ADVANCED DATABASE SYSTEMS

Query Execution & Processing I

04

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

LAST CLASS

Last two lectures were about minimize the amount of data that the DBMS processes when executing sequential scans.

We are now going to start discussing ways to improve the DBMS's query execution performance.

SEQUENTIAL SCAN OPTIMIZATIONS

Data Encoding / Compression

Prefetching / Scan Sharing

Task Parallelization / Multi-threading

Clustering / Sorting

Late Materialization

Materialized Views / Result Caching

Data Skipping

Data Parallelization / Vectorization

Code Specialization / Compilation

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.

Andy's Unscientific Top-3 Optimizations:

→ 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.

OPTIMIZATION GOALS

Approach #1
→ Use fewer in

Approach #2
→ Execute mor

Approach #3
→ Use multipl

From: Linus Torvalds <torvalds@linux-foundation.org>
To: Arnd Bergmann <arnd@kernel.org>, "Jason A. Donenfeld" <Jason@zx2c4.com>
CC: Linux Kernel Mailing List <linux-kernel@vger.kernel.org>
Subject: Re: [PATCH v2 07/13] asm-generic: unaligned always use struct helpers
Date: Tue, 18 May 2021 06:12:03 -1000 [thread overview]
Message-ID: <CAHk-=wjuoGyxDhAF8SsrTkN0-YfCx7E6jUN3ikC_tn2AKWTTsA@mail.gmail.com> (raw)
In-Reply-To: <CAK8P3a3hbts4k+rrfnE8Z78ezCaME0UVgwqkdLW5NOps2YHUQQ@mail.gmail.com>

On Tue, May 18, 2021 at 5:42 AM Arnd Bergmann <arnd@kernel.org> wrote:
>
> To be on the safe side, we could pass -fno-tree-loop-vectorize along
> with -O3 on the affected gcc versions, or use a bigger hammer
> (not use -O3 at all, always set -fno-tree-loop-vectorize, ...).

I personally think -O3 in general is unsafe.

It has historically been horribly buggy. It's gotten better, but this case clearly shows that "gotten better" really isn't that high of a bar.

QUERY EXECUTION

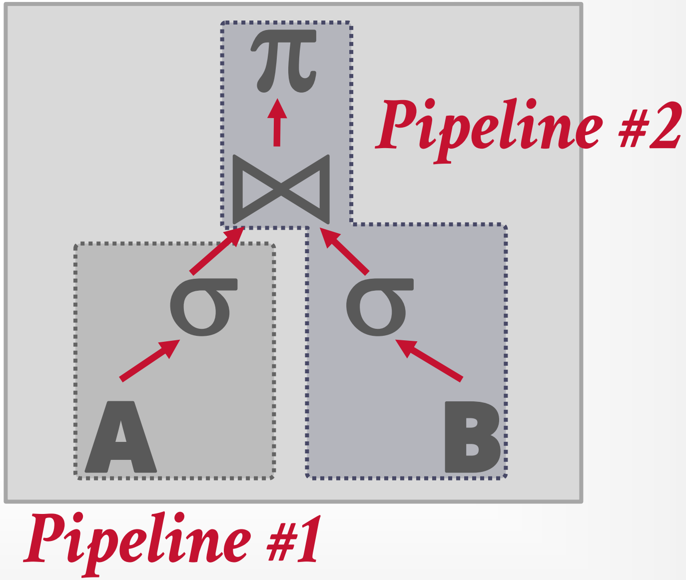
A query plan is a DAG of **operators**.

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.

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



TODAY'S AGENDA

MonetDB/X100 Analysis
Processing Models
Plan Processing Direction
Filter Representation

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.

MONETDB/X100: HYPER-PIPELINING QUERY EXECUTION
CIDR 2005

MONETDB/X100 (2005)

Low-level an
memory DBI
→ Show how I
CPU archite

Based on the
DBMS callec
→ Renamed to
→ Rebranded

MONETDB/X100: HYPER-PIPELINING QUERY EXECUTION
CIDR 2005

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

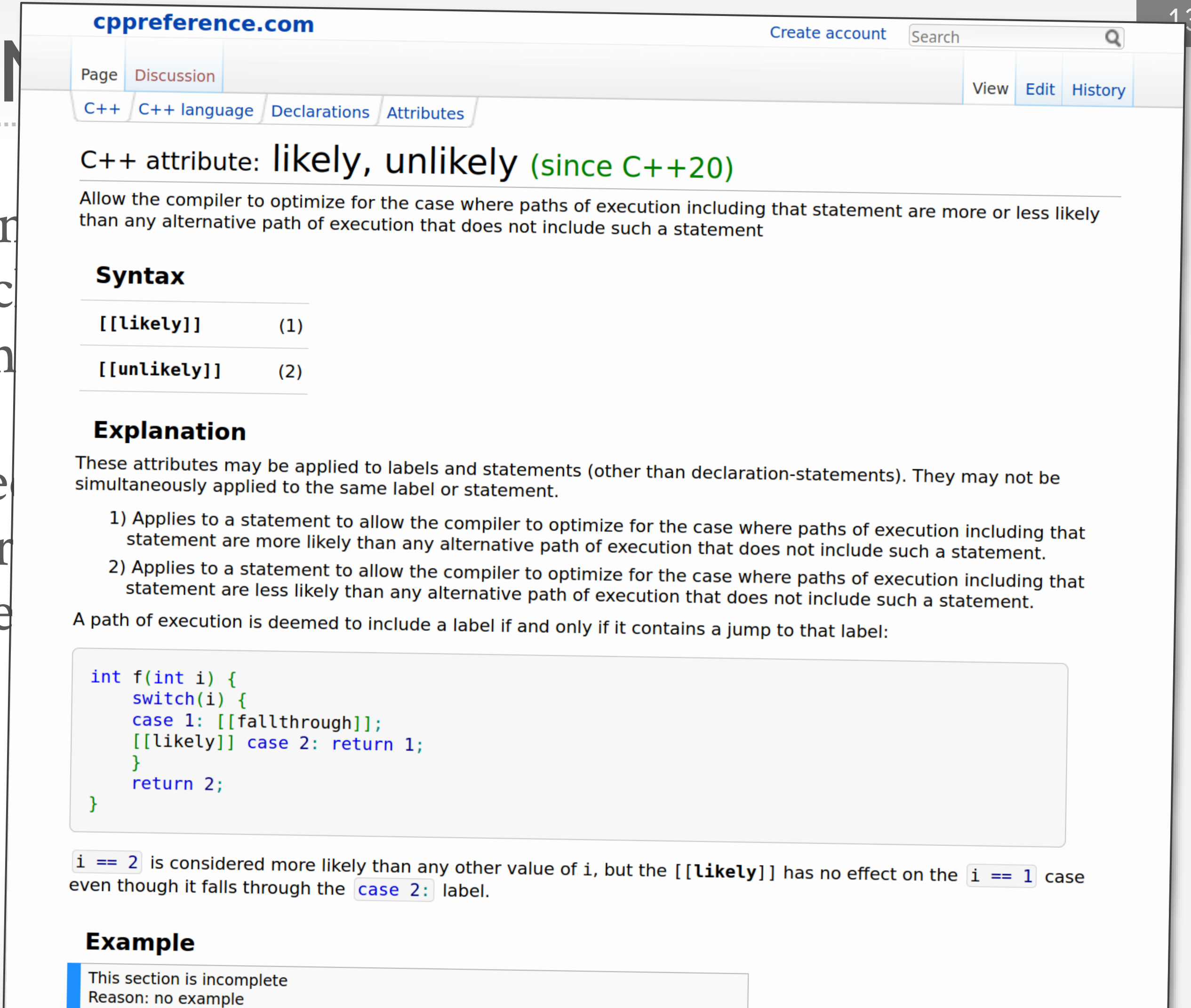
Because of long pipelines, CPUs will speculatively execute branches. This potentially hides the long stalls between dependent instructions.

The most executed branching code in a DBMS is the filter operation during a sequential scan.
But this is (nearly) impossible to predict correctly.

BRAN

Because of lon
execute branc
stalls between

The most exe
the filter oper
But this is (ne



cppreference.com

Create account | Search

Page | Discussion | View | Edit | History

C++ / C++ language / Declarations / Attributes

C++ attribute: likely, unlikely (since C++20)

Allow the compiler to optimize for the case where paths of execution including that statement are more or less likely than any alternative path of execution that does not include such a statement

Syntax

`[[likely]]`	(1)
`[[unlikely]]`	(2)

Explanation

These attributes may be applied to labels and statements (other than declaration-statements). They may not be simultaneously applied to the same label or statement.

1) Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are more likely than any alternative path of execution that does not include such a statement.

2) Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are less likely than any alternative path of execution that does not include such a statement.

A path of execution is deemed to include a label if and only if it contains a jump to that label:

```
int f(int i) {
    switch(i) {
    case 1: [[fallthrough]];
    [[likely]] case 2: return 1;
    }
    return 2;
}
```

`i == 2` is considered more likely than any other value of `i`, but the `[[likely]]` has no effect on the `i == 1` case even though it falls through the `case 2:` label.

Example

This section is incomplete
Reason: no example

cppreference.com

Create account

Search

View Edit History

ly (since C++20)

ere paths of execution including that statement are more or less likely
es not include such a statement

atements (other than declaration-statements). They may not be
ement.

er to optimize for the case where paths of execution including that
ive path of execution that does not include such a statement.

r to optimize for the case where paths of execution including that
e path of execution that does not include such a statement.

and only if it contains a jump to that label:

lue of i, but the [[likely]] has no effect on the `i == 1` case

Don't use the [[likely]] or [[unlikely]] attributes

Posted on 2020-08-27 by Aaron Ballman

C++20 introduced the likelihood attributes `[[likely]]` and `[[unlikely]]` as a way for a programmer to give an optimization hint to their implementation that a given code path is more or less likely to be taken. On its face, this seems like a great set of attributes because you can give hints to the optimizer in a way that is hopefully understood by all implementations and will result in faster performance. What's not to love?

The attribute is specified to appertain to arbitrary statements or labels with the recommended practice "to optimize for the case where paths of execution including it are arbitrarily more likely|unlikely than any alternative path of execution that does not include such an attribute on a statement or label." Pop quiz, what does this code do?

```
if (something) {
  [[likely]];
  [[unlikely]];
  foo(something);
}
```

Sorry, but the answer key for this quiz is currently unavailable. However, one rule you should follow about how to use these attributes is: never allow both attributes to appear in the same path of execution. Lest you think, "but who would write such bad code?", consider this reasonable-looking-but-probably-very-unfortunate code:

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)
```

Source: Bogdan Raducanu

SELECTION SCANS

Scalar (Branching)

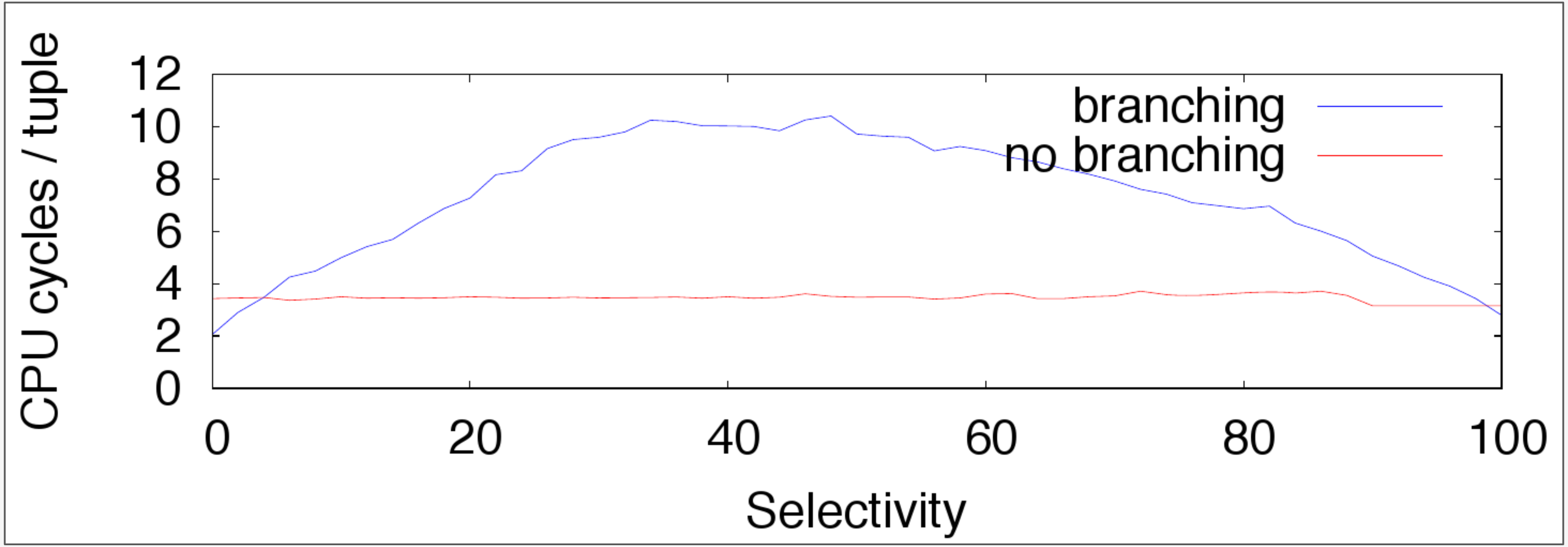
```
i = 0
for t in table:
  key = t.key
  if (key>low) && (key<high):
    copy(t, output[i])
    i = i + 1
```

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
```

Source: Bogdan Raducanu

SELECTION SCANS



Source: Bogdan Raducanu

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.

EXCESSIV

The DBMS needs to
it must check a value
operation on that va
→ This is usually imple
→ Also creates more br
CPU to predict reliab

Example: Postgres' a

```
/* ----------
 * add_var() -
 *
 *  Full version of add functionality on variable level (handling signs).
 *  result might point to one of the operands too without danger.
 * ----------
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
```

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

Control Flow →
Data Flow →

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

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()

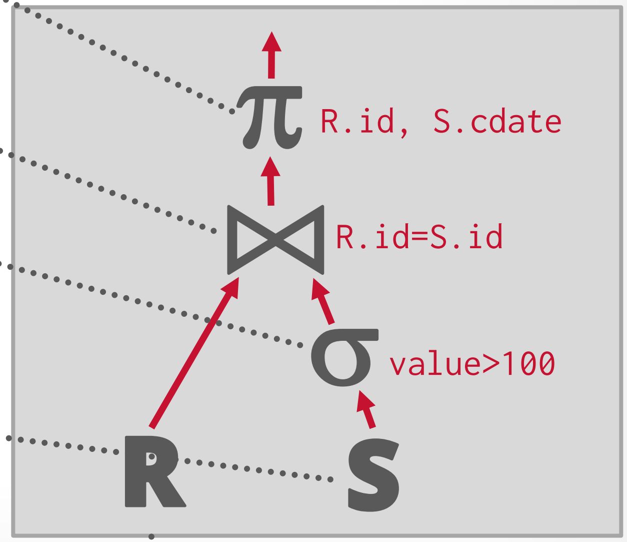
```
for t in child.Next():
  if evalPred(t): emit(t)
```

Next()

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

Next()

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



ITERATOR MODEL

Control Flow →
Data Flow →

1

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

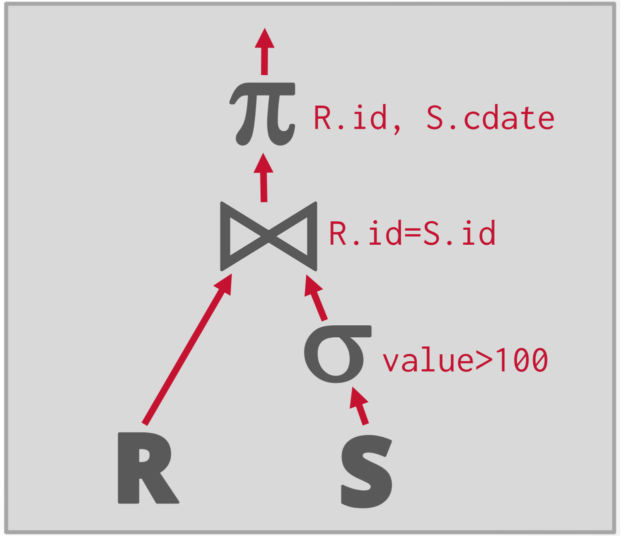
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
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

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

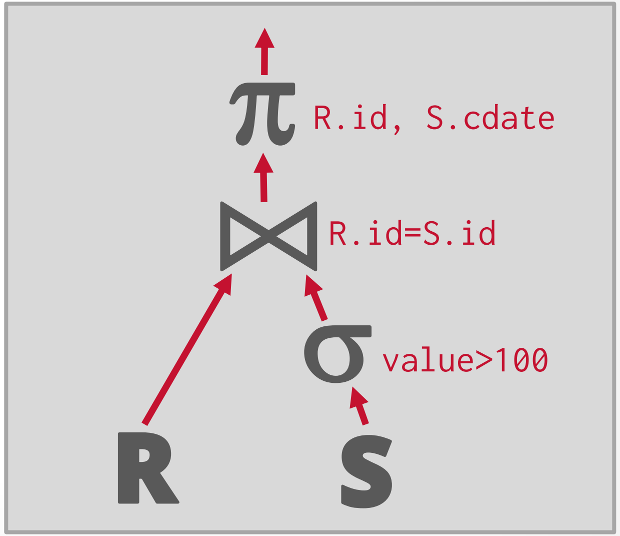
```
for t in child.Next():
  if evalPred(t): emit(t)
```

3

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

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

```
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))
```

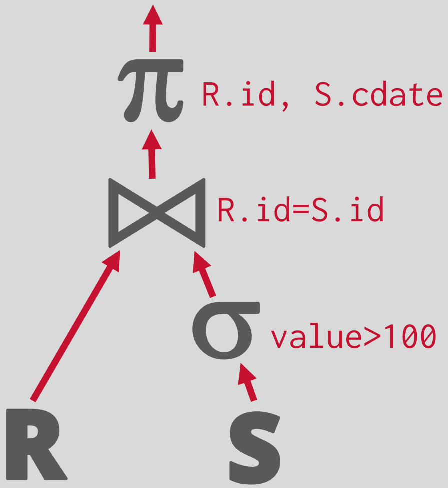
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

Control Flow →
Data Flow →

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

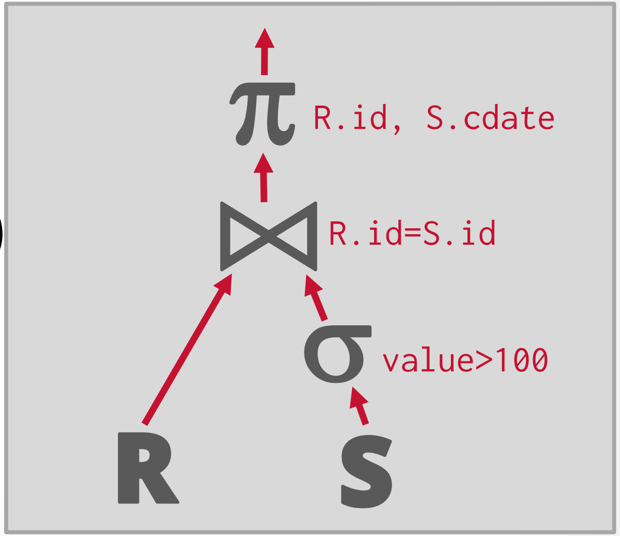
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

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

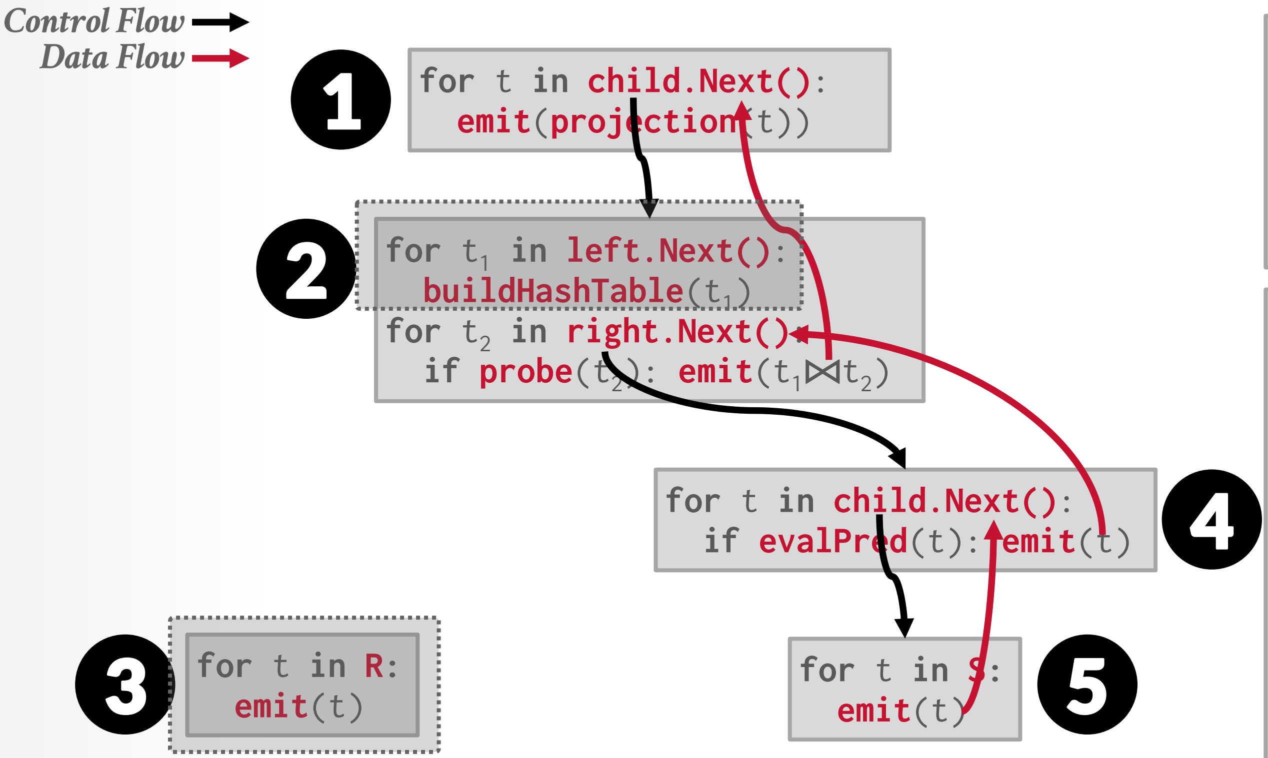
4
```
for t in child.Next():
  if evalPred(t): emit(t)
```

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

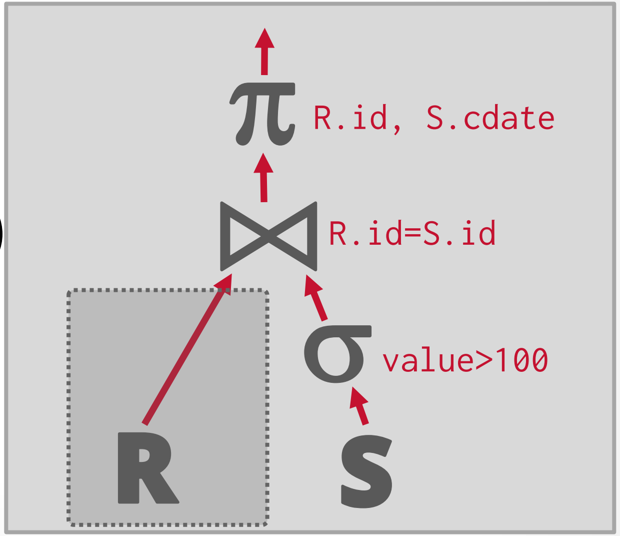
```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



ITERATOR MODEL

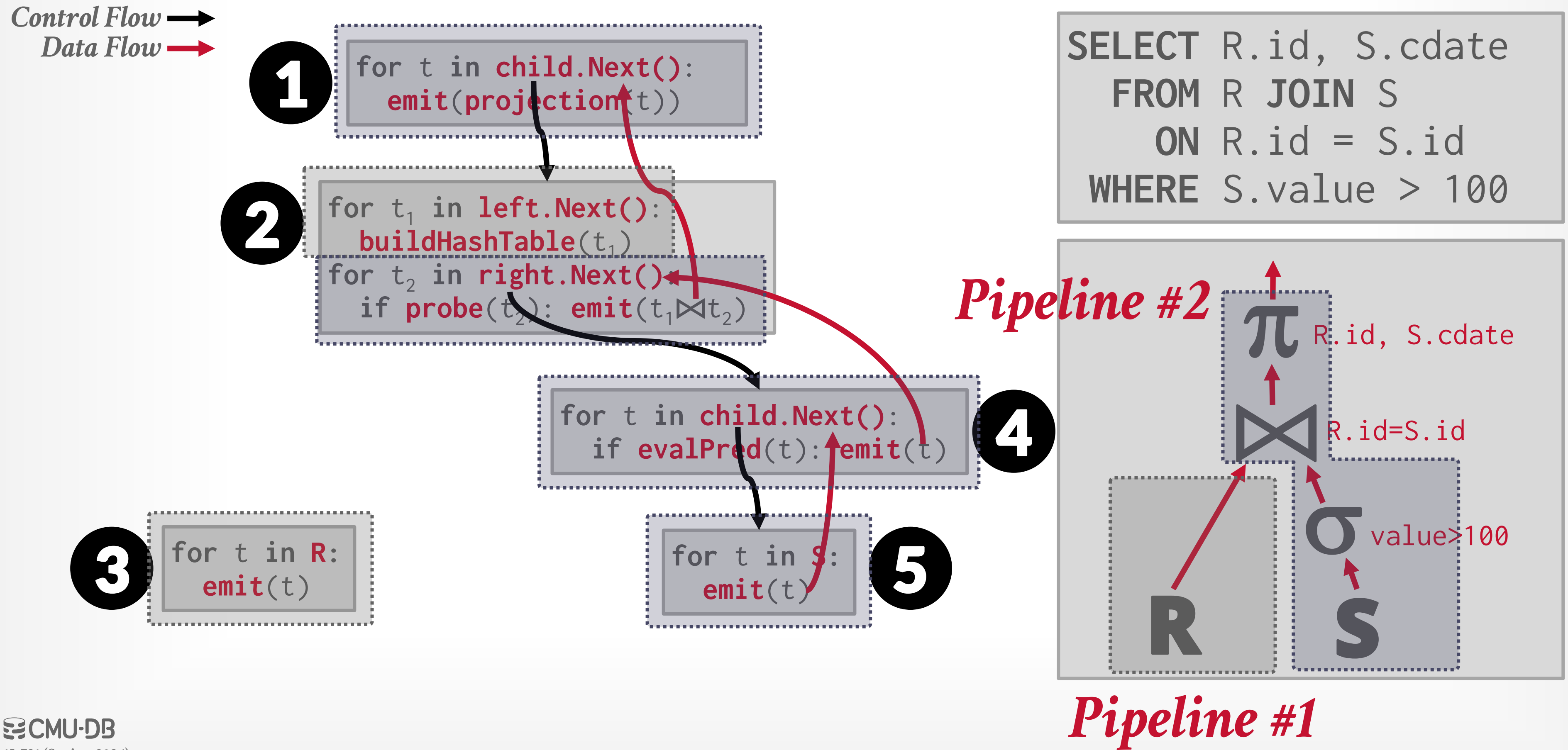



```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Pipeline #1

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

Control Flow →
Data Flow →

❶

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

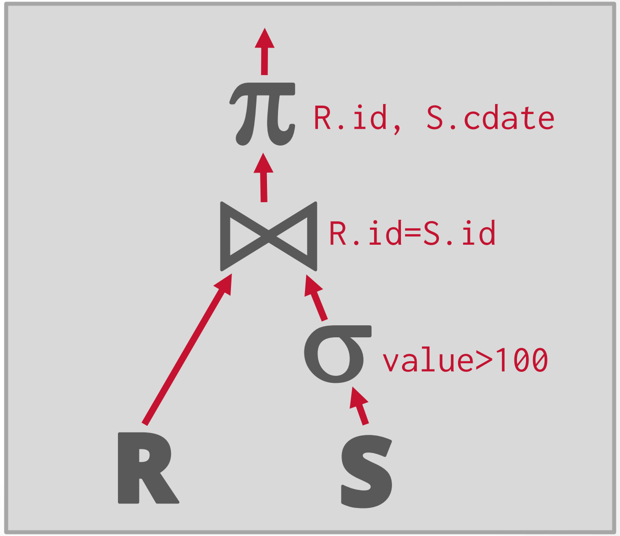
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
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
```

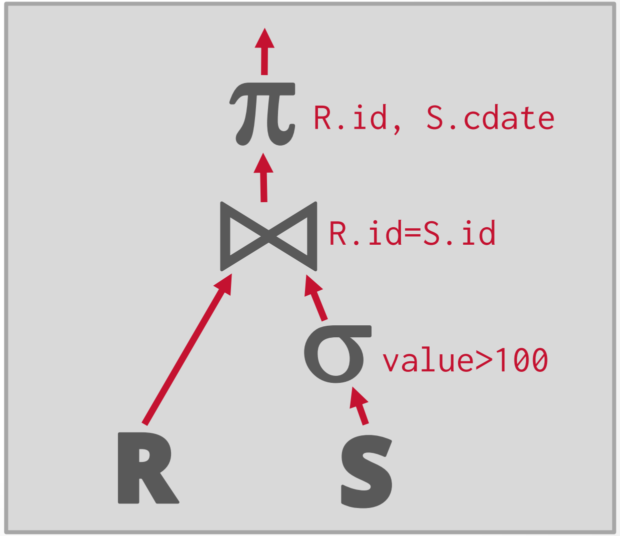
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
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
```

```
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
```

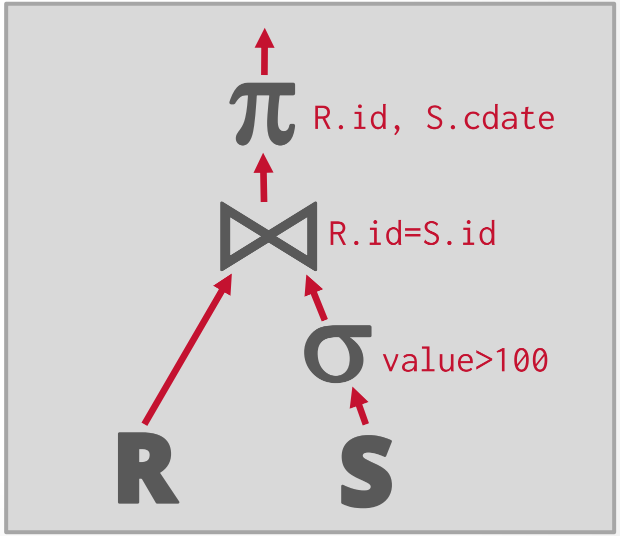
All Tuples

3

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

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

```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
```

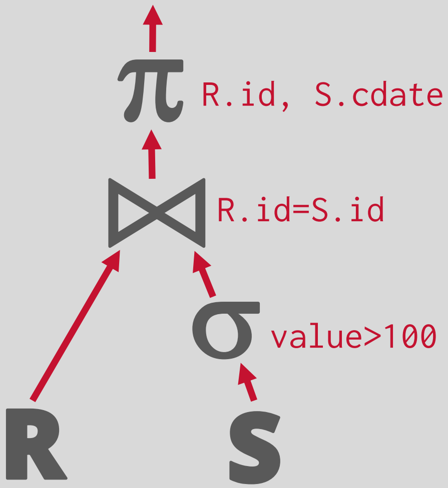
4

```
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
```

```
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
```

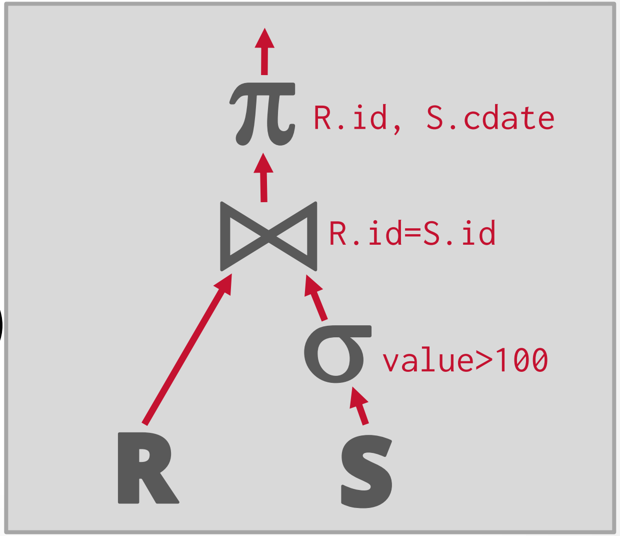
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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
```

```
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

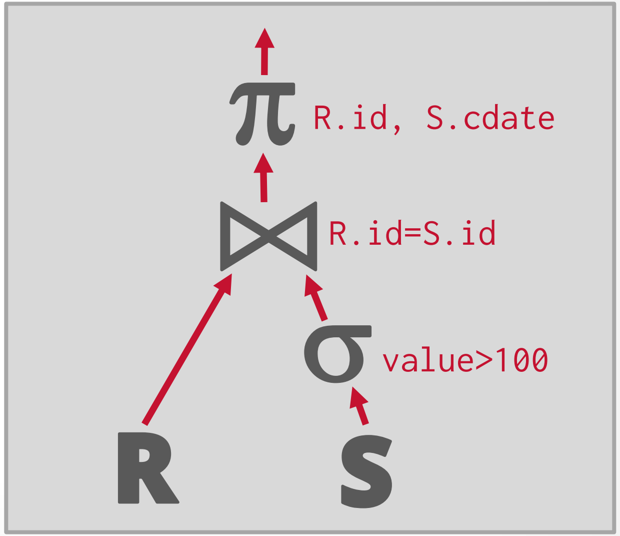
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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

3

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

```
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.

→ 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)
```

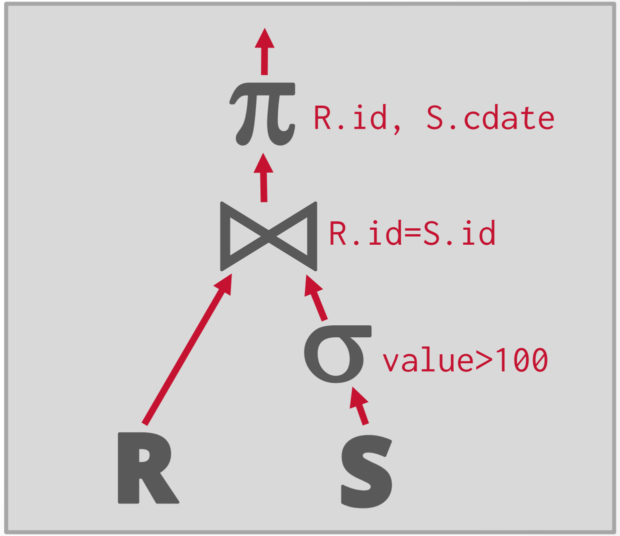
2
```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

```
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)
```

```sql
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)
```

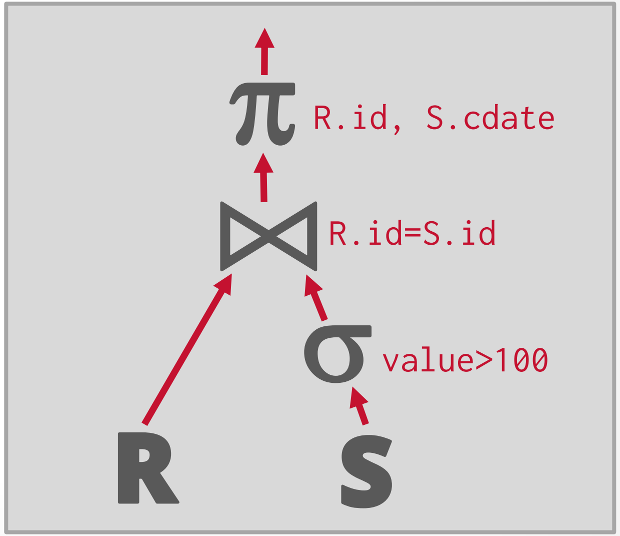
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

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 →

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)
```

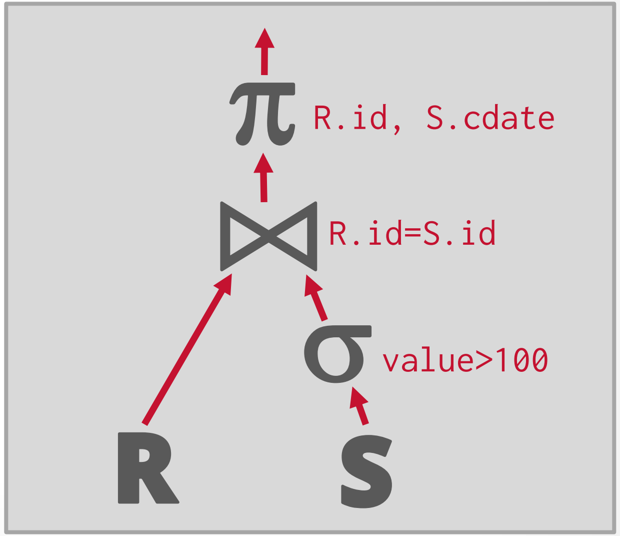
3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

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

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)
```

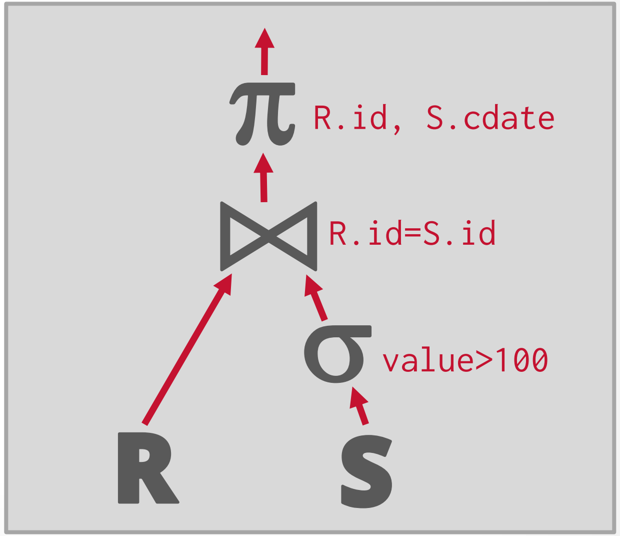
3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

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)
```

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



VECTORIZATION MODEL

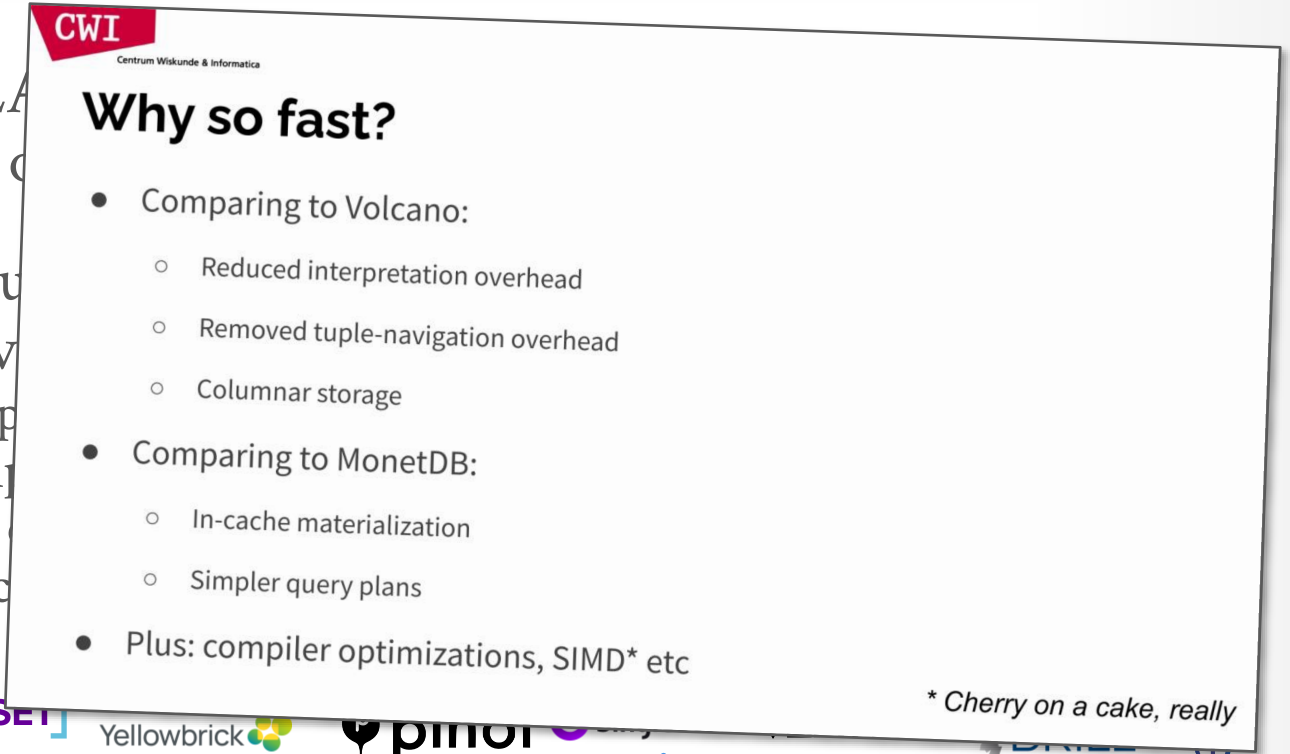
Ideal for OLAP queries because it greatly reduces the number of invocations per operator.

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 OLA
the number o

Allows an ou
operators ov
→ Operators p
which comp
→ No data or
→ Hot instruc


CWI
Centrum Wiskunde & Informatica

Why so fast?

- Comparing to Volcano:
  - Reduced interpretation overhead
  - Removed tuple-navigation overhead
  - Columnar storage
- Comparing to MonetDB:
  - In-cache materialization
  - Simpler query plans
- Plus: compiler optimizations, SIMD* etc

* *Cherry on a cake, really*

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).

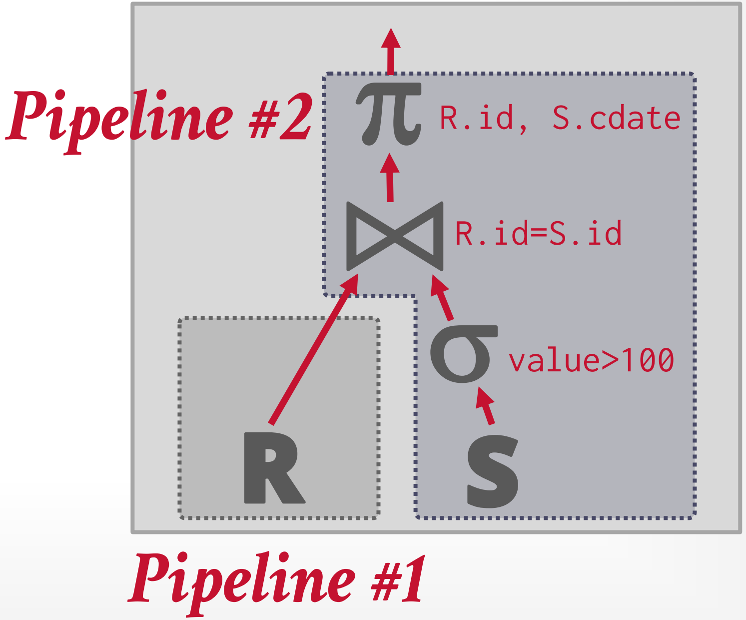
Approach #2: Bottom-to-Top (Push)

→ 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
```



PUSH-BASED ITERATOR MODEL

Control Flow →
Data Flow →

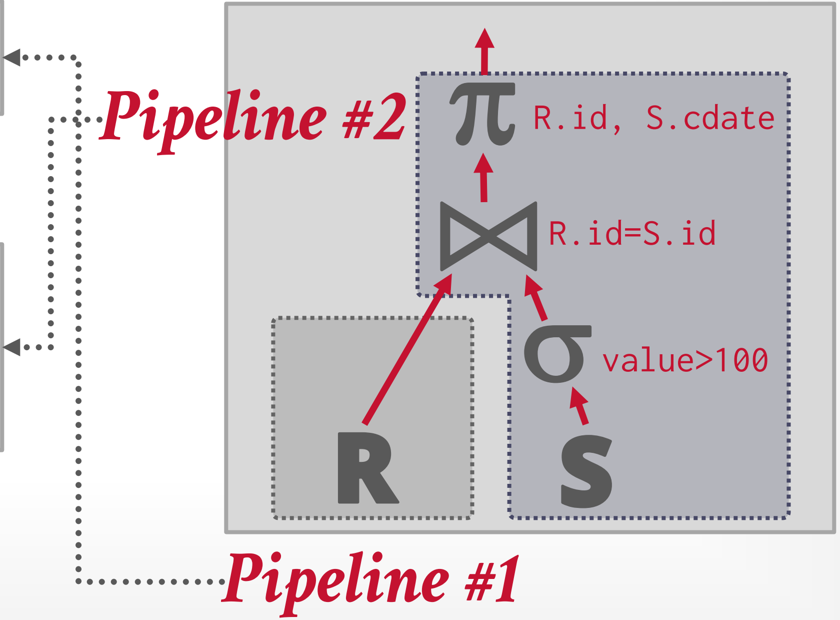
```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

1

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

2

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



PUSH-BASED ITERATOR MODEL

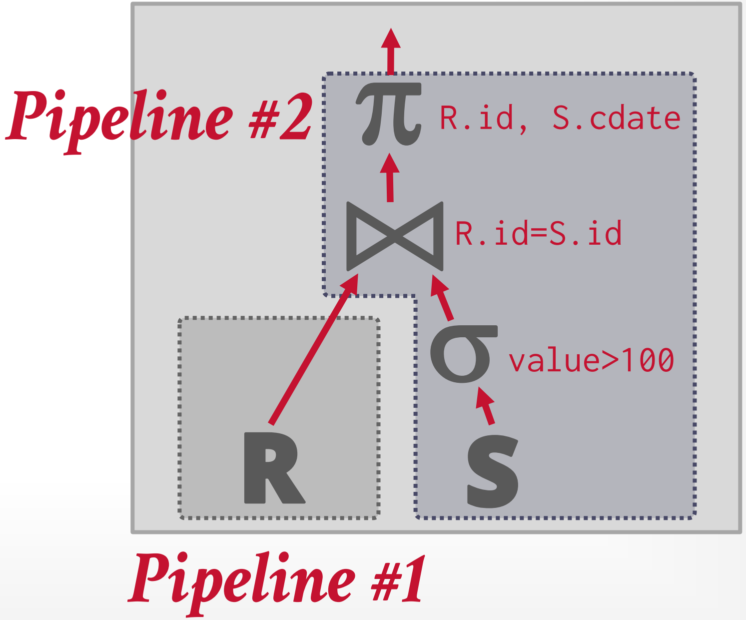
Control Flow →
Data Flow →

Scheduler

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

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

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



PUSH-BASED ITERATOR MODEL

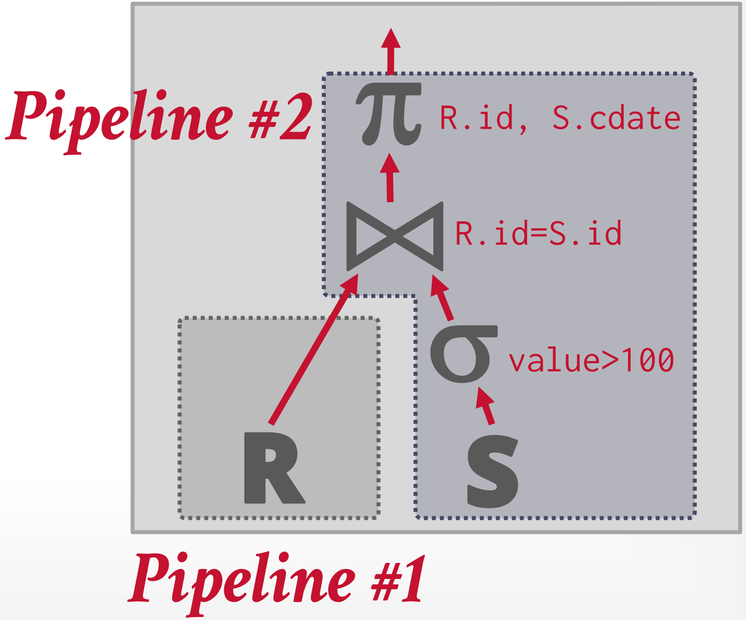
Control Flow →
Data Flow →

Scheduler

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

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

```
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)

→ 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).

PUSH VS. PULL-BASED LOOP FUSION IN QUERY ENGINES
ARXIV 2016

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.

```
SELECT * FROM xxx
 WHERE col0 IS NULL
    OR col1 LIKE 'b%';
```

col0: int32

data	null?
55	0
66	0
77	0
–	1
88	0

col1: varchar

data	null?
aa	0
bb	0
–	1
cc	0
bbb	0

→

col0: int32

data	null?
66	0
–	1
88	0

col1: varchar

data	null?
bb	0
cc	0
bbb	0

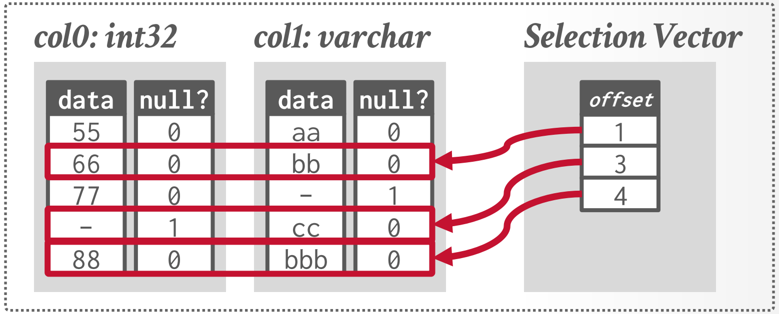
FILTER REPRESENTATION

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```

Approach #1: Selection Vectors

→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.

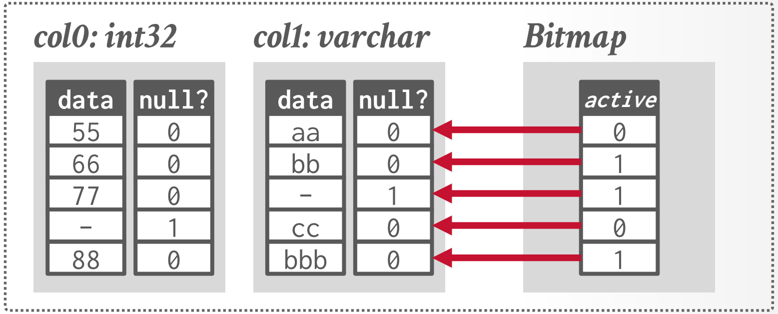
→ Pre-allocate selection vector as the max-size of the input vector.



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.



FILTER REPRESENTATION IN VECTORIZED QUERY EXECUTION
DAMON 2021

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

Design of an Execution Engine

More Parallel Execution