Carnegie Mellon University

DATABASE SYSTEMS

Query Execution – Pt.1

LECTURE #13 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #2 is due Sunday Oct 26th @ 11:59pm
→ See Recitation Video (**@158**)
→ Office Hours: Saturday Oct 25th @ 3:00-5:00pm in GHC 5207

Mid-term exam grades posted
→ Come to Andy's OH to view your grade and solution.

UPCOMING DATABASE TALKS

Columnar (DB Seminar)

→ Monday Oct 20th @ 4:30pm ET

→ Zoom

Astronomer Tech Talk

→ Tuesday Oct 21st @ 12:00pm

→ GHC 8115

SingleStore (DB Seminar)

→ Monday Oct 27th @ 12:00pm

→ Zoom

LAST CLASS

We discussed different join operator algorithms.

We now know how to implement basic query operators
→ Sorting, Aggregations, Joins

The next two weeks are about how put all the pieces together to execute queries…

QUERY EXECUTION

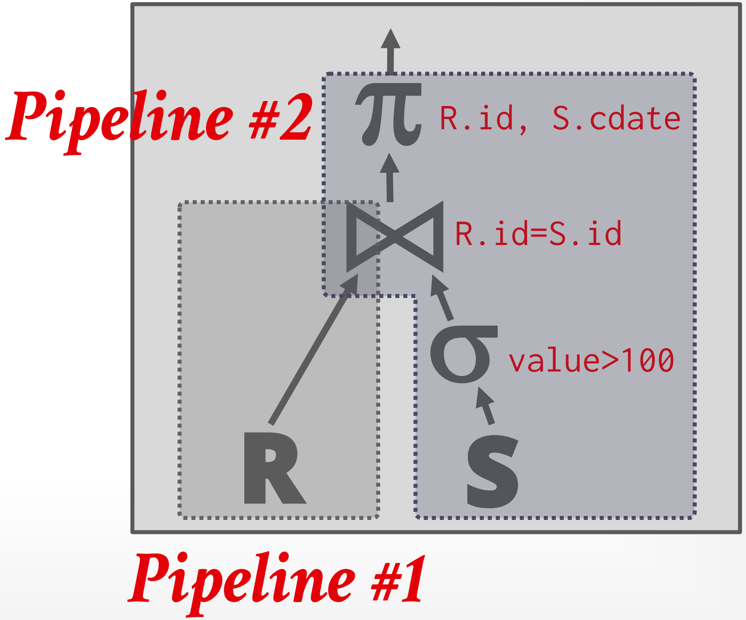
A query plan is a DAG of **operators**.

A **pipeline** is a sequence of operators where tuples continuously flow between them without intermediate storage.

A **pipeline breaker** is an operator that cannot finish until all its children emit all their tuples.

→ Joins (Build Side), Subqueries, Order By

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



Today's Agenda

Processing Models

Access Methods

Modification Queries

Expression Evaluation

PROCESSING MODEL

A DBMS's **processing model** defines how the system executes a query plan and moves data from one operator to the next.
→ Different trade-offs for workloads (OLTP vs. OLAP).

Each processing model is comprised of two types of execution paths:
→ **Control Flow:** How the DBMS invokes an operator.
→ **Data Flow:** How an operator sends its results.

The output of an operator can be either whole tuples (NSM) or subsets of columns (DSM).

PROCESSING MODEL

Approach #1: Iterator Model ← *Most Common*

Approach #2: Materialization Model ← *Rare*

Approach #3: Vectorized / Batch Model ← *Common*

ITERATOR MODEL

Each query plan **operator** implements a `Next()` function.

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

→ The operator implements a loop that calls `Next()` on its children to retrieve their tuples and then process them.

Each operator implementation also has `Open()` and `Close()` functions.

→ Analogous to constructors/destructors, but for operators.

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

ITERATOR MODEL

Control Flow →
Data Flow →

Next()

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

Next()

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

Next()

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

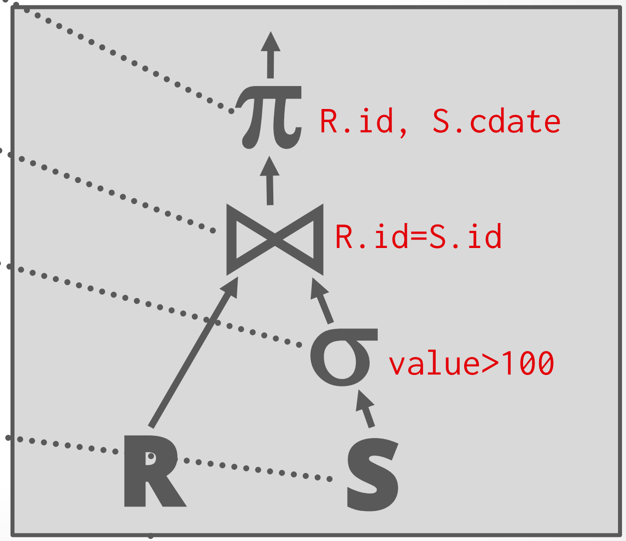
Next()

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

Next()

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

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



ITERATOR MODEL

Control Flow →
Data Flow →

1

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

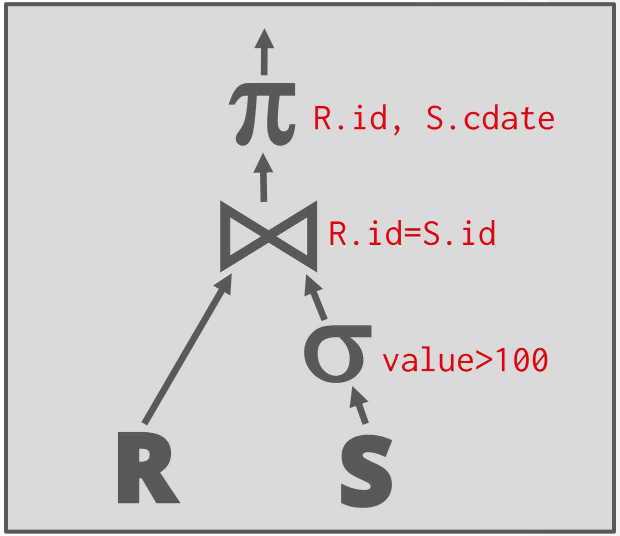
```
for t₁ in left.Next():
  buildHashTable(t₁)
for t₂ in right.Next():
  if probe(t₂): emit(t₁⋈t₂)
```

```
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

Control Flow →
Data Flow →

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

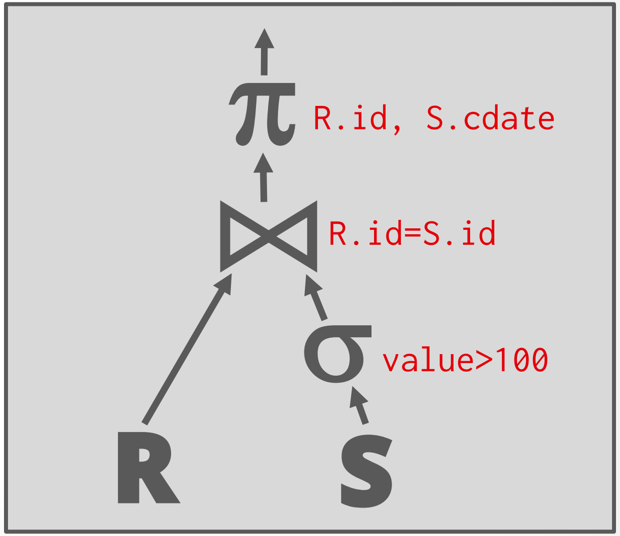
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

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

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

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



ITERATOR MODEL

Control Flow →
Data Flow →

1

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

2

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

Single Tuple

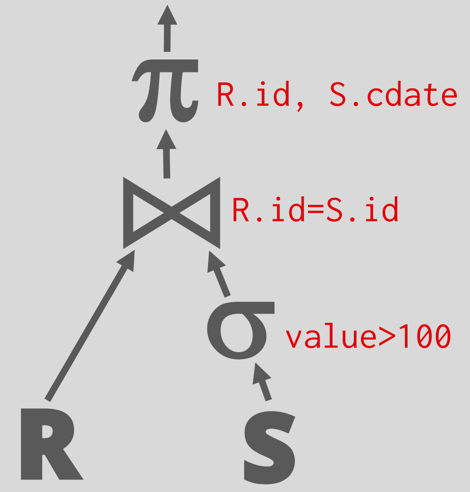
```
for t in child.Next():
  if evalPred(t): emit(t)
```

3

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

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

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



ITERATOR MODEL

Control Flow →
Data Flow →

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

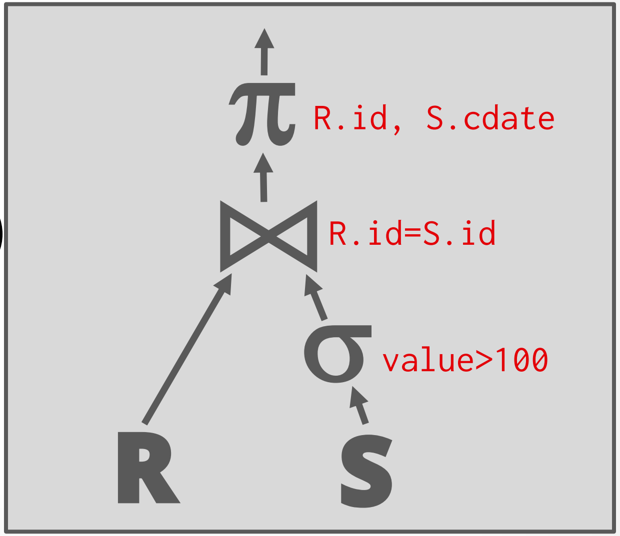
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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)
```

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



ITERATOR MODEL

Control Flow →
Data Flow →

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

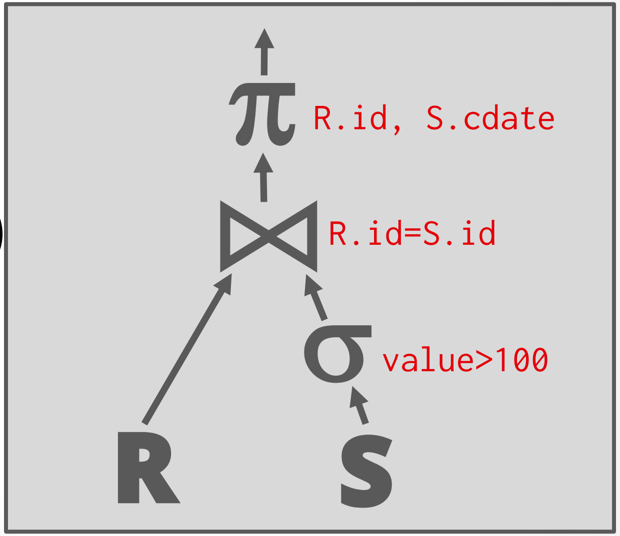
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

Control Flow →
Data Flow →

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

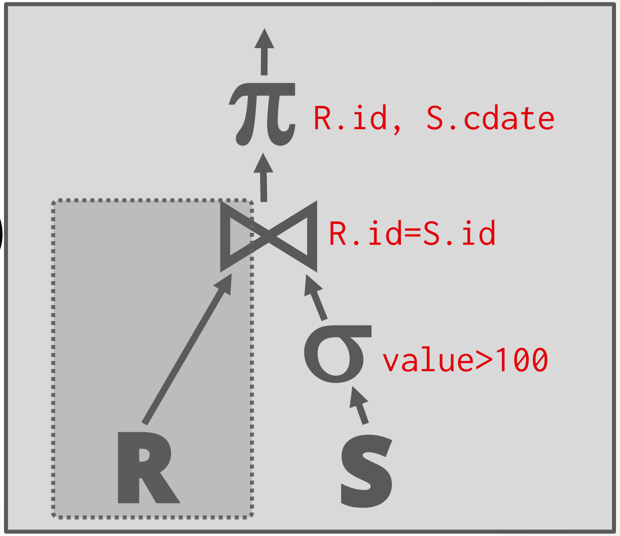
2
```
for t₁ in left.Next():
  buildHashTable(t₁)
for t₂ in right.Next():
  if probe(t₂): emit(t₁⋈t₂)
```

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)
```

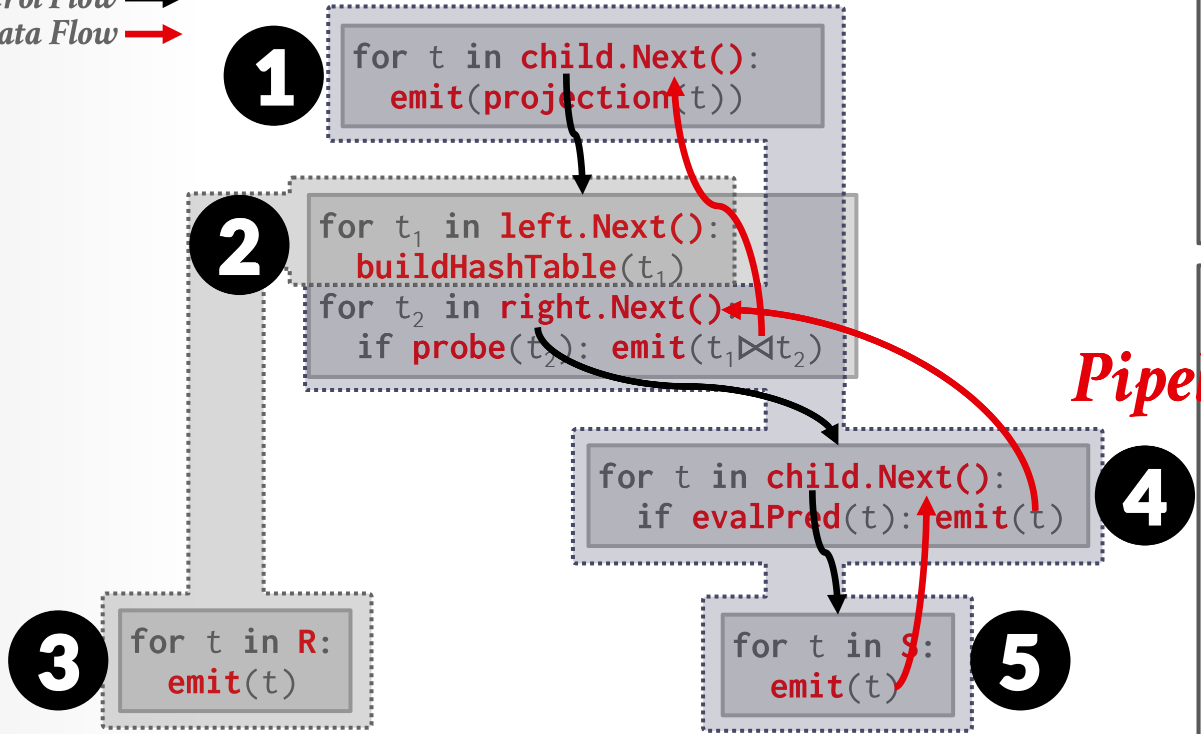
```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



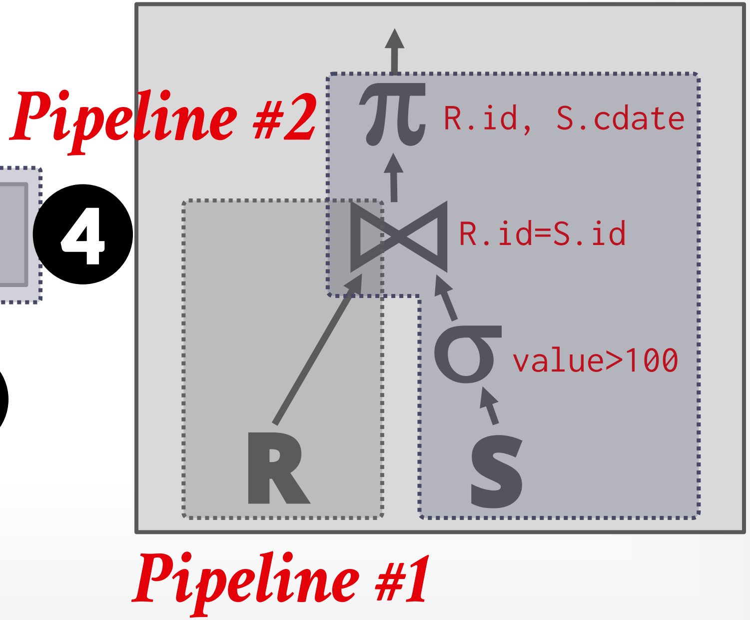
Pipeline #1

ITERATOR MODEL

Control Flow →
Data Flow →




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



ITERATOR MODEL

The Iterator model is used in almost every DBMS.
→ Easy to implement / debug.
→ Output control works easily with this approach.

Allows for **pipelining** where the DBMS tries to process each tuple through as many operators as possible before retrieving the next tuple.

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).

→ Originally developed by MonetDB in the 1990s to process entire columns at a time instead of single tuples.

MATERIALIZATION MODEL

Control Flow →
Data Flow →

1

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

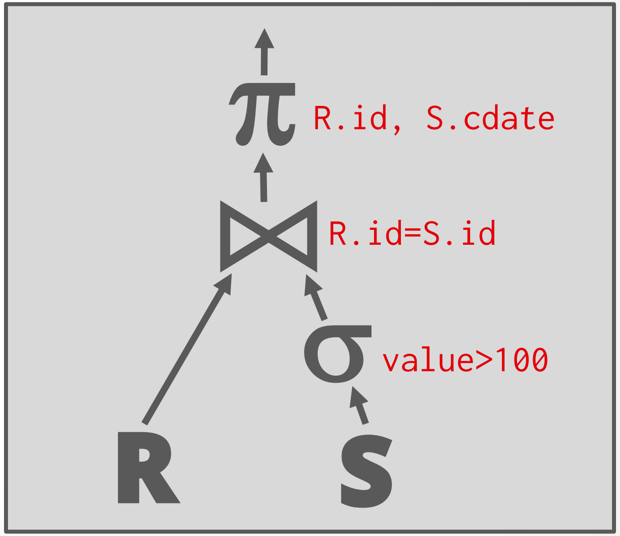
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
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

Control Flow →
Data Flow →

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
```

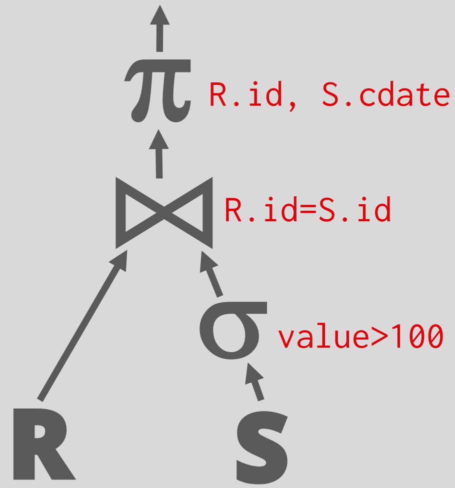
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

3

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

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

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



MATERIALIZATION MODEL

Control Flow →
Data Flow →

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
```

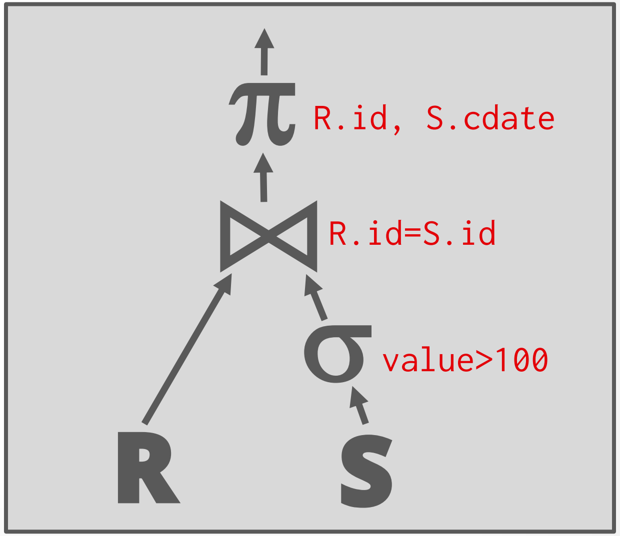
All Tuples

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

Control Flow →
Data Flow →

1
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

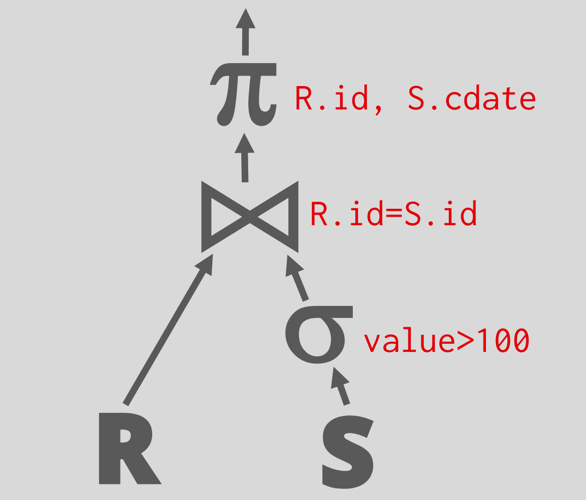
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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
```

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



MATERIALIZATION MODEL

Control Flow →
Data Flow →

1
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

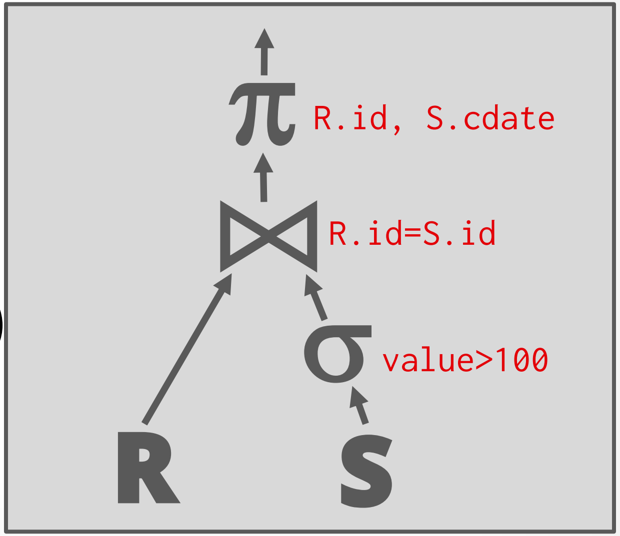
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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
```

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

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



MATERIALIZATION MODEL

Control Flow →
Data Flow →

1
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

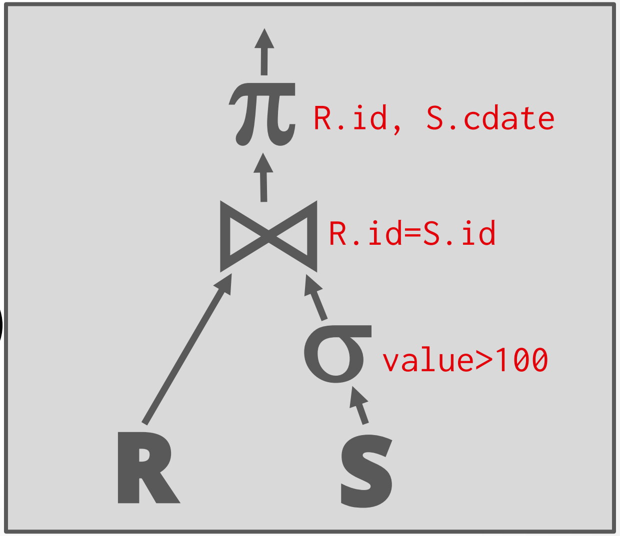
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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

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

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

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



MATERIALIZATION MODEL

Control Flow →
Data Flow →

1

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

2

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

3

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

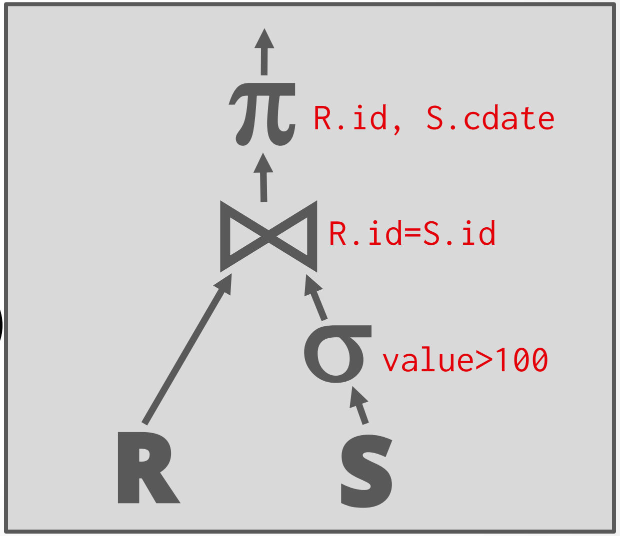
4

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

5

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

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



MATERIALIZATION MODEL

Control Flow →
Data Flow →

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
```

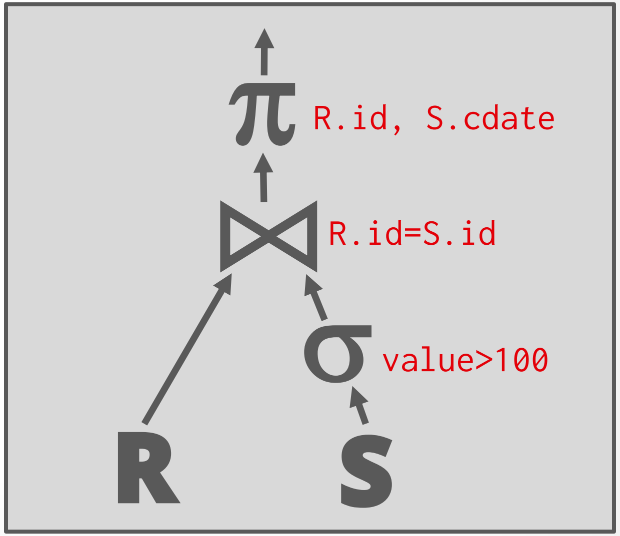
```
out = [ ]
for t in S:
  if evalPred(t): out.add(t)
return out
```

3

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

Operator Fusion

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



MATERIALIZATION MODEL

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

→ Lower execution / coordination overhead.

→ Fewer function calls.

Not ideal for OLAP queries with large intermediate results because DBMS must allocate buffers.

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.
→ Each batch will contain one or more columns each their own null bitmaps.

VECTORIZATION MODEL

Control Flow →
Data Flow →

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)
```

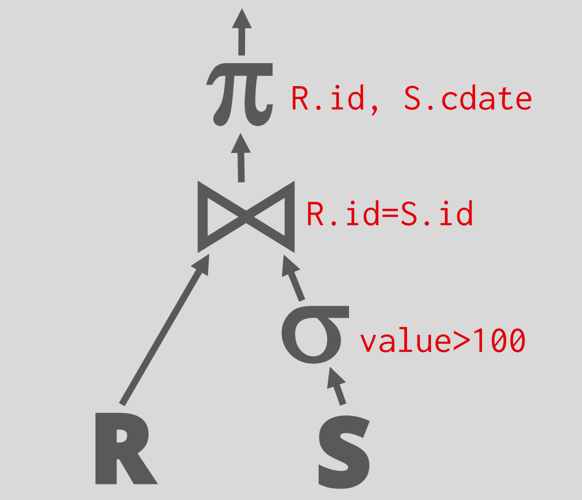
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

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

Control Flow →
Data Flow →

1

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

2

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

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

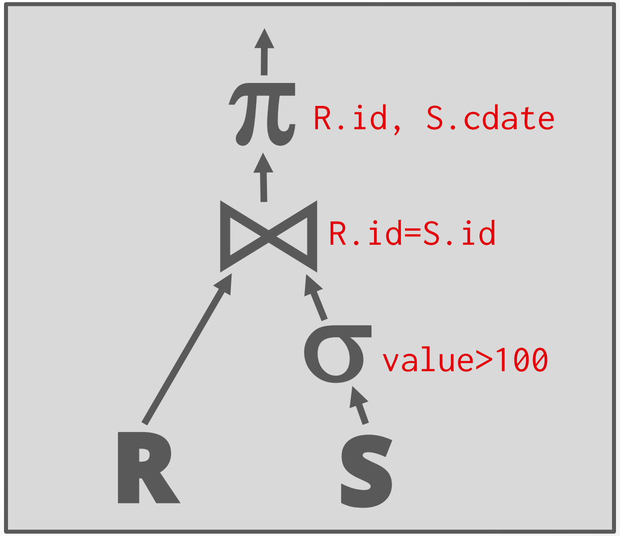
3

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

Tuple Batch

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

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



VECTORIZATION MODEL

Control Flow →
Data Flow →

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

1

```
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)
```

2

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

4

3

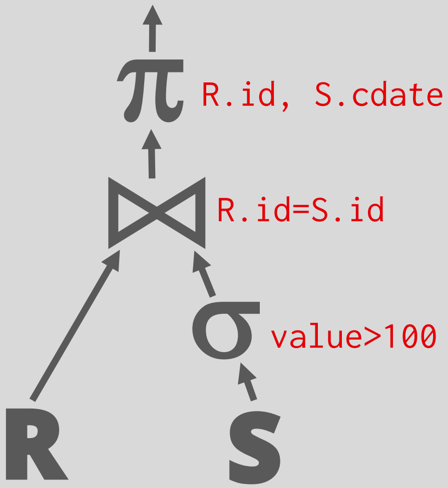
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

Tuple Batch

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

5

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



VECTORIZATION MODEL

Control Flow →
Data Flow →

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

2
```
out = [ ]
for t₁ in left.Next():
  buildHashTable(t₁)
for t₂ in right.Next():
  if probe(t₂): out.add(t₁⨝t₂)
  if |out|>n: emit(out)
```

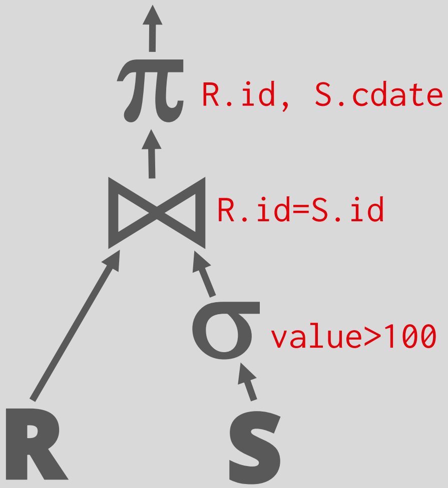
4
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

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

Tuple Batch

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

```
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 an out-of-order CPU to efficiently execute operators over batches of tuples.
→ Operators perform work in tight for-loops over arrays, which compilers know how to optimize / vectorize.
→ No data or control dependencies.
→ Hot instruction cache.

VECTORIZATION MODEL

Ideal for OLAP queries because it greatly reduces the number of invocations per operator.

Allows an out-of-order CPU to efficiently execute operators over batches of tuples.

→ Operators perform work in tight for-loops over arrays, which compilers know how to optimize / vectorize.
→ No data or control dependencies.
→ Hot instruction cache.

Opteryx, presto, Google Big Query, Microsoft SQL Server, vectorwise, ArcticDB, IBM DB2, TERADATA, Yellowbrick, Polars, pinot, AlloyDB, ORACLE, snowflake, FIREBOLT, SingleStore, ClickHouse, amazon REDSHIFT, DuckDB, druid, GREENPLUM DATABASE, databricks, DORIS, MatrixOne, QuestDB, VERTICA, APACHE DRILL, Velox, DATA FUSION, HYRISE, CockroachDB, trino

OBSERVATION

In the previous examples, the DBMS starts executing a query by invoking `Next()` at the root of the query plan and ***pulling*** data up from leaf operators.

This is the how most DBMSs implement their execution engine.

PLAN PROCESSING DIRECTION

Approach #1: Top-to-Bottom (Pull)

→ Start with the root and "pull" data up from its children.
→ Tuples are always passed between operators using function calls (unless it's a pipeline breaker).

Approach #2: Bottom-to-Top (Push)

→ Start with leaf nodes and "push" data to their parents.
→ Can "fuse" operators together within a for-loop to minimize intermediate result staging.

PLAN PROCESSING DIRECTION

Approach #1: Top-to-Bottom (Pull)

→ Start with the root and "pull" data up from its children.
→ Tuples are always passed between operators using function calls (unless it's a pipeline breaker).

Approach #2: Bottom-to-Top (Push)

→ Start with leaf nodes and "push" data to their parents.
→ Can "fuse" operators together within a for-loop to minimize intermediate result staging.

PLAN PROCESSING DIRECTION

Approach #1: Top-to-Bottom (Pull)

→ Start with the root and "pull" data up from its children.

→ Tuples are always passed between operators using function calls (unless it's a pipeline breaker).

Approach #2: Bottom-to-Top (Push)

→ Start with leaf nodes and "push" data to their parents.

→ Can "fuse" operators together within a for-loop to minimize intermediate result staging.

PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

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

2

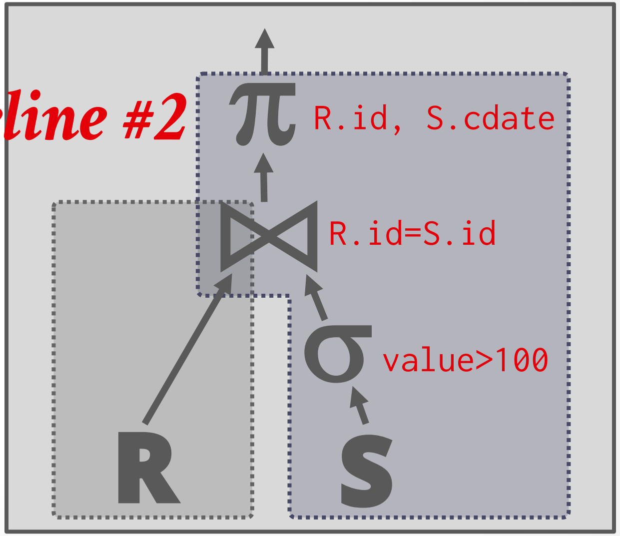
```
for t2 in S:
 if evalPred(t):
  if probeHashTable(t2):
   emit(projection(t1⋈t2))
```

Operator Fusion

1

```
for t1 in R:
 buildHashTable(t1)
```

Pipeline #2



Pipeline #1

PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

Scheduler

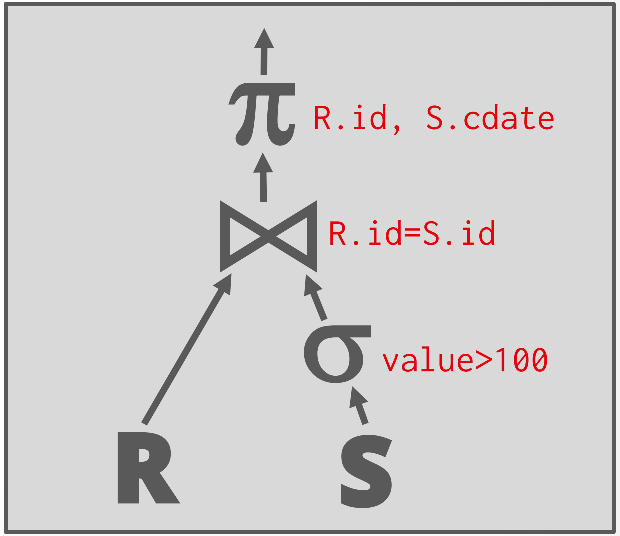
2

```
for t2 in S:
 if evalPred(t):
  if probeHashTable(t2):
   emit(projection(t1⋈t2))
```

1

```
for t1 in R:
 buildHashTable(t1)
```

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



PUSH-BASED ITERATOR MODEL

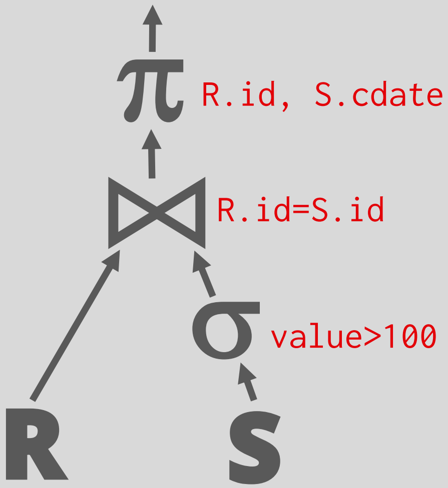
Control Flow →
Data Flow →

Scheduler

2
```
for t₂ in S:
 if evalPred(t):
  if probeHashTable(t₂):
   emit(projection(t₁⋈t₂))
```

1
```
for t₁ in R:
 buildHashTable(t₁)
```

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



PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

Scheduler

2

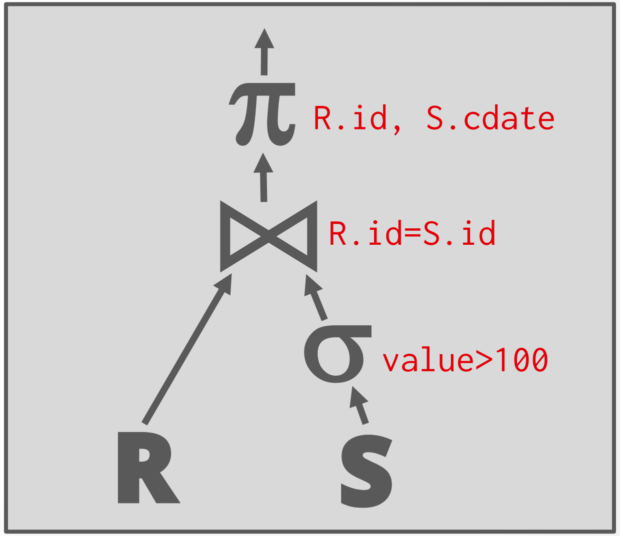
```
for t2 in S:
 if evalPred(t):
  if probeHashTable(t2):
   emit(projection(t1⋈t2))
```

1

```
for t1 in R:
 buildHashTable(t1)
```

Output Buffers

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



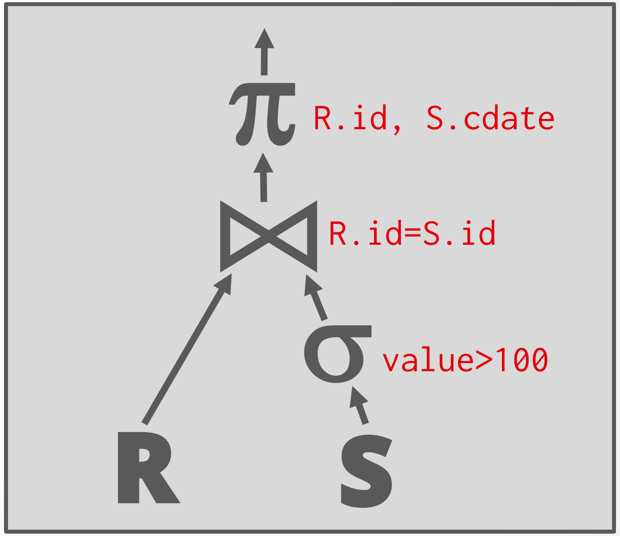
PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

Scheduler

Output Buffers

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



2

```
for t2 in S:
 if evalPred(t):
  if probeHashTable(t2):
   emit(projection(t1⋈t2))
```

1

```
for t1 in R:
 buildHashTable(t1)
```

PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

Scheduler

Output Buffers

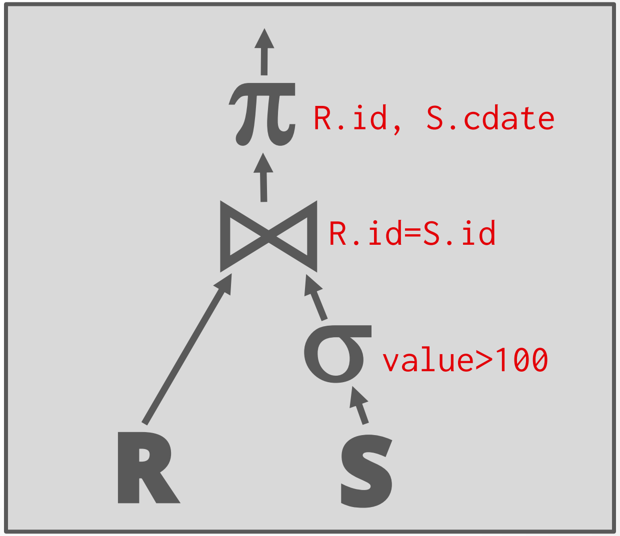
2

```
for t2 in S:
 if evalPred(t):
  if probeHashTable(t2):
   emit(projection(t1⨝t2))
```

1

```
for t1 in R:
 buildHashTable(t1)
```

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



PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

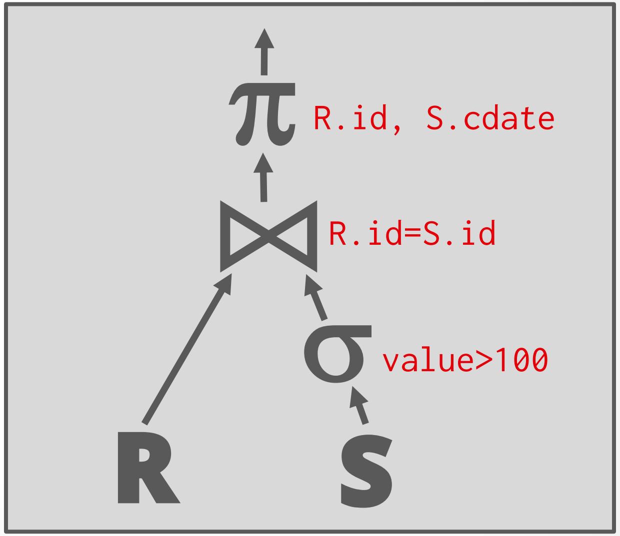
Scheduler

Output Buffers

2
```
for t2 in S:
 if evalPred(t):
  if probeHashTable(t2):
   emit(projection(t1⋈t2))
```

1
```
for t1 in R:
 buildHashTable(t1)
```

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

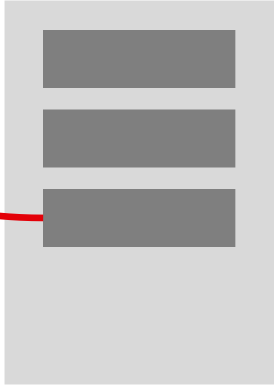


PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

Scheduler

Output Buffers



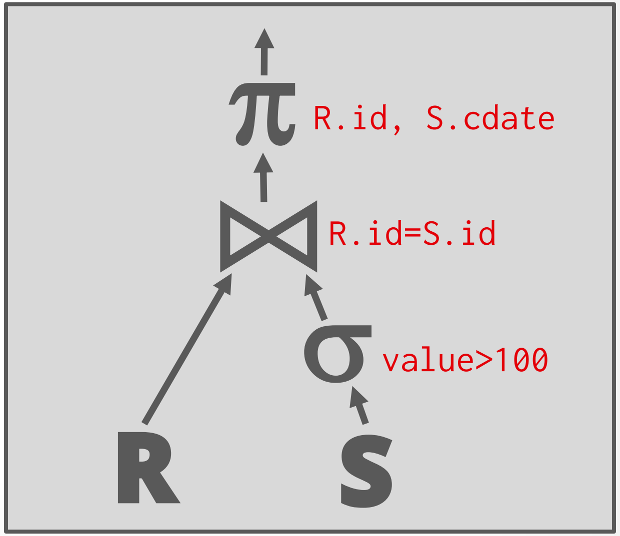
2

```
for t2 in S:
 if evalPred(t):
  if probeHashTable(t2):
   emit(projection(t1⋈t2))
```

1

```
for t1 in R:
 buildHashTable(t1)
```

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



PLAN PROCESSING DIRECTION

Approach #1: Top-to-Bottom (Pull) ← *Most Common*

→ Easy to control output via **LIMIT**.
→ Parent operator blocks until its child returns with a tuple.
→ Additional overhead because operators' **Next()** functions are implemented as virtual functions.
→ Branching costs on each **Next()** invocation.

Approach #2: Bottom-to-Top (Push) ← *Rare-ish*

→ Allows for tighter control of caches/registers in pipelines.
→ May not have exact control of intermediate result sizes.
→ Difficult to implement some operators (Sort-Merge Join).

HyPer UMBRA snowflake DuckDB

ClickHouse FIREBOLT CedarDB

ACCESS METHODS

An **access method** is the how the DBMS retrieves data stored in a table.

→ There is not a specific operator for this defined in relational algebra.

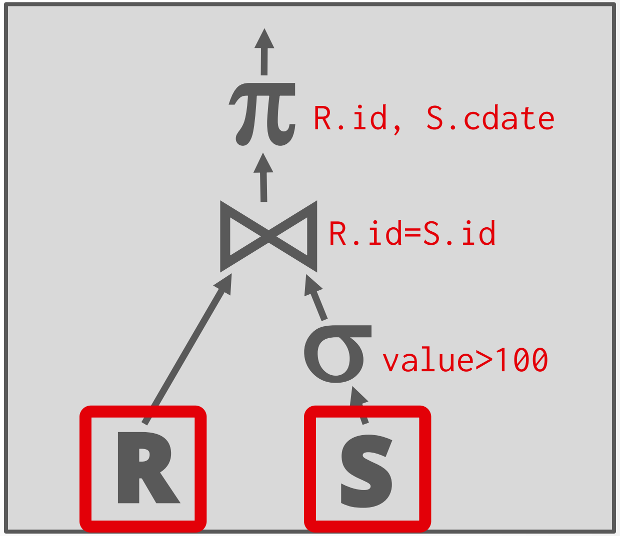
Three basic approaches:

→ Sequential Scan.

→ Index Scan (many variants).

→ Multi-Index Scan.

```sql
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 emit it to the next operator.

```
for page in table.pages:
 for t in page.tuples:
  // Do Something!
```

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

SEQUENTIAL SCAN: OPTIMIZATIONS

- **Lecture #05** Data Encoding / Compression
- **Lecture #06** Prefetching / Scan Sharing / Buffer Bypass
- **Lecture #14** Task Parallelization / Multi-threading
- **Lecture #08** Clustering / Sorting
- **Lecture #12** Late Materialization
- Materialized Views / Result Caching
- Data Skipping
- **Lecture #14** Data Parallelization / Vectorization
- Code Specialization / Compilation

DATA SKIPPING

Approach #1: Approximate Queries (Lossy)

→ Execute queries on a sampled subset of the entire table to produce approximate results.
→ **Examples**: BlinkDB, Redshift, ComputeDB, XDB, Oracle, Snowflake, Google BigQuery, DataBricks

Approach #2: Zone Maps (Lossless)

→ Pre-compute columnar aggregations per page that allow the DBMS to check whether queries need to access it.
→ Trade-off between page size vs. filter efficacy.
→ **Examples**: Oracle, Vertica, SingleStore, Netezza, Snowflake, Google BigQuery

ZONE MAPS

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

```
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

Parquet vortex

Apache orc™ LanceDB

ZONE MA

Pre-computed aggregates for the
page. DBMS checks the zone ma
whether it wants to access the p

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

Original Data

val
100
200
300
400
400

Parquet vortex Apache ORC LanceDB

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

Guido Moerkotte
moer@pi3.informatik.uni-mannheim.de
Lehrstuhl für praktische Informatik III, Universität Mannheim, Germany

Abstract

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

1 Introduction

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

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

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

The main problem encountered is that some queries

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

INDEX SCAN

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

Lecture #15

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'
```

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 available for a query, the DBMS does not have to pick only one:

→ Compute sets of Record IDs using each matching index.
→ Combine these sets based on the query's predicates (union vs. intersect).
→ Retrieve the records and apply any remaining predicates.

Examples:

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

MULTI-INDEX SCAN

Given the following query on a database with an index #1 on **age** and an index #2 on **dept**:

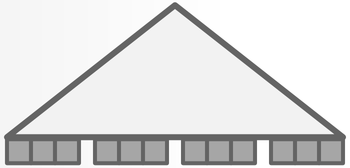
→ We can retrieve the Record IDs satisfying **age<30** using index #1.
→ Then retrieve the Record IDs satisfying **dept='CS'** using index #2.
→ Take their intersection.
→ Retrieve records and check **country='US'**.

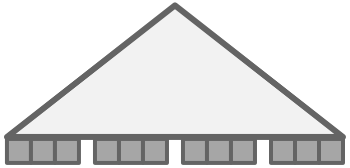
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```

MULTI-INDEX SCAN

Compute set intersection of matching record ids from multiple indexes using bitmaps or hash tables.

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

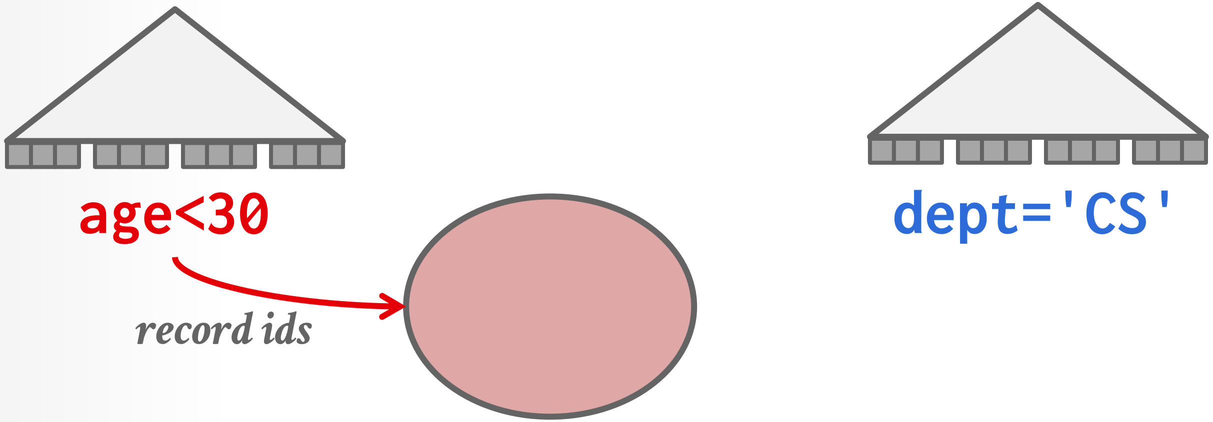




MULTI-INDEX SCAN

Compute set intersection of matching record ids from multiple indexes using bitmaps or hash tables.

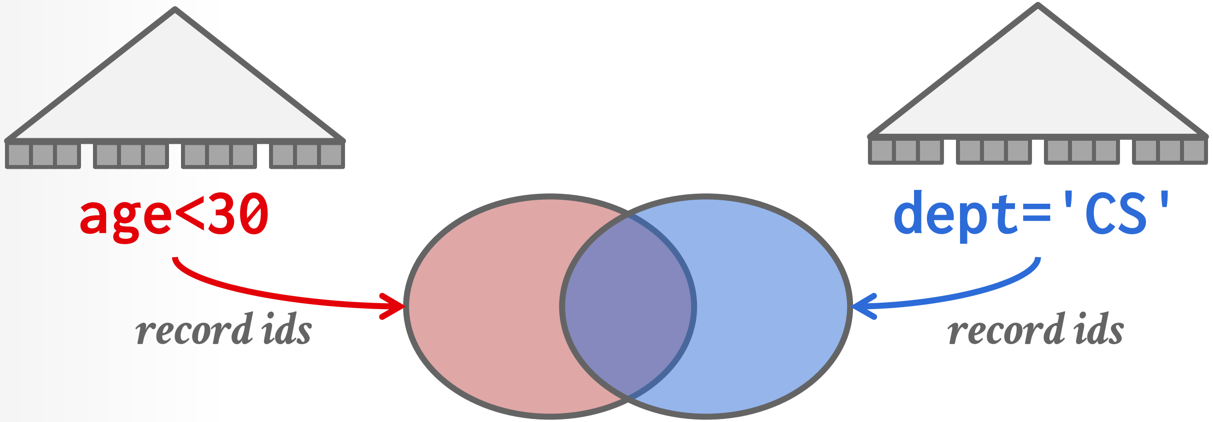
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



MULTI-INDEX SCAN

Compute set intersection of matching record ids from multiple indexes using bitmaps or hash tables.

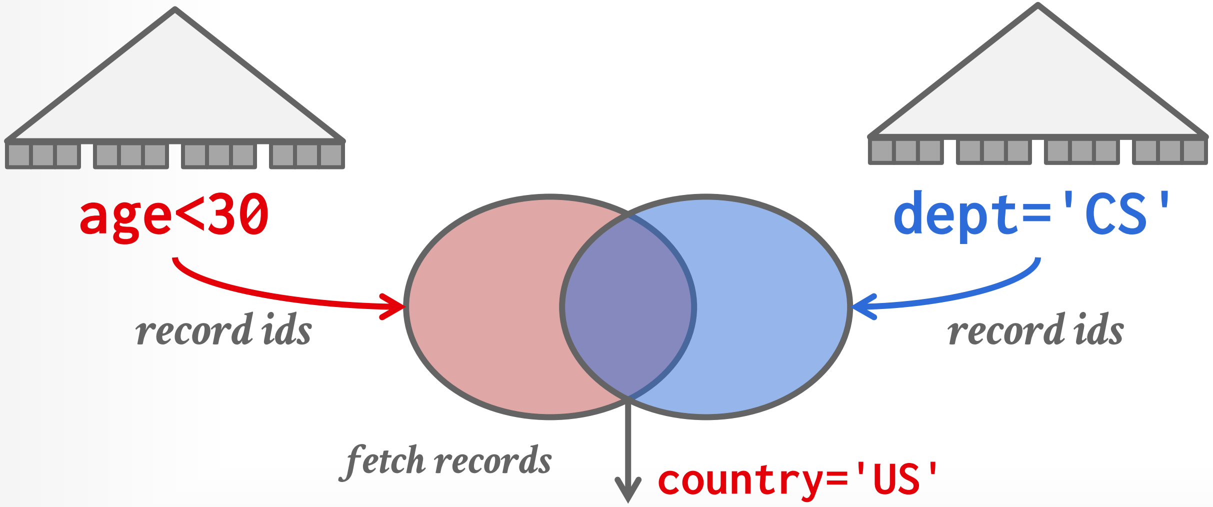
```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



MULTI-INDEX SCAN

Compute set intersection of matching record ids from multiple indexes using bitmaps or hash tables.

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



MODIFICATION QUERIES

Operators that modify the database (**INSERT**, **UPDATE**, **DELETE**) are responsible for modifying the target table and its indexes.

→ Constraint checks can either happen immediately inside of operator or deferred until later in query/transaction.

The output of these operators can either be Record Ids or tuple data (i.e., **RETURNING**).

MODIFICATION QUERIES

UPDATE/DELETE:

→ Child operators pass Record IDs for target tuples.
→ Must keep track of previously seen tuples.

INSERT:

→ **Choice #1**: Materialize tuples inside of the operator.
→ **Choice #2**: Operator inserts any tuple passed in from child operators.

UPDATE QUERY PROBLEM

Control Flow →
Data Flow →

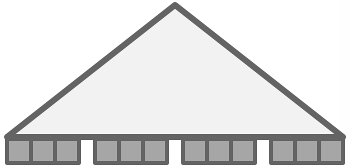
```
CREATE INDEX idx_salary
    ON people (salary);
```

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

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

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

Index(people.salary)



UPDATE QUERY PROBLEM

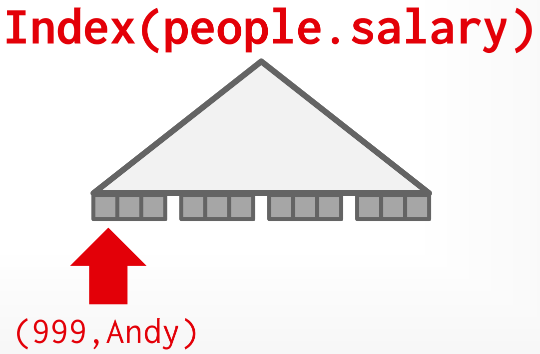
Control Flow →
Data Flow →

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

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

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

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



UPDATE QUERY PROBLEM

Control Flow →
Data Flow →

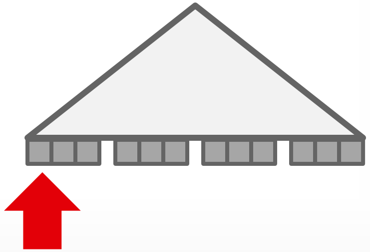
```
for t in child.Next():        (999,Andy)
  removeFromIndex(idx_salary, t.salary, t)
  updateTuple(t.salary = t.salary + 100)
  insertIntoIndex(idx_salary, t.salary, t)
```

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

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

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

Index(people.salary)



UPDATE QUERY PROBLEM

Control Flow →
Data Flow →

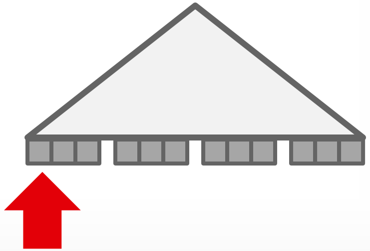
```
CREATE INDEX idx_salary
    ON people (salary);
```

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

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

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

Index(people.salary)



UPDATE QUERY PROBLEM

Control Flow →
Data Flow →

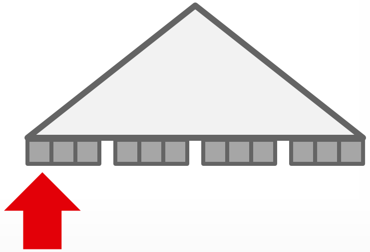
```
CREATE INDEX idx_salary
    ON people (salary);
```

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

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

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

Index(people.salary)



UPDATE QUERY PROBLEM

Control Flow →
Data Flow →

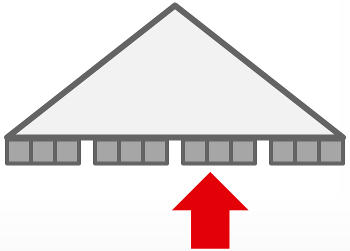
```
for t in child.Next():
  removeFromIndex(idx_salary, t.salary, t)
  updateTuple(t.salary = t.salary + 100)
  insertIntoIndex(idx_salary, t.salary, t)
```

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

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

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

Index(people.salary)



(1099,Andy)

UPDATE QUERY PROBLEM

Control Flow →
Data Flow →

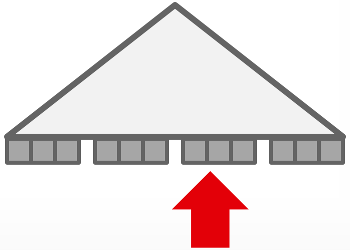
```
for t in child.Next():                (1099,Andy)
  removeFromIndex(idx_salary, t.salary, t)
  updateTuple(t.salary = t.salary + 100)
  insertIntoIndex(idx_salary, t.salary, t)
```

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

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

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

Index(people.salary)



UPDATE QUERY PROBLEM

Control Flow →
Data Flow →

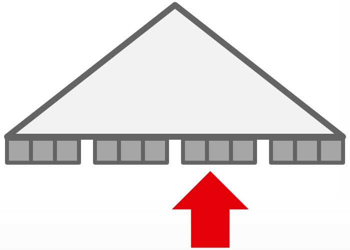
```
for t in child.Next():          (1199,Andy)
  removeFromIndex(idx_salary, t.salary, t)
  updateTuple(t.salary = t.salary + 100)
  insertIntoIndex(idx_salary, t.salary, t)
```

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

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

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

Index(people.salary)



HALLOWEEN PROBLEM

Anomaly where an update operation changes the physical location of a tuple, which causes a scan operator to visit the tuple multiple times.

→ Can occur on clustered tables or index scans.

First discovered by IBM researchers while working on System R on Halloween day in 1976.

Solution: Track modified record ids per query.

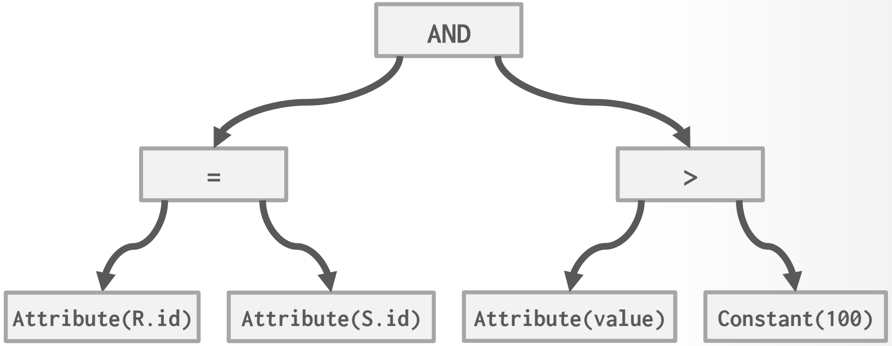
EXPRESSION EVALUATION

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
- → Functions

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



EXPRESSION EVALUATION

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

```
EXECUTE xxx(991)
```

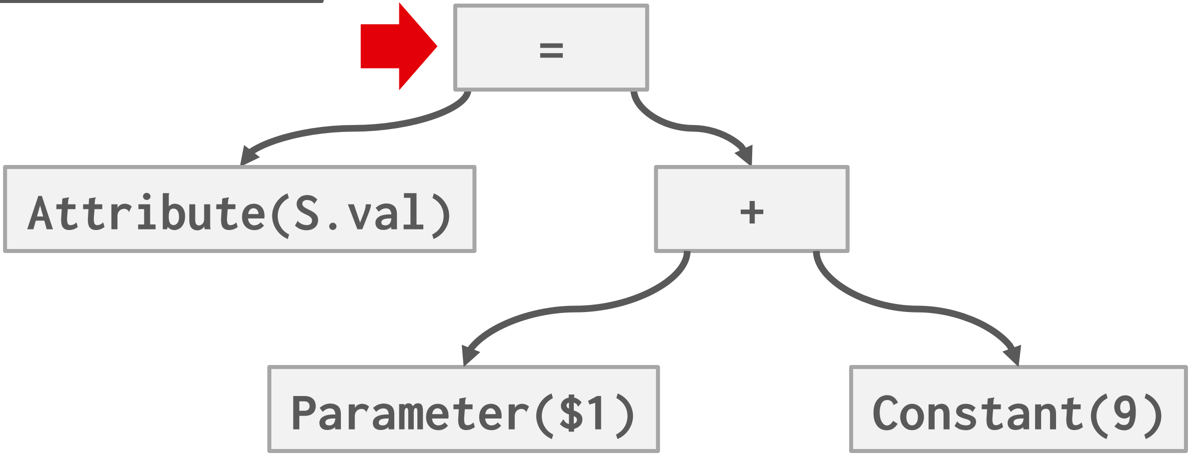
EXPRESSION EVALUATION

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

```
EXECUTE xxx(991)
```

Execution Context

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



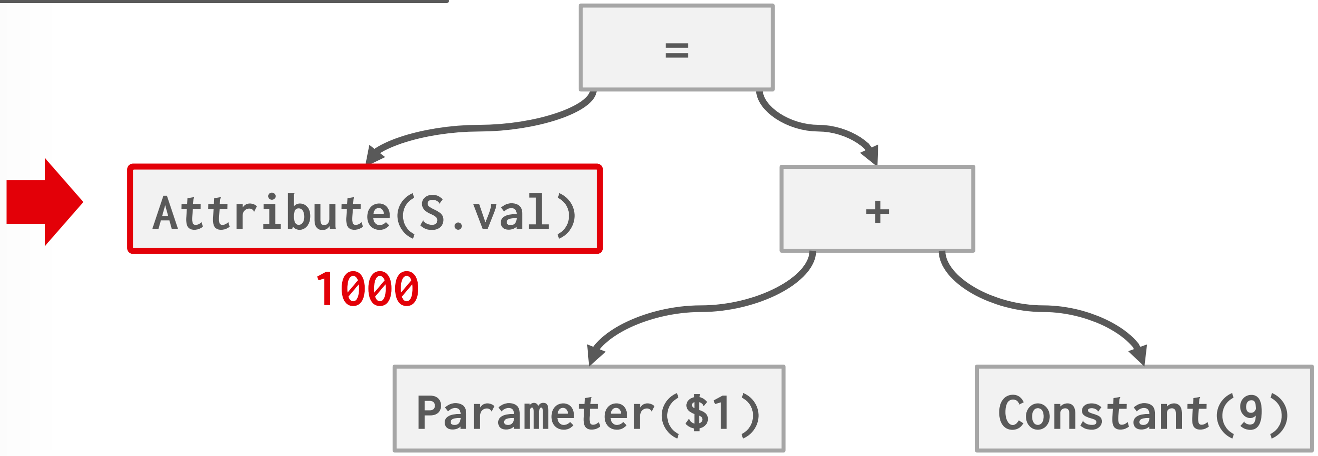
EXPRESSION EVALUATION


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

```
EXECUTE xxx(991)
```

Execution Context

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



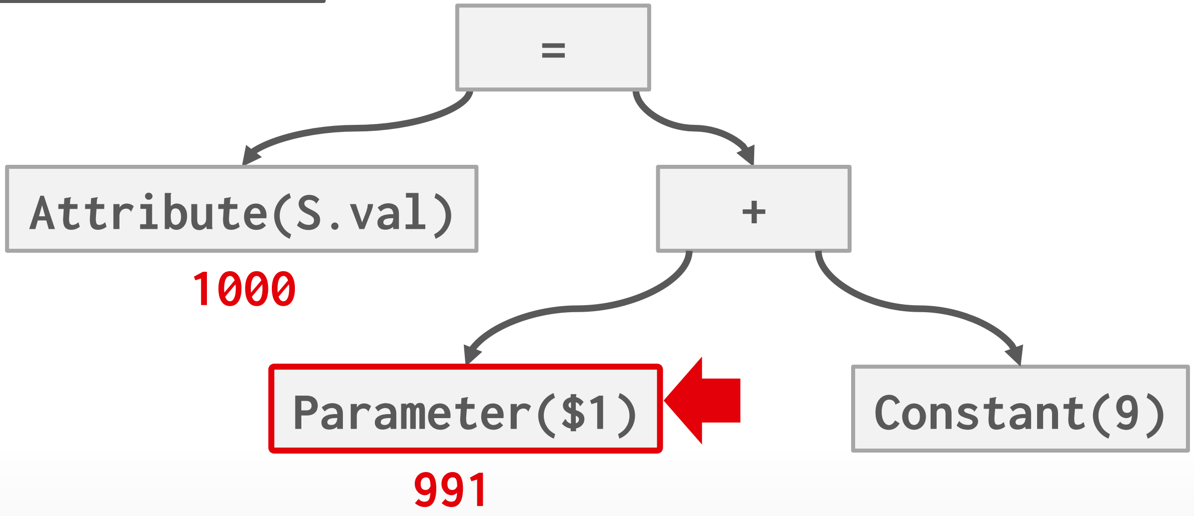
EXPRESSION EVALUATION

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

```
EXECUTE xxx(991)
```

Execution Context

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



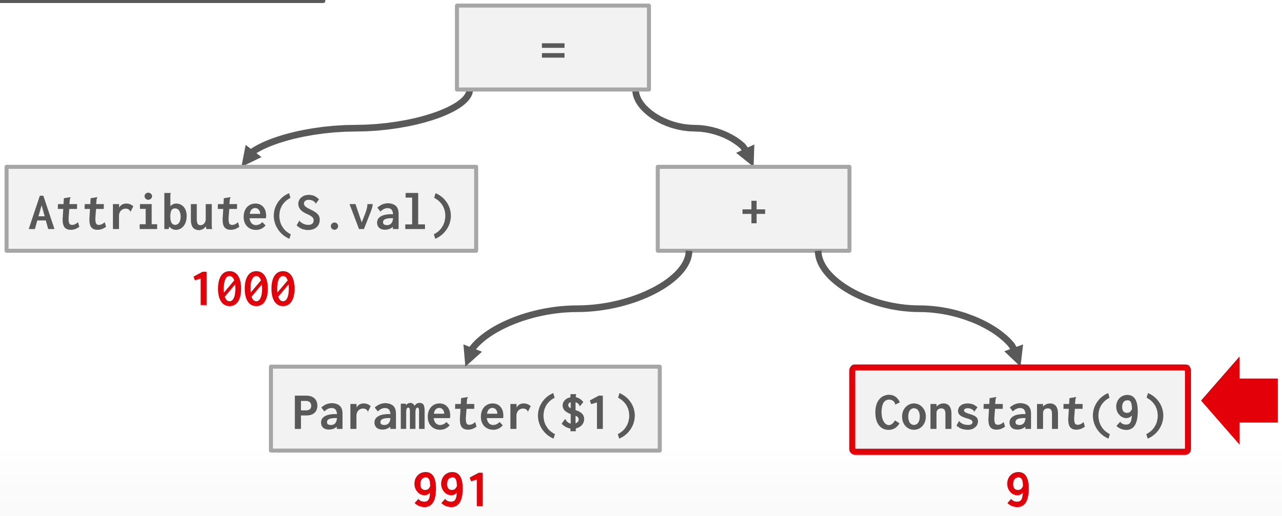
EXPRESSION EVALUATION

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

```
EXECUTE xxx(991)
```

Execution Context

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



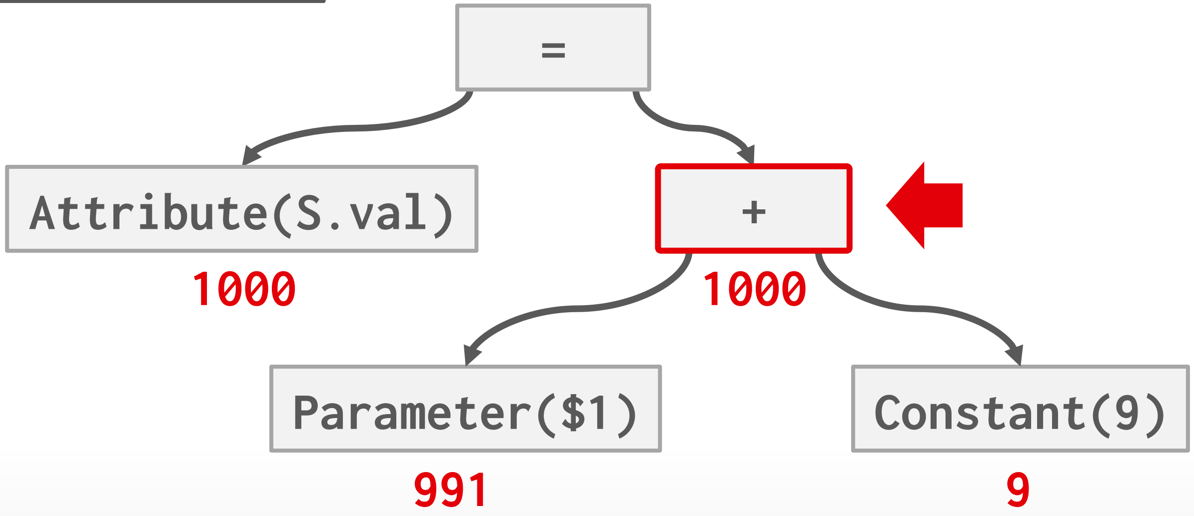
EXPRESSION EVALUATION

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

```
EXECUTE xxx(991)
```

Execution Context

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



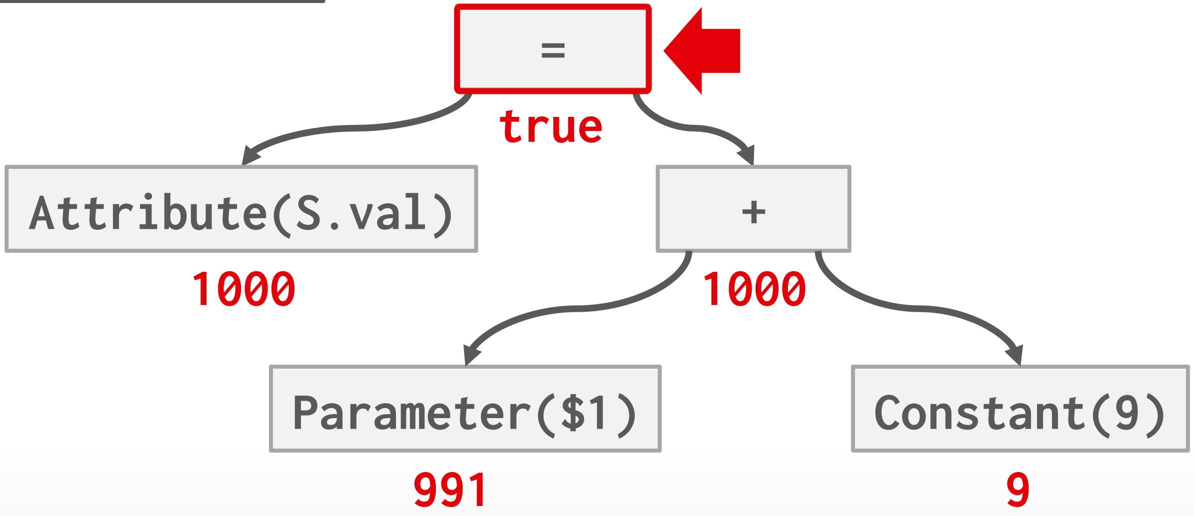
EXPRESSION EVALUATION

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

```
EXECUTE xxx(991)
```

Execution Context

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



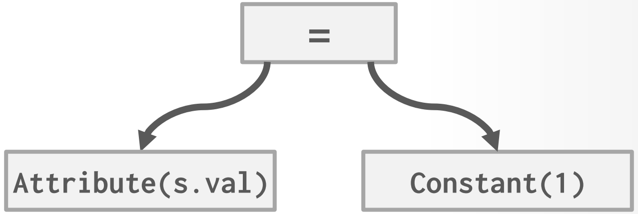
EXPRESSION EVALUATION

Evaluating predicates by traversing a tree is terrible for the CPU.
→ The DBMS traverses the tree and for each node that it visits, it must figure out what the operator needs to do.

A better approach is to evaluate the expression directly.

An even better approach is to **vectorize** it evaluate a batch of tuples at the same time…

```
SELECT * WHERE s.val = 1;
```




```
bool check(val) {
  return (val == 1);
}
```

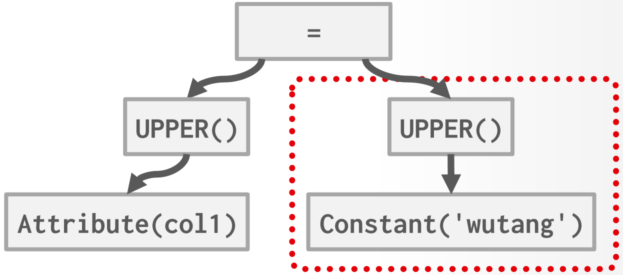
EXPRESSION EVALUATION: OPTIMIZATIONS


```
WHERE UPPER(col1) = UPPER('wutang');
```

Constant Folding:

→ Identify redundant / unnecessary operations that are wasteful.

→ Compute a sub-expression on a constant value once and reuse result per tuple.

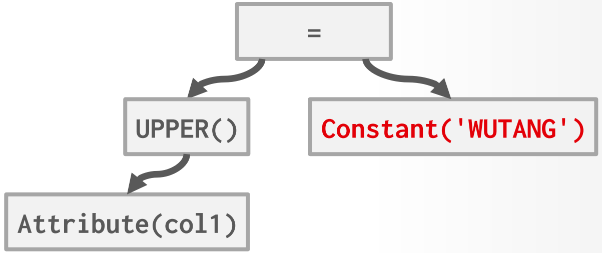


EXPRESSION EVALUATION: OPTIMIZATIONS

```
WHERE UPPER(col1) = UPPER('wutang');
```

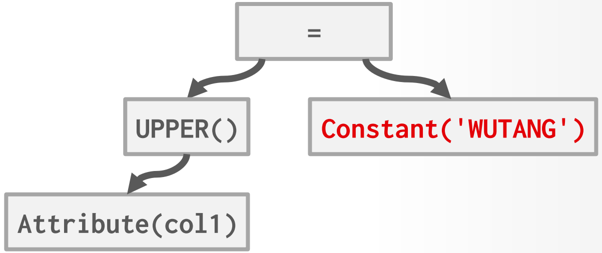
Constant Folding:

→ Identify redundant / unnecessary operations that are wasteful.
→ Compute a sub-expression on a constant value once and reuse result per tuple.



EXPRESSION EVALUATION: OPTIMIZATIONS

```sql
WHERE UPPER(col1) = UPPER('wutang');
```

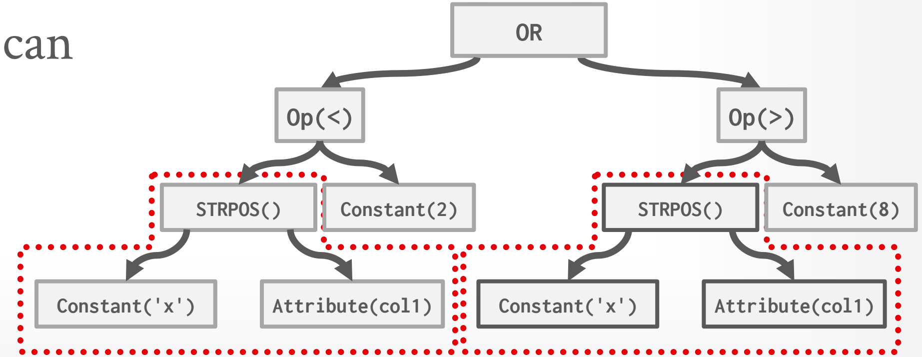


Constant Folding:

→ Identify redundant / unnecessary operations that are wasteful.

→ Compute a sub-expression on a constant value once and reuse result per tuple.

```sql
WHERE STRPOS('x', col1) < 2
   OR STRPOS('x', col1) > 8
```



Common Sub-Expr. Elimination:

→ Identify repeated sub-expressions that can be shared across expression tree.

→ Compute once and then reuse result.

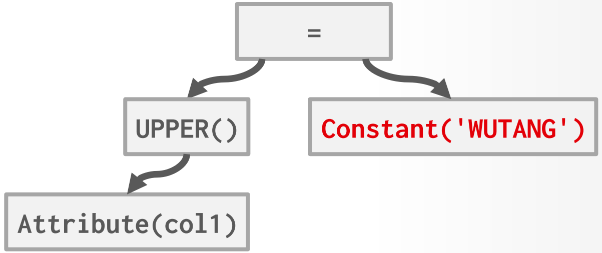
EXPRESSION EVALUATION: OPTIMIZATIONS

Constant Folding:

→ Identify redundant / unnecessary operations that are wasteful.

→ Compute a sub-expression on a constant value once and reuse result per tuple.

```
WHERE UPPER(col1) = UPPER('wutang');
```

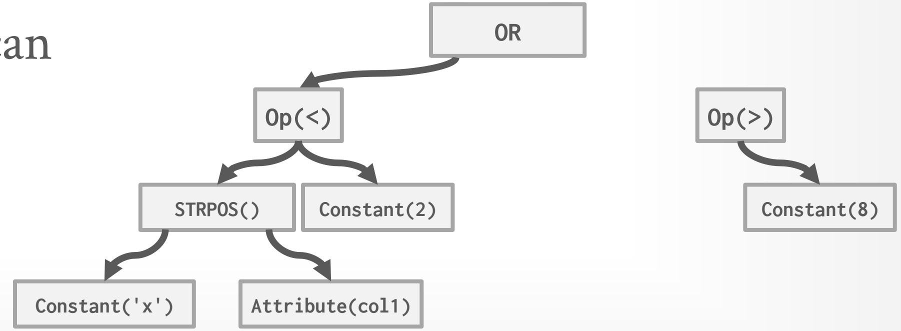


Common Sub-Expr. Elimination:

→ Identify repeated sub-expressions that can be shared across expression tree.

→ Compute once and then reuse result.

```
WHERE STRPOS('x', col1) < 2
   OR STRPOS('x', col1) > 8
```



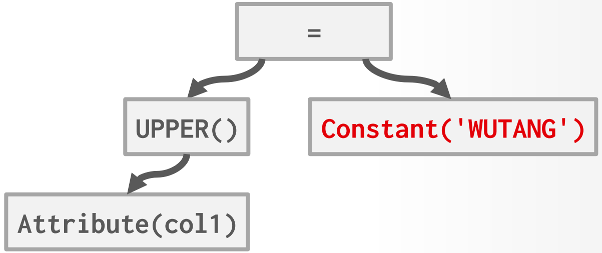
EXPRESSION EVALUATION: OPTIMIZATIONS

Constant Folding:

→ Identify redundant / unnecessary operations that are wasteful.

→ Compute a sub-expression on a constant value once and reuse result per tuple.

```
WHERE UPPER(col1) = UPPER('wutang');
```

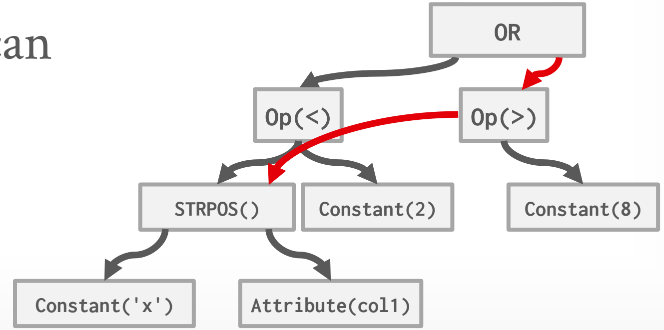


Common Sub-Expr. Elimination:

→ Identify repeated sub-expressions that can be shared across expression tree.

→ Compute once and then reuse result.

```
WHERE STRPOS('x', col1) < 2
   OR STRPOS('x', col1) > 8
```



CONCLUSION

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

OLTP DBMSs want to use index scans as much as possible.

Expression trees are flexible but slow.
JIT compilation can (sometimes) speed them up.

NEXT CLASS

Parallel Query Execution