ADVANCED
DATABASE
SYSTEMS

Query Execution & Processing II

05

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

LAST CLASS

Query Processing Models
Plan Processing Direction
Filter Representation

Vectorized + **Push-based** query processing model is the superior approach for OLAP workloads.

A **Push-based** model with centralized scheduling enables fine-grained control of execution.
→ Pausing due to backpressure + blocking I/O

TODAY'S AGENDA

Parallel Execution

Operator Output

Intermediate Data Representation

Expression Evaluation

Adaptive Execution

PARALLEL EXECUTION

The DBMS executes multiple tasks simultaneously to improve hardware utilization.

→ Active tasks do <u>not</u> need to belong to the same query.
→ High-level approaches do <u>not</u> 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

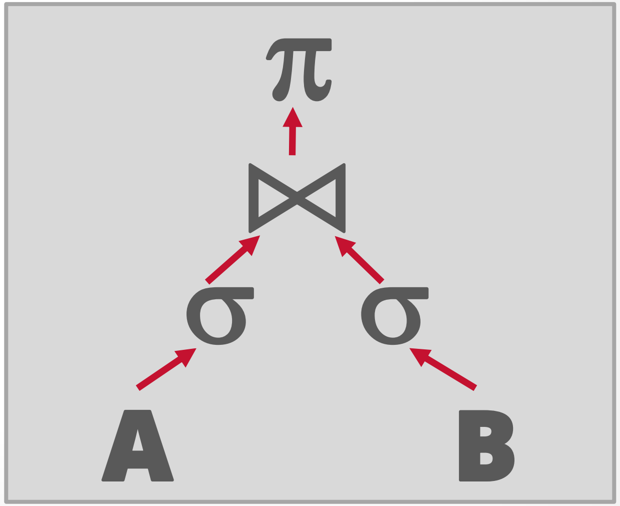
Approach #1: Intra-Operator (Horizontal)

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

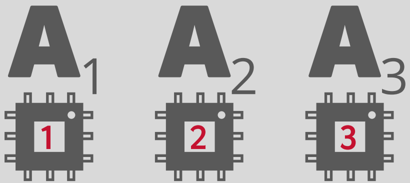
The DBMS inserts an **exchange** operator into the query plan to coalesce results from children operators.

INTRA-OPERATOR PARALLELISM

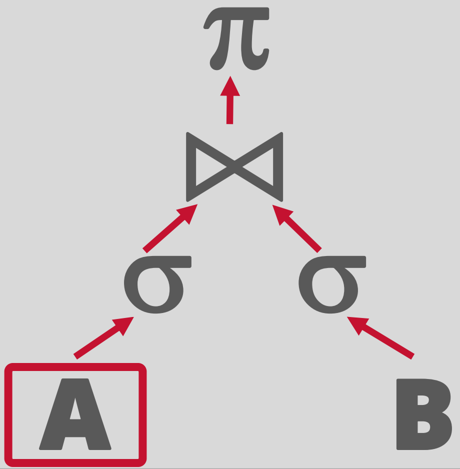
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



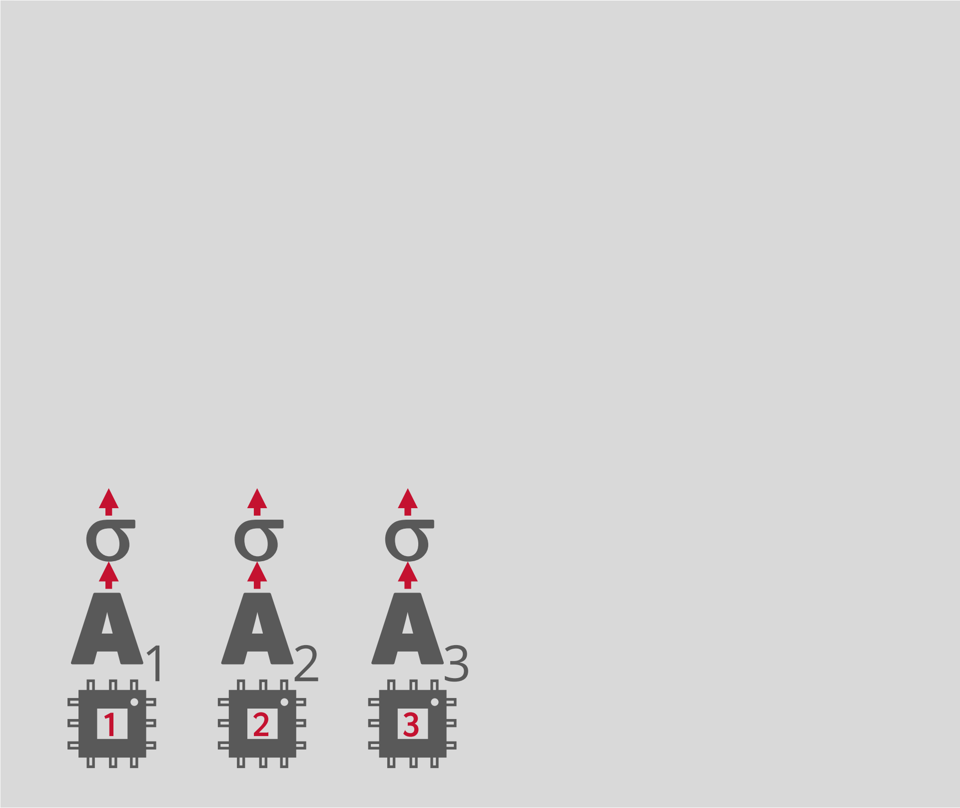
INTRA-OPERATOR PARALLELISM



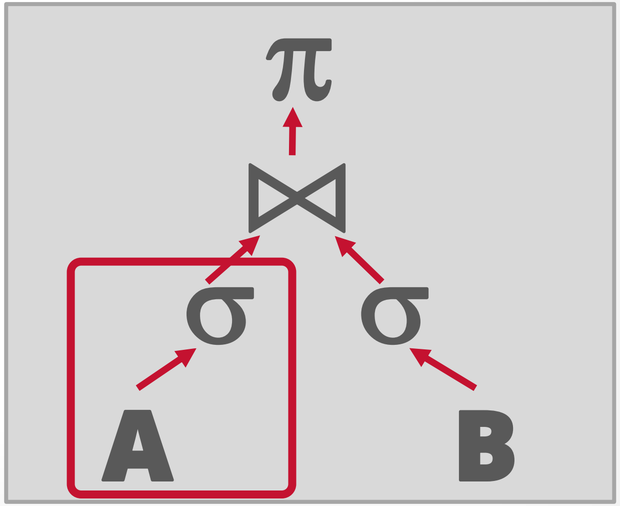

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



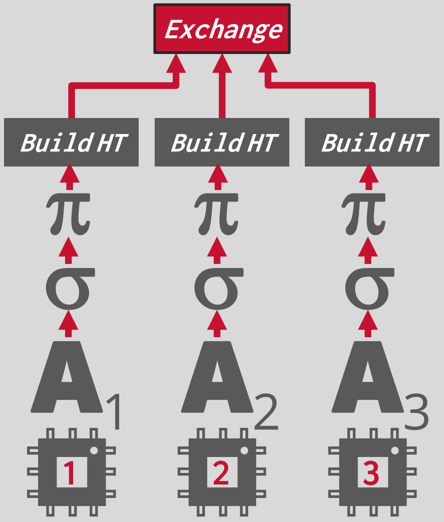
INTRA-OPERATOR PARALLELISM



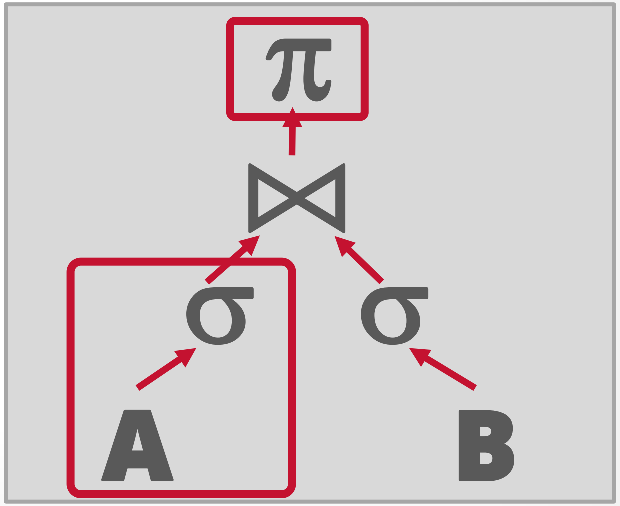

```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



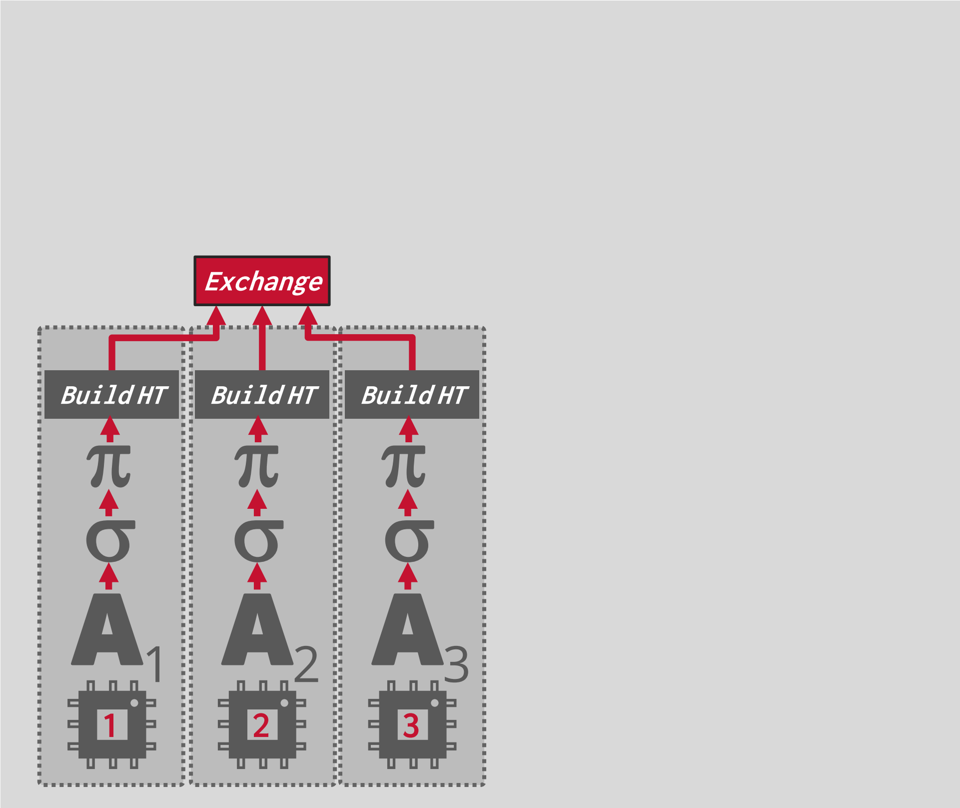
INTRA-OPERATOR PARALLELISM



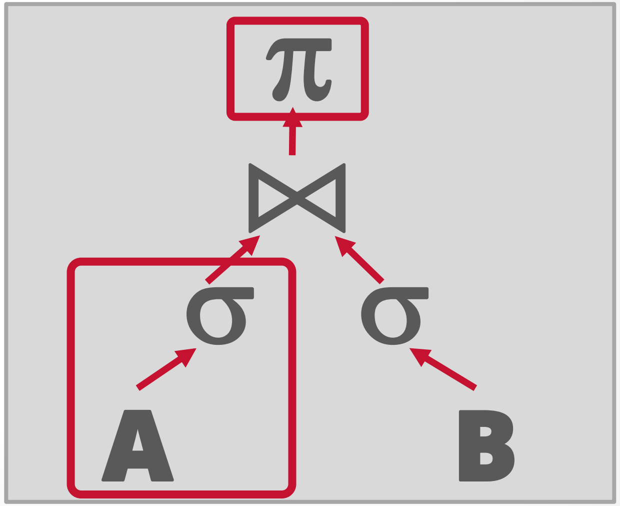

```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



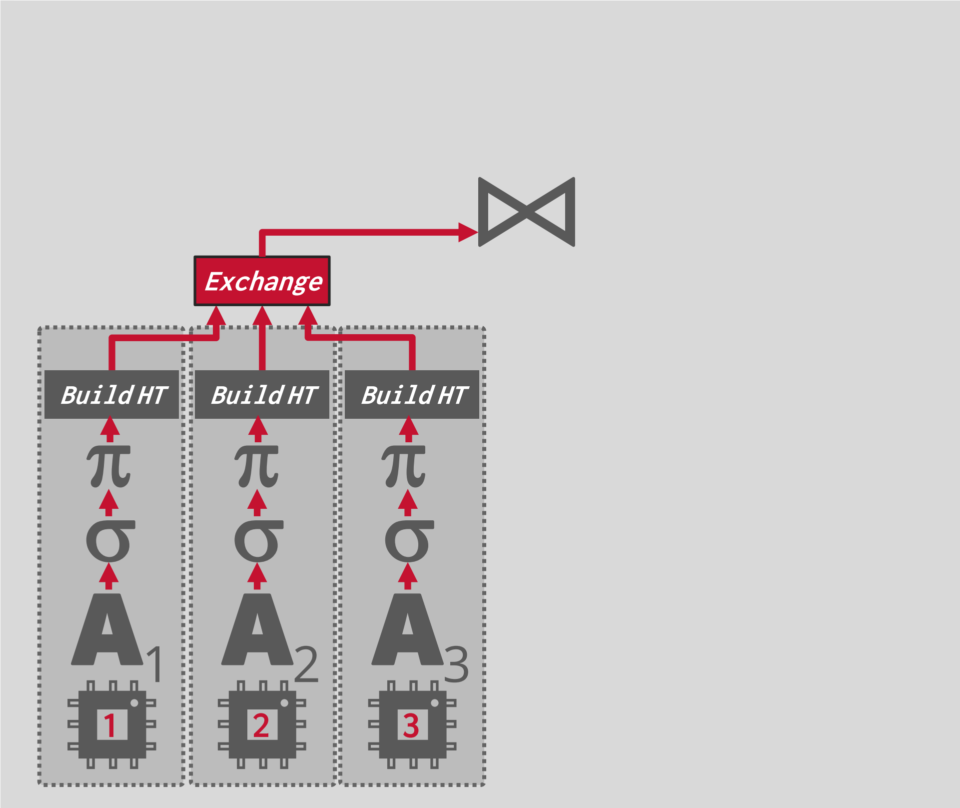
INTRA-OPERATOR PARALLELISM



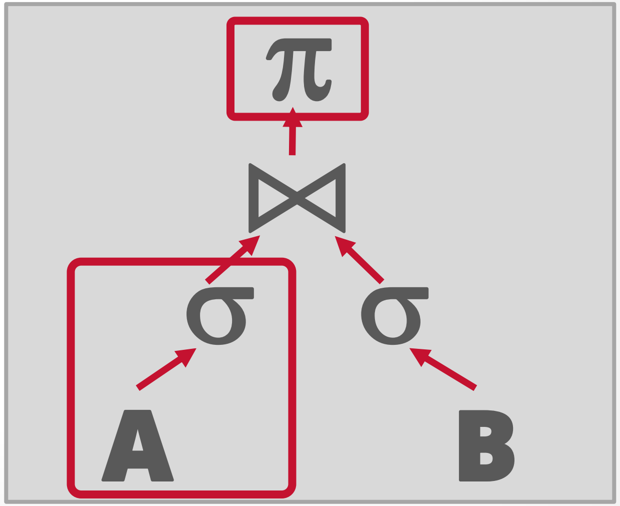

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



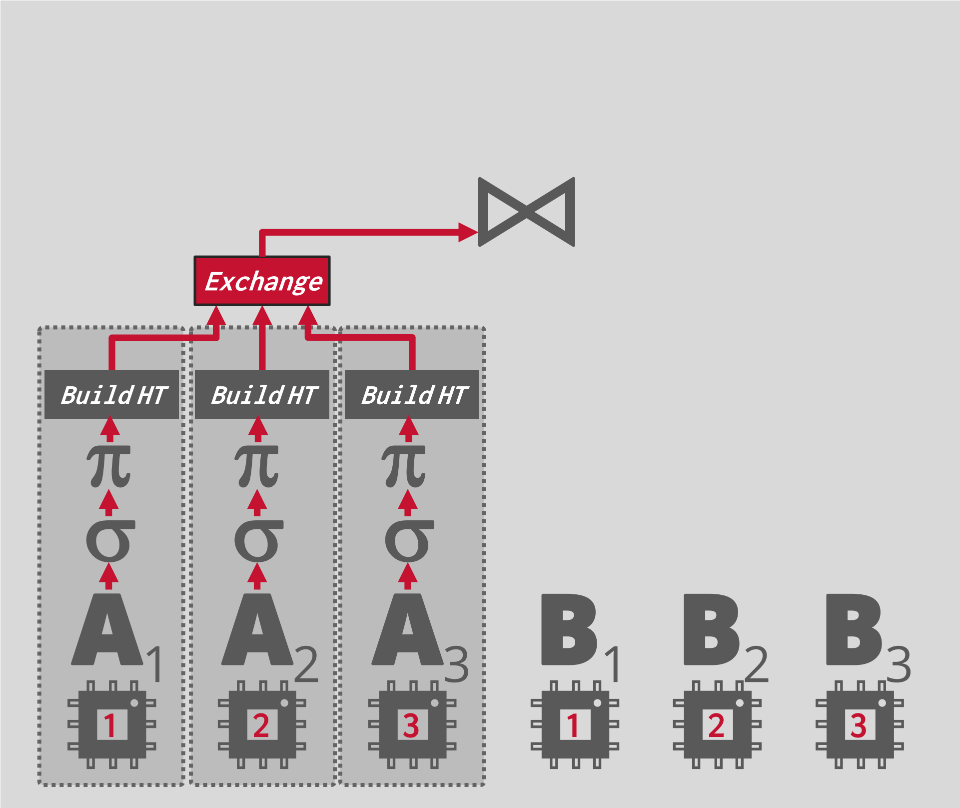
INTRA-OPERATOR PARALLELISM



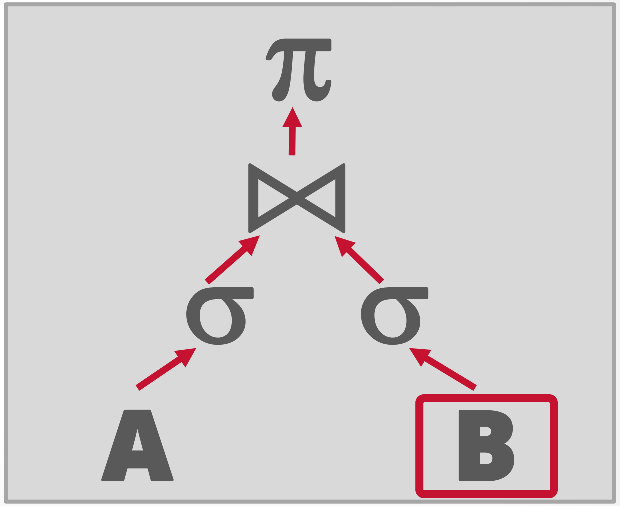

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



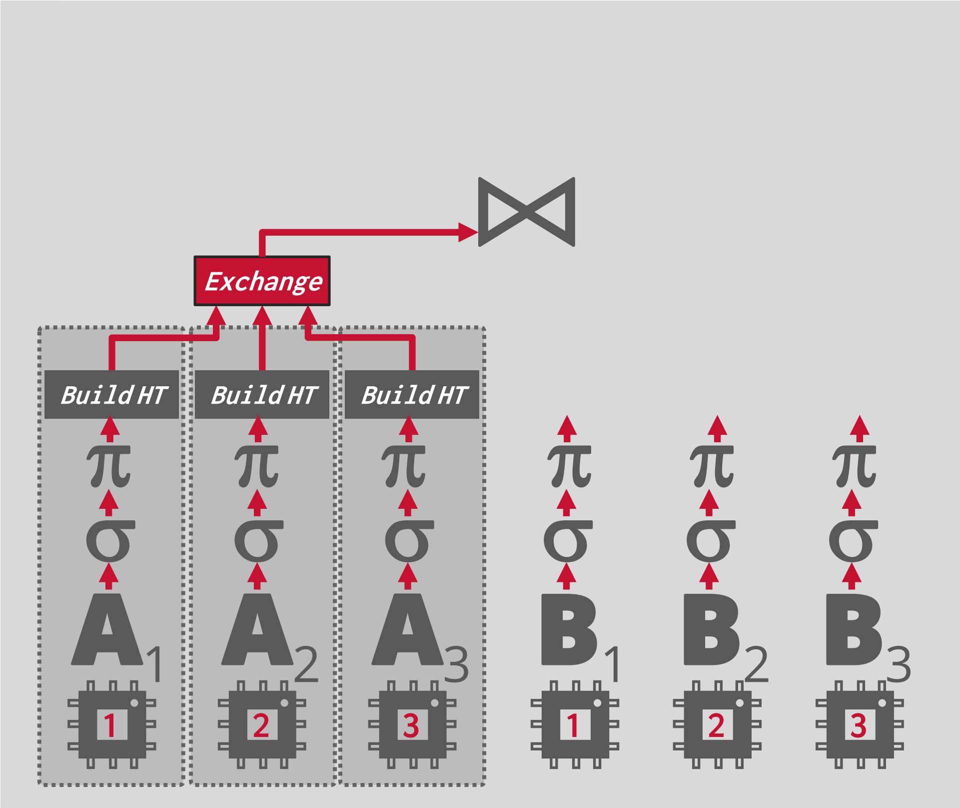
INTRA-OPERATOR PARALLELISM



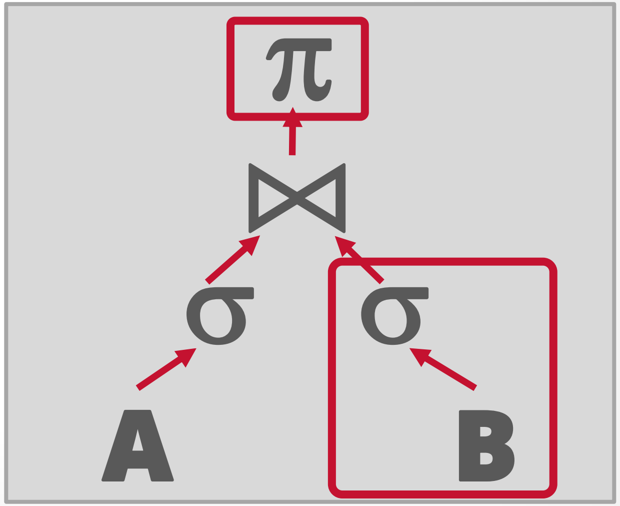

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



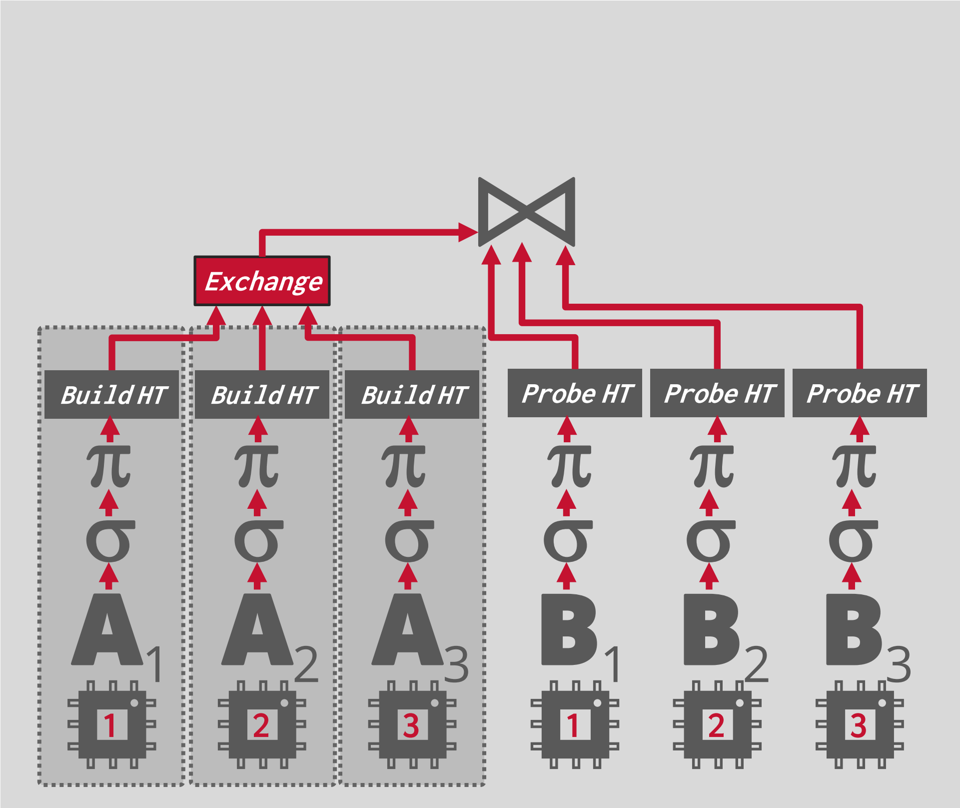
INTRA-OPERATOR PARALLELISM



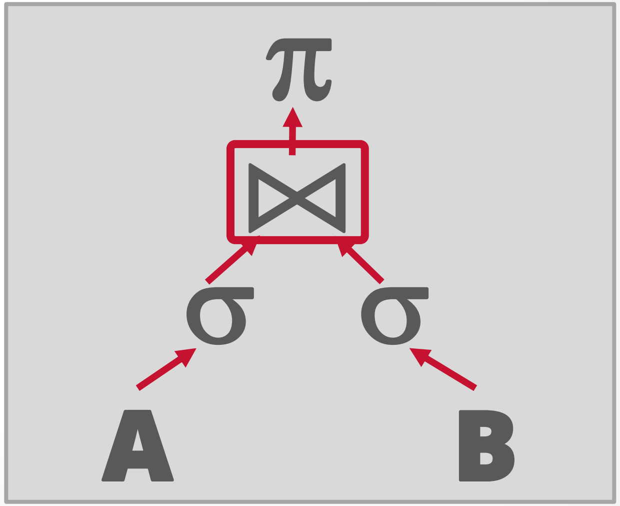

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



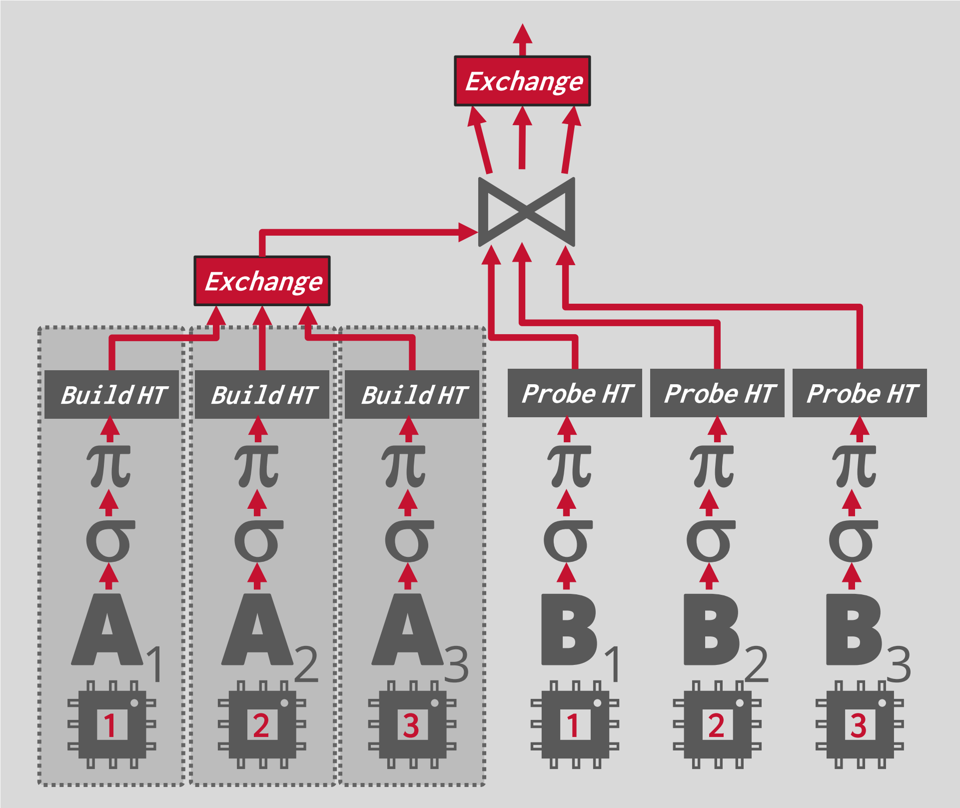
INTRA-OPERATOR PARALLELISM



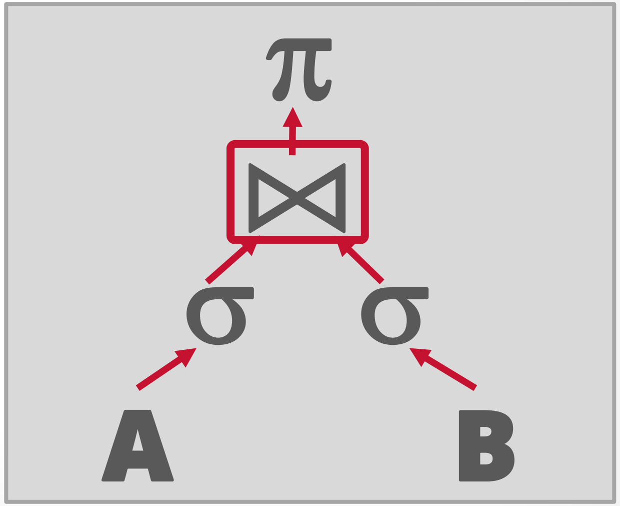

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



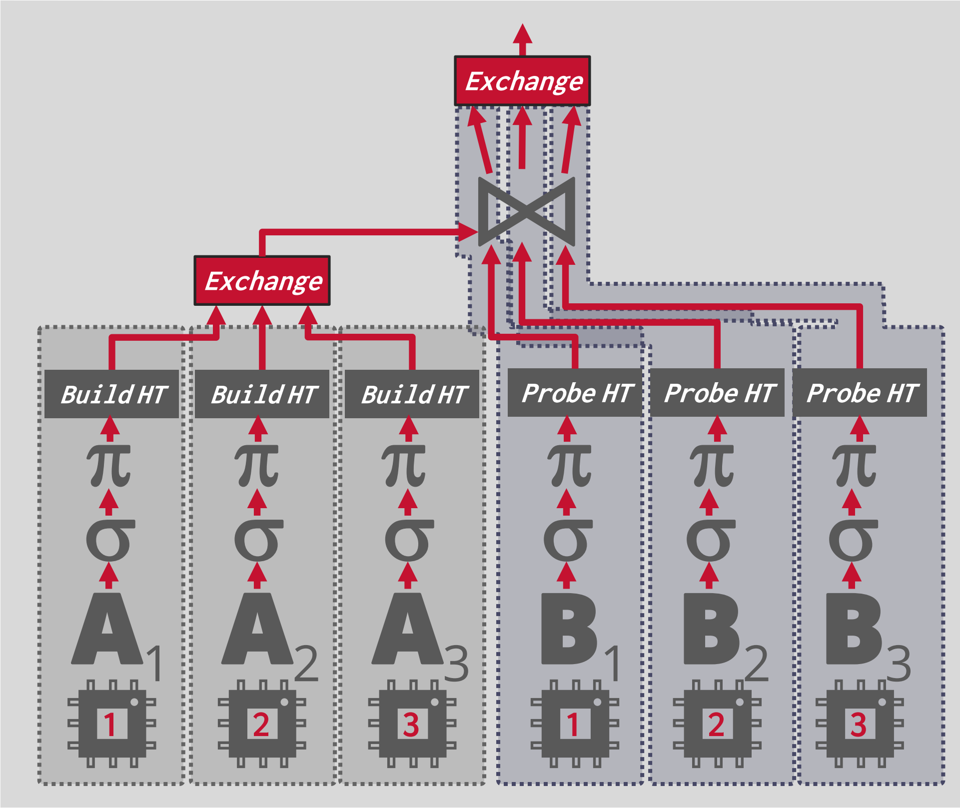
INTRA-OPERATOR PARALLELISM



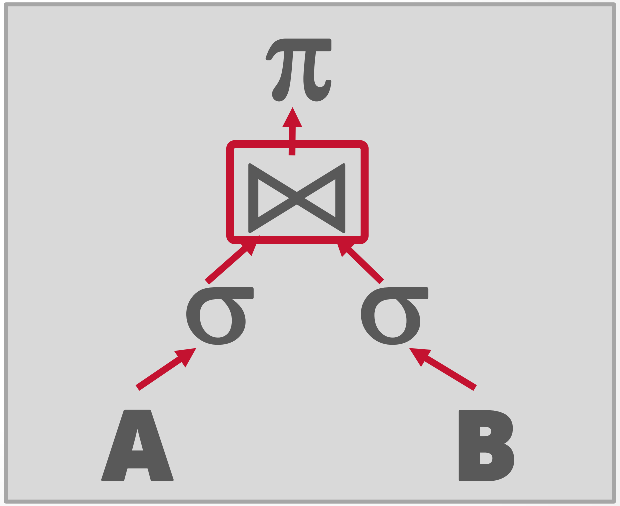

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



INTRA-OPERATOR PARALLELISM




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



EXCHANGE OPERATOR

Exchange Type #1 – Gather

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

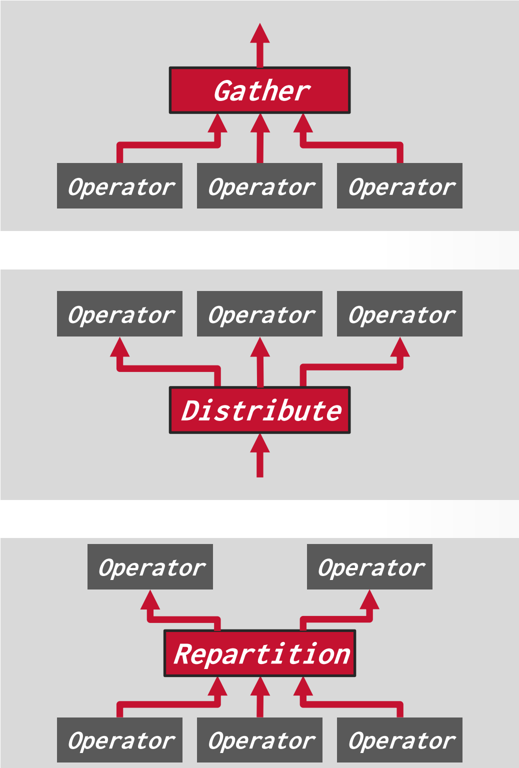
Exchange Type #2 – Distribute

→ 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).



Source: Craig Freedman

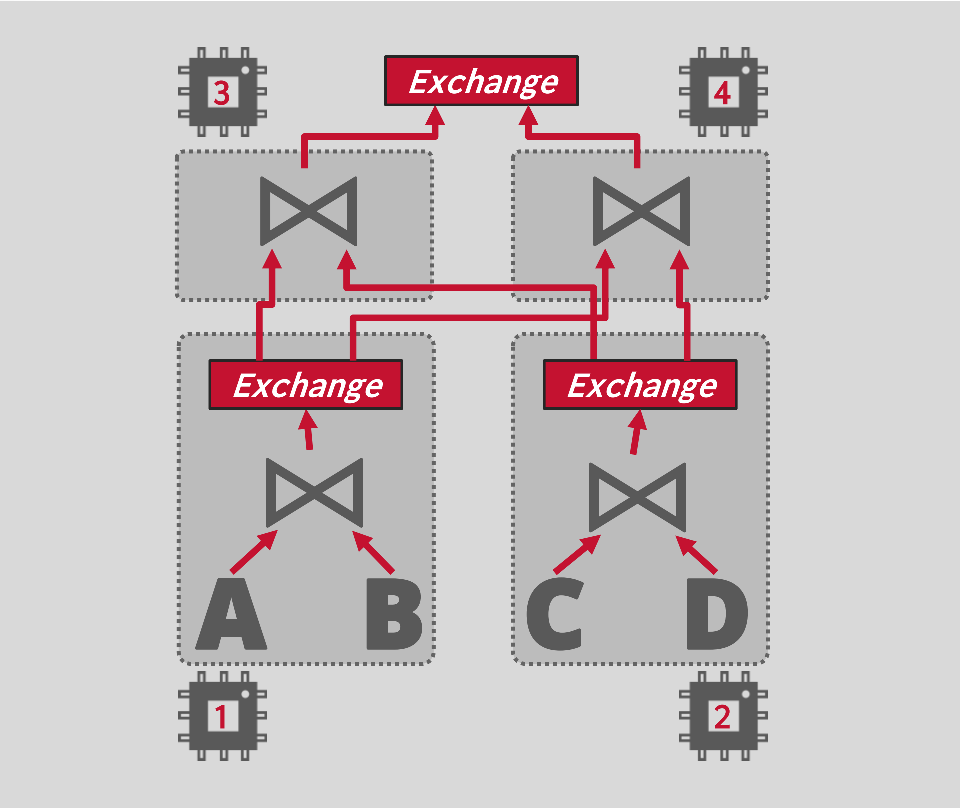
INTER-OPERATOR PARALLELISM

Approach #2: Inter-Operator (Vertical)

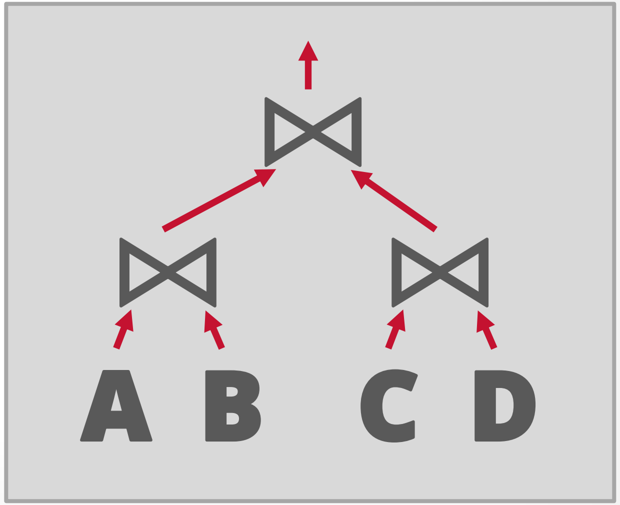
→ Operations are overlapped to pipeline data from one stage to the next without materialization.
→ Workers execute multiple operators from different segments of a query plan at the same time.
→ Still need exchange operators to combine intermediate results from segments.

Also called **pipelined parallelism**.

INTER-OPERATOR PARALLELISM




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



OBSERVATION

Instead of building a new DBMS from scratch, one can instead use standalone libraries for executing vectorized query operators on columnar data.

→ Input is a DAG of physical operators.

→ Require external scheduling and orchestration.

Notable implementations:

→ Velox

→ DataFusion

→ Intel OAP

→ Polars

THE COMPOSABLE DATA MANAGEMENT SYSTEM MANIFESTO
VLDB 2023

META VELOX

Extensible C++ library to support high-performance single-node query execution.

→ No SQL parser!
→ No meta-data catalog!
→ No cost-based optimizer!

Velox takes in a physical plan (DAG of operators) as its input for execution. It then produces the output to the specified location.

VELOX: META'S UNIFIED EXECUTION ENGINE
VLDB 2022

VELOX: OVERVIEW

Push-based Vectorized Query Processing

Precompiled Primitives + Codegen Expressions (C++)

Arrow Compatible (extended)

Adaptive Query Optimization

Sort-Merge + Hash Joins

VELOX: STORAGE

Velox does not "own" data and it does not have a proprietary on-disk data format.

Instead, it exposes APIs to define **connectors** to retrieve data from systems and **adapters** to decode/encode storage formats.

→ Systems: S3, HDFS

→ Formats: Parquet, ORC/DWRF, Alpha

VELOX: COMPONENTS

Type System

Expression Engine

Internal Data Representation

Function API

Operator Engine

Storage Connectors / Adapters

Resource Manager

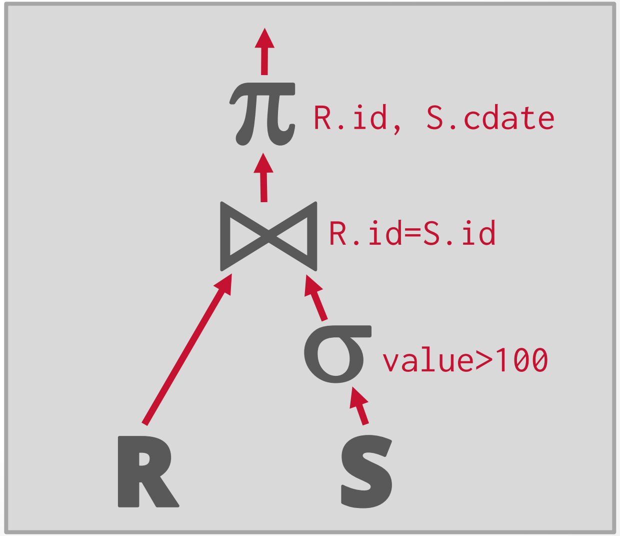
Source: Pedro Pedreira

OPERATOR OUTPUT

For tuple **r** ∈ **R** and tuple **s** ∈ **S** that match on join attributes, concatenate **r** and **s** together into a new tuple.

Output contents can vary:
→ Depends on processing model
→ Depends on storage model
→ Depends on data requirements in query

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



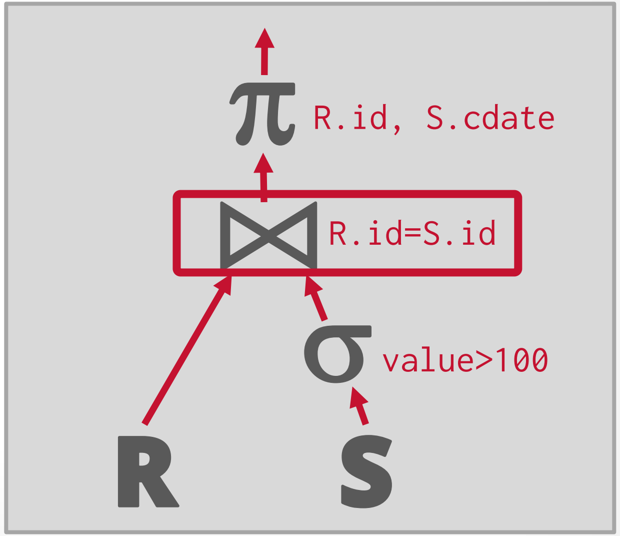
OPERATOR OUTPUT

For tuple **r** ∈ **R** and tuple **s** ∈ **S** that match on join attributes, concatenate **r** and **s** together into a new tuple.

Output contents can vary:

→ Depends on processing model
→ Depends on storage model
→ Depends on data requirements in query

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



OPERATOR OUTPUT: DATA

Early Materialization:

→ Copy the values for the attributes in outer and inner tuples into a new output tuple.

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

R(id,name)

id	name
123	abc

⋈

S(id,value,cdate)

id	value	cdate
123	1000	2/14/2024
123	2000	2/14/2024

R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	2/14/2024
123	abc	123	2000	2/14/2024

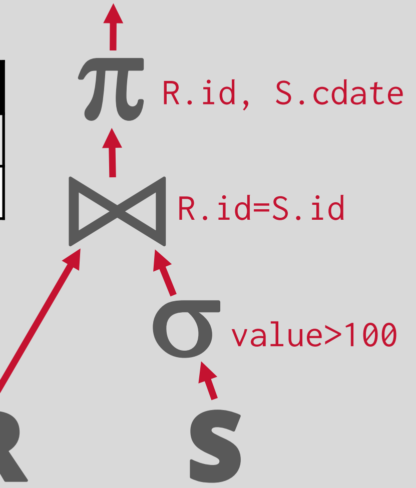
OPERATOR OUTPUT: DATA

Early Materialization:

→ Copy the values for the attributes in outer and inner tuples into a new output tuple.

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

R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	2/14/2024
123	abc	123	2000	2/14/2024



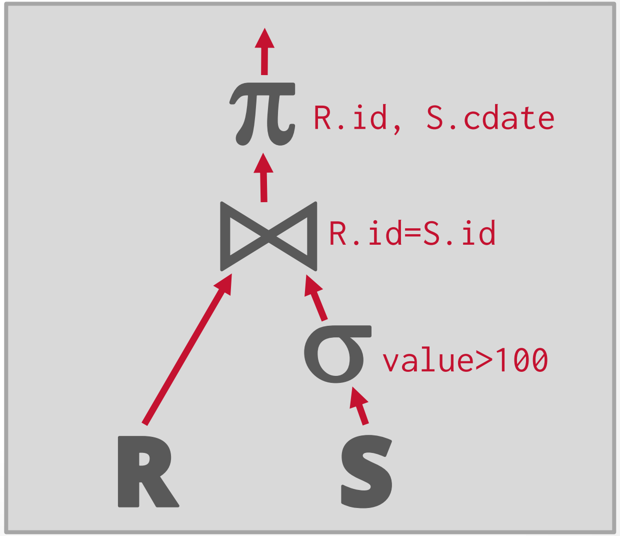
OPERATOR OUTPUT: DATA

Early Materialization:

→ Copy the values for the attributes in outer and inner tuples into a new output tuple.

Subsequent operators in the query plan never need to go back to the base tables to get more data.

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



OPERATOR OUTPUT: RECORD IDS

Late Materialization:

→ Only copy the joins keys along with the tuple IDs (e.g., column offsets) of the matching tuples.

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

R(id,name)

id	name
123	abc

⋈

S(id,value,cdate)

id	value	cdate
123	1000	2/14/2024
123	2000	2/14/2024

R.id	R.TID	S.id	S.TID
123	R.###	123	S.###
123	R.###	123	S.###

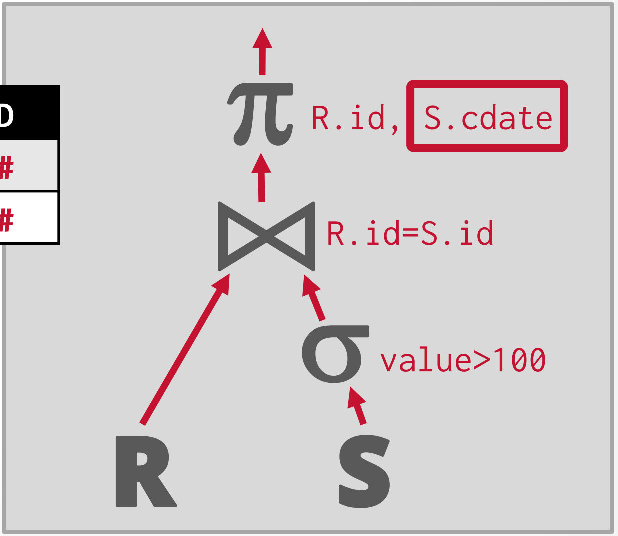
OPERATOR OUTPUT: RECORD IDS

Late Materialization:

→ Only copy the joins keys along with the tuple IDs (e.g., column offsets) of the matching tuples.

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

R.id	R.TID	S.id	S.TID
123	R.###	123	S.###
123	R.###	123	S.###



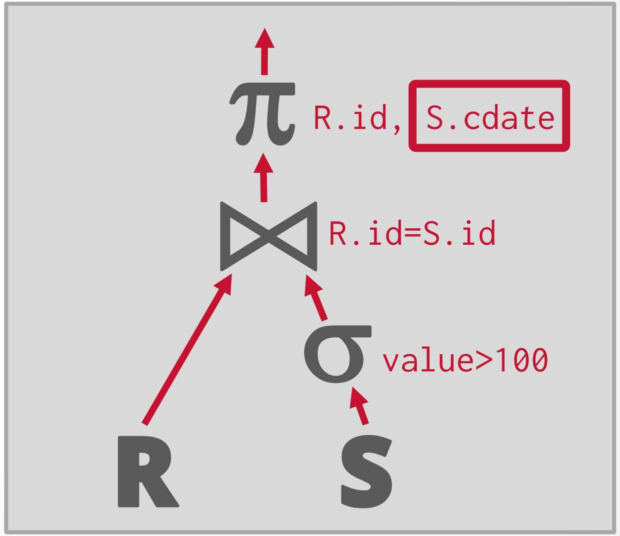
OPERATOR OUTPUT: RECORD IDS

Late Materialization:

→ Only copy the joins keys along with the tuple IDs (e.g., column offsets) of the matching tuples.

Ideal for column stores because the DBMS does not copy data that is not needed for the query.

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



OBSERVATION

The encoding schemes for Parquet, ORC, and other file formats are different enough that the DBMS cannot use the same handler code for each format.

→ Too much engineering overhead to maintain multiple version of the same operators.

Instead, the DBMS converts all input data to a single **<u>internal representation</u>** that it propagates through a query plan.

INTERNAL REPRESENTATION

How the DBMS stores and encodes vectors of data that it passes between query operators.

→ All values must be fixed-length to use offsets to find corresponding values across columns.

Ideal properties:

→ Move data structures without serializing.

→ Zero-copy shared memory access.

APACHE ARROW

Self-describing, language-agnostic in-memory columnar data format for cache-efficient + vectorized execution engines.

- → Supports both random + sequential access patterns.
- → Compiles basic expressions with LLVM (Gandiva).
- → Also provides additional resource management and communication components.

Arrow only supports two lightweight encoding schemes (Dictionary, RLE).

Source: Wes McKinney

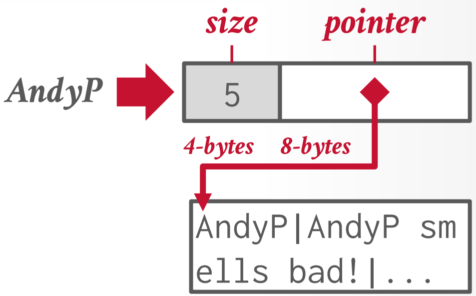
STRING STORAGE

Arrow originally stored strings as fixed-length pointers to an offset in a byte array.

Velox extended Arrow it to use **German-style String Storage**

→ Fixed-length portion contains size + prefix + payload.

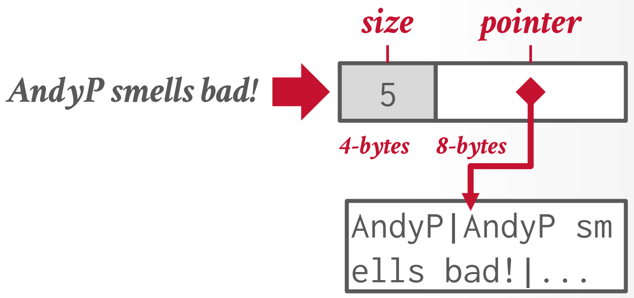
→ Payload contains full-string if it is 16-bytes or less. Otherwise, it is pointer ot the full string.



STRING STORAGE

Arrow originally stored strings as fixed-length pointers to an offset in a byte array.

Velox extended Arrow it to use **German-style String Storage**
→ Fixed-length portion contains size + prefix + payload.
→ Payload contains full-string if it is 16-bytes or less. Otherwise, it is pointer ot the full string.



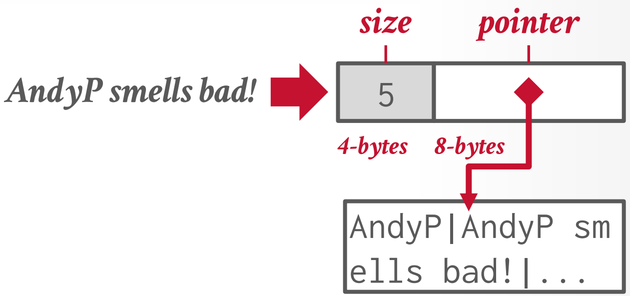
STRING STORAGE

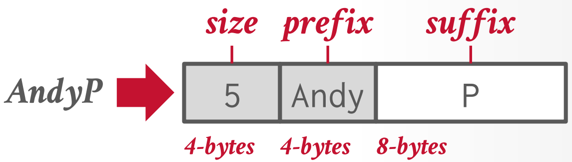
Arrow originally stored strings as fixed-length pointers to an offset in a byte array.

Velox extended Arrow it to use **German-style String Storage**

→ Fixed-length portion contains size + prefix + payload.

→ Payload contains full-string if it is 16-bytes or less. Otherwise, it is pointer ot the full string.





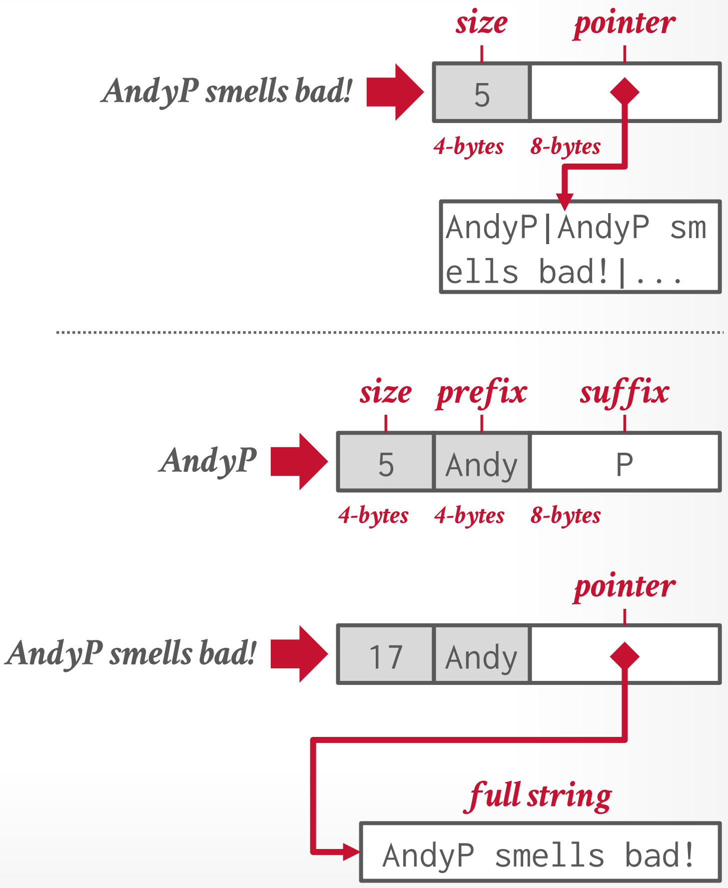
STRING STORAGE

Arrow originally stored strings as fixed-length pointers to an offset in a byte array.

Velox extended Arrow it to use **German-style String Storage**

→ Fixed-length portion contains size + prefix + payload.

→ Payload contains full-string if it is 16-bytes or less. Otherwise, it is pointer ot the full string.



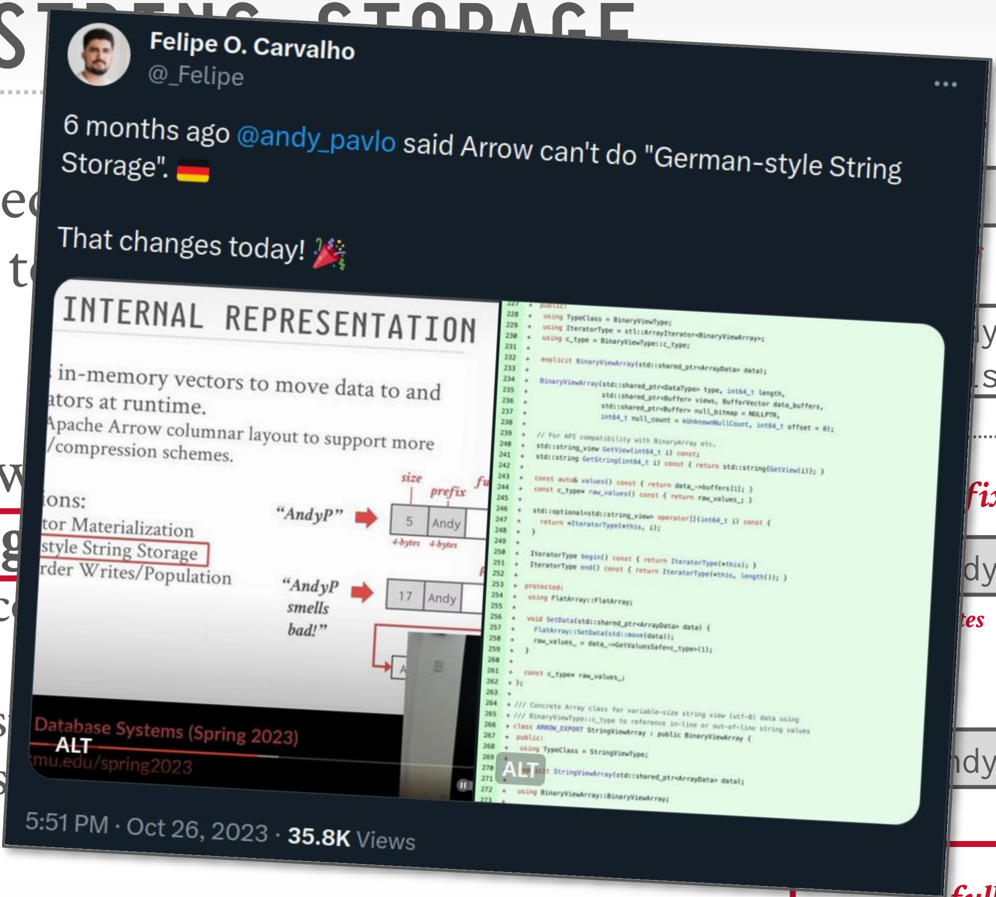
STRING STORAGE

Arrow originally stored
fixed-length pointers t
byte array.

Velox extended Arrow
German-style String

→ Fixed-length portion c
prefix + payload.

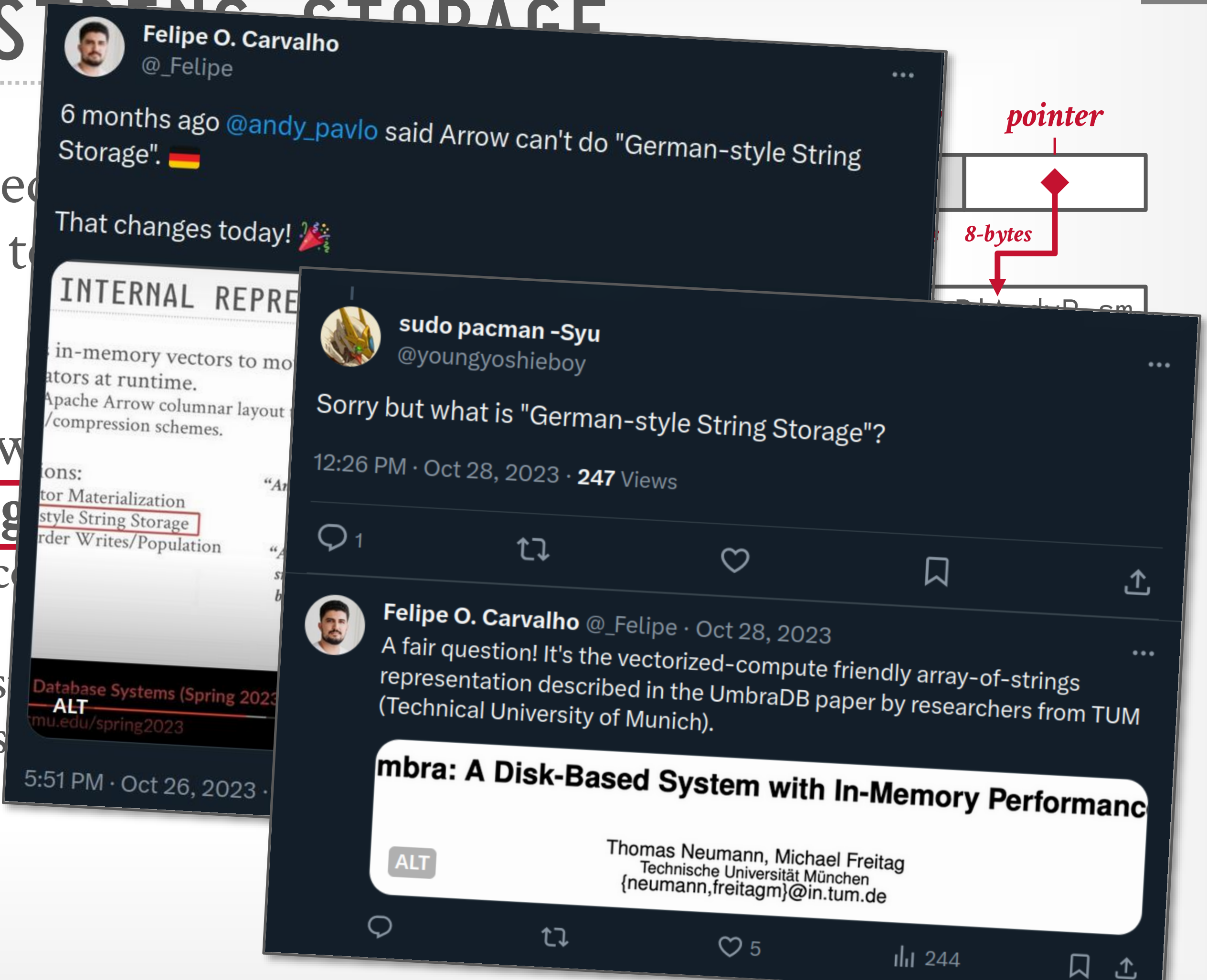
→ Payload contains full-s
bytes or less. Otherwis
the full string.



STRING STORAGE

Arrow originally stored
fixed-length pointers t
byte array.

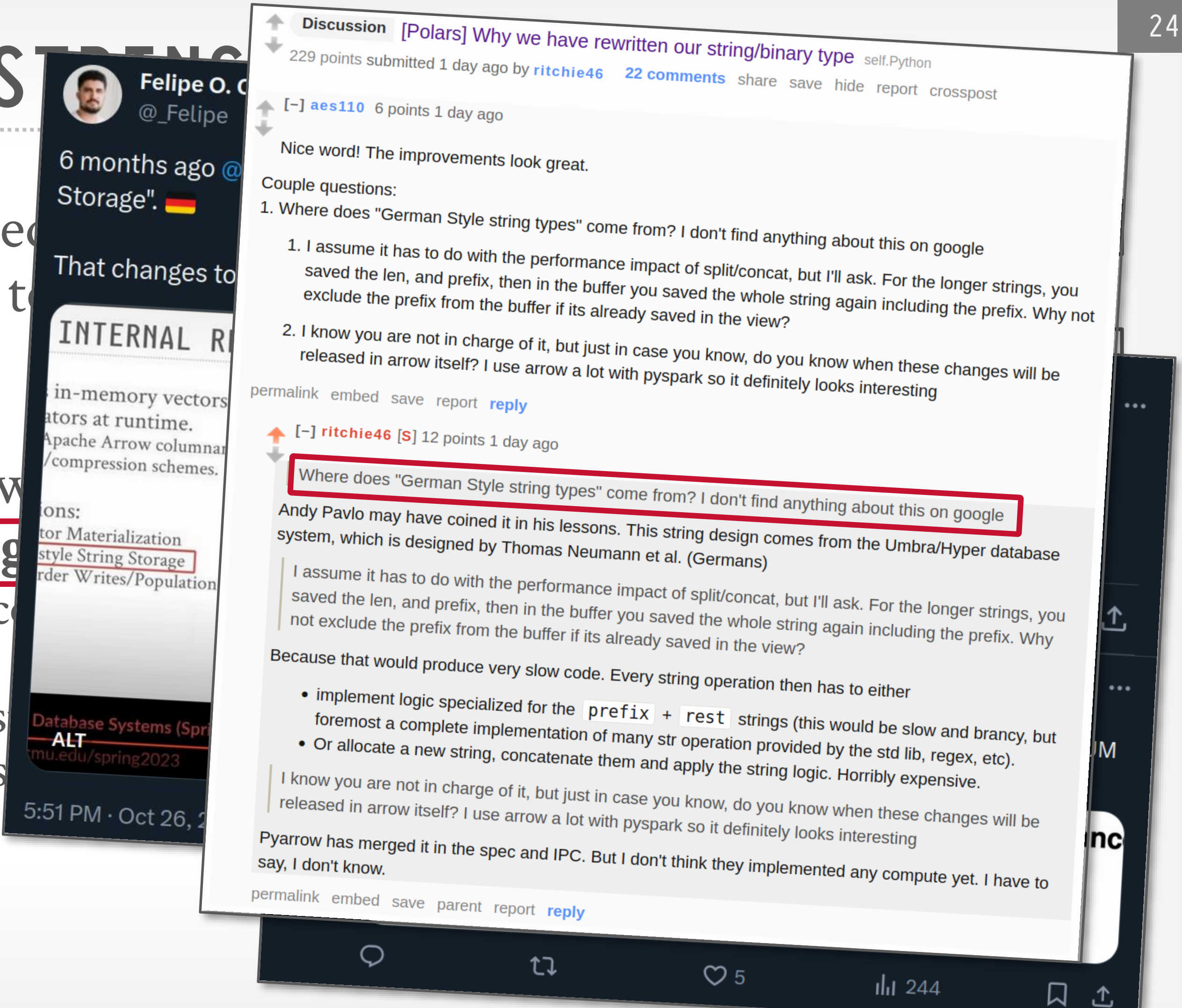
Velox extended Arrow
German-style String
→ Fixed-length portion c
prefix + payload.
→ Payload contains full-s
bytes or less. Otherwis
the full string.



S

Arrow originally stored fixed-length pointers t byte array.

Velox extended Arrow
German-style String
→ Fixed-length portion c prefix + payload.
→ Payload contains full-s bytes or less. Otherwis the full string.



Discussion [Polars] Why we have rewritten our string/binary type self.Python
229 points submitted 1 day ago by ritchie46 22 comments share save hide report crosspost

[–] aes110 6 points 1 day ago

Nice word! The improvements look great.

Couple questions:

1. Where does "German Style string types" come from? I don't find anything about this on google
   1. I assume it has to do with the performance impact of split/concat, but I'll ask. For the longer strings, you saved the len, and prefix, then in the buffer you saved the whole string again including the prefix. Why not exclude the prefix from the buffer if its already saved in the view?
2. I know you are not in charge of it, but just in case you know, do you know when these changes will be released in arrow itself? I use arrow a lot with pyspark so it definitely looks interesting

permalink embed save report reply

[–] ritchie46 [S] 12 points 1 day ago

> Where does "German Style string types" come from? I don't find anything about this on google

Andy Pavlo may have coined it in his lessons. This string design comes from the Umbra/Hyper database system, which is designed by Thomas Neumann et al. (Germans)

> I assume it has to do with the performance impact of split/concat, but I'll ask. For the longer strings, you saved the len, and prefix, then in the buffer you saved the whole string again including the prefix. Why not exclude the prefix from the buffer if its already saved in the view?

Because that would produce very slow code. Every string operation then has to either

- implement logic specialized for the `prefix` + `rest` strings (this would be slow and brancy, but foremost a complete implementation of many str operation provided by the std lib, regex, etc).
- Or allocate a new string, concatenate them and apply the string logic. Horribly expensive.

> I know you are not in charge of it, but just in case you know, do you know when these changes will be released in arrow itself? I use arrow a lot with pyspark so it definitely looks interesting

Pyarrow has merged it in the spec and IPC. But I don't think they implemented any compute yet. I have to say, I don't know.

permalink embed save parent report reply

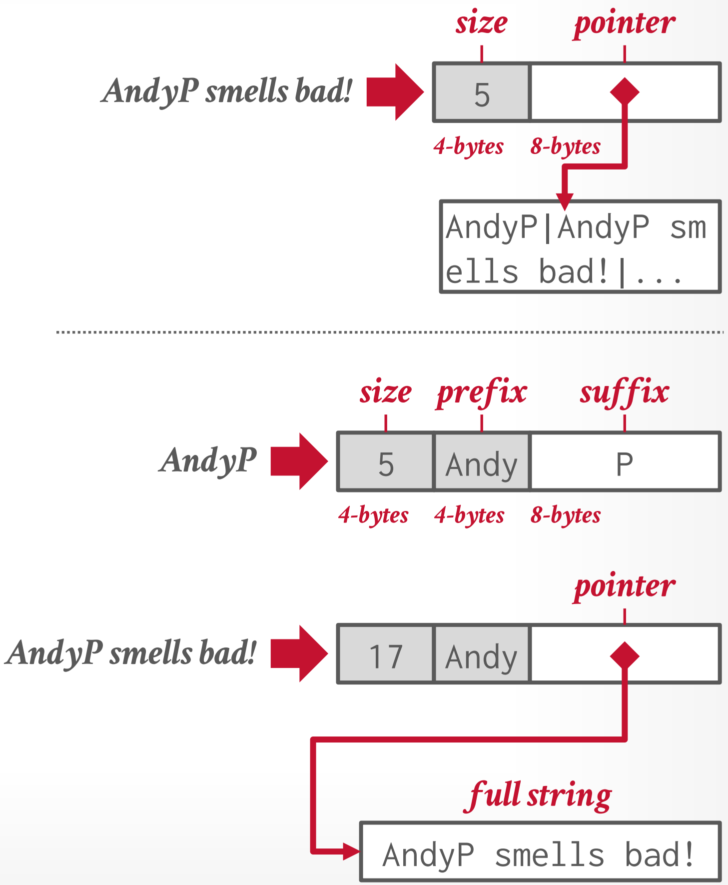
STRING STORAGE

Arrow originally stored strings as fixed-length pointers to an offset in a byte array.

Velox extended Arrow it to use **Umbra-style String Storage**

→ Fixed-length portion contains size + prefix + payload.

→ Payload contains full-string if it is 16-bytes or less. Otherwise, it is pointer ot the full string.



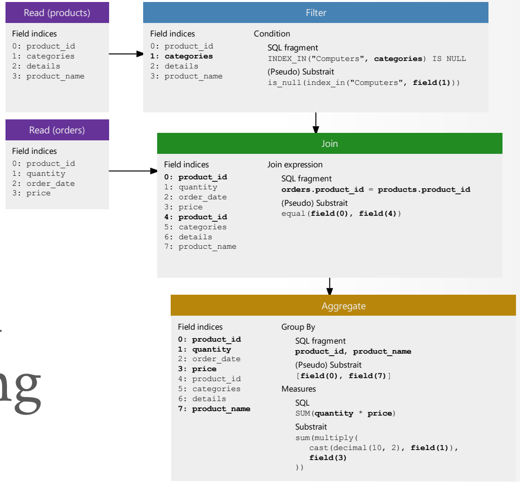
SUBSTRAIT (2021)

Open-source specification to represent relational algebra query plans.

→ Think of it like Arrow but for query plans.

The idea is that systems can share physical query plans with each other without having to convert them into a native API/DSL.

→ Federated DBMSs are hard.



DATAFUSION (2019)

Extensible vectorized execution library for Apache Arrow data.

→ Written in Rust for the kids!

Provides more front-end functionality features to build a complete DBMS than Velox

→ SQL and DataFrame APIs.

→ Query Optimizer

Examples: InfluxDB, CeresDB, CnosDB, Seafowl

TODAY'S AGENDA

~~Parallel Execution~~

~~Operator Output~~

~~Intermediate Data Representation~~

Expression Evaluation

Adaptive Execution

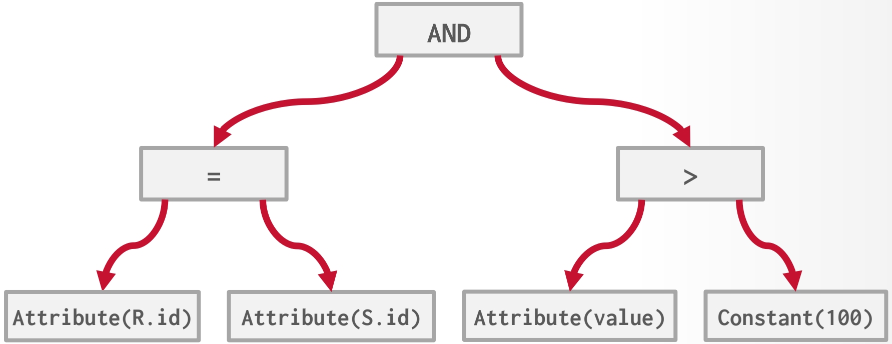
EXPRESSION EVALUATION

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

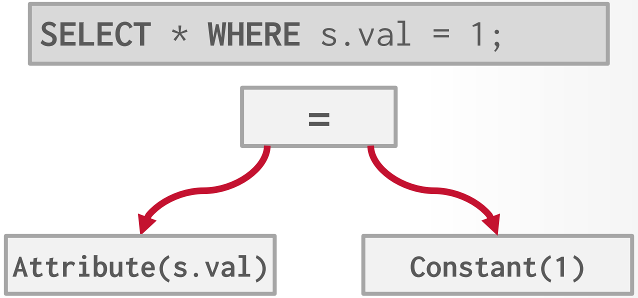
```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100;
```



EXPRESSION EVALUATION

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.



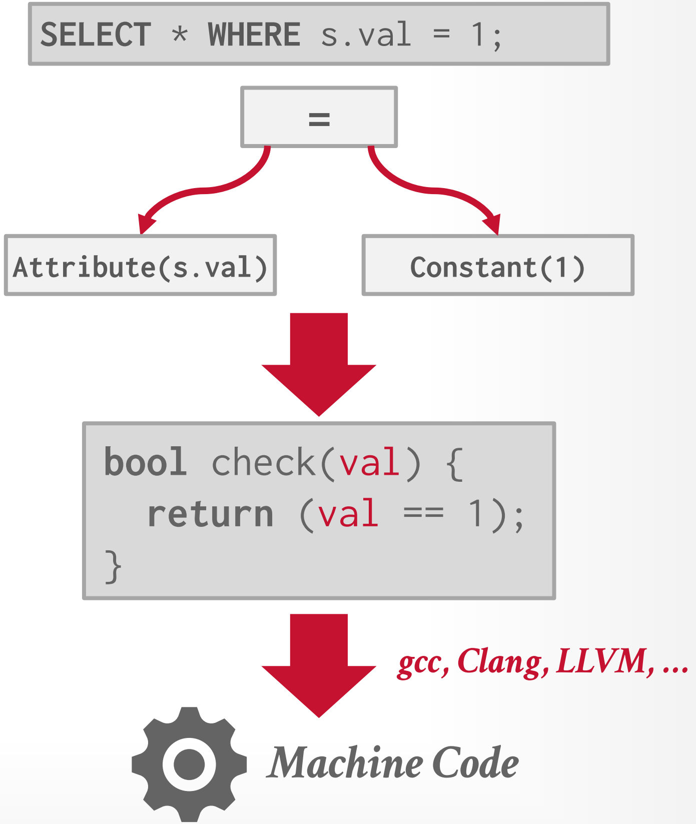
EXPRESSION EVALUATION

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…



VELOX: EXPRESSION ENGINE

Velox converts expression trees into a flattened intermediate representation that they then execute during query processing.

→ Think of it like an array of function pointers to precompiled (untemplated) primitives.

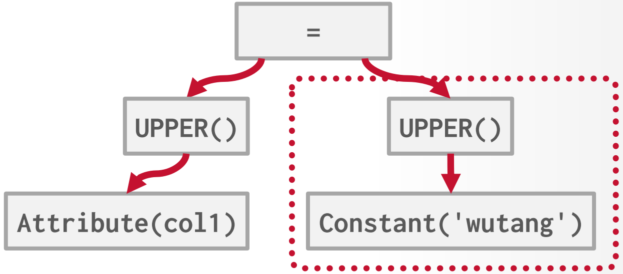
Experimental branch transpiles IR into C++ code and then compiles to machine code via exec.

VELOX: EXPRESSION ENGINE

Constant Folding:

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

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



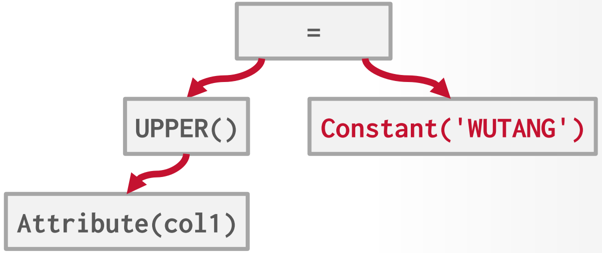
Source: Deepak Majeti

VELOX: EXPRESSION ENGINE

Constant Folding:

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

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



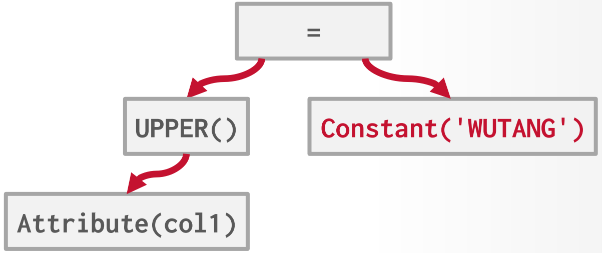
Source: Deepak Majeti

VELOX: EXPRESSION ENGINE

Constant Folding:

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

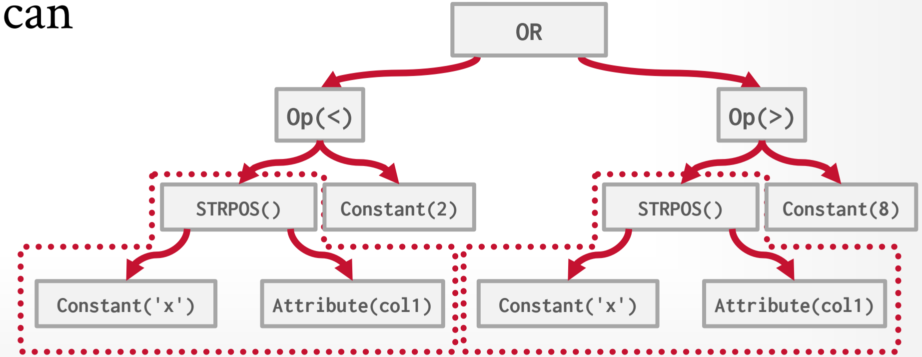
```
WHERE UPPER(col1) = UPPER('wutang');
```



Common Sub-Expr. Elimination:

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

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



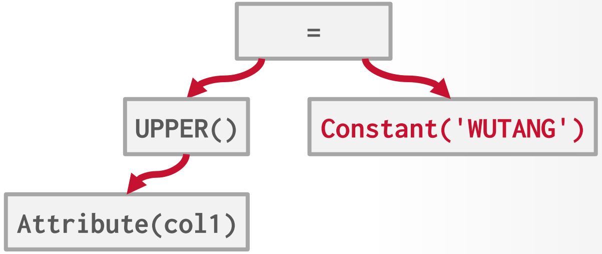
Source: Deepak Majeti

VELOX: EXPRESSION ENGINE

Constant Folding:

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

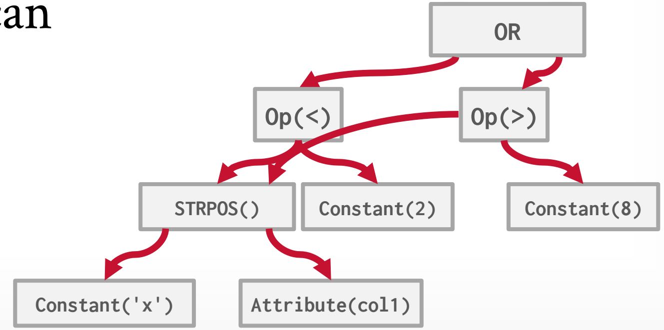
```
WHERE UPPER(col1) = UPPER('wutang');
```



Common Sub-Expr. Elimination:

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

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



OBSERVATION

An execution engine is only as good as the query plan that it has. Query optimizers rely on cost models derived from statistics extracted from data.

- → Bad query plans negate all the optimizations that we've talked about so far.

But how can the DBMS optimize a query if there are no statistics?

- → Data files the DBMS has never seen before.
- → Query APIs from other DBMSs (connectors).

ADAPTIVE QUERY PROCESSING

Allow the execution engine to modify a query's plan and expression trees while it is running.

The goal is to use information gathered from executing some part of the query to decide how to best proceed with executing the rest of the query.

→ In the extreme case, the DBMS can give up and return the query to the optimizer but with new information.

We will discuss how to modify query plans later in the semester.

ADAPTIVE QUERY PROCESSING IN THE LOOKING GLASS
CIDR 2005

VELOX: EXPRESSION ADAPTIVITY

Predicate Reordering

→ Decide the ordering of predicates based on their selectivity and computational cost.

Column Prefetching

→ Asynchronous retrieval of columns during expression evaluations.

Not Null Fast Paths

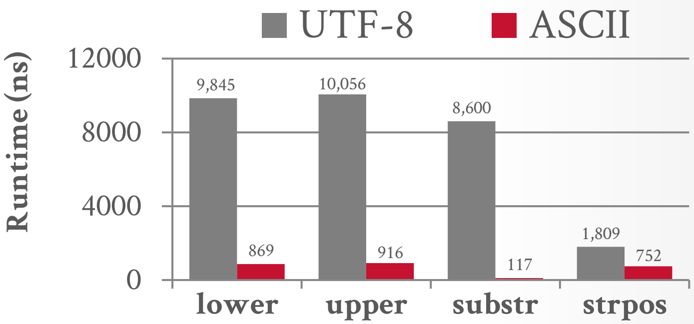
→ Switch to faster functions that skip null checking if input vector has no null values.

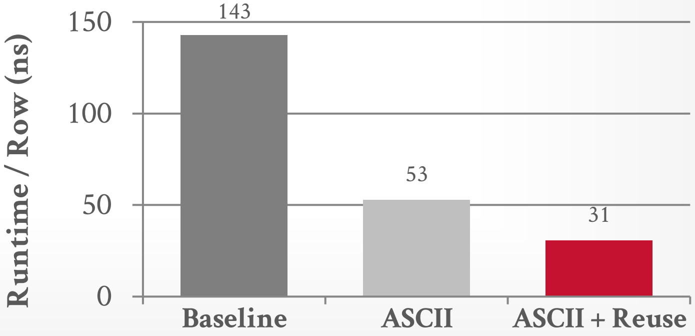
Elide ASCII Encoding Checks

→ Use faster ASCII funcs if no UTF-8 data.

→ Bonus: Reuse buffers for output!

```
WHERE SLOW_FUNC(col1) = true
  AND FAST_FUNC(col2) = true
```





Source: Pedro Pedreira

PARTING THOUGHTS

Today's lecture is a quick overview of more design considerations when building an execution engine.
→ Each of these topics could be an entire lecture on its own.

Arrow is the best choice for internal data representation. It continues to evolve and improve.

NEXT CLASS

Vectorized Operator Algorithms