Query execution & processing

Iztok Savnik, FAMNIT

October, 2025.

Sources

- Course:
  - Carnegie Mellon University (CMU)
  - Advanced Database Systems
  - Prof. Andy Pavlo
  - Transparencencies
- Papers:
  - Conferences VLDB, SIGMOD, CIDR, etc.
  - Journals ACM TODS, IS, VLDBJ, etc.

EXECUTION OPTIMIZATION

DBMS engineering is an orchestration of a bunch of optimizations that seek to make full use of hardware. There is not a single technique that is more important than others.

Techniques we will study:

- Data Parallelization (Vectorization)
- Task Parallelization (Multi-threading)
- Code Specialization (Pre-Compiled / JIT)

OPTIMIZATION GOALS

Approach #1: Reduce Instruction Count

– Use fewer instructions to do the same amount of work.

Approach #2: Reduce Cycles per Instruction

– Execute more CPU instructions in fewer cycles.

Approach #3: Parallelize Execution

– Use multiple threads to compute each query in parallel.

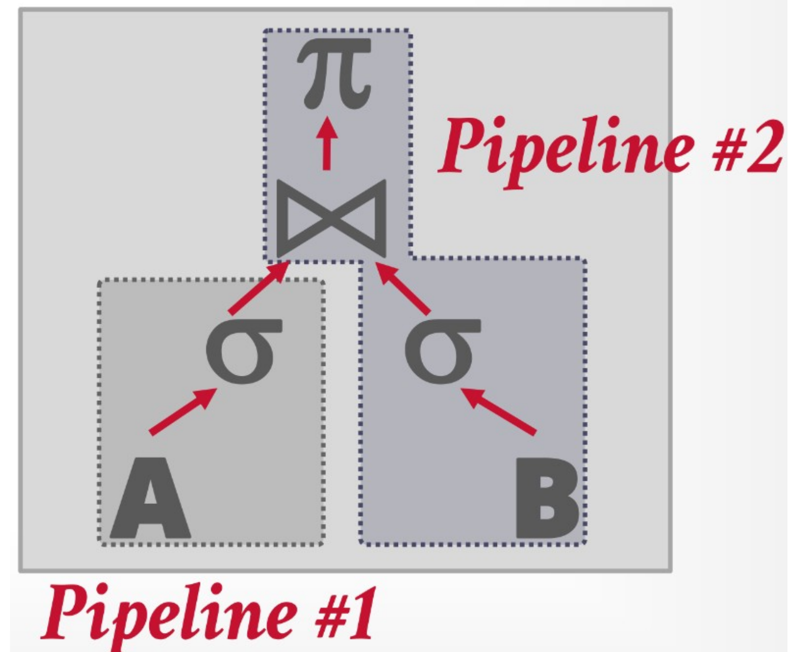
QUERY EXECUTION

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

An **operator instance** is an invocation of an operator on a unique segment of data.

A **task** is a sequence of one or more operator instances.

A **task set** is the collection of executable tasks for a logical pipeline.



Outline

MonetDB/X100 Analysis

Processing Models

Plan Processing Direction

Filter Representation

Types of parallelism

MONETDB/X100 (2005)

Low-level analysis of execution bottlenecks for in-memory DBMSs on OLAP workloads.

- Show how DBMS are designed incorrectly for modern CPU architectures.

Based on these findings, they proposed a new DBMS called MonetDB/X100.

- Renamed to Vectorwise and acquired by Actian in 2010.
- Rebranded as Vector and Avalanche.

CPU OVERVIEW

CPUs organize instructions into **pipeline stages**.

The goal is to keep all parts of the processor busy at each cycle by masking delays from instructions that cannot complete in a single cycle.

Super-scalar CPUs support multiple pipelines.

- Execute multiple instructions in parallel in a single cycle if they are independent (**out-of-order execution**).

Everything is fast until there is a mistake

DBMS / CPU PROBLEMS

Problem #1: **Dependencies**

- If one instruction depends on another instruction, then it cannot be pushed immediately into the same pipeline.

Problem #2: **Branch Prediction**

- The CPU tries to predict what branch the program will take and fill in the pipeline with its instructions.
- If it gets it wrong, it must throw away any speculative work and flush the pipeline.

BRANCH MISPREDICTION

Because of long pipelines, CPUs will speculatively execute branches. This potentially hides the long stalls between dependent instructions.

The most executed branching code in a DBMS is the filter operation during a sequential scan.

But this is (nearly) impossible to predict correctly.

SELECTION SCANS

```
SELECT * FROM table
 WHERE key > $(low)
   AND key < $(high)
```

SELECTION SCANS

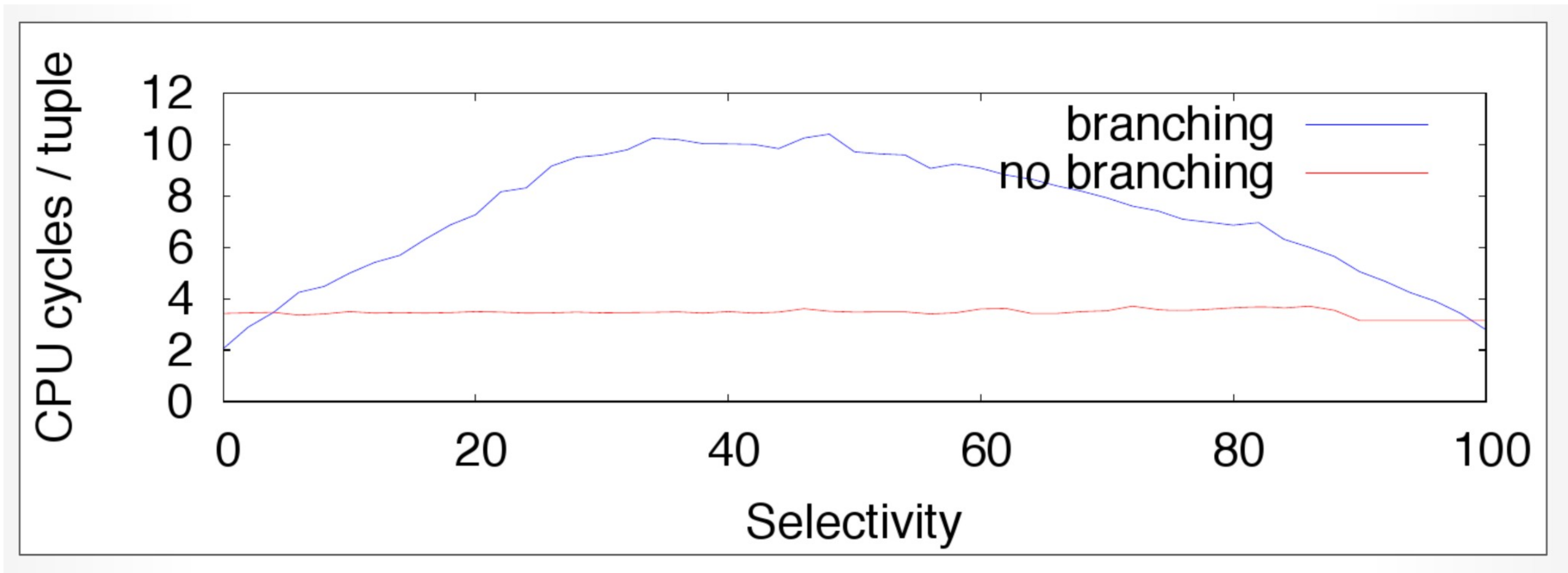
Scalar (Branching)

```
i = 0
for t in table:
  key = t.key
  if (key>low) && (key<high):
    copy(t, output[i])
    i = i + 1
```

Scalar (Branchless)

```
i = 0
for t in table:
  copy(t, output[i])
  key = t.key
  delta = (key>low ? 1 : 0) &
          ↳(key<high ? 1 : 0)
  i = i + delta
```

SELECTION SCANS



EXCESSIVE INSTRUCTIONS

The DBMS needs to support different data types, so it must check a values type before it performs any operation on that value.

- This is usually implemented as giant switch statements.
- Also creates more branches that can be difficult for the CPU to predict reliably.

Example: Postgres' addition for NUMERIC types.

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

Approach #2: Materialization Model

Approach #3: Vectorized / Batch Model

ITERATOR MODEL

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

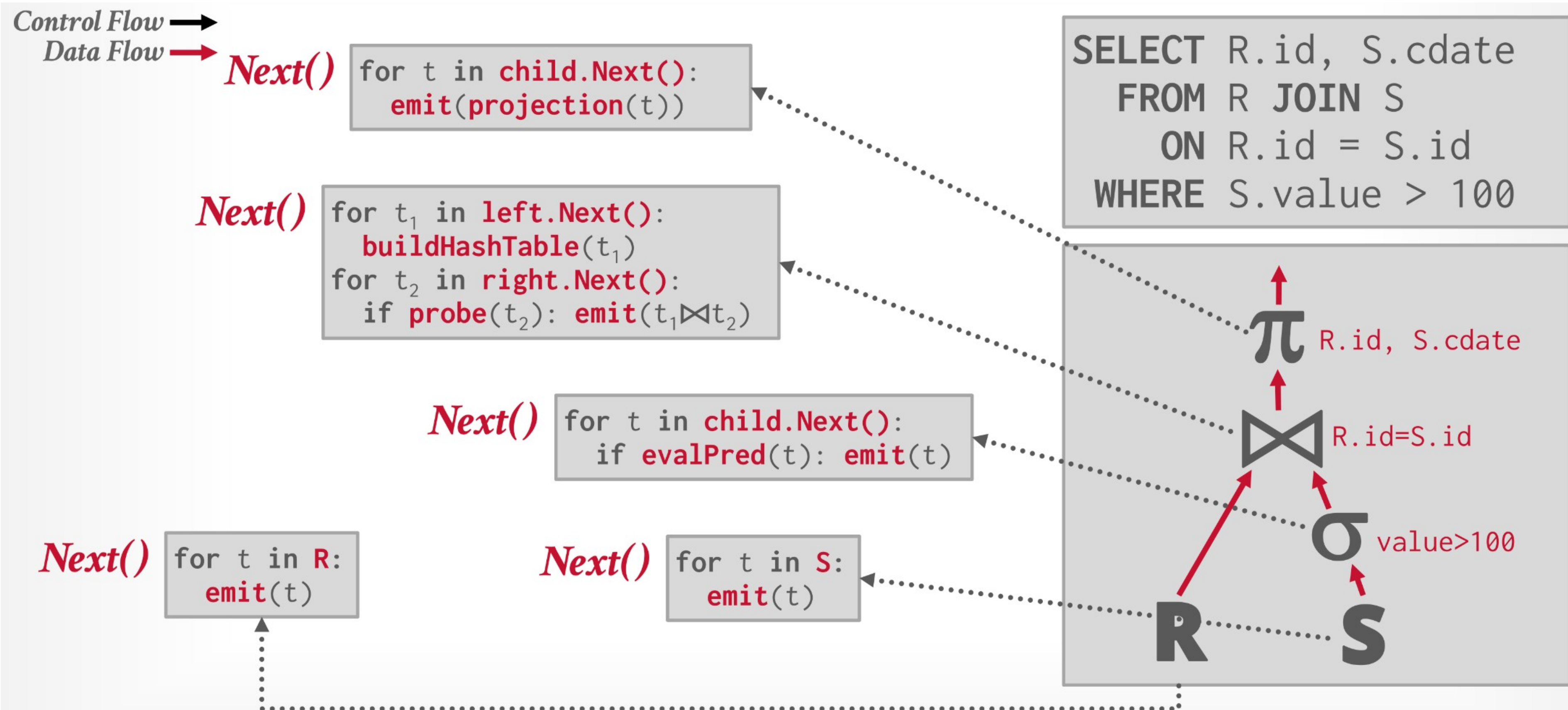
- On each invocation, the operator returns either a single tuple or a EOF marker if there are no more tuples.
- The operator implements a loop that calls next on its children to retrieve their tuples and then process them.

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

- Analogous to constructors/destructors, but for operators.

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

ITERATOR MODEL



ITERATOR MODEL

Control Flow →
Data Flow →

1

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

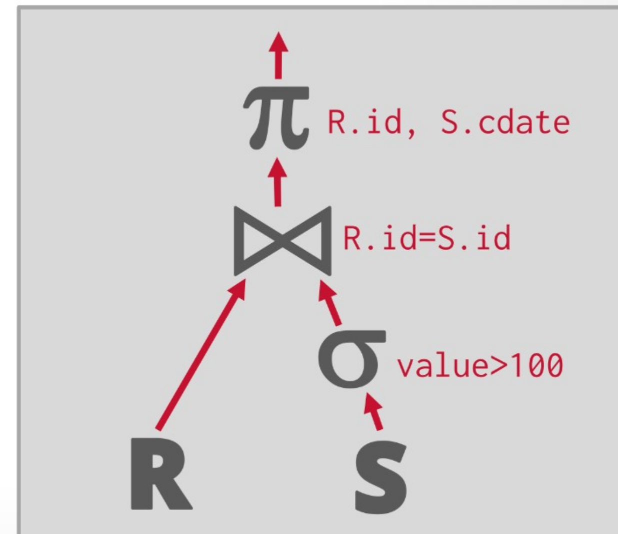
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

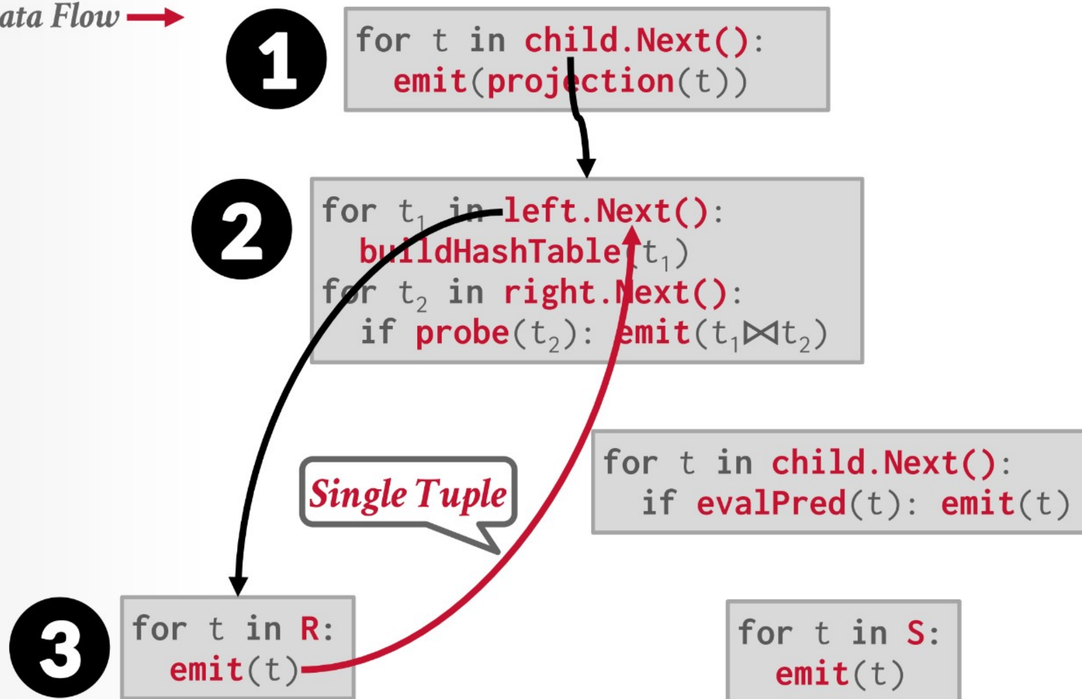
```
for t in R:
  emit(t)
```

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

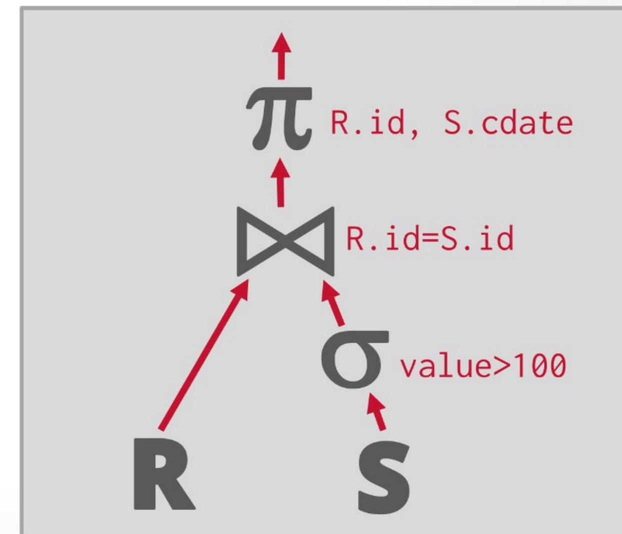
```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



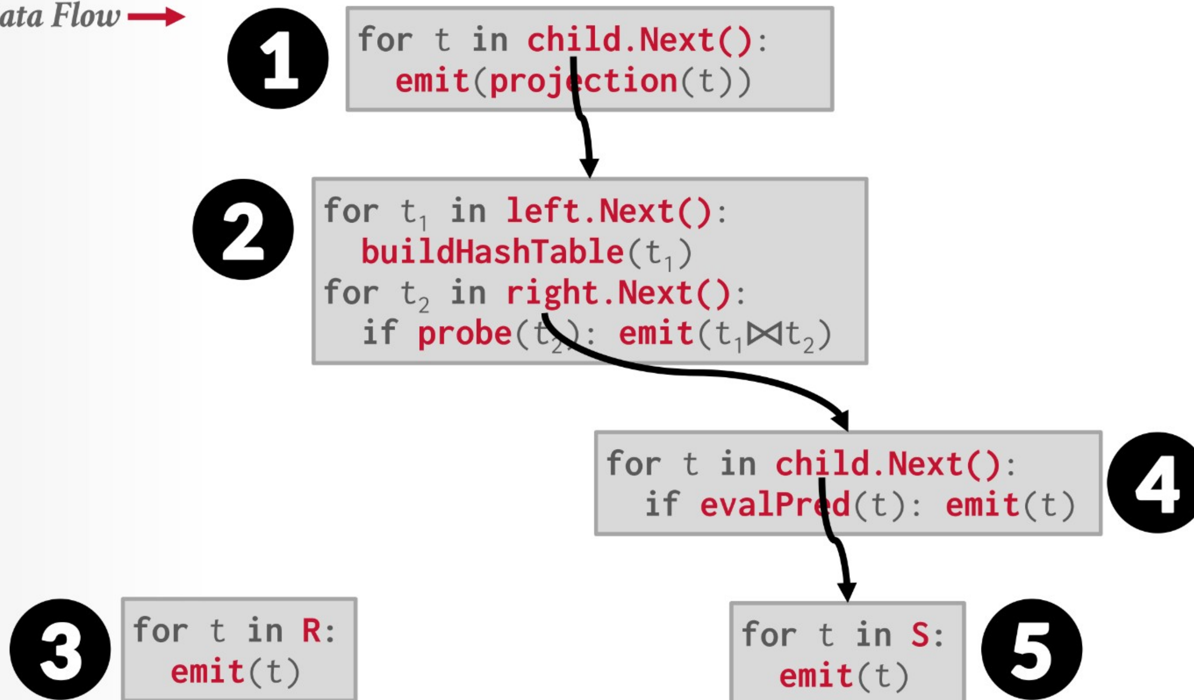
ITERATOR MODEL

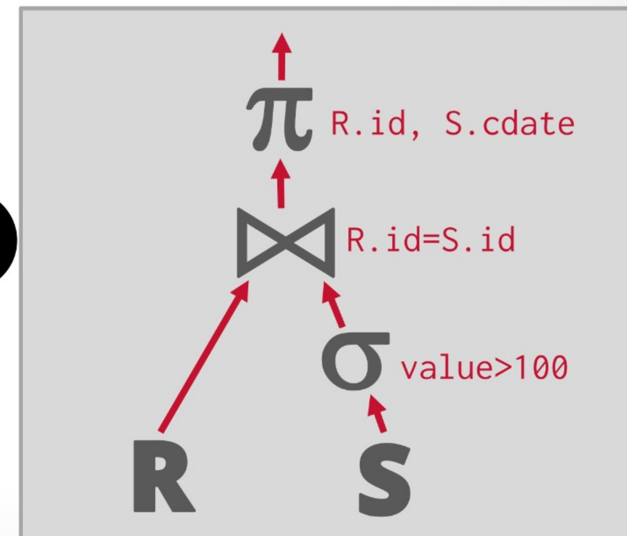


SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100

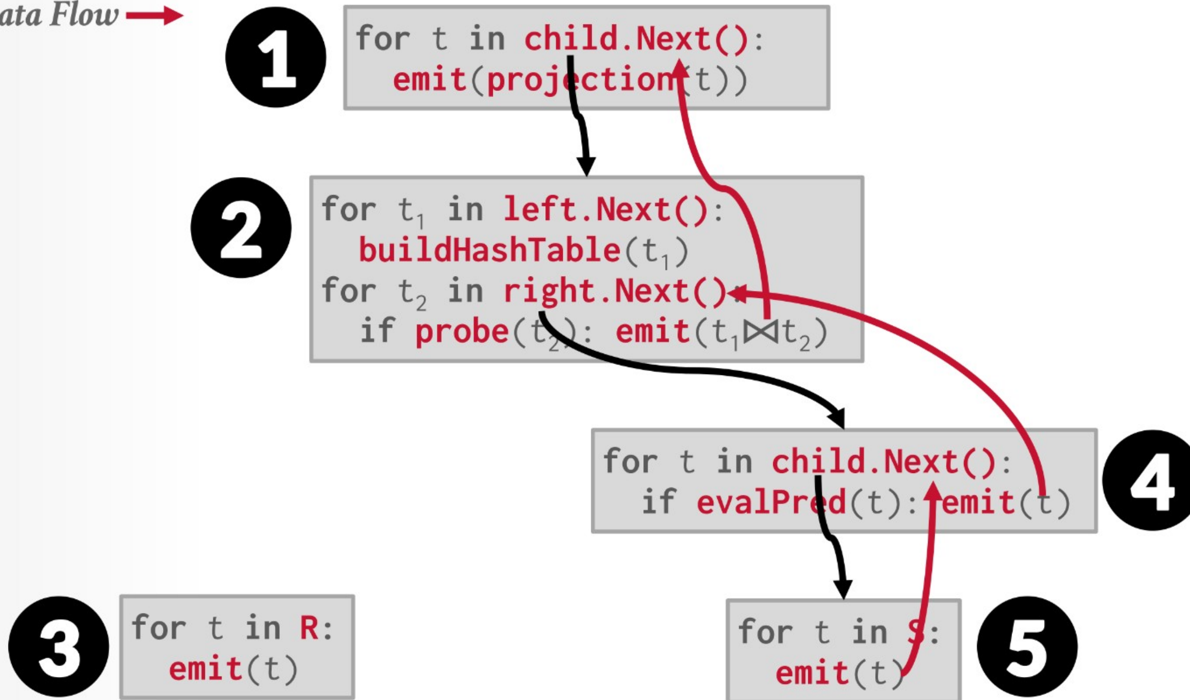


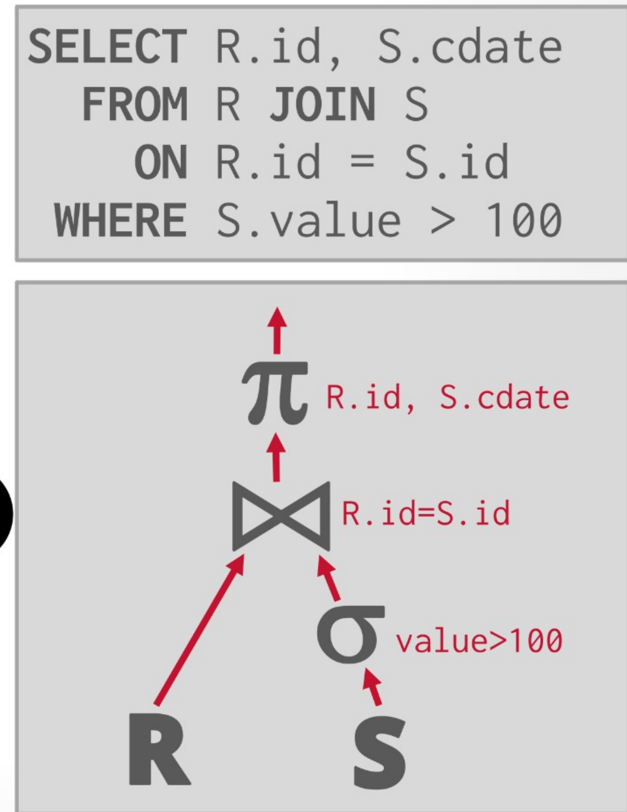
ITERATOR MODEL



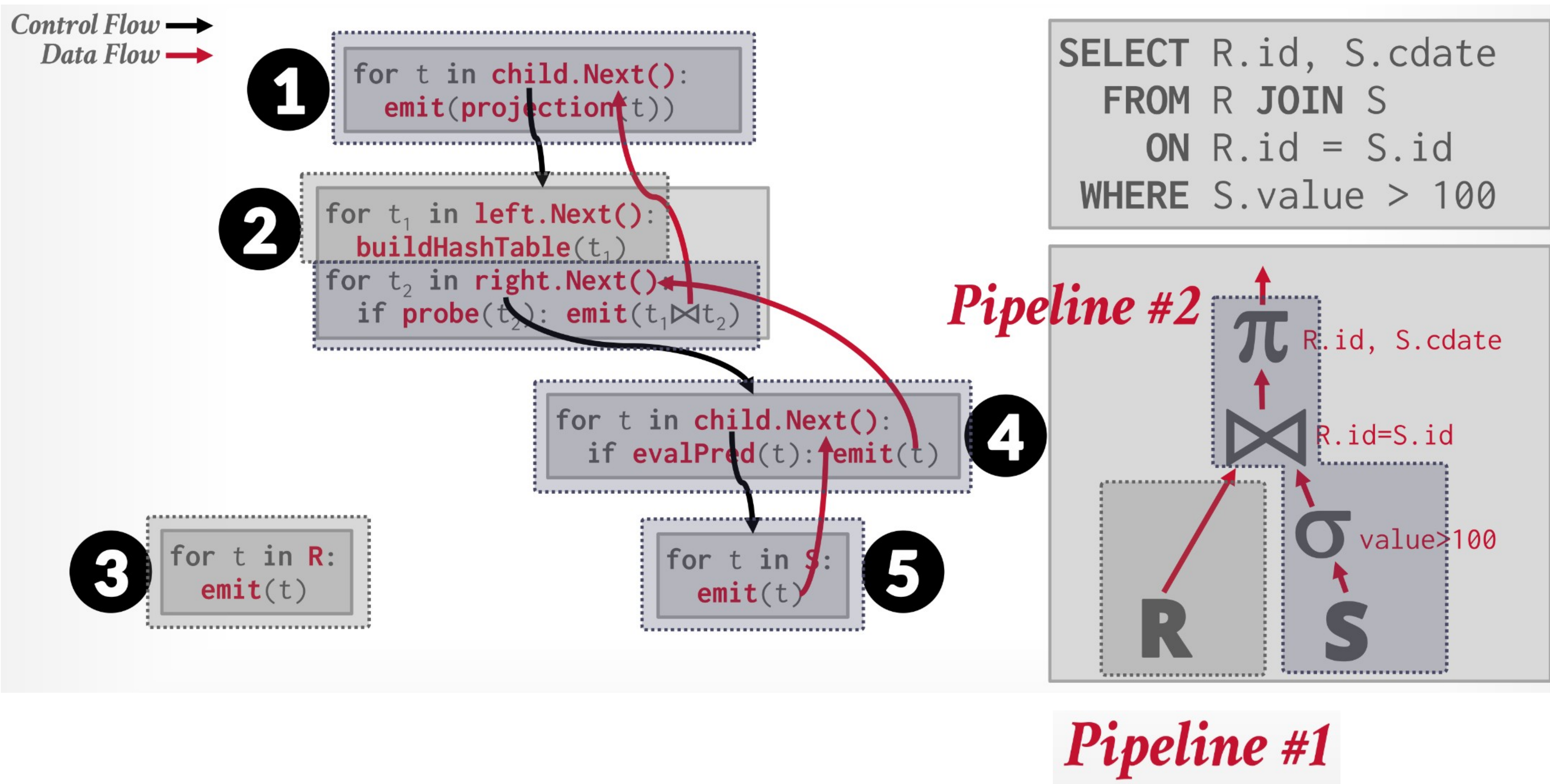


ITERATOR MODEL





ITERATOR MODEL



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 retrieivng the next tuple.

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

– Joins (Build Side), Subqueries, Order By

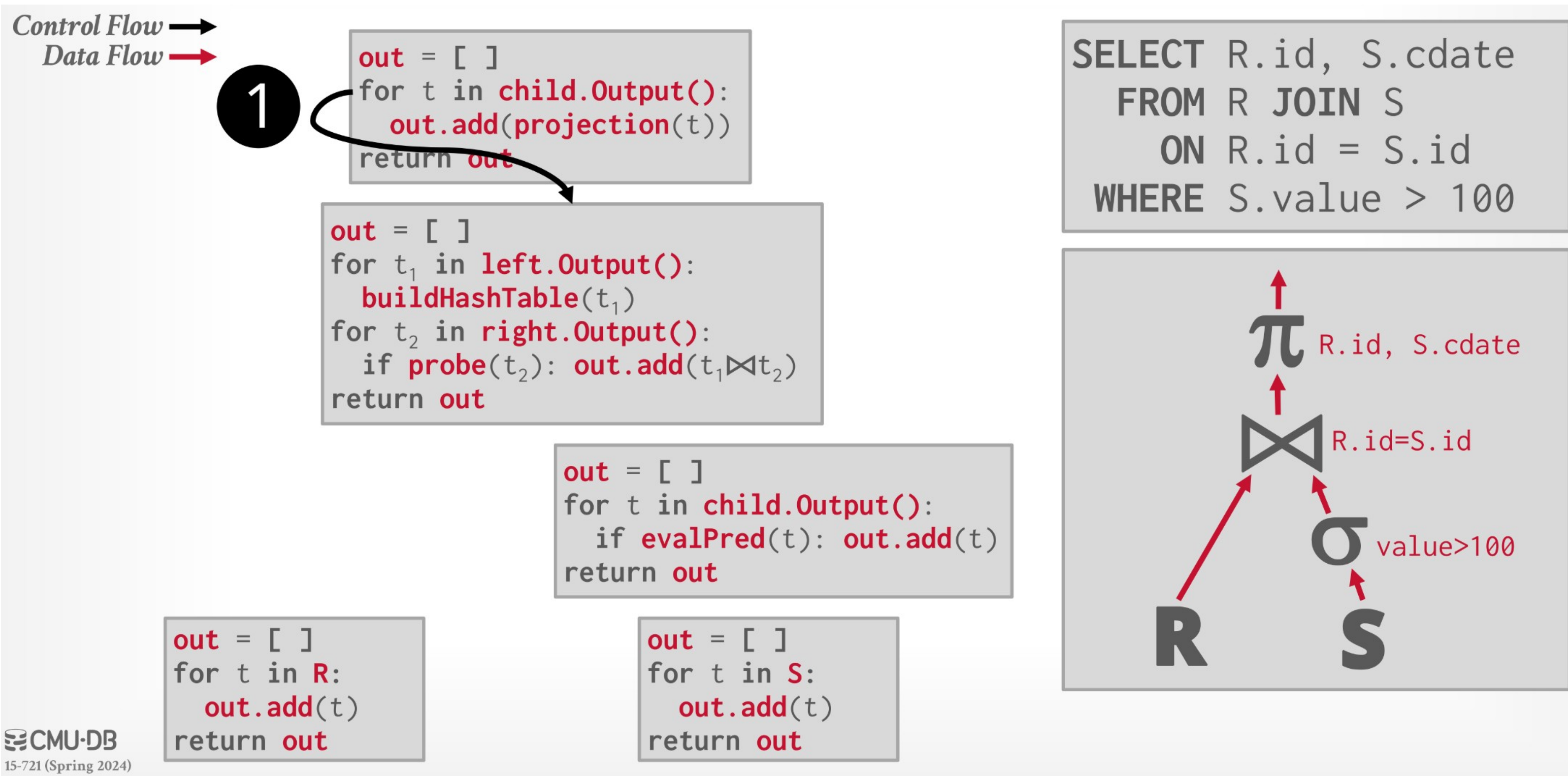
MATERIALIZATION MODEL

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.

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

MATERIALIZATION MODEL



MATERIALIZATION MODEL

Control Flow →
Data Flow →

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

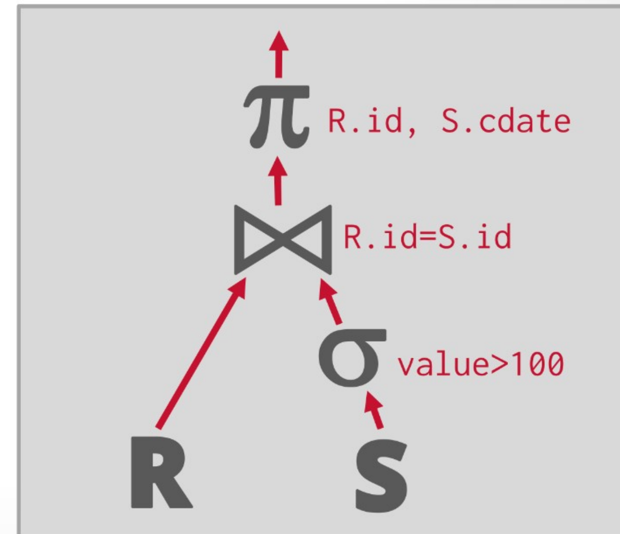
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
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
```

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



CMU-DB
15-721 (Spring 2024)

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
```

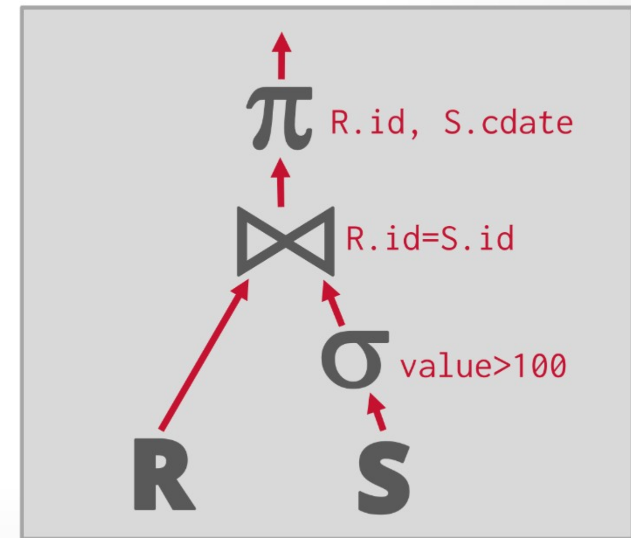
All Tuples

3

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

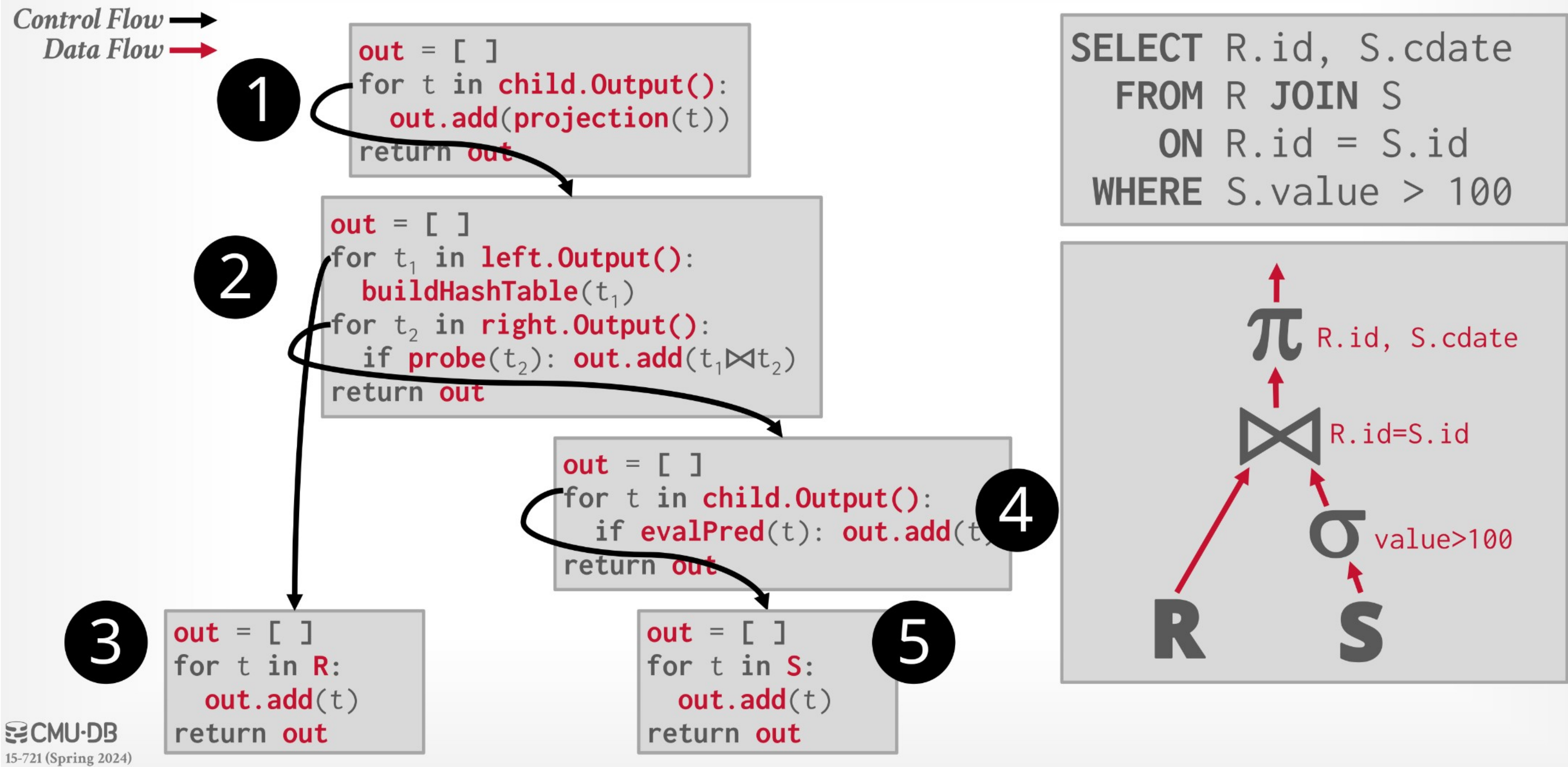
```
out = [ ]
for t in S:
  out.add(t)
return out
```

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



CMU-DB
15-721 (Spring 2024)

MATERIALIZATION MODEL



MATERIALIZATION MODEL

Control Flow →
Data Flow →

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

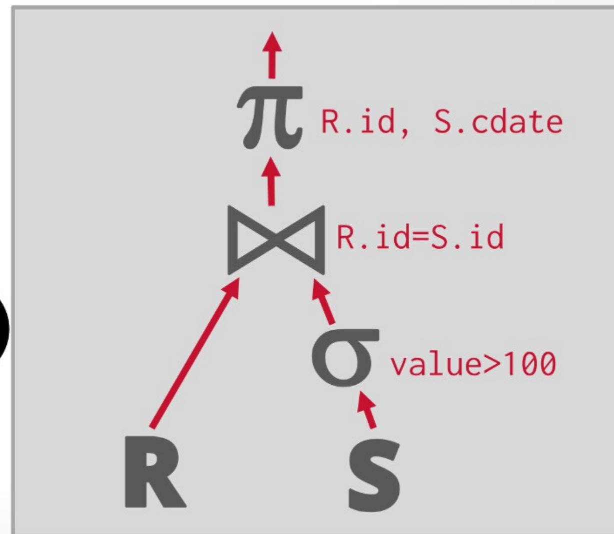
❷
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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

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

❺
```
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
```



CMU·DB
15-721 (Spring 2024)

MATERIALIZATION MODEL

Control Flow →
Data Flow →

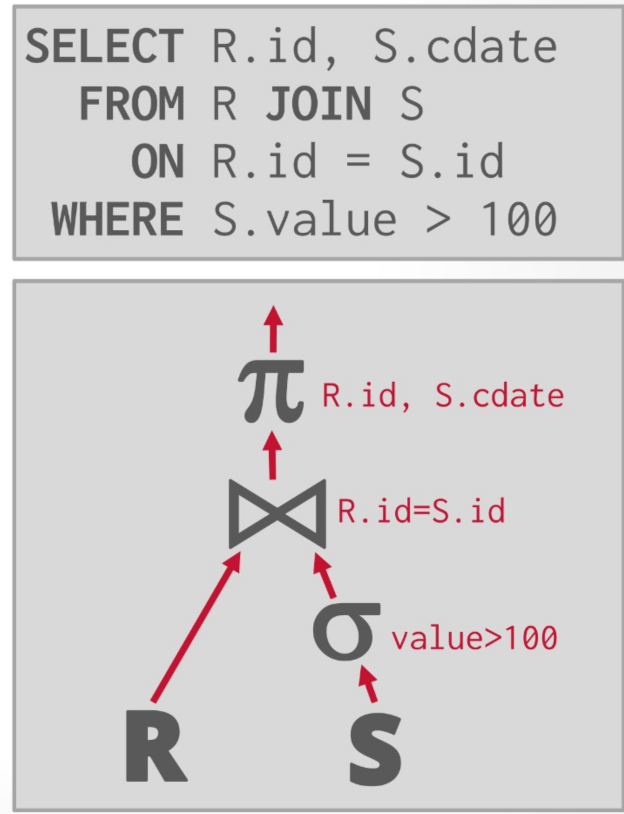
1
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

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
```

CMU-DB
15-721 (Spring 2024)



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

VECTORIZATION MODEL

Control Flow →
Data Flow →

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

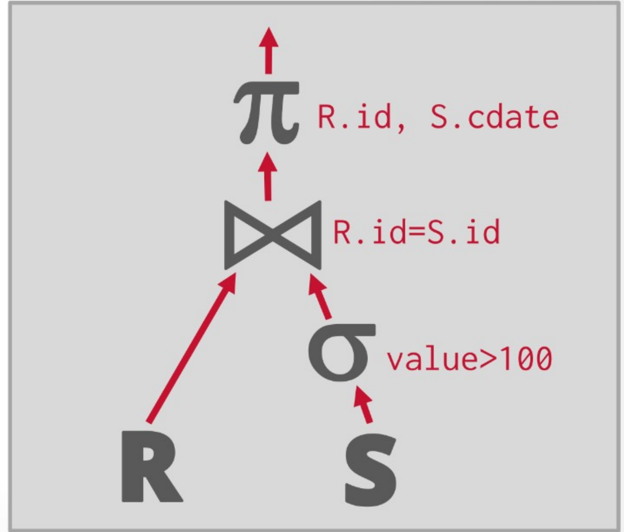
```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

```
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)
```

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



CMU-DB
15-721 (Spring 2024)

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)
```

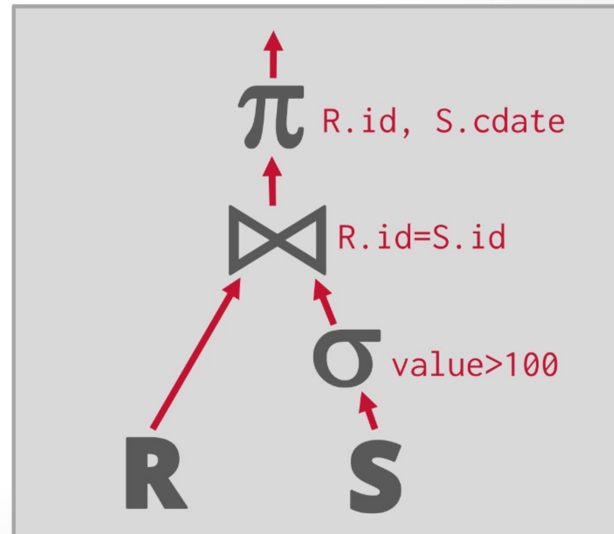
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

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)
```

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



CMU·DB
15-721 (Spring 2024)

VECTORIZATION MODEL

Control Flow →
Data Flow →

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

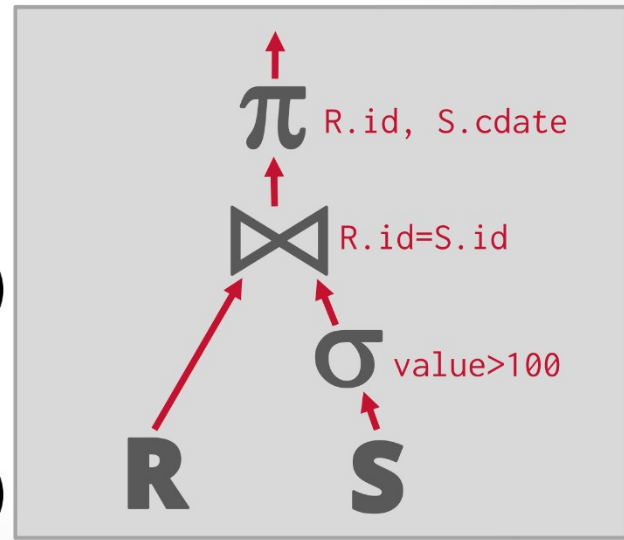
2
```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

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)
```



CMU-DB
15-721 (Spring 2024)

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)
```

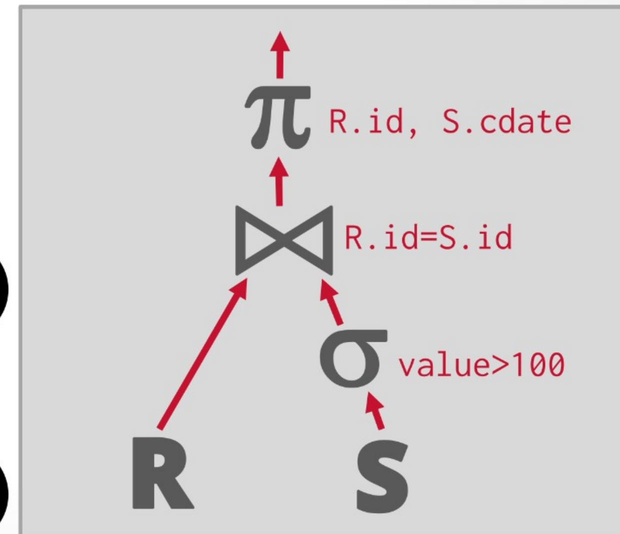
3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

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)
```

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



CMU-DB
15-721 (Spring 2024)

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.

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).

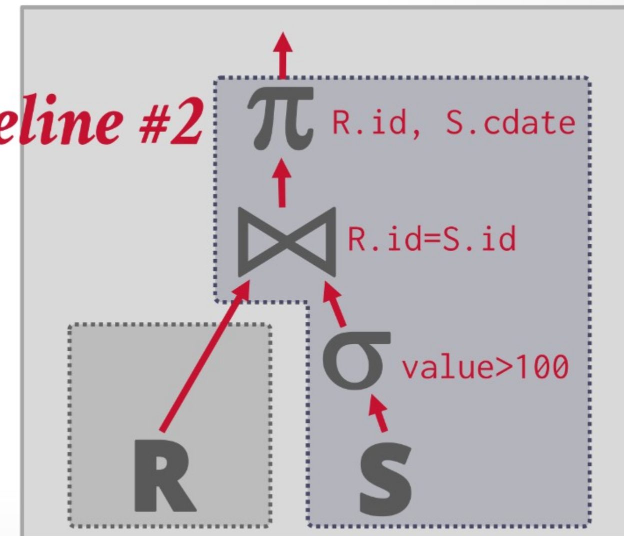
Approach #2: Bottom-to-Top (Push)

– Start with leaf nodes and "push" data to their parents.
– Can "fuse" operators together within a for-loop to minimize intermediate result staging.
– We will see this technique again later in HyPer and Peloton ROF.

PUSH-BASED ITERATOR MODEL

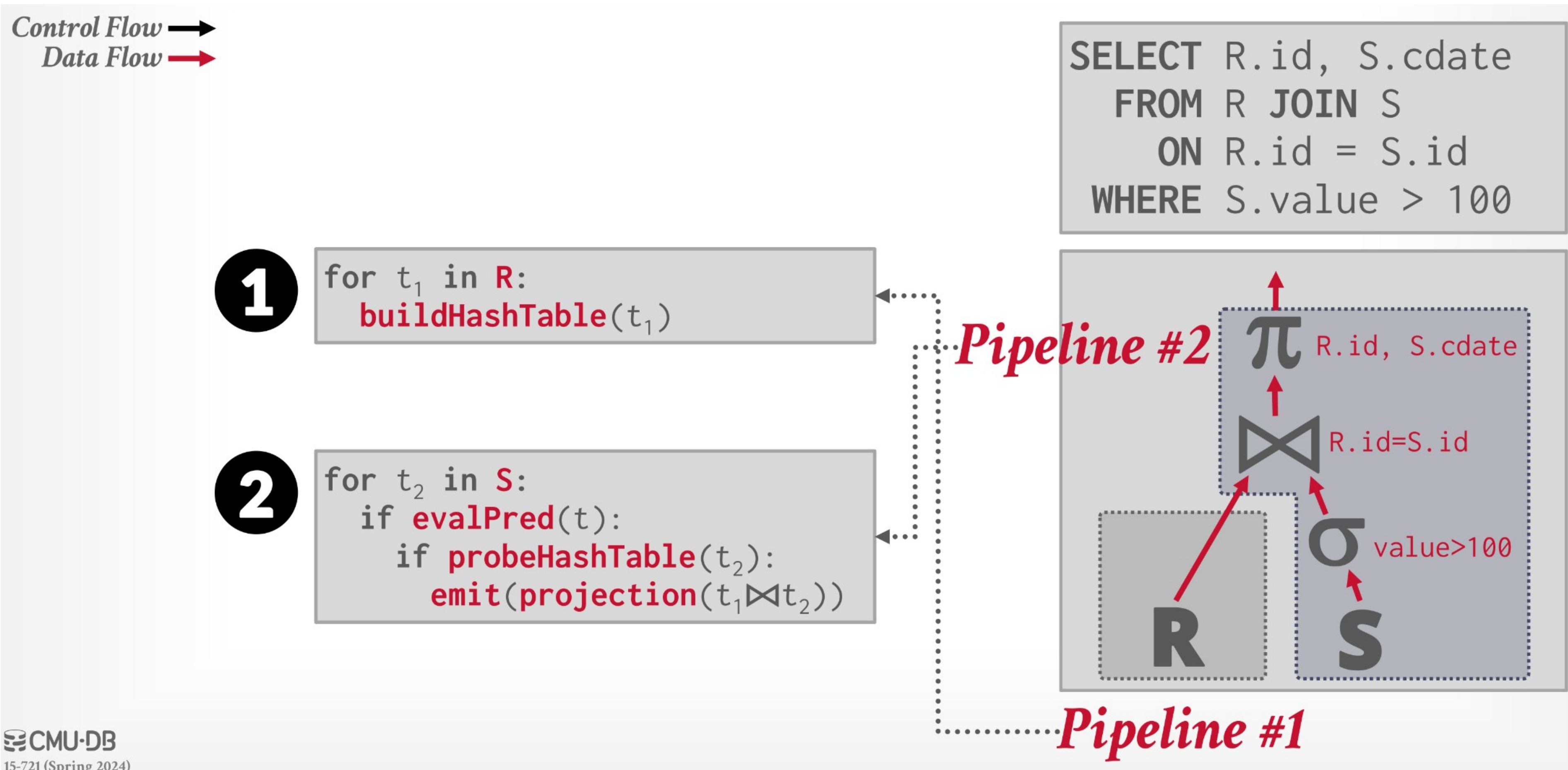
Control Flow →
Data Flow →

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

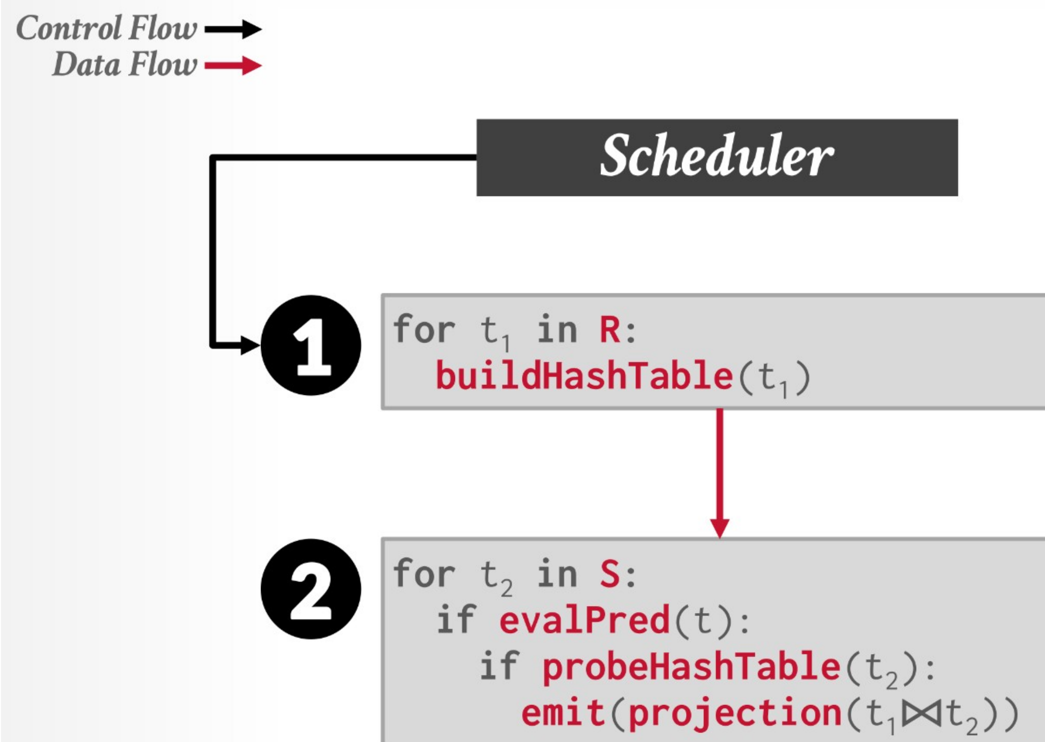


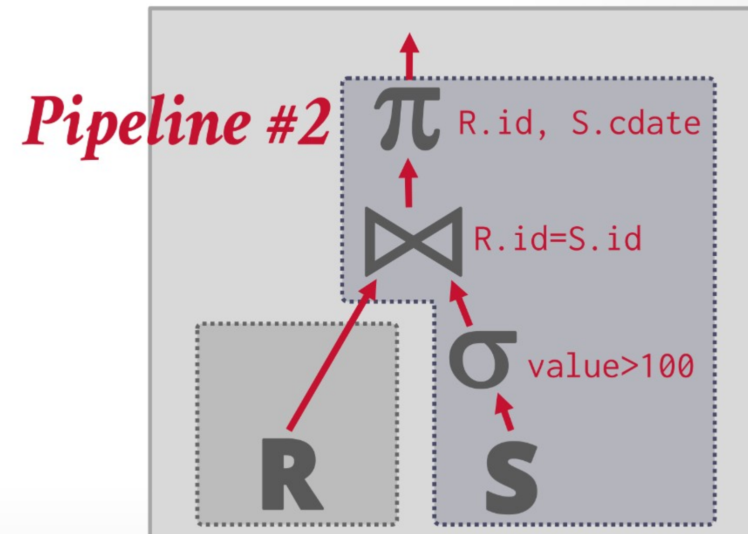
CMU-DB
15-721 (Spring 2024)

PUSH-BASED ITERATOR MODEL

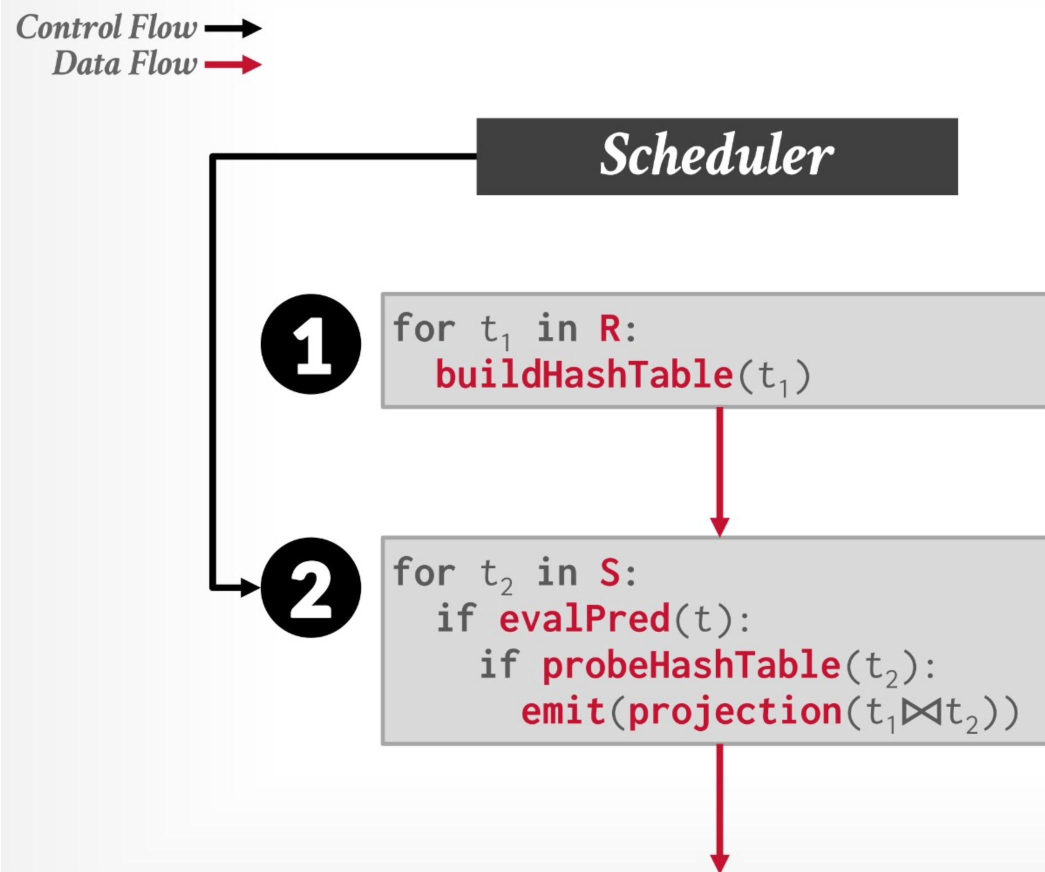


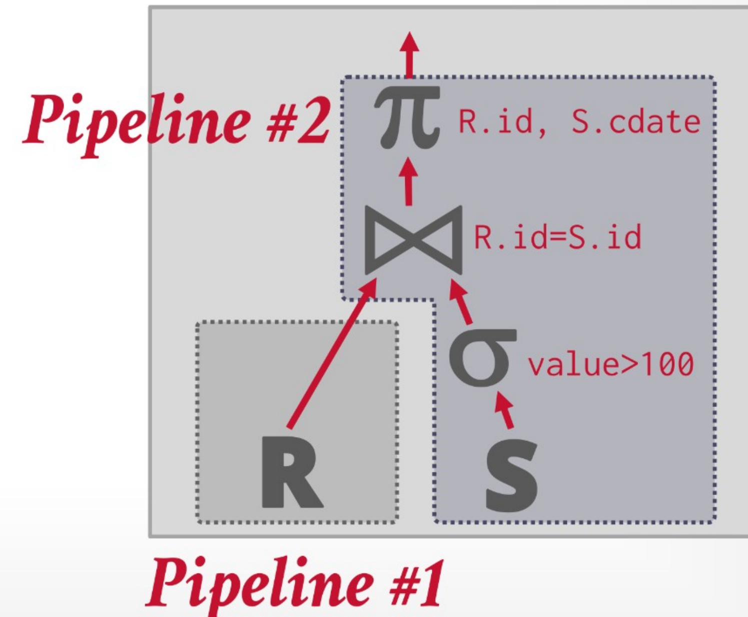
PUSH-BASED ITERATOR MODEL





PUSH-BASED ITERATOR MODEL





PLAN PROCESSING DIRECTION

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

- 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.

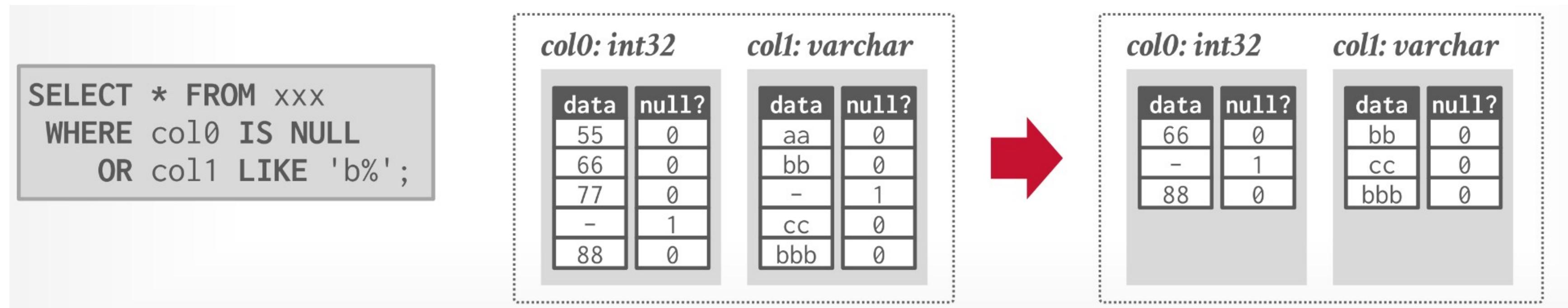
Approach #2: Bottom-to-Top (Push)

- 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).

OBSERVATION

With the Iterator model, if a tuple does not satisfy a filter, then the DBMS just invokes Next() again on the child operator to get another tuple.

In the Vectorized model, however, a vector / batch may contain some tuples that do not satisfy filters.



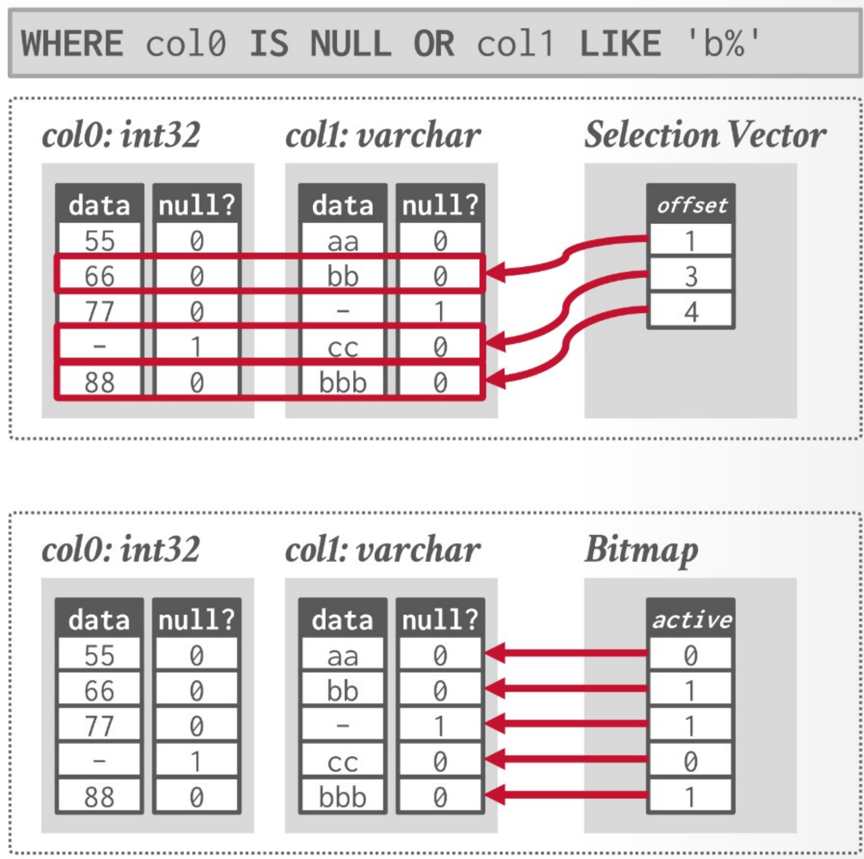
FILTER REPRESENTATION

Approach #1: Selection Vectors

- Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.
- Pre-allocate selection vector as the max-size of the input vector.

Approach #2: Bitmaps

- Positionally-aligned bitmap that indicates whether a tuple is valid at an offset.
- Some SIMD instructions natively use these bitmaps as input masks.



PARALLEL EXECUTION

The DBMS executes multiple tasks simultaneously to improve hardware utilization.

- Active tasks do not need to belong to the same query.
- High-level approaches do not vary on whether the DBMS is multi-threaded, multi-process, or multi-node.

Approach #1: Inter-Query Parallelism

Approach #2: Intra-Query Parallelism

INTER-QUERY PARALLELISM

Improve overall performance by allowing multiple queries to execute simultaneously.

- Most DBMSs use a simple first-come, first-served policy.

OLAP queries have parallelizable and non-parallelizable phases. The goal is to always keep all cores active.

We will discuss scheduling queries and multiplexing tasks on cores in future lectures.

INTRA-QUERY PARALLELISM

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

Approach #1: Intra-Operator (Horizontal)

Approach #2: Inter-Operator (Vertical)

These techniques are not mutually exclusive.

There are parallel algorithms for every relational operator.

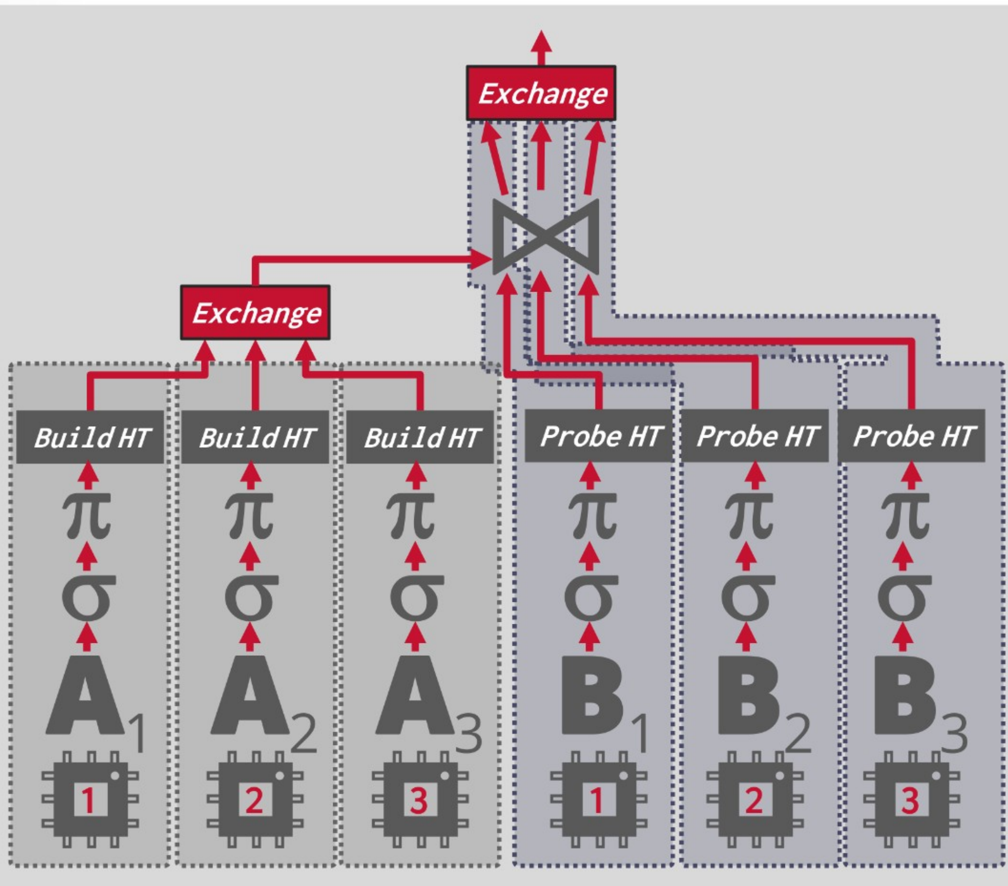
INTRA-OPERATOR PARALLELISM

Approach #1: Intra-Operator (Horizontal)

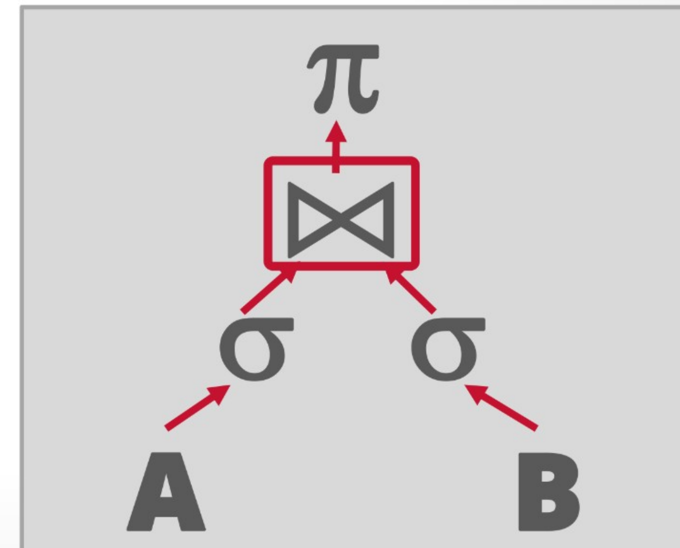
- Operators are decomposed into independent instances that perform the same function on different subsets of data.

The DBMS inserts an **exchange** operator into the query plan to coalesce results from children operators.

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



EXCHANGE OPERATOR

Exchange Type #1 – Gather

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

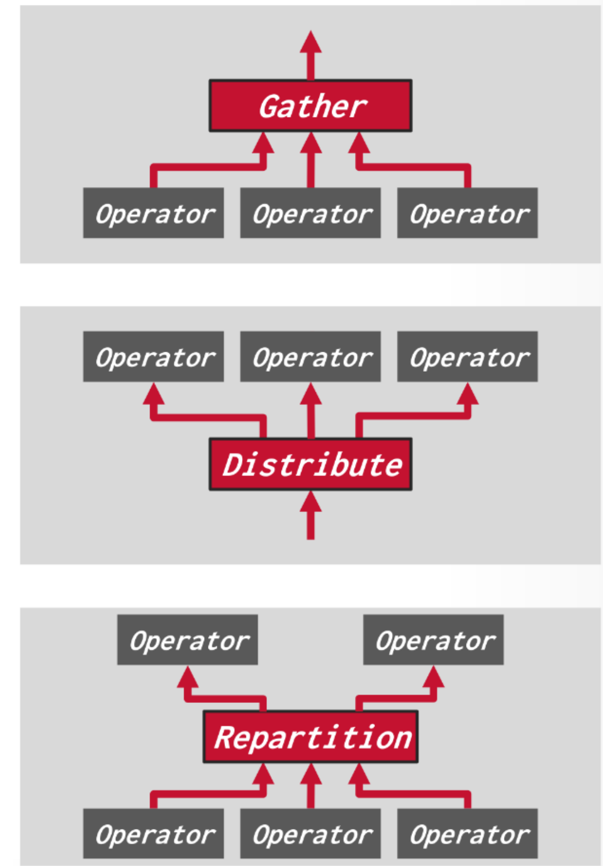
Exchange Type #2 – Distribute

– Split a single input stream into multiple output streams.

Exchange Type #3 – Repartition

– Shuffle multiple input streams across multiple output streams.

– Some DBMSs always perform this step after every pipeline (e.g., Dremel/BigQuery).



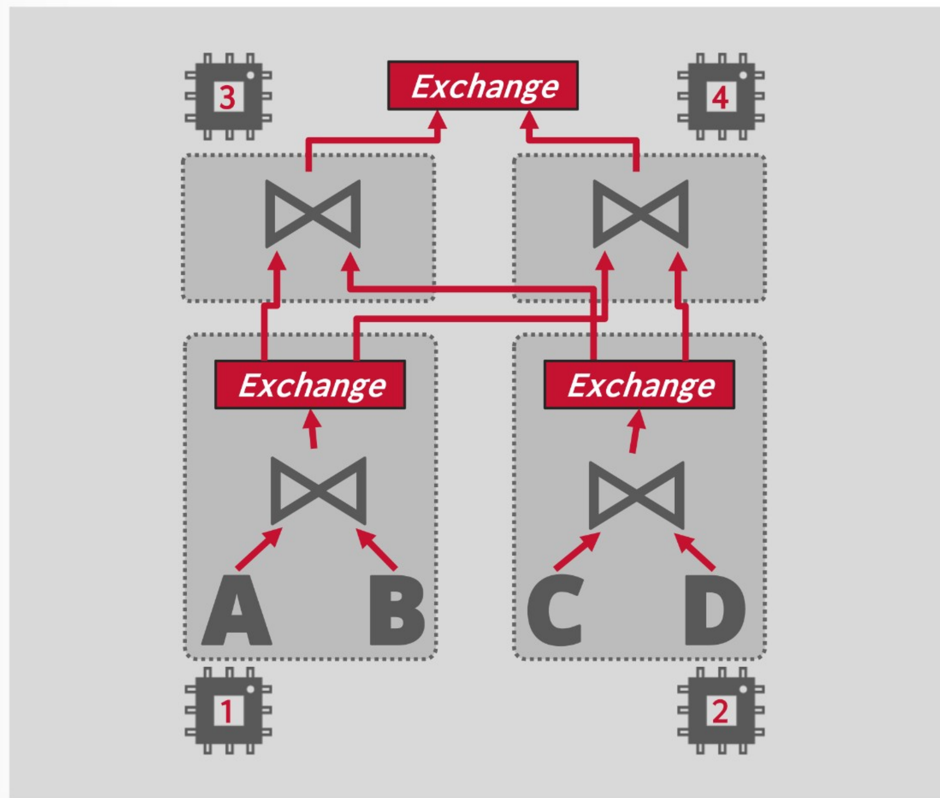
INTER-OPERATOR PARALLELISM

Approach #2: Inter-Operator (Vertical)

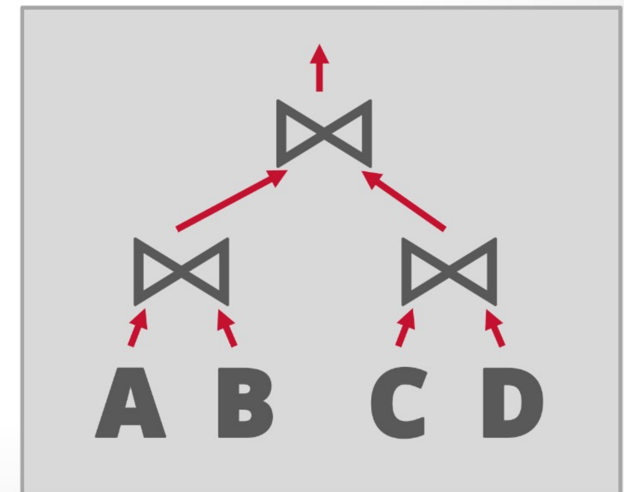
- Operations are overlapped to pipeline data from one stage to the next without materialization.
- Workers execute multiple operators from different segments of a query plan at the same time.Still need exchange operators to combine intermediate Results from segments.

Also called **pipelined parallelism**.

INTER-OPERATOR PARALLELISM




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



PARTING THOUGHTS

The easiest way to implement something is not going to always produce the most efficient execution strategy for modern CPUs.

Vectorized / bottom-up execution almost always will be the better way to execute OLAP queries.

NEXT CLASS

- Vectorization Fundamentals
- Vectorized DBMS Algorithms