Intro to Databases (COMP_SCI 339)

10 Query Execution

Northwestern University

WINTER 2024

Andrew Crotty

ADMINISTRIVIA

Project #3 is due Sunday 2/18 @ 11:59pm

Exam #2 will be on 2/14 from 3:30-4:50pm

EXAM #2

Who: You

What: Exam #2

Where: Here

When: Wednesday 2/14 from 3:30-4:50pm

What to bring:

→ Pencil or pen with dark-colored ink

→ <u>**One**</u> double-sided 8.5x11" page of handwritten notes

TODAY'S AGENDA

Processing Models
Access Methods
Modification Queries
Expression Evaluation
Parallel Execution

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.

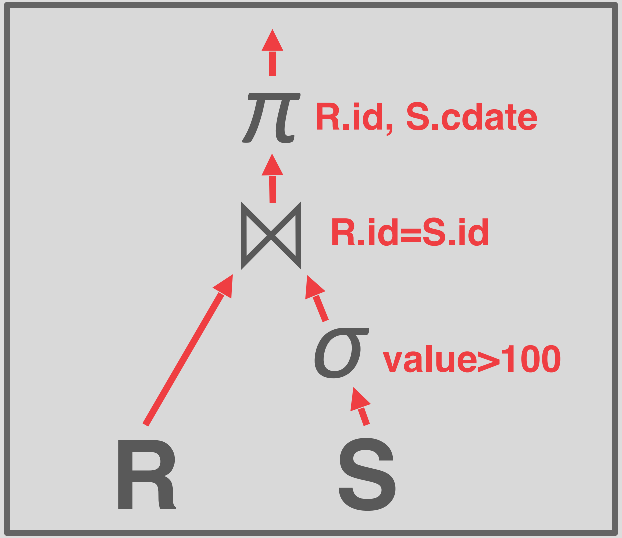
→ On each invocation, the operator returns either a single tuple or a **null** marker if there are no more tuples.

→ 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

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



ITERATOR MODEL

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

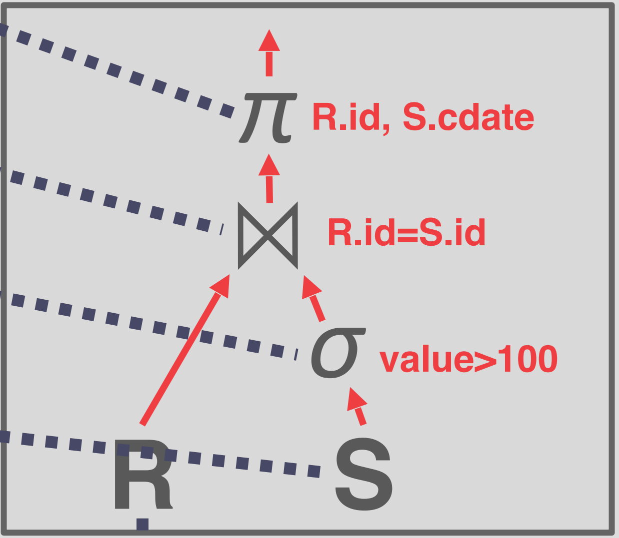
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

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

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

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



ITERATOR MODEL

Next()

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

Next()

for t_1 in left.Next():
  buildHashTable(t_1)
for t_2 in right.Next():
  if probe(t_2): emit($t_1 \bowtie t_2$)

Next()

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

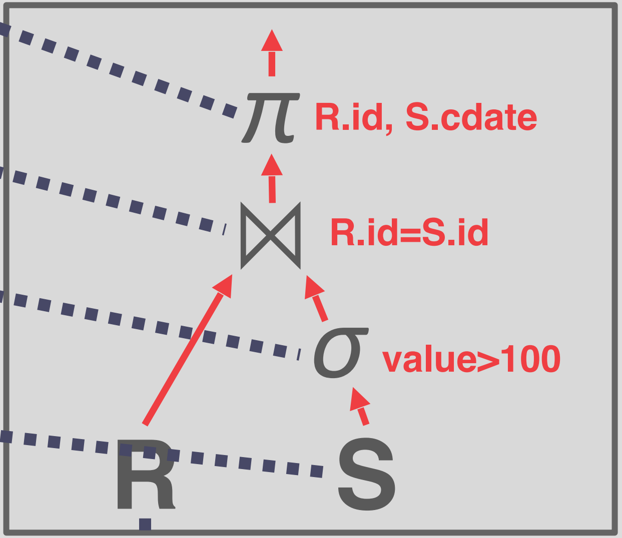
Next()

for t in R:
  emit(t)

Next()

for t in S:
  emit(t)

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



ITERATOR MODEL

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

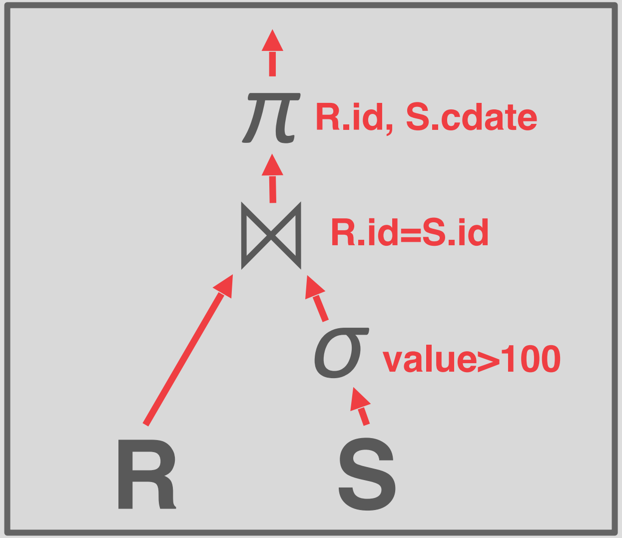
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
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))
```

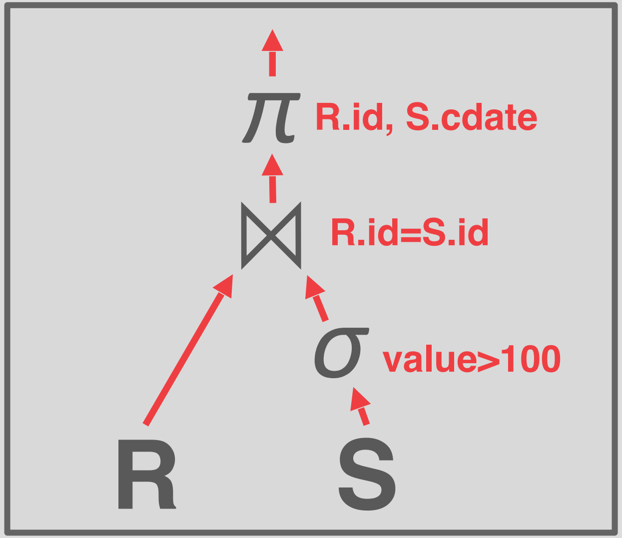
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
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))
```

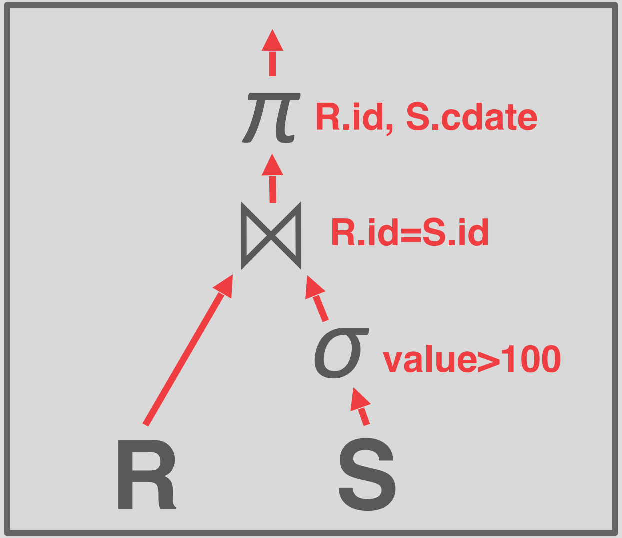
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

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

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

```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

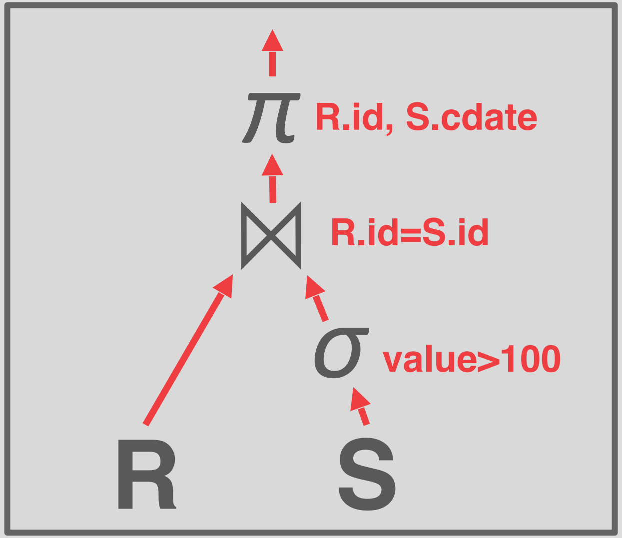
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
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 t_1 in left.Next():
  buildHashTable(t_1)
for t_2 in right.Next():
  if probe(t_2): emit($t_1 \bowtie t_2$)

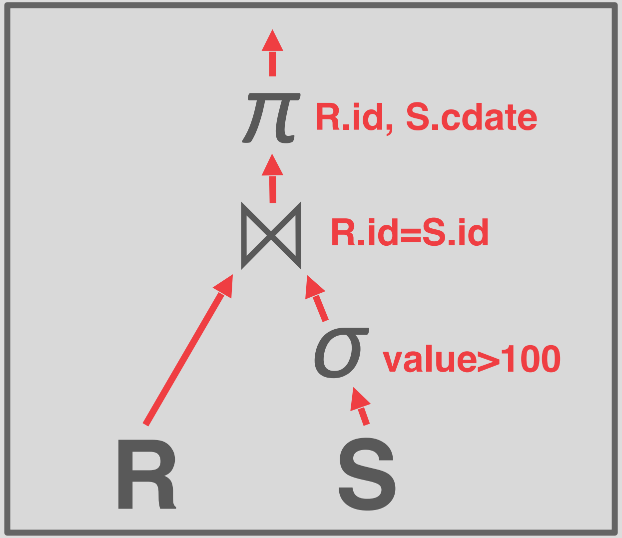
for t in child.Next():
  if evalPred(t): emit(t)

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

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)
```

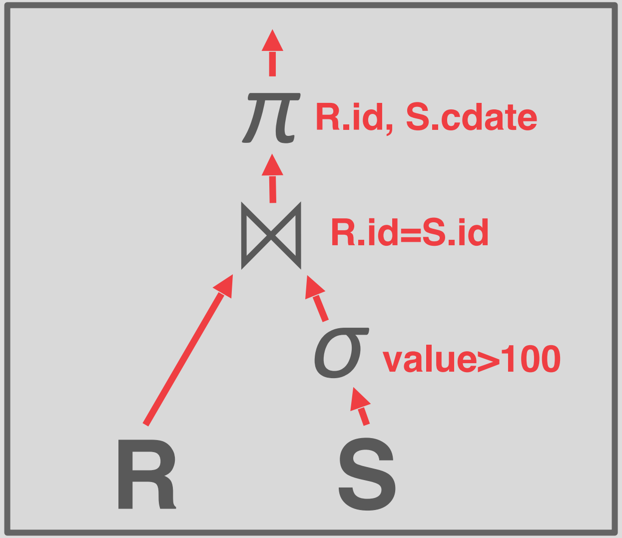
Single Tuple

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

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)
```

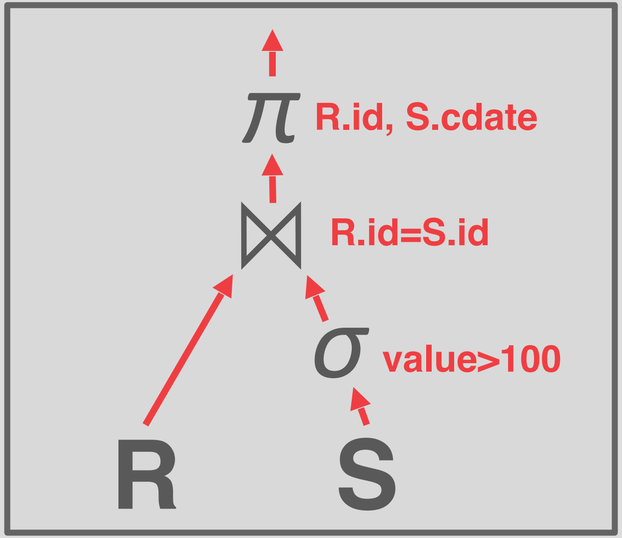
```
for t in child.Next():
  if evalPred(t): emit(t)
```

3

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

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

```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))
```

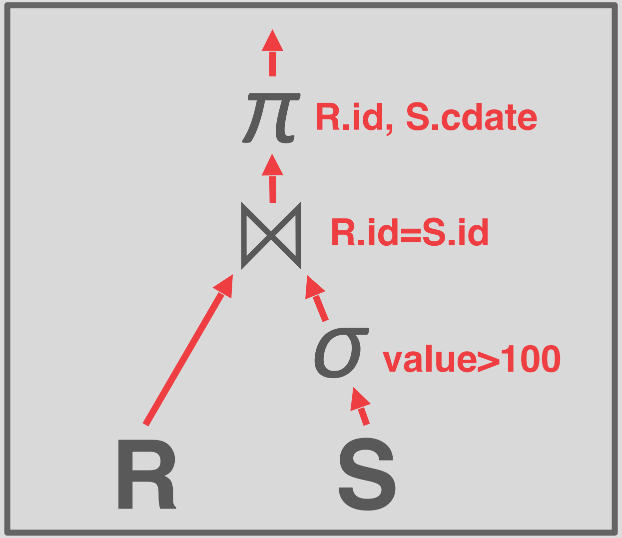
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

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)
```

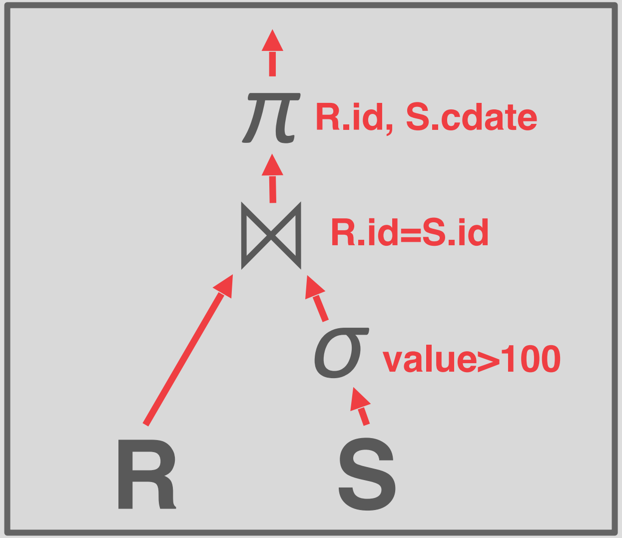
4

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

5

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

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)
```

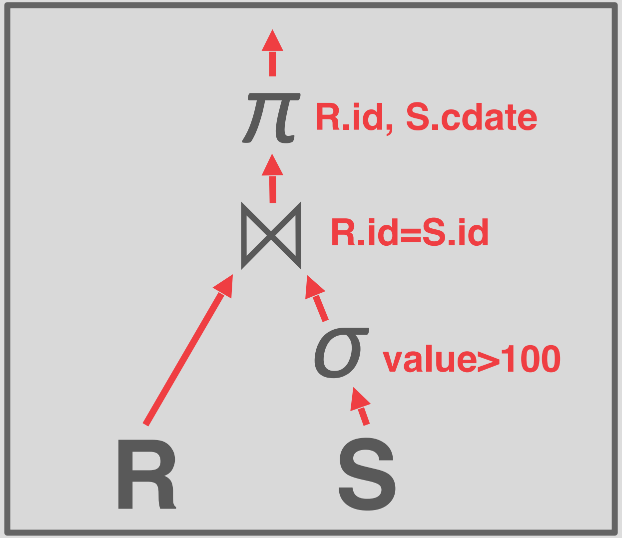
3

```
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

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)
```

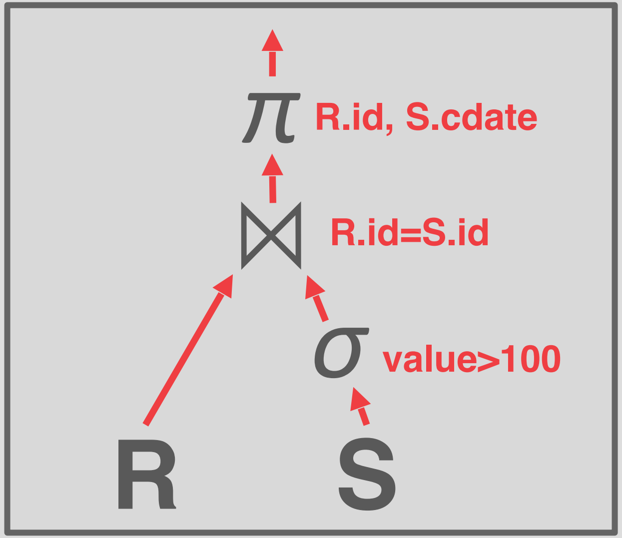
4

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

5

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

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



ITERATOR MODEL

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

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

Output control is easy to implement.

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
```

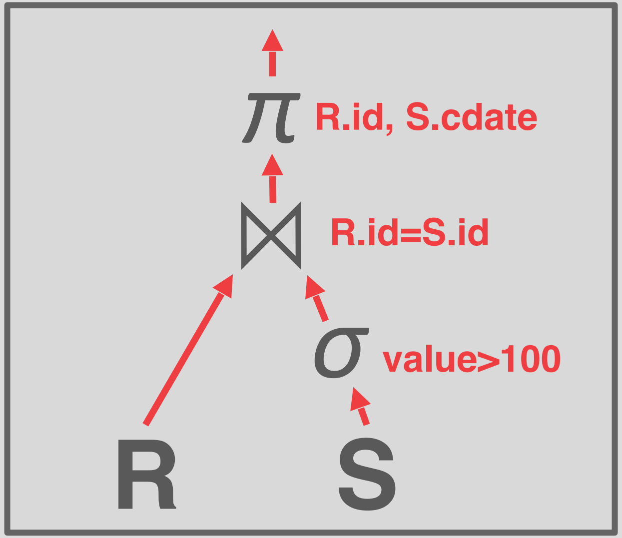
```
out = [ ]
for t₁ in left.Output():
  buildHashTable(t₁)
for t₂ in right.Output():
  if probe(t₂): out.add(t₁⋈t₂)
return out
```

```
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
```

```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
```

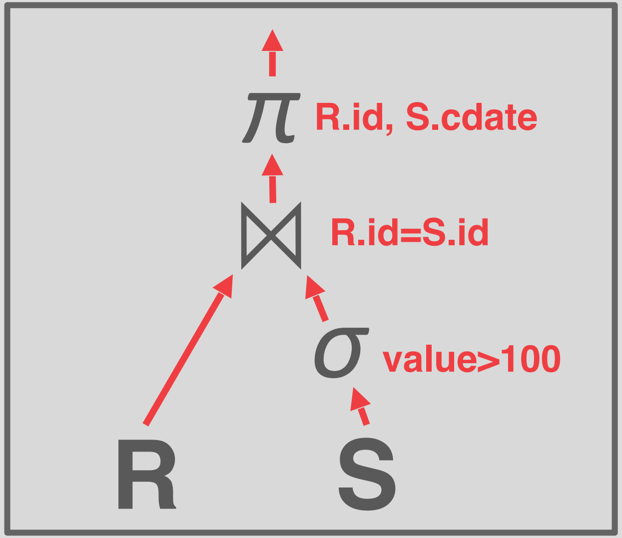
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
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
```

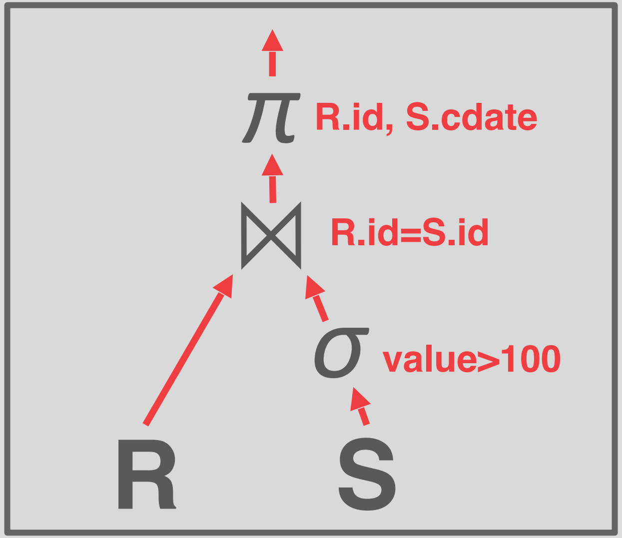
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

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
```

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

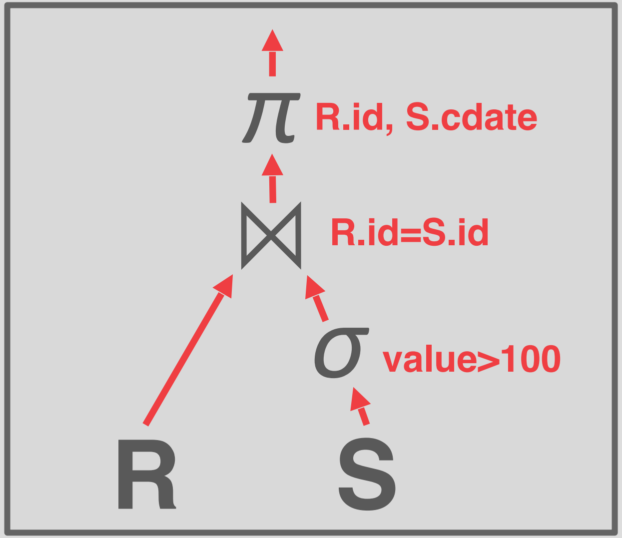
All Tuples

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
```

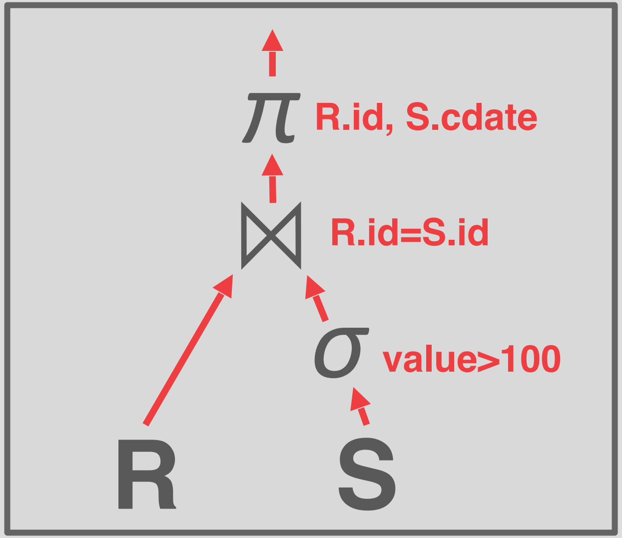
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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

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
```

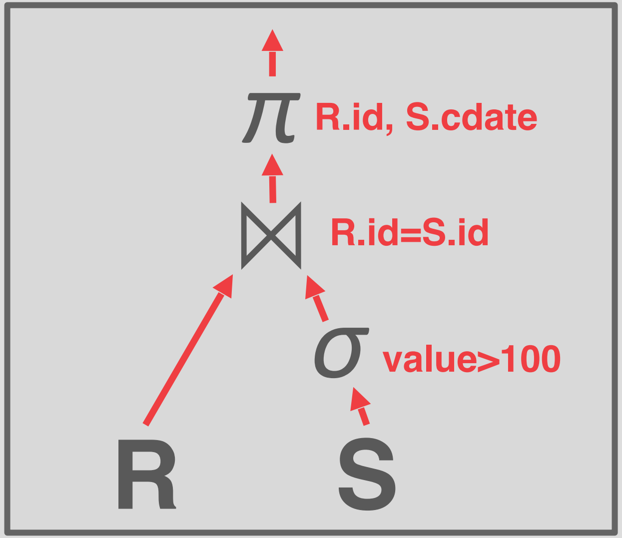
4

```
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
```

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
```

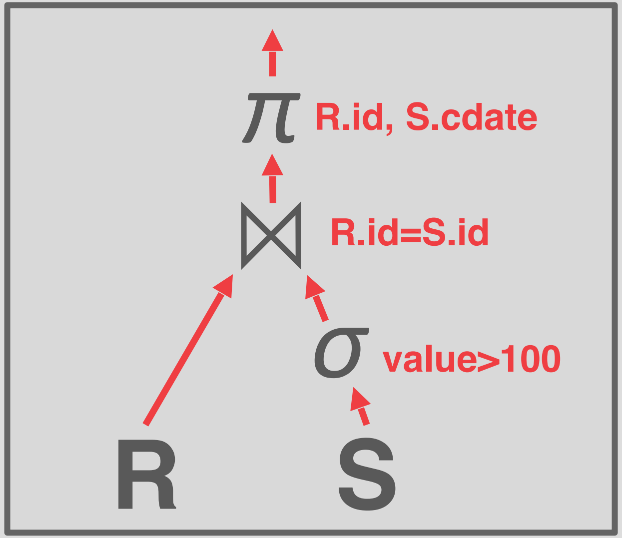
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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
```

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, 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)
```

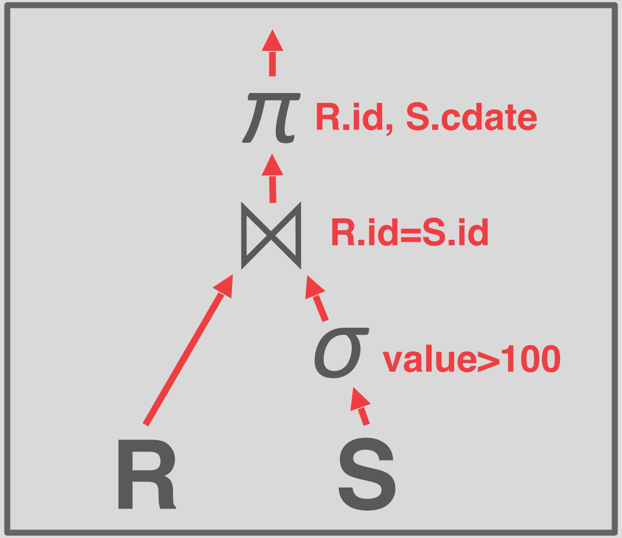
```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

```
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)
```

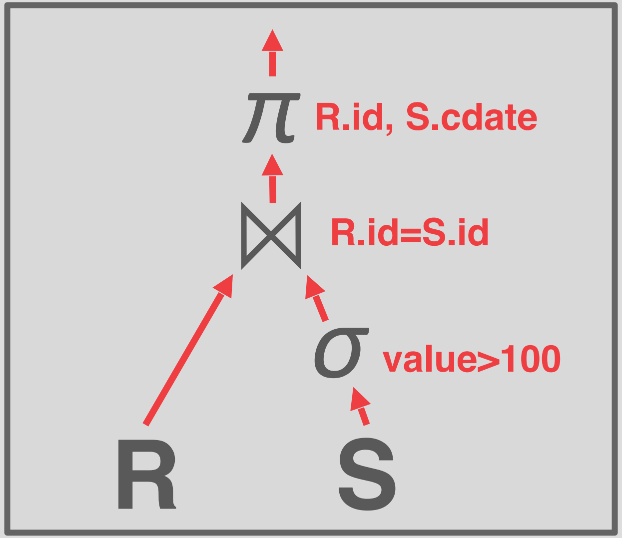
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

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

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)
```

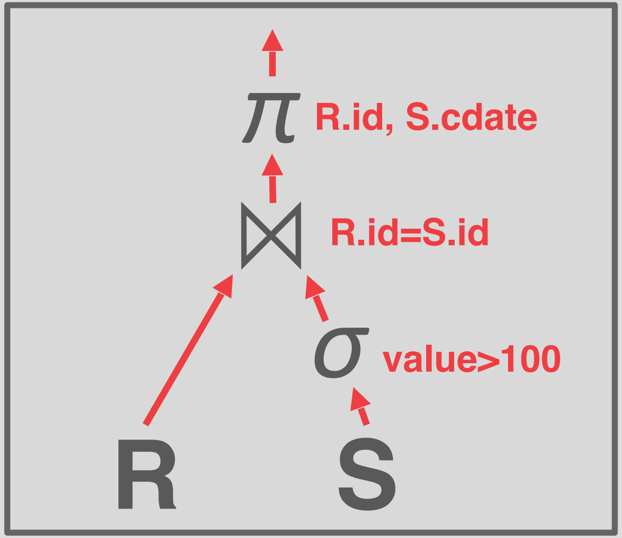
Tuple Batch

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

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)
```

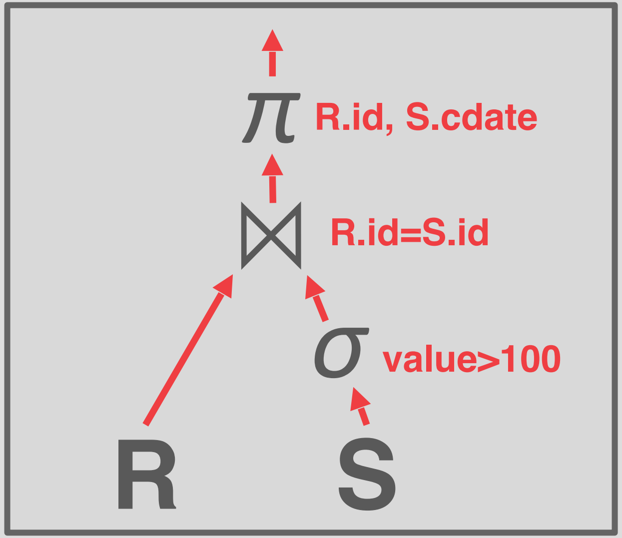
3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

Tuple Batch

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

5
```
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

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

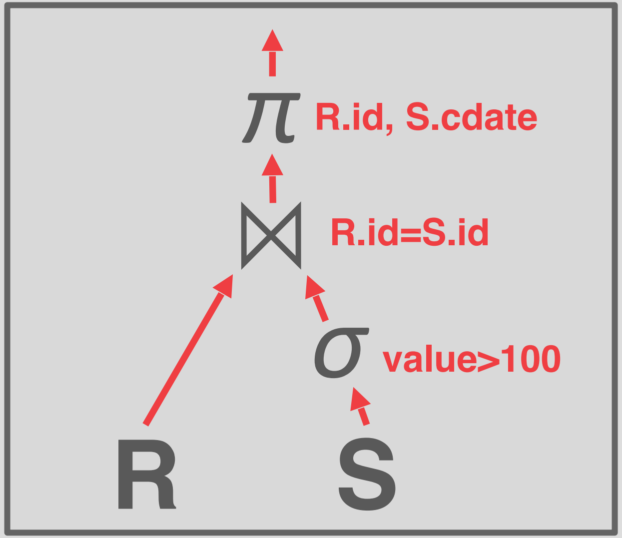
→ Start with leaf nodes and push data to their parents.
→ Allows for tighter control of caches/registers in pipelines.

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
```

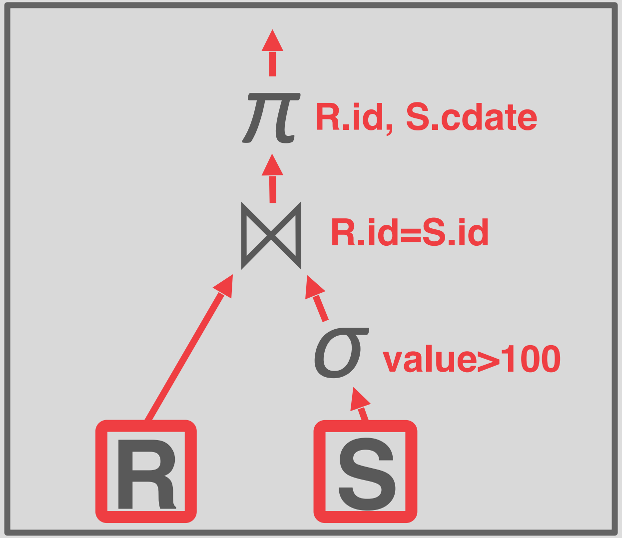


ACCESS METHODS

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.

```
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.

ZONE MAPS

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

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

Original Data

val
100
200
300
400
400

Zone Map

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

INDEX SCAN

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

Which index to use depends on:
→ What attributes the index contains
→ What attributes the query references
→ The attribute's value domains
→ Predicate composition
→ Whether the index has unique or non-unique keys

INDEX SCAN

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

Lecture #11

Which index to use depends on:
→ What attributes the index contains
→ What attributes the query references
→ The attribute's value domains
→ Predicate composition
→ Whether the index has unique or non-unique keys

INDEX SCAN

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

→ Index #1: **age**

→ Index #2: **dept**

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

INDEX SCAN

Suppose that we have 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.

```
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 have 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.

Examples:

→ DB2 Multi-Index Scan
→ PostgreSQL Bitmap Scan
→ MySQL Index Merge

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.

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
```

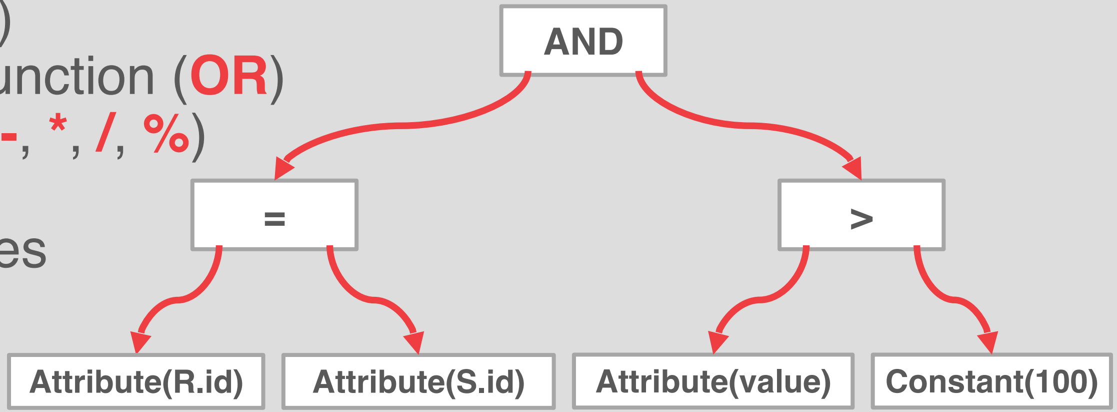
EXPRESSION EVALUATION

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

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



EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

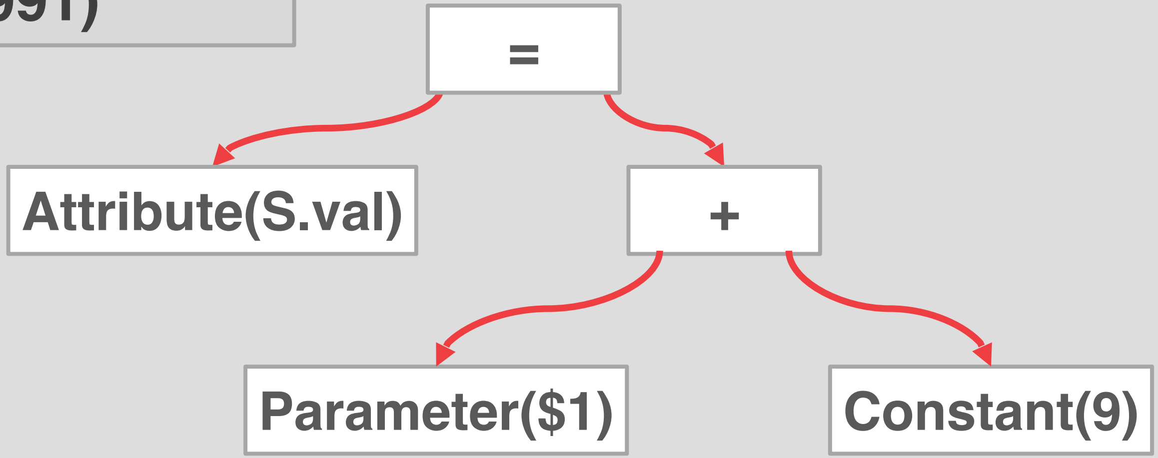
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



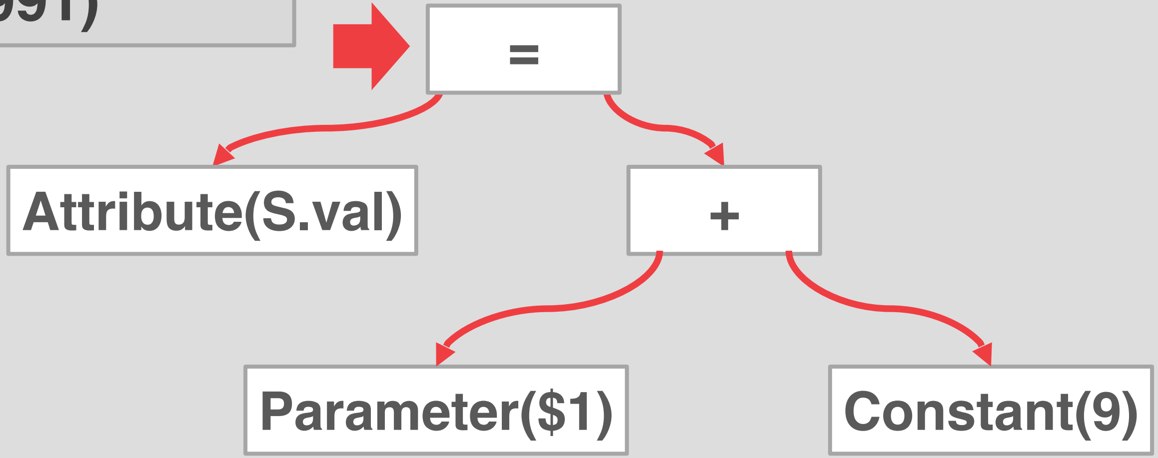
EXPRESSION EVALUATION

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

EXECUTE f(991)

Execution Context

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



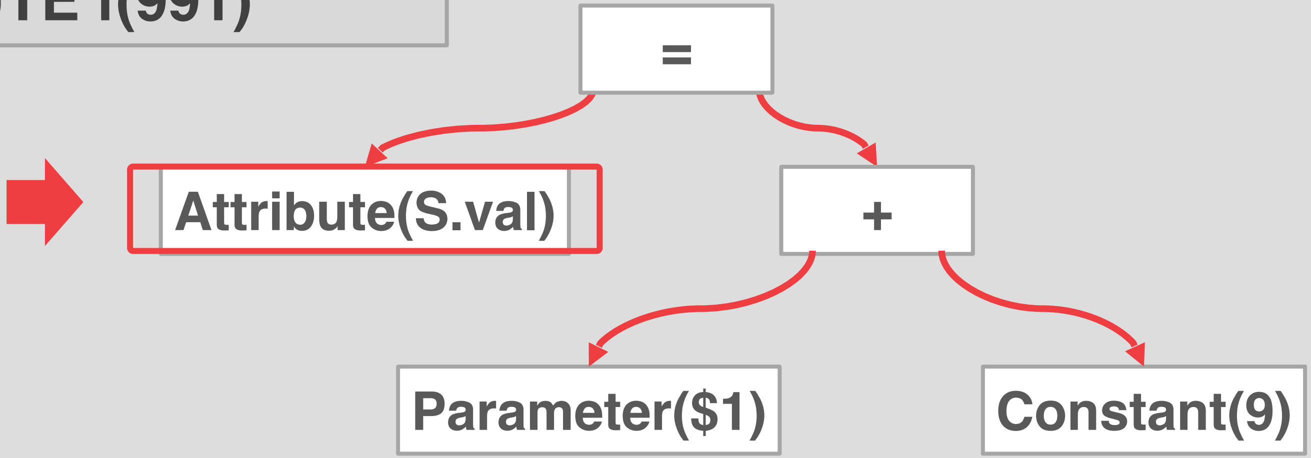
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



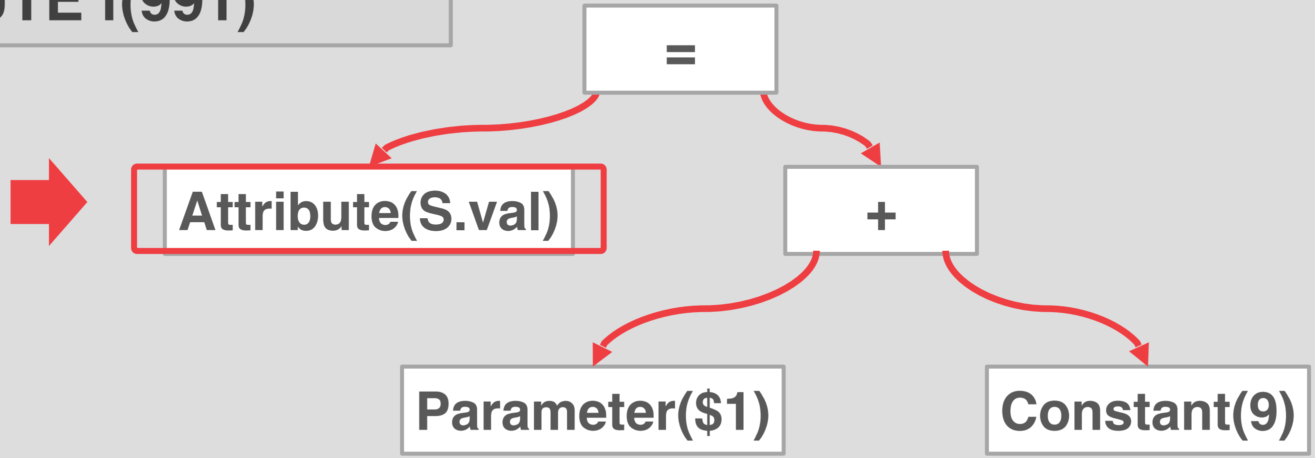
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



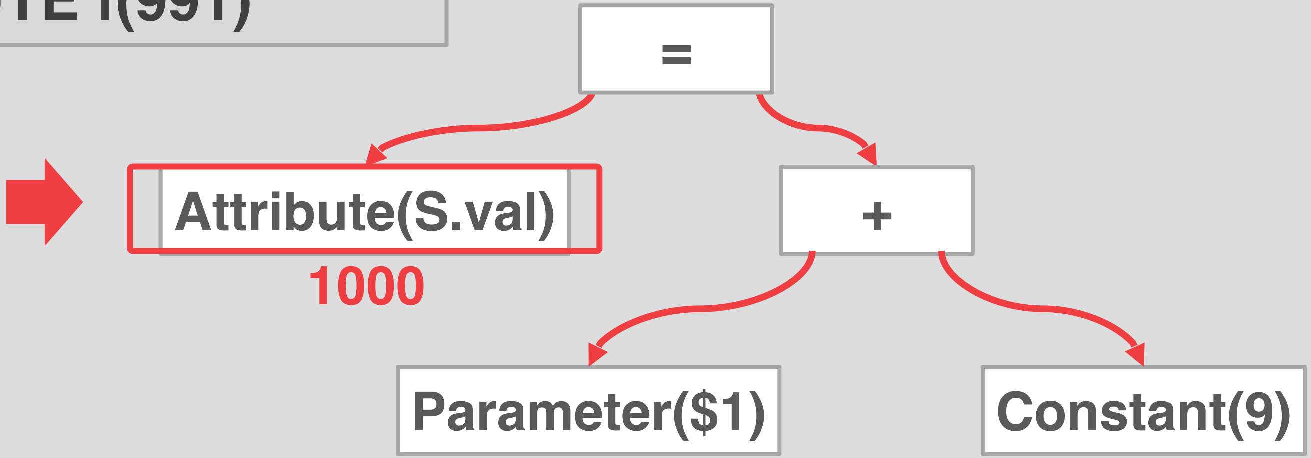
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



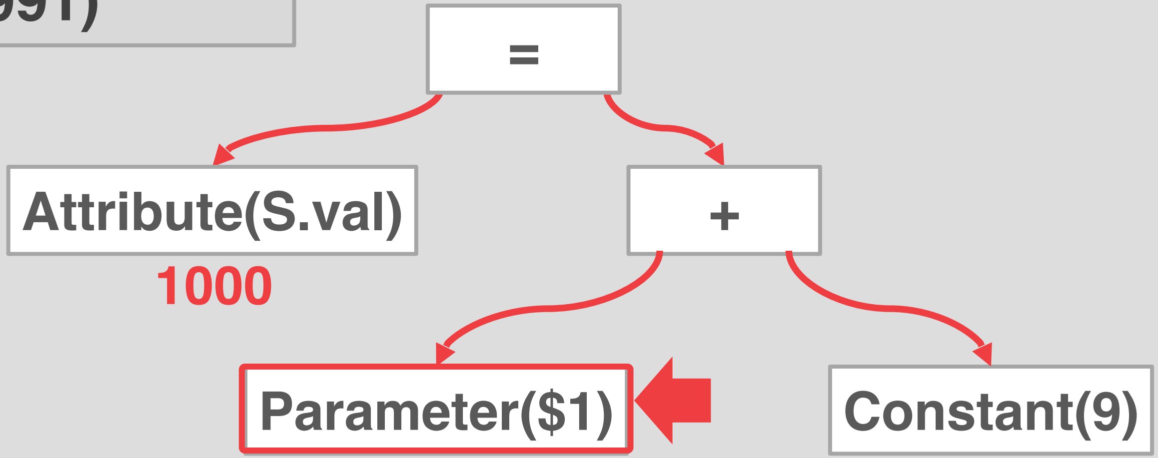
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



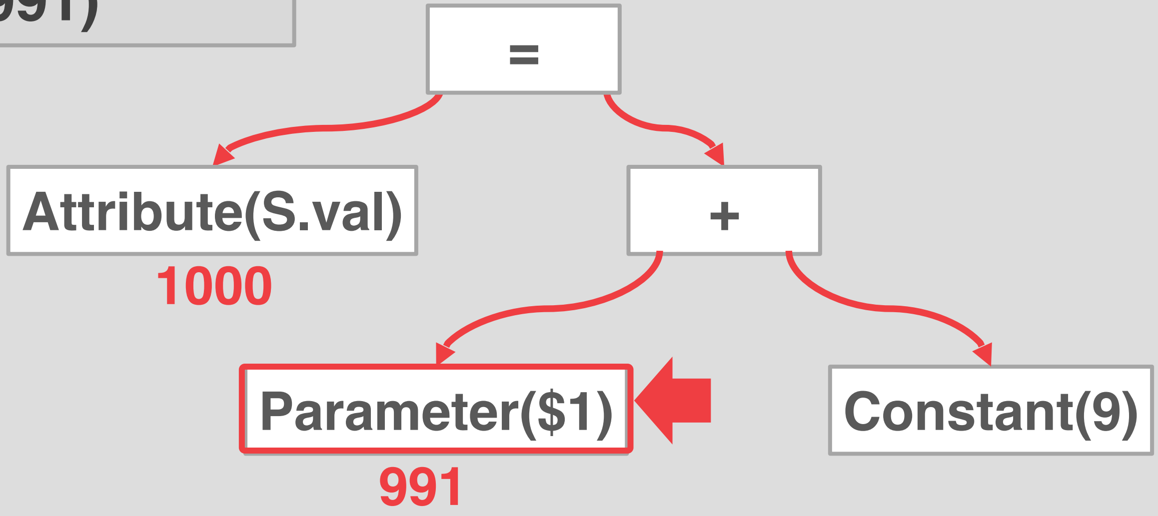
EXPRESSION EVALUATION

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

EXECUTE f(991)

Execution Context

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



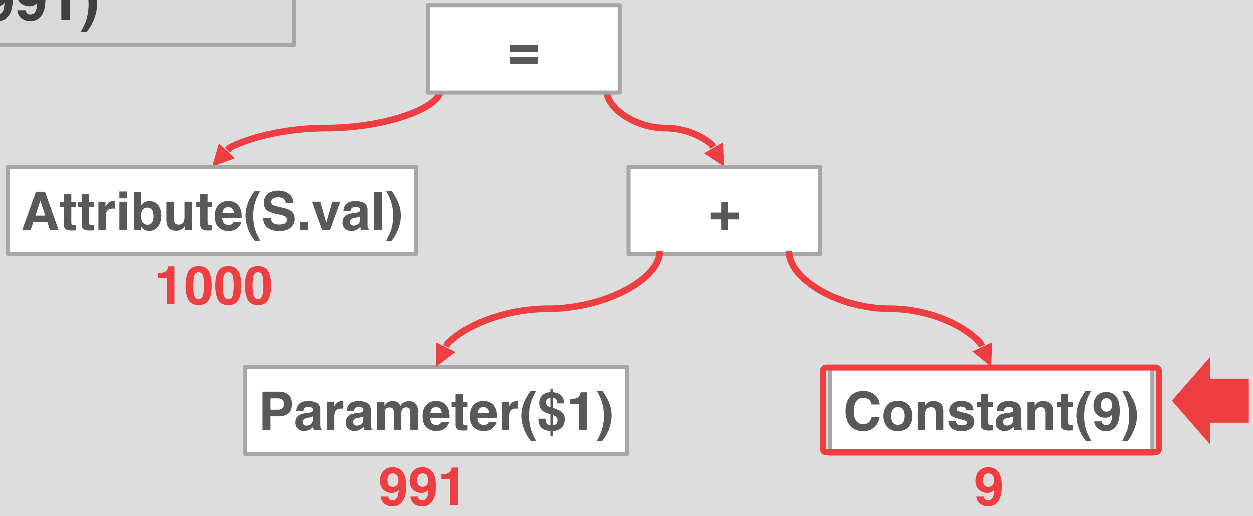
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



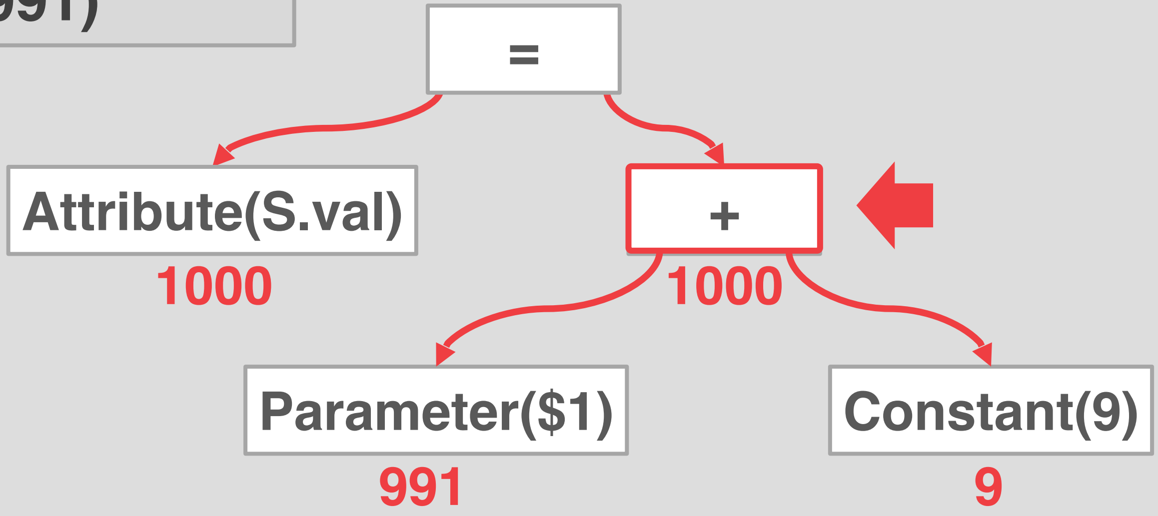
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



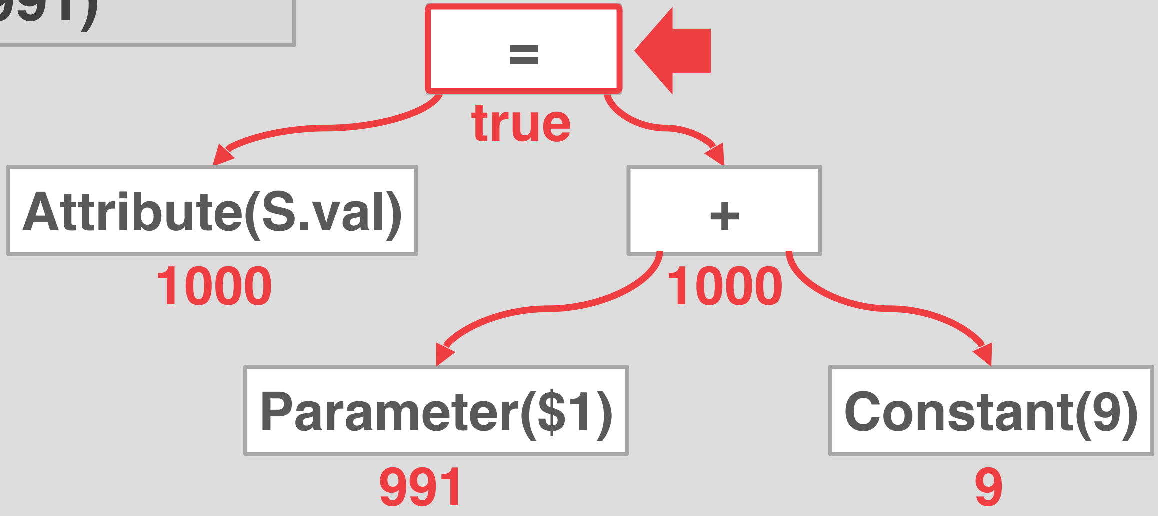
EXPRESSION EVALUATION

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

```
EXECUTE f(991)
```

Execution Context

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



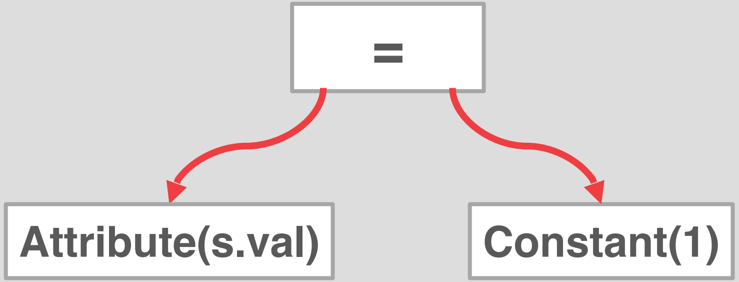
EXPRESSION EVALUATION

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



EXPRESSION EVALUATION

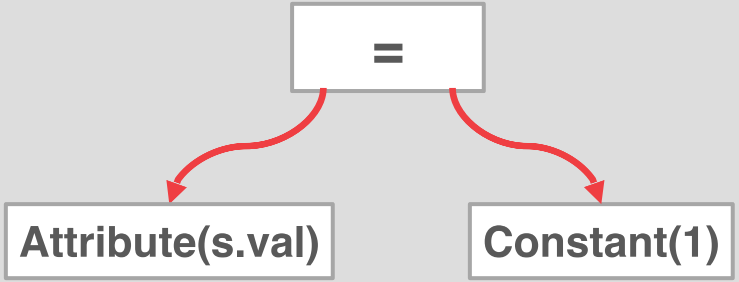
Evaluating predicates in this manner is slow.

→ 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.

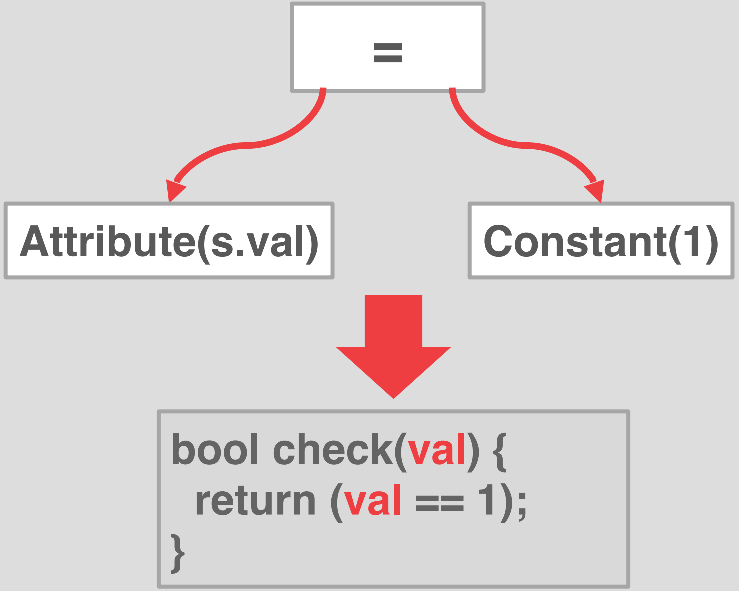
→ The DBMS traverses the tree and for each node that it visits it must figure out what the operator needs to do.

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.

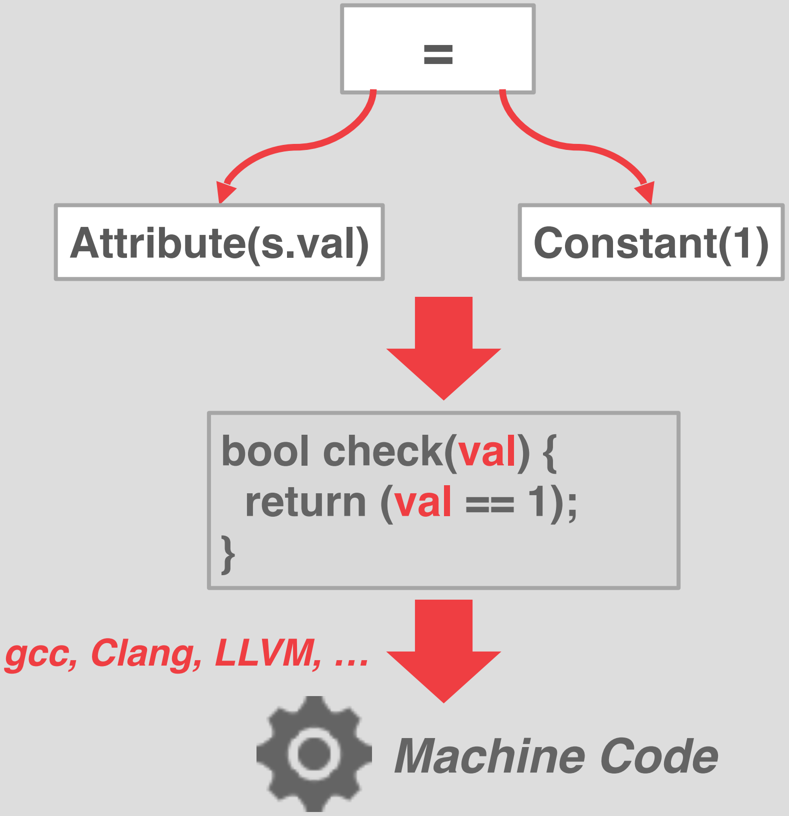
→ The DBMS traverses the tree and for each node that it visits it must figure out what the operator needs to do.

Consider this predicate:

WHERE S.val=1

A better approach is to just evaluate the expression directly.

→ Think JIT compilation



OBSERVATION

So far, we have assumed that all queries execute single-threaded.

We will now discuss how to execute queries in parallel.

WHY CARE ABOUT PARALLELISM?

WHY CARE ABOUT PARALLELISM?

Increased performance for potentially the same hardware resources.
→ Higher Throughput
→ Lower Latency

WHY CARE ABOUT PARALLELISM?

Increased performance for potentially the same hardware resources.
→ Higher Throughput
→ Lower Latency

WHY CARE ABOUT PARALLELISM?

Increased performance for potentially the same hardware resources.
→ Higher Throughput
→ Lower Latency

Increased responsiveness of the system.

WHY CARE ABOUT PARALLELISM?

Increased performance for potentially the same hardware resources.
→ Higher Throughput
→ Lower Latency

Increased responsiveness of the system.

WHY CARE ABOUT PARALLELISM?

Increased performance for potentially the same hardware resources.
→ Higher Throughput
→ Lower Latency

Increased responsiveness of the system.

Potentially lower ***total cost of ownership*** (TCO)
→ Fewer machines means less parts / physical footprint / energy consumption.

PARALLEL VS. DISTRIBUTED

Parallel DBMSs

→ Resources are physically close to each other.
→ Resources communicate over high-speed interconnect.
→ Communication is assumed to be cheap and reliable.

Distributed DBMSs

→ Resources can be far from each other.
→ Resources communicate using slow(er) interconnect.
→ Communication cost and problems cannot be ignored.

PARALLEL VS. DISTRIBUTED

Parallel DBMSs

→ Resources are physically close to each other.
→ Resources communicate over high-speed interconnect.
→ Communication is assumed to be cheap and reliable.

Distributed DBMSs

→ Resources can be far from each other.
→ Resources communicate using slow(er) interconnect.
→ Communication cost and problems cannot be ignored.

SCHEDULING

For each query plan, the DBMS decides where, when, and how to execute it.

→ How many tasks should it use?
→ How many CPU cores should it use?
→ What CPU core should the tasks execute on?
→ Where should a task store its output?

The DBMS ***always*** knows more than the OS.

TYPES OF PARALLELISM

Inter-Query: Execute multiple disparate queries simultaneously.
→ Increases throughput & reduces latency.

Intra-Query: Execute the operations of a single query in parallel.
→ Decreases latency for long-running queries, especially for OLAP queries.

INTER-QUERY PARALLELISM

Improve overall performance by allowing multiple queries to execute simultaneously.

If queries are read-only, then this requires almost no explicit coordination between queries.
→ Buffer pool can handle most of the sharing if necessary

If multiple queries are updating the database at the same time, then this is hard to do correctly…

INTER-QUERY PARALLELISM

Improve overall performance by allowing multiple queries to execute simultaneously.

If queries are read-only, then this requires almost no explicit coordination between queries.

→ Buffer pool can handle most of the sharing if necessary.

Lecture #12

If multiple queries are updating the database at the same time, then this is hard to do correctly…

INTRA-QUERY PARALLELISM

Improve the performance of a single query by executing its operators in parallel.

Think of organization of operators in terms of a ***producer/consumer*** paradigm.

There are parallel versions of every operator.
→ Can either have multiple threads access centralized data structures or use partitioning to divide work up.

INTRA-QUERY PARALLELISM

Approach #1: Intra-Operator (Horizontal)

Approach #2: Inter-Operator (Vertical)

Approach #3: Bushy

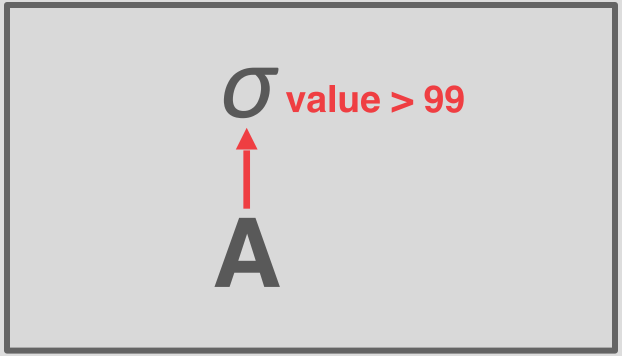
INTRA-OPERATOR PARALLELISM

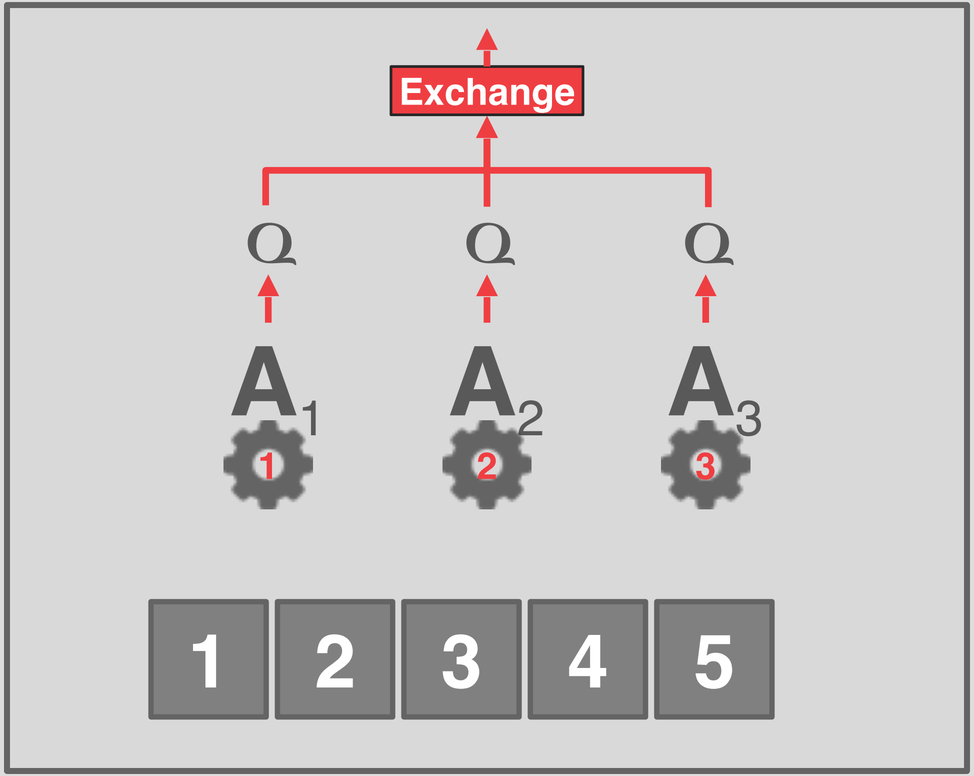
Approach #1: Intra-Operator (Horizontal)
→ Decompose operators into independent **fragments** that perform the same function on different subsets of data.

The DBMS inserts an **exchange** operator into the query plan to coalesce/split results from multiple children/parent operators.
→ Postgres calls this "gather"

INTRA-OPERATOR PARALLELISM

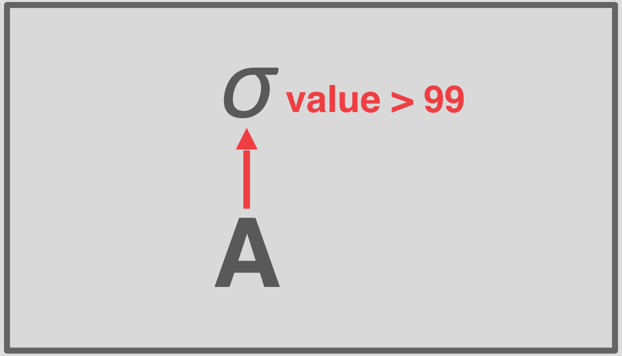
SELECT * FROM A
WHERE A.val > 99

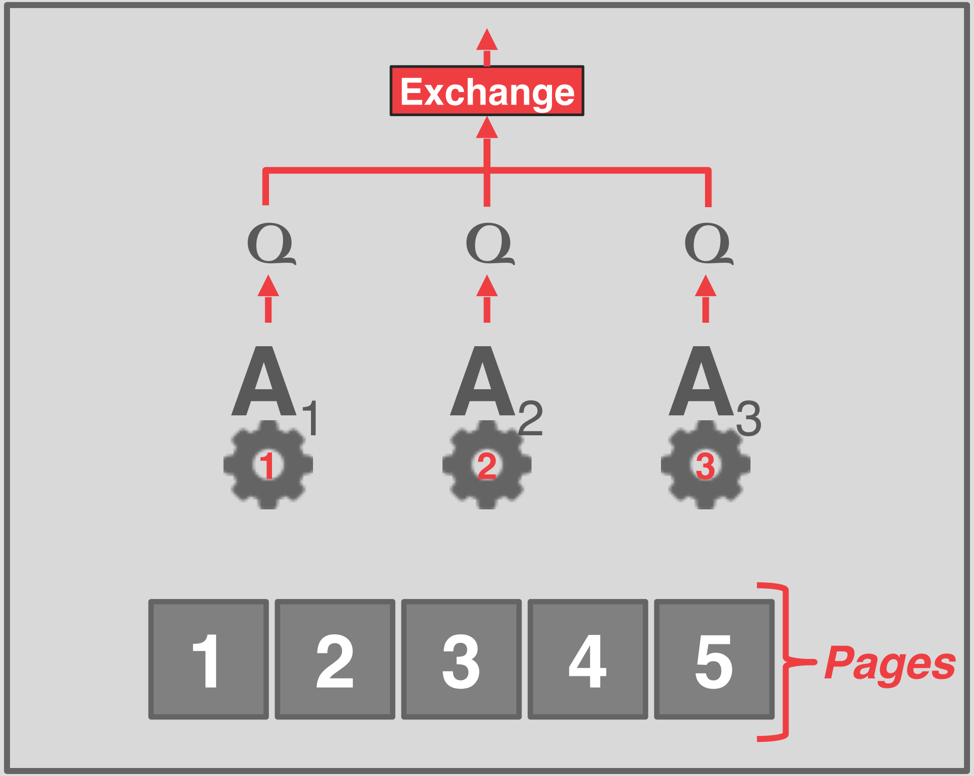




INTRA-OPERATOR PARALLELISM

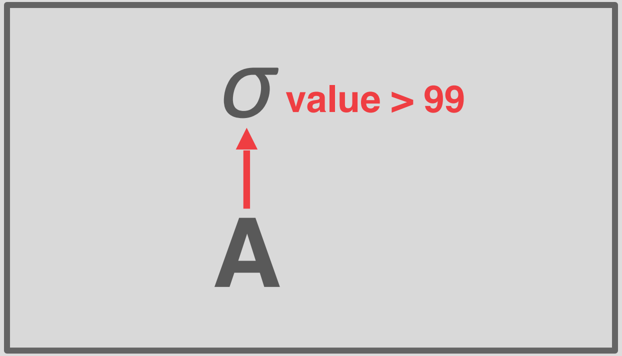
```
SELECT * FROM A
 WHERE A.val > 99
```

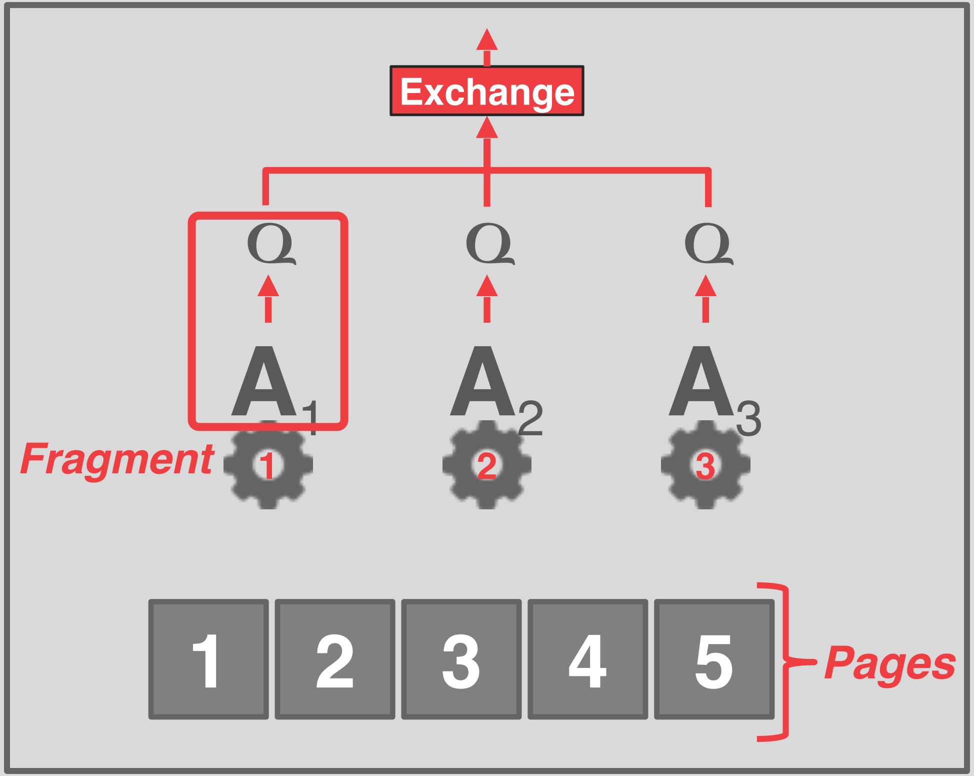




INTRA-OPERATOR PARALLELISM

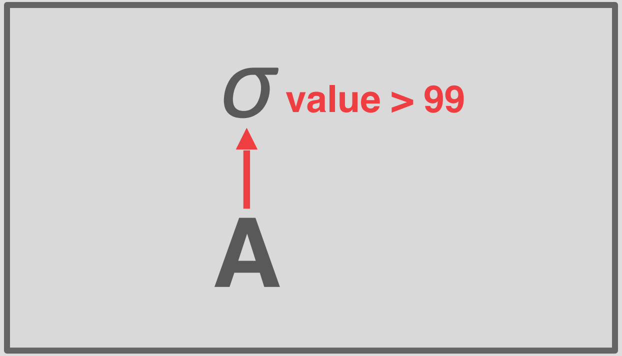
SELECT * FROM A
WHERE A.val > 99

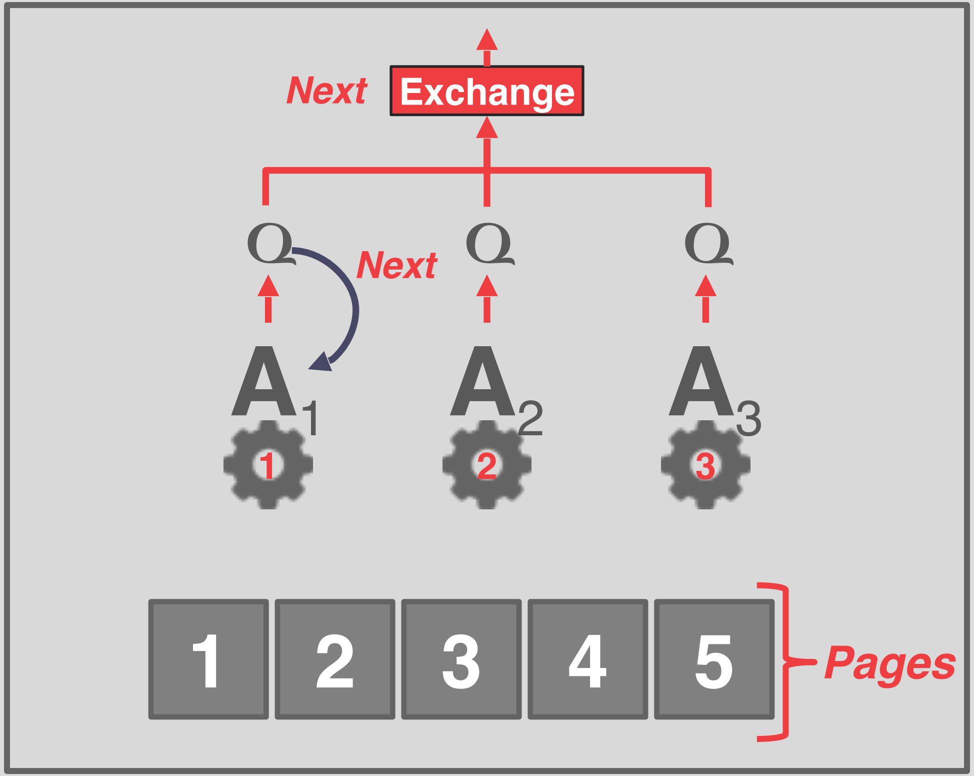




INTRA-OPERATOR PARALLELISM

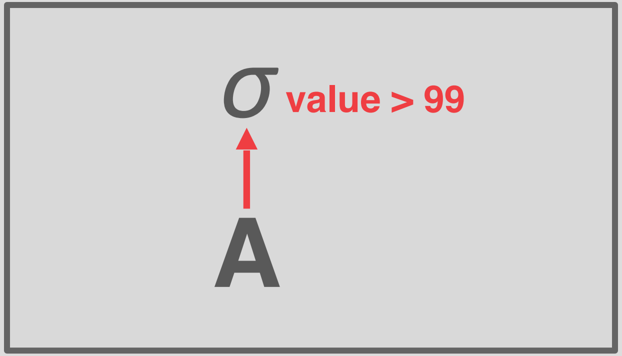
```
SELECT * FROM A
 WHERE A.val > 99
```

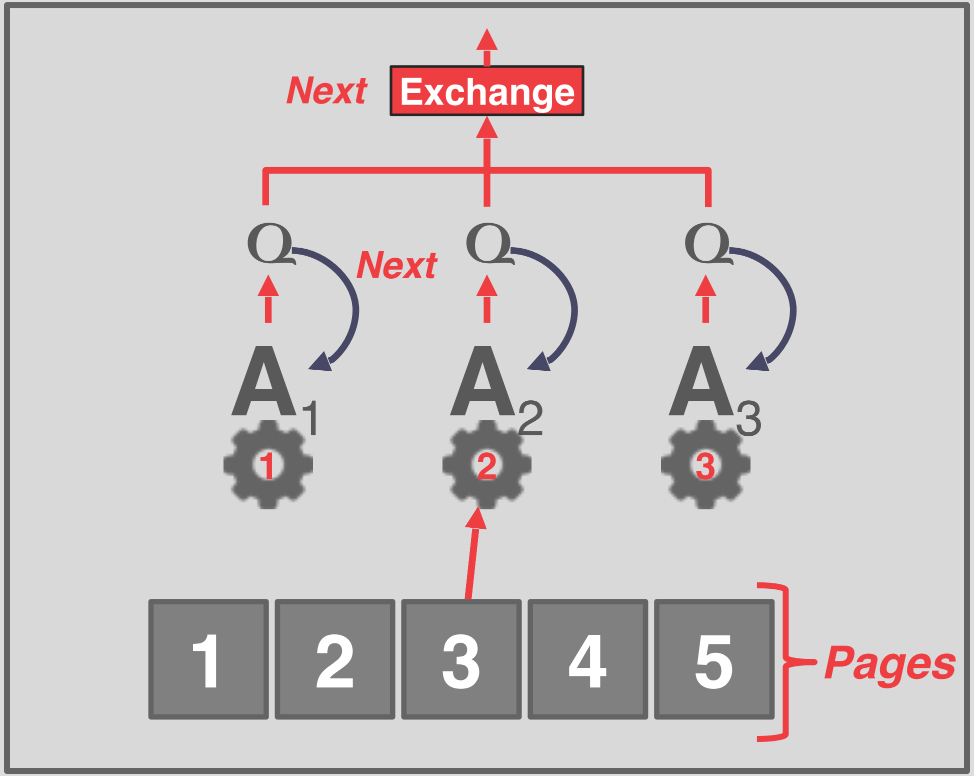




INTRA-OPERATOR PARALLELISM

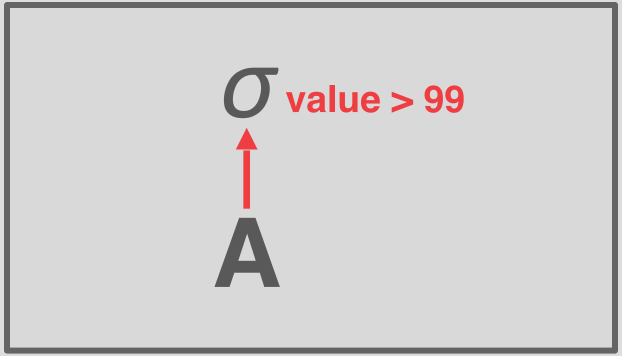
SELECT * FROM A
WHERE A.val > 99

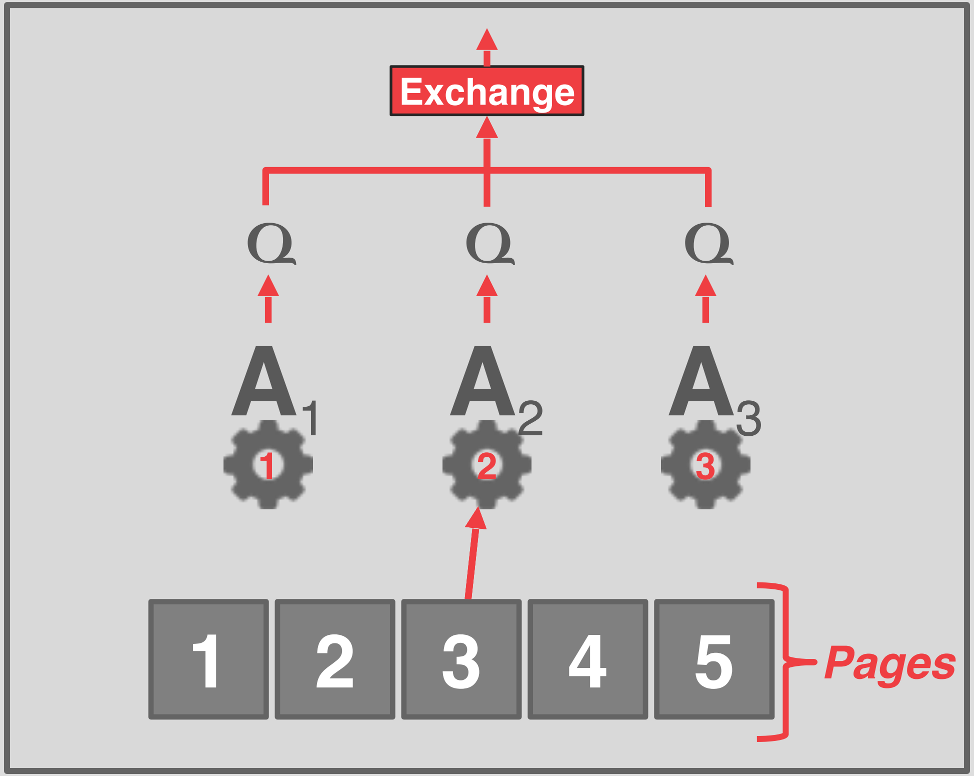




INTRA-OPERATOR PARALLELISM

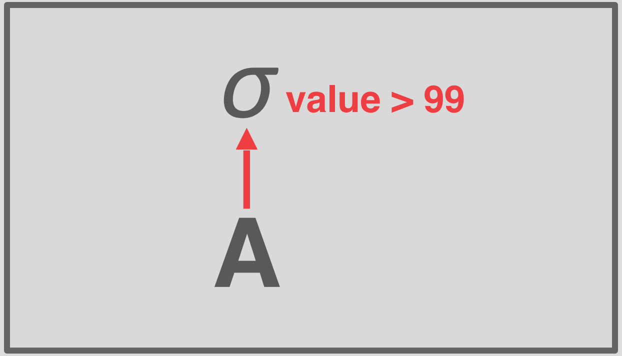
```
SELECT * FROM A
 WHERE A.val > 99
```

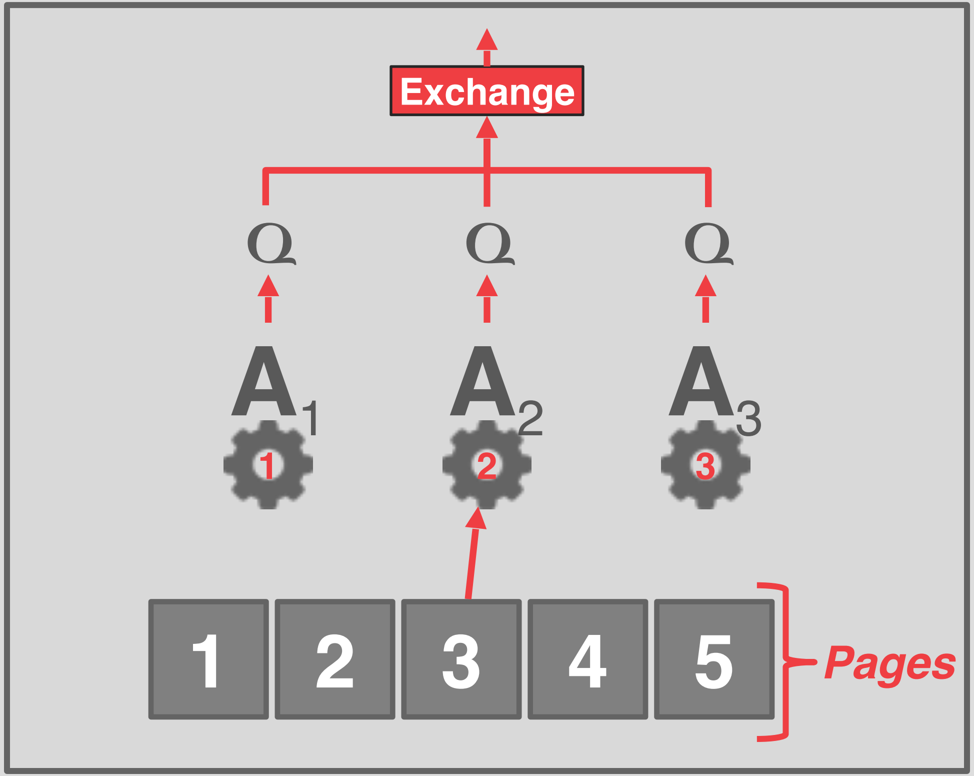




INTRA-OPERATOR PARALLELISM

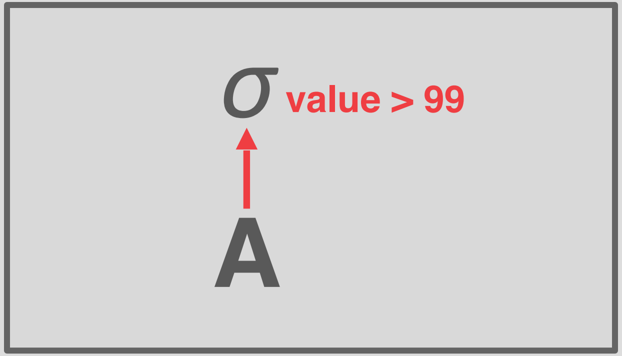
```
SELECT * FROM A
 WHERE A.val > 99
```

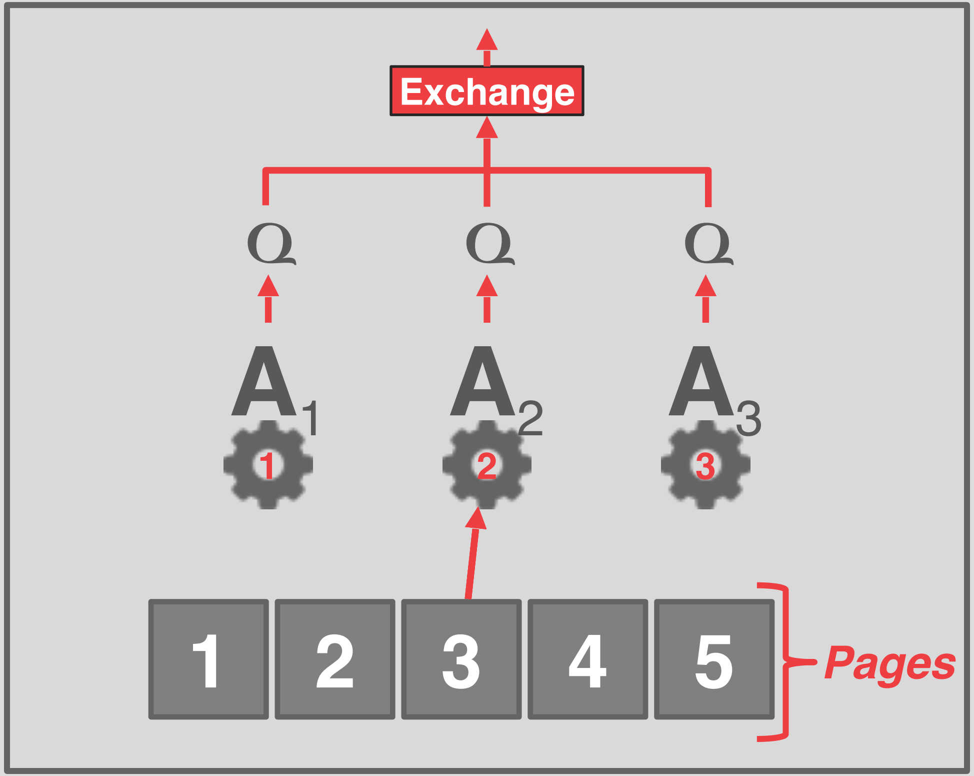




INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A
 WHERE A.val > 99
```





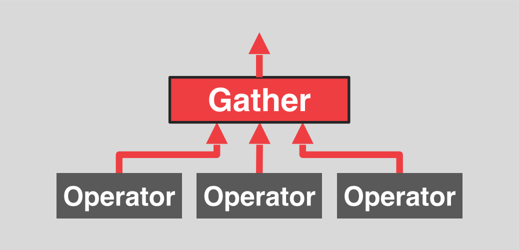
EXCHANGE OPERATOR

Source: Craig Freedman

EXCHANGE OPERATOR

Exchange Type #1 – Gather

→ Combine the results from multiple workers into a single output stream.

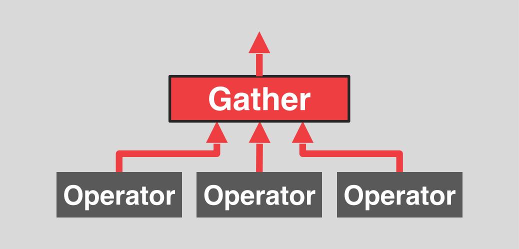


Source: [Craig Freedman]()

EXCHANGE OPERATOR

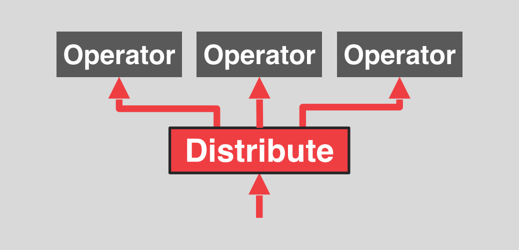
Exchange Type #1 – Gather

→ Combine the results from multiple workers into a single output stream.



Exchange Type #2 – Distribute

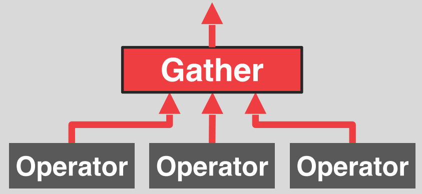
→ Split a single input stream into multiple output streams.



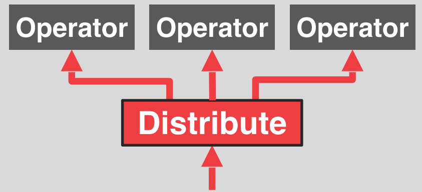
Source: Craig Freedman

EXCHANGE OPERATOR

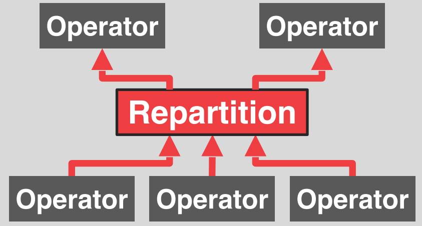
Exchange Type #1 – Gather
→ Combine the results from multiple workers into a single output stream.



Exchange Type #2 – Distribute
→ Split a single input stream into multiple output streams.



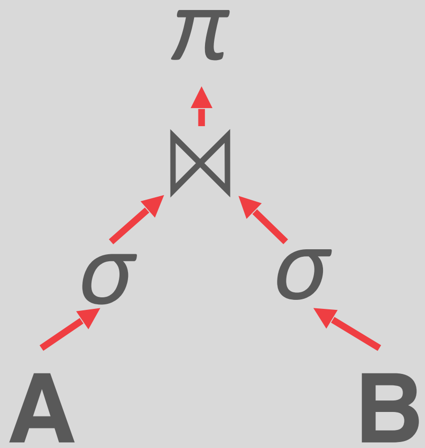
Exchange Type #3 – Repartition
→ Shuffle multiple input streams across multiple output streams.



Source: [Craig Freedman]

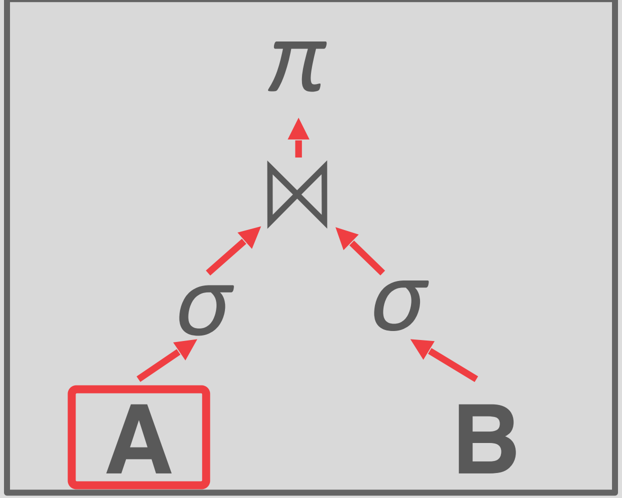
INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



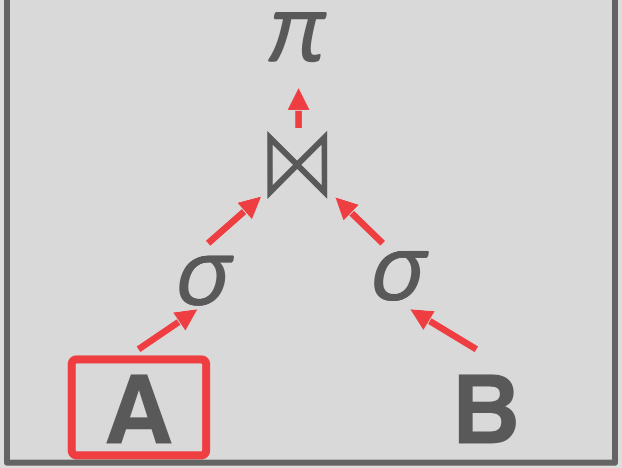
INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



INTRA-OPERATOR PARALLELISM

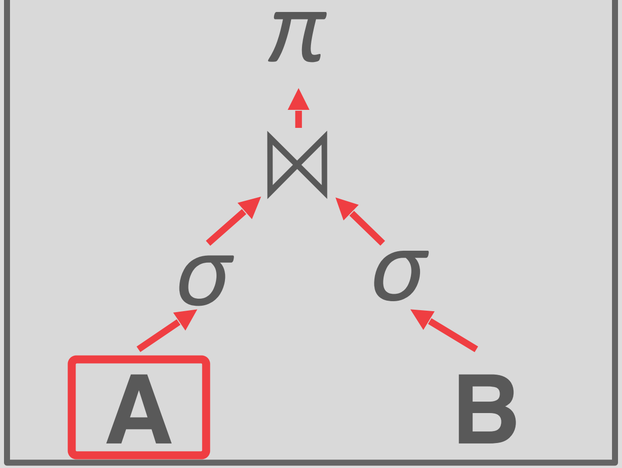
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



A_1 A_2 A_3

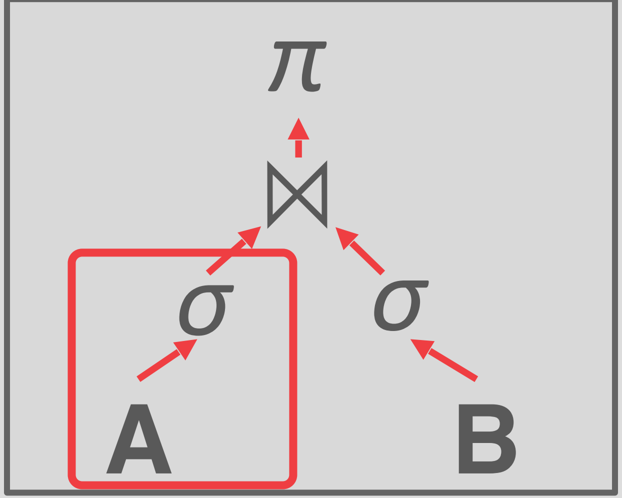
INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



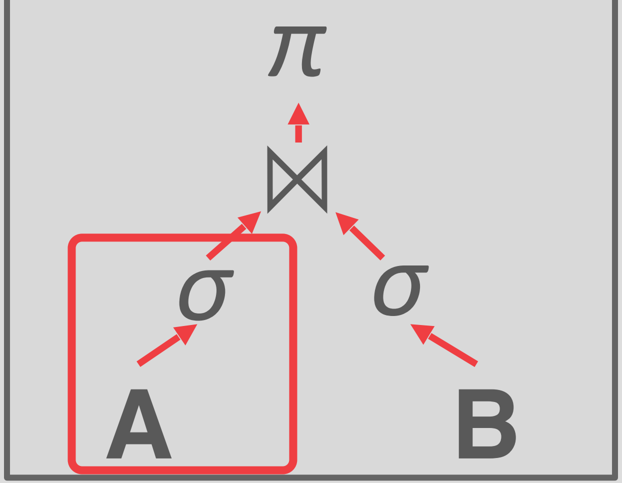
INTRA-OPERATOR PARALLELISM

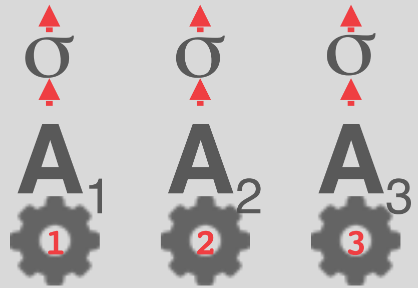
```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



INTRA-OPERATOR PARALLELISM

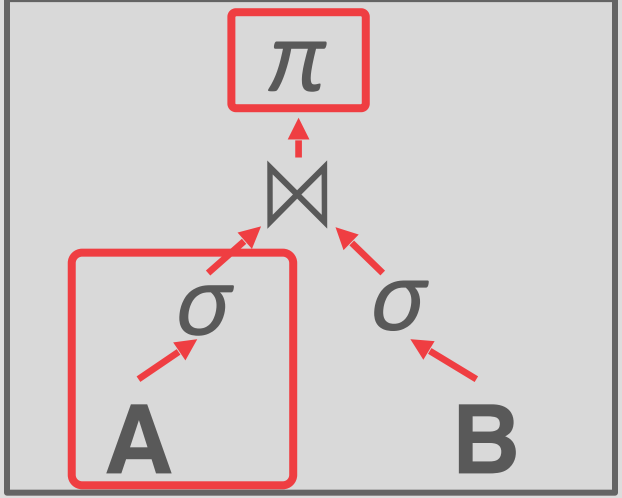
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

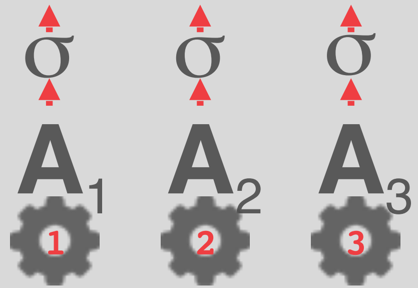




INTRA-OPERATOR PARALLELISM

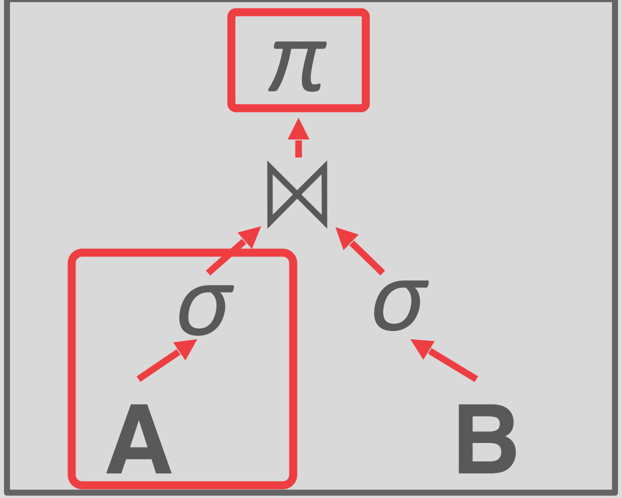
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

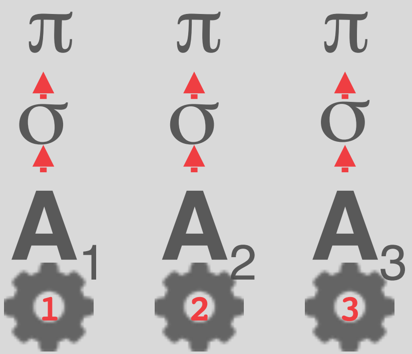




INTRA-OPERATOR PARALLELISM

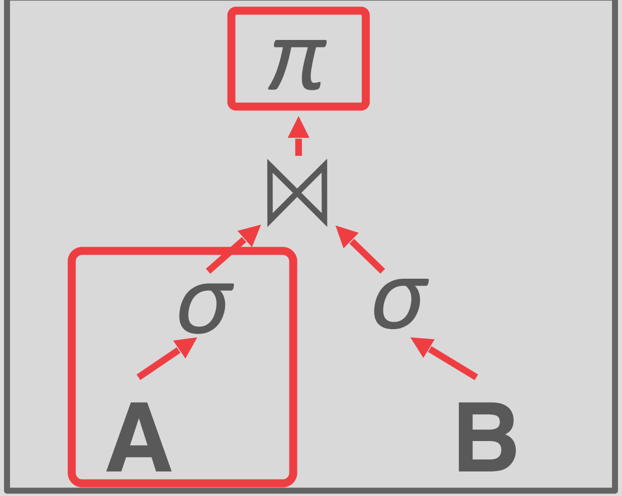
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

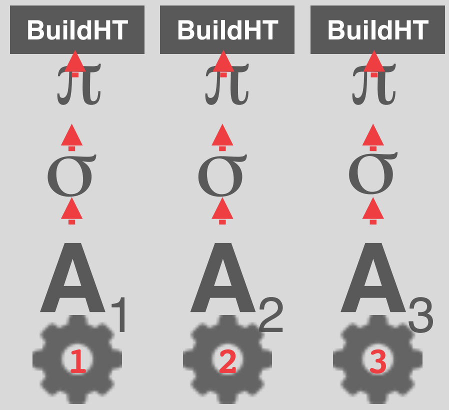




INTRA-OPERATOR PARALLELISM

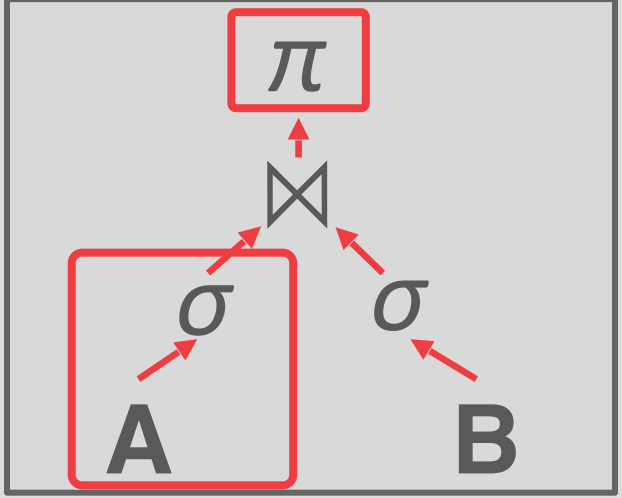
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

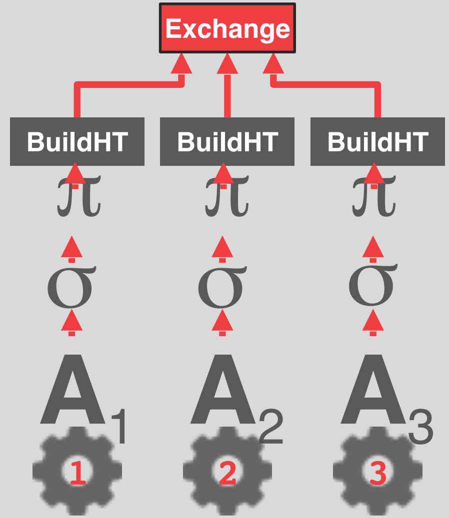




INTRA-OPERATOR PARALLELISM

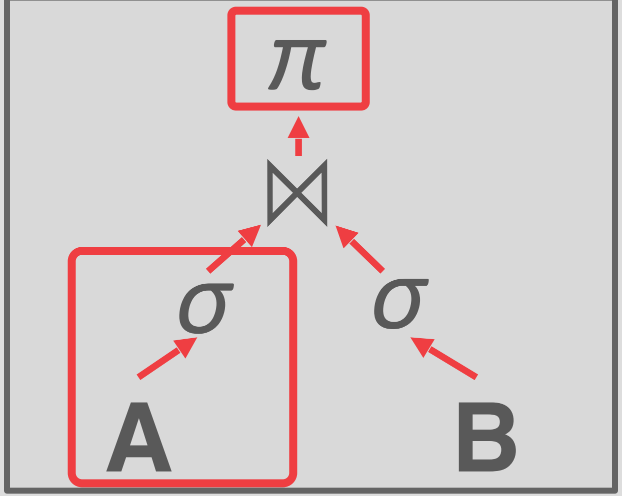
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

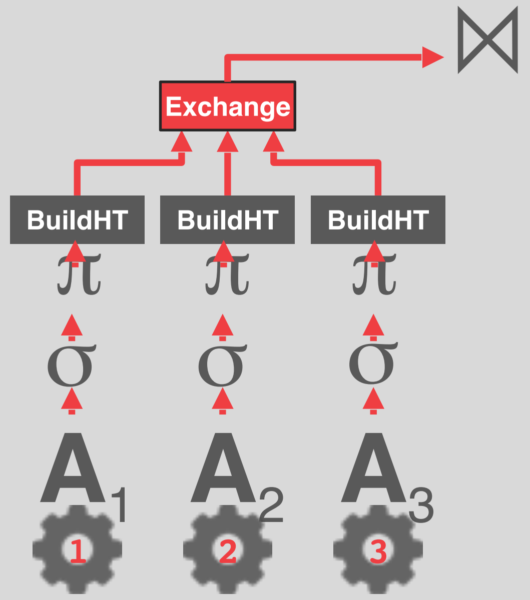




INTRA-OPERATOR PARALLELISM

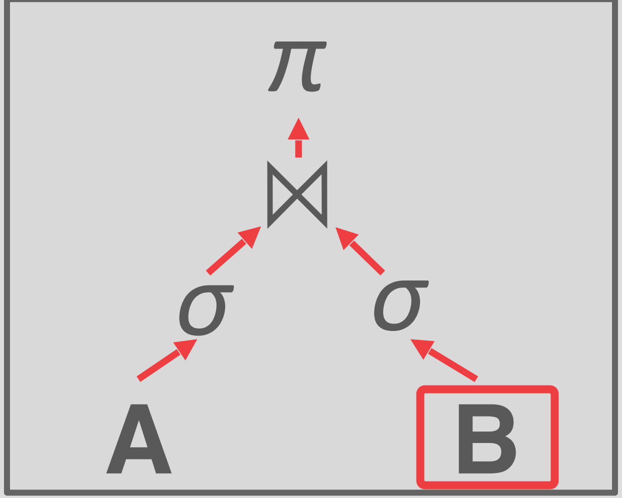
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

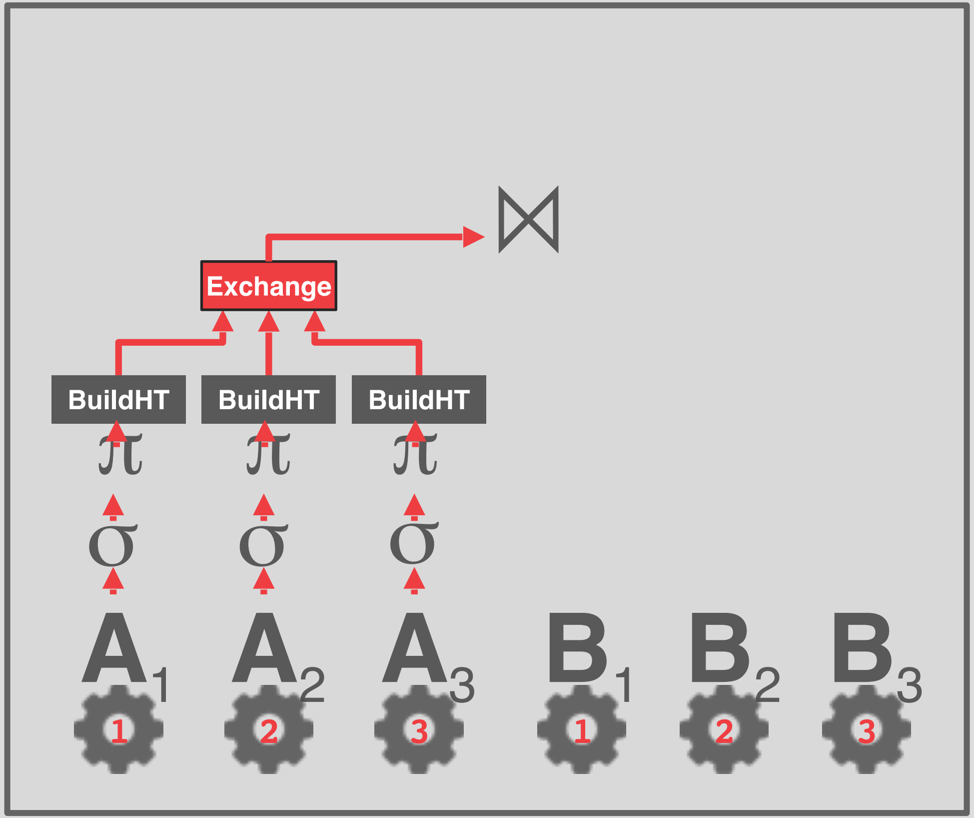




INTRA-OPERATOR PARALLELISM

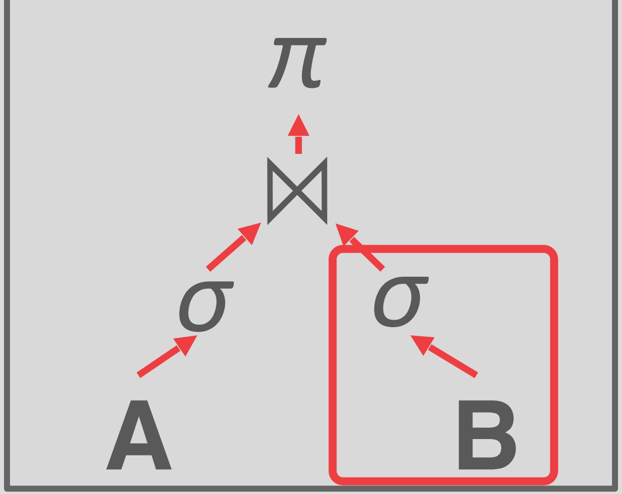
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

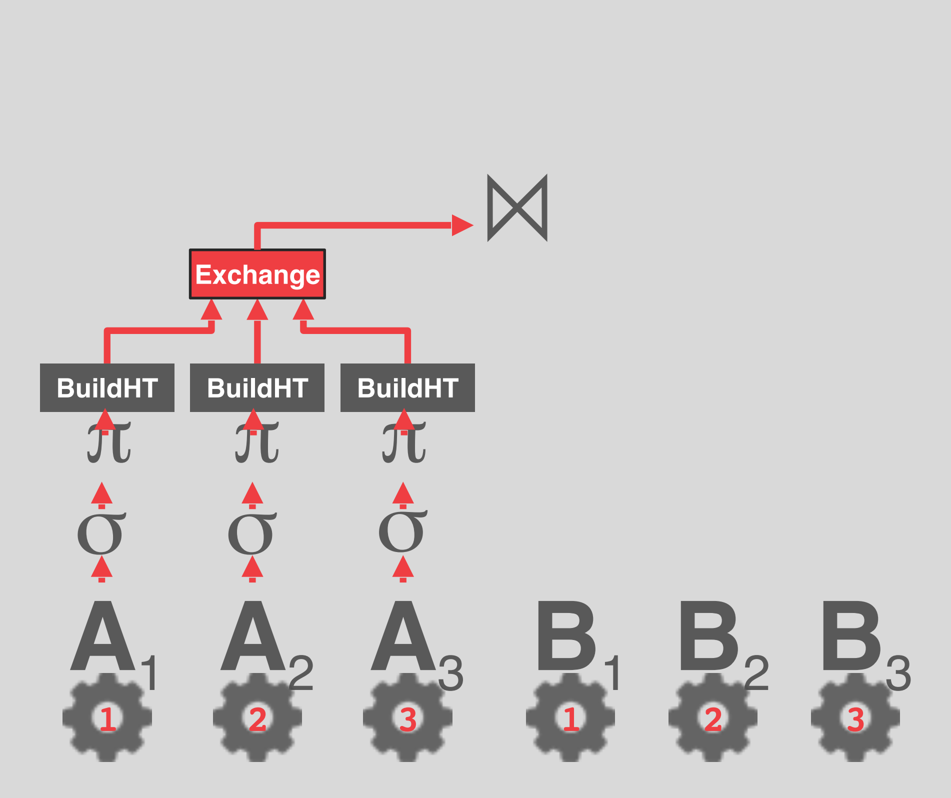




INTRA-OPERATOR PARALLELISM

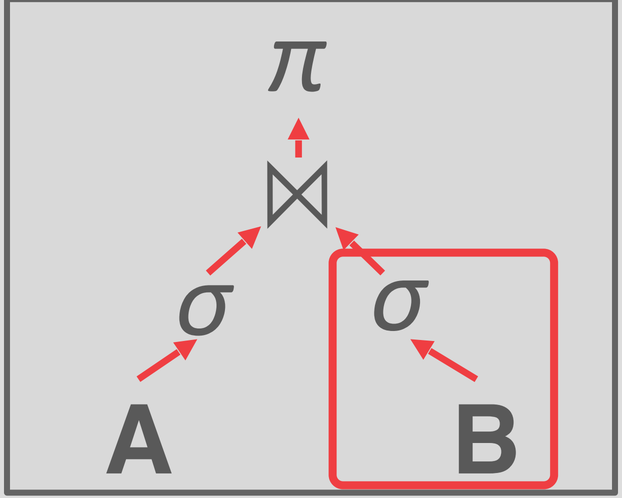
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

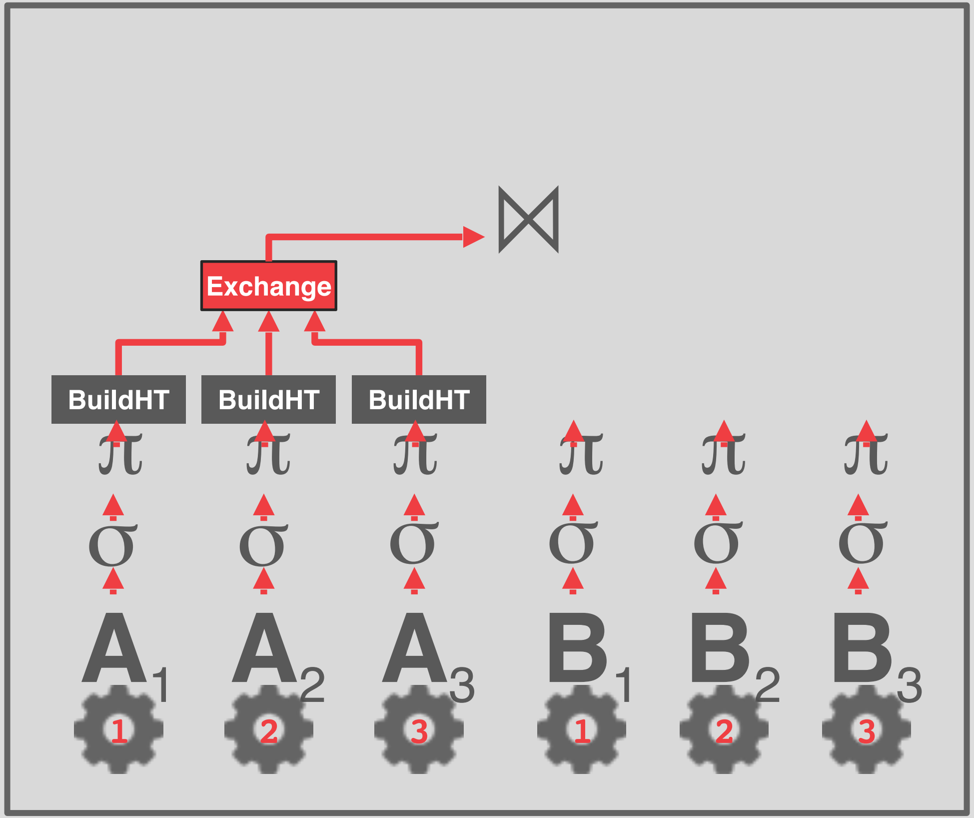




INTRA-OPERATOR PARALLELISM

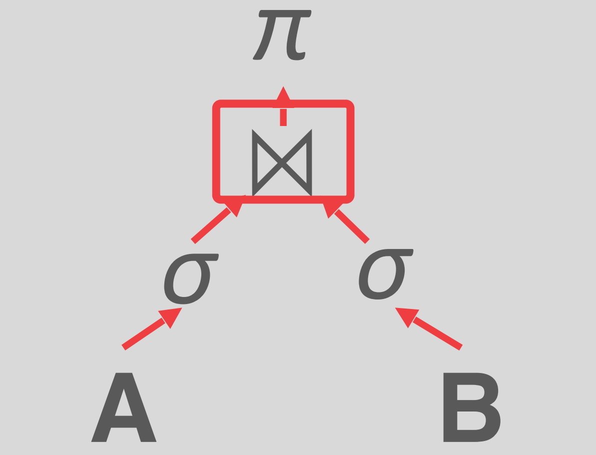
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

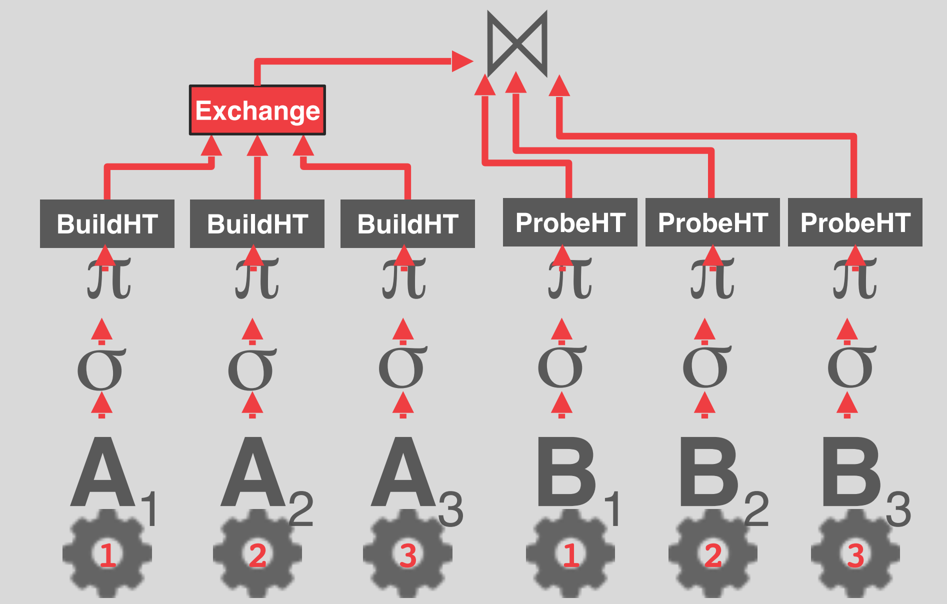




INTRA-OPERATOR PARALLELISM

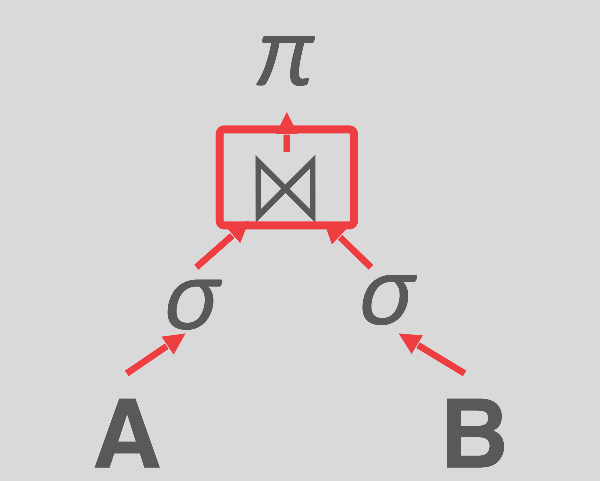
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
  AND B.value > 100
```

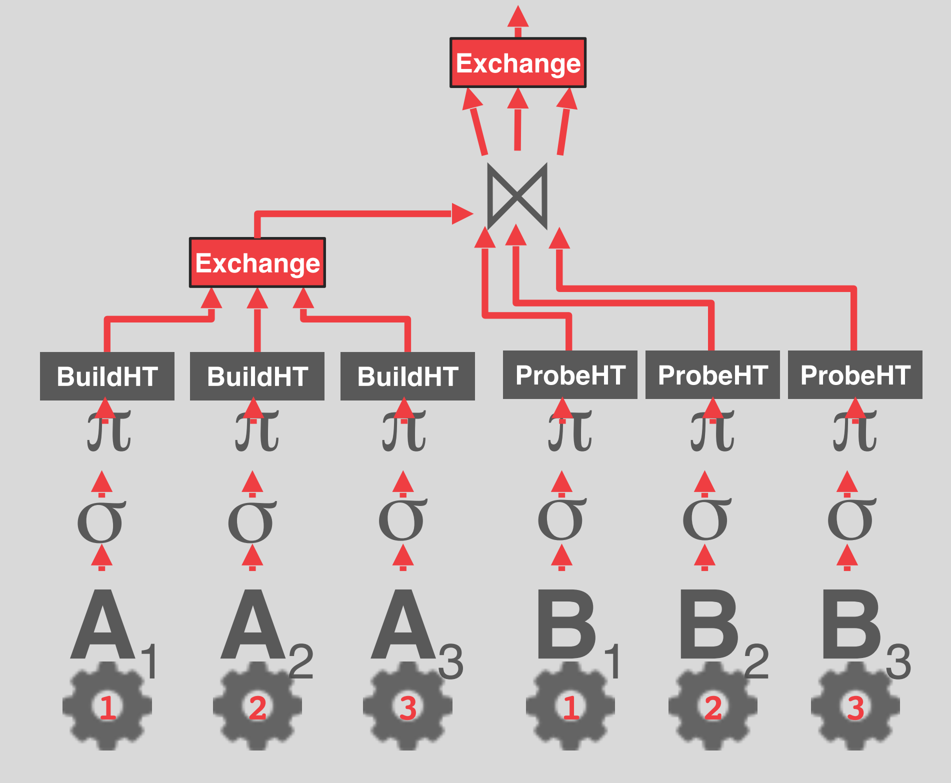




INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```





INTER-OPERATOR PARALLELISM

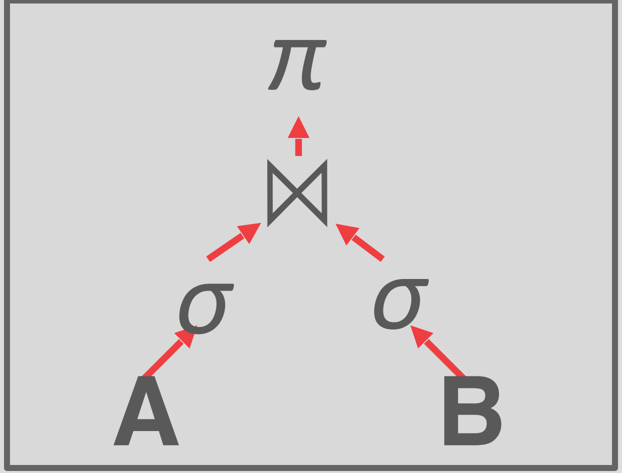
Approach #2: Inter-Operator (Vertical)

→ Operations are overlapped in order to pipeline data from one stage to the next without materialization.
→ Workers execute operators from different segments of a query plan at the same time.
→ More common in streaming systems (continuous queries)

Also called **pipeline parallelism**.

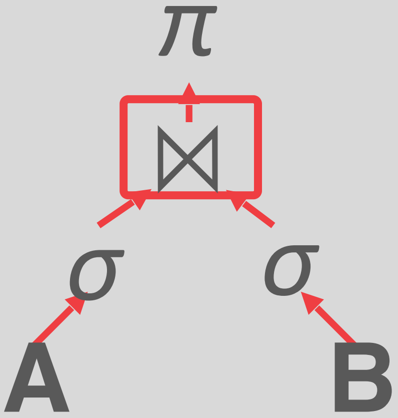
INTER-OPERATOR PARALLELISM

```
SELECT A.id, B.value
  FROM A JOIN B
   ON A.id = B.id
 WHERE A.value < 99
  AND B.value > 100
```



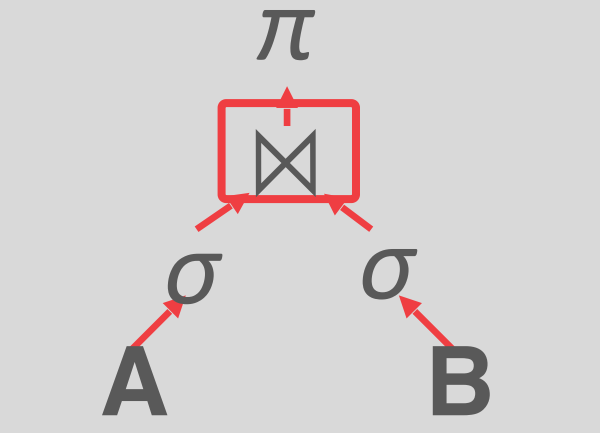
INTER-OPERATOR PARALLELISM

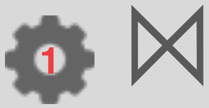
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



INTER-OPERATOR PARALLELISM

```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

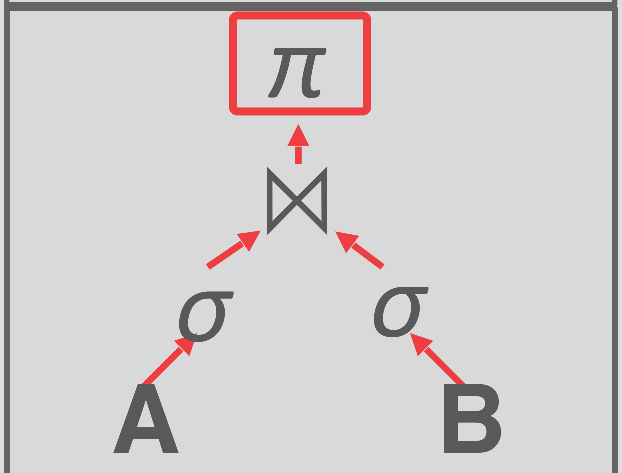


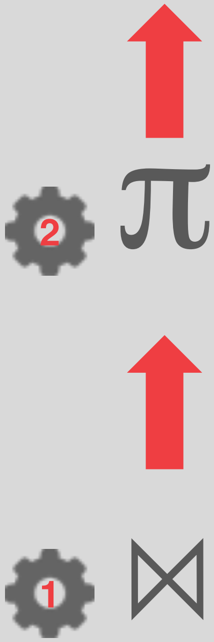


for $r_1 \in$ outer:
  for $r_2 \in$ inner:
    emit($r_1 \bowtie r_2$)

INTER-OPERATOR PARALLELISM

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



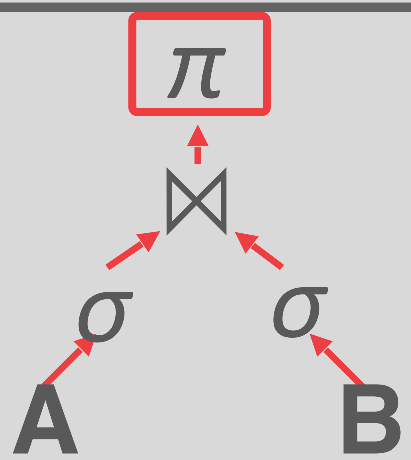


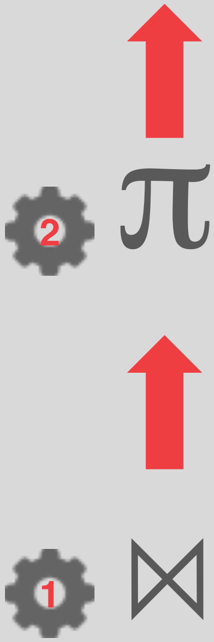
for r ∈ incoming:
  emit(πr)

for r_1 ∈ outer:
  for r_2 ∈ inner:
    emit($r_1 \bowtie r_2$)

INTER-OPERATOR PARALLELISM

SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100





for $r \in$ incoming:
  emit(πr)

for $r_1 \in$ outer:
  for $r_2 \in$ inner:
    emit($r_1 \bowtie r_2$)

BUSHY PARALLELISM

Approach #3: Bushy Parallelism

→ Hybrid of intra- and inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

→ Still need exchange operators to combine intermediate results from segments.

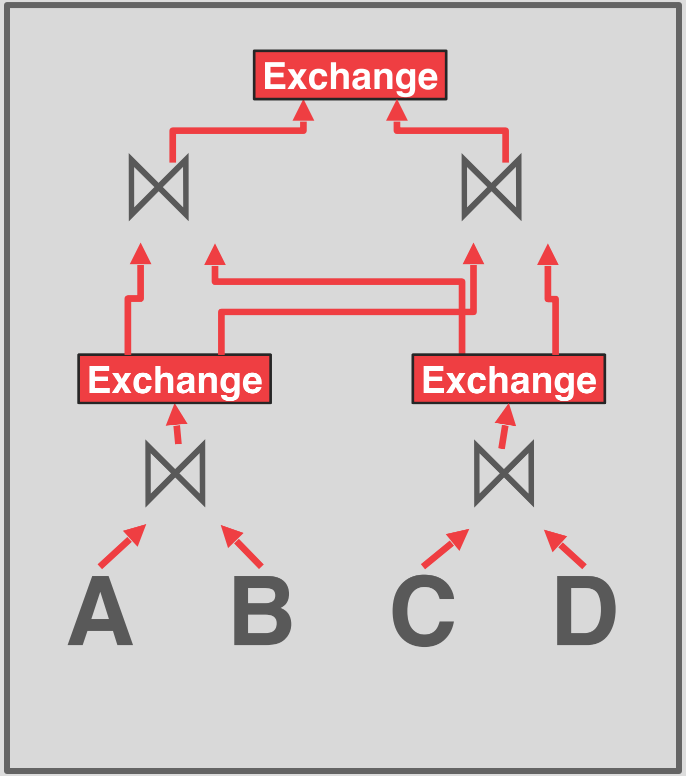
BUSHY PARALLELISM

Approach #3: Bushy Parallelism

→ Hybrid of intra- and inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

→ Still need exchange operators to combine intermediate results from segments.

```
SELECT *
FROM A JOIN B JOIN C JOIN D
```



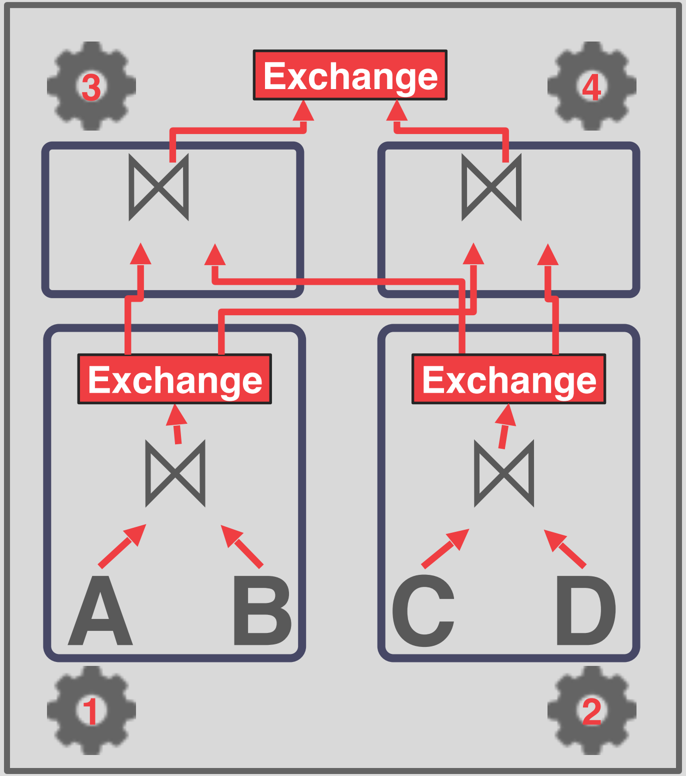
BUSHY PARALLELISM

Approach #3: Bushy Parallelism

→ Hybrid of intra- and inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

→ Still need exchange operators to combine intermediate results from segments.

```
SELECT *
FROM A JOIN B JOIN C JOIN D
```



OBSERVATION

Using additional processes/threads to execute queries in parallel won't help if the disk is always the main bottleneck.

It can sometimes make performance worse if a thread is accessing different segments of the disk at the same time.

I/O PARALLELISM

Split the DBMS across multiple storage devices to improve disk bandwidth latency.

Many different options that have trade-offs:
→ Multiple Disks per Database
→ One Database per Disk
→ One Relation per Disk
→ Split Relation across Multiple Disks

Some DBMSs support this natively. Others require admin to configure outside of DBMS.

MULTI-DISK PARALLELISM

Configure OS/hardware to store the DBMS's files across multiple storage devices.

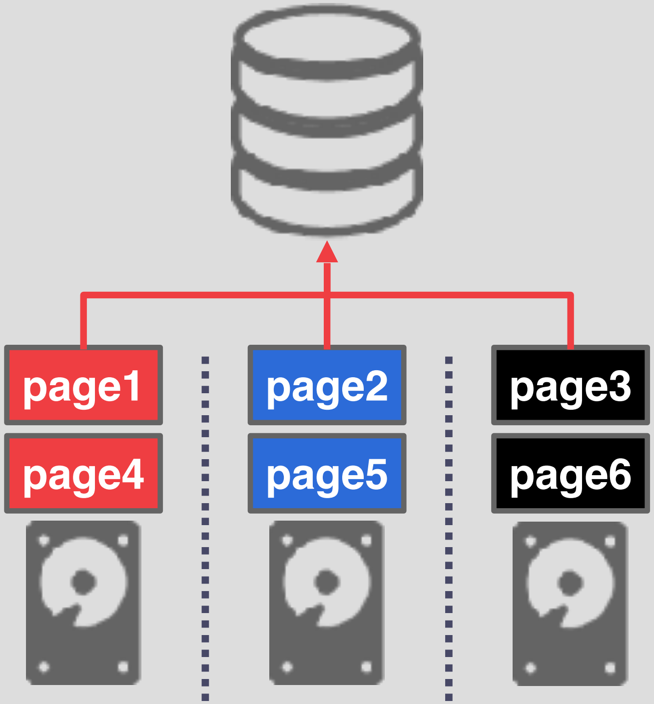
→ Storage Appliances
→ RAID Configuration

This is **transparent** to the DBMS.

MULTI-DISK PARALLELISM

Configure OS/hardware to store the DBMS's files across multiple storage devices.
→ Storage Appliances
→ RAID Configuration

This is **transparent** to the DBMS.

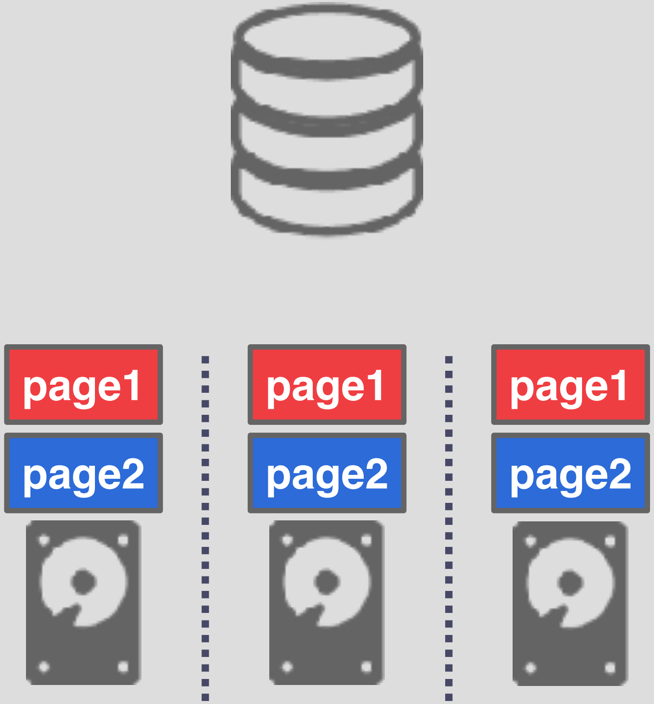


RAID 0 (Striping)

MULTI-DISK PARALLELISM

Configure OS/hardware to store the DBMS's files across multiple storage devices.
→ Storage Appliances
→ RAID Configuration

This is **transparent** to the DBMS.



RAID 1 (Mirroring)

CONCLUSION

The same query plan can be executed in multiple different ways.

Expression trees are flexible but slow.

Parallel execution is important, which is why (almost) every major DBMS supports it.

However, it is hard to get right.
→ Coordination Overhead
→ Scheduling / Resource Contention
→ Concurrency Issues

NEXT CLASS

Query Planning & Optimization