DuckDB

In-Process Analytical Database System

Mark Raasveldt

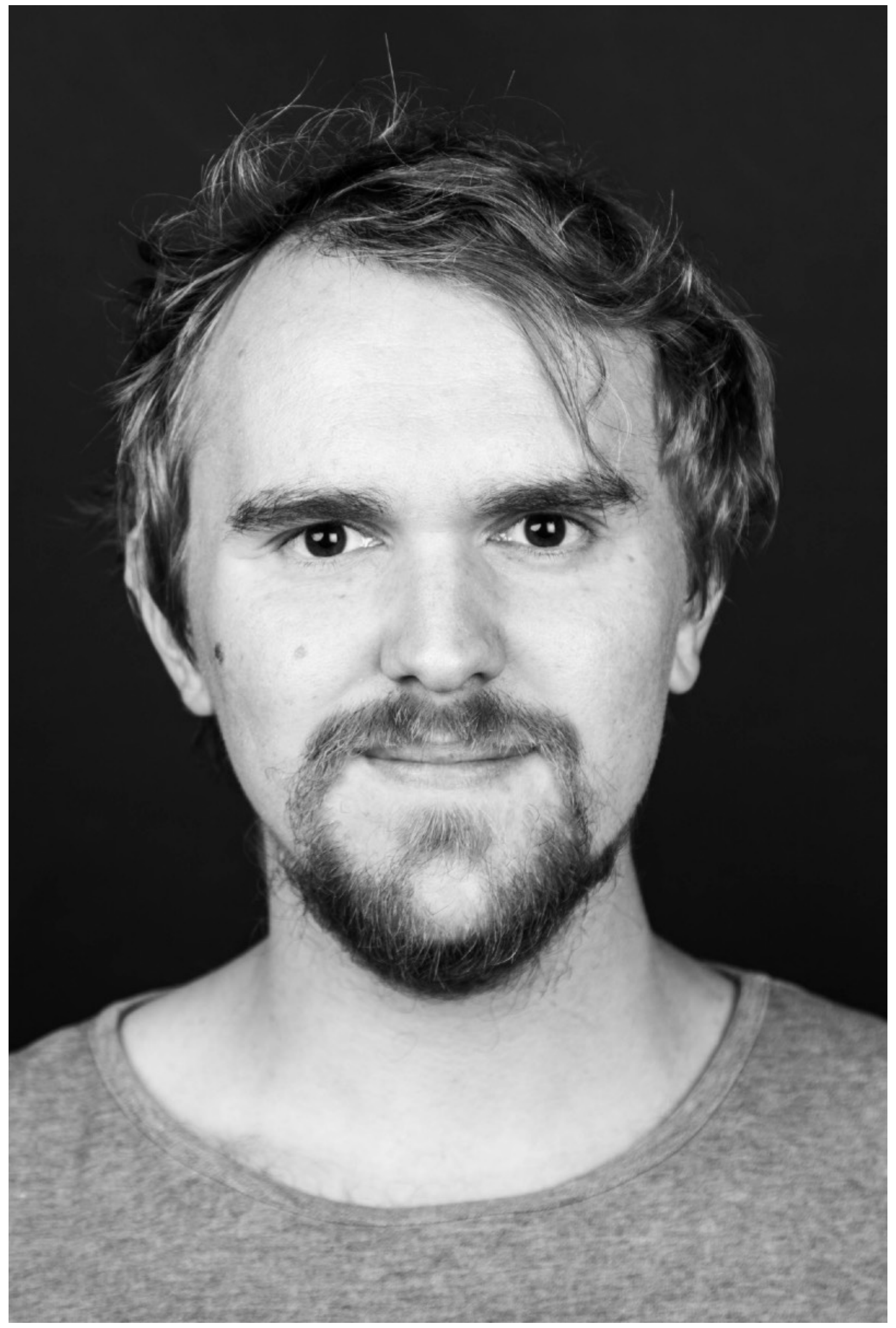
DuckDB Labs

Who Am I?

Mark Raasveldt

CTO of DuckDB Labs

Previously at CWI, Database Architectures



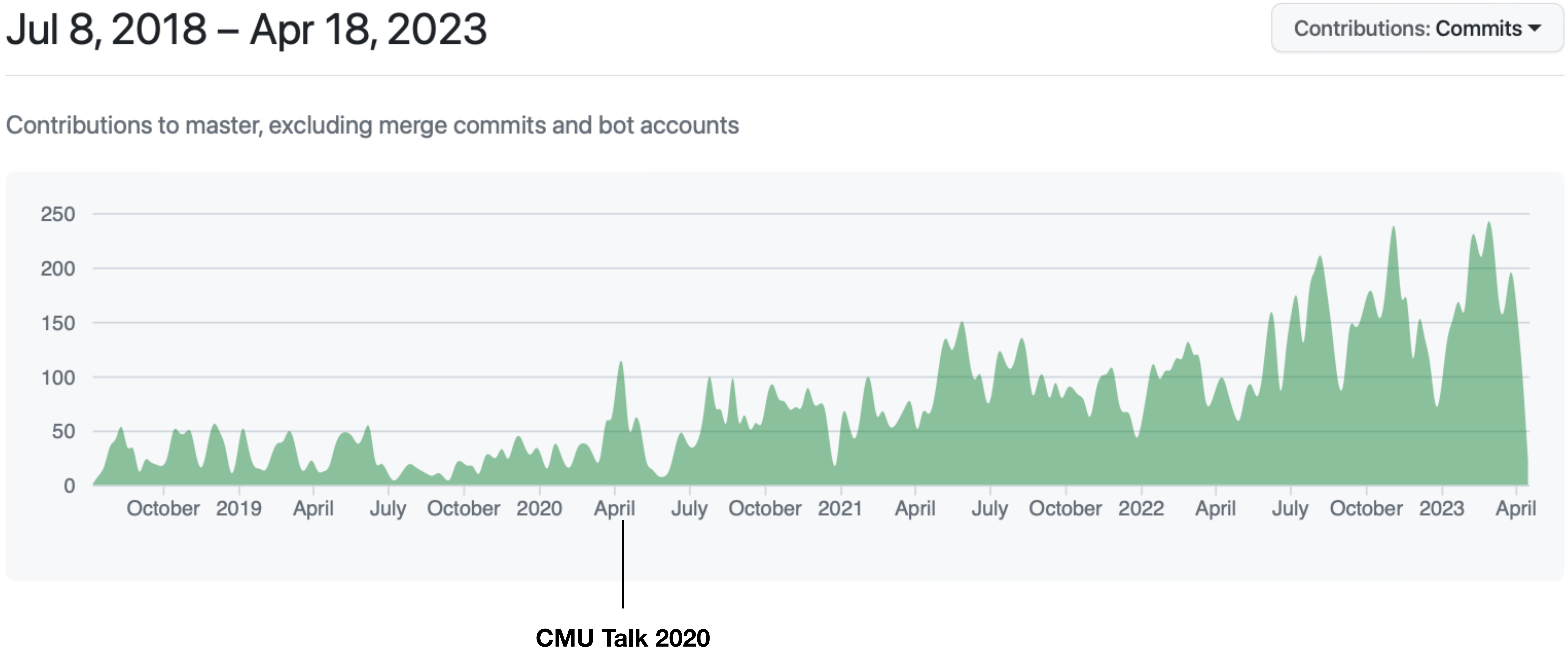
What is DuckDB?

- DuckDB
- In-Process OLAP DBMS
  - “The SQLite for Analytics”
- Free and Open Source (MIT)
- duckdb.org

DuckDB


Carnegie Mellon University
Quarantine 2020
Database Talks
Mark Raasveldt
(DuckDB, CWI)
1:08:01

DuckDB


Jul 8, 2018 – Apr 18, 2023
Contributions: Commits
Contributions to master, excluding merge commits and bot accounts
250
200
150
100
50
0
October
2019
April
July
October
2020
April
July
October
2021
April
July
October
2022
April
July
October
2023
April
CMU Talk 2020

DuckDB

Team


Hannes Mühleisen
Co-Founder & CEO


Mark Raasveldt
Co-Founder & CTO


Pedro Holanda
Chief of Operations


Richard Wesley
Software Engineer


Laurens Kuiper
Software Engineer


Sam Ansmink
Software Engineer


Tania Bogatsch
Software Engineer


Thijs Bruineman
Software Engineer


Elliana May
Software Engineer


Tom Ebergen
Software Engineer


Max Gabrielsson
Software Engineer


Carlo Piovesan
Software Engineer


Alex Monahan
Training & Documentation


Marie Wiener
Office Manager & HR

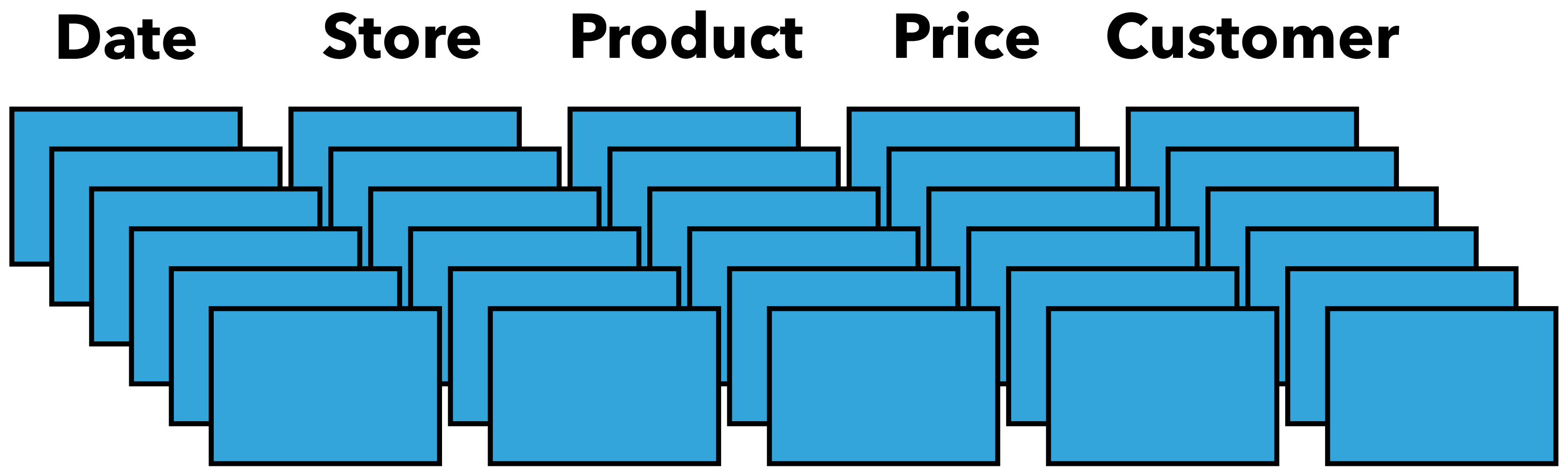

Alina Stiben
Admin Assistant & Event Manager


Lars Verdoes
Intern

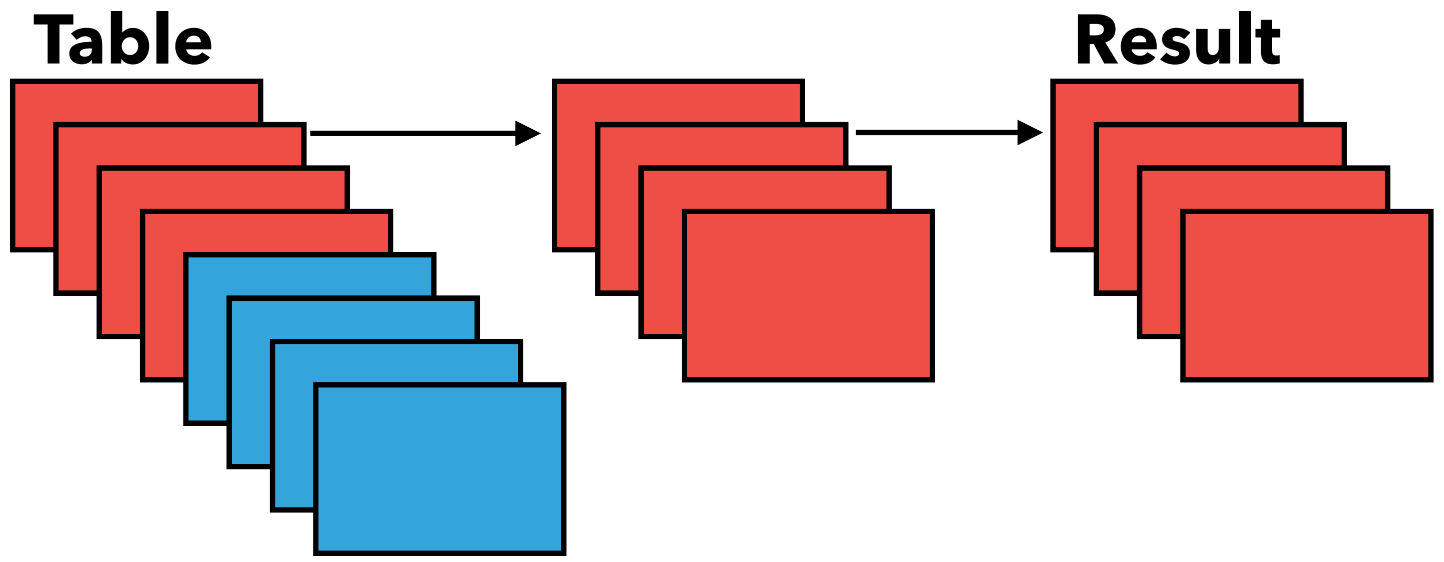
System Overview

DuckDB - Overview

Column-Store



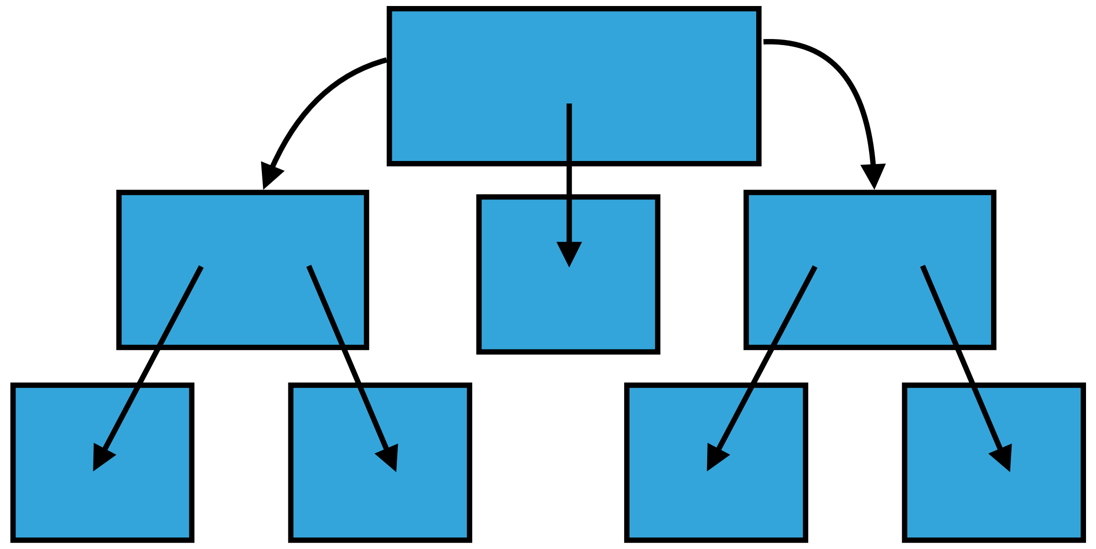
Vectorized Processing



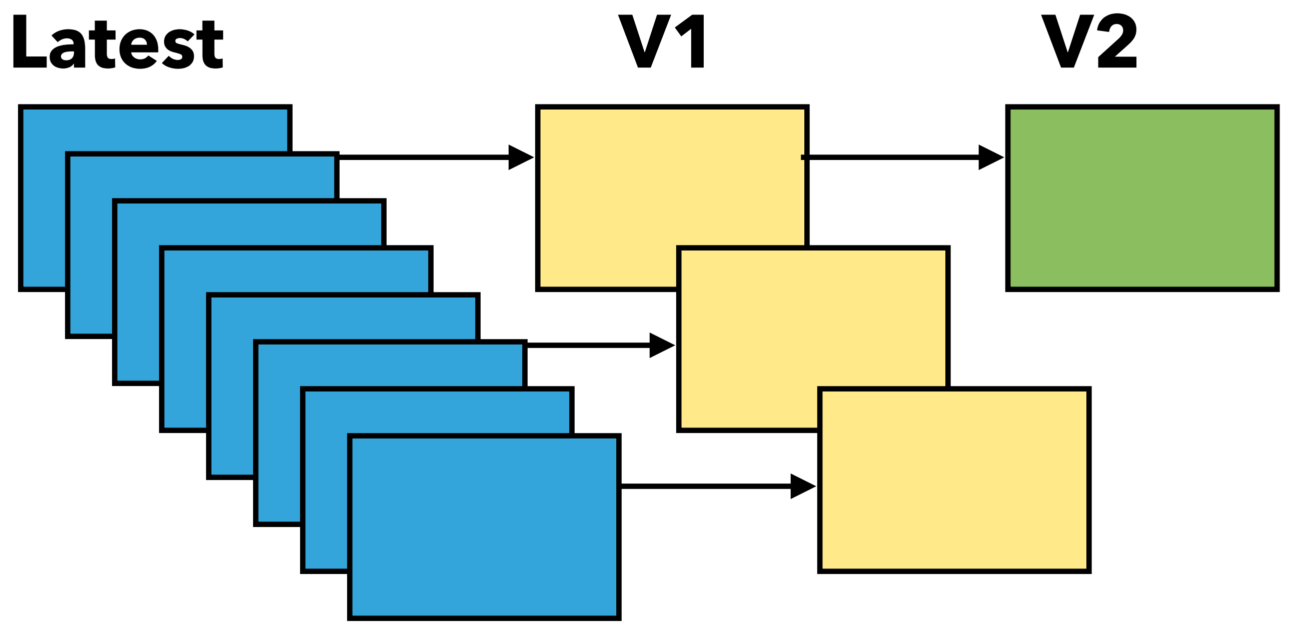
Single-File Storage

database.db

ART Index



MVCC

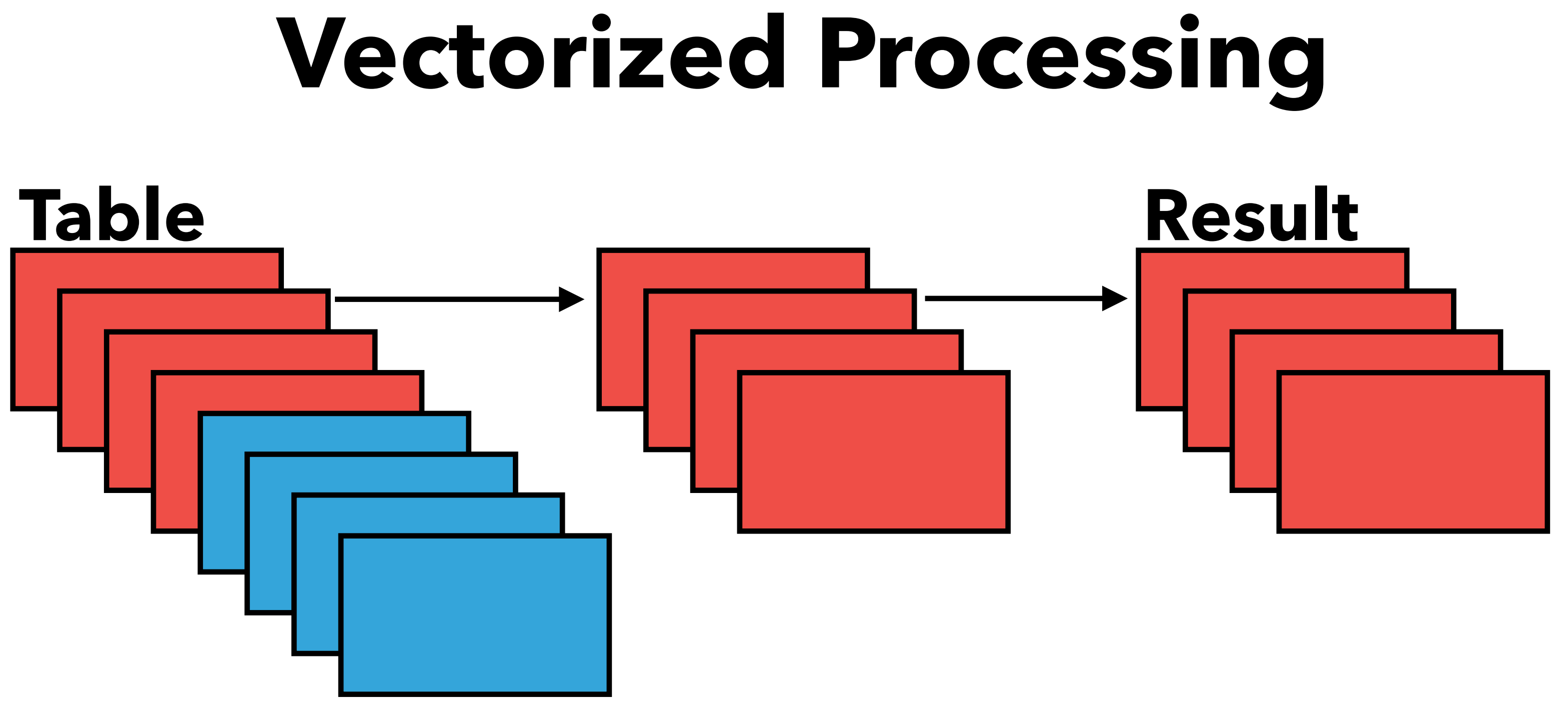


Parser

DuckDB - Vectors

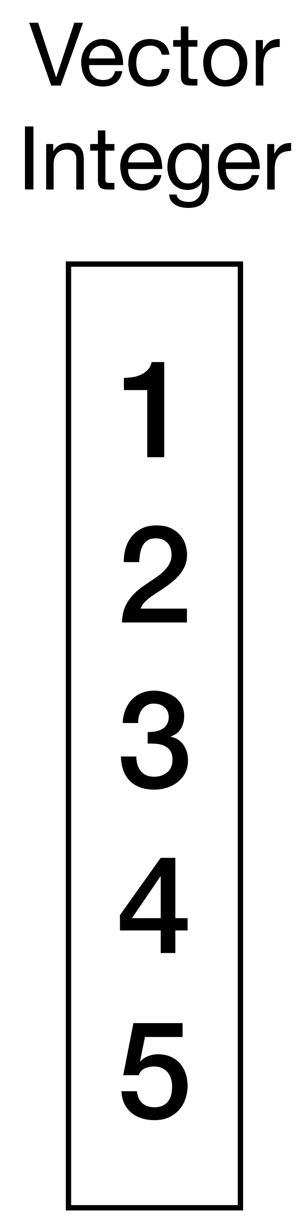
- DuckDB uses a **vectorized push-based model**
  - Vectors flow through the operators

- Vectors are the bread and butter of the engine
- DuckDB has a custom **vector format**
  - Similar to Arrow - but designed for execution
  - Co-designed with Velox team

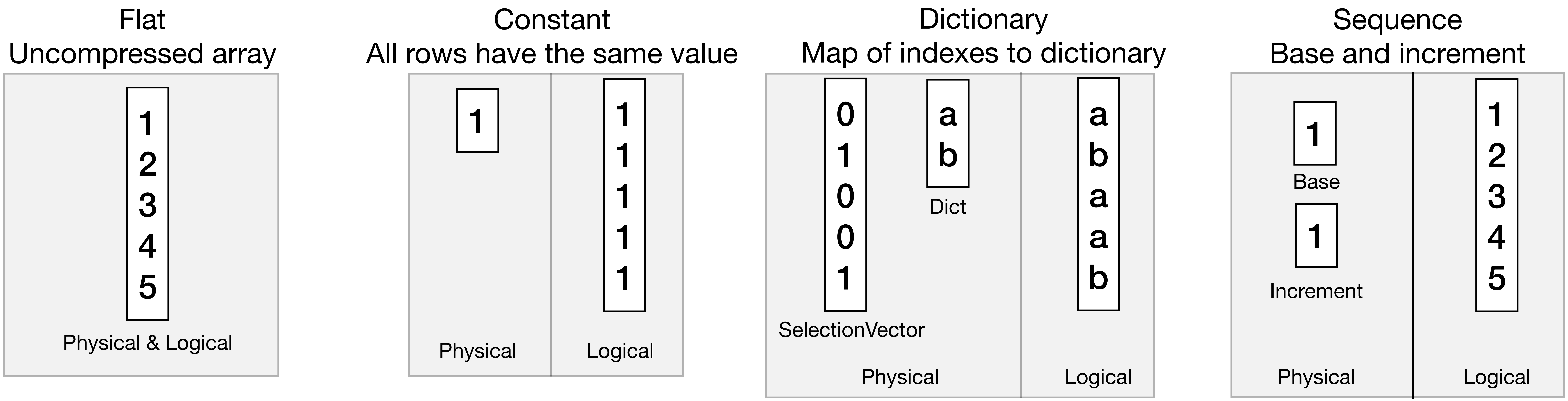


DuckDB - Vectors

- Vectors hold data of a single type
- For scalar types vectors are **logically** arrays
- `VectorType` determines **physical representation**
    - Allows us to push **compressed data** into the engine!



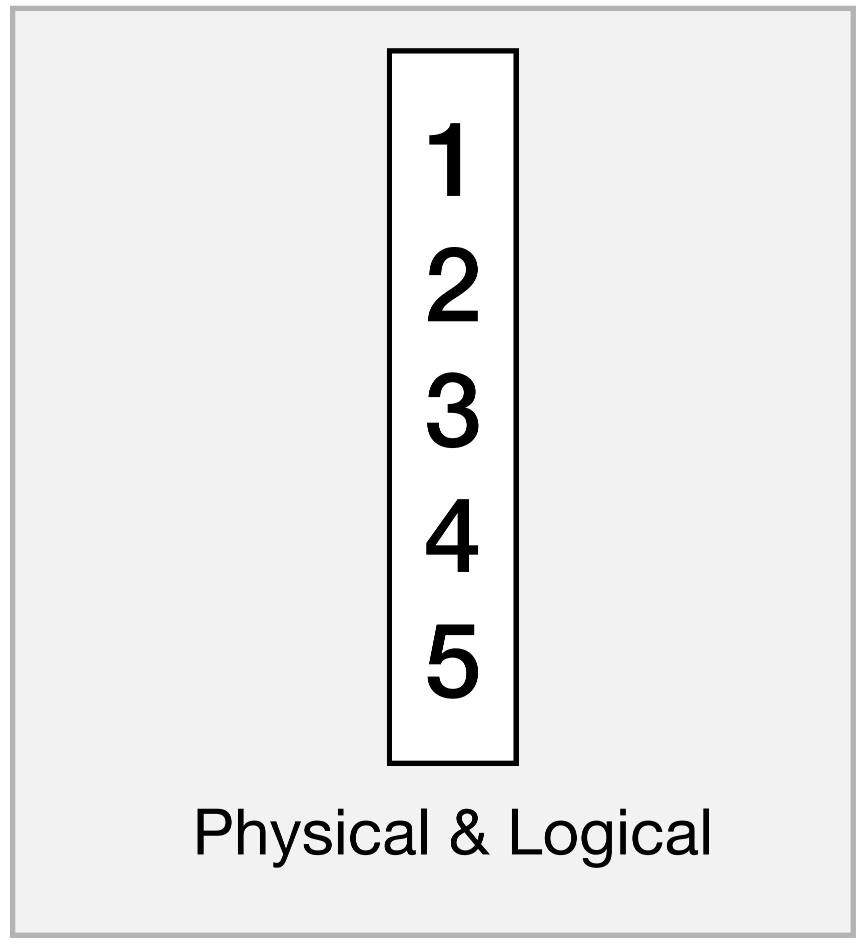
DuckDB - Vectors



DuckDB - Vectors

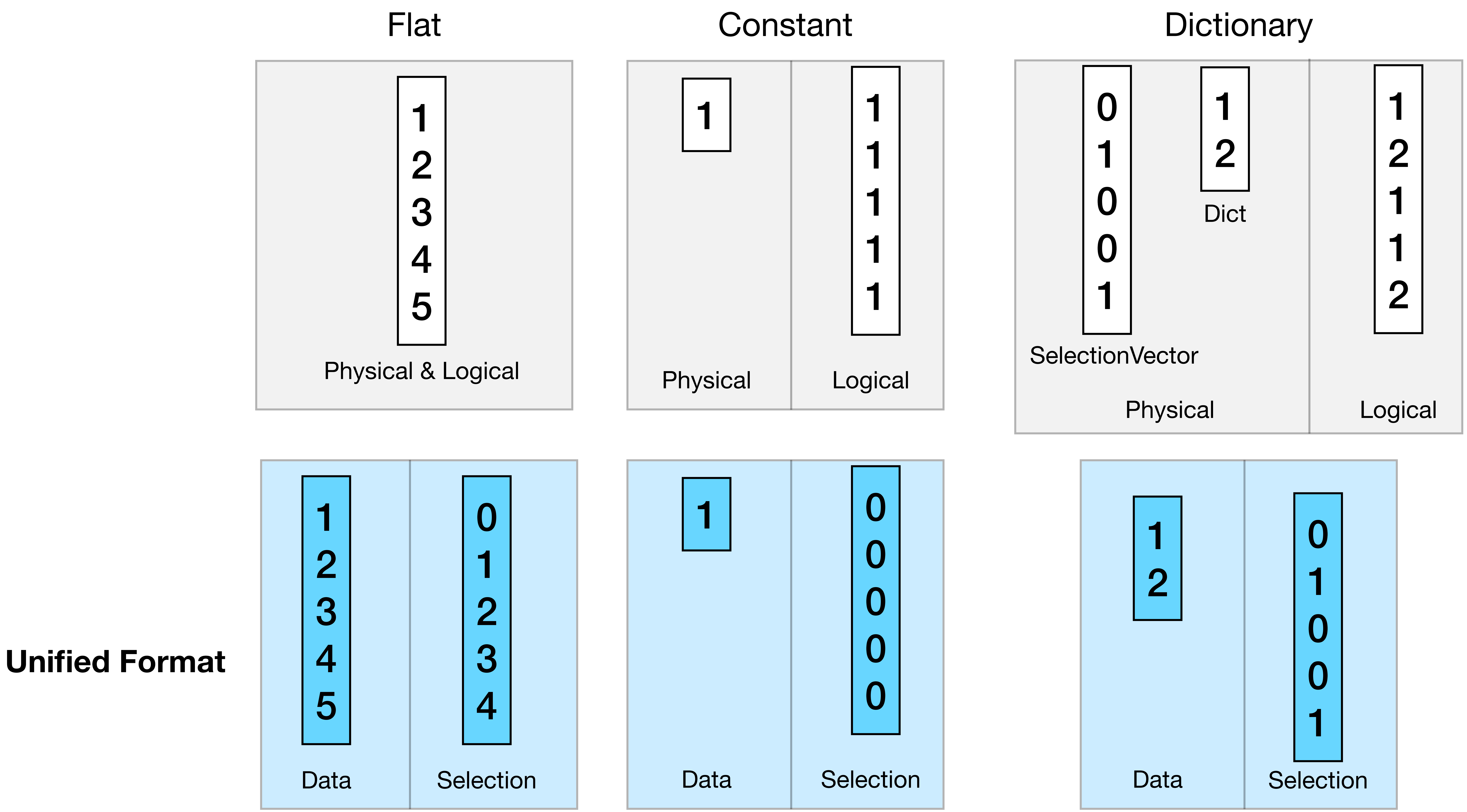
- Vectors can be processed as-is (compressed execution)
  - **Problem:** combinatorial explosion!
  - Giant code footprint

Flat
Uncompressed array



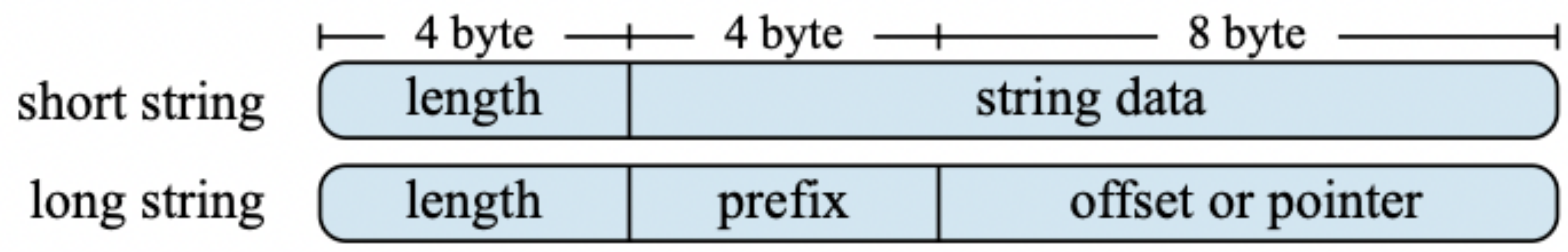
- **Flatten** - Convert vector into Flat Vector (i.e. decompress)
  - Need to move/copy data around!
- **ToUnified** - Convert vector to **unified format**

DuckDB - Vectors



No data copy or data movement required!

DuckDB - Vectors



- Strings are stored in the same format as Umbra
  - 16 bytes
  - Short strings are inlined (≤ 12 bytes)
  - Long strings have a prefix + pointer
- **Fast early-out in comparison**


Umbra: A Disk-Based System with In-Memory Performance

Thomas Neumann, Michael Freitag

[1] Umbra: A Disk-Based System with In-Memory Performance

DuckDB - Vectors

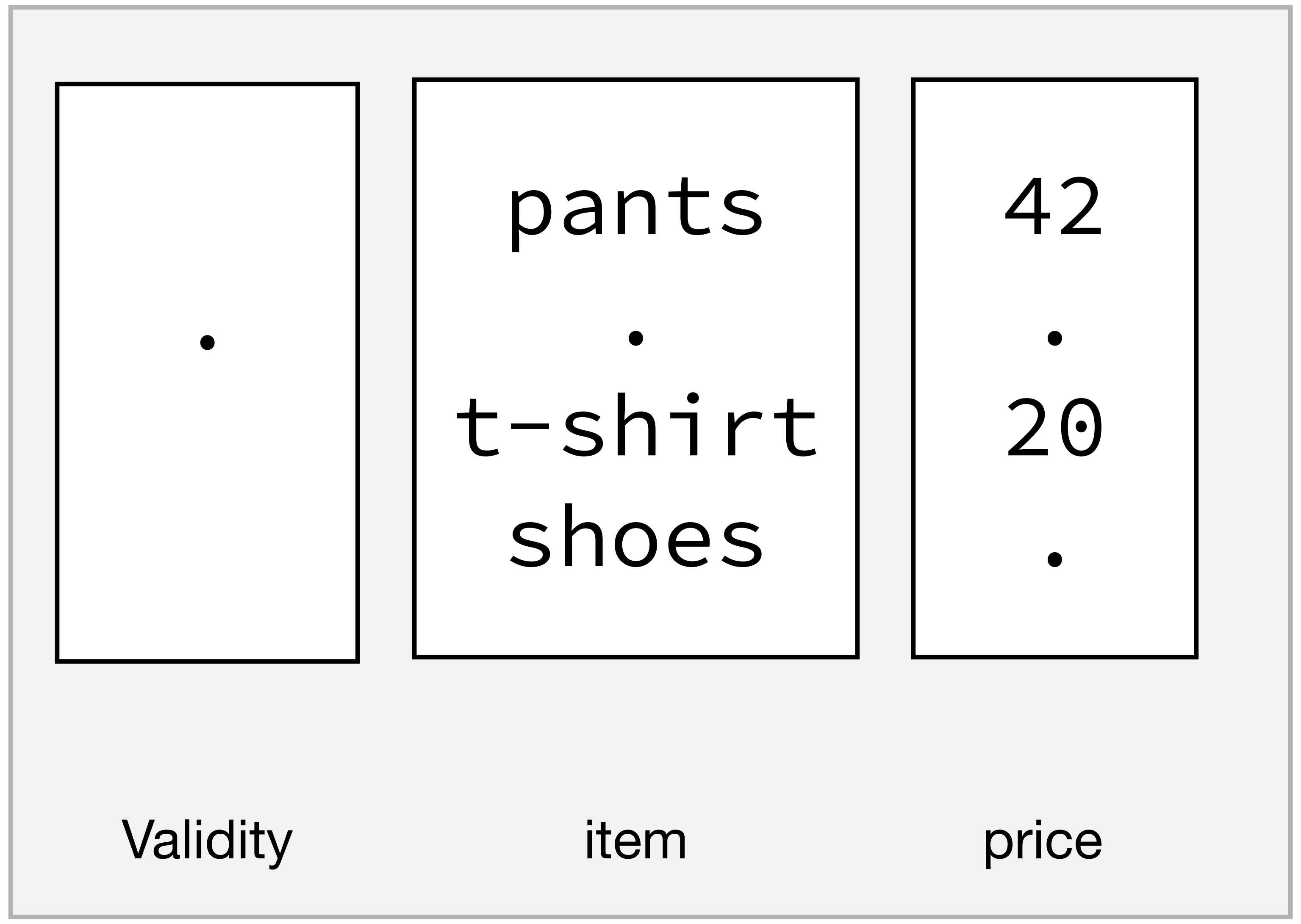
- Nested types - important for analytics
  - Possible solution: store as blobs or strings
  - **Slow!**
- **Nested types** are stored **recursively using vectors**
  - Allows for highly efficient processing
- Two main nested types: **structs** and **lists**

```
struct
{'item': pants, 'price': 42}
NULL
{'item': t-shirt, 'price': 20}
{'item': shoes, 'price': NULL}
```

```
list
[1, 2, 3]
[]
[4, NULL, 6, 7, 8]
NULL
```

DuckDB - Vectors

- Structs store their **child vectors** and a validity mask

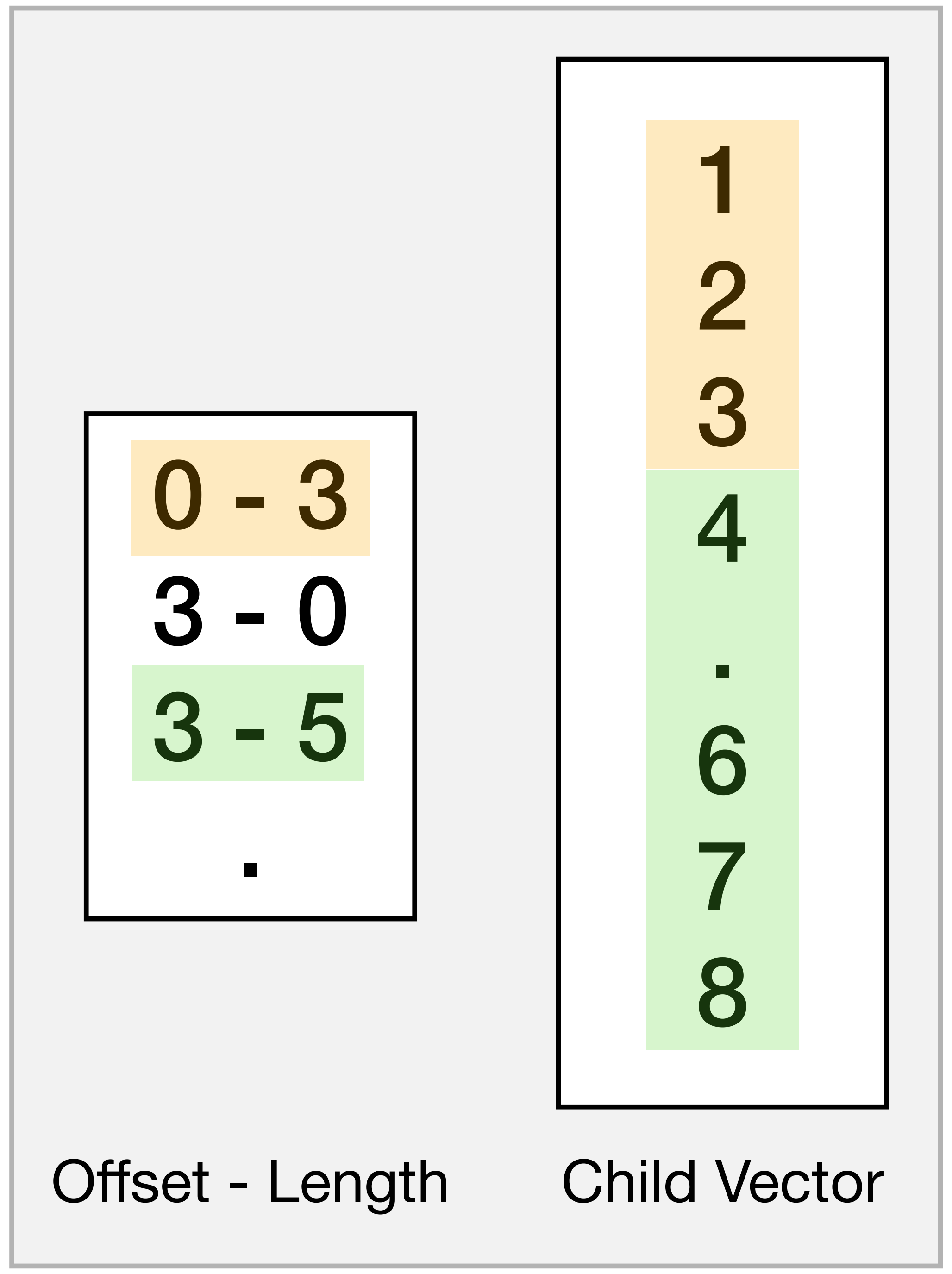



```
struct
{'item': pants, 'price': 42}
NULL
{'item': t-shirt, 'price': 20}
{'item': shoes, 'price': NULL}
```

DuckDB - Vectors

- Lists are stored as combination of **offset/lengths** and a **child vector**
  - The child vector can have a different length!

```
struct list_entry_t {
    uint64_t offset;
    uint64_t length;
};
```




```
l
[1, 2, 3]
[]
[4, NULL, 6, 7, 8]
NULL
```

Query Execution

DuckDB - Pull-Based Model

- DuckDB started as a **pull-based system**
  - “Vector Volcano”
- Every operator implements `GetChunk`
- Query starts by calling `GetChunk` on the root
- Nodes recursively call `GetChunk` on children

DuckDB - Pull-Based Model

- Simplified Hash Join Example

```
void HashJoin::GetChunk(DataChunk &result) {
    if (!build_finished) {
        // build the hash table
        while(right_child->GetChunk(child_chunk)) {
            BuildHashTable(child_chunk);
        }
        build_finished = true;
    }
    // probe the hash table
    left_child->GetChunk(child_chunk);
    ProbeHashTable(child_chunk, result);
}
```

DuckDB - Pull-Based Model

- In this model:
  - Single-threaded execution is straightforward
  - Multi-threaded not so much...
- In `CURRENT_TIMESTAMP`, multi-threaded execution is **required!**

DuckDB - Parallelism Model

- AWS instances go up to **192 cores**
  - Multi-threading = potential two-orders of magnitude speed-up!

Instance Size	vCPU	Memory (GiB)
c6a.large	2	4
c6a.xlarge	4	8
c6a.2xlarge	8	16
c6a.4xlarge	16	32
c6a.8xlarge	32	64
c6a.12xlarge	48	96
c6a.16xlarge	64	128
c6a.24xlarge	96	192
c6a.32xlarge	128	256
c6a.48xlarge	192	384
c6a.metal	192	384

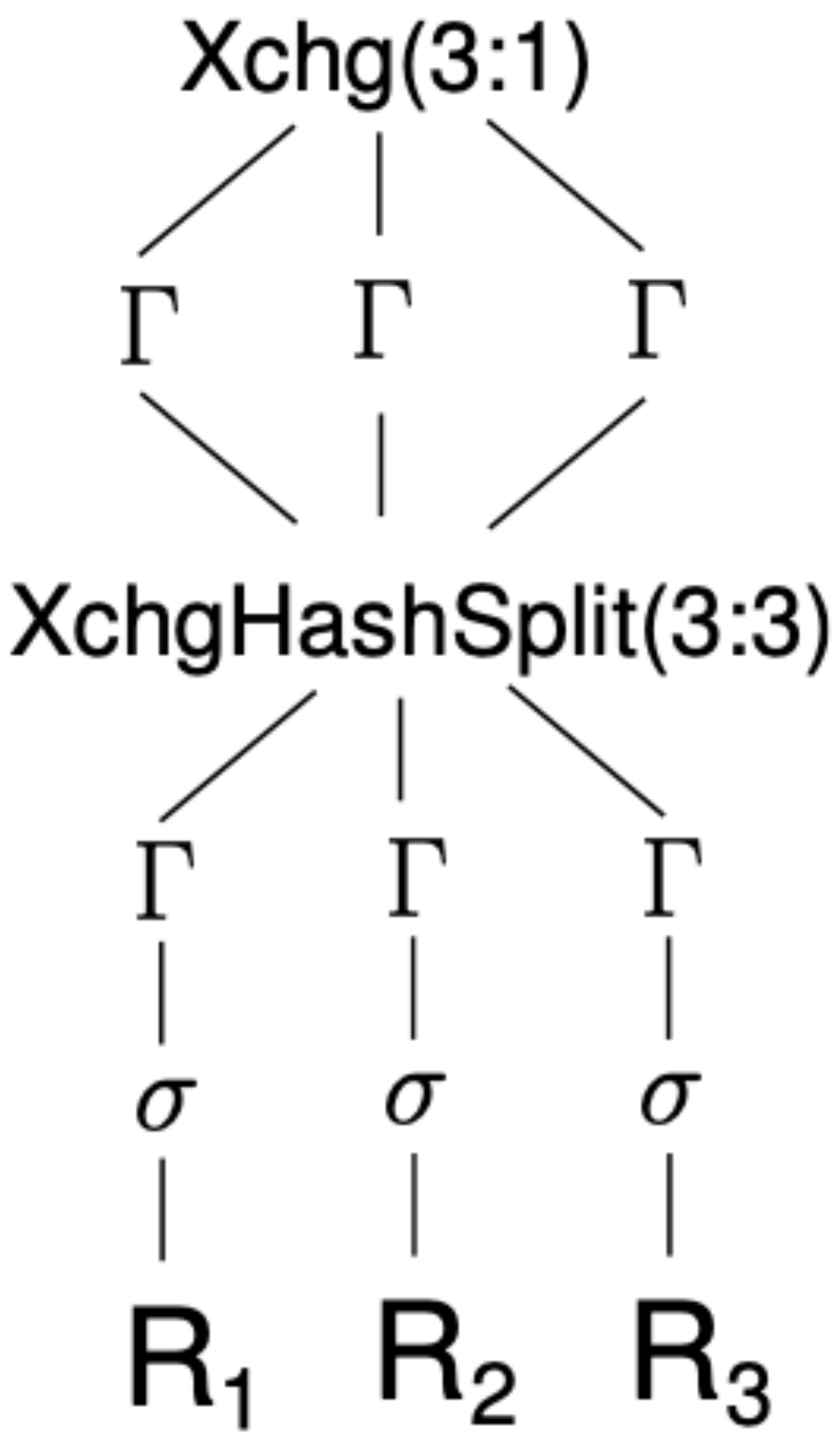
DuckDB - Parallelism Model

192 cores

Multi-Threaded	Single-Threaded
1 sec	> 3 minutes
1 min	> 3 hours

DuckDB - Parallelism Model

- Exchange operator
  - Optimizer splits plan into multiple partitions
  - Partitions are executed independently
- Operators do not need to be parallelism aware!

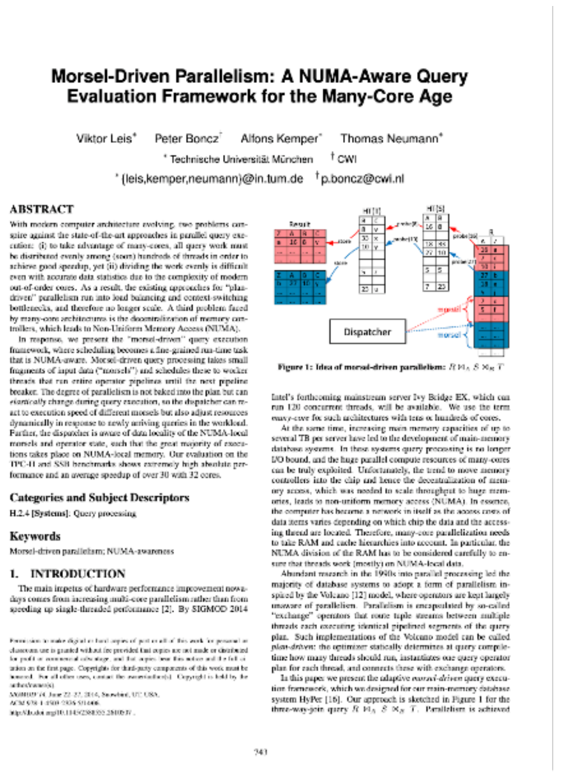


DuckDB - Parallelism Model

- Great for bolting parallelism onto a single-threaded system
  - But has many problems!
- Plan explosion
- Load imbalance issues
- Added materialization costs

DuckDB - Parallelism Model

- Morsel driven parallelism
- Individual operators are parallelism-aware
- Input data is distributed **adaptively**
    - Parallelism is not baked into the plan


Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis* Peter Boncz† Alfons Kemper* Thomas Neumann*

* Technische Universität München † CWI

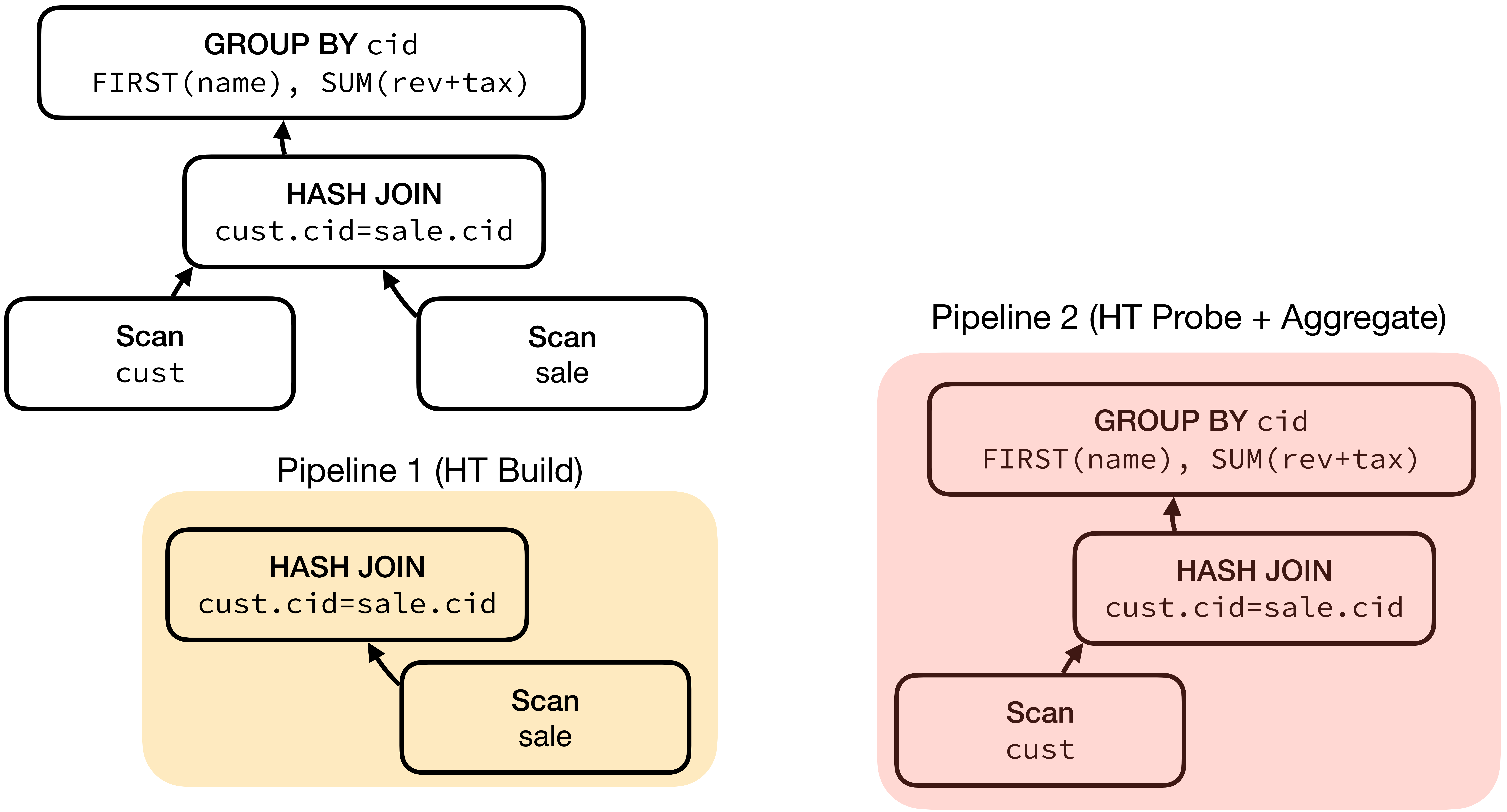
* {leis,kemper,neumann}@in.tum.de † p.boncz@cwi.nl

ABSTRACT

[2014] Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis et al.

DuckDB - Pipelines



DuckDB - Pull-Based Model

- How do we implement this in a **pull-based volcano model?**
- **Everything is entangled!**

```
void HashJoin::GetChunk(DataChunk &result) {
    if (!build_finished) {
        // build the hash table
        while(right_child->GetChunk(child_chunk)) {
            BuildHashTable(child_chunk);
        }
        build_finished = true;
    }
    // probe the hash table
    left_child->GetChunk(child_chunk);
    ProbeHashTable(child_chunk, result);
}
```

DuckDB - Push-Based Execution

- Switch to **push-based model**
- Separate interfaces for **sink, source** and **operator**
  - **Source** and **sink** are **parallelism aware!**

Source

```
void GetData(
    DataChunk &chunk,
    GlobalSourceState &gstate,
    LocalSourceState &lstate);
```

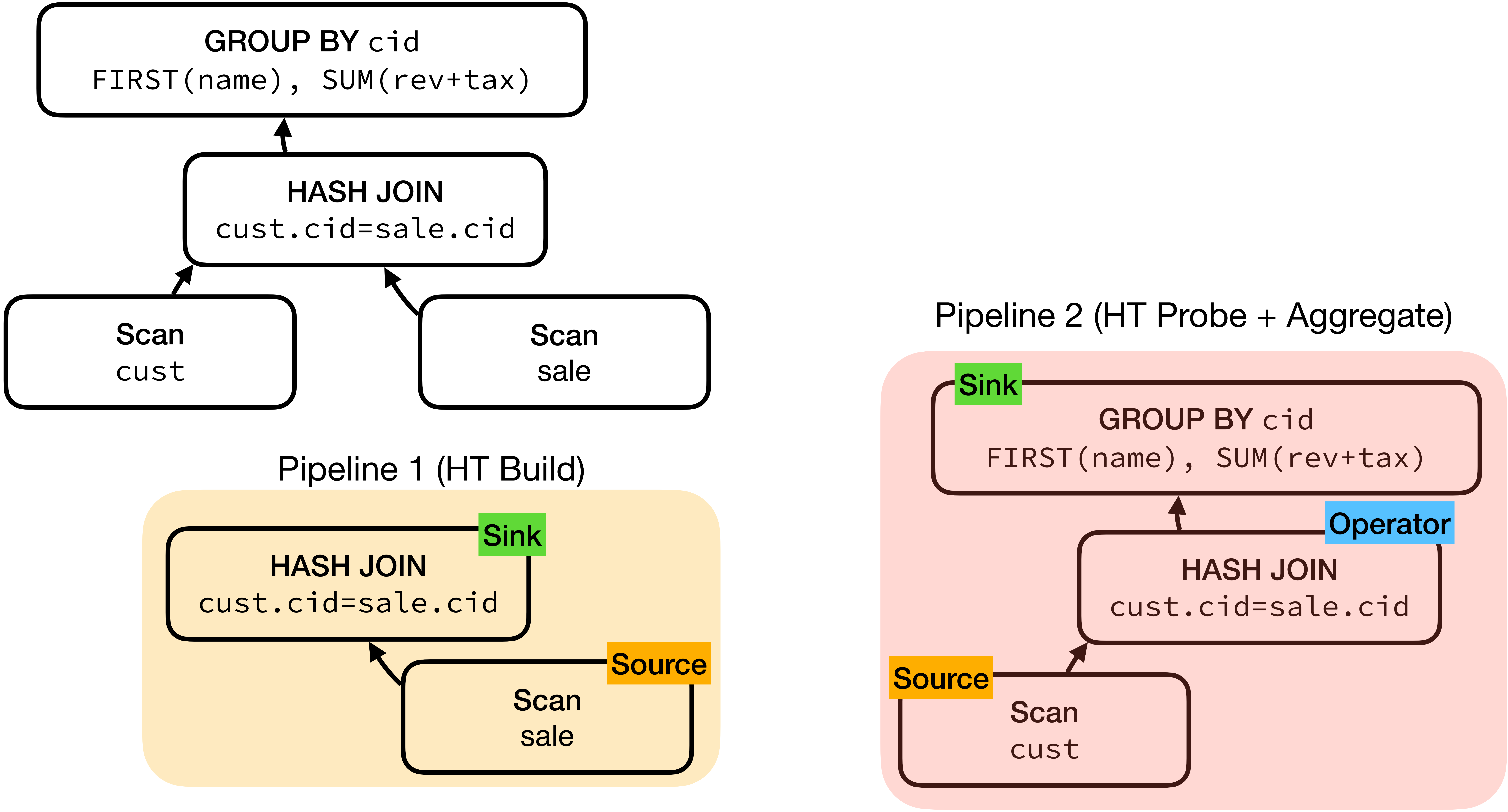
Operator

```
OperatorResultType Execute(
    DataChunk &input,
    DataChunk &chunk,
    OperatorState &state);
```

Sink

```
void Sink(
    GlobalSinkState &gstate,
    LocalSinkState &lstate,
    DataChunk &input);
void Combine(
    GlobalSinkState &gstate,
    LocalSinkState &lstate);
void Finalize(
    GlobalSinkState &gstate);
```

DuckDB - Pipelines



DuckDB - Push-Based Execution

- Pull-based: the control flow lives **insides** the operator
  - Very flexible!
- Built on recursive calls - the call stack holds all state

Volcano

```
void Projection::GetChunk(DataChunk &result) {
    // get the next chunk from the child
    child->GetChunk(child_chunk);
    if (child_chunk.size() == 0) {
        return;
    }

    // execute expressions
    executor.Execute(child_chunk, result);
}
```

```
HashAggregate::GetChunk(DataChunk &result)
  HashJoin::GetChunk(DataChunk &result)
    Projection::GetChunk(DataChunk &result)
      Table::GetChunk(DataChunk &result)
```

DuckDB - Push-Based Execution

- Push-based: control flow happens **in a central location**
    - This has a number of advantages

Push-Based

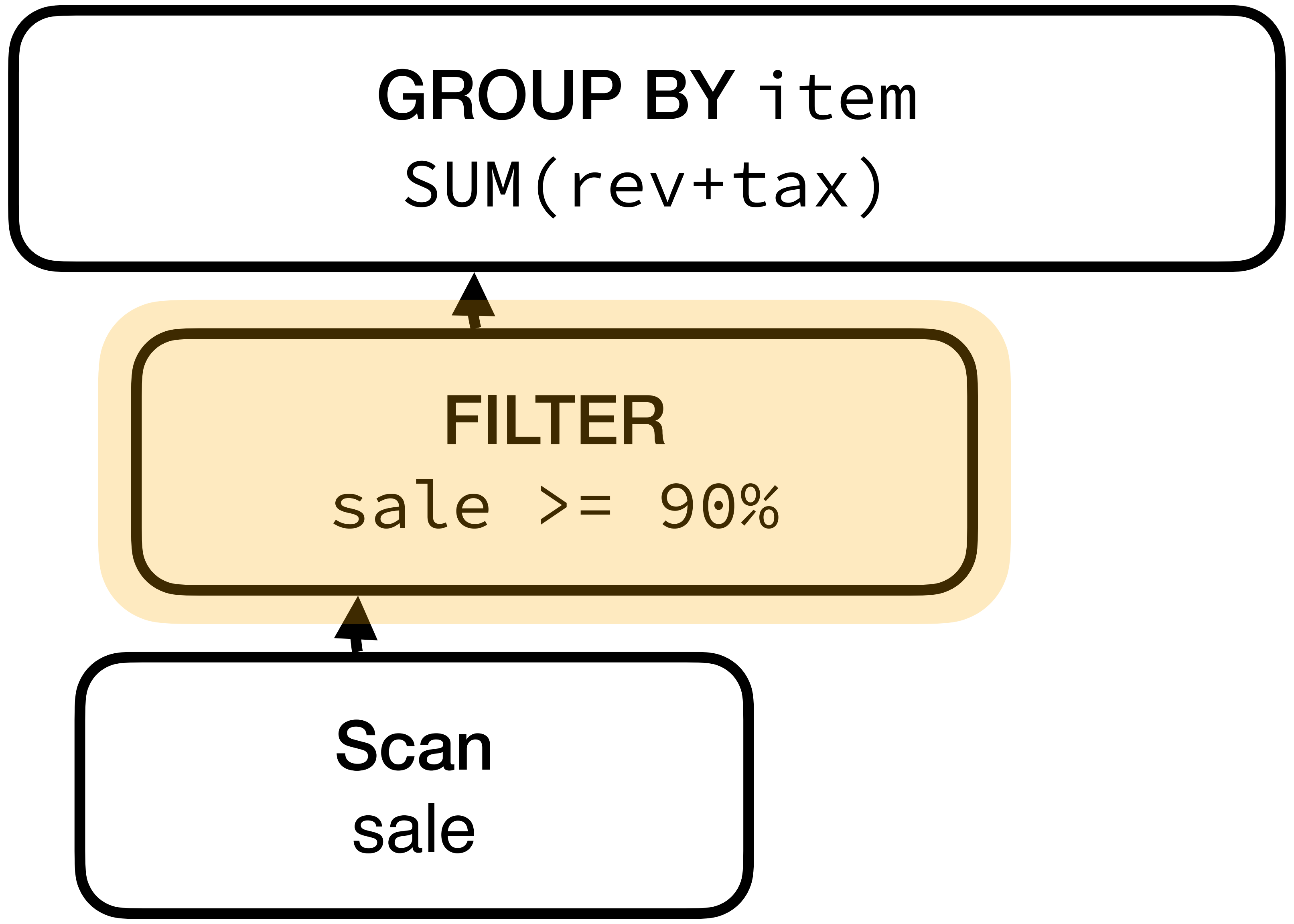
```
void Projection::Execute(DataChunk &input, DataChunk &result) {
    executor.Execute(input, result);
}
```

```
class PipelineState {
public:
   //! Intermediate chunks for the operators
   vector<unique_ptr<DataChunk>> intermediate_chunks;
}
```

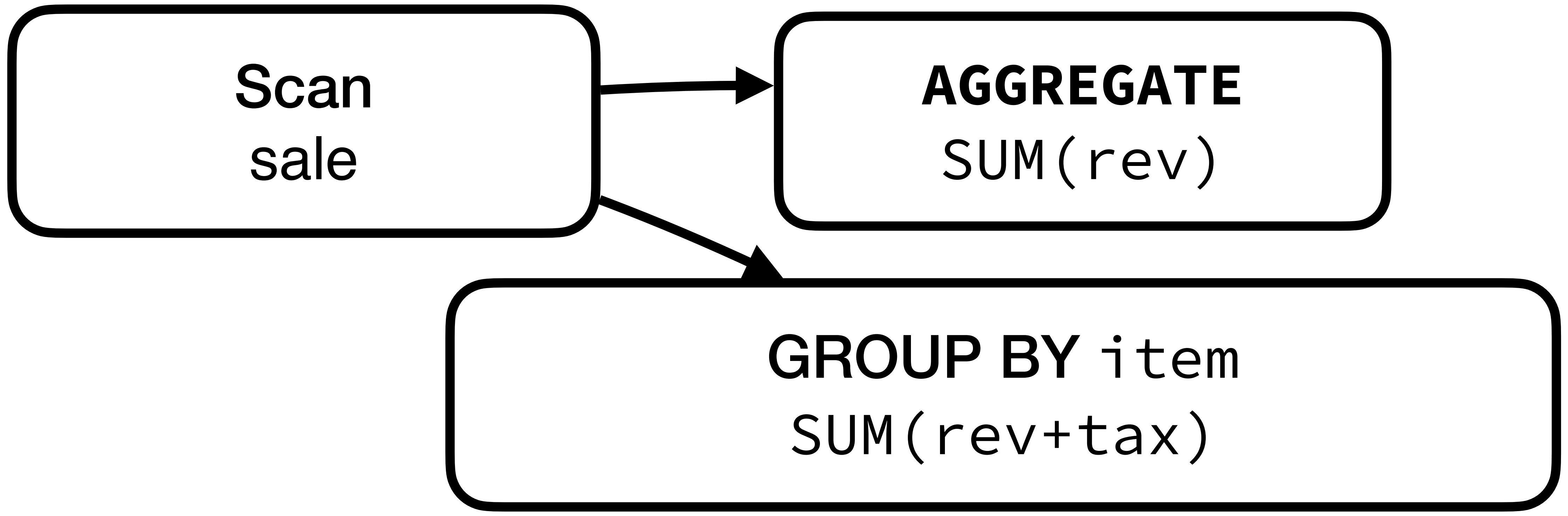
DuckDB - Push-Based Execution

- Handling control flow in a central location enables optimizations

Vector Cache
Add **small caches** between operators



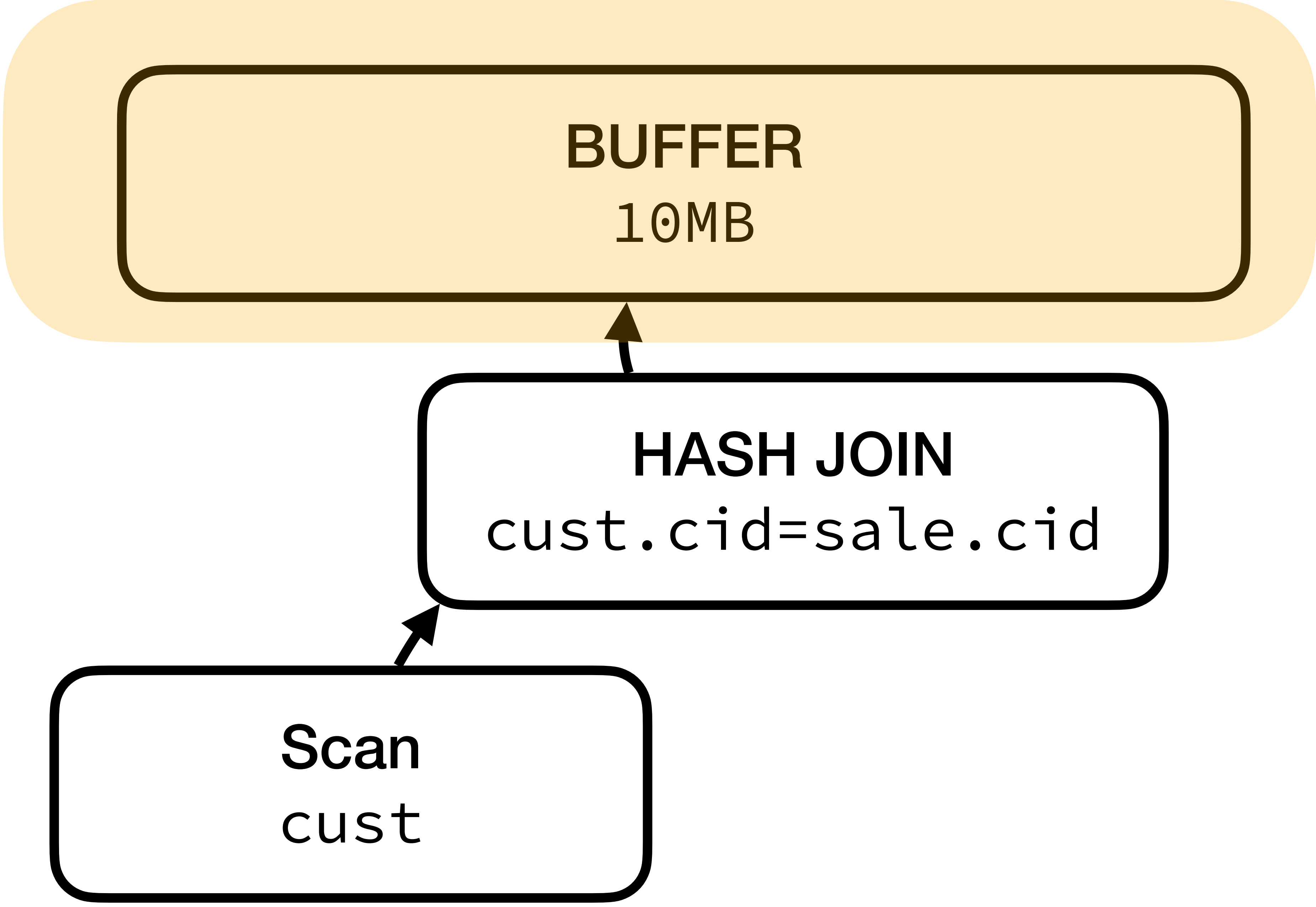
Scan Sharing
We can push results of one scan
into multiple sinks



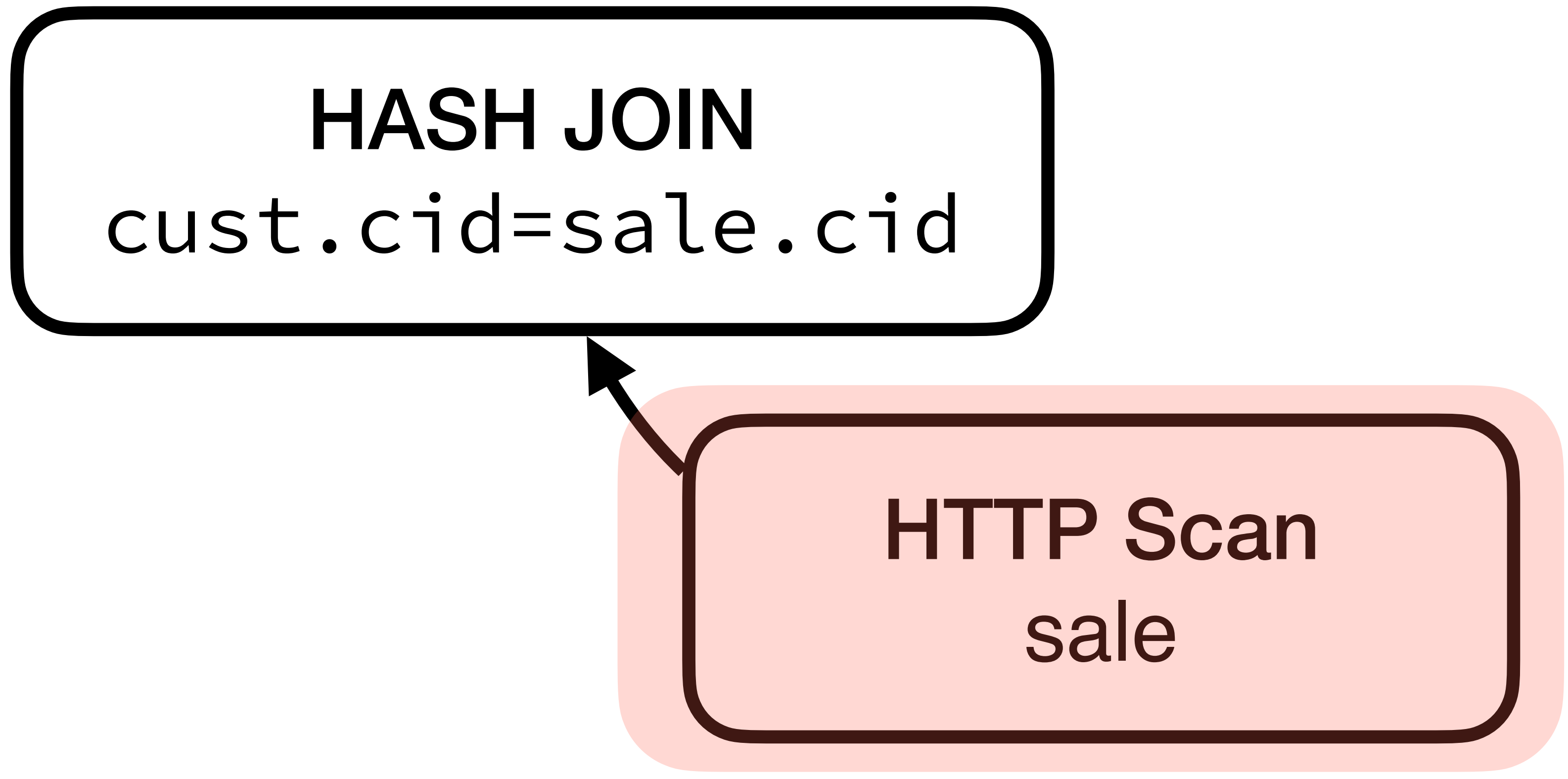
DuckDB - Push-Based Execution

- Storing state in a central location allows us to **pause execution**

Backpressure
Pause pipeline when buffer is full
Resume when buffer is empty



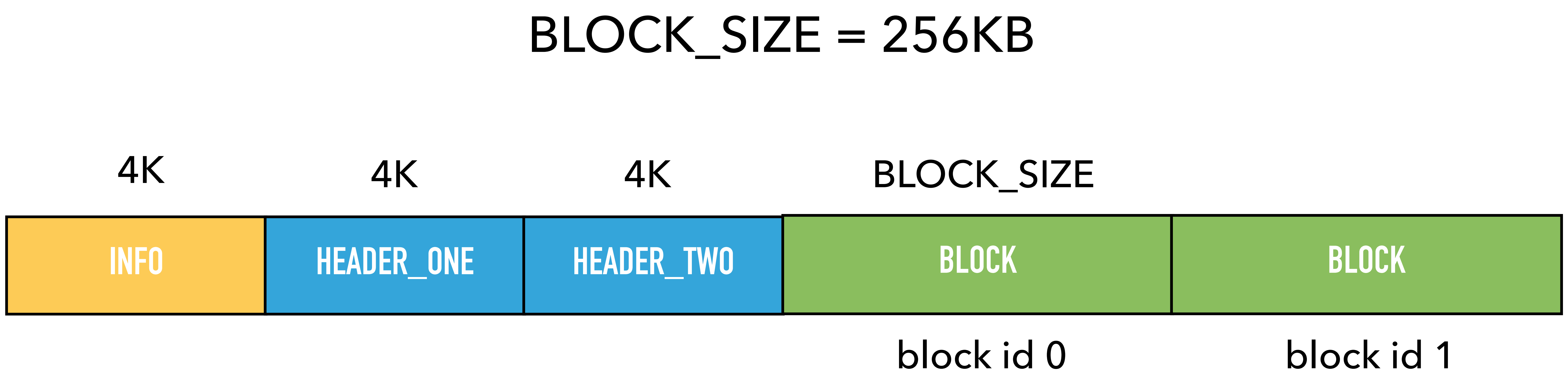
Async I/O
Pause pipeline while HTTP request is in progress
Resume when data is available



Storage

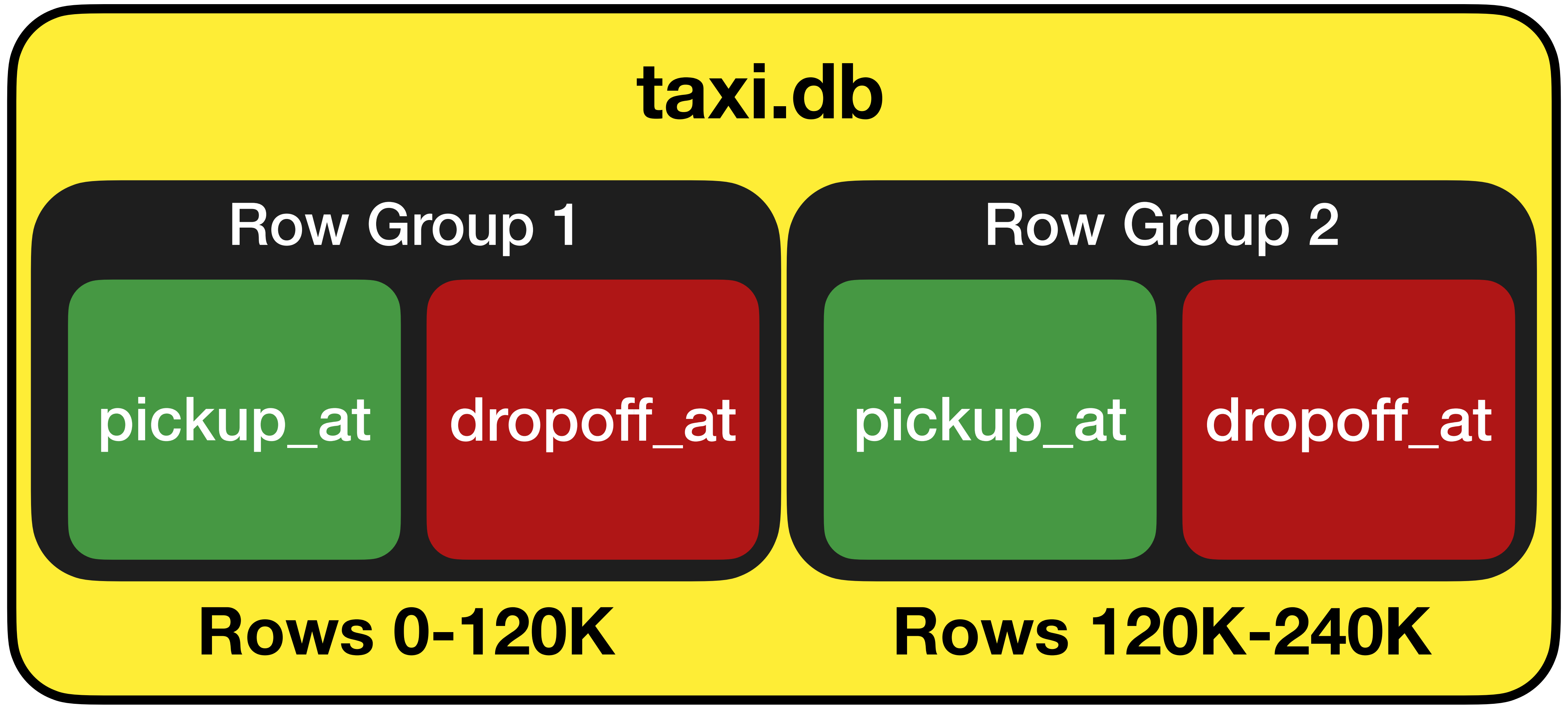
DuckDB - Table Storage

- DuckDB uses a **single-file block-based** storage format
- WAL is stored as a separate file
- Support for ACID using **headers**



DuckDB - Table Storage

- Tables are partitioned into **row groups**
  - Each row group has **120K~ rows**
- Row groups are the parallelism and checkpoint unit



DuckDB - Table Storage

- Compression works **very well** with columnar storage
  - Speeds up I/O
  - Can speed up execution (see vectors!)
- Compression can make data smaller **and** queries faster

DuckDB - Table Storage

- General-purpose, heavy-weight compression
  - `gzip, zstd, snappy, lz4`
  - Finds patterns in **bits**

- Special purpose, lightweight compression
  - RLE, bitpacking, dictionary, FOR, delta, ...
  - Finds specific patterns in **data**

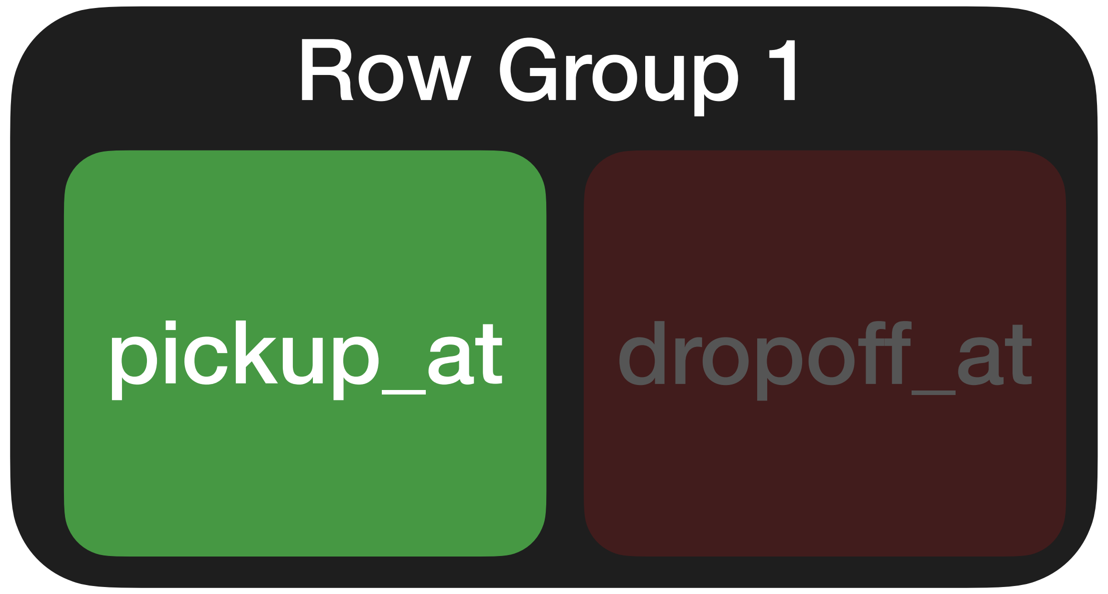
DuckDB - Table Storage

- General-purpose compression is simple to apply
- Works great for **space saving!**
- However...
  - Higher (de)compression speed slows down execution
  - Need to decompress in bulk - no random seeks or compressed execution!

DuckDB - Table Storage

- Lightweight compression detects **specific patterns**
  - Very fast!
  - Patterns can be exploited during execution
- **Downside:** No effect if the patterns are not there!
- We need to implement/use **multiple different algorithms**

DuckDB - Table Storage



- Compression works per-column per row-group
- **Two phases:**
- **Analyze**
  - Figure out which compression method is best
- **Compress**
  - Use the best compression method to compress the column

DuckDB - Table Storage

Version	Taxi	On Time	Lineitem	Notes	Date
DuckDB v0.2.8	15.3GB	1.73GB	0.85GB	Uncompressed	July 2021
DuckDB v0.2.9	11.2GB	1.25GB	0.79GB	RLE + Constant	September 2021
DuckDB v0.3.2	10.8GB	0.98GB	0.56GB	Bitpacking	February 2022
DuckDB v0.3.3	6.9GB	0.23GB	0.32GB	Dictionary	April 2022
DuckDB v0.5.0	6.6GB	0.21GB	0.29GB	FOR	September 2022
DuckDB dev	4.8GB	0.21GB	0.17GB	FSST + Chimp	November 2022
CSV	17.0GB	1.11GB	0.72GB		
Parquet (Uncompressed)	4.5GB	0.12GB	0.31GB		
Parquet (Snappy)	3.2GB	0.11GB	0.18GB		
Parquet (ZSTD)	2.6GB	0.08GB	0.15GB		

duckdb.org/2022/10/28/lightweight-compression.html

Lightning Round

DuckDB - Buffer Manager

- Custom **lock-free** buffer manager
  - Inspired by lean-store
- Granularity: 256KB blocks

- Traditional buffer manager functionality:
  - Set memory limit
  - **Pin blocks** to fix them in memory
  - **Unpin blocks** to tell the system it is alright to release them

DuckDB - Out-Of-Core

- DuckDB supports **larger-than-memory execution**
  - Streaming engine
  - Special join, sort & window algorithms

- **Goal:** Gracefully degrade performance
  - Avoid performance cliff!

Out-Of-Core Hash Join

Memory limit (GB)	Time (s)
10	1.96
9	1.97
8	2.22
7	2.44
6	2.39
5	2.32
4	2.45
3	3.20
2	3.28
1	4.35

DuckDB - Transactions

- DuckDB supports ACID transactions
- Based on HyPer **MVCC** model
  - Optimized for vectorized processing
- DuckDB supports **snapshot isolation**

- **Optimistic concurrency control**
  - Changes to the same rows → transaction abort

Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems

Thomas Neumann · Tobias Mühlbauer · Alfons Kemper

Technische Universität München
{neumann, muehlbau, kemper}@in.tum.de

ABSTRACT

Multi-Version Concurrency Control (MVCC) is a widely employed concurrency control mechanism, as it allows for execution modes where readers never block writers. However, most systems implement only snapshot isolation (SI) instead of full serializability. Adding serializability guarantees to existing SI implementations tends to be *prohibitively expensive*.

We present a novel MVCC implementation for main-memory database systems that has very little overhead compared to serial execution with single-version concurrency control, even when maintaining serializability guarantees. Updating data in-place and storing versions as before-image deltas in undo buffers not only allows us to retain the high scan performance of single-version systems but also forms the basis of our cheap and fine-grained serializability validation mechanism. The novel idea is based on an adaptation of precision locking and verifies that the (extensional) writes of recently committed transactions do not intersect with the (intensional) read predicate space of a committing transaction. We experimentally show that our MVCC model allows very fast processing of transactions with point accesses *as well as* read-heavy transactions and that there is little need to prefer SI over full serializability any longer.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Multi-Version Concurrency Control; MVCC; Serializability

1. INTRODUCTION

Transaction isolation is one of the most fundamental features offered by a database management system (DBMS). It provides the user with the illusion of being alone in the database system, even in the presence of multiple concurrent users, which greatly simplifies application development. In the background, the DBMS ensures that the resulting concurrent access patterns are safe, ideally by being serializable.

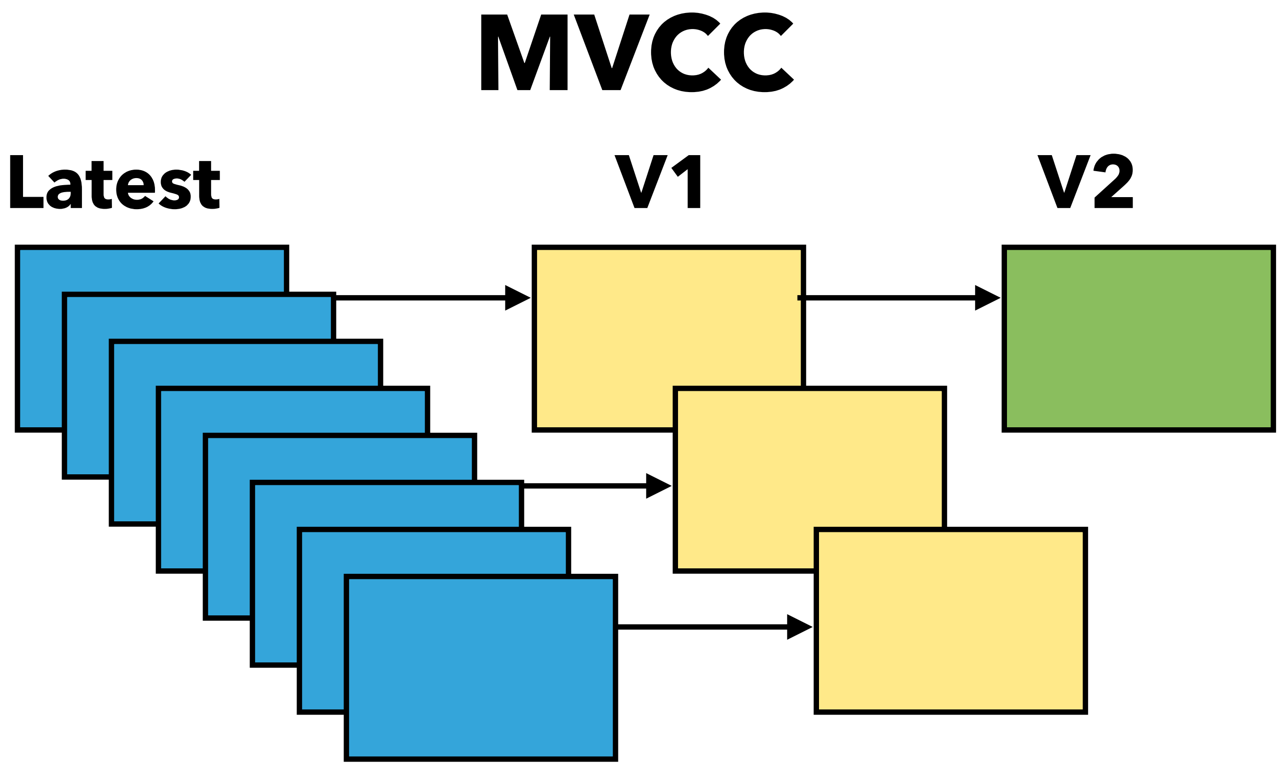
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749436.

Serializability is a great concept, but it is hard to implement efficiently. A classical way to ensure serializability is to rely on a variant of *Two-Phase Locking* (2PL) [42]. Using 2PL, the DBMS maintains read and write locks to ensure that conflicting transactions are executed in a well-defined order, which results in serializable execution schedules. Locking, however, has several major disadvantages: First, readers and writers block each other. Second, most transactions are read-only [33] and therefore harmless from a transaction-ordering perspective. Using a locking-based isolation mechanism, no update transaction is allowed to change a data object that has been read by a potentially long-running read transaction and thus has to wait until the read transaction finishes. This severely limits the degree of concurrency in the system.

Multi-Version Concurrency Control (MVCC) [42, 3, 28] offers an elegant solution to this problem. Instead of updating data objects in-place, each update creates a new version of that data object, such that concurrent readers can still see the old version while the update transaction proceeds concurrently. As a consequence, read-only transactions never have to wait, and in fact do not have to use locking at all. This is an extremely desirable property and the reason why many DBMSs implement MVCC, e.g., Oracle, Microsoft SQL Server [8, 23], SAP HANA [10, 37], and PostgreSQL [34]. However, most systems that use MVCC do not guarantee serializability, but the weaker isolation level *Snapshot Isolation* (SI). Under SI, every transaction sees the database in a certain state (typically the last committed state at the beginning of the transaction) and the DBMS ensures that two concurrent transactions do not update the same data object. Although SI offers fairly good isolation, some non-serializable schedules are still allowed [1, 2]. This is often reluctantly accepted because making SI serializable tends to be *prohibitively expensive* [7]. In particular, the known solutions require keeping track of the entire read set of every transaction, which creates a huge overhead for read-heavy (e.g., analytical) workloads. Still, it is desirable to detect serializability conflicts as they can lead to silent data corruption, which in turn can cause hard-to-detect bugs.

In this paper we introduce a novel way to implement MVCC that is very fast and efficient, both for SI and for full serializability. Our SI implementation is admittedly more carefully engineered than totally new, as MVCC is a well understood approach that recently received renewed interest in the context of main-memory DBMSs [23]. Careful engineering, however, matters as the performance of version maintenance greatly affects transaction *and* query processing. It



DuckDB - External Formats

- DuckDB supports **querying directly over many formats**
  - Parquet, CSV, JSON, Arrow, Pandas, SQLite, Postgres, …

```
$ duckdb
D FROM lineitem.parquet;
┌────────────┬───────────┬───────────┬───┬───────────────────┬────────────┬──────────────────────┐
│ l_orderkey │ l_partkey │ l_suppkey │ … │  l_shipinstruct   │ l_shipmode │      l_comment       │
│   int32    │   int32   │   int32   │   │      varchar      │  varchar   │       varchar        │
├────────────┼───────────┼───────────┼───┼───────────────────┼────────────┼──────────────────────┤
│          1 │    155190 │      7706 │ … │ DELIVER IN PERSON │ TRUCK      │ to beans x-ray car…  │
│          1 │     67310 │      7311 │ … │ TAKE BACK RETURN  │ MAIL       │  according to the …  │
│          1 │     63700 │      3701 │ … │ TAKE BACK RETURN  │ REG AIR    │ ourts cajole above…  │
│          1 │      2132 │      4633 │ … │ NONE              │ AIR        │ s cajole busily ab…  │
│          1 │     24027 │      1534 │ … │ NONE              │ FOB        │  the regular, regu…  │
│          · │         · │         · │ · │   ·               │  ·         │          ·           │
│          · │         · │         · │ · │   ·               │  ·         │          ·           │
│          · │         · │         · │ · │   ·               │  ·         │          ·           │
│    5999975 │      7272 │      2273 │ … │ COLLECT COD       │ REG AIR    │ ld deposits aga      │
│    5999975 │      6452 │      1453 │ … │ DELIVER IN PERSON │ SHIP       │ ffily along the sly  │
│    5999975 │     37131 │      2138 │ … │ DELIVER IN PERSON │ FOB        │ counts cajole even…  │
│    6000000 │     32255 │      2256 │ … │ TAKE BACK RETURN  │ MAIL       │ riously pe           │
│    6000000 │     96127 │      6128 │ … │ NONE              │ AIR        │ pecial excuses nag…  │
├────────────┴───────────┴───────────┴───┴───────────────────┴────────────┴──────────────────────┤
│ 6001215 rows (10 shown)                                                  16 columns (6 shown) │
└───────────────────────────────────────────────────────────────────────────────────────────────┘
```

DuckDB - Pluggable Catalog

- DuckDB supports **attaching multiple databases** and has a **fully pluggable catalog**

```
$ duckdb
D ATTACH 'sqlite.db' (TYPE sqlite);
D SELECT database_name, path, type FROM duckdb_databases();
┌───────────────┬───────────┬─────────┐
│ database_name │   path    │  type   │
│    varchar    │  varchar  │ varchar │
├───────────────┼───────────┼─────────┤
│ sqlite        │ sqlite.db │ sqlite  │
│ memory        │ NULL      │ duckdb  │
└───────────────┴───────────┴─────────┘
D USE sqlite;
D CREATE TABLE lineitem AS FROM 'lineitem.parquet';
D CREATE VIEW lineitem_subset AS
  SELECT l_orderkey, l_partkey, l_suppkey, l_comment FROM lineitem;

$ sqlite3 sqlite.db
sqlite> SELECT * FROM lineitem_subset LIMIT 3;
┌────────────┬───────────┬───────────┬──────────────────────────────────────┐
│ l_orderkey │ l_partkey │ l_suppkey │              l_comment               │
├────────────┼───────────┼───────────┼──────────────────────────────────────┤
│ 1          │ 155190    │ 7706      │ to beans x-ray carefull              │
│ 1          │ 67310     │ 7311      │  according to the final foxes. qui   │
│ 1          │ 63700     │ 3701      │ ourts cajole above the furiou        │
└────────────┴───────────┴───────────┴──────────────────────────────────────┘
```

DuckDB - Pluggable File System + HTTP/Object Store Reads

- DuckDB has a **pluggable file system**
  - Used for querying over HTTP/S3/object stores

```
$ duckdb
D LOAD httpfs;
D FROM 'https://github.com/duckdb/duckdb-data/releases/download/v1.0/yellowcab.parquet';
┌──────────┬─────────────────────┬─────────────────────┬───┬──────────────┬────────────────────┬──────────────┐
│ VendorID │ tpep_pickup_datetime│ tpep_dropoff_datet… │ … │ tolls_amount │ improvement_surcha…│ total_amount │
│  int32   │       varchar       │       varchar       │   │   varchar    │      varchar       │   varchar    │
├──────────┼─────────────────────┼─────────────────────┼───┼──────────────┼────────────────────┼──────────────┤
│        2 │ 2016-01-01 00:00:00 │ 2016-01-01 00:00:00 │ … │ 0            │ 0.3                │ 8.8          │
│        2 │ 2016-01-01 00:00:00 │ 2016-01-01 00:00:00 │ … │ 0            │ 0.3                │ 19.3         │
│        2 │ 2016-01-01 00:00:00 │ 2016-01-01 00:00:00 │ … │ 0            │ 0.3                │ 34.3         │
│        2 │ 2016-01-01 00:00:00 │ 2016-01-01 00:00:00 │ … │ 0            │ 0.3                │ 17.3         │
│        2 │ 2016-01-01 00:00:00 │ 2016-01-01 00:00:00 │ … │ 0            │ 0.3                │ 8.8          │
│        · │          ·          │          ·          │ · │ ·            │  ·                 │  ·           │
│        · │          ·          │          ·          │ · │ ·            │  ·                 │  ·           │
│        · │          ·          │          ·          │ · │ ·            │  ·                 │  ·           │
│        1 │ 2016-01-01 13:03:57 │ 2016-01-01 13:10:39 │ … │ 0            │ 0.3                │ 7.8          │
│        2 │ 2016-01-01 13:03:57 │ 2016-01-01 13:15:13 │ … │ 0            │ 0.3                │ 13.33        │
│        1 │ 2016-01-01 13:03:58 │ 2016-01-01 13:39:53 │ … │ 10.5         │ 0.3                │ 98.8         │
│        2 │ 2016-01-01 13:03:58 │ 2016-01-01 13:07:47 │ … │ 0            │ 0.3                │ 6.36         │
│        2 │ 2016-01-01 13:03:58 │ 2016-01-01 13:21:01 │ … │ NULL         │ NULL               │ NULL         │
├──────────┴─────────────────────┴─────────────────────┴───┴──────────────┴────────────────────┴──────────────┤
│ 234118 rows (10 shown)                                                               19 columns (6 shown) │
└───────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

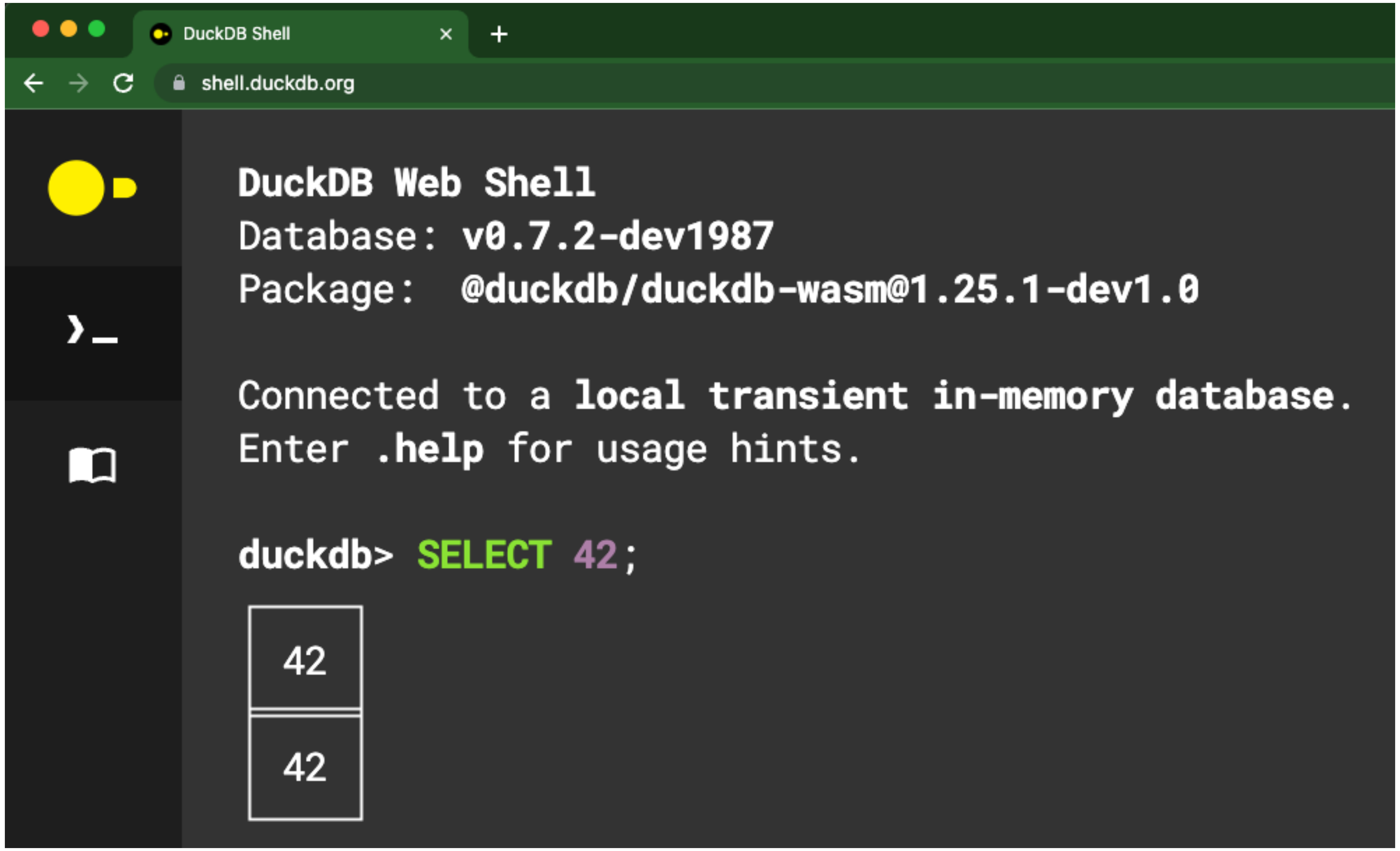
DuckDB - Extensions

- DuckDB offers support for **extensions**
- Distributed through **INSTALL** and **LOAD** commands
  - Can be loaded as a **shared library**
- Many of our core features are implemented as extensions

extension_name	loaded	installed	install_path	description
fts	false	false		Adds support for Full-Text Search Indexes
httpfs	false	false		Adds support for reading and writing files over a HTTP(S) connection
icu	true	true	(BUILT-IN)	Adds support for time zones and collations using the ICU library
json	false	false		Adds support for JSON operations
parquet	true	true	(BUILT-IN)	Adds support for reading and writing parquet files
postgres_scanner	false	false		Adds support for reading from a Postgres database
sqlite_scanner	false	false		Adds support for reading SQLite database files
substrait	false	false		Adds support for the Substrait integration
tpcds	false	false		Adds TPC-DS data generation and query support
tpch	true	true	(BUILT-IN)	Adds TPC-H data generation and query support

DuckDB - WASM

- DuckDB has a **WASM build**
- Runs inside the browser
  - And it is actually fast!



DuckDB - Conclusion

- DuckDB is free and open source
  - Contributions are welcome!
- We have a website - https://duckdb.org/
- Source code - https://github.com/duckdb/duckdb

Thanks for having me!

Any questions?

@duckdb

duckdb.org

discord.gg/tcvwpjfnZx