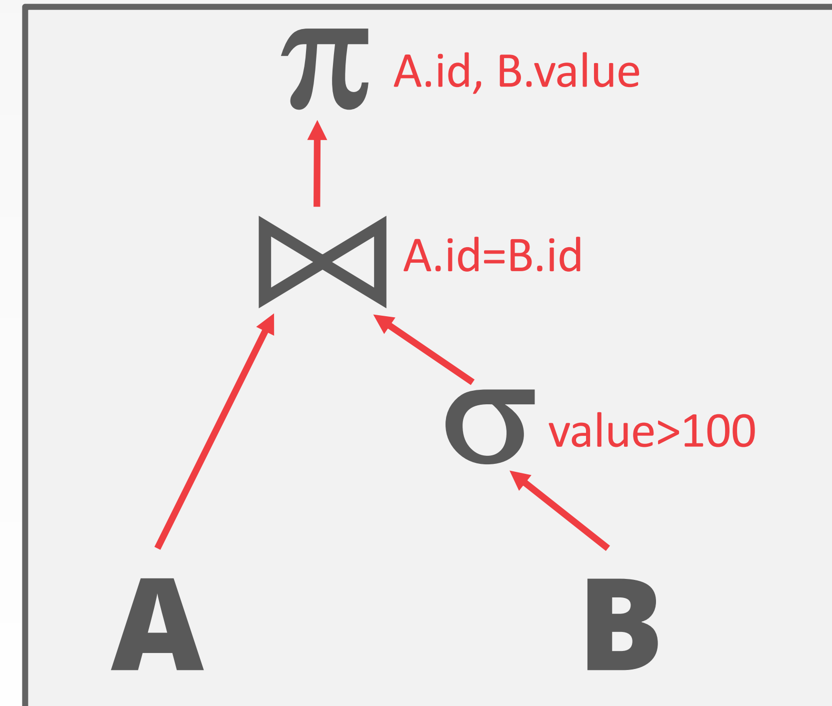
1

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

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

2

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



#2: OPERATOR-AT-A-TIME MODEL

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

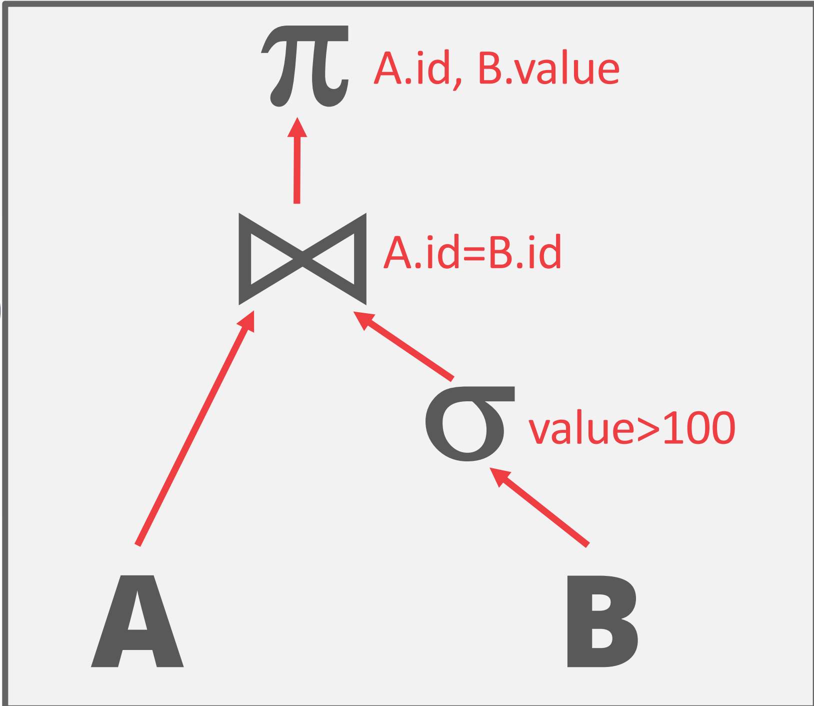
```
out = { }
for t₁ in left.Output():
  buildHashTable(t₁)
for t₂ in right.Output():
  if probe(t₂): out.add(t₁⋈t₂)
```

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

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

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

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



#2: OPERATOR-AT-A-TIME MODEL

```
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)
```

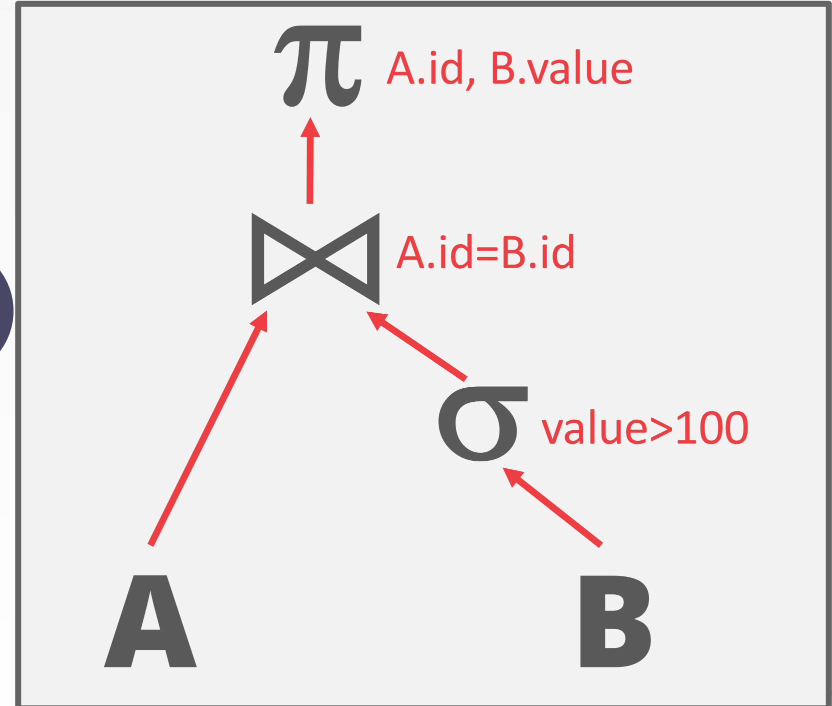
1

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

2

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

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



#2: OPERATOR-AT-A-TIME MODEL

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

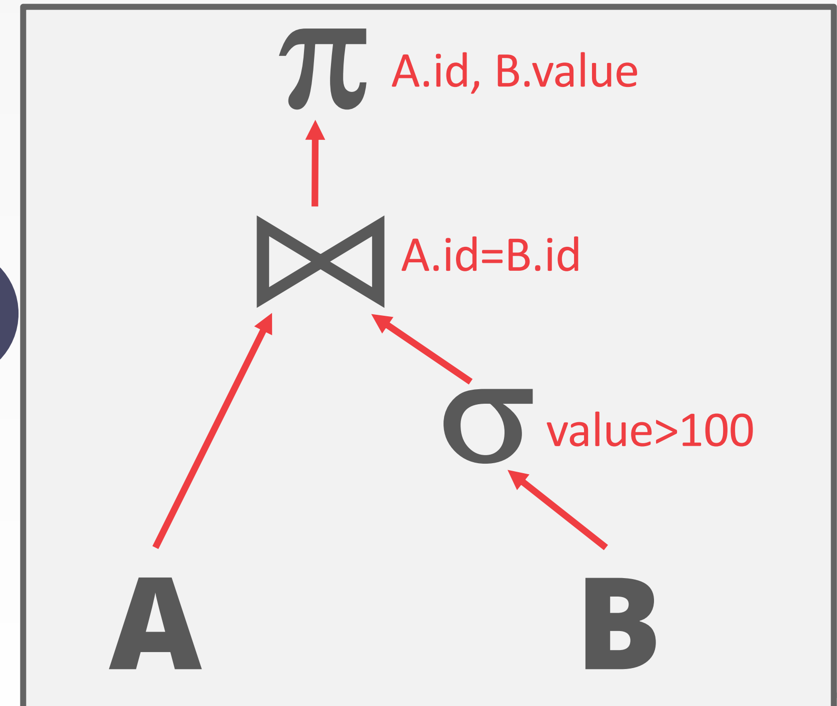
4
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
```

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)
```

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



#2: OPERATOR-AT-A-TIME MODEL

- Better for OLTP workloads
  - Transactions typically only access a small number of tuples at a time.
  - Lower execution / coordination overhead.
- Not good for OLAP queries with large intermediate results.

#3: VECTOR-AT-A-TIME MODEL

- Like tuple-at-a-time model, each operator implements a **next** function.
- Each operator emits a **vector** (i.e., **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.

#3: VECTOR-AT-A-TIME MODEL

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

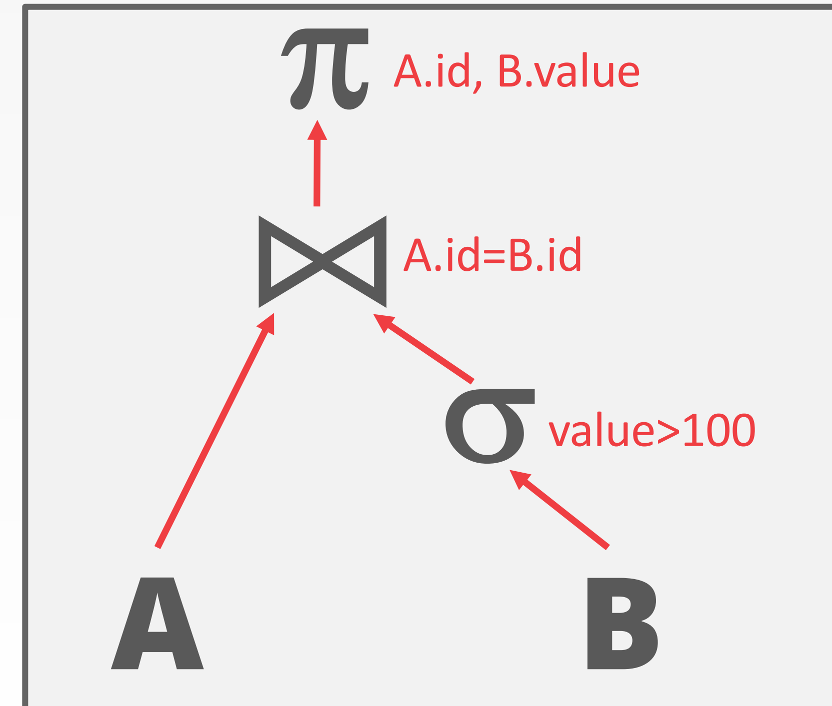
```
out = { }
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out| > n: emit(out)
```

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

```
out = { }
for t in 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
```



Georgia Tech

#3: VECTOR-AT-A-TIME 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)
```

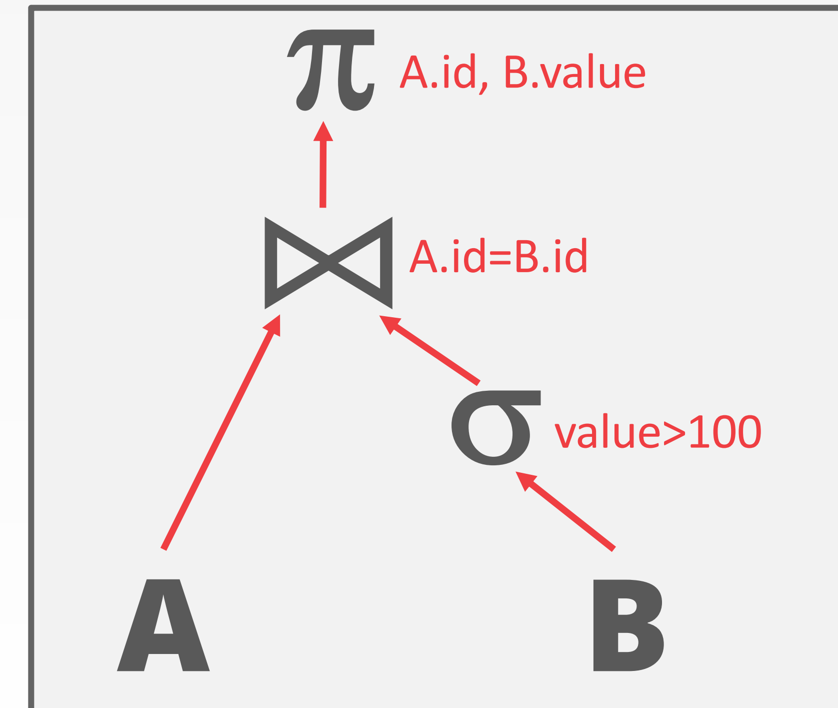
```
out = { }
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|> n : emit(out)
```

3

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

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

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



#3: VECTOR-AT-A-TIME MODEL

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

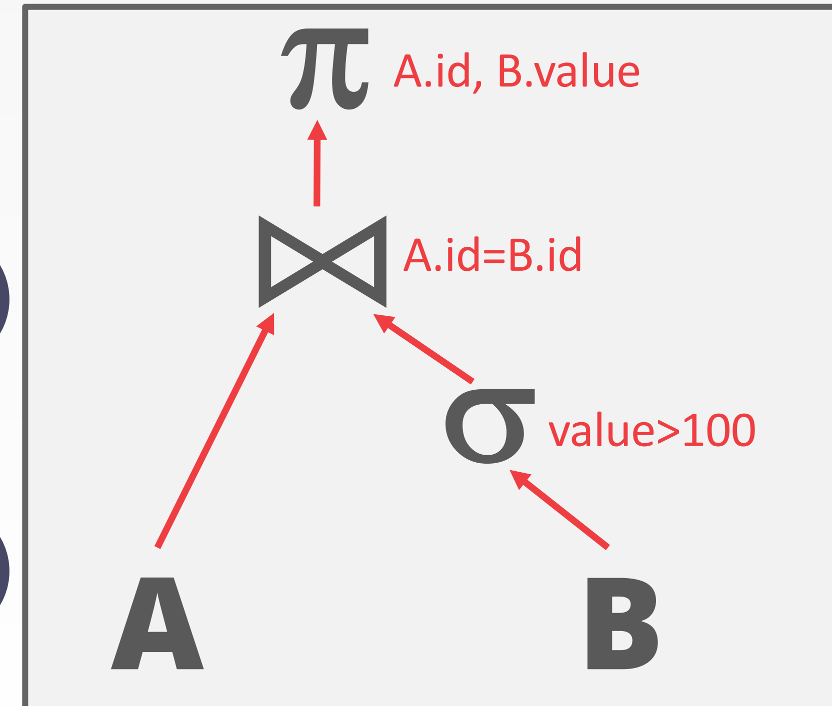
2
```
out = { }
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|> n: emit(out)
```

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

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

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

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



#3: VECTOR-AT-A-TIME MODEL

#3: VECTOR-AT-A-TIME MODEL

- **Q:** What is the target workload for this model: OLTP or OLAP?
  - Reduces number of **next** function invocations
  - Works well for OLAP
- **Q:** Why is it better than operator-at-a-time model in traditional disk-centric DBMSs?
  - Intermediate tables may not fit in main-memory
  - Fetching batches of tuples reduces number of page accesses

#3: VECTOR-AT-A-TIME MODEL

- **Q:** What is the target workload for this model: OLTP or OLAP?
  – Reduces number of **next** function invocations
  – Works well for OLAP
- **Q:** Why is it better than operator-at-a-time model in traditional disk-centric DBMSs?
  – Intermediate tables may not fit in main-memory
  – Fetching batches of tuples reduces number of page accesses

#3: VECTOR-AT-A-TIME MODEL

- **Q:** What is the target workload for this model: OLTP or OLAP?
  - Reduces number of **next** function invocations
  - Works well for OLAP
- **Q:** Why is it better than operator-at-a-time model in traditional disk-centric DBMSs?
  - Intermediate tables may not fit in main-memory
  - Fetching batches of tuples reduces number of page accesses

#3: VECTOR-AT-A-TIME MODEL

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

QUERY PROCESSING MODELS: SUMMARY

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

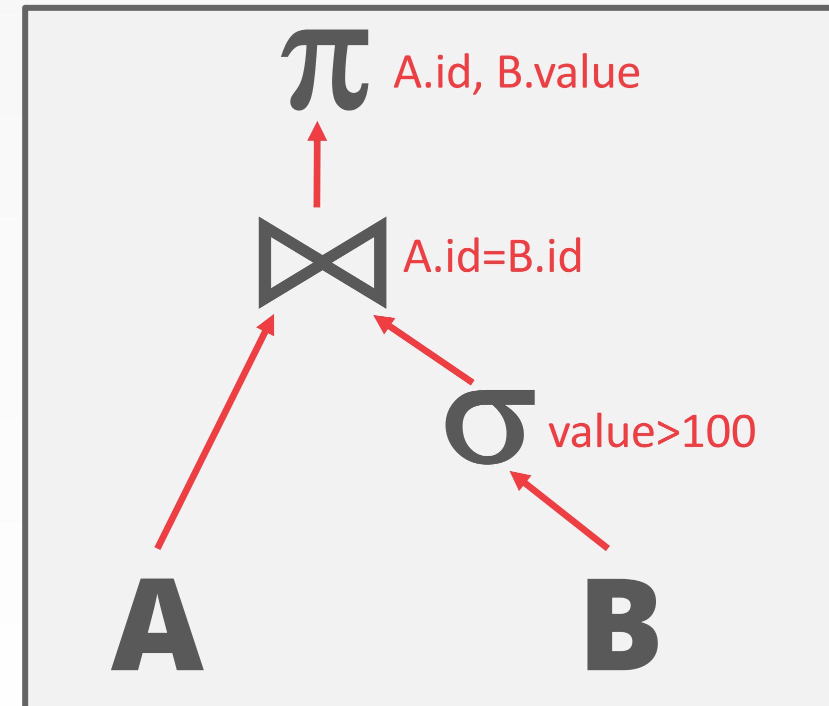
Operator-at-a-time
→Direction: Bottom-Up
→Emits: Entire Tuple Set
→Target: OLTP

Vector-at-a-time
→Direction: Top-Down
→Emits: Tuple Batch
→Target: OLAP

ACCESS METHODS

- An **access method** is a way that the DBMS can access the data stored in a table.
  - Not defined in relational algebra
  - Physical database design
- Three basic methods:
  - Sequential Scan
  - Index Scan
  - Multi-index / "Bitmap" Scan

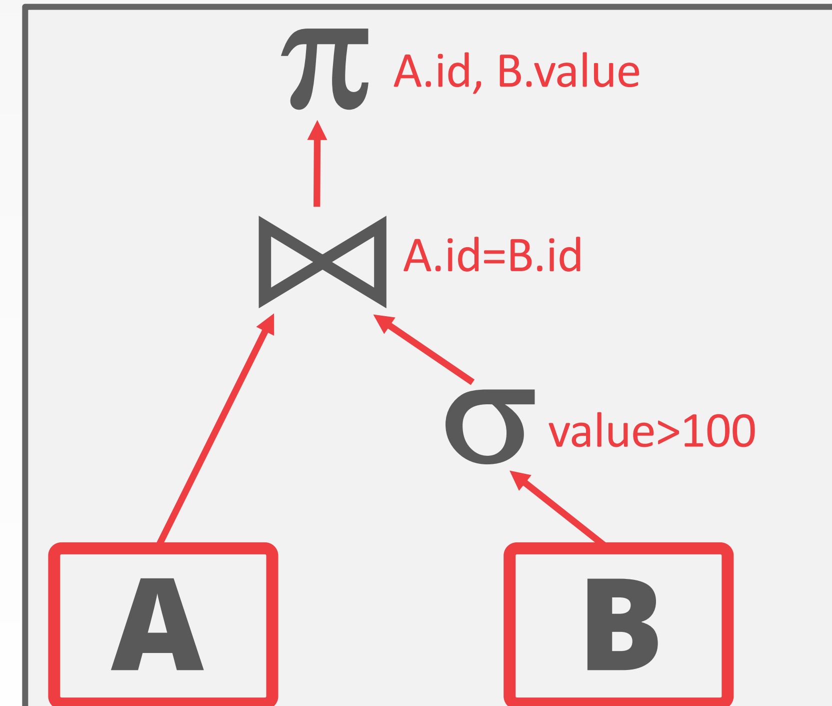
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ACCESS METHODS

- An **access method** is a way that the DBMS can access the data stored in a table.
  - Not defined in relational algebra
  - Physical database design
- Three basic methods:
  - 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
```



#1: 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.

#1: SEQUENTIAL SCAN

- 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

#1: SEQUENTIAL SCAN

- 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

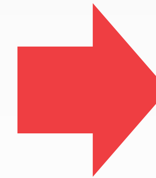
val
100
200
300
400
400

ZONE 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**

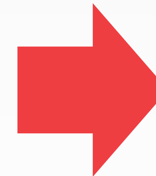
ZONE MAPS

- 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.

LATE MATERIALIZATION

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

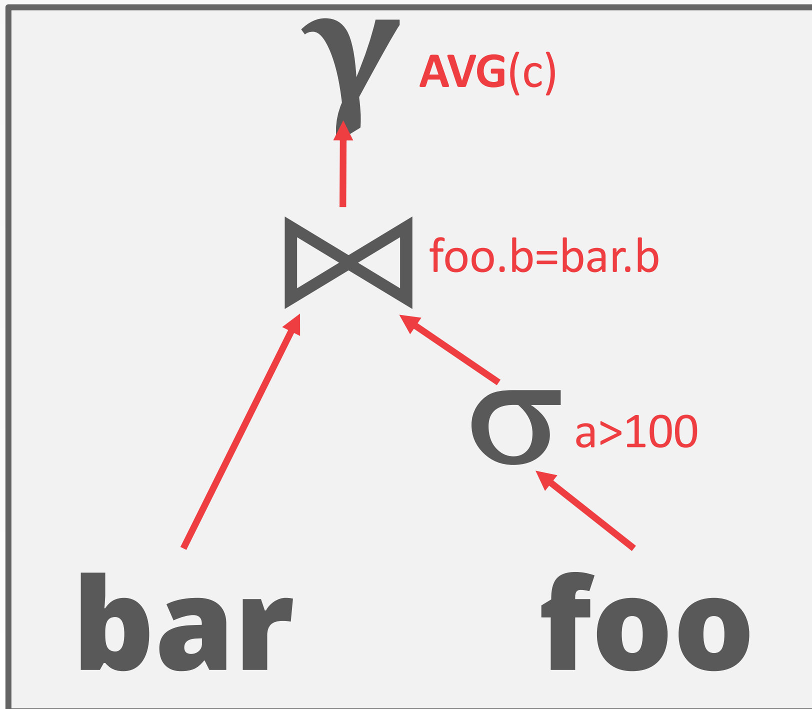
```
SELECT AVG(C)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE a > 100
```

	a	b	c
0			
1			
2			
3			

Georgia Tech

LATE MATERIALIZATION

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

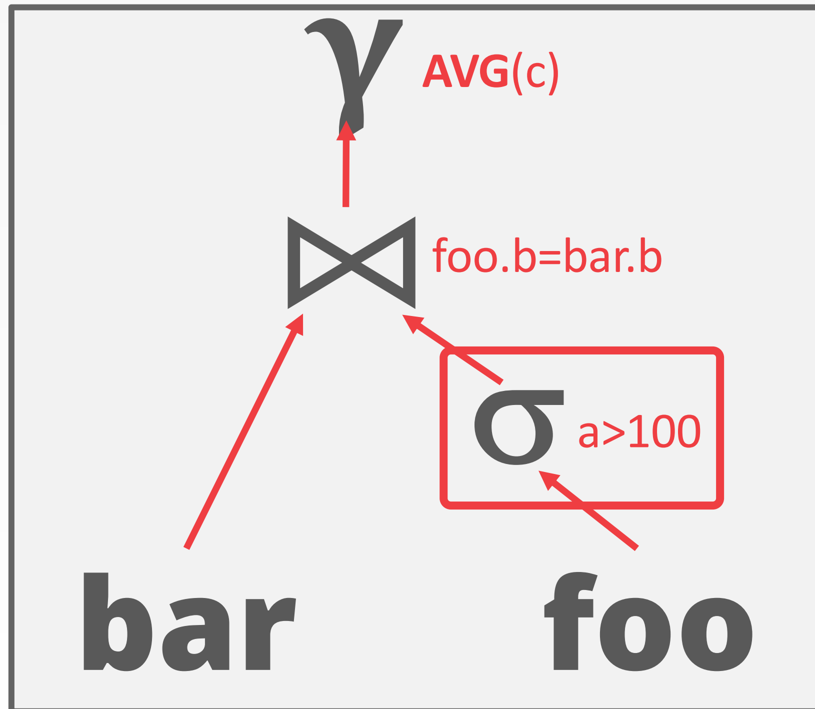



```
SELECT AVG(C)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE a > 100
```

	a	b	c
0			
1			
2			
3			

LATE MATERIALIZATION

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



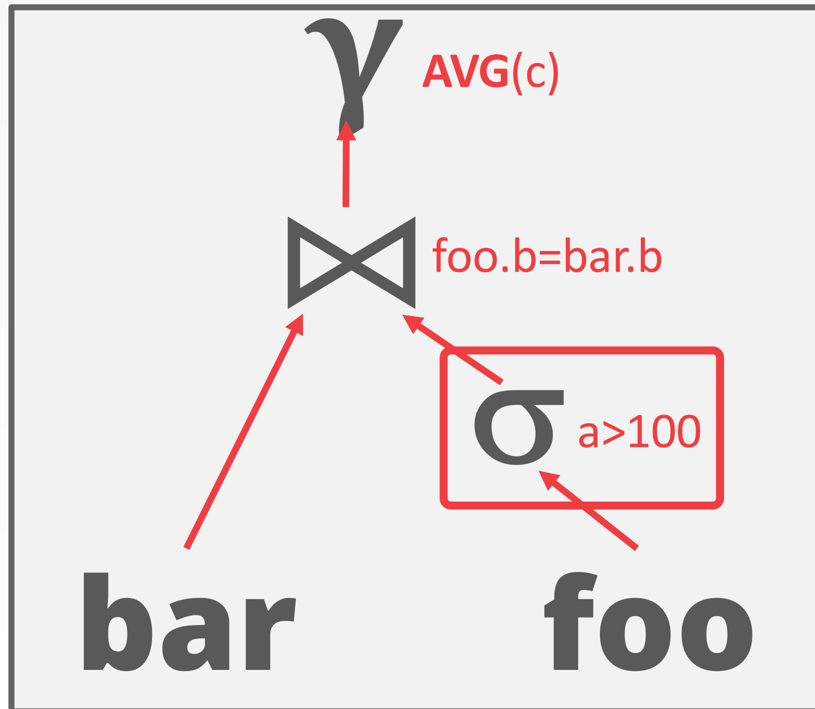

```
SELECT AVG(C)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE a > 100
```

	a	b	c
0			
1			
2			
3			

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
```

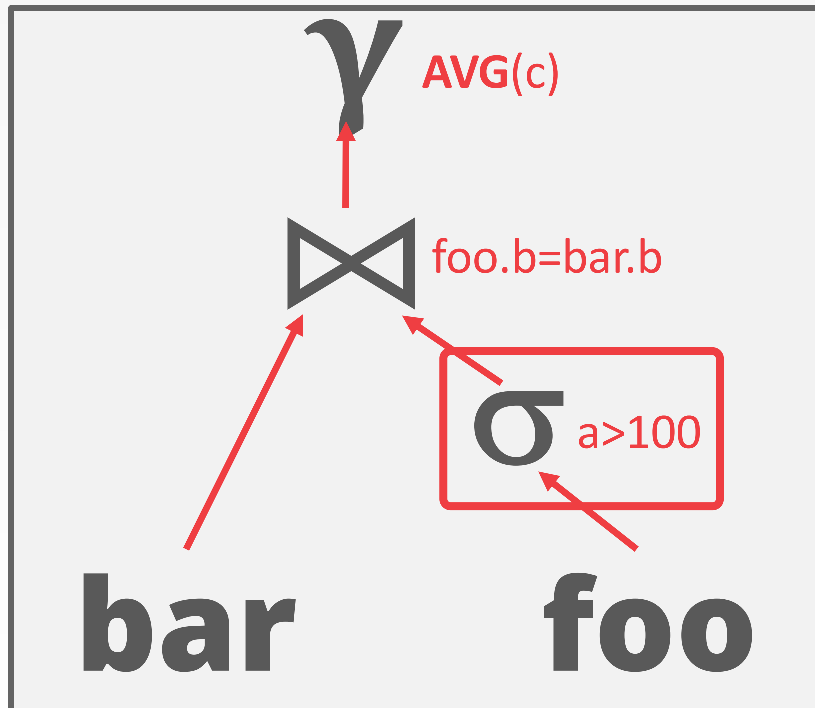


	a	b	c
0			
1			
2			
3			

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
```



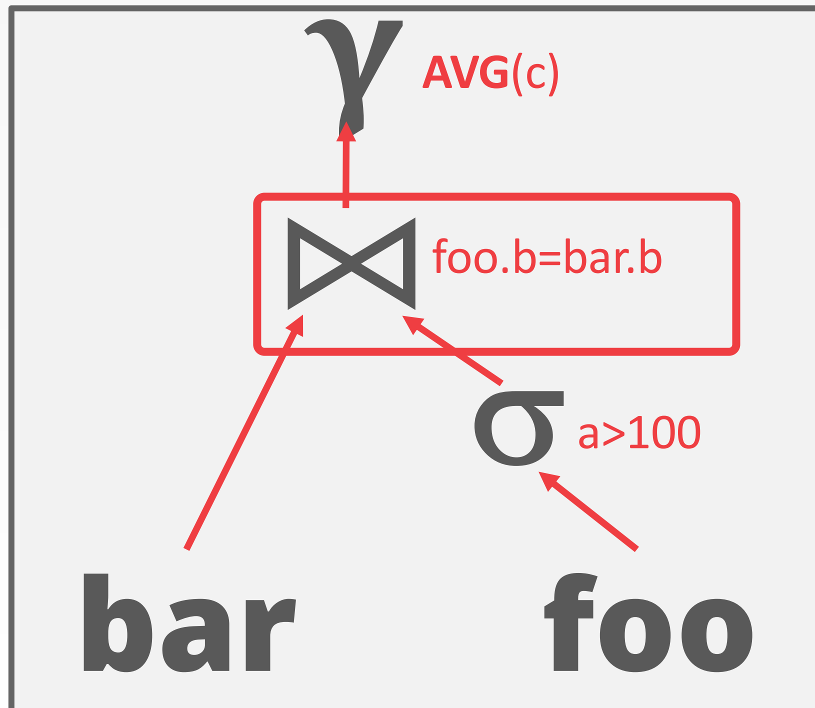
Offsets

	a	b	c
0			
1			
2			
3			

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
```



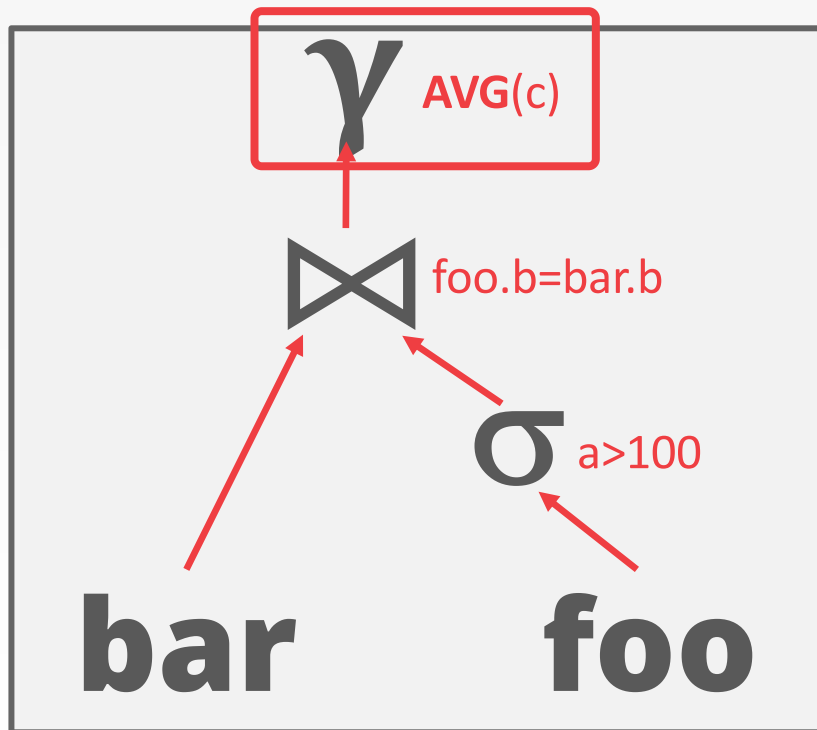
Offsets

Offsets

	a	b	c
0			
1			
2			
3			

LATE MATERIALIZATION

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



Result

Offsets

Offsets

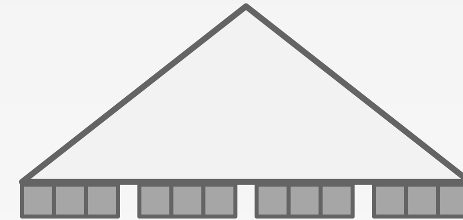
```
SELECT AVG(C)
  FROM foo JOIN bar
    ON foo.b = bar.b
 WHERE a > 100
```

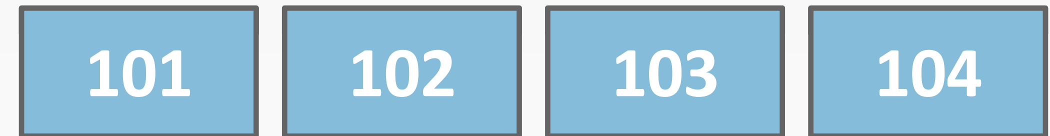
	a	b	c
0			
1			
2			
3			

Georgia Tech

HEAP CLUSTERING

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

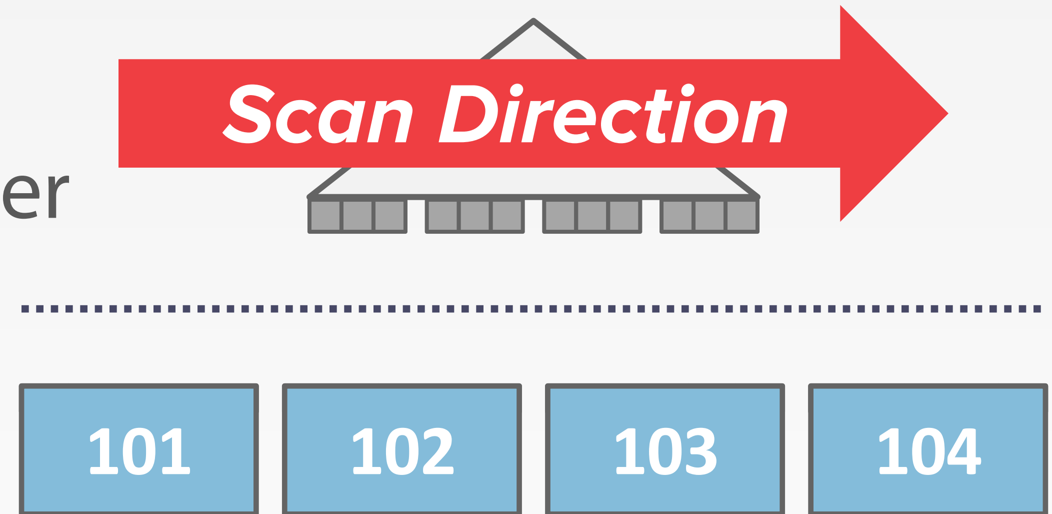




- If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.

HEAP CLUSTERING

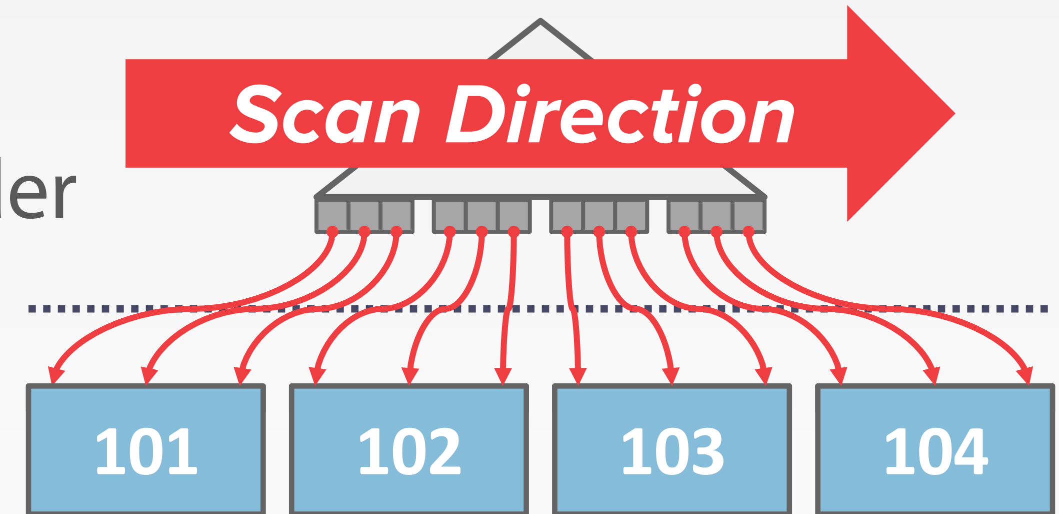
- Tuples are sorted in the heap's pages using the order specified by the clustering index.



- 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 the <u>clustering index</u>.



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

#2: 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

#2: 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

#2: INDEX SCAN

- Suppose that we a single table with 100 tuples and 2 indexes:
  - Index #1: **age**
  - Index #2: **dept**

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

#2: INDEX SCAN

- Suppose that we a single table with 100 tuples and 2 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.

#2: INDEX SCAN

- Suppose that we a single table with 100 tuples and 2 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.

Georgia Tech

#3: 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

#3: MULTI-INDEX SCAN

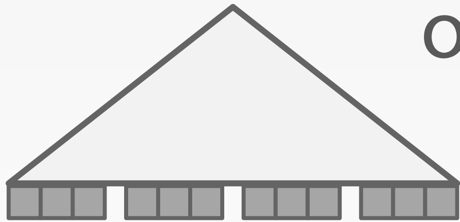
- With an index on **age** and an index on **dept**,
  - We can retrieve the record ids satisfying **age<30** using the first,
  - Then retrieve the record ids satisfying **dept='CS'** using the second,
  - Take their intersection
  - Retrieve records and check **country='US'**.

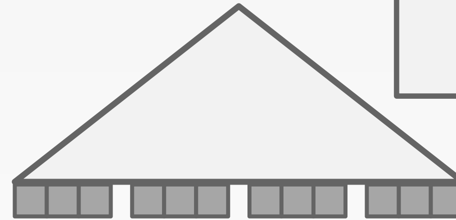
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```

#3: 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'
```

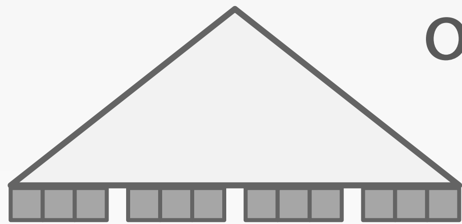


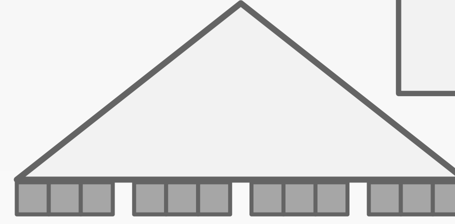


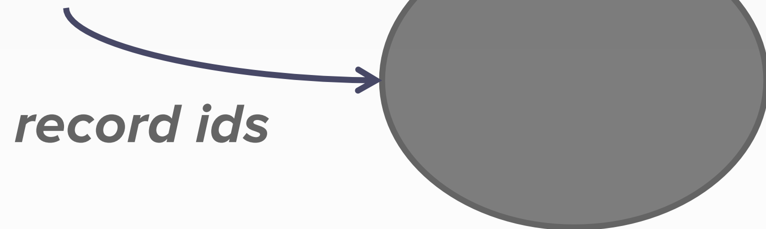
#3: MULTI-INDEX SCAN

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

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



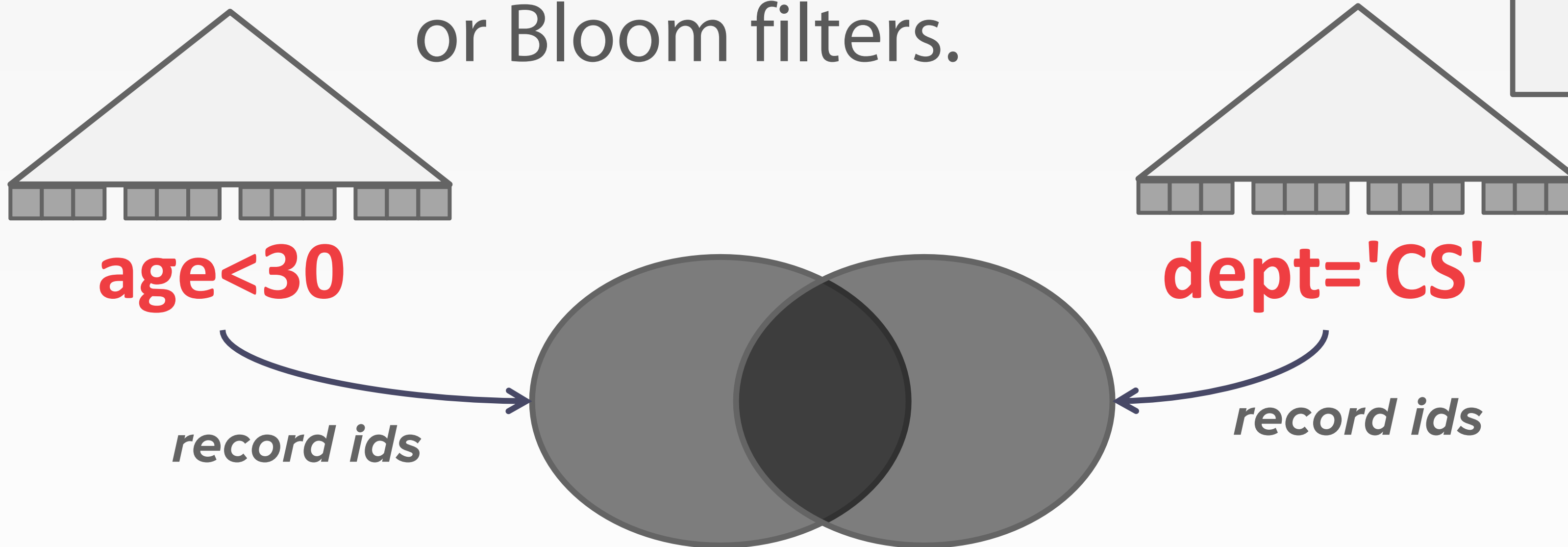




#3: MULTI-INDEX SCAN

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

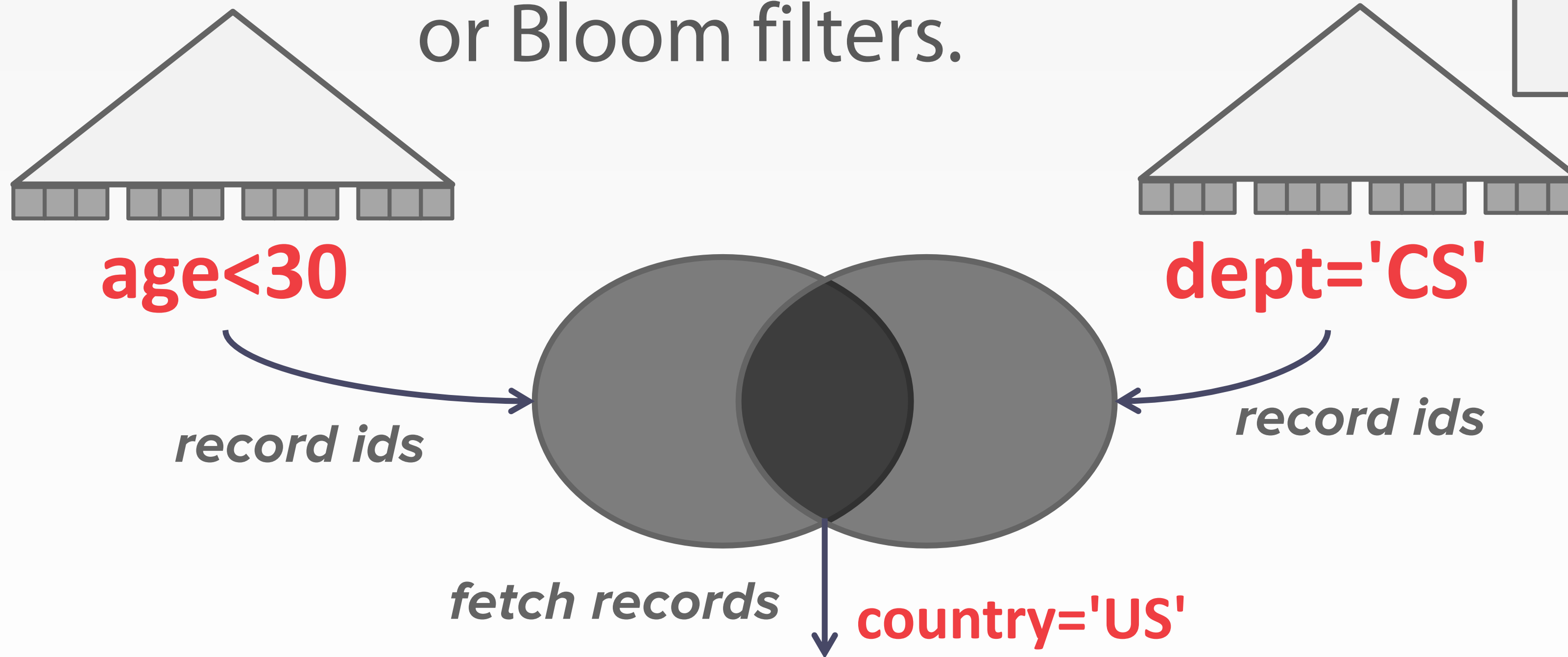
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



#3: MULTI-INDEX SCAN

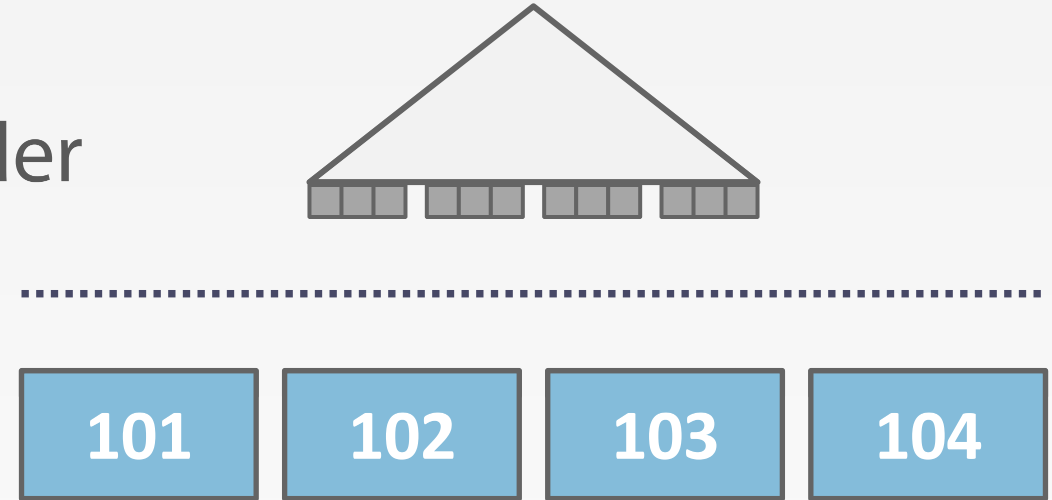
- Set intersection can be done with bitmaps, hash tables, or Bloom filters.

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



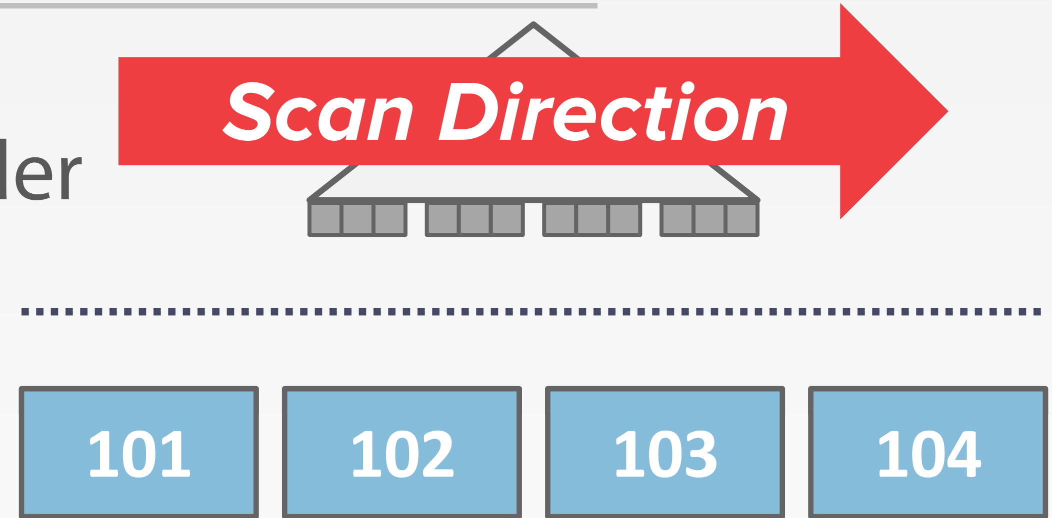
INDEX SCAN PAGE SORTING

- 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.



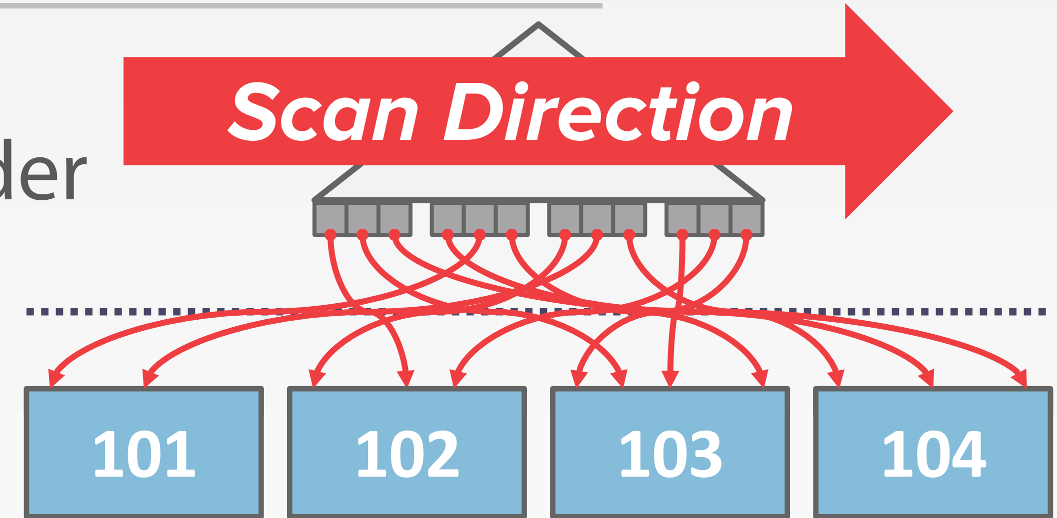
INDEX SCAN PAGE SORTING

- 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.



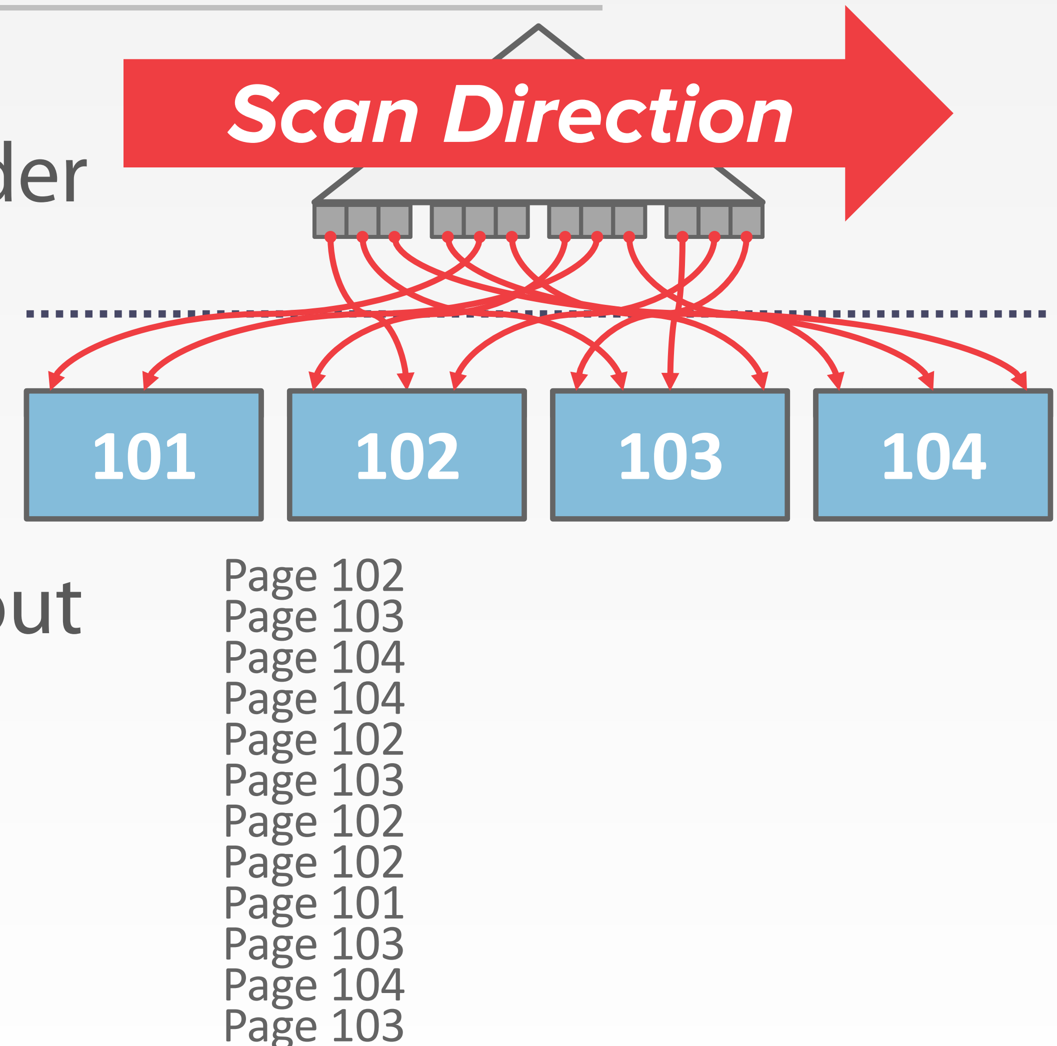
INDEX SCAN PAGE SORTING

- 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.



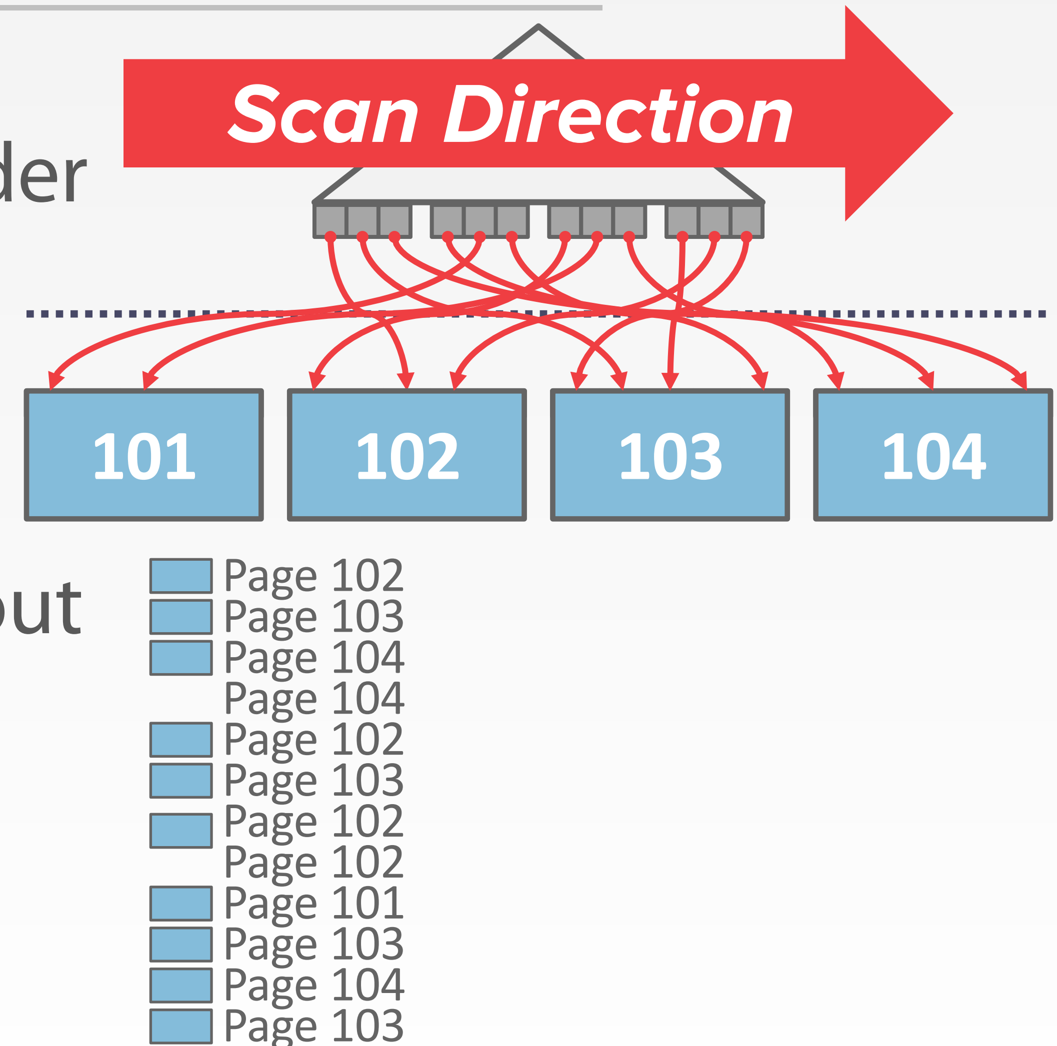
INDEX SCAN PAGE SORTING

- 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.



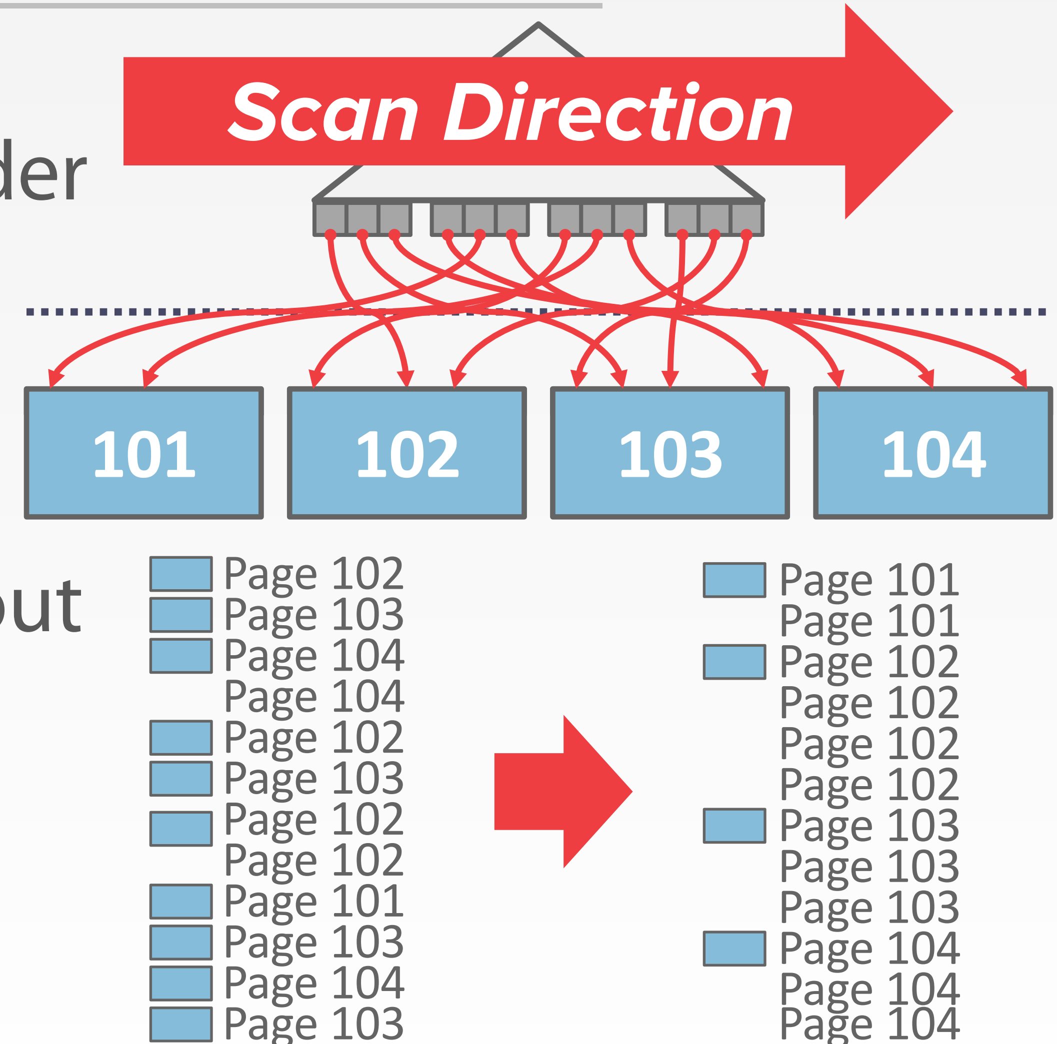
INDEX SCAN PAGE SORTING

- 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.



CLUSTERED VS UNCLUSTERED INDEX

CLUSTERED VS UNCLUSTERED INDEX

- **Q:** What is the difference between a clustered and unclustered index?
  - Clustered: Tuples are stored physically on disk in the same order as the index. Only one per table.
  - Unclustered: Ordered differently. Values are only pointers to the physical tuples.

EXPRESSION EVALUATION

EXPRESSION EVALUATION

- The DBMS represents a **WHERE** clause as an **expression tree**.

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

Georgia Tech

EXPRESSION EVALUATION

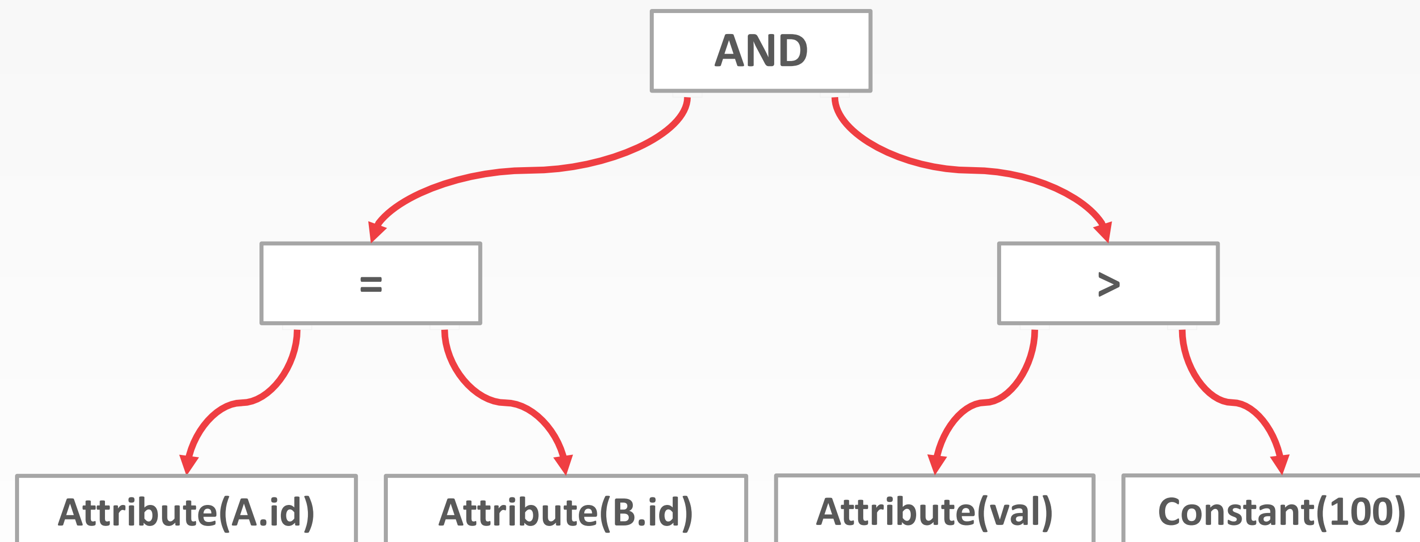
- The DBMS represents a **WHERE** clause as an **expression tree**.

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

EXPRESSION EVALUATION

- The DBMS represents a **WHERE** clause as an **expression tree**.

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



EXPRESSION EVALUATION

- The nodes in the tree represent different expression types:
  - Comparison Operators (**=, <, >, !=**)
  - Logical Operators (**AND, OR**)
  - Arithmetic Operators (**+, -, *, /, %**)
  - Constant Values
  - Tuple Attribute References

EXPRESSION EVALUATION

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

EXPRESSION EVALUATION

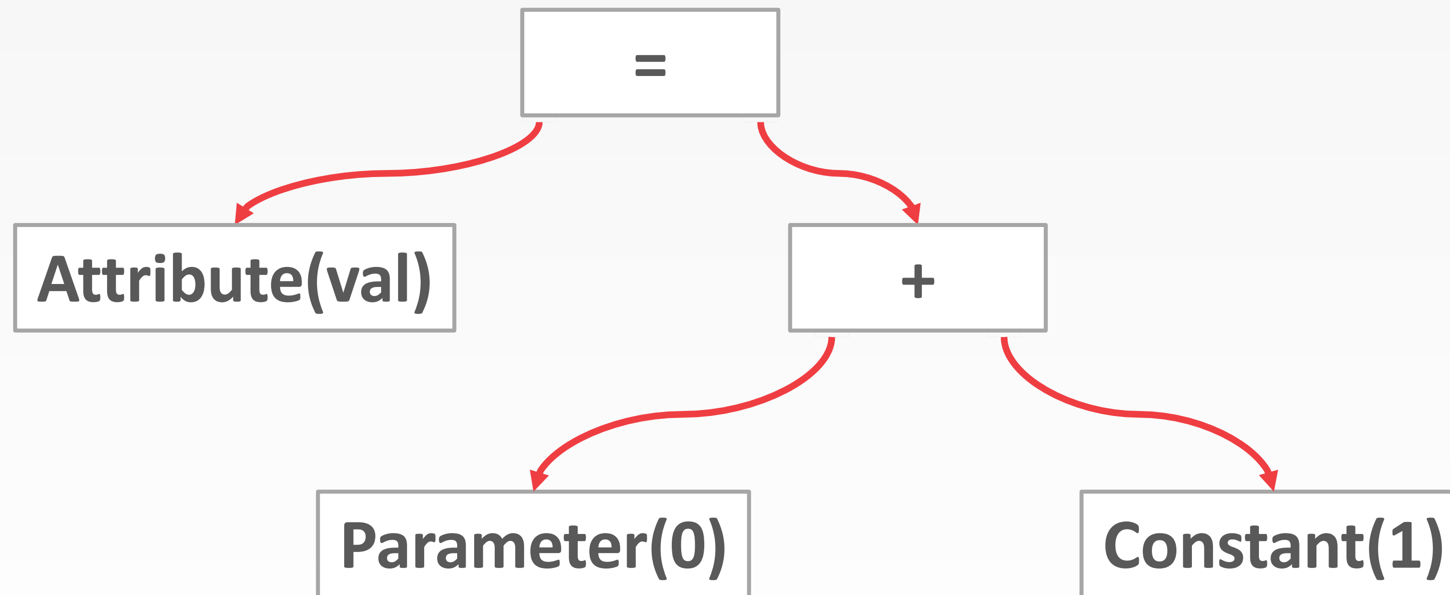
```
SELECT * FROM B
 WHERE B.val = ? + 1
```

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)

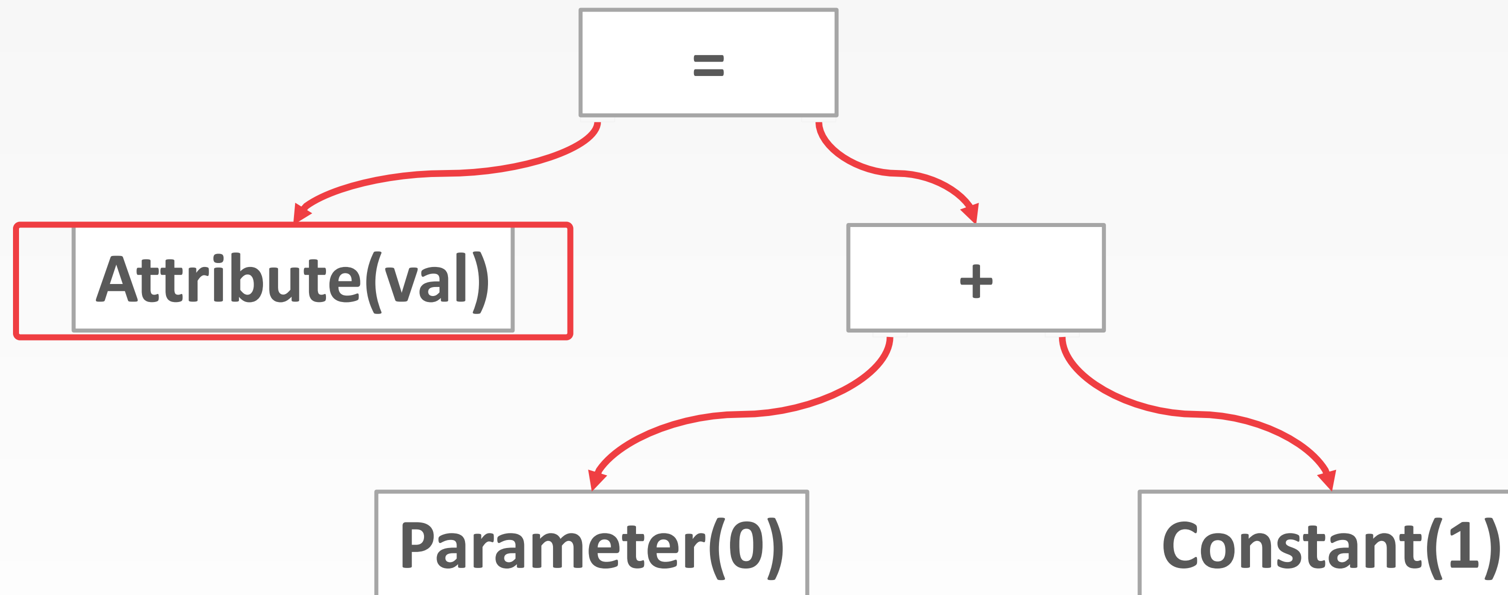


EXPRESSION EVALUATION

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

Execution Context

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

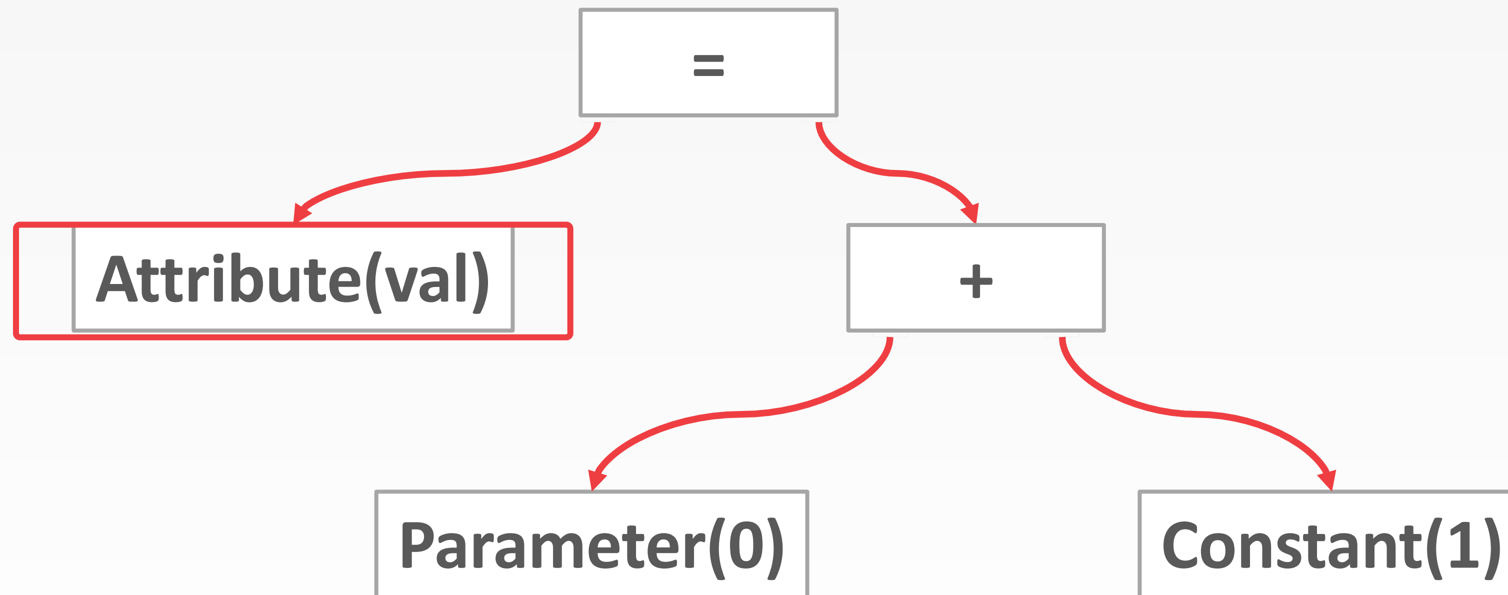


EXPRESSION EVALUATION

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

Execution Context

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

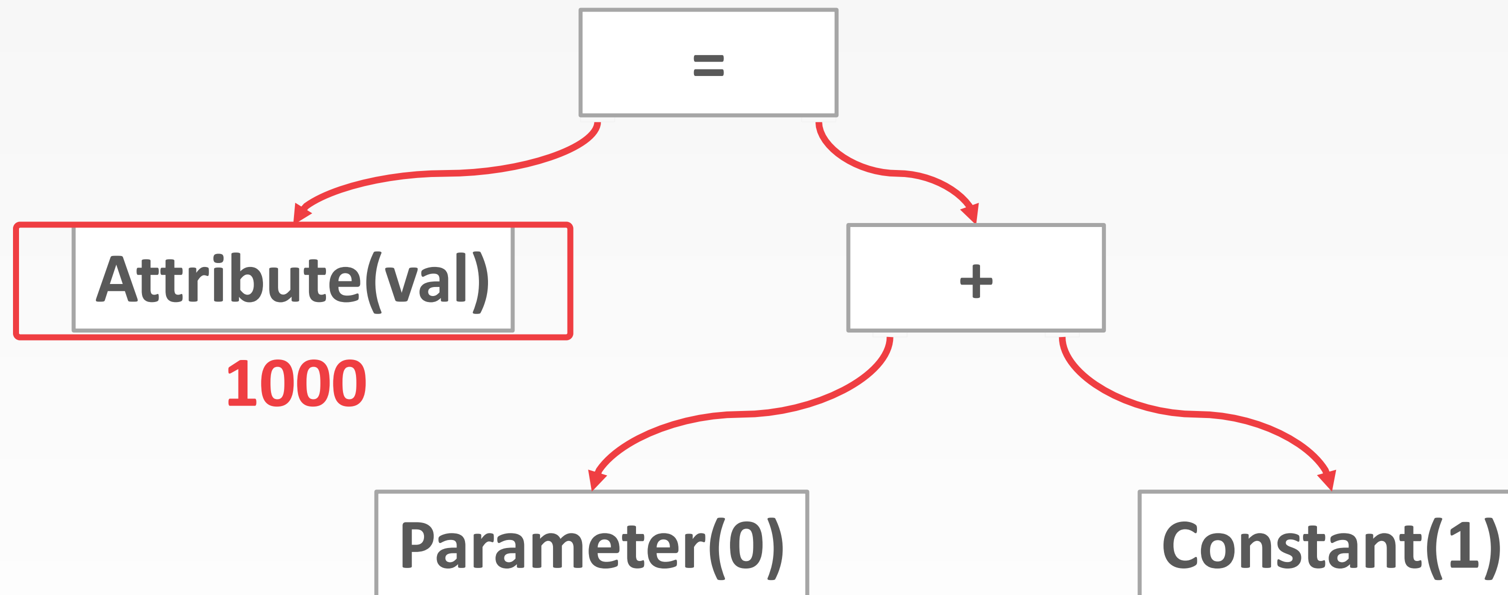


EXPRESSION EVALUATION

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

Execution Context

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

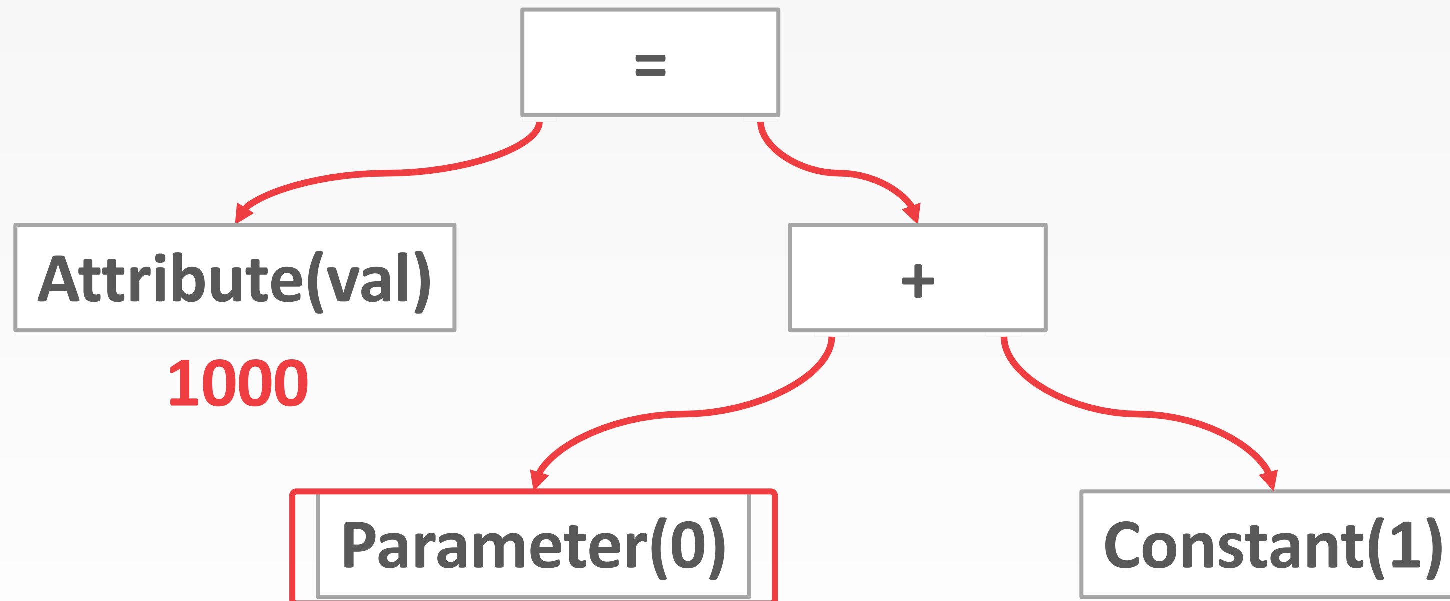


EXPRESSION EVALUATION

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

Execution Context

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

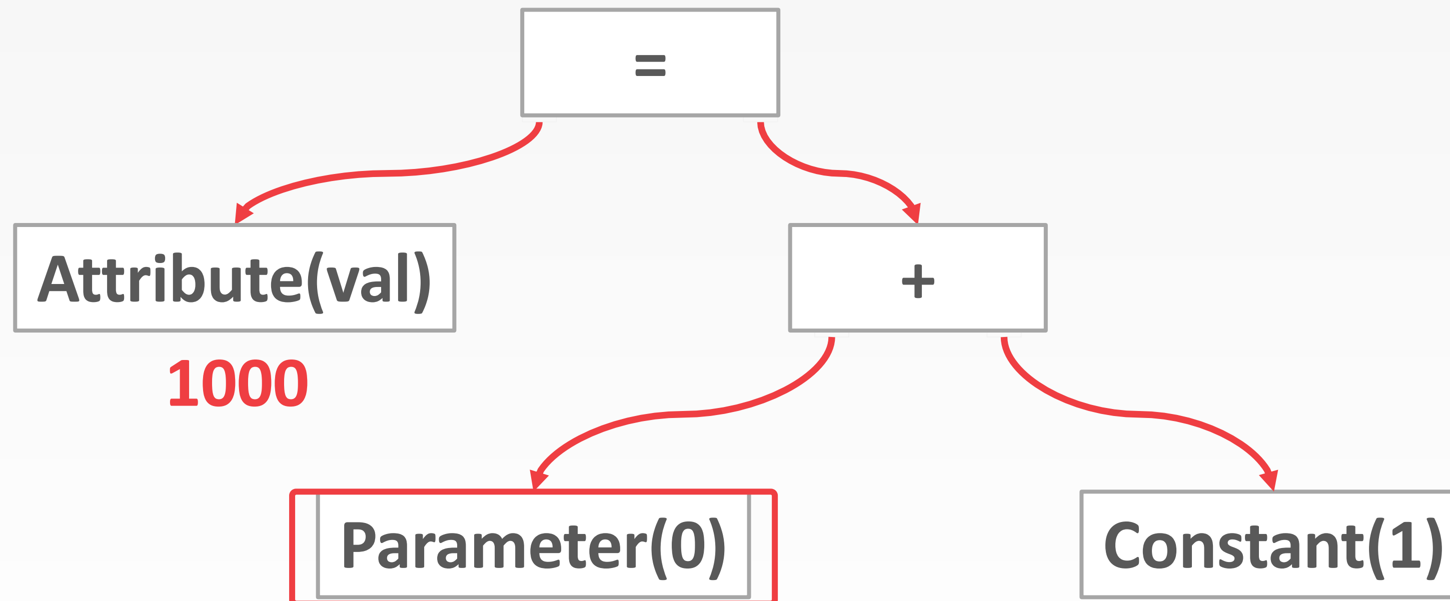


EXPRESSION EVALUATION

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

Execution Context

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

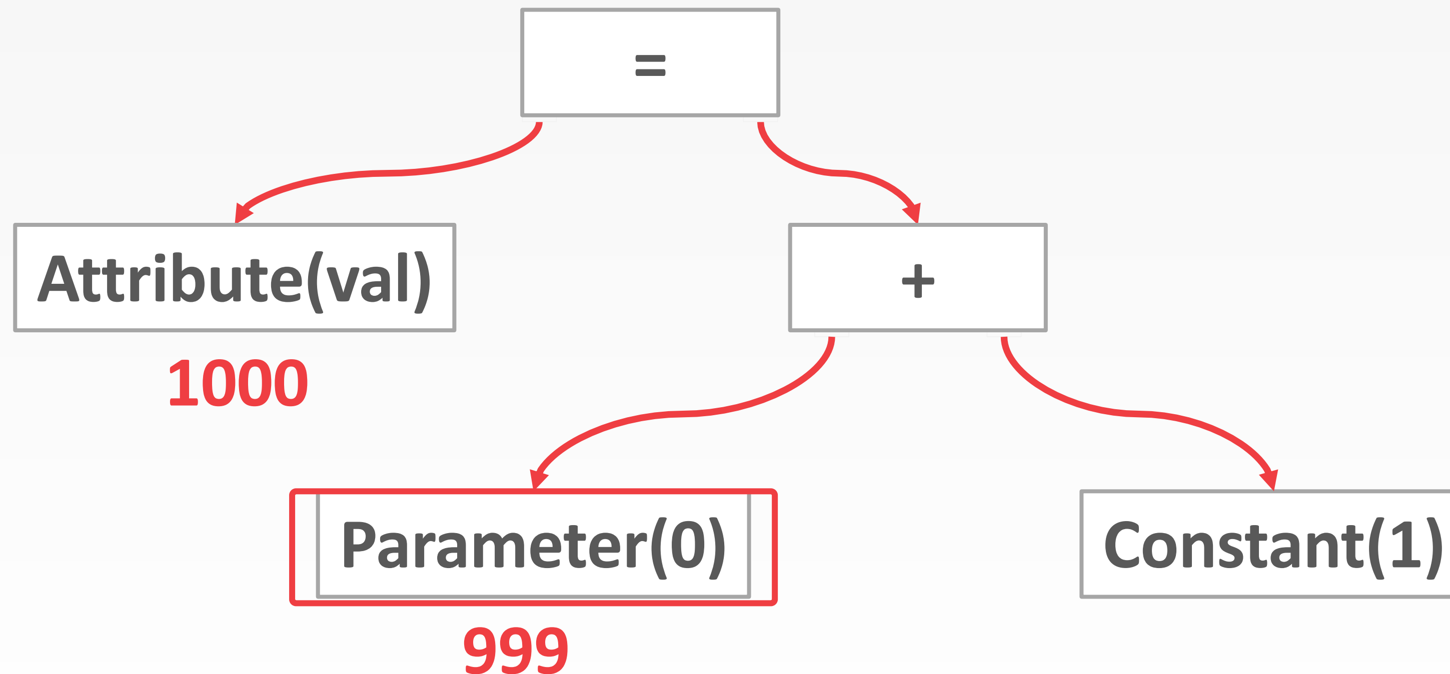


EXPRESSION EVALUATION

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

Execution Context

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

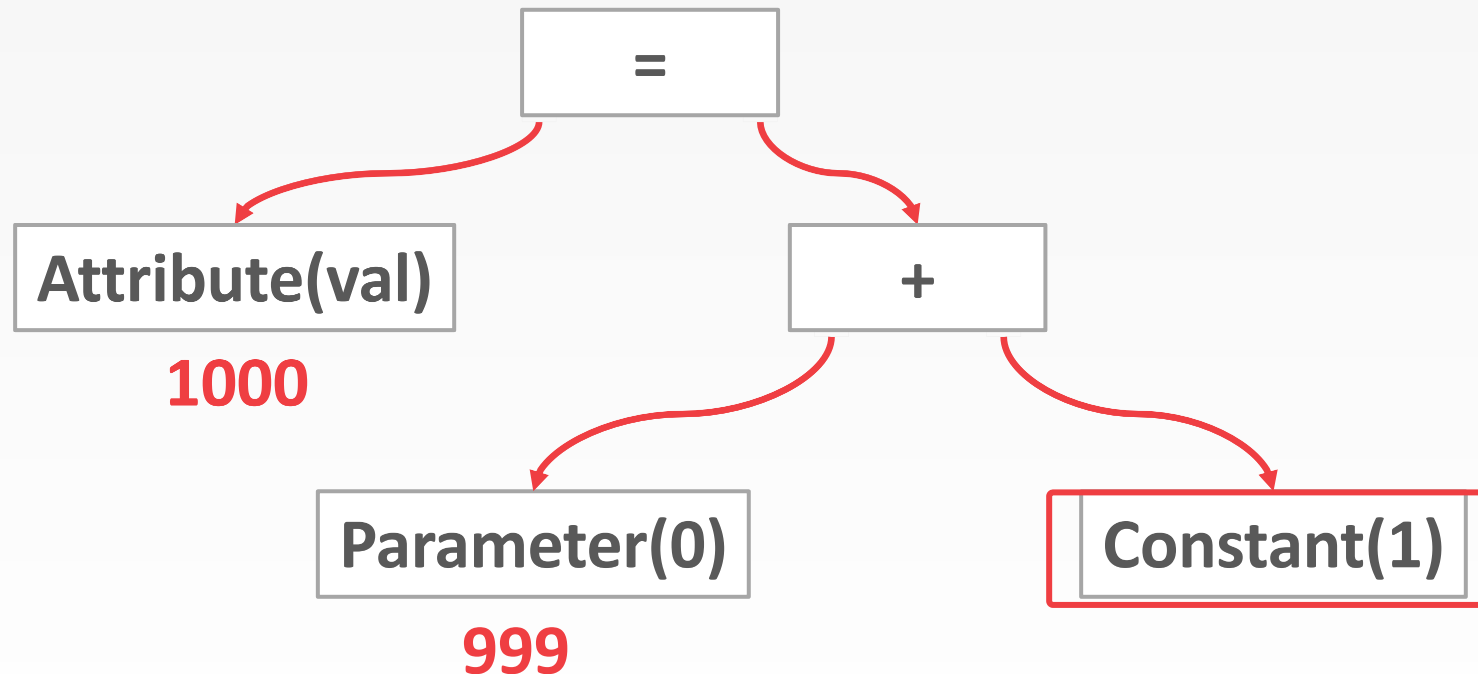


EXPRESSION EVALUATION

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

Execution Context

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

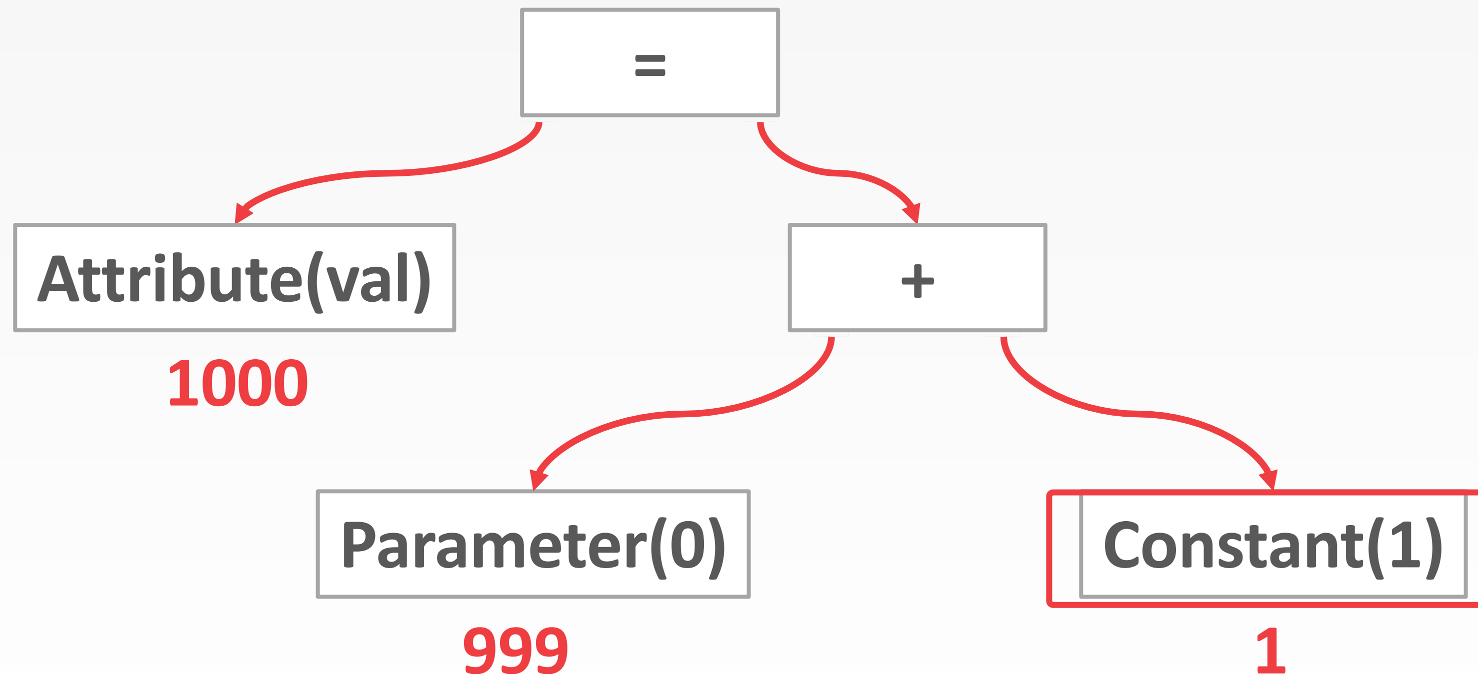


EXPRESSION EVALUATION

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

Execution Context

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

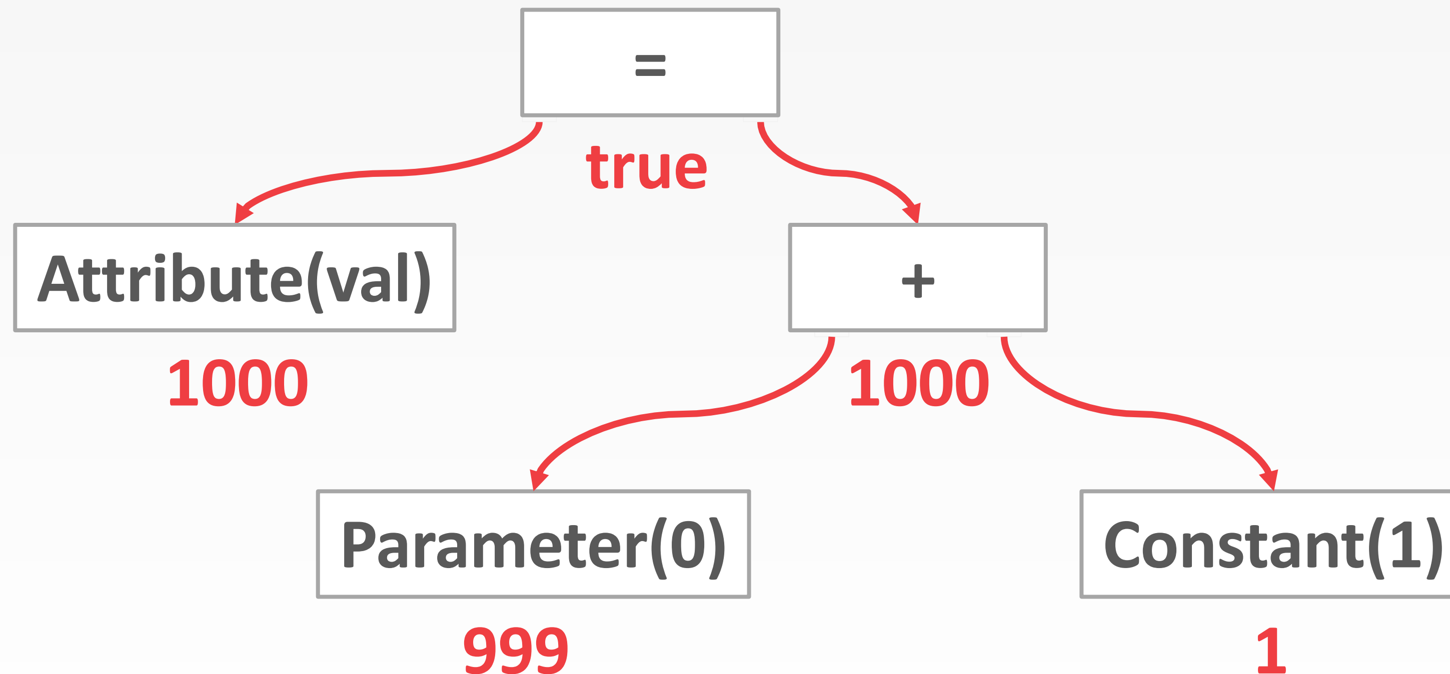


EXPRESSION EVALUATION

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

Execution Context

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



TREE VS STRING REPRESENTATION

Georgia Tech

TREE VS STRING REPRESENTATION

- **Q:** Why are queries and expressions represented as a **tree** as opposed to a **string**?
  - Structural elements simplify manipulation as opposed to a string representation
  - Works well for complex recursive structures (e.g., sub-queries, nested expressions)

SUMMARY

- 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.

VISUAL QUERY EXECUTION ENGINE

TABLE

- BLAZEIT
  - Each table corresponds to a video
  - Each tuple corresponds to a detected object

Object ID	Frame ID	Class	Mask	Features	Content
1	1	Bus	Mask	[F1, F2]	Content

  - Each table should correspond to an unique **entity**
  - Object re-identification

TABLE

- Prior visual query engines
  - Assume that data is already populated (i.e., data created externally typically by humans).
  - In contrast, BLAZEIT automatically populates the data using an off-the-shelf classifier
- Late materialization depending on query
  - Similar to an un-materialized view.
  - This laziness enables several optimizations

EARLY VS LATE MATERIALIZATION

- Early materialization is inefficient
  - **Ad-hoc queries**: Pre-computing all possible features would be expensive. Wasteful for ad-hoc queries since many of the columns with extracted features may never be used.
  - **Online queries**: For queries on live newscasts or broadcast games, it could be faster to execute the queries and classifier directly on the live data.

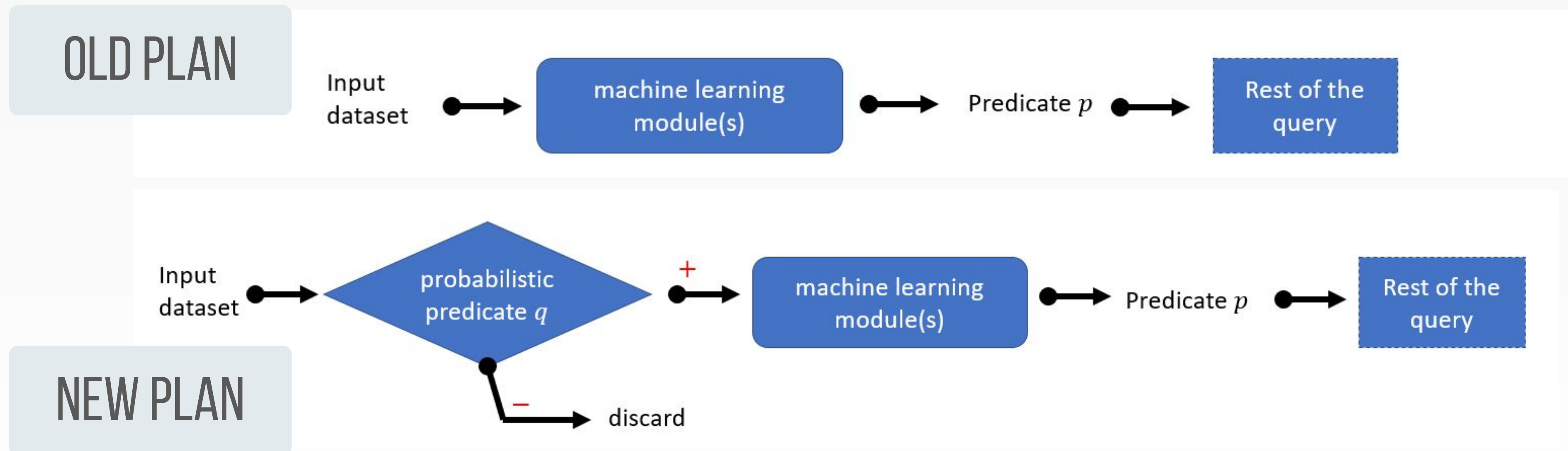
QUERY EXAMPLE

- Imagine having to narrow down a video of passing vehicles captured on a traffic camera to red SUVs that are exceeding the speed limit

```
SELECT cameraID, frameID,
𝒞1(ℱ1(vehBox)) AS vehType,
𝒞2(ℱ2(vehBox)) AS vehColor
FROM (PROCESS inputVideo
PRODUCE cameraID, frameID, vehBox
USING VehDetector)
WHERE vehType=SUV ∧ vehColor=red;
```

#1: FILTERING

- To speed up this query, we can train a filtering classifier that drops frames that are unlikely to satisfy the predicate



#1: FILTERING

- A filtering classifier must be much faster relative to the ML model that it bypasses.
  – It must **filter out** a large portion of the input and have few false negatives
  – Also known as a **probabilistic predicate** or **specialized model**

#1: FILTERING

- Different classifiers are appropriate for different inputs and predicates
  - Linear SVMs
  - Deep neural networks
- Content-based filtering
  - Based on low-level visual features (e.g., avg. color)
  - If an analyst were to query for “red buses”, we could filter the video to have a certain number of red pixels, or a certain level of redness.

#1: FILTERING

- Temporal filtering
  - Filtering based on temporal cues
  - **Example:** Analyst may want to find buses in the scene for at least K frames. In this case, sub-sample the video at a rate of (K–1)/2.
- Spatial filtering
  - Only regions of interest (ROIs) of the scene are considered.
  - ROI can be used to train smaller faster models

#2: SAMPLING

- When the query contains a tolerated error rate, we can sample from the video
  - **Example:** User is interested in some statistic over the data, such as average number of cars per frame
  - Only populate a small number of tuples (or not populate them at all) for faster execution.
  - In cases where the filtering classifier is accurate enough, we can return the answer directly from the filtering classifier

#2: SAMPLING

- In a cardinality-limited filtering query, user is typically interested in a **rare event**
  - **Example:** A clip containing at least one bus and five cars
  - Intuition is to bias the search towards regions of the video that likely contain the event
  - Train a filtering classifier to bias which frames to sample.

QUERY EXECUTION: SUMMARY

- Key Optimizations
  - Filtering, Sampling
- Query processing model
  - Tuple-at-a-time
- Access methods
  - Sequential scan with a complex predicate
  - Index scan: Filtering classifier (e.g., vehType)
  - Multi-Index scan: Spatial access (e.g., R-trees)

NEXT LECTURE

- Given an error budget, how to select between different filtering classifiers?
  - Query optimization