Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master's Thesis

Declarative Caching in MotherDuck

Author: Niclas Haderer (2766114)

1st supervisor:	Prof. Dr. Peter Boncz
daily supervisor:	Boaz Leskes (MotherDuck)
2nd reader:	Dr. Pedro Holanda

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

October 31, 2024

Abstract

Hybrid query execution enables parts of queries to be executed simultaneously on a client and a server database. For hybrid execution, having data locally is very beneficial, as this allows for more local processing, resulting in higher local resource utilization and reduced query latencies. We introduce a new concept called Accelerated Approximate Views, which enables developers to create a partial client-side query cache. A custom declarative SQL syntax is proposed to interact with Accelerated Views, enabling users to read, create, and modify them. This has been implemented in DuckDB and DuckDB Wasm within MotherDuck, a serverless cloud data warehouse. Accelerated Views serve as a base for query optimization techniques, starting with partial result reading and, in future works, among others, count estimations. To evaluate Accelerated Views accordingly, a user study has been conducted investigating the workloads executed in online SQL notebooks. Our experiments show that Accelerated Views reduce query latency compared to alternative approaches. However, they exhibit performance problems in Wasm, which can be attributed to the non-collaborative nature of the DuckDB Wasm scheduler.

Contents

List of Figures **v**

List of Tables **ix**

List of Listings **xi**

Acronyms **xiii**

1 Introduction **1**

1.1 Contributions 2

1.2 Outline 3

2 Background **5**

2.1 Relational Database Management Systems 5

2.1.1 Views and Materialized Views 5

2.2 Client Server Architecture 6

2.3 Relational Algebra 6

2.4 DuckDB 8

2.4.1 Architecture 8

2.4.2 Query lifecycle 11

2.4.3 DuckDB Internals 12

2.4.3.1 Views 12

2.4.3.2 Table Functions 12

2.4.3.3 Physical Operators 13

2.4.3.4 Query Pipelines and Scheduling 13

2.4.3.5 Result collectors 15

2.4.3.6 Streaming Queries 16

2.4.3.7 Pending Queries 16

2.5 Web Assembly 18

2.6 Hybrid Query Processing . . . 20
2.6.1 Wasm specific adaptions . . . 20
2.7 Inner workings of the MotherDuck UI . . . 23

3 Literature Review **27**
3.1 Usage Patterns . . . 27
3.1.1 Workload sizes . . . 28
3.1.1.1 Data size . . . 28
3.1.1.2 Execution times . . . 31
3.1.2 Data types . . . 32
3.1.3 Query Patterns . . . 33
3.1.4 Caching implications . . . 35
3.2 Database caching . . . 36
3.2.1 Query Recycling . . . 37
3.2.1.1 Don't Trash your Intermediate Results, Cache 'em . . . 37
3.2.1.2 Cache-On-Demand: Recycling with Certainty . . . 38
3.2.1.3 Recycling in pipelined query evaluation . . . 38
3.2.1.4 Caching & Reuse of Subresults across Queries . . . 39
3.2.2 Caching in Hybrid Query Shipping systems . . . 40
3.2.2.1 Performance tradeoffs for client-server query processing . . . 40
3.2.2.2 Caching Multidimensional Queries Using Chunks . . . 41
3.2.2.3 Cache Tables: Paving the Way for an Adaptive Database Cache . . . 42
3.2.2.4 FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS . . . 43
3.2.2.5 MotherDuck: DuckDB in the cloud and in the client . . . 44
3.2.3 Data caching . . . 44
3.2.3.1 Building an elastic query engine on disaggregated storage . . . 44
3.2.4 Query Caching . . . 45
3.2.4.1 Extending a Database System with Procedures . . . 45
3.2.4.2 Dynamic Caching of Query Results for Decision Support Systems . . . 45

4 Implementing Accelerated Approximate Views in DuckDB **47**
4.1 Mechanics of Accelerated Approximate Views 48
4.2 Extending the SQL Syntax for Accelerated Approximate Views 50
4.3 Managing Accelerated Views 53
4.3.1 Cache Creation 53
4.3.1.1 Storing Cache (meta-)data 54
4.3.1.2 Background Cache Creation 56
4.3.1.3 Wasm-specific scheduling problems 59
4.3.2 Reading from Approximate Accelerated Views 61
4.3.3 Modifying Accelerated Approximate Views 65
4.3.4 Cache eviction 65
4.4 Integration into the MotherDuck UI 67

5 User Study **69**
5.1 Collecting User Metrics 69
5.2 User Workloads 71
5.3 Caching Impact 76

6 Experimental Evaluation **81**
6.1 Benchmarking Set-Up 82
6.2 Benchmarking the MotherDuck UI 83
6.2.1 Results 85
6.3 Micro Benchmarks 88
6.3.1 Time to First Tuple 88
6.3.1.1 C++ Results 88
6.3.1.2 Wasm Results 89
6.3.2 Write Speed 92
6.3.2.1 Results 92
6.4 Query Materialization 95
6.4.1 C++ Results 96
6.4.2 Wasm Results 97

7 Limitations and Future Work **99**
7.1 Hybrid Caching 99
7.2 Query Batching 100
7.3 Analyze non-MotherDuck UI workloads 100

7.4 Persisting Accelerated Views in Wasm . . . 101
7.5 Technical limitations . . . 101
7.6 Query optimization improvements . . . 102
7.7 Automatic cache evictions . . . 104
7.8 Evaluation limitations . . . 104
7.8.1 AAV performance under varying network conditions . . . 104
7.8.2 AAV performance with different DuckDB settings . . . 105
7.8.3 Bypassing AAVs . . . 105
7.8.4 Representative queries . . . 105
7.8.5 Standard Benchmarks . . . 105

8 Conclusion **107**

References **109**

A Benchmarking Queries **119**
A.1 Set-Up . . . 119
A.2 Queries . . . 120
A.3 Write Benchmark Query . . . 123

List of Figures

2.1 Database Schema containing the student relations. The *enrolled_in* table references the *ID*s of both the *student* and *study_program* tables. 8

2.2 Unoptimized query plan consisting of relational algebra operators for Listing 2.1, selecting the names of all students enrolled in the *'Big Data Engineering'* track. 9

2.3 Optimized relational query plan of Figure 2.2. 9

2.4 Multi Threaded task scheduler 14

2.5 Single Threaded task scheduler 15

2.6 Life-cycle of a result collector. The *result collector* is filled by the data produced by the query pipelines. A user-consumable query result is created once the result collector is filled or the query is done. The blue box represents busy spinning for materialized queries when all query pipelines are blocked, while the green box indicates the same condition for streaming queries. Busy spinning occurs because all pipelines are blocked. Therefore, the thread spins until a pipeline becomes unblocked. 17

2.7 Mode of operation in DuckDB Web Assembly (Wasm): Instead of executing the entire query in one go, the Web Worker containing DuckDB is periodically instructed to execute parts of the query. 19

2.8 Hybrid query plan of Figure 2.3. Joins are split into hash table *building* operators and *probing* operators. Blocks in red are executed locally, while green blocks are executed remotely. Operators within the same pipeline are within the same colored box. 21

2.9 Extension of the execution model of DuckDB Wasm. While the core mechanics are the same, communication with the MotherDuck backend is done through a separate grpc Remote Procedure Calls (gRPC) worker. 22

LIST OF FIGURES

2.10 **MotherDuck UI**. The execution of a `SELECT` statement results in creating a pivot table, allowing users to explore the returned data. On the right, you can see the column explorer displaying column-related statistics. 23

2.11 Queries run in the MotherDuck UI after a user executes a `SELECT` statement. 24

2.12 Queries run in the MotherDuck UI after a user executes a `SELECT` statement with a local cache, thereby enabling fully local data exploration. 25

3.1 Comparison of database sizes for the Snowset and Redset datasets 29

3.2 Comparison of table sizes from the Tableau and Redset datasets 29

3.3 Comparison of query sizes for the Snowset and Redset datasets 30

3.4 Usage patterns reported by [54] 31

3.5 Comparison of query operators of Sloan Digital Sky Survey (SDSS) and SQLShare 34

3.6 Performance results for [48] with the predicates restricted to equalities. . . . 38

3.7 Average time per TPC-H stream. *OFF* represents no caching, while *HIST* decides whether to cache based on historical queries. In contrast, *SPEC* improves on this by additionally caching smaller results anticipating later reuse. Meanwhile, *PA* represents a proactive strategy that executes an initially more expensive query for later gains. [40] 39

3.8 Execution speed on the Y-Axis with the number of cached relations on the X-Axis. [24] 40

3.9 The chunked relations *Product*, *Region*, and *Time*. [19]. 42

3.10 Operation of a federated DB2 working as a cache for the backend DB [2] . . 43

3.11 Performance comparison on an Amazon Elastic Compute Cloud (EC2) c5a.8xlarge. The cache size varies on the left, while the skewness varies on the right [74]. 44

4.1 Interaction with the accelerated view 49

4.2 Decision tree used to determine whether the cache should be used to answer the query accessing the view. 51

4.3 Overview of the actions executed when creating the cache. 57

4.4 Task execution order in Wasm. Because the query thread does not distinguish between query-related and unrelated tasks, tasks associated with other queries might also be executed by the query thread. 60

4.5 Overview of the actions executed when accessing the cache. 62

4.6 Accessing the cache while the limit is not yet satisfied. Each colored box is a transaction 65

5.1 Queries run in the MotherDuck UI whenever a `SELECT` statement is executed. First, the cache is created, and if the entire query fits in the cache, the cache is afterward used to power the tabular preview and the column explorer. If the query exceeds the cache, it is bypassed in favor of the original query. . . 70
5.2 Number of rows returned by `SELECT` queries. Outliers were removed from the graph, but the mean and median were computed with outliers. 71
5.3 Comparison of column count distributions. Outliers were removed from the graphs, but the mean and median were computed with outliers. 72
5.4 Data type usage of `SELECT` queries run in the MotherDuck UI 73
5.5 Analysis of SQL query complexity and feature usage. 73
5.6 Percentage of queries using different SQL features. `WHERE` operators doing implicit `JOIN` s have been converted to `JOIN` operators. 74
5.7 Queries using only a **single** query operator. E.g., queries in the `LIMIT` bucket contain **only** a `LIMIT` operator. Most queries contain various combinations of query operators and are grouped in *other*. `WHERE` operators doing implicit `JOIN` s have been converted to JOIN operators. 75
5.8 Correlation of the total returned tuple count and the time it takes to complete the query. Times were measured with a limit of 1024 applied to the query. Without the limit, the query would have returned the number of rows indicated on the x-axis. Queries left of the cache limit were run on the locally created cache, while queries right of the cache were rerun. 77
5.9 Same as Figure 5.8, but this time, the time to create the cache is also included in Time Until Completion. Even though queries right of the cache limit do not use the cache, as the query returned to many tuples, the time it took to create the partial cache is included. 78
5.10 Comparison of the time it takes to create a cache (steps 1) to 4)), compared to the time it takes to execute queries 5) and 6). Because the queries of steps 5) and 6) run in parallel, we take $max(t_{end\_5)}, t_{end\_6)})$ to determine the runtime. The scale of the X-Axis changes after the cache limit has been reached and outliers have been removed. Additionally, the query complexity is plotted over the total number of rows. 79

LIST OF FIGURES

6.1 Queries executed to measure the impact on execution speed when replacing the old caching mechanism with accelerated approximate views. 83
6.2 UI simulation running in Wasm. 84
6.3 Time until the first tuples have been read in C++ . Streaming queries measure the time until the first DataChunk of 1024 tuples was retrieved. `CREATE TABLE AS SELECT` (CTAS) and Accelerated Approximate View (AAV) measure the time starting before the creation of the AAV/table ending after the first tuples were read from the relation. 89
6.4 Time until the first tuples have been read in Wasm. Streaming queries measure the time until the first DataChunk of 1024 tuples was retrieved. CTAS and AAV measure the time starting before the creation of the AAV/table ending after the first tuples were read from the relation. 90
6.5 Slowdown of an AAV compared to a CTAS statement when running a query with almost no computational overhead. 92
6.6 Comparison of the write speeds of in-memory and disk-based accelerated views. We plot the slowdown resulting from disk-based storage compared to in-memory storage. 94
6.7 Execute queries until the query has completed running or the cache has completed building. 96
6.8 Execute queries until the query has completed running or the cache has completed building. 97

7.1 Optimization applied to the underlying query of Listing 7.1 (left), compared to the same query when used in a `UNION ALL` (right). 103

List of Tables

3.1 Queries aggregated by consumed CPU time. *Avg(read)* represents the average number of GB read by a query, *Any(mat)* represents the percentage of queries that materialized any data, with *Avg(net)* representing the average amount of data exchanged for distributed queries. [47] 32
3.2 Redset queries and runtime distribution [46] 32
3.3 Data type distribution in Redshift [46] and Tableau [66] 33
3.4 Percentage of queries containing a query operator in column one, with the maximum number of the same operator in column two for Tableau [66] . . . 34
3.5 Query Types in [46] 35

5.1 Data type usage in MotherDuck, Redshift ([46]) and TPC-[H|DS] 74

6.1 Benchmarking System 82

A.1 Variables used in the following queries 119

LIST OF TABLES

List of Listings

2.1 SQL query selecting the names of all students enrolled in the *'Big Data Engineering'* track from the database schema defined in Figure 2.1 8

4.1 Result syntax 52
4.2 Creation of a approximate accelerated view called my_result 54
4.3 Parsed representation of Listing 4.2 54
4.4 Accessing the accelerated view. The accelerated view is the relation called *my_result*. 61

7.1 Union access with `LIMIT` 103

A.1 Setup 119
A.2 Query #1 120
A.3 Query #2 121
A.4 Query #3 121
A.5 Query #4 121
A.6 Query #5 121
A.7 Query #6 121
A.8 Query #7 122
A.9 Query #8 122
A.10 Query #9 122
A.11 Query #10 122
A.12 Query #11 123
A.13 Write benchmark 123

Acronyms

AAV Accelerated Approximate View. viii, 1–3, 52, 54, 60, 63, 65–67, 79, 82, 83, 85, 87–90, 92, 93, 95–105, 107, 108, 119

ACID atomicity, consistency, isolation, durability. 5

API application programming interface. 8, 105

AST Abstract syntax tree. 11, 61

AWS Amazon Web Services. 82, 104

BI Business Intelligence. 27, 28, 31

CPU Central Processing Unit. 93

CTAS `CREATE TABLE AS SELECT`. viii, 88–90, 92, 93, 95, 96, 98, 107, 119

CTE Common Table Expression. 69, 100

DAG Directed Acyclic Graph. 37

DBMS database management system. 6, 7, 10, 41, 43–45

DOM Document Object Model. 18

DRY Don't Repeat Yourself. 5

E2E End To End. 87

EC2 Amazon Elastic Compute Cloud. vi, 44, 82, 104

ETL Extract, Transform, Load. 35

FIFO First in First out. 14, 15

FPDB FlexPushdownDB. 43

Acronyms

gRPC grpc Remote Procedure Calls. v, 20, 22

I/O input/output. 11, 19, 41

JDBC Java Database Connectivity. 100

LCS Largest Cached Size. 37

LFU Least Frequently Used. 43, 65, 66, 104

LRU Least Recently Used. 37, 39, 41, 45, 66, 104

OPFS Origin private file system. 24, 47, 101

QUEL Query Language. 45

RAM Random-access memory. 82, 92

RDBMS relational database management system. 5, 7

S3 Amazon Simple Storage Service. 10, 20, 44

SDSS Sloan Digital Sky Survey. vi, 28, 30, 31, 34

SIMD single instruction, multiple data. 10

SQL Structured Query Language. 2, 3, 5, 7, 11, 23, 24, 31, 33, 44, 48, 50, 53, 54, 65, 70, 72, 87, 107

SSB Star Schema Benchmark. 43, 81, 105

SSD solid-state drive. 44

Wasm Web Assembly. v, vi, viii, 1–3, 5, 18–20, 22, 23, 44, 48, 56–58, 60, 63, 72, 82, 84, 88–91, 97–99, 101, 102, 104, 105, 107, 108, 119

1

Introduction

In the rapidly evolving landscape of data management and analytics, the efficiency and effectiveness of database operations are essential. DuckDB is an in-process analytical database engine that handles small and large datasets [45]. MotherDuck is a data warehousing tool that uses DuckDB, enabling it to scale beyond the resources available on a local device. This is done using hybrid query execution (in the following called hybrid execution), which enables queries to be executed locally on an engineer's device and simultaneously in the cloud. The decision on where to execute specific query parts depends on the data's location, which can be either on the user's computer or in the cloud [4]. In addition to running natively as an embedded library, DuckDB can run in the browser's Wasm environment [38]. This feature makes using DuckDB for analytical dashboards like Streamlit [60], the MotherDuck UI, or other data apps easy. In combination with hybrid execution, this architecture allows for running queries on local data and remote data, allowing for the offloading of heavy data operations and data persistence to MotherDuck.

However, this also comes with challenges, as transferring data between MotherDuck and the browser adds additional latency to hybrid queries. Additionally, hybrid query execution is most effective when local data is available, as MotherDuck does not use local computing resources if that is not the case.

Therefore, the most significant challenges in providing fast experiences for data apps are the latency introduced by transferring query results from the cloud to DuckDB Wasm and the absence of local data, which could be used to run parts of the query locally. This thesis aims to develop functionality that addresses these issues by creating a "database view"-like operator that extends the standard database view's behavior by caching parts of the database view. The name of this concept is Accelerated Approximate View (AAV). The idea is to populate the view's cache in the background whenever possible and with

minimal impact on other running queries. Whenever the AAV is subsequently used, it checks if the (partially built) cache would satisfy the query being run on it. Maintaining the cached data is out of the scope of this thesis. As a result, a change in one of the relations underlying data will not be reflected in the AAV cache. This leads us to the following research questions:

RQ1 How would an Structured Query Language (SQL) extension look like that allows for creating a view with associated locally cached data?

RQ1.a What query optimization opportunities are created by this?

RQ2 How can background materialization be implemented?

RQ2.a How is background materialization possible in single-threaded environments like Wasm?

RQ3 How do you evaluate the effectiveness of a solution that primarily speeds up queries that only read parts of a view?

RQ3.a What is the resource overhead one must pay for caching parts of views?

RQ3.b What kind of workloads will the the AAV be used for?

1.1 Contributions

- **SQL Extension:** Entails the design of a SQL extension, allowing for the creation and management of a new database object facilitating the AAV functionality.
- **User Study:** We collected usage data of MotherDuck UI users to analyze the workloads, allowing us to draw conclusions about the complexity and nature of user-issued SQL queries.
- **Literature Review:** We conducted a literature review to analyze different caching architectures and compare them with each other. Additionally, we investigated common database query usage patterns to better understand the impact caching might have on them.
- **Wasm-compatible Background Materialization:** Introduction of a new background execution mechanism designed for both native and Wasm execution.
- **Optimized Access Times:** To improve cache access times, accessing queries are examined for certain characteristics, allowing the optimizer to decide whether to use or bypass the cache.

1.2 Outline

We begin by providing necessary background information in Chapter 2, which includes details on database systems and relational algebra in Section 2.1 an Section 2.3. Following this, DuckDB internals are covered in Section 2.4 followed by an explanation about DuckDB Wasm in Section 2.5, hybrid query execution and details on hybrid query execution in Wasm in Section 2.6.

Following this, we investigate Section 3.1 and Section 3.2 in Chapter 3 covering existing literature.

Chapter 4 covers the implementation, starting at the design of the SQL extension used for AAVs in Section 4.2, followed by a description of the cache creation process in Section 4.3.1 and a section on the reading from AAVs in Section 4.3.2.

After that, in Chapter 5, we will cover the query workloads run in the MotherDuck UI (see Section 5.2) as well as the current impact of caching on query execution in Section 5.3.

The in the implementation section discussed approach is then evaluated in Chapter 6 focusing on microbenchmarks (see Section 6.3), as well as E2E caching impacts (see Section 6.4.

The insight discovered in the experiments section allows us to discuss the current design limitations in Chapter 7, ending with a conclusion in Chapter 8.

2

Background

This chapter presents some relevant background knowledge about database systems, focusing specifically on DuckDB. It introduces key concepts related to relational database management system (RDBMS), including relational algebra, views, materialized views, and the relational model. We then explore DuckDB, detailing its general goals, high-level architecture, and internal components. Following this, the capabilities of Web Assembly (Wasm) are examined, along with DuckDB's ability to run within a Wasm environment, emphasizing the different execution models of DuckDB and DuckDB Wasm. Lastly, the integration of MotherDuck into DuckDB is discussed.

2.1 Relational Database Management Systems

RDBMS are a cornerstone of modern data management and have quickly become the norm [25, p. 3], enabling the efficient storage, retrieval, and manipulation of relational data. Based on the relational model by Codd [13], most RDBMS use SQL as a standardized interface to interact with the database. Integrity and consistency are ensured through mechanisms such as transactions following the atomicity, consistency, isolation, durability (ACID) properties [30].

2.1.1 Views and Materialized Views

In software development and database management, the Don't Repeat Yourself (DRY) principle is crucial for reducing redundancy. In the context of RDBMS, **views** embody this principle by allowing users to define virtual relations based on queries and reuse them later [27, p. 290]. Views can be seen as a "macro" for larger queries [57]. Moreover, as

database relations evolve, views allow creating a virtual relation, modeling earlier versions of the database relation, thereby retaining backward compatibility [57].

A **Materialized view** stores the results of views physically to speed up access to the view's data. However, for the materialized view to accurately reflect the content of its underlying relations, the view has to be maintained, a concept called *view maintenance* [9]. Because it would be wasteful to recompute the view's data entirely, it is updated incrementally, which is called *incremental view maintenance* [33]. Materialized views are beneficial when referencing relations that infrequently change, thereby making the view maintenance inexpensive, but are queried often, thereby saving on the cost of having to recompute the view's underlying data, again and again [28].

2.2 Client Server Architecture

The majority of database systems are implemented as server-client architectures. This architecture can be categorized into two primary models: *query shipping* and *data shipping* [24]. In data shipping, the query processing occurs on the client side, while the server provides the necessary data for the client to execute the queries. The data may be cached on the client side for future use. A significant advantage of this model is the efficient use of the client's resources, which is particularly beneficial when the server's resources are constrained.

Query shipping, on the other hand, executes all queries on the server side and only sends the results back to the client. This model is favorable when the client has limited resources, and the server has abundant resources. At the time of writing, all of the top 10 database management system (DBMS) in db-engines are client-server databases using query shipping, except for SQLite. Unlike the others, SQLite does not use a client-server architecture; instead, it is an embedded database system linked as a library to the application that uses it [23].

2.3 Relational Algebra

In the relational model, which was first introduced by Codd in 1970 [13], different tables represent the data and the relationships between the various data elements. Tables consist of rows and columns; in this thesis, the terms "rows" and "tuples" will be used interchangeably. Relational algebra takes these relations and, by applying relational algebra operators, such

as selection, projection, Cartesian product, and union, creates a new relation [53, p. 48 - 59].

1. $\sigma_{\phi}(r)$: The select operation selects tuples from the relation r that satisfy the predicate ϕ.

2. $\Pi_{a_1,...,a_n}(r)$: The project operation selects the attributes $(a_1, ...a_n$ from the projection r, thereby creating a new relation.

3. $r_1 \times r_1$: The Cartesian product operator allows combining two relations. Creating a Cartesian product $r_1 \times r_1$ of tuples t_1 and t_2 produces a new tuple that concatenates the attributes of t_1 and t_2.

4. $r_1 \bigcup r_2$: The union operator combines the tuples of relations r_1 and r_2 into a single set. Duplicates are removed, ensuring each tuple appears only once in the resulting set.

Further, the join operator denoted as $r_1 \bowtie_{\phi} r_2$ is the result of applying both the Cartesian product and a selection operation ($r_1 \bowtie_{\phi} r_2 = \sigma_{\phi}(r_1 \times r_2)$). Several types of joins refine this operation:

1. The *natural join* $r_1 \bowtie r_2$ creates a new relation by combining tuples from relations r_1 and r_2 if the values of all shared attributes are equal.

2. The *equi-join*s predicate checks for equality between two attributes ($\phi :=$ (student.id = enrolled_in.student_id))

3. Lastly, *theta-join*s combines all tuples from the relations r_1 and r_2 that satisfy the predicate ϕ.

While relational algebra can express most operations performed in an RDBMS, SQL is typically used for this purpose. For example, consider a query that retrieves the names of students enrolled in the *'Big Data Engineering'* track from a database schema containing the *student*, *enrolled_in*, and *study_program* relations (see Figure 2.1 and Listing 2.1). The relational query plan, which consists of relational algebra operators for the query in Listing 2.1, is illustrated in Figure 2.2. However, this initial plan is far from optimal. It can be made more efficient through techniques like Filter Pushdown, which minimizes the size of relations early by applying filters as soon as possible. This approach reduces the size of relations before applying other relational operators without affecting the final result. In DBMS, this process is called query optimization.

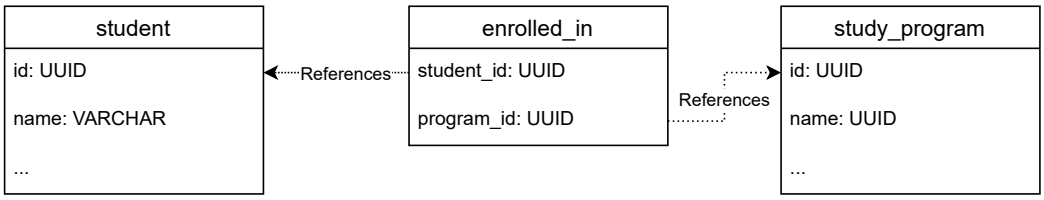


Figure 2.1: Database Schema containing the student relations. The *enrolled_in* table references the *ID*s of both the *student* and *study_program* tables.

```
SELECT student.name
FROM student,
     enrolled_in,
     study_program
WHERE student.id = enrolled_in.student_id AND
      enrolled_in.program_id = study_program.id AND
      study_program.name = 'Big Data Engineering';
```

Listing 2.1: SQL query selecting the names of all students enrolled in the *'Big Data Engineering'* track from the database schema defined in Figure 2.1

2.4 DuckDB

DuckDB is an embedded database system like SQLite. However, while SQLite is tailored for transactional workloads, excelling at short-running queries that retrieve or update a small number of rows, DuckDB is optimized for analytical workloads, which involve summarizing and processing large volumes of data. The following sections will discuss DuckDB's architecture, execution model, scheduling mechanics, and query lifecycle.

2.4.1 Architecture

DuckDB operates entirely within the calling process, eliminating the need for a separate server. This design decision was made to facilitate on-device data analysis and support "edge" computing [45]. As a result, every DuckDB application programming interface (API), regardless of language and runtime environment, comes with its own DuckDB instance.

While traditional analytical database systems like BigQuery and Spark scale horizontally, distributing workloads across multiple nodes, DuckDB scales vertically, leveraging modern hardware with multiple terabytes of RAM and hundreds of CPU cores [6].

Single-machine parallelism provides scalability benefits while keeping the system simple and easy to set up. Unlike scale-out systems, which are often more complex to configure,

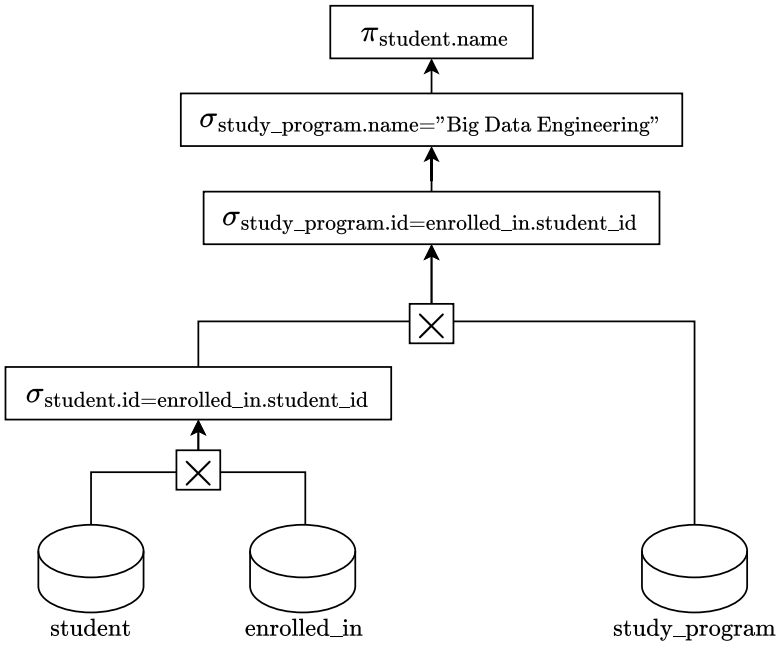


Figure 2.2: Unoptimized query plan consisting of relational algebra operators for Listing 2.1, selecting the names of all students enrolled in the *'Big Data Engineering'* track.

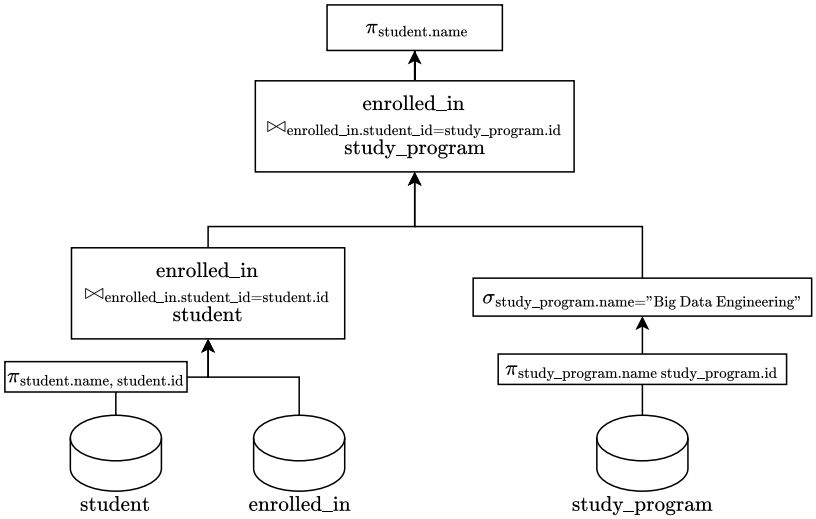


Figure 2.3: Optimized relational query plan of Figure 2.2.

slower to start, and susceptible to networking overhead [75] [17], single-node setups have the advantage of being easy to deploy. DuckDB and MotherDuck argue in their blog "Big Data is Dead" [62] which is also supported by the recent paper "Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet" that this simple architecture is enough for 99% of data analytics use cases, which is also supported by "Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet" [46] [63].

Extensibility DuckDB's functionality can be extended by installing custom extensions, which enable adding new features without modifying the core query engine. For example, several extensions add support for different data types like JSON or Parquet and enable accessing remote sources like Amazon Simple Storage Service (S3).

Consider the JSON extension as an example. When executing a query like `SELECT * FROM 'my.json';` , the extension first binds the relation *my.json* by analyzing the structure of the JSON file. After that, the file is read and converted into a format the query engine can use and queried like a standard DuckDB table.

Vectorized Query Execution In its implementation, DuckDB uses vectorized query execution [77] [11] [45]. The idea of vectorized query execution is to pass a vector (containing parts of a column), to physical operators. This differs from tuple-at-a-time execution, which continuously processes rows of tuples.

Compared to a tuple-at-a-time execution, this reduces the overhead per operator call, enables the compiler to perform loop optimizations - often leading to the compiler generating single instruction, multiple data (SIMD) instructions, and allows DuckDB to pass small, cacheable vectors from operator to operator [11]. SIMD allows a single instruction to process multiple data elements simultaneously, thereby increasing the throughput by performing parallel operations on vectorized data. Additionally, as columns are stored independently, the reading of a single column becomes more efficient by skipping non-relevant columns while still maintaining sequential data access [1].

Processing Model Query execution in DBMS can generally be categorized into two models based on the direction of data flow: the *push-based* model, also known as bottom-up, and the *pull-based* model, also referred to as top-down. In a pull-based model, parent physical operators request data from their children when needed. This allows parent operators to control the data flow, as it can stop requesting data when blocked or when sufficient data has been retrieved. In the push-based model, child physical operators

generate and send data to their parent operates without waiting to be polled. Unlike a pull-based model, where operators can stop pulling data from their children when blocked by certain conditions, in the push-based model, the parent operator has to signal its child to stop sending data when it cannot handle more data [44].

DuckDB initially used the pull-based model to execute queries. It has since transitioned to a push-based execution model to gain more flexibility during pipeline execution, improve parallelism, and reduce code duplication [43]. Despite this transition, physical source operators, which are operators responsible for emitting data, continue to operate in a pull-based manner [44].

2.4.2 Query lifecycle

In DuckDB, the query lifecycle begins with parsing the query using the DuckDB parser, returning the Abstract syntax tree (AST) representation of the query. If this parser fails to parse the query, an extension hook allows extensions to parse the query for DuckDB, enabling the extension of the SQL syntax. Following the query's parsing, the binding phase generates a tree of Logical Operators, known as the Logical Query Plan. References to tables and columns are resolved during this process by linking them to internal database objects by performing lookups in the database catalog, which contains metadata defining database objects like tables. If a table cannot be found, DuckDB checks if any extension can bind the relation, a process known as Scan Replacement . During the binding phase, Table Functions, which are parameterizable SQL functions (see Section 2.4.3.2 for more details), are either bound to their corresponding database objects or replaced by another relation.

Once binding is complete, DuckDB optimizes the Logical Query Plan by, amongst other things, rearranging logical operators to minimize the computational cost and input/output (I/O). After DuckDB's built-in optimizations are completed, extensions can perform additional optimizations on the plan.

The optimized Logical Query Plan is then translated into a Physical Query Plan by mapping Logical Operators to their corresponding Physical Operators (see Section 2.4.3.3). Extensions that introduced new Logical Operators are expected to provide physical counterparts. Subsequently, the Physical Query Plan is divided into multiple Pipelines (see Section 2.4.3.4), with each pipeline beginning with a Physical Source Operator and ending with a Physical Sink Operator. Pipelines are broken down into tasks, which are subsequently scheduled and managed by the Task Scheduler.

Execution of these tasks depends on the runtime environment. Tasks are executed using a thread pool in environments where multi-threading is possible. In single-threaded environments, query execution is entirely driven by the calling thread.

2.4.3 DuckDB Internals

2.4.3.1 Views

This section examines how DuckDB internally handles views. Although materialized views are not yet supported in DuckDB, there are ongoing efforts to implement incremental view maintenance, which is crucial for the effective use of materialized views [7].

When a `CREATE VIEW` statement is issued in DuckDB, the creation of the view stars by binding and parsing the query. During the binding phase, a *LogicalCreate* operator is created, containing all information about the view. This logical operator is then transformed into a *PhysicalCreateView*, which handles the actual creation of the view. The view is subsequently stored in the *CatalogSchema*, a transaction-protected storage location for database objects located in the catalog, which, in the following will be used as a synonym for database.

DuckDB's binder resolves the view name to its corresponding database reference when accessing a view. After locating the view and its associated query, it is converted into a subquery within the accessing query.

2.4.3.2 Table Functions

The GET operator in DuckDB is a generic table scan that sequentially produces tuples. Its most common uses are to scan DuckDB tables and external data sources, such as Parquet/CSV/JSON files, or tables from other database systems, such as PostgreSQL and SQLite. In the SQL syntax, external scans also take the form of functions like `parquet_scan()` with parameters, (e.g. `SELECT * FROM parquet_scan('s3://mybucket/myfile.parquet');`). To make this more user-friendly, DuckDB also allows the syntax `SELECT * FROM s3://mybucket/myfile.parquet;` . It allows table functions to register a *replacement scan* that will inspect a name/path of non-bindable reference and, if suitable, replace that name/path with the proper table function (as in the first example). DuckDB also uses GET for meta-data and information functions, like viewing configuration settings, and MotherDuck also uses table scans to provide, amongst others, information on cloud databases.

Beyond substituting a non-bindable reference with a table function, table functions can be replaced with subqueries or other relations, a mechanism called *bind replacement*. The table function in the query `CALL replace_with_subquery('table_name');` is substituted with `SELECT * FROM table_name;` during the binding of the table function, thereby completely removing the table function.

2.4.3.3 Physical Operators

Physical operators provide the implementation of relational algebra in DuckDB. They are arranged in *pipelines*: linear sequences of operators that start at a source operator, with a possible streaming/pipelined operator in the middle, and ending in a sink operator. Physical operators can be categorized into sinks, sources, or both. Sinks are responsible for collecting data, whereas sources provide data. Some operators, like the hash join operator, serve as both a sink and a source. A table scan operator, on the other hand, is a source, while "upload bridges" (discussed in Section 2.6) function only as sinks. DuckDB does not directly schedule physical operators; instead, it uses pipelines containing these operators to manage task scheduling. Physical operators are also responsible for creating these pipelines. During execution, physical operators are organized into nested pipelines, which may depend on other pipelines. Any pipeline has a source operator at the bottom and a sink operator at the top. That sink can be a dependency for other operators; for instance, the build of a hash table is a sink operator, and pipelines containing a physical operator that probes this hash table depend on it. Another example is an aggregation, a sink that constructs a hash table with the aggregates. It is a dependency for the pipeline that scans the aggregation result. In both examples, a physical operator (hash-join respective hash-aggregation) functions as a sink and source. There are instances where a physical operator cannot proceed with execution. This situation arises, for instance, when the output buffer of a physical operator becomes full. In such a scenario, the execution is halted until the condition is resolved. Once the condition is cleared, the scheduler has to be notified that the operator is no longer blocked so that the operator can be rescheduled and executed. This process of pausing and resuming execution is known as "resumable pipelines".

2.4.3.4 Query Pipelines and Scheduling

Each query has an instance of an Executor responsible for executing the query. This executor bootstraps the creation of the pipelines by creating a root pipeline. After this, the

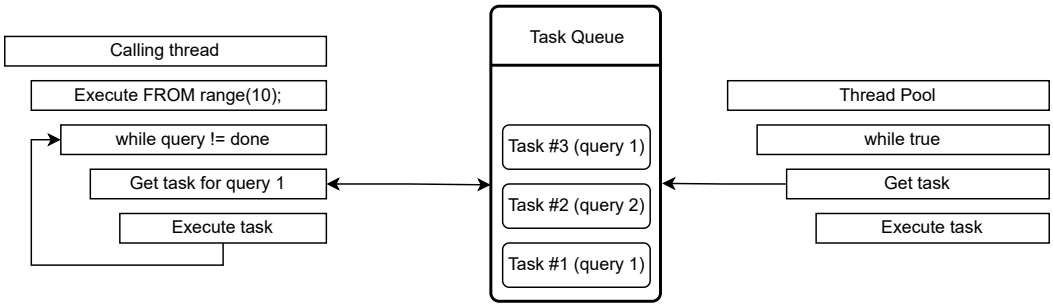


Figure 2.4: Multi Threaded task scheduler

tree of physical operators is traversed, allowing physical operators to either add themselves to a pipeline or create their own child pipelines. Pipelines have associated **pipeline events** that coordinate when a pipeline is executed. Pipeline events can depend on other pipeline events, indicating that a dependent pipeline event should only be dispatched after the completion of its corresponding dependency.

After all pipelines have been created, the executor starts scheduling events that do not depend on other events, meaning they can be executed without concern for execution order. These events will internally execute the pipeline's physical operators. Additionally, these events dispatch new pipeline events. For example, the pipeline finish event will be sent after completing a pipeline, notifying other pipelines that that pipeline has stopped executing. To execute the physical operators that an event or a pipeline represents, the pipeline event schedules **tasks** in the Task Scheduler. The Task Scheduler then assigns these tasks for execution. Once a pipeline event has been completed, it signals its dependent events, allowing them to start scheduling tasks. Once all pipelines associated with a query have been completed, a query is deemed done.

The task scheduler determines the execution order of tasks queued by pipeline events. Each task in the queue is associated with a query identifier, enabling a match between tasks and queries. The tasks are maintained in a queue, but the dequeuing strategy can differ based on the environment. In multi-threaded environments, the task scheduler utilizes a pool of threads that continuously process the scheduled tasks using First in First out (FIFO) independent of their query. Any thread that issues a query will also contribute to executing tasks associated with that query (see Figure 2.4).

In single-threaded environments, the execution of scheduled tasks is solely driven by the query thread. Additionally, the query thread will not distinguish between tasks related to the query and query-unrelated tasks. Instead, it will process all tasks using the FIFO

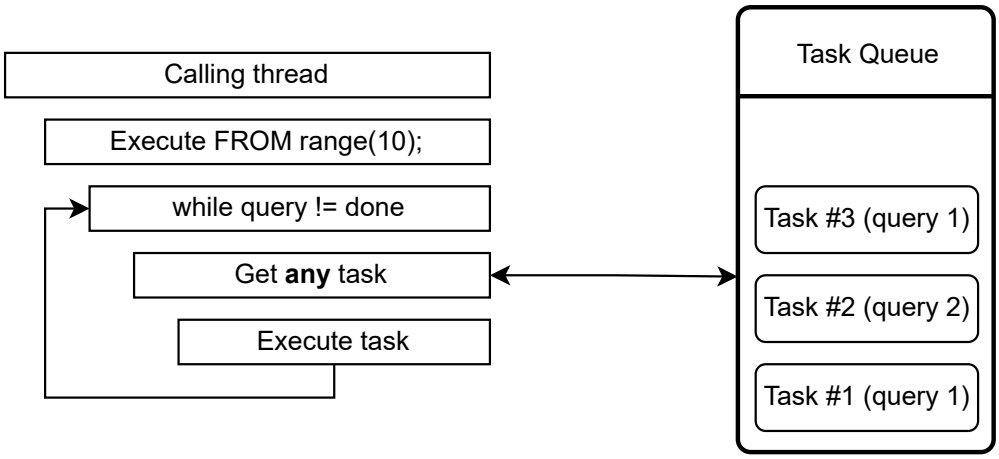


Figure 2.5: Single Threaded task scheduler

scheduling algorithm. The thread will, however, stop retrieving and executing tasks from the task queue once the query has been completed, even if there are still tasks in the task queue. Every task that remains in the task queue after the query has finished will only be executed once the next query is run, as this will cause the calling thread to once again retrieve tasks from the task scheduler using FIFO until the query has been completed. As illustrated in Figure 2.5, tasks from multiple queries can be in the task queue, even within a single-threaded environment. This occurs when a query, during its execution, initiates another query (e.g., a table function that runs a separate query while being executed), thereby adding new tasks to the Task Queue. It can also happen that queries are not executed entirely but only partially, which is possible using the pending query API (this allows the creation of a query without executing it directly; for more, see Section 2.4.3.7), resulting in multiple simultaneous queries, even in single-threaded environments.

2.4.3.5 Result collectors

The last physical operator executed during a query is the result collector. It is a specialized physical operator responsible for gathering the data produced by its child physical operators and preparing its collected data for consumption. Different query types require different kinds of result collectors. For example, the materialized query (a type of query that collects all created data before exposing it to the user) uses a result collector that aggregates all data before producing a consumable result for the user. In contrast, the streaming result collector processes data by filling a buffer and only fetching additional data when the buffer

is depleted. This approach allows for the consumption of query data even while the query is still running, which is the fundamental mechanism behind streaming queries.

Figure 2.6 illustrates the lifecycle of the result collector. It is important to note that this figure does not represent the execution process of DuckDB. Instead, it depicts the sequence of decisions and the order in which actions are taken.

2.4.3.6 Streaming Queries

Streaming queries do not execute the entire query in one go; they only execute it until the streaming queries result collector buffer has been filled. Once the buffer is filled, the physical result collector blocks, pausing the query. The query remains paused until enough data has been removed from the result collector's buffer by the user consuming the result, creating space for new query data to be buffered. Furthermore, streaming queries continue executing as long as none of the physical operators, including the result collector, block. When a physical operator blocks, the result collector creates a StreamQueryResult, allowing the user to access the query's intermediate results. This StreamQueryResult is returned to the user even though the query execution is not fully completed and is temporarily paused. The StreamQueryResult reads from the result collector's buffer to provide these results. When the buffer is depleted, the *StreamQueryResult* resumes executing the paused query itself to retrieve additional data.

Unlike before, where the execution was driven within the Connection, execution is now driven by the StreamQueryResult. When one of the query's operators blocks a pipeline this time, execution control is not returned [1] to the user like before[2]. Instead, the Result object busy spins, continuously attempting to execute the pipeline until the operator becomes unblocked, allowing the query to resume execution and the result collectors buffer to be filled.

2.4.3.7 Pending Queries

Pending queries in the DuckDB API give the query creator greater control over the execution of the query pipelines. Unlike streaming queries, the calling thread does not participate in the query's execution. Instead, tasks are scheduled in the TaskQueue, and in multi-threaded environments, the Scheduler's thread pool processes these queued tasks.

[1] `https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/main/buffered_data/simple_buffered_data.cpp#L58-L65`

[2] `https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/main/pending_query_result.cpp#L70`

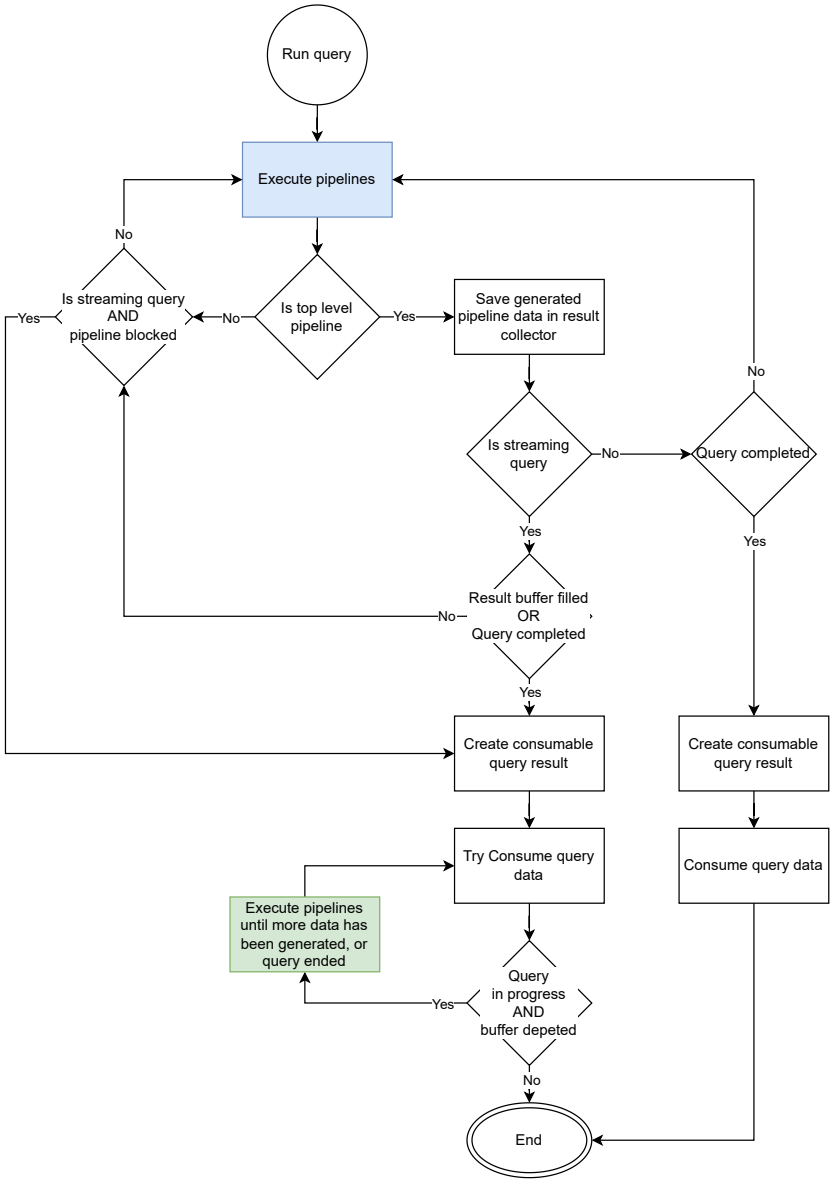


Figure 2.6: Life-cycle of a result collector. The *result collector* is filled by the data produced by the query pipelines. A user-consumable query result is created once the result collector is filled or the query is done. The blue box represents busy spinning for materialized queries when all query pipelines are blocked, while the green box indicates the same condition for streaming queries. Busy spinning occurs because all pipelines are blocked. Therefore, the thread spins until a pipeline becomes unblocked.

However, in single-threaded environments without a thread pool, issuing a query through the pending query API does not result in the query being executed. In such cases, the query is executed exclusively by invoking the `ExecuteTask` method on the pending query object, which will execute precisely one task of the pending query.

2.5 Web Assembly

Wasm is a "safe, portable, low-level code format" designed for efficient execution [68]. Other than the name implies, Wasm is not only intended for the web but is a compilation target for different programming languages that can be executed in any Wasm runtime. The browser can provide this runtime, but Wasm can also be executed through standalone tools such as Wasmer [71] or WasmEdge [70]. In the browser, Wasm is mainly used to speed up performance-critical tasks, where using JavaScript would be too slow [69]. While it was designed to run at near-native speed, there is still a significant performance gap compared to native execution [36].

DuckDB also supports a Wasm compilation target, enabling it to run directly in the browser. Since the browser UI is single-threaded, all UI interactivity is managed by a single event loop, in the following called the *main event loop*, with no other process being able to modify the Document Object Model (DOM), which is an "API for accessing and manipulating documents (in particular, HTML and XML documents)" which is used to make websites interactive [72]. However, the browser provides Web Workers [73], isolated processes that can execute code in parallel while communicating with the main event loop that instantiates them. Web Workers are themselves single-threaded and have their own event loop. There is a compilation target for DuckDB Wasm, which supports multiple threads. However, MotherDuck does not use this compilation target. Therefore, the rest of the thesis will assume that Wasm is single-threaded.

Given that a single-threaded event loop manages all UI interactivity, it is crucial to avoid performing complex computational operations directly within the website's main loop. DuckDB Wasm operates within a Web Worker to address this. The Web Worker can communicate with either the main event loop or other Web Workers through the browser's Web Worker communication API. When a user initiates a query in DuckDB, it is sent to the Web Worker [3], where DuckDB processes it. The query results are then returned to the

[3] `https://github.com/duckdb/duckdb-wasm/blob/de39e10e74f64aad94955353c84bdd225c66ef60/packages/duckdb-wasm/src/parallel/async_bindings.ts#L108-L115`

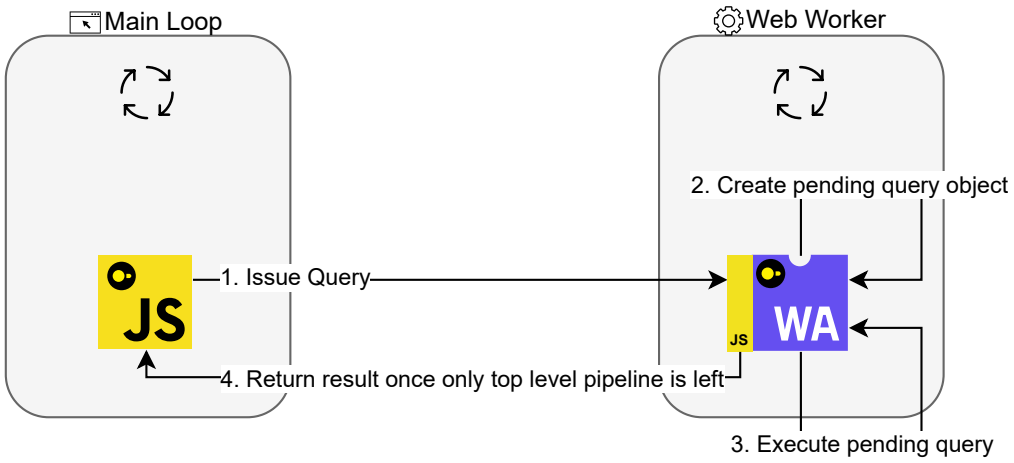


Figure 2.7: Mode of operation in DuckDB Wasm: Instead of executing the entire query in one go, the Web Worker containing DuckDB is periodically instructed to execute parts of the query.

main event loop using the same communication API.

Since Web Workers are single-threaded, no other task can be executed in the Web Worker while a query is executed. To maintain responsiveness for incoming requests, DuckDB has to periodically interrupt query execution, allowing the Web Worker to handle other events, such as incoming messages or asynchronous I/O operations. Yielding execution is especially important for blocking physical operators where busy spinning (as done in streaming queries) would lead to a non-responsive Web Worker, halting any query process.

This mode of operation is achieved by creating a pending query (see Section 2.4.3.7) within the Web Worker. Depending on whether the Wasm environment supports having multiple Web Workers or only a single Web Worker, the pending query is then either executed in bursts [4] [5] or in one go. Once every pipeline, except for the pipeline directly below the result collector, has been executed, the result collector is wrapped in a Query Result object, allowing the query data to be consumed and sent to the main event loop (see Figure 2.7.

[4] https://github.com/duckdb/duckdb-wasm/blob/de39e10e74f64aad94955353c84bdd225c66ef60/packages/duckdb-wasm/src/bindings/connection.ts#L38-L57

[5] https://github.com/duckdb/duckdb-wasm/blob/de39e10e74f64aad94955353c84bdd225c66ef60/lib/src/webdb.cc#L167-L187

2.6 Hybrid Query Processing

The previous sections discussed DuckDB's operation in standalone mode. In this thesis, the focus shifts to MotherDuck. MotherDuck is a DuckDB extension designed to facilitate communication and collaborative query execution with a cloud-hosted DuckDB instance. It extends DuckDB's in-process query execution model by enabling the execution of query segments on MotherDuck servers. Unlike query or data shipping, where the entire query is executed either on the server or the client, MotherDuck's approach, called hybrid query processing, allows for the distribution of query execution between the server and the client.

The decision on where to execute each physical operator is made during query optimization. The query will only be executed on the client if the queried data is entirely stored locally. However, when some data is stored on MotherDuck servers or remote locations, like S3, query execution becomes hybrid, as the query is executed both on the server and the client. During the optimization of a hybrid query, logical operators within the logical query plan are assigned execution locations, namely remote or local, to increase data locality. When there is a transition between local and remote execution in adjacent operators, a logical bridge operator must be inserted [4]. This bridge operator facilitates the data exchange between the two operators, which typically occurs locally but now must happen over a network. Bridge operators consist of pairs: a sink, responsible for uploading data, and a source, responsible for downloading data from the sink. Both bridge sources and sinks can block their pipelines if data cannot be sent or received.

Consider the execution of the query in Listing 2.1, which selects the names of all students enrolled in the Big Data Engineering program. In this case, the *students* and *study_program* tables are stored remotely, while the *enrolled_in* table is stored locally. The resulting query plan does not differ structurally from the optimized plan in Figure 2.3; however, the relational operators are annotated with their execution locations, namely remote (R) or local (L). Bridges are inserted between nodes labeled (R) and (L) to facilitate the necessary data exchange between server and client and at the top of the query plan to send the data to the client.

2.6.1 Wasm specific adaptions

MotherDuck also supports execution within a Wasm environment, but certain adaptations to the bridge operation must be made. Specifically, exchanging data in the bridge operators using the C++ gRPC library is not possible, as C++ gRPC is incompatible with Wasm [51]. Consequently, data transmission must be handled in JavaScript. This process is

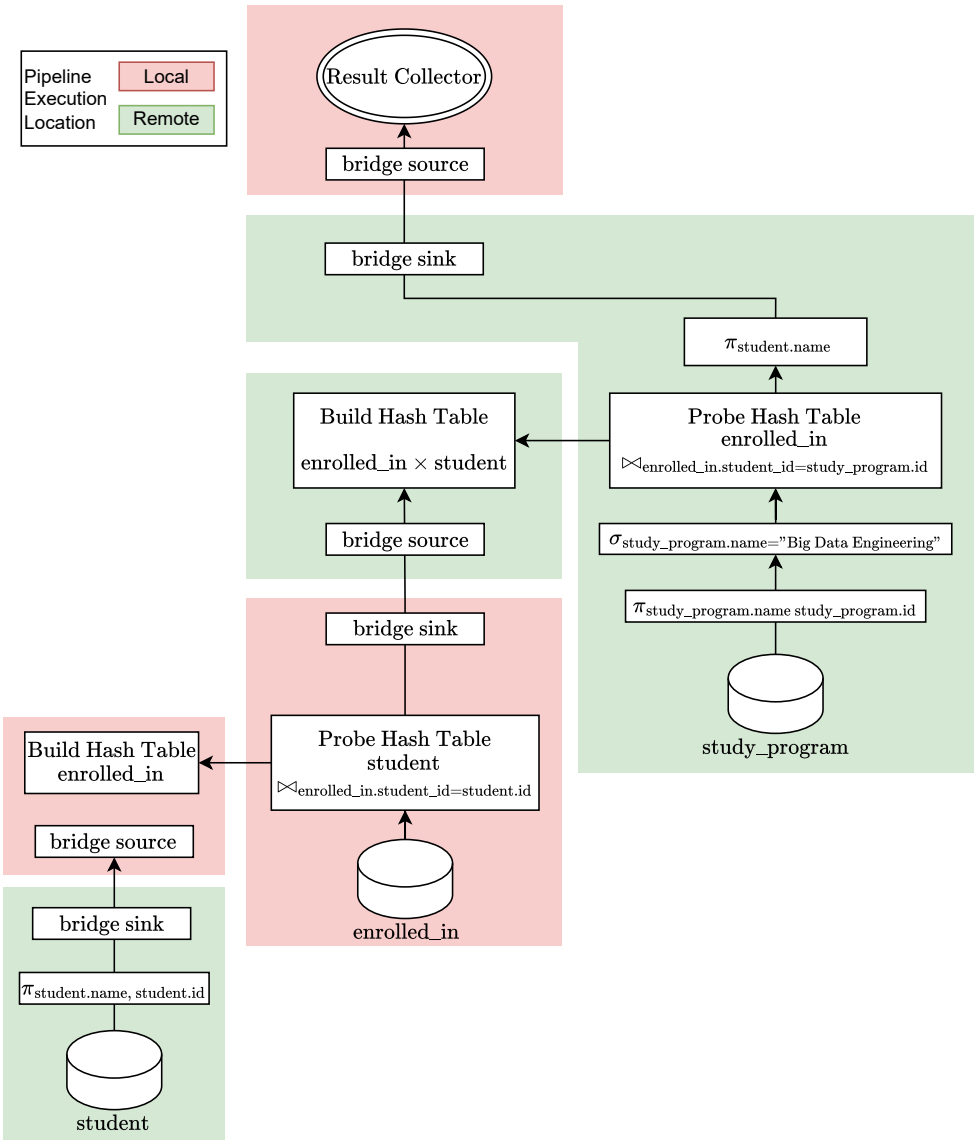


Figure 2.8: Hybrid query plan of Figure 2.3. Joins are split into hash table *building* operators and *probing* operators. Blocks in red are executed locally, while green blocks are executed remotely. Operators within the same pipeline are within the same colored box.

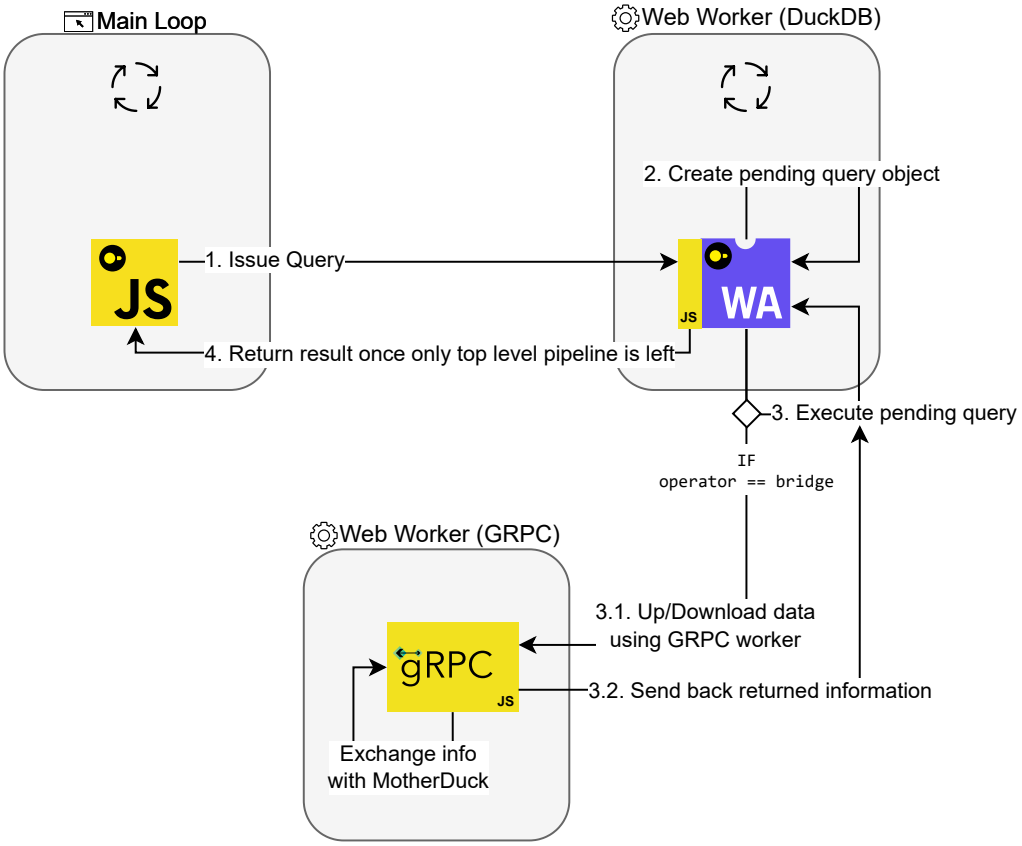


Figure 2.9: Extension of the execution model of DuckDB Wasm. While the core mechanics are the same, communication with the MotherDuck backend is done through a separate gRPC worker.

managed by a dedicated web worker, during which the bridge operator is blocked using resumable pipelines (see Figure 2.9). Once the data exchange is completed, the gRPC worker notifies the Scheduler that the pipeline is unblocked, allowing the pipeline to resume.

As detailed in Section 2.5, queries in Wasm are executed in short bursts, driven by the pending query API, until the top-level pipeline is reached. At this stage, the query is transformed into a result object that then drives the remainder of the execution. This, however, causes a deadlock, as streaming queries do not support non-blocking "blocking" pipelines and would be busy spinning until the pipeline is resumed. A logical limit operator with a limit parameter of 100% is added to each plan to force materialization of the result and avoid a top-level blocking pipeline. The Physical Limit Percent operators act as both sources and sinks by collecting all the generated query data before emitting the specified

percentage of tuples. Because sinks act as pipeline breakers, this process creates a new top-level pipeline that cannot block, thereby resolving the potential deadlock issue in Wasm.

2.7 Inner workings of the MotherDuck UI

The MotherDuck UI is a web-based notebook that allows users to execute arbitrary SQL queries. It has three essential parts (see Figure 2.10 for a screenshot of the MotherDuck UI).

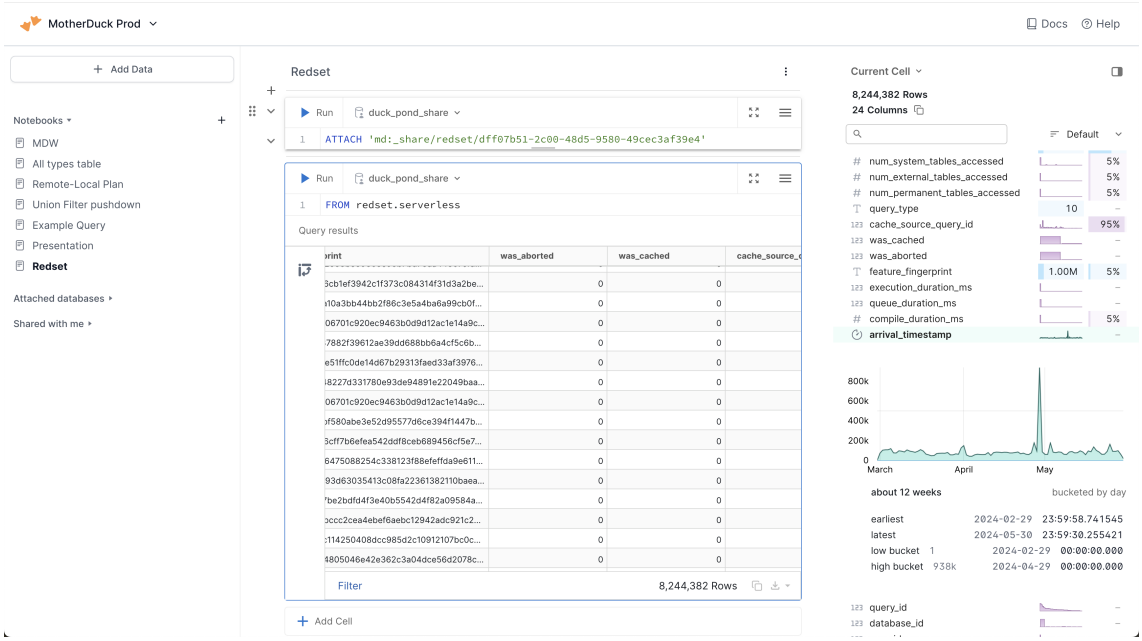


Figure 2.10: MotherDuck UI. The execution of a `SELECT` statement results in creating a pivot table, allowing users to explore the returned data. On the right, you can see the column explorer displaying column-related statistics.

1. The **text editor**, in which the SQL is written. The SQL blocks are executed, and if the last statement is a `SELECT` statement, the **pivot table** and the **column explorer** will be displayed.

2. The job of the **pivot table** is to display results of `SELECT` statements. It is more than just a table; it also supports sorting, pivoting, and filtering of data. The pivot table supports this by running a series of queries on the relation returned by the last query of the SQL code block.

3. The **column explorer** aims to give users a quick impression of the returned data. The statistics are thereby collected by running a series of SQL statements on the data returned by the user query.

Whenever a user runs a query in the notebook, the column explorer and the pivot table execute a series of queries. Because currently, it is impossible to store DuckDB databases in Origin private file system (OPFS), almost all queries are run exclusively on remote data, making them slower than entirely local queries. Although Origin private file system (OPFS) is not supported at the moment of writing, there are ongoing efforts to support it [3].

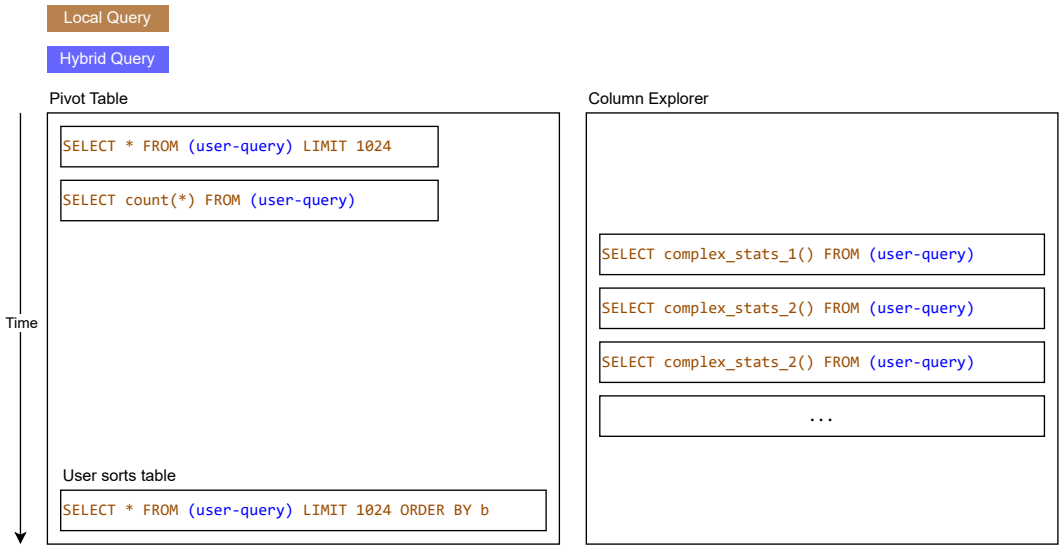


Figure 2.11: Queries run in the MotherDuck UI after a user executes a `SELECT` statement.

As mentioned, the pivot table and the column explorer run SQL queries on the result of the last `SELECT` statement. This is done by making the user query a subquery of the query calculating statistics/displaying the results in the table. As a result, every time the pivot table or the column explorer executes a query, the user's query gets re-executed. You can see the sequence of queries run in Figure 2.11. We start by getting the first 1024 tuples from the query result, as the pivot table lazy loads the result data in chunks when needed. After that, it computes the total number of tuples of the result by running a `count(*)` query on the result. After those two queries have completed running, the column explorer executes several queries to generate column statistics.

Finally, the user might want to re-order the query results by column b, which results in the last query of the pivot table, ordering the results by column b.

Because this mechanism does not utilize any caching, all queries use the users query as a sub-query. This turns all queries into hybrid queries and re-executes the original query.

To prevent consecutive re-runs of the same query, a cache is created in a local in memory or disk-based database using the user's query (see Figure 2.12). After the cache has been created, the user's query is replaced with the cache table relation, making all queries after the initial cache creation query local-only queries.

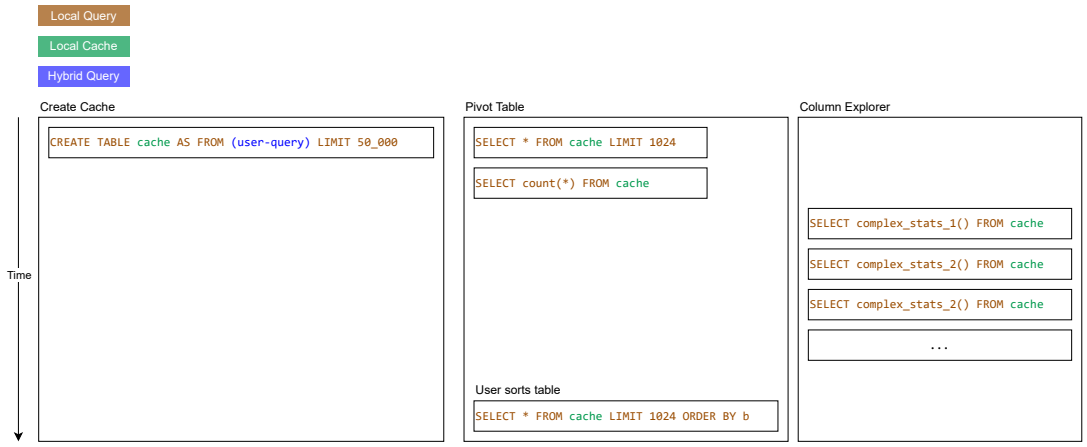


Figure 2.12: Queries run in the MotherDuck UI after a user executes a `SELECT` statement with a local cache, thereby enabling fully local data exploration.

As you can see, Figure 2.12 contains a `LIMIT 50_000` when creating a user cache to protect the client's resources from overly large query results. So, if the query exceeds this limit, only a fraction of the query is saved. This would result in wrong results for the last query, which re-orders the pivot table, the `count(*)` query, and the column explorers queries, which are interested in the entirety of the result. Because the pivot table and the column explorer are oblivious to the cache and only execute queries on the relation passed to them, they cannot determine which query is safe to execute on the cache and which should bypass the cache. As a result, whenever a query does not fit fully in the cache, we completely fall back to the original behavior described in Figure 2.11.

3

Literature Review

The following sections examine the workloads executed in data warehouses. This will give us a frame of reference for usage patterns, allowing us to gauge their usefulness for different workloads later when exploring caching strategies. While examining caching strategies, we will group different approaches and compare them. Finally, we will look at the caching strategy used by the four largest data warehouses: Snowflake, Databricks, Redshift, and Bigquery.

3.1 Usage Patterns

The release of in-depth usage patterns for data warehouses is rare. Therefore, the following section will also contain insights from non-classic data warehouses. The following publications are the main sources for the following section:

1. Jain et al. collected 24275 queries on 7958 datasets in their system **SQLShare**, which was heavily used by academic researchers in the field of physical-, life-, and social sciences. SQLShare was provided free of charge but had a data limit, resulting in a total volume of ~150 GB of data in the entire system. (Published 2016) [35]

2. In "Get Real," Vogelsgesang et al. describe the workloads run in **Tableau**, a Business Intelligence (BI) tool emitting machine-generated query workloads, which are used for exploring and visualizing datasets. (Published 2018) [66]

3. For "Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet," the authors collected usage data from Amazon Redshift, a cloud data warehouse, releasing the anonymized query logs in **Redset**. (Published 2024)[46]

4. Besides the data released in Redset, the only other data-warehouse-related usage statistics were made available by Vuppalapati et al. who published statistics collected for ~70 million queries in a dataset called **Snowset**. (Published 2020)[67]

5. Lastly, we will cover the query patterns discovered by Singh et al., who analyzed the **SkyServer** system, an online platform built for querying the multiple TB of astronomical data collected for **SDSS**. (Published 2007)[54]

While [8], [10] and [37] were considered for the following section, they are too far removed from data-warehouse-centric workloads to be relevant for this section. Of the five data sources chosen for the following section, only Redset and Snowset provide insights into *big data* data warehouse workloads. Tableau is used for smaller BI workloads. While the SkyServer system operates on $\approx$ 5 TB of data [37], its system is not a data warehouse in the classical sense. Finally, SQLShare, while being used by data analysts to explore their data, has size limits that make the queries run more relevant for smaller-scale operations.

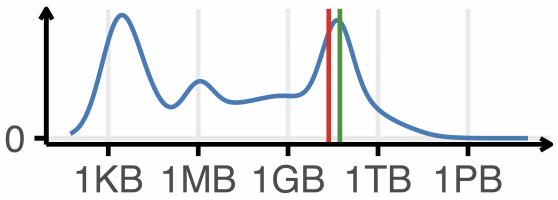
3.1.1 Workload sizes

3.1.1.1 Data size

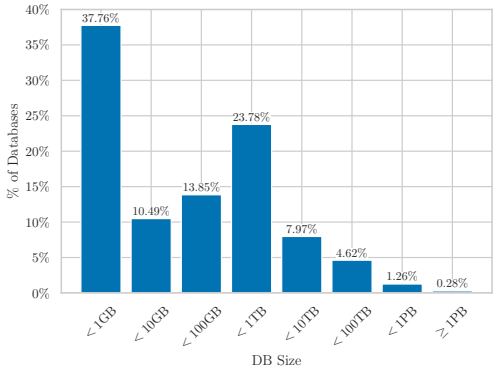
Both Red- and Snowset give broad insights into the workload sizes, defined as the amount of data used per query. Both sources agree that most queries operate on relatively small amounts of data, with "Cloud Analytics Benchmark" reporting that "most queries only touch a few megabytes of data (median = 5.3 MB)"[47] for Snowset. The same is true for Redset, where most tables have fewer than a million rows, with 98% having fewer than one billion rows. This is further illustrated through Figure 3.1b, which shows that over 90% of database tables are below 10 TB, with the bucket containing database tables smaller 1 GB being the largest one with 37.76%.

SQLShare has been excluded for having a limit on data size and only containing 150GB of data in the entire system. Additionally, the SQLShare paper does not report on the scale of workloads executed on their systems. SDSS reports some preliminary numbers for query sizes. However, one has to remember that users cannot upload their data, and the numbers reported in Figure 3.4 only cover the tuples returned by queries, not the amount of data scanned by it.

The data sizes observed by Tableau are similar, with only 553 out of the 62 thousand workbooks containing more than ten million tuples, with most of the datasets used containing less than a thousand tuples (see Figure 3.2a). However, this data might be



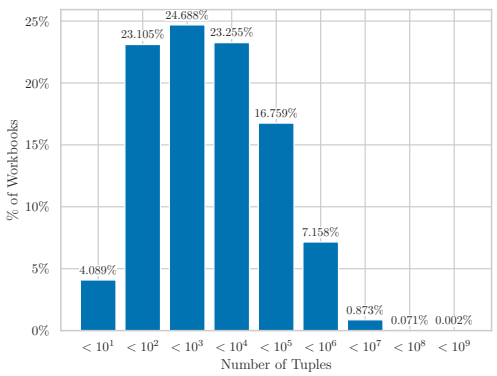
(a) Snowset density function of the database size per customer with the red line being TPC-H (SF=100), green being TPC-DS (SF=100) and blue being the Redset [47]



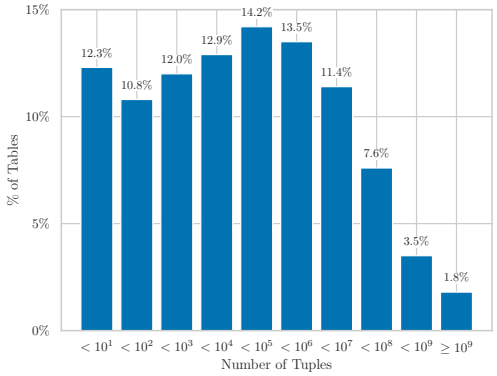
(b) Database size in Redset bucketed by order of magnitude [63] [46].

Figure 3.1: Comparison of database sizes for the Snowset and Redset datasets

skewed, as size-limited free-tier users are also included [66]. The distribution for the number of tuples found by Tableau in Figure 3.2a is very different from the tuple distribution found by the Redset authors (see Figure 3.2b. While for Tableau, most tables contain between 10^1 and 10^5 tuples, tables in Redshift are more evenly distributed between 0 and 10^7 tuples. In Figure 3.1, both the Snowset and the Redset display a similar distribution for their database sizes. Assuming that database size and table tuple count correlate heavily, we can infer that the tuple count distribution of the Snowset is similar to that of the Redset.



(a) Tableau dataset sizes expressed through the number of tuples bucketed by order of magnitude [66]



(b) Redset table sizes expressed through the number of tuples bucketed by order of magnitude [46]

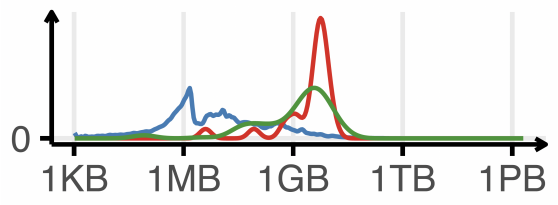
Figure 3.2: Comparison of table sizes from the Tableau and Redset datasets

Similar size-related trends can be seen regarding database sizes in bytes, where both the Snowset (see Figure 3.1a) and the Redset (see Figure 3.1b) show that most user databases

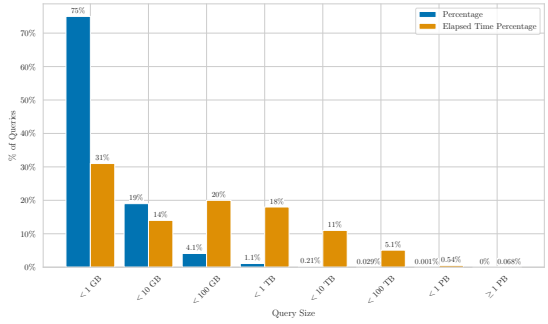
contain less than 1TB of user data, with a few outliers exceeding 1PB and databases $<$ 1GB dominating both datasets. Only a few outliers exceed 1PB in data.

When it comes to bytes read by queries, the distribution is even more skewed towards the lower end of bytes, with 75% of queries reading fewer than 1GB in Section 3.1.1.1, which the Snowset also corroborates with a median of 5.3 MB of touched data per query, which is attributed to a lot of short-running queries. However, there are also 55K (0.01%) queries, exceeding 1TB of consumed data [47].

When looking at the importance of queries touching a certain amount of data in the Redset (expressed through execution time in Section 3.1.1.1), queries reading a small amount of data lose relevance, while queries reading between 10 and 100 TB of data take up 5.1% of the execution time, even though only 0.03% of queries read as much data [63].



(a) Snowset density function of the read bytes per query with the red line being TPC-H (SF=100), green being TPC-DS (SF=100) and blue being the Redset [47].



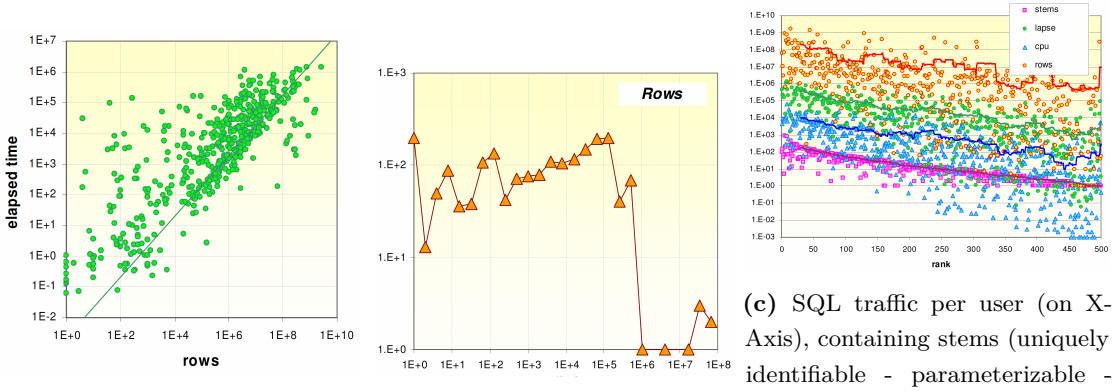
(b) Table scan size, percentage of queries, and percentage of elapsed time for executing queries reading n number of bytes [63] [46]

Figure 3.3: Comparison of query sizes for the Snowset and Redset datasets

Finally, investigating the number of tuples returned by queries in SDSS reveals that most queries run by users ranked 0 to 500 return between 10^6 10^9 tuples (see Figure 3.4c). The average query runs for about 10 minutes, uses 15 CPU seconds, and generates 205K tuples [54], with median-sized queries (110s to 130s) returning between 0 and 10^6 tuples with almost no skew in query count (see Figure 3.4b). There is a steep drop-off after 10^6 tuples. However, this is likely because queries generating more tuples would exceed the 130s time limit. The queries of users 0 to 500 seem to generate and, therefore, most likely also touch more tuples than those in Snow-/Redset. However, without more precise data, this cannot be said definitively. Additionally, an average query in SDSS runs much longer than an average query in Redshift or Snowflake. However, this can be partially explained by hardware advancements and similar workload sizes.

Key Findings: Most workloads and databases consume less than 1 TB of data for Snowflake and Redshift, with even fewer queries consuming a large amount of data. As a BI tool, Tableau operates on smaller data than both data warehouses.

3.1.1.2 Execution times



(a) Average number of rows returned on the X-Axis, with the median-length elapsed time (s) on the Y-Axis [54] [54]

(b) Number of rows returned by a (110s to 130s) query

(c) SQL traffic per user (on X-Axis), containing stems (uniquely identifiable - parameterizable - queries), lapse (E2E execution time (s)), CPU-time (s), and the number of rows returned. [54]

Figure 3.4: Usage patterns reported by [54]

When analyzing the query execution times, both the Redset and the Snowset queries show similar characteristics, with most executed queries being shorter than 1s (see Table 3.2 and Table 3.1). The average execution time for queries in the Snowset is 2.2s, with only $\approx 3\%$ of queries exceeding a runtime of one minute and only 0.0086 % of queries running longer than an hour. The CPU time for those queries is, in fact, even lower, with a median of 870ms because of intra-query parallelization. Additionally, scheduling and initialization are missing from the query CPU time metric [47].

However, this does not mean smaller queries dominate the total execution time. Renen and Leis found that for the Snowset, few large queries utilize most CPU resources and read terabytes of data [47]. This is also in line with the findings of Tigani, who stated that when analyzing the query's data scan size and weighing the queries with their execution time, a very small amount of large queries account for a non-substantial amount of execution time [63](see Section 3.1.1.1).

An interesting observation from SDSS is that query execution times and returned number of tuples seem to correlate almost linearly (see Figure 3.4a). Besides this, not a lot of information has been published on the query execution times.

Key Findings: Most queries are executed in a short amount of time. However, longer-running queries represent a substantial percentage of the total workload.

CPU-Time bucket	Count [%]	CPU time [%]	Avg(read) [GB]	Avg(write) [GB]	Avg(mat) [GB]	Any(mat) [%]
<1 s	52.42	0.2	0	0	0	0.4
<10 s	31.69	1.3	0.1	0	0	6.9
<1 min	9.97	2.7	0.6	0.1	0	12.7
<10 min	4.65	9.2	4.6	0.3	0.1	9.3
<1 h	0.98	15.0	26.0	2.2	0.1	27.4
<10 h	0.25	26.1	159.2	11.7	8.36	49.3
≥10 h	0.02	45.6	1990.6	215.8	26.24	52.5

Table 3.1: Queries aggregated by consumed CPU time. *Avg(read)* represents the average number of GB read by a query, *Any(mat)* represents the percentage of queries that materialized any data, with *Avg(net)* representing the average amount of data exchanged for distributed queries. [47]

Time bucket	% of queries	% of sum(runtime)
<10ms	13.7	0.01
<100ms	48.3	0.40
<1s	24.9	2.30
<10s	9.9	7.30
<1min	2.2	13.30
<10min	0.86	35.70
<1h	0.08	25.20
<10h	0.008	14.30
>=10h	0.00009	1.60

Table 3.2: Redset queries and runtime distribution [46]

3.1.2 Data types

According to the findings of Vogelsgesang et al. and Renen et al., most of the data stored and processed in databases are strings, followed by number and date column types [66] [46] (see Table 3.3). String types seem prevalent because they enable the storage of non-cleaned data. Cleanup happens later during querying, where casting is frequently used for values stored in string columns [66]. One of the findings was that types like Booleans are frequently stored in string fields, with 60% of the *single character strings* being 0 or 1 for Tableau with “Y”, “N”, “yes” and “no” making up 4.5% [66]. Another common (anti-)pattern is to store years or dates as strings and then, during query time, cast them to a timestamp [66].

Data Type	Stored Columns Redset	Predicate Columns Redset	Tableau
varchar	52.1%	21.3%	48.8%
numeric(P, S)	10.2%	14.8%	
integer	9.1%	19.7%	25.6%
bigint	7.0%	11.5%	-
timestamp w/o tz	6.2%	-	2.6% (datetime)
double	4.5%	-	20.5% (real)
boolean	3.9%	-	0.9%
date	2.2%	6.5%	1.5%
smallint	2.1%	-	-
char(N)	1.7%	26.2%	-
float	0.4%	-	-
timestamp w/ tz	0.4%	-	-

Table 3.3: Data type distribution in Redshift [46] and Tableau [66]

This is corroborated by the findings of Jain et al., who discovered that many users created views to cast values of the underlying relation [35]

Key takeaways: With about 50% of the columns being string columns, strings are by far the most prevalent data type, containing not only data that should be encoded as a string but also booleans, dates, and other data types that have not been cleaned enough to qualify for their actual data type.

3.1.3 Query Patterns

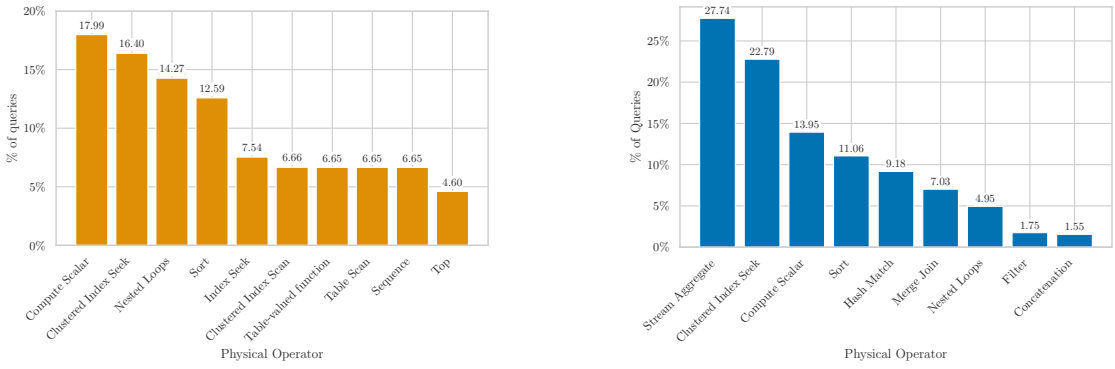
With a substantial amount of columns being string columns, it is not surprising that a non-substantial amount of the sorting operations depends on sorting strings in a case (in-)sensitive way "with over 85% of the string columns in our dataset having a collation" [66].

When it comes to the query operators used in SQL queries Singh et al. found that multi-way joins were commonly used [54], while Jain et al. found that sorting (24%), top k (2%) and outer joins (11%) where relatively common operators used by their users [35] (see Figure 3.5). In Tableau, most queries perform a table scan, followed by a `GROUP BY` and `SORT` ing. Joining is done relatively infrequently. However, joining tables in Tableau is an uncommon use case [66].

Finally, queries issued through Tableau are more complicated as they are machine generated, but queries of all the systems were more complex than those of standard benchmarks like TPC-[H|DS] [66] [46] [47] [54].

Operator	Percentage	Max OP
Table Scan	97.8%	273
Group By	80.7%	253
Sort	17.7%	254
Inner-Join	4.5%	164
Temp. Table Creation	2.2%	-
Outer-Join	1%	247
Percentiles	0.5%	132
Group-Join	0.3%	36

Table 3.4: Percentage of queries containing a query operator in column one, with the maximum number of the same operator in column two for Tableau [66]



(a) Operator frequency in SDSS [35]

(b) Operator frequency in SQLShare [35]

Figure 3.5: Comparison of query operators of SDSS and SQLShare

The composition of read and write queries is also important because writing to a relation either invalidates caches built upon them or requires some kind of view maintenance algorithm to update the cache to the new values. In the Snowset, $\approx 59\%$ of the queries were read/write queries, with $\approx 28\%$ being read-only and only 13% being write-only queries. This composition is likely because many Extract, Transform, Load (ETL) jobs are being run, which can be indicated by the read/write queries reading roughly the same amount of data as they write [47]. In Redshift, read/write queries are less common, making up 39.2% of the queries, with read-only queries accounting for 48.9% [46] (see Table 3.5). While $\approx 40 - 60\%$ of queries might change data, we need more detailed information on the percentage of *relations* affected by these writes and the frequency of those to draw any conclusions on how the read/write ratio might affect caching.

There is more information about query patterns, out-of-core processing statistics, etc. However, this is not too relevant for caching, where query execution times, complexity, and the amount of caching data are the main factors to be considered.

Table 3.5: Query Types in [46]

Category	Query Type	Fleet
RO (48.9%)	Select	48.9%
RW (39.2%)	Insert	19.5%
	Copy	7.4%
	Delete	6.6%
	Update	3.8%
	CTAS	1.9%
Sys (11.9%)	Maint.	8.2%
	Other	3.7%

Key takeaways: Queries are often very complex, and relation updates are quite common.

3.1.4 Caching implications

Snowflake caches non-local network-attached data on the machines executing the query. This caching mechanism results in a cache-hit ratio between 60% and 80%, even though the cache size covers, on average, only 0.1% of the database size. This indicates "temporal and spatial skew in the workloads" [47] [67].

In "SQLShare," the authors reported that 90% of the queries were duplicates, making query result caching very useful. They then try and simulate the potential gains of sub-

result caching while excluding duplicate queries and conclude that 14% of the runtime could be saved. Additionally, they discovered that caching was either very beneficial for the query (more than 90%) or negligible (less than 10%) [35].

With most queries having fewer than a million rows, on-node caching of remote data like that employed by Snowflake would benefit Redshift [46]. While Redshift already uses a result cache, improving execution speeds for the many repeating queries ("80%), they only achieve a hit rate of $\approx 20\%$ across all queries, which can, among others, be attributed to the cache already being evicted when the repeating query is executed, or a mandatory cache miss, because the underlying data changed [46].

Key takeaways: Real-world query workloads have a lot of repeating queries and contain temporal and spatial skew, making caching very advantageous.

3.2 Database caching

The following section will introduce different caching concepts and compare them with each other. The main concepts are thereby:

1. **Query Recycling** referees to the concept of keeping the (sub-)results of previously executed queries around, thereby allowing for the later reuse of said cached results without having to re-execute the query the result is based on. (see [48] [61] [40] [31])

2. **Data caching** describes the process of caching the files/pages of databases (see [67])

3. **Caching in Hybrid Query Shipping systems:** Hybrid Query Shipping allows for the collaborative execution of one query on multiple databases, mostly one server database and one or more client/proxy databases. To improve query execution speed and increase client-side computation, the client database usually creates a cache, allowing it to compute the result (partially) on the client, thereby reducing communication with the server database. (see [24] [19] [2] [74] [4])

4. **Query caching** details the process of caching the result of a query/procedure or other database operation (see [58], Shim, Scheuermann, and Vingralek)

Each approach will be analyzed for the following metrics.

1. *What is cached* (e.g. pages, (sub-)queries)?

2. *How is the cache created?* For instance, is the cache a byproduct of query execution, or is the cache created separately?
3. *When and why are caches evicted*, studying the cache eviction strategies used.
4. *How is the cache used.* Is the paper using a declarative approach or a view-matching technique to leverage the created cache?

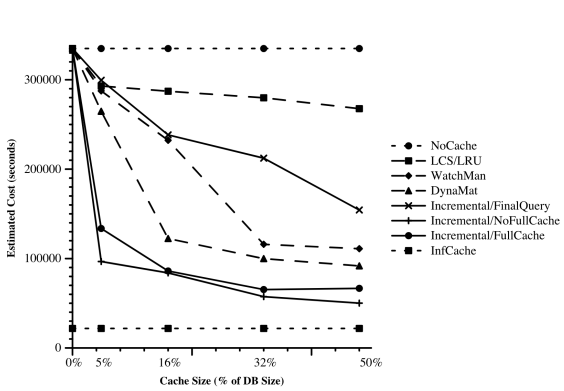
3.2.1 Query Recycling

3.2.1.1 Don't Trash your Intermediate Results, Cache 'em

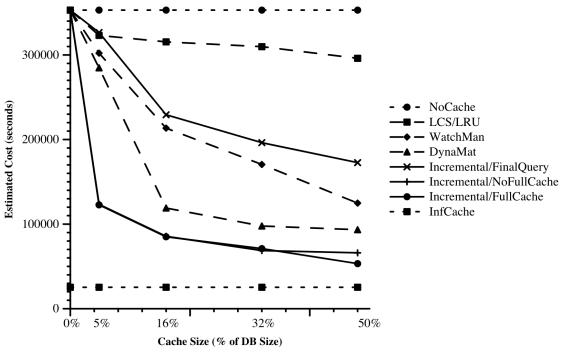
The paper by Roy et al. explores a query-recycling-based caching approach that optimizes queries based on historical queries to determine the best (sub-)query results to cache. Previously executed queries are represented as a Directed Acyclic Graph (DAG). These historical queries are subsequently used to decide if an (intermediate) result should be cached for a newly executed query. The decision which parts of a query to cache and which cached results to evict is made by their *Incremental Greedy Algorithm*. This algorithm iteratively picks the best query nodes for caching by repeatedly selecting the nodes that maximize the *benefit per unit space* metric. It terminates if the benefit falls below zero or the cache size is exceeded. Already cached nodes are replaced using a Largest Cached Size (LCS)/Least Recently Used (LRU) strategy, where LRU is used as a tie-breaker, which has also been used by Chen and Roussopoulos. Executing queries are also represented as a DAG. Their Volcano-based optimizer [26] then uses this to determine the optimal query plan considering both the cost of re-executing the query and reading from the cached result.

Results The evaluation was done using 1000 generated queries based on TPC-D. Then, different caching implementations were compared, starting with *no cache* and *infinite cache*, which employ no caching or can cache an infinite amount of data. These baselines are then compared to *Incremental/FinalQuery* (which only caches the final query result), *Incremental/NoFullCache* (which caches (sub-)results and tries to optimize the cost-benefit metric), *Incremental/FullCache* (which attempts to fill the cache as best as possible with (intermediate) results), as well as WatchMan [50] and DynaMat[39] two other caching systems.

The results for this can be seen in Figure 3.6 with general trends showing that *Incremental/FullCache* and *Incremental/NoFullCache* outperform all other algorithms, except *infinite cache*, while having one-tenth of the cache size.



(a) Performance for a workload where group by and selections are equally likely. Additionally, only equality predicates are used. [48]



(b) Performance of a Zipfian distributed workload where query patterns such as group bys and selections rotate. Additionally, only equality predicates are used. [48]

Figure 3.6: Performance results for [48] with the predicates restricted to equalities.

The time spent on optimization is negligible when compared with the execution time but increases the larger the cache size is and takes significantly longer than the standard Volcano optimizer [48].

3.2.1.2 Cache-On-Demand: Recycling with Certainty

In "Cache-on-demand"(COD), Tan, Goh, and Ooi implement a cache sharing algorithm, which allows new queries to reuse (partial) results of currently running queries. This works by first creating a normal plan for incoming queries and then processing the query plan to take potential caches into account. If the incoming query should now reuse the result of a running query, the running query will create a cache.

For the second optimization step, Tan, Goh, and Ooi propose two implementations, *Conform-COD*, which keeps the order of the plan of incoming queries, and *Scramble-COD*, which allows the optimizer to reorder the query plan to gain cache matches.

Results Both Conform-COD and Scramble-COD perform better than having no cache, with the conform variant reducing execution time by 20% and the scramble variant by 45%. That being said, their system's performance depends entirely on queries that are executed simultaneously and require the same or similar data.

3.2.1.3 Recycling in pipelined query evaluation

In [40] Nagel, Boncz, and Viglas investigate the viability of building a query result recycling system for Vectorwise [77], a vectorized query execution engine which does not materialized

intermediate results as a byproduct of query execution. Intermediate results are cached if their benefit outweighs the costs captured in the cost-benefit metric. This is done by the recycler, which also tracks intermediate results in the recycler tree. The cost-benefit metric comprises the materialization cost, the reuse potential, and the potential savings resulting from cache reuse. Historical data is used to estimate potential future gains, falling back on speculation if no historical data is available. This is paired with statistics about previous results stored in the recycler tree. When the cache space runs low, results with a lower benefit are evicted, assuming their eviction creates enough space for the new result.

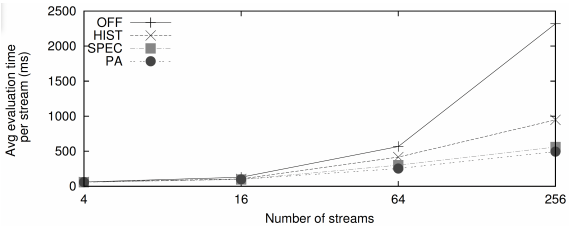


Figure 3.7: Average time per TPC-H stream. *OFF* represents no caching, while *HIST* decides whether to cache based on historical queries. In contrast, *SPEC* improves on this by additionally caching smaller results anticipating later reuse. Meanwhile, *PA* represents a proactive strategy that executes an initially more expensive query for later gains. [40]

Results Throughout all investigated scales (4 to 256 query streams), the authors saw improvements ranging from 10% up to 79%, favoring more query streams (see Figure 3.7). At the same time, matching costs stayed low and are orders of magnitude lower than the execution times, showing only moderate growth for larger recycling graphs.

3.2.1.4 Caching & Reuse of Subresults across Queries

In their blog, Hall built a custom caching solution[31] for Firebolt [42], a ClickHouse-based data warehouse. They support caching of (sub-)results and hash tables by inserting a *MaybeCache* operator. When executing a query, the *cache fingerprint* (derived from the cached plan) is compared with the query plan of the executing query. For matching fingerprints, the cache is reused. Cache invalidation is handled by including table IDs in the hash. Table IDs change when updating a table, thereby making the cache inaccessible. The cache eviction strategy removes cache entries when running out of memory based on the LRU related strategy proposed by [40], which includes the time needed to calculate a result, the access frequency and its size.

Results They evaluated the performance of their caching solution on real-world workloads, reporting a savings of 10x for caching hash tables and varying performance improvements for their top 15 customers ranging from $< 5\%$ to up to 95% for full query caching.

3.2.2 Caching in Hybrid Query Shipping systems

3.2.2.1 Performance tradeoffs for client-server query processing

While caching was not the primary research goal of Franklin, Jónsson, and Kossmann when investigating hybrid query shipping systems, it plays a crucial role in its performance evaluation, comparing *data-shipping* (client DB retrieves data from the server and executes queries locally), *query-shipping* (queries are sent to the server, which executes them and sends the result back) and *hybrid-shipping* (utilize the processing power of server and client at the same time by executing queries both on the server and the client), which is done by executing queries in a simulated environment, representing a peer-to-peer database system [24].

They do not detail how relations are cached nor provide insights into cache eviction or maintenance strategies. However, the results showcase the importance of locally cached data for hybrid environments (see Figure 3.8).

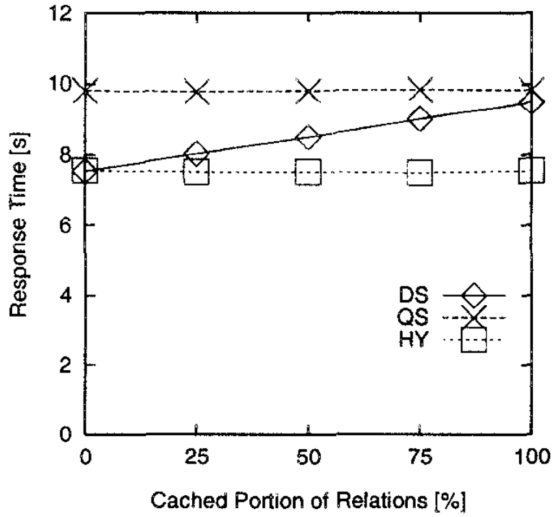


Figure 3.8: Execution speed on the Y-Axis with the number of cached relations on the X-Axis. [24]

Results *Query-shipping* suffers from disk contention, resulting in higher response times, while *data-shipping* starts better but runs into the same disk contention problems as

query-shipping. On the other hand, hybrid shipping utilizes the locally created cache to balance the disk reads between the two database systems, thereby reducing I/O on both systems. While I/O in modern systems can still be a bottleneck, it is nowhere near as pronounced as for [24].

3.2.2.2 Caching Multidimensional Queries Using Chunks

The main idea of Deshpande et al. is to not cache query results but Chunks in the Paradise DBMS [20]. The system used by the authors is implemented as a middle-tier architecture with the query coordinator and cache manager acting as a proxy for the backend DBMS. Chunks are multidimensional representations of relations (see Figure 3.9), allowing for more granular cache management than query results or database pages. Each query is split into several cachable Chunks. Subsequent queries are also divided into Chunks before execution to determine which Chunk can be read from the cache and which Chunk has to be re-computed by the backend system. The on-disk relations also use a chunk-based storage format, providing the additional benefit of speeding up multidimensional clustering operations. Cached Chunks are evicted based on a benefit metric, which is measured by benefit $= \frac{|D|}{n}$ where $|D|$ denotes the size of the base table and n the number of Chunks returned by a, for instance, `GROUP BY` operation. The benefit metric is then combined with the *ClockBenefit* algorithm, which can be described as a benefit-based, LRU algorithm more inclined to keep newer, more beneficial Chunks in memory. While the Chunk selection algorithm is not detailed in this paper, I assume they also use the ClockBenefit algorithm to select potential Chunks to cache.

Results A custom benchmark is used to evaluate the effectiveness of Semantic Chunk-based caching, where queries are executed on uniformly distributed data. Additionally, the database has *hot regions*, which are queried more frequently. The executed queries display adjacency, meaning they are interested in the same aggregations but select different predicates. Deshpande et al. conclude that their semantic Chunk caching strategy performs better than tuple and page caching, which could not reuse partial (aggregate) results. They discovered that while smaller Chunk sizes result in better caching, significant overhead is associated with having more but smaller Chunks. Finally, their benefit-based-LRU cache eviction strategy performed better than a simple LRU strategy.

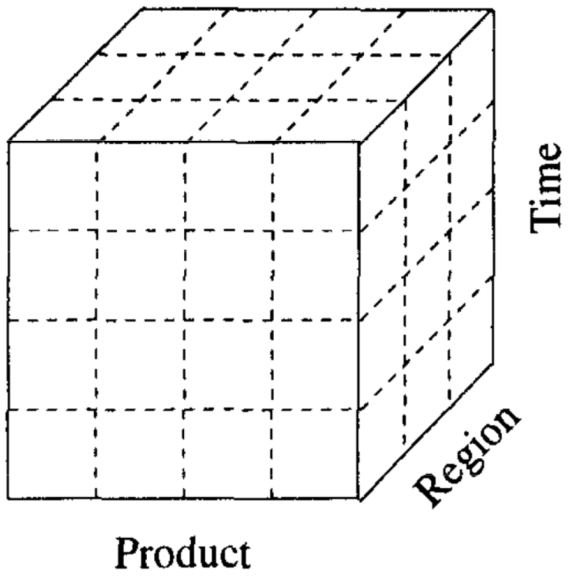


Figure 3.9: The chunked relations *Product*, *Region*, and *Time*. [19].

3.2.2.3 Cache Tables: Paving the Way for an Adaptive Database Cache

Altınel et al. create a DB2 [16] extension intended to speed up queries of Transactional Web Applications[2]. Applications connect to one of the middle-tier edge databases, which either answer the query using their cache or forward it to the backend system. The decision of where to execute the query is made by first probing the cache table for the required data and then executing the query locally or remotely, depending on the availability of the data. The data underlying the cache tables is kept up-to-date by reusing existing logic for materialized views[76] in DB2. Caches can either be created manually using `CREATE CACHE TABLE AS <cache_name> FROM <table_name> WHERE <predicate>` or automatically by creating a cache table without predicate and letting the optimizer decide which part of the table to cache.

Results Altınel et al. mainly investigated the introduced overhead. This was done using IBM's Trade2 J2EE Benchmark [34], modeling an online web service. They conclude that their caching implementation introduces only minimal overhead, with the benefits of increased response times and higher availability outweighing the downsides of introducing additional overhead.

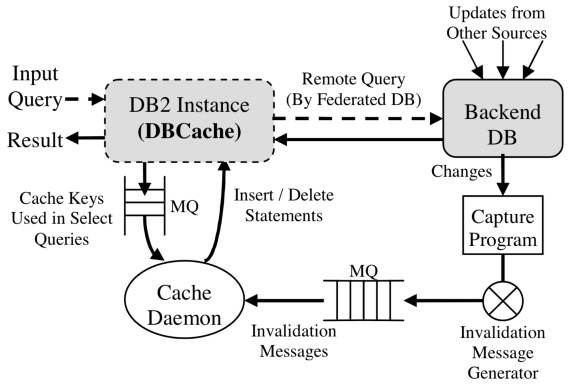


Figure 3.10: Operation of a federated DB2 working as a cache for the backend DB [2]

3.2.2.4 FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS

FlexPushdownDB (FPDB) [74] is a hybrid DBMS designed to use *caching* and *computational pushdown* in conjunction. Caching base-table data on the client side enables a new optimization. Namely, for each query experiencing cache misses, the optimizer decides if a client-side table cache should be created. If not, the computation is pushed down to the server. If the caching policy chooses to create a table cache, FPDB has two options: (1) wait for the cache to be loaded before executing the query or push down the computation. The Cache manager decides which tables to cache and evict by collecting metadata like the access frequency and the pushdown cost. The pushdown cost is the cost that is associated with having to execute a query on the segments A and B. If A is always accessed using a heavily filtering predicate and B is not, evicting A makes less sense, as A can benefit from *computational pushdown*. By weighing access frequency with this metric, they created a Least Frequently Used (LFU)-based algorithm evicting the segments with the lowest benefit.

Results Yang et al. use a Star Schema Benchmark (SSB) [41] based benchmark (SF=100) in addition to a random query generator, generating SSB compliant queries. Several benchmarks were run with skewed access patterns, resulting in some records being accessed more often.

The results can be seen in Figure 3.11 with the best strategy being the *hybrid* one. More experiments on the efficiency of the different solutions have been conducted, investigating different cache replacement strategies and the influence of network bandwidth, which showed that the purely caching-based strategy heavily benefits from higher connection speeds.

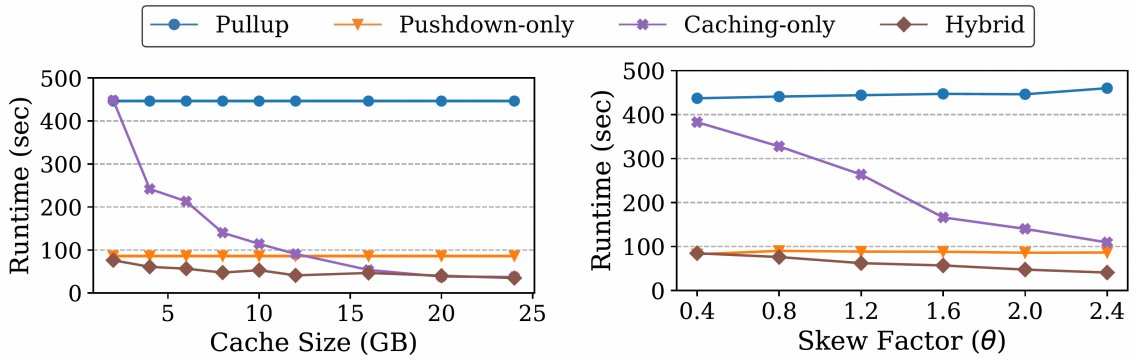


Figure 3.11: Performance comparison on an EC2 c5a.8xlarge. The cache size varies on the left, while the skewness varies on the right [74].

3.2.2.5 MotherDuck: DuckDB in the cloud and in the client

In "MotherDuck: DuckDB in the cloud and in the client," the authors introduce a new data warehouse built on top of DuckDB, an embedded analytical DBMS, where each distribution comes with a client-side, as well as a server-side DuckDB instance. MotherDuck [4] utilizes *hybrid query processing* to run queries on local and remote data using both the client's and the server's processing power. To speed up their DuckDB-Wasm based UI, they introduce a query result cache, allowing them to run follow-up queries on the previously cached result. The caching is done declaratively using standard SQL features to create an in-memory cache table of a limited size. Because the approach is in-SQL, MotherDuck does not do any query rewriting to reduce execution time using local caches. Instead, queries must explicitly specify the cache they want to read from. Because the cache is size-limited, query results can also be partially cached. A partial cache prevents follow-up queries from using the result cache, as the correctness cannot be guaranteed. Consequently, the cache is bypassed when it is incomplete.

Unfortunately, the effectiveness of their in-SQL query caching solution is not evaluated.

3.2.3 Data caching

3.2.3.1 Building an elastic query engine on disaggregated storage

Snowflake [14], a cloud-based data warehouse, uses multiple layers of storage when executing a query. First, the local memory is exhausted, resulting in the worker spilling to a local solid-state drive (SSD) disk until the storage space of the SSD is also running out, resulting in spilling to S3 [67].

Because in Snowflake, the DBMS workers themselves do not have any data locally, all data is initially fetched from remote storage. When executing a query, the worker decides if the remote data should be cached locally. Each worker has a cache composed of remote file headers and individual columns, as queries download only the required columns [14]. Data is evicted using the LRU replacement policy when caching space runs low.

Results While neither paper reports the performance of caching while executing queries, they report a cache hit rate of almost 80% for read-only queries and 60% for read-write queries [67].

3.2.4 Query Caching

3.2.4.1 Extending a Database System with Procedures

This work extends Ingress [59] by adding support for database procedures inside relation fields. Database procedures in this context are a collection of Query Language (QUEL)[32] statements. When accessing the field the procedure is stored in the procedure's result is returned. Caching comes in two forms: (1) The caching of query plans of database procedures and (2) the caching of the query results of the database procedures [59]. Only (2) is relevant in this context.

For the caching of database procedures Stonebraker et al. propose two different strategies for result caching. Smaller objects are cached directly in the defining fields, while larger relations are cached in external relations. Large objects are discarded in accordance LRU when space runs out. While pointer-based storage of database objects (store pointers to the tuples containing the data instead of copying the data) was considered for caching, this was discarded because pointer-based storage comes with a larger read-overhead. This design decision favors read-heavy workloads with infrequently caching tuples.

Cache invalidation is envisioned using locks, where each result sets a lock on the objects it originates from. Once the underlying tuples are modified, the lock is lifted, invalidating the cache. The performance of the cache is not evaluated.

3.2.4.2 Dynamic Caching of Query Results for Decision Support Systems

Shim, Scheuermann, and Vingralek create a caching system that caches query results. Admissions for new query results are granted if the result in question increases the overall profit of the system. The benefit metric comprises "its average rate of reference, its average update rate, its size and the benefit of cost saving for the processing of associated queries"

[52]. This benefit metric is used for cache-admission and eviction alike, with evictions using LNC-R (Least Normalized Cost Replacement) [50], which sorts cache entries in order of profit and then selects eviction candidates.

To decide if a new query can use a cache entry, the cache manager first analyzes if the query is in the form of a *canonical query*, which allows it to quickly determine if a cache entry has the required data using *query split* algorithm. If the query is not in a canonical form, the cache manager looks for an exact query match.

Results The performance of their caching implementation was evaluated using a TPC-D-like workload, and experiments showed an on average pf37% to 70% (skewed workloads) performance improvements.

4

Implementing Accelerated Approximate Views in DuckDB

DuckDB and, by extension, MotherDuck are particularly good for building data apps for the browser. Data apps are applications that use data to create insights, which are typically represented as (interactive) graphs or tables. DuckDB's ability to run in the browser makes it stand out by enabling on-device data analysis and manipulation, thereby removing network latency and improving responsiveness.

However, local data analysis is only possible if data is available locally. Currently, DuckDB does not support storing data in the browser using OPFS or *IndexedDB* - the two browser native solutions for storing larger amounts of data, making it harder to keep data on the client. To fully utilize MotherDuck's hybrid architecture, a mechanism for on-device data caching will be created to reduce computational overhead and increase data locality.

The approach of this thesis is to create an *approximate accelerated view*, which enables partial local caching of view data without maintaining the view data. Keeping materialized view data local can reduce response times, as queries accessing the view can utilize the views cache instead of having to recompute the view. This also benefits hybrid queries as the network overhead of querying remote data can be avoided. Additionally, having low-level control over the cached data enables more granular memory management, enabling automatic cache evictions when the memory is low. One differentiating factor to unmaintained materialized views is that an *approximate accelerated view* enables access to partially cached view data while it is still being built.

The following requirements have been defined for accelerated views.

REQ1 Accelerated views are created by users using an intuitive standard SQL-like syntax.

REQ2 Accelerated views, after creation, automatically populate a cache in the background.

REQ3 Cache population should be done with minimal impact on other queries. This is especially important in the Wasm environment where only one thread is available.

REQ4 An accelerated view should optimize execution times by dynamically deciding whether to use the cache or the view's query to answer any query accessing the accelerated view.

The structure of this chapter is thereby as follows. We begin by explaining the user-facing mechanics of an accelerated view in Section 4.1. After this, the lifecycle of the accelerated view is explored, starting by covering the SQL extension necessary to satisfy **REQ1** in Section 4.2, followed by Section 4.3 which goes into details on how an accelerated view is created, accessed and deleted, covering **REQ2**, **REQ3** and **REQ4**.

4.1 Mechanics of Accelerated Approximate Views

One key concern when creating an accelerated view is that other running queries cannot be disrupted. This is especially important for Wasm as Wasm is single-threaded, making it essential that the background cache creation does not interfere too much with other running queries.

The resulting mechanics of an accelerated approximate view can be seen in Figure 4.1. We start by creating the accelerated view by executing a `CREATE RESULT my_result AS ...` statement, which triggers the building of its cache in the background. Later, when accessing the view, one of three scenarios occurs:

1. **Sufficient Cache:** If the cache is large enough, the cached result is accessed
2. **Pending Cache:** If it can be anticipated that the cache will reach a sufficient size, access to the cache is delayed until it contains enough data.
3. **Insufficient Cache:** If it cannot be guaranteed that the cache can contain the entirety of the accelerated view, the cache is bypassed. Instead, the query used to create the accelerated view is executed as a subquery within the query accessing

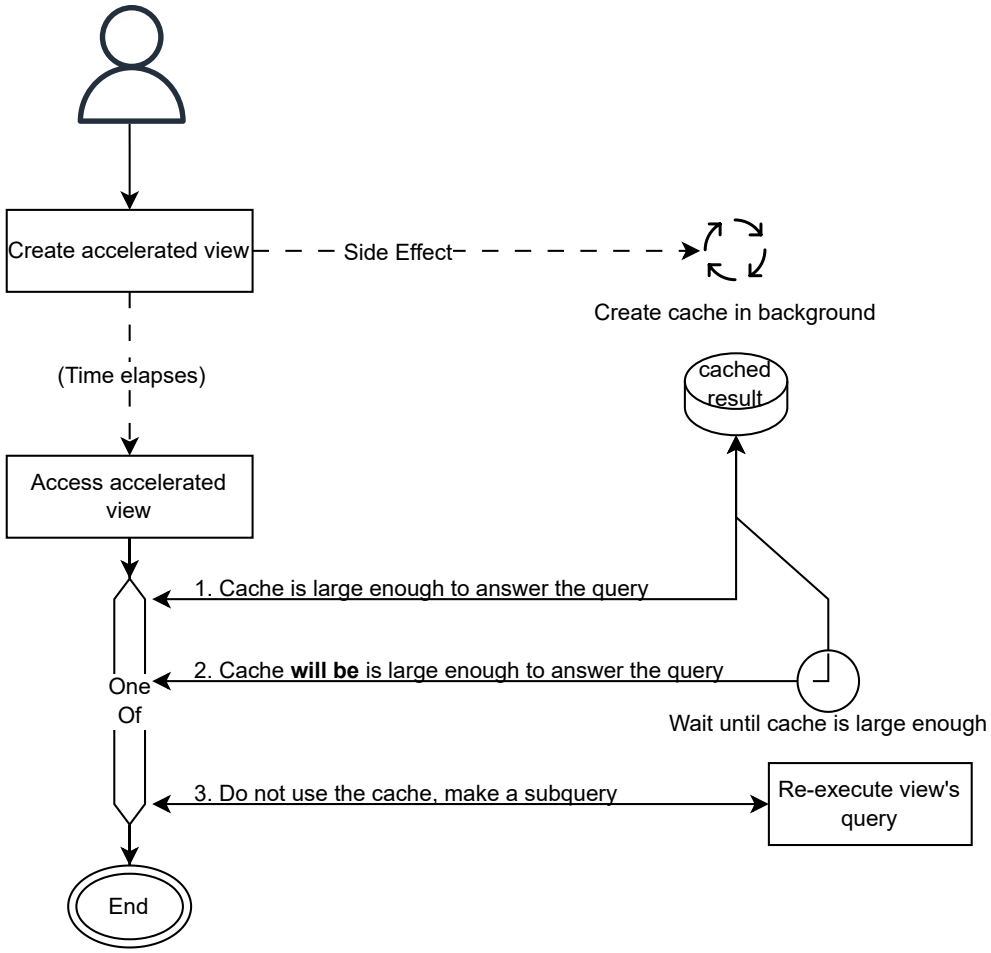


Figure 4.1: Interaction with the accelerated view

the view. In the later sections, this will be called *query inlining*. In a scenario like this, an accelerated view acts like a standard view. Notable here is that the view cache is still created; however, at some point, cache creation will be halted because of resource restrictions.

The decision tree in Figure 4.2 is followed to determine which of the three states the cache is in. If the cache contains enough tuples to satisfy the query accessing the view, the cache's data is read instead of inlining the original query. The cache can still be in the building stage for this to be the case, if the relevant data for the query accessing the view has already been inserted.

If the necessary amount of data has not been inserted into the cache, one can either wait for more tuples to be inserted into the cache or inline the view query. This is where the concept of *guaranteed satisfiability* comes into play, which, if satisfied, makes it safe to wait for the cache to accumulate more tuples.

Definition 1. A query accessing an accelerated view is only **guaranteed satisfiable** if it states how many tuples it is interested in and if the amount of tuples it wants to read from the view is below a certain threshold. This is done by applying a numerical limit to the view and would be `LIMIT foo` for Figure 4.2. Only queries that specify a limit below a certain threshold can be guaranteed to be completed by only reading data from the cache after a waiting period. This is because queries that exceed that limit might require a cache larger than the available resources to satisfy their limit. Therefore, queries without limits or limits that are too large are inlined. Additionally, no aggregations, sorting, etc., are allowed to be directly applied to the relation of the accelerated view, as this would make it impossible to judge whether the required data has already been cached.

4.2 Extending the SQL Syntax for Accelerated Approximate Views

One of the requirements (see **REQ1**) specifies that the SQL used for the creation of the "accelerated view" should be intuitive to use. Since accelerated views behave similarly to standard views, its SQL API should also be similar or inspired by the caching behavior of other data warehouses.

Snowflake's approach to caching queries is very implicit, as it automatically caches the result of every query for 24 hours. The cached result can be accessed using the unique query ID with `RESULT_SCAN('query_id');` [55]. However, this caching method is not ideal for our needs, as every query is cached without providing an opt-out mechanism. Moreover, it requires the user to remember a random query ID, making the cache retrieval process unintuitive.

Denodo offers a method that closely resembles the approximate view concept discussed in this thesis. They use an associated partial cache to verify at runtime if the data in the cache can answer the query accessing the view. However, Denodo does not determine whether a view should have a cache associated with it during its creation. Instead, view caching is enabled using a UI setting [18]. Enabling caching using a UI setting is impractical for MotherDuck, as DuckDB is not a managed service, and not every DuckDB API has an associated UI. Therefore, creating an accelerated approximate view with only SQL

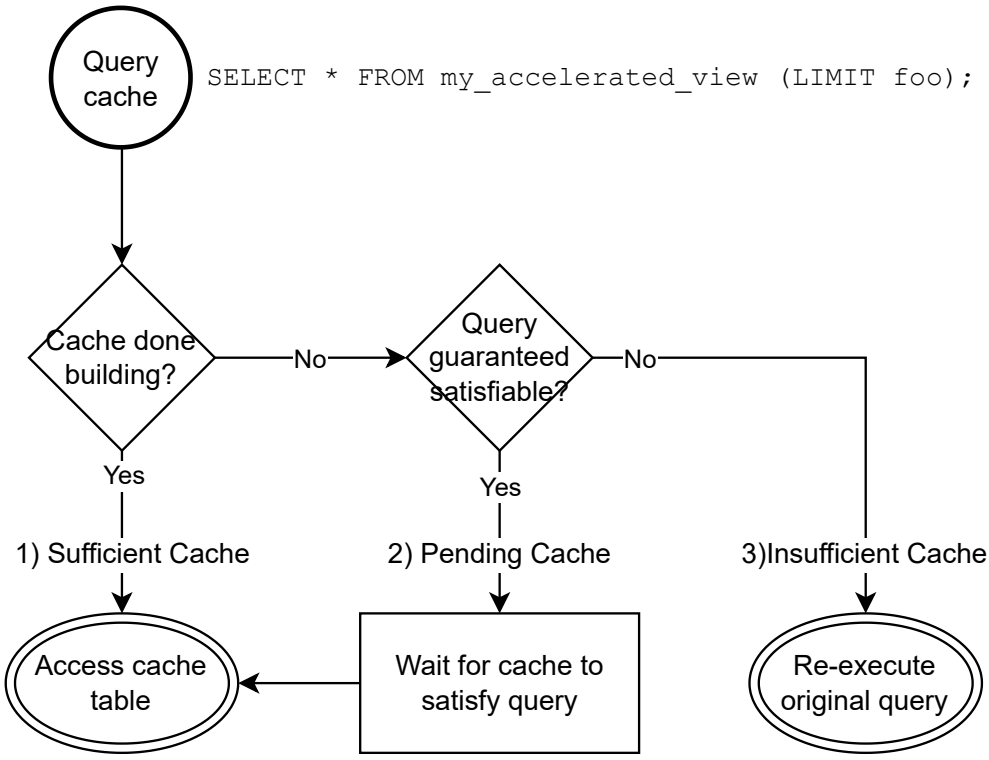


Figure 4.2: Decision tree used to determine whether the cache should be used to answer the query accessing the view.

must be possible. Additionally, making a standard view return outdated results poses the risk of users not realizing that the accelerated approximate view might contain outdated information.

The accelerated approximate view shares some commonalities with a materialized view `CREATE MATERIALIZED VIEW my_view AS SELECT ...` ; namely, an associated cache used to answer queries accessing the view. Therefore, it is logical to model the creation of an accelerated view similarly to the approach used for materialized views, resulting in the syntax `CREATE CACHED VIEW my_view AS SELECT ...` . However, this syntax fails to communicate that the cached information might be outdated, and in fact, the cache is merely a *result* of the view query.

The following syntax is proposed to communicate this problem better: `CREATE RESULT my_result as SELECT ...` . It closely aligns with the creation of views but still differs to indicate that other than views, this concept does not guarantee up-to-date information but contains the result of a previous query. The final syntax for creating and managing an AAV can be seen in Listing 4.1.

```
1  -- Create the cache from a query (optional OR REPLACE, IF NOT EXISTS)
2  CREATE RESULT my_result AS SELECT foo FROM bar WHERE bar.a > 42;
3
4  -- There are different access methods. In this case, the cache is used as
       is, even if it is not fully materialized
5  SELECT * FROM MD_RESULT(my_result, incomplete=true) WHERE foo > 10;
6
7  -- This uses the cache if there are more than 100 results in the cache;
       otherwise, it will block until 100 tuples have been materialized
8  SELECT * FROM my_result LIMIT 100;
9
10 -- Pause cache building
11 PAUSE RESULT my_result;
12
13 -- Continue cache building and start where the building has been paused
       before
14 CONTINUE RESULT my_result;
15
16 -- Delete the cache (optional IF NOT EXISTS)
17 DROP RESULT my_result;
18
19 -- Get statistics about the cache
20 FROM md_results() SELECT name, status, row_count, creation_time, name WHERE
       name = 'my_result';
21 SHOW ALL RESULTS;
22
23 -- Change the path of the result catalog. By default :memory:
```

```
SET result_cache_path TO 'persisted_results.ddb';

-- Change the maximal combined size results are allowed to have
SET combined_result_size TO '1KiB';

-- Set a soft limit for a result's size.
-- For this configuration, results will never contain more than the first
    ~4321 tuples (+maximal one vector of 2048 elements).
SET result_row_count_soft_limit TO 4321;

-- Only relevant for wasm
-- Set the time spent on building the cache to 10ms per building interval;
SET result_building_time_ms TO 10;

-- Only relevant for wasm
-- Try to build the cache every 100ms
SET result_building_interval_ms TO 100;
```

Listing 4.1: Result syntax

4.3 Managing Accelerated Views

This section outlines the complete lifecycle of an accelerated approximate view. It begins with parsing the proposed SQL syntax, followed by a section on how an accelerated view is created and the cache is built in the background. Next, the process of accessing the view is examined. Finally, we briefly overview other operations, including deleting the view, pausing, and resuming cache building.

4.3.1 Cache Creation

This section covers the creation of the accelerated view through the following steps:

1. Parse the custom SQL syntax created for accelerated views using a DuckDB parser extension

2. Store and persist the cache and its metadata in a custom catalog

3. Create a mechanism that periodically executes the cache-creation query in the background without interfering with other running queries too much.

4. Design a system reducing access time to cached data.

We start by creating a new accelerated approximate view using the SQL statement in Listing 4.2. The first query processing step involves parsing the custom SQL. In DuckDB, this can be done by registering custom parsers which allow extensions to expand on the DuckDB SQL. Parser extensions are expected to return table functions, which in turn return relations. These table functions then implement the custom syntax. The query in Listing 4.2 will be parsed, and the table function `MD_CREATE_RESULT` in Listing 4.3 is returned. This table function will subsequently be bound and executed. During the execution phase, the `MD_CREATE_RESULT` table function will create the data structures necessary to store the cache and the accelerated view metadata and trigger the cache creation process.

```
CREATE RESULT my_result AS SELECT * FROM my_relation;
```

Listing 4.2: Creation of a approximate accelerated view called my_result

```
CALL MD_CREATE_RESULT('my_result', 'SELECT * FROM my_relation');
```

Listing 4.3: Parsed representation of Listing 4.2

4.3.1.1 Storing Cache (meta-)data

DuckDB stores *views* in its catalog. When queries access a view, the reference to the view is replaced with the query used to define the view during binding (see Section 2.4.3.1). This replacement mechanism makes it difficult to detect if a query accesses a view in an optimizer extension since DuckDB executes this replacement during binding before extensions optimize the logical query plan. This makes augmenting standard views with caching impractical, so we introduced **results** as a new concept instead. The AAV's metadata could be, similar to standard views, stored in a DuckDB catalog; however, to simplify the design, it is for now stored outside DuckDB.

After parsing the `CREATE RESULT` statement and storing the accelerated views metadata, the next decision is determining how and where to store the cache. Two main requirements have to be satisfied by the storage location:

1. The cache should be capable of being persisted on disk so that it can be re-loaded after a client-side database restart. This is important as a DuckDB database might be short-lived because it depends on its parent progress.

2. The storage location should be resilient to failures by, for instance, being transaction-protected. At the same time, however, a transaction should be able to read tuples

from the cache that materialize after its transaction has started. This is important to support waiting for new data.

We evaluated three different options:

1. Store the cache independent of DuckDB in a **custom in-memory structure**. One significant benefit of this approach is its independence from DuckDB transactions. As a result, accessing the latest cached information is always possible. However, the cache is not guaranteed to be correct, as a failed update could corrupt it. In addition, spilling to disk and other operations, like reading/writing, must be implemented from scratch.

2. **Temporary Tables** are in-memory tables that can spill to disk. However, they are only visible to the connection that created them. As DuckDB manages temporary tables, they are protected by transactions, which helps to prevent data corruption. However, this transactional protection complicates accessing the most recent cached data.

3. DuckDB allows users to attach multiple catalogs, also called databases, simultaneously. These catalogs can, amongst others, be file-based or in-memory. One alternative to using temporary tables for storing the cache is to create a new **catalog** specifically for the tables containing cached data. Although this approach addresses one of the limitations of temp tables by being readable from all connections while adding the option for persistent storage, transaction mechanics are still the same.

Temporary tables are not a viable option due to two aspects of DuckDB. Firstly, DuckDB can only execute one query per connection at any time. Secondly, a temporary table can only be accessed by a single connection. Hence, no other connection can be used to build the cache in the background. However, building the cache on the same connection as the `MD_CREATE_RESULT` statement was issued violates **REQ3**, which states that other queries should not be disrupted when building the cache.

Bypassing DuckDB's transaction mechanics by storing the cache data externally adds significant challenges. For one, this would require implementing data persistence, integrity, and an API for DuckDB to read the external data, diminishing the practicality of this approach. In contrast, storing the data within a custom catalog only presents the challenge of managing transaction semantics. Therefore, we decided to store the cached data in their own relation (similar to [58]) in a custom catalog.

In addition to storing the cache data, metadata must be collected to answer queries accessing the accelerated view. This includes the **view name**, the **query** used to create the view (allowing the cache to be re-created or bypassed), and the **status of the view**, which determines if the cache has finished building. Additionally, the **time** the cache creation query was executed should be saved. This allows for the cache evictions for older caches. Moreover, the **row count** of the cache should be stored, as it is used to determine if a query can be answered using only the cache.

A subset of this metadata must be persisted whenever it changes to allow accelerated views to be loaded from disk. Since the row count changes frequently, persisting this information to disk would introduce a high overhead, making saving it impractical. Instead, the row count can be restored when loading the cache from disk, as the cached data is stored in a DuckDB table, which also contains metadata describing the number of tuples in the table. Since the cache data is stored in its own DuckDB catalog, storing the cache metadata within the same catalog is logical.

4.3.1.2 Background Cache Creation

REQ3 specifies that the view cache must be populated with minimal impact on other queries. In addition, **REQ2** requires that the cache creation happens in the background, thereby freeing the connection used to create the accelerated view. In Figure 4.3, the high-level process for the cache creation can be seen. The following chapter will go into the last step of the diagram, the execution of the query, whose result will ultimately be used as the cache for the accelerated view.

Regardless of how the cache is built, it cannot continuously use the calling thread, which would block it from running other queries. In multi-threaded systems, this can be achieved by creating a background thread or utilizing the DuckDB scheduler's thread pool. However, Wasm presents a challenge due to its single-threaded nature.

DuckDB is compiled to Wasm using the Emscripten toolchain, which provides an API that allows C++ to call into JavaScript. One of the APIs provided by Emscripten allows a Wasm function to be executed after a specified delay. This mechanism works by putting a task in the JavaScript event loop once the timeout has passed [22]. The scheduled task is executed once all previously queued tasks have been run. Interactivity is maintained as long as no task, including the queued one, blocks the JavaScript event loop for too long. If the queued function reschedules *itself* using the same API, it can periodically be executed without blocking the JavaScript event loop. We utilize this mechanism by periodically scheduling a background task that builds the cache in the background without disturbing

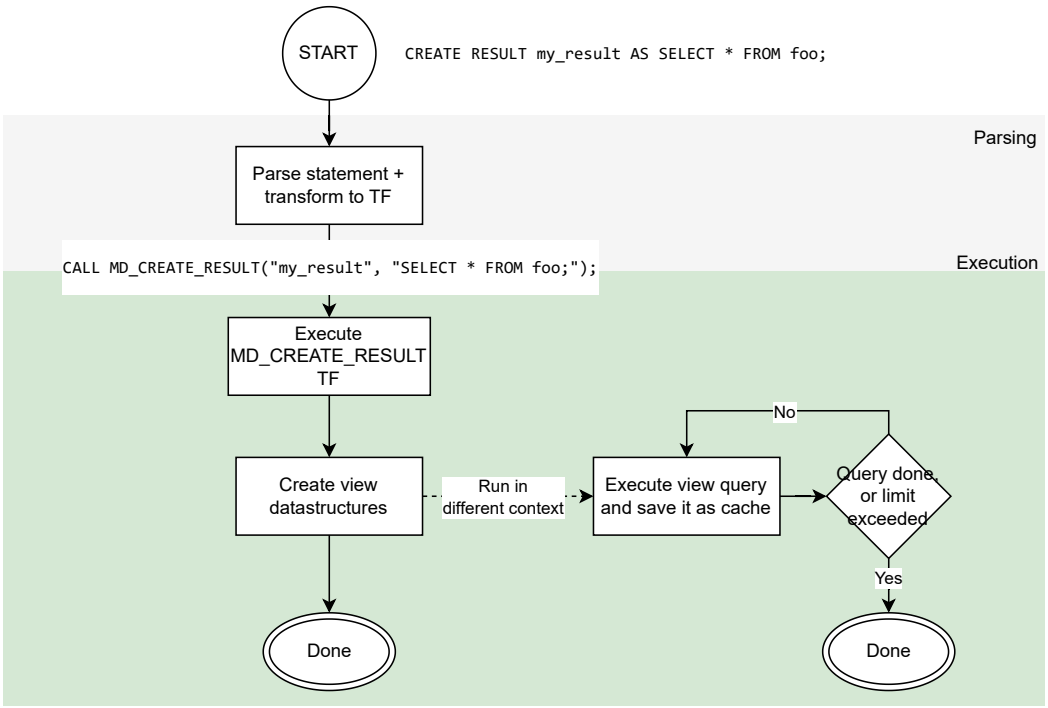


Figure 4.3: Overview of the actions executed when creating the cache.

other queries. The goal is to preserve the same programming model in multi-threaded environments like C++ to reduce the complexity that having multiple execution models would introduce. Therefore, in addition to the thread pool, which (partially) executes every query in multi-threaded environments, a background execution mechanism, built on top of the DuckDB thread pool, has been created to simulate the Wasm execution model.

Now that the execution process has been defined, the next step is determining how the query should be executed in smaller parts. This has to be done to minimize extended execution time and prevent the background function from taking too much time from other queries. Ultimately, I created a custom execution driver and result collector). In the following, I will explain why working with streaming queries is not viable and what advantages the final design has.

Streaming Queries To execute a query in smaller parts, one can use streaming queries. Unlike *materialized* queries, which execute the entire query in one go, streaming queries pause execution once a certain amount of result data has been buffered in the streaming queries result collector (see Section 2.4.3.6). Subsequently, query execution resumes only when the result collector buffer has been depleted.

However, a notable drawback is that a lot of time may pass until the buffer of the result collector has been filled, as all physical operators within the query have to be partially executed for this to happen, potentially leading to extended execution times. This contradicts the requirement that building the cache should occur in the background without disturbing other running queries.

Another problem is that streaming queries (see Section 2.4.3.6) busy spin if any of the pipelines block two or more times [1]. As a result, any hybrid queries, which always have potentially "blocking" physical operators, will inevitably block and stay blocked in single-threaded environments. Therefore, slowly materializing results using streaming queries is not possible in DuckDB. The same is true for every other way to consume a query result in a "standard" DuckDB way, as each of its APIs is suffering from this problem.

DuckDB works around these limitations in Wasm by not using streaming queries or materializing queries, but instead the `PendingQuery` API (see Section 2.4.3.7). This API gives the caller explicit control over when to execute a task. In contrast to streaming queries, a pending query in DuckDB Wasm does not execute a blocked pipeline continuously but instead yields to the caller. DuckDB uses the pending query API to execute a query in bursts of, by default, 100ms. After that time, DuckDB Wasm yields control to the caller, the JavaScript function calling the native DuckDB code. This is done repeatedly to advance the query process.

Therefore, to work in Wasm, background materialization must imitate the DuckDB Wasm behavior.

Custom Execution Driver and Result Collector The behavior of DuckDB's Wasm can be emulated by creating a pending query while the view data structures are created. This pending query is subsequently advanced at intervals via the background function mechanism described earlier and works both in Wasm and C++ . Two options are available to consume the generated data: Create a query result object from the pending query or

[1] `https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/main/buffered_data/simple_buffered_data.cpp#L58-L65`

create a custom result collector allowing us to access data chunks while they are being produced.

Creating a `QueryResult` from a pending query is impossible, as this forces the pending query to be executed in one go, causing the query to *busy spin* if any of its pipelines block. To avoid this, the pending query can be driven to completion before a consumable query result is created. However, this approach delays access to the first data of the query until the entire query has finished running.

The method that was ultimately chosen to retrieve the query data is to create a custom result collector, which is called every time the query produces new data in combination with the background function to advance the query progress by executing one task at a time, as long as the query's pipelines are not blocked.

The data can then be stored in the previously created cache table using the DuckDB Appender, a single-threaded utility for appending data to tables. This approach prevents busy spinning and provides quicker access to a query's initial results by eliminating any result collector buffers.

4.3.1.3 Wasm-specific scheduling problems

The scheduling of query-related tasks in single-threaded environments significantly differs from that in multi-threaded environments, where you have multiple execution drivers, namely the DuckDB thread pool and the thread that issued the query. While the thread pool will execute tasks independent of their query, the query thread will only execute tasks related to its query. In single-threaded environments, the thread that issued a query is solely responsible for its execution. However, this time, the query thread does not differentiate between tasks associated with the query and tasks left in the task scheduler by different queries. As a result, the query thread will also execute completely unrelated tasks. For instance, when executing a query such as `CREATE RESULT my_result AS FROM so⌋ me_table WHERE a > 1;` , the query thread begins by parsing the query and subsequently adds the query tasks to the DuckDB task scheduler. Assuming the task scheduler's queue was initially empty, only tasks associated with this query will be in the queue. The tasks of this query will subsequently be executed until the `MD_CREATE_RESULT` table function is called. At that point, a pending query will be created, intended to be executed in the background to fill the cache. Even though creating a pending query will not execute the cache creation query, tasks associated with the query will still be put in the task scheduler. After making a pending query, the table function returns, and the thread executing the `CREATE RESULT` query executes the next task in the task queue.

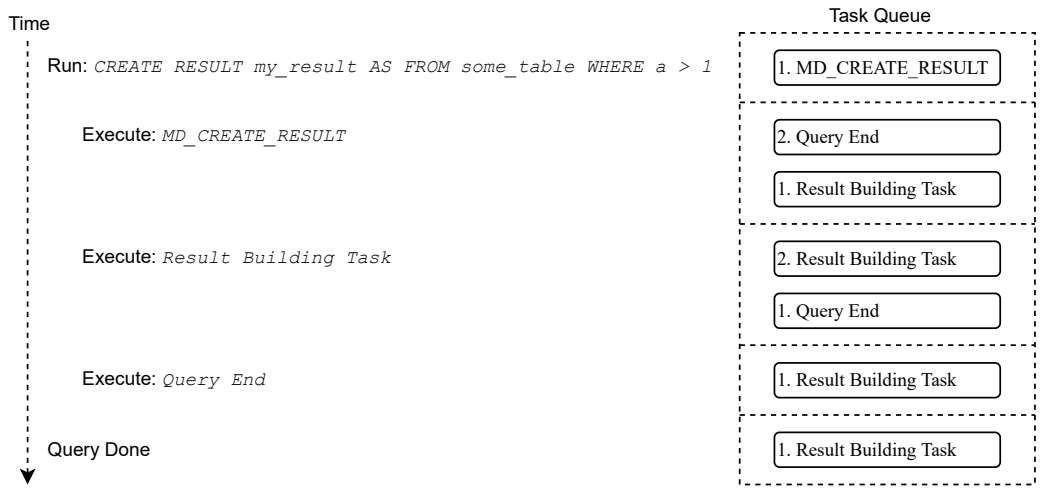


Figure 4.4: Task execution order in Wasm. Because the query thread does not distinguish between query-related and unrelated tasks, tasks associated with other queries might also be executed by the query thread.

At this point, the next task may not be related to the query, which creates the AAV. Instead, it could be associated with the query supposedly building the cache in the background, as illustrated in Figure 4.4. Because the scheduler does not differentiate the tasks in the queue, the task associated with the cache-creation query is executed next. Only after that task has been completed will a task for the AAV creation query be executed. After this, the task of the AAV creation query and the cache creation query will be executed alternately until the AAV creation query is complete. In addition, every query run in DuckDB will also execute one or multiple tasks from the cache-creation query until the cache creation has been completed. This slows down the execution speed of queries unrelated to the AAV creation. As per **REQ3**, this is not allowed and therefore has to be avoided.

To avoid interfering with other queries, we have altered the scheduler of DuckDB so the query thread only executes tasks associated with the query, even in single-threaded mode. We run all benchmarks with this customization of DuckDB, which has also been up-streamed to DuckDB [2].

[2] `https://github.com/duckdb/duckdb/pull/13326`

4.3.2 Reading from Approximate Accelerated Views

Accessing the accelerated view is not as simple as reading the table that contains the cached data. Depending on the cache size and the query accessing the cache, the cache is accessed immediately, delayed, or not used at all (see Figure 4.2 for the decision tree). Therefore, each query accessing the cache must be analyzed and assigned to one of the three cases.

```
1 SELECT *
2 FROM my_result
3 JOIN some_table ON my_result.id = some_table.id;
```

Listing 4.4: Accessing the accelerated view. The accelerated view is the relation called *my_result*.

But we begin with a query accessing the accelerated view as seen in Listing 4.4. Because the accelerated view is unknown to DuckDB, resolving it will result in a binding error indicating that *my_result* is not a valid object in the catalog. To resolve this, DuckDB extensions can use *scan_replacement*s to enable DuckDB to bind and access unknown references. Using this API, it is possible to either return a reference to the cache table or the view's query, thereby inlining the view. However, to determine which operation should be undertaken, we must first decide whether a query is *guaranteed satisfiable*, as defined in Section 4.1, meaning the query can be answered using only the cache. To determine if a query does fulfill this property, the logical query plan has to be analyzed for the `LIMIT` operator. However, because the *scan_replacement* is done during binding, a complete and optimized logical query plan is not available yet. Additionally, inspecting the AST representation of the query also does not work, as the DuckDB optimizer might reorder the Logical Operators in the Logical Query Plan, potentially pushing a limit from the top of the query plan down to the view.

The question of how to bind the—for DuckDB—unknown reference remains important, as the DuckDB optimizer will optimize the Logical Query Plan based on the reference returned by the *scan_replacement*. Ideally, DuckDB would optimize both scenarios: inlining the view query and reading from the cache table. This can be achieved by making the scan replacement step return a `UNION ALL` subquery containing the cache on one side and the view query on the other. This approach allows DuckDB to optimize both paths, delaying the decision of which path to keep until later. A potential downside, however, is that the estimated cardinality might be off by a factor of up to 2 since the cardinality of the union is derived from the sum of the cardinalities of both sides.

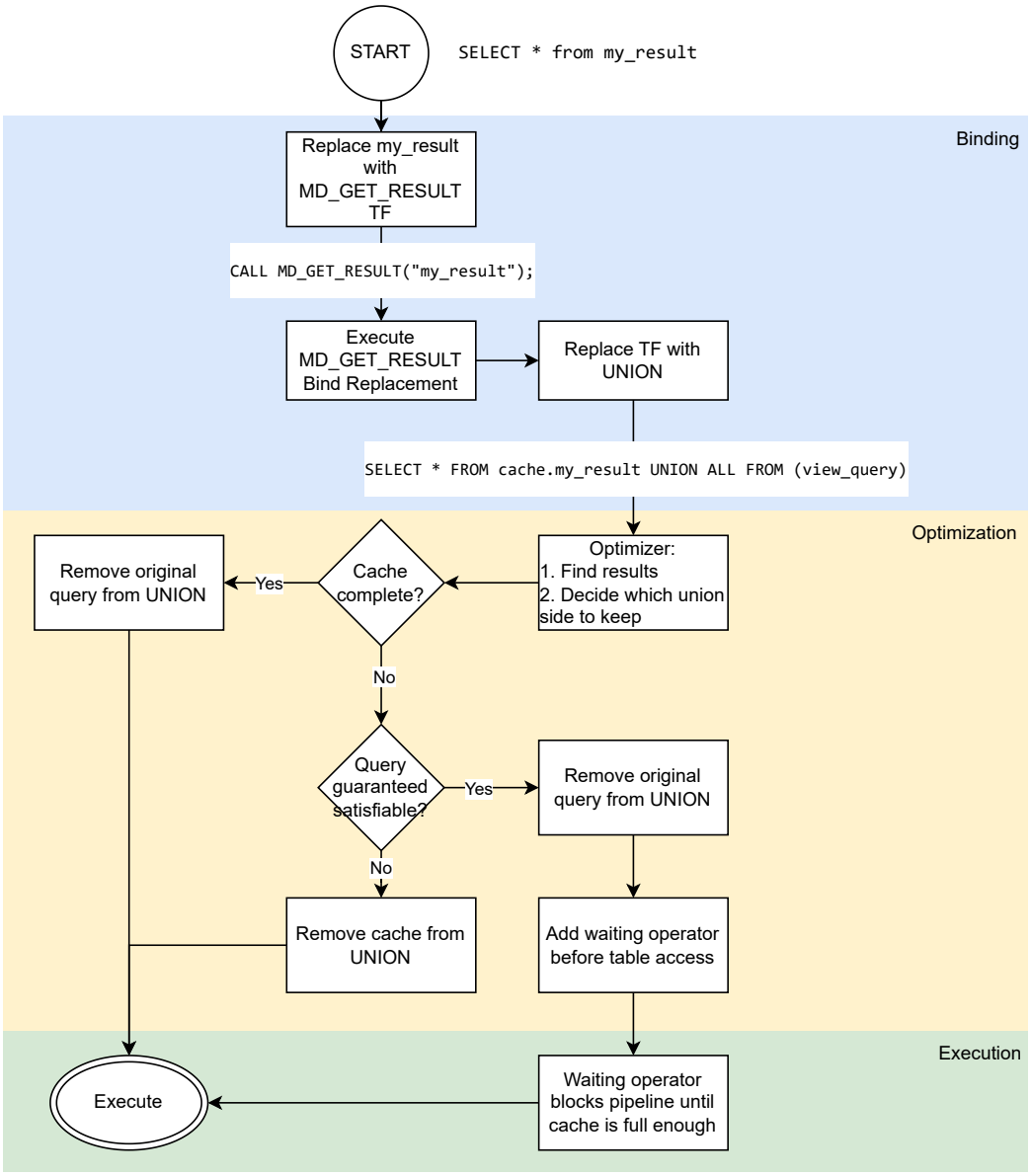


Figure 4.5: Overview of the actions executed when accessing the cache.

The decision to read from the cache or inline the view query is subsequently made in the extensions' optimization phase after DuckDB optimizes the Logical Query Plan. Because the entire plan has been parsed and optimized, the cache's satisfiability can be decided by inspecting whether the accelerated view is accessed with a limit. However, it is impossible to non-blockingly delay the access to the cache until enough data has been cached. This, however, is essential for Wasm, where the execution has to be yielded while waiting for more data to be inserted into the cache. Non-blocking waiting can only be done during the execution phase; therefore, a new physical waiting operator, whose purpose is to delay access to the cache until enough data has been cached, has to be created.

Non-blocking waiting for the cache to fill The waiting operation has to fulfill the following requirements:

1. Waiting has to be done in a way that allows DuckDB to return execution control to the JavaScript event loop, which can then schedule the AAV-cache-building operation mentioned in Section 4.3.1.2, thereby inserting more data and eventually enabling the suspended query to resume execution.

2. The query can only continue running once enough data has been inserted into the cache

Suspending execution in a way that allows DuckDB to yield execution control to the event loop is possible using resumable pipelines. The idea is that, as soon as a physical operator blocks, DuckDB will stop trying to execute that query and only try to execute the pipeline again once the signal that the pipeline is unblocked has been given. The "physical waiting operator" thereby only blocks the query execution if there is insufficient data in the cache and only unblocks it once enough data is available. For this physical waiting operator to block the execution of the pipeline reading from the cache, the cache table reading pipeline has to depend on the physical waiting operator, meaning it can only run once the physical waiting operator has completed running. This, however, will only be the case once enough data has been inserted into the cache.

This leaves one last hurdle to overcome. Inserting new tuples into the cache table occurs in a different transaction than reading from it.

Transaction Compliance When examining the order of transaction creation (see Figure 4.6), it is evident that the transaction responsible for inserting the last 2000 tuples is created after the transaction that reads the cache. As a result, this transaction will discard any rows inserted into the cache table after its creation - which, when accessing tables, occurs during the binding phase - and will not be able to read the last 2000 tuples.

It is, therefore, necessary to do one of three things:

1. Create a custom catalog including custom transaction logic
2. Reset the transaction for the accelerated view's catalog after waiting
3. Create a new physical operator or table function that reads all data of a table, regardless of the currently active transaction

However, even creating a catalog with a custom transaction logic or resetting the transaction will not suffice to read newly inserted data from the cache. This is because the table function (*seq_scan*), used for reading from the DuckDB tables, initialized its global state during the pipeline scheduling phase [3]. The initialization of the global state saves two important bits of information, namely the root segment of the table, containing the underlying table data [4], as well as the maximal number of rows in the table [5]. As these two things were set before the waiting operator started waiting, they do not reflect the current state of the table, as the number of rows and the root segment might have changed. So, even if the transaction is reset, it is also necessary to update those two values. This is possible by resetting the Source of the pipeline the table scan is in, thereby recomputing the global table function state. This should be safe to do, as the dependency on the waiting operator ensures that the pipeline the physical table scan is in will not be executed.

However, this is still a workaround. To ensure no unwanted side effects appear, a new custom table function should be created to read the latest data from a table, regardless of its transaction. However, because of time constraints, this has not been done within the scope of this thesis. Instead, the solution of resetting the transaction has been chosen as a preliminary workaround for reading the latest data from a table.

[3] https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/parallel/executor.cpp#L155

[4] https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/storage/table/row_group_collection.cpp#L167

[5] https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/storage/table/row_group_collection.cpp#L169

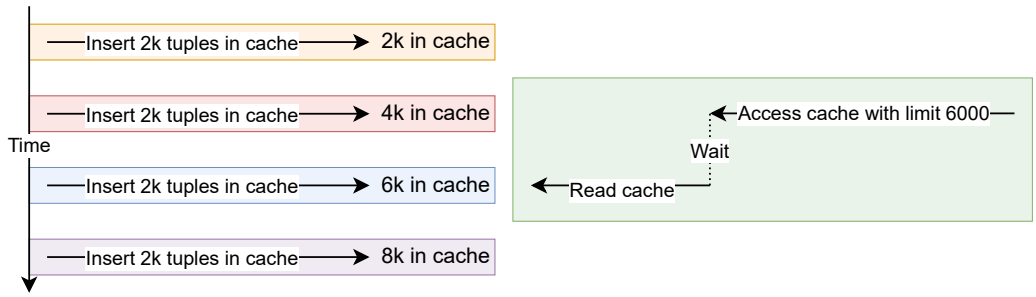


Figure 4.6: Accessing the cache while the limit is not yet satisfied. Each colored box is a transaction

4.3.3 Modifying Accelerated Approximate Views

Other operations, such as deleting the cache or pausing and resuming cache building, are easier to implement because they only modify the accelerated view state or remove cache data without involving asynchronous operations. Pausing and resuming cache building is as simple as stopping to execute the tasks of the pending query in the background cache creation function. This stops the insertion of new data into the cache. To continue inserting new data, the background function has to start executing the cache creation query tasks. One thing to remember is that pausing the cache creation process results in an open transaction, which might lead to unwanted side effects.

If an accelerated view is deleted while a query waits for its cache to materialize sufficiently, the query waiting for the cache is aborted, and the cache is removed. Deleting an accelerated view while a query is already reading its cache is also possible, as transactions safeguard the cached data while a query reads from it.

4.3.4 Cache eviction

While out-of-memory protection has been implemented in the context of the thesis, cache eviction has not been tackled. To prevent DuckDB from running out of memory AAVs, stop the insertion of new data if the memory limit (configurable through SQL - Section 4.2) has been reached.

For cache eviction, we propose implementing a strategy based on LFU in combination with aging to prevent AAV caches from becoming too stale as a starting point. Being able to track the effectiveness of this solution (expressed through cache hits/misses see Section 5.1) would then allow for more targeted optimizations.

In the following, I have evaluated different cache eviction strategies used for query caching:

Shim, Scheuermann, and Vingralek created a decision process that considers the execution cost of the query, as well as the size of a cached result in combination with the frequency of its access. This, in addition to the cache maintenance cost, allows them to efficiently decide which cache entry to evict [52]. Notable here is that the cache maintenance cost would not have to be considered when implementing a similar concept for accelerated views.

In "Recycling in pipelined query evaluation" Nagel, Boncz, and Viglas, create a benefit metric that calculates the optimal eviction policy based on historical queries, the influence materialized results have on each other and an aging mechanism that reduces the influence of older queries on the benefit metric over time. This allows them only to evict results that are not actively used by recent queries [40].

Chen and Roussopoulos achieved the best results by keeping the largest cache and using LFU and LRU as a tie-breaker in [12]. This cache eviction strategy has been evaluated against LRU and LFU.

Finally, Dar et al. created a cache replacement strategy that replaces their cache based on the semantic cache regions that are most active, resulting in the eviction of currently irrelevant data [15].

While all of these cache eviction strategies are valid, I argue that [12] is not relevant for accelerated approximate views, as keeping the largest cache entry does not work in an environment where the same query is executed infrequently during the same session, which is the case in the MotherDuck UI [citation needed:]. And while deciding which semantic region is most active is certainly the right approach for Dar et al., it is too complicated for accelerated views, where the caching granularity is predetermined, as caching is done on a query result basis.

However, [52] and [40] remain relevant, as they take access patterns and cache age into account, which is probably beneficial for MotherDuck's current workloads. They extended my proposed solution by factoring in the query cost, which might be an interesting idea to avoid re-computing expensive AAV caches.

4.4 Integration into the MotherDuck UI

After building the AAV, basic support for AAVs has been added to the MotherDuck UI. Scrolling through the pivot table (see Section 2.7) without re-executing the original query for cases where the cache was exceeded made it work substantially quicker for this activity thanks to its larger cache size and ability to answer queries from partial caches. However, there was no perceivable improvement regarding time-to-first-tuple or column explorer queries.

Because only preliminary testing has been done, most optimizations for the old caching setup are still in place. A larger effort must be made to move away from the old caching approach and to AAVs.

5

User Study

To evaluate the effectiveness of the Accelerated Approximate View, it is crucial to know how it will be used. For this, I will examine how the MotherDuck UI currently utilizes caching by collecting user workloads. These queries will then be compared to the workloads run in "Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet" and "Cloud Analytics Benchmark," as well as TPC-[H|DS]

5.1 Collecting User Metrics

Knowing how the UI works, we can now start collecting query metrics. These metrics are collected at three different points in the UI. The first logged event is before creating the cache (see Figure 5.1). Afterward, the size of the cache is determined, and if its size is equal to 50 000, it is assumed that not the entire query could be cached. Consequently, a Common Table Expression (CTE), called `read_query_result` , is created, which, from that point on, will be used to access the user query. If the entire query could fit into the cache, the CTE is equivalent to reading from the created cache table.

At this point, we log a second event, informing us that the cache creation has been completed. In the following, several queries are run on `read_query_result` to fill the pivot table and the column explorer. We start by reading the first 1024 tuples to display them in the table, followed by a *count start* query to determine the total number of rows. We then log individually when the `count(*)` and the `LIMIT 1024` queries have been completed. Finally, on average, over three queries are run to populate the column explorer. The exact number of queries depends on the data types returned by the `SELECT` statement. The execution of these statements has not been logged to reduce implementation time. It

would, however, be interesting to see the impact of the query cache on these statements as well.

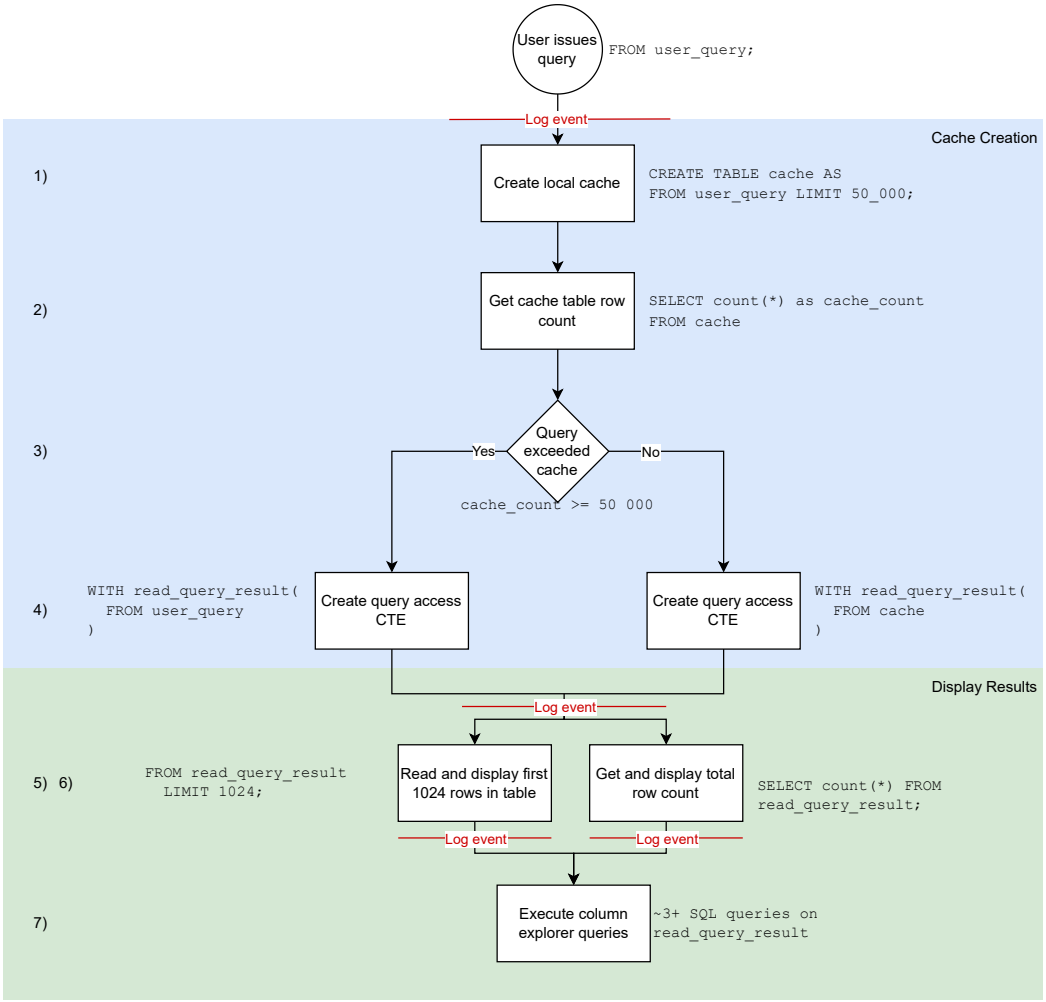


Figure 5.1: Queries run in the MotherDuck UI whenever a `SELECT` statement is executed. First, the cache is created, and if the entire query fits in the cache, the cache is afterward used to power the tabular preview and the column explorer. If the query exceeds the cache, it is bypassed in favor of the original query.

Besides timing information, the added logging statements collect data types, column count, row counts, and the complexity of the queries, expressed by the SQL operators used in queries and the query token count.

5.2 User Workloads

Over 61 days, I collected query patterns of 23 720 `SELECT` statements by 973 users issued through the MotherDuck UI. Out of these 23 720 queries, 36.43% were queries that have run more than once, although by different users. With a mean user query duplicate count of 17.75% and a median of 4.76%, we are much lower when it comes to duplicate user queries, making it apparent that many users are running the same queries as other users.

> **Opportunity 1.** The large number of duplicate queries between users highlights opportunities for server-side cross-user query caching, assuming the queries operate on the same data. However, this can have serious security implications, and further investigation into the query complexity is needed to determine if the duplicate queries are worth caching.

When investigating the number of rows returned by these queries, it is evident that most queries return fewer than 150 tuples (see Figure 5.2). There is, however, a significant variance in the number of tuples returned, ranging from 0 tuples to 16.4 billion tuples, explaining the significant difference between the median of 148.0 and the mean of 971 479.64. This is in line with the query statistics collected by Renen et al. where 75% of all queries scanned less than 1GB [63] with the difference being that Tigani investigated in *Redshift Files* the data queried, not the number of tuples returned.

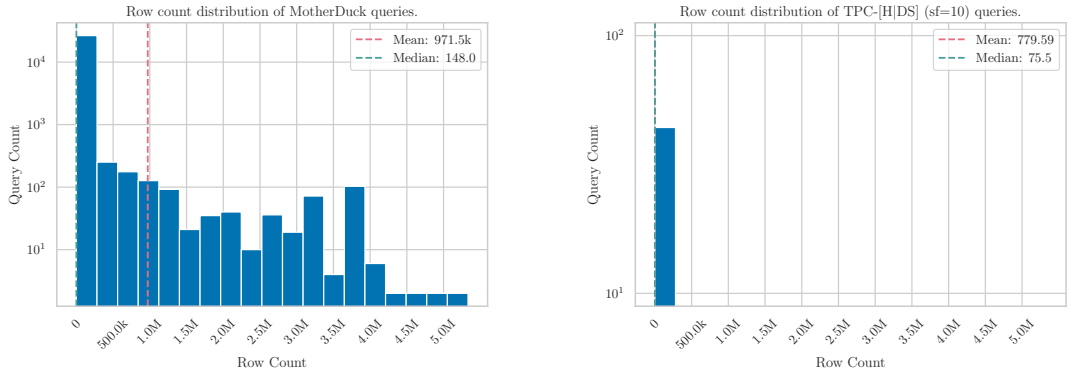


Figure 5.2: Number of rows returned by `SELECT` queries. Outliers were removed from the graph, but the mean and median were computed with outliers.

The number of columns returned by queries has a smaller variance with a mean of 18.17 and a median of 7.0 (see Figure 5.3). Hence, there are queries returning many columns, with one containing 3315 exclusively integer columns.

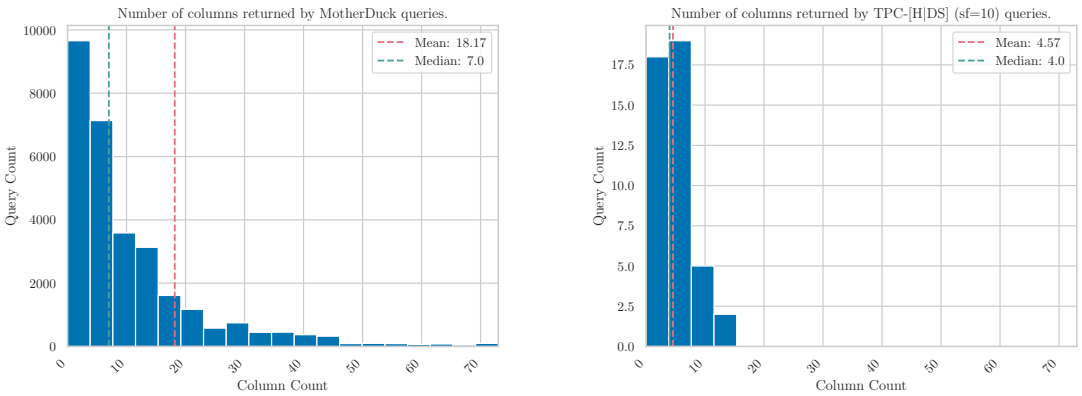


Figure 5.3: Comparison of column count distributions. Outliers were removed from the graphs, but the mean and median were computed with outliers.

When investigating what data types these columns contain in Figure 5.4 and Table 5.1, it becomes apparent that with 57.68% most columns have a `varchar` type, followed by various number and date formats. On the other hand, complex types like maps and structs are well below 1%, while blobs only make up 0.03% of the data types. This is in line with what was observed by Renen et al. [46]. Please note that some data types like DuckDB's `hugeint` are not supported in DuckDB Wasm and are therefore not represented here.

The complexity of the queries run in the MotherDuck UI is relatively low and does not require too much SQL expressed through a median token count of 11.0 (see Figure 5.5a). However, some queries are more complex, resulting in a total token count of 6743. This is very different from the data reported in the "Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet" where "customer queries [where] much longer than the ones from standard benchmarks" [46] with the largest one being 16MB of SQL code. This can probably be attributed to the MotherDuck UI primarily being used for data exploration, explaining the shorter queries.

This reduced complexity can also be seen in Figure 5.6. 51.26% of the queries use only **one** of the following SQL operators, namely `JOIN`, followed by `LIMIT` and `WHERE`, while 30.87% of the queries only execute a table scan.

In Figure 5.7, you can see the exact kind of queries that are run in the MotherDuck UI. With 27.95%, about a quarter of the queries executed only read (remote) data. This further highlights that the types of queries run in the UI are vastly different compared to the TPC-[H|DS] benchmarks or the workloads observed by the Redshift team, which were not focused on notebook usage.

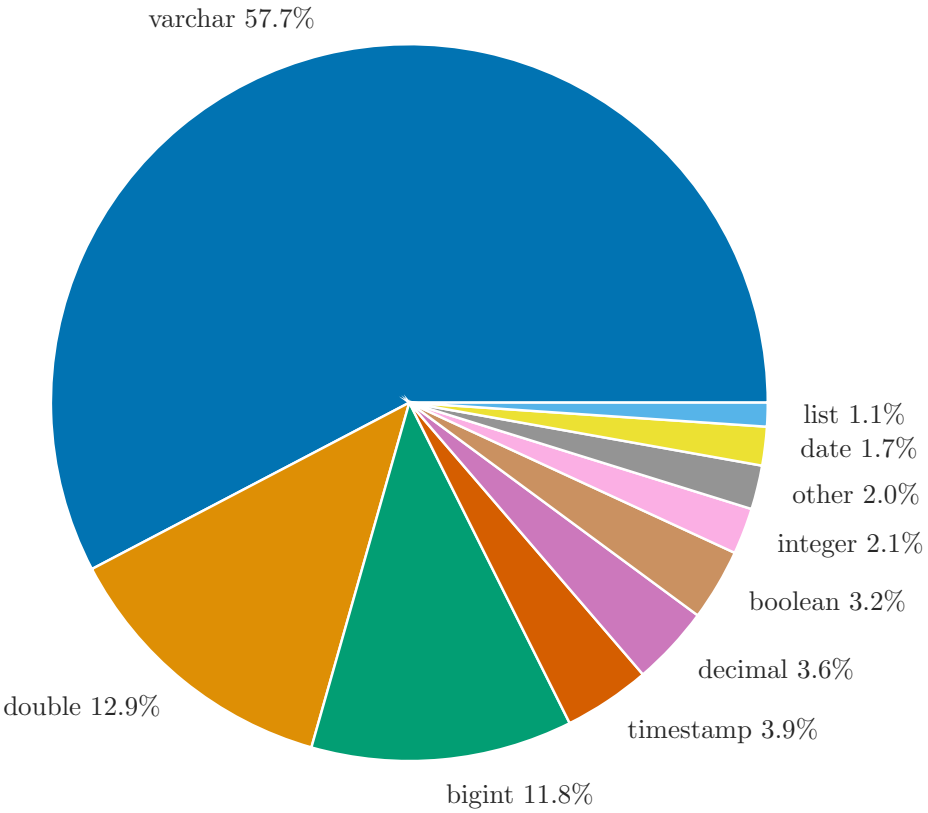
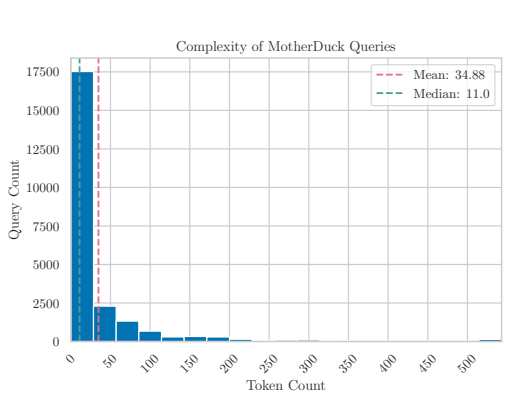
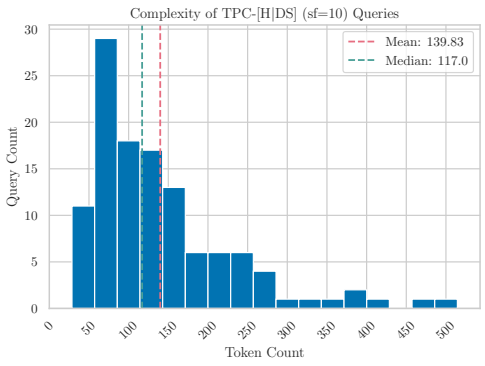


Figure 5.4: Data type usage of `SELECT` queries run in the MotherDuck UI



(a) Number of tokens in `SELECT` statements. Outliers were removed from the graph, but the mean and median were computed with outliers.



(b) Number of tokens in the TPC-H and TPC-DS benchmarks. Outliers were removed from the graph, but the mean and median were computed with outliers.

Figure 5.5: Analysis of SQL query complexity and feature usage.

Table 5.1: Data type usage in MotherDuck, Redshift ([46]) and TPC-[H|DS]

Datatype	MotherDuck	TPC-[H|DS]	Redshift
varchar	57.68%	37.81%	56.20%
double	12.91%	21.89%	2.20%
bigint	11.79%	12.94%	9.40%
timestamp	3.92%	-	6.20%
decimal	3.59%	16.92%	-
boolean	3.23%	-	1.50%
integer	2.09%	6.97%	11.60%
date	1.74%	1.00%	3.20%
list	1.07%	-	-
tinyint	0.87%	-	-
real	0.53%	-	0.20%
struct	0.22%	-	-
smallint	0.13%	-	2.30%
interval	0.08%	-	-
time	0.03%	-	-
enum	0.03%	-	-
map	0.03%	-	-
blob	0.03%	-	-
array	0.02%	-	-
numeric	-	-	7.00%
hugeint	-	2.49%	-

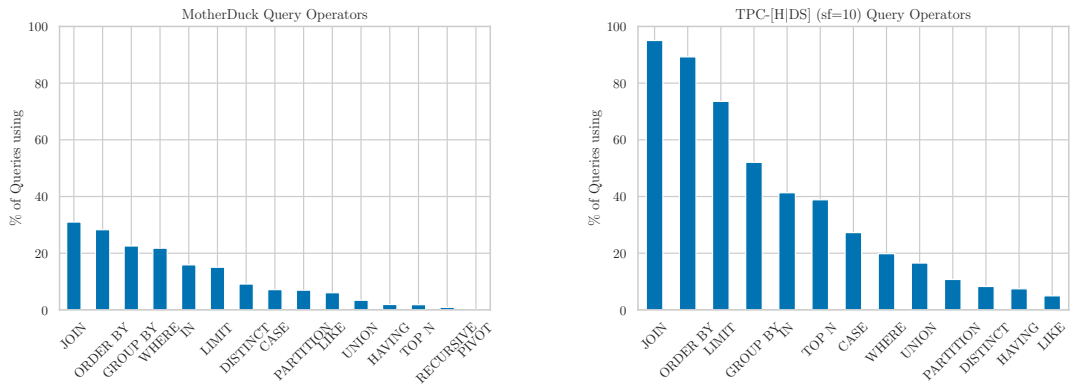


Figure 5.6: Percentage of queries using different SQL features. `WHERE` operators doing implicit `JOIN` s have been converted to `JOIN` operators.

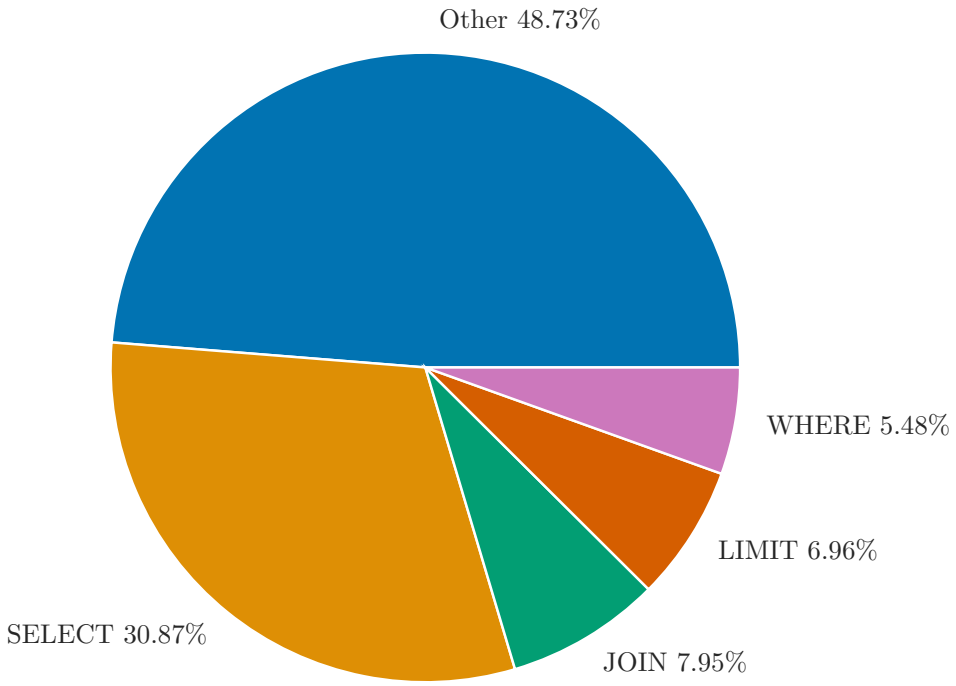


Figure 5.7: Queries using only a **single** query operator. E.g., queries in the `LIMIT` bucket contain **only** a `LIMIT` operator. Most queries contain various combinations of query operators and are grouped in *other*. `WHERE` operators doing implicit `JOIN` s have been converted to JOIN operators.

5.3 Caching Impact

The previous analysis only investigated query statistics of standalone queries. The following paragraph will examine how caching is currently used in the MotherDuck UI and how caching impacts performance.

In Figure 5.8, you can see the execution times of steps 5) and 6) of Figure 5.1, while Figure 5.9 contains the execution times from steps 1) to 6). Reducing the time from 1) to 6) is crucial, as this will allow the table to display the first results, thereby shaping the perceived responsiveness of the application. On the X-Axis, you can see the number of tuples that *would have* been returned in step 5) if there had not been a limit of 1024 applied to it. On the Y-Axis, you can see the time it took to run steps 5) and 6) for Figure 5.8 or steps 1) to 6) for Figure 5.9.

As you can see in Figure 5.8, there is a jump in query execution time for queries left of the cache barrier compared to queries right of the caching barrier. This is expected, as queries on the left will run purely on the local cache. While the gap is not large, it is noticeable, showing the advantage of local caching.

In contrast to Figure 5.8, Figure 5.9 contains the cache creation process. Because the cache creation process is included for queries left of the cache barrier, they exhibit higher execution times, even though the cache is used for later queries. However, because, on average, less data is transmitted for queries left of the cache barrier, the average query still completes quicker than queries operating on more significant amounts of data.

Notably, none of these graphs show the full benefit of creating a query cache, as the column explorer queries and additional filter/pivot/sort/scroll operations of the pivot table are not included. So, this benefit is, in reality, even higher as more queries benefit from caching.

Finally, let's investigate how much time is spent on the cache creation process compared to queries 5) and 6). In Figure 5.10, you can see that most of the time is spent creating the cache, even later, when the cache is effectively bypassed for queries 5) and 6). This is to be expected because query 5) only transfers 1024 tuples over the network, with query 6) sending only one tuple, which is a lot lower compared to the up to 50 000 tuples that have to be sent to the client when creating the cache. Because most of the queries made in the MotherDuck UI are relatively simple (see low query token count in Figure 5.5a), the total execution time is dominated by the network bandwidth rather than the query execution time. This becomes more apparent when examining the execution times for the row counts from 0 to 48k tuples. The execution times rise from 1310ms to 2640ms, suggesting that

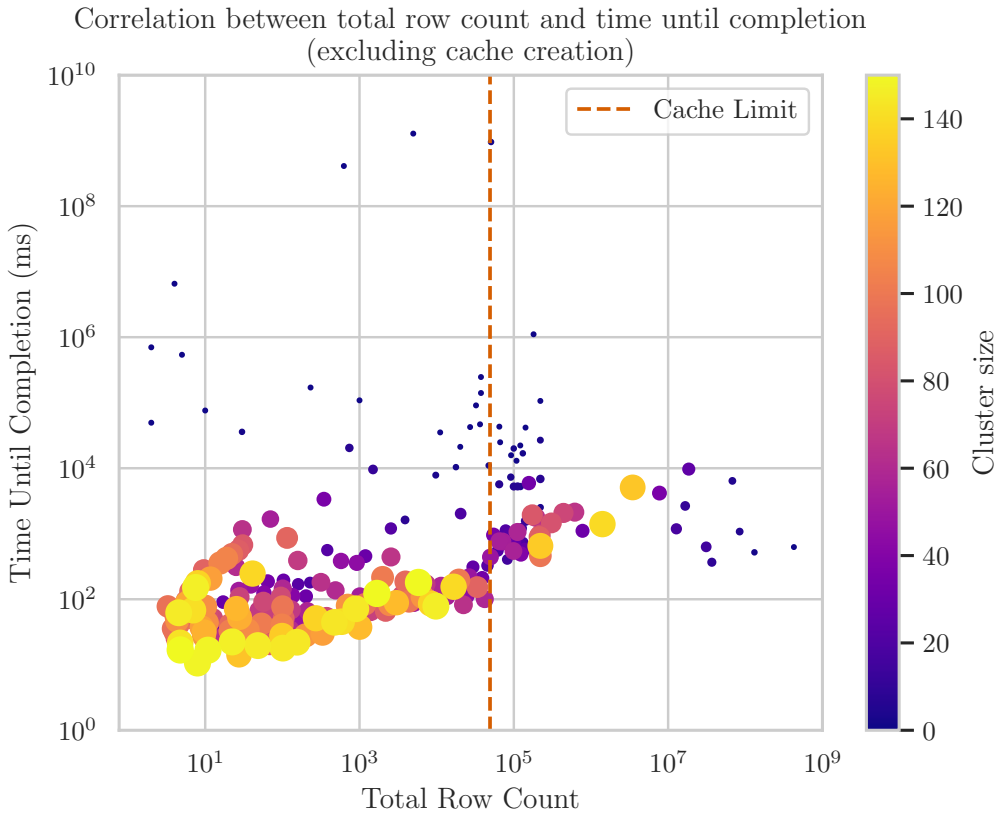


Figure 5.8: Correlation of the total returned tuple count and the time it takes to complete the query. Times were measured with a limit of 1024 applied to the query. Without the limit, the query would have returned the number of rows indicated on the x-axis. Queries left of the cache limit were run on the locally created cache, while queries right of the cache were rerun.

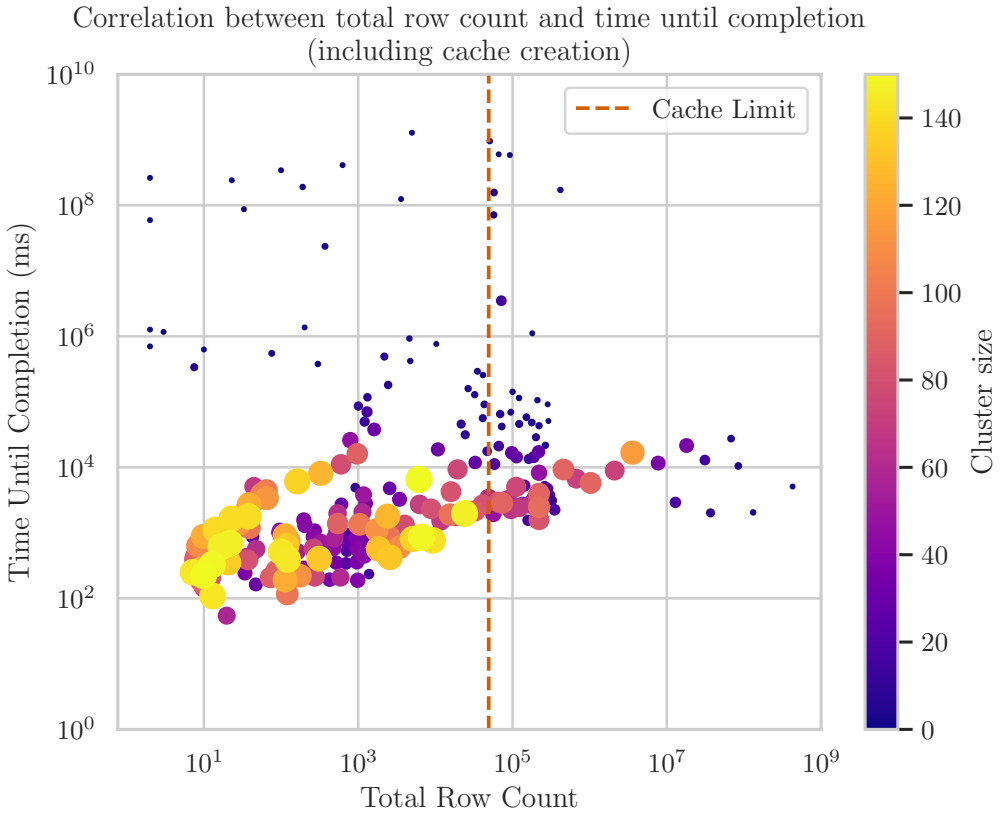


Figure 5.9: Same as Figure 5.8, but this time, the time to create the cache is also included in Time Until Completion. Even though queries right of the cache limit do not use the cache, as the query returned to many tuples, the time it took to create the partial cache is included.

most users are bandwidth-limited when executing these queries. This, however, does not mean that caching is not advantageous because more data is transferred to create the cache. We are missing statistics about the column explorer queries, which are significantly more complex and could run on the cache, thereby increasing its benefits. Additionally, when considering AAVs instead of the current caching mechanism, we have the advantage of more granular cache control, making it possible to cache only the first n tuples, after that, pause the cache-building process and then continue the process when more cached data is needed.

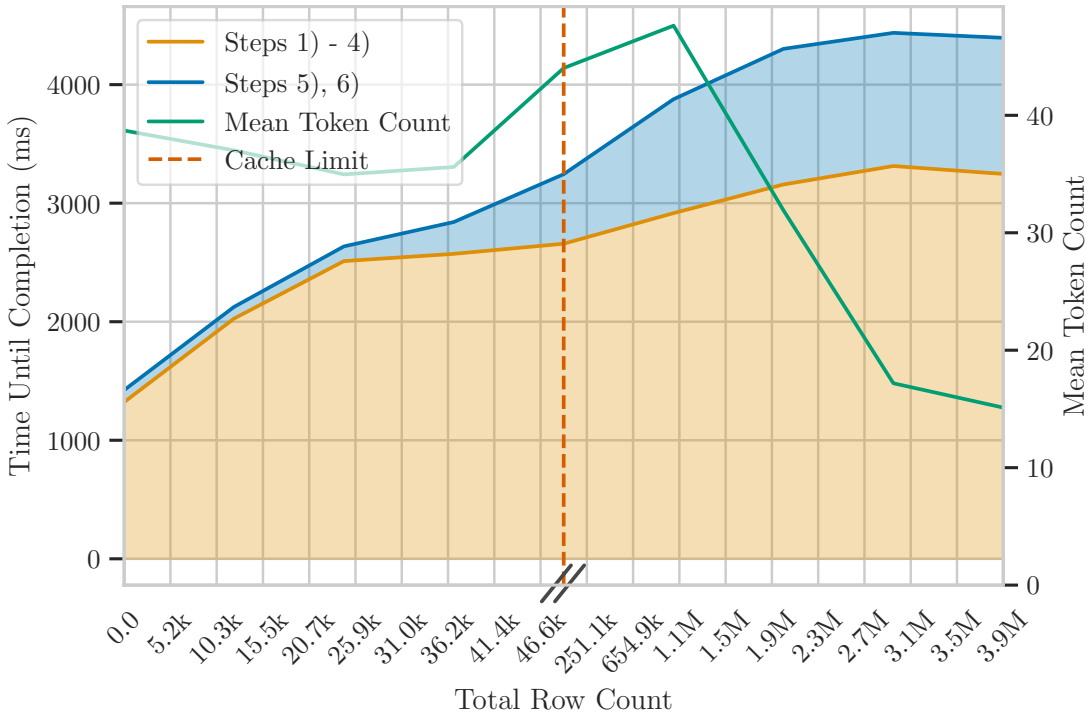


Figure 5.10: Comparison of the time it takes to create a cache (steps 1) to 4)), compared to the time it takes to execute queries 5) and 6). Because the queries of steps 5) and 6) run in parallel, we take $max(t_{end\_5)}, t_{end\_6)})$ to determine the runtime. The scale of the X-Axis changes after the cache limit has been reached and outliers have been removed. Additionally, the query complexity is plotted over the total number of rows.

6

Experimental Evaluation

The following chapter evaluates the accelerated approximate view. Running benchmarks like TPC-H or TPC-DS [64][65] provides value for systems trying to improve query execution performance. Accelerated views, on the other hand, will cache the results of queries and will, therefore, always be quicker than re-executing a reasonably complex query.

Additionally, the TPC-[H|DS] benchmark queries are not representative of the queries run in the MotherDuck UI. This is especially apparent when comparing the query complexity of the TPC queries with MotherDuck user queries, as evident through the number of tokens per query (see Figure 5.5a) and the query operators used in each query (see Figure 5.6), which differ vastly from the workloads run in TPC-[H|DS]. The same is true for the number of returned rows (see Figure 5.2) where MotherDuck queries return several orders of magnitude more tuples on average than TPC-[H|DS].

With a median of 75.5 returned tuples in TPC-[H|DS] (sf=10), a mean of 779.59 and a max of 27 840, all but two queries return fewer than one data chunk of 2048 elements. This makes caching the TPC-[H|DS] queries an uninteresting experiment, as almost no data is being cached.

While the SSB [41] would have been an alternative to TPC-[H|DS], it is based on TPC-H. It, therefore, comes with similar problems, like high query complexity, which are not in line with the queries executed in the MotherDuck UI. Additionally, the number of returned tuples (SF=1) is on the lower side with a mean of 39 867.85, a median of 8228, and a max of 205 714, which does not make it a good fit for benchmarking the caching performance of large queries, however ultimately it was decided not to use the SSB for the evaluation.

Therefore, a set of queries that better represent the MotherDuck UI workloads has been designed and will be used throughout the evaluation section (see Appendix A for the queries). We begin by benchmarking the UI using a local simulation and comparing the

currently employed caching implementation with the new accelerated view-based approach. After that, we will investigate different methods to retrieve query data as fast as possible, including the new accelerated view. Lastly, we will go into more detail about the overhead of accelerated views. There are instances where a specific query would not work in the context of a AAV. In this case, the query is skipped.

6.1 Benchmarking Set-Up

All benchmarks are run on the machine described in Table 6.1. While the network speed is up to 10GiB, this speed is not necessarily reached when communicating between different Amazon Web Services (AWS) regions. Each benchmark is run 10 times to remove the influence of potential outliers. All AAV/tables are created in an in-memory database if nothing contrary is mentioned. The same queries are run in C++ and Wasm. However, in C++ , the benchmarks are run with a scale factor of 5 while the scale factor in Wasm is 1. This was done, as the execution in C++ is faster than Wasm because C++ uses multiple threads and is generally more efficient than C++ that was compiled to Wasm [36] and uses a different execution model.

To prevent DuckDB from using any disk in case the memory is running low, the `temp_directory` has been set to a non-writable directory, causing DuckDB to crash if it tries to spill to disk. Additionally, `max_temp_directory_size` has been set to 0.

Table 6.1: Benchmarking System

Instance	EC2 c5.4xlarge
Instance Location	Frankfurt
CPU	16vCPUs
Memory	32GiB of Random-access memory (RAM)
Network	up to 10 Gps[5]
OS	Ubuntu 22.04.4
g++	11.4.0
Emscripten	3.1.53
Wasm runtime	Node.js v18.20.4

6.2 Benchmarking the MotherDuck UI

The UI currently creates a cache for the first 50 000 tuples of a `SELECT` query. If this cache could fit the entire query result, it powers the pivot table and the column explorer (see Section 2.7 for more details).

In the following, we will benchmark the MotherDuck UI by simulating the execution of a series of representative `SELECT` queries. To remove unrelated interference with query execution, a replica of the MotherDuck UI has been created. This replica executes the same queries but has no UI and is run in Node.js instead of the browser to remove potential sources of inaccuracies introduced by the rendering of UI elements.

We thereby benchmark steps 1) to 7) seen in Figure 5.1, representing the current caching strategy. In Figure 6.1, you can see the workflow from Figure 5.1 adapted to incorporate accelerated approximate views. In this case, steps 1) to 4) have been measured. The difference in complexity stems from the manual cache management required in the old caching setup. AAVs remove this application complexity, as the decision process of when to read from the cache or the view is already built in.

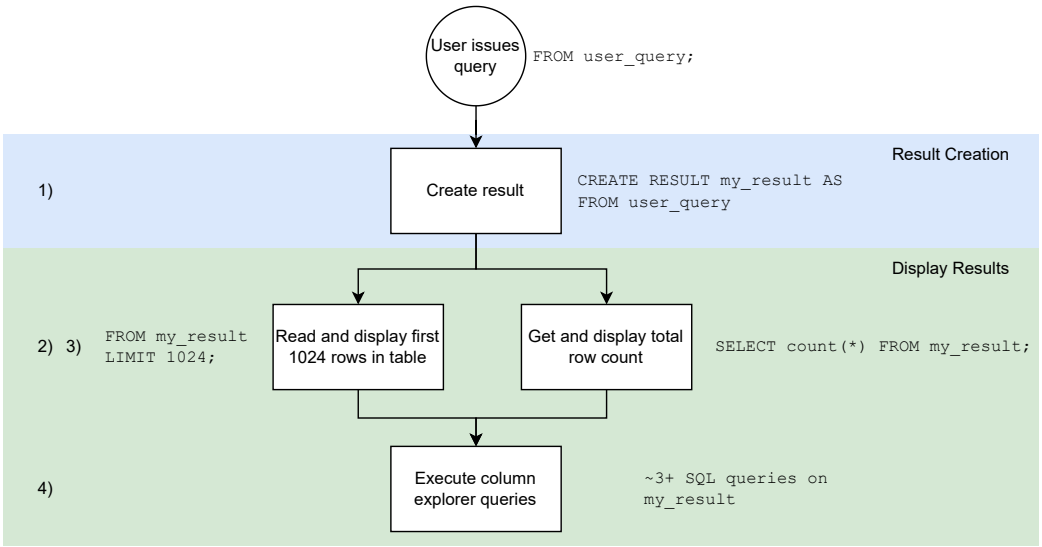


Figure 6.1: Queries executed to measure the impact on execution speed when replacing the old caching mechanism with accelerated approximate views.

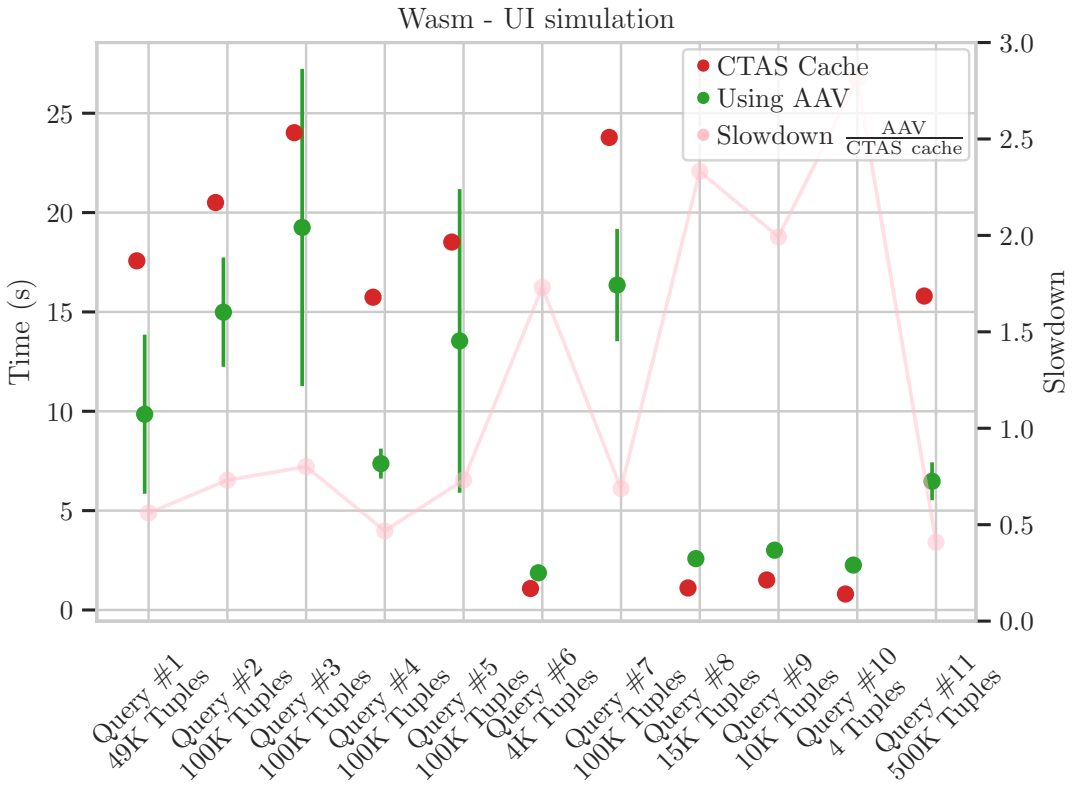


Figure 6.2: UI simulation running in Wasm.

6.2.1 Results

In Figure 6.2, you can see the results of the previously described benchmark. We have the old caching implementation in red and the accelerated view-based implementation in green. As you can see, accelerated views outperform the current caching implementation in 7 of the 11 benchmarks. This is mainly because accelerated views cache more tuples. This is possible without risking out-of-memory errors because of the lower-level nature of accelerated views, allowing for more fine-grained cache memory management, such as the eviction of caches when memory is running low (not implemented yet, but planned), which is impossible with the current approach. AAVs also protects the user from running out of memory because of a single cache entry, as ongoing caching efforts will be aborted if the memory is running low and no other cached entries can be evicted. While AAVs also provides earlier access to the cached data, this cannot be observed in an end-to-end benchmark like this one, as being able to read the first 1024 tuples earlier in 2) does not translate to any time saved, because we still have to wait for the `count(*)` query to complete in 3) and a count star query requires for the entire cache to be materialized.

However, the larger amount of cached tuples does not explain why the AAV is **only** quicker for some queries. I hypothesize that the reason for the improved performance is connected with the concept of *guaranteed satisfiability*, which states that only queries that explicitly specify the tuples they are interested in can wait for the cache to contain these tuples (see Definition 1 for more details). When investigating the queries accessing the accelerated view, it becomes apparent that only 2) is guaranteed satisfiable, as it is the only query specifying a limit. This query can, therefore, be answered using only the cache, even if the cache is not yet large enough/the query did not fit into the cache, which was not possible with the old caching setup. However, all other queries are not guaranteed satisfiable. After answering 2) from the cache, 3) cannot be answered using an incomplete cache. Therefore, the original query is inlined, re-executing the original query for 3). After 3), the column explorer queries are executed. But because we have re-executed the original query once for the `count(*)` query, the cache has had more time to continue building in the background. If lucky, the cache has completed building in the time it took to re-execute the original query for 3). If that is the case, all column explorer queries will be executed on the local cache. However, if that is not the case, (some) column explorer queries re-execute the original query again. Depending on the state of the cache after 3), accelerated approximate views are either quicker because the cache has finished building while 2) was executed, about the same if some of the column explorer queries could use the cache, or slower, as

creating a cache in the background slows down other simultaneously running queries. This also explains the significant variance in the query execution times for accelerated views in this simulation.

Opportunity 2. Currently, we decide during query planning whether to:

1. Access the cache immediately
2. Delay the cache access
3. Bypass the cache

The decision whether to delay or bypass the cache access is made by inspecting the logical query plan of the query accessing the accelerated view. If the query accessing the accelerated view is guaranteed satisfiable, we wait until enough data has been cached to answer the query safely.

To improve the performance of queries that are not guaranteed satisfiable, we could always optimistically wait until the cache is large enough. If the accelerated view runs out of caching space, we fall back to inlining the view query. This would vastly improve the query performance in our case, as all queries used in this benchmark fit in memory. Doing this in DuckDB would involve the dynamic scheduling of query pipelines depending on whether we want to read from the cache or fall back to query inlining.

However, whenever the query does not entirely fit into the cache, we have spent time waiting, which could also have been spent executing the query with an inlined view.

Opportunity 3. One way to further improve the End To End (E2E) execution speed is to return an approximation of the query count for the `count(*)` query executed in step 3) using DuckDB's query progress estimation. This could be done by collecting the progress information while the cache is built in the background. This, of course, has the disadvantage of not being entirely accurate. However, when paired with repeated execution of the `count(*)` query, this would allow for a faster initial execution. Repeatedly executing `count(*)` would result in more accurate count values as time progresses, as the DuckDB query progress indicator becomes more accurate over time. However, because this approach results in "wrong" results, this is an optimization that has to be explicitly enabled by the user, be it through a configuration setting or a SQL extension like an `approximate_count(*)` function.

An alternative for the repeated executions of the `approximate_count(*)` function is to give users a UI element or a configuration, which would force an accurate count after the initial estimate has been displayed.

Opportunity 4. A less complicated alternative for Opportunity 2 is to pause the cache building if we know that the column explorer queries cannot read from the cache, thereby improving the query execution speed by removing unnecessary background activity. We know if the column explorer queries can use the cache because 2) returned the exact number of tuples for the executed query. Alternatively, the `approximate_count(*)` could be used to determine the expected number of tuples for a AAV conform tuple count. If this number is larger than the maximal caching size, the column explorer queries will bypass the cache, as the cache creation process will be aborted as soon as the limit is reached. We can, therefore, pause the ongoing caching efforts until the column explorer queries have finished running to reduce the interference of the caching process with the column explorer queries. After the column explorer has finished executing its queries, the cache creation process can be continued, so potential follow-up queries issued by the pivot table can benefit from the cache.

If the number of tuples is lower than the caching limit, the column explorer queries can be instructed to wait for the cache to finish building. After that, they are executed, utilizing the now fully built cache.

6.3 Micro Benchmarks

The following will contain benchmarks measuring different performance characteristics of AAVs. We will start by comparing an AAV to streaming queries by investigating the time it takes until the first tuple is consumed. After that, we will investigate the time it takes to build a cache using accelerated approximate views and CTAS.

6.3.1 Time to First Tuple

When using the MotherDuck UI, receiving the first 1024 tuples as fast as possible is crucial, allowing the UI to display the Pivot Table as quickly as possible (center in Figure 2.10). In the following section, we will investigate three different query execution methods and compare them by how quickly they provide access to the first tuples of the result:

1. Execution using a **Streaming Query** and the consumption of the initial DataChunk.
2. Creation of a table using a **CTAS** statement followed by reading the first tuples.
3. Creation of a **AAV** followed by the reading of the first tuples.

Remember that in Wasm, streaming queries currently have a `LIMIT 100%` as a top-level operator, removing the benefit of having a small buffer that Streaming Queries typically provide (see Section 2.6.1).

6.3.1.1 C++ Results

The results for the local and hybrid execution in C++ can be seen in Section 6.3.1.1. On the left Y-Axis, we have the execution time of the queries, while the X-Axis contains the executed queries and the returned number of tuples. The Y-Axis on the right displays the slowdown using an AAV introduced when comparing it to CTAS. A slowdown of below 1 is desirable, as this means that the AAV outperforms the CTAS operation. An AAV is compared to a CTAS statement and not a Streaming Query, as AAVs are supposed to replace the currently CTAS based caching logic.

For entirely local queries executed in C++ , we see that streaming queries often outperform AAV and CTAS queries. However, reading from a AAV is sometimes quicker than reading the first tuples of a streaming query, although only by a small margin. This margin increases when the queries also include remote data. As soon as this is the case, accelerated views more consistently outperform streaming queries. This is because streaming queries fill an internal buffer before giving the user access to the query result. Accelerated views, on

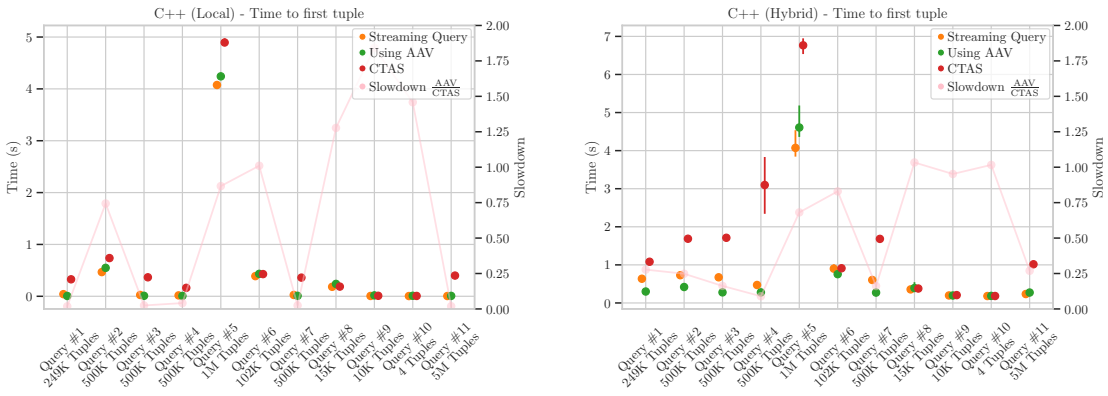


Figure 6.3: Time until the first tuples have been read in C++ . Streaming queries measure the time until the first DataChunk of 1024 tuples was retrieved. CTAS and AAV measure the time starting before the creation of the AAV/table ending after the first tuples were read from the relation.

the other hand, have been engineered to remove any buffering and, therefore, outperform streaming queries. The result of this engineering effort is more visible for remote operations because remote operations contain bridges that buffer data and send it over the network as soon as a specific buffer size is reached. Because streaming queries have a result collector that also buffers incoming data, streaming queries must wait for multiple bridge-sending operations to complete before exposing their results. Accelerated views, on the other hand, expose the received data as soon as the first bridge data arrives.

6.3.1.2 Wasm Results

AAVs in Wasm behave very differently compared to C++ . Large outliers are rather common in Wasm (see Figure 6.4). This is mainly caused by the single-threaded nature of Wasm, where one long-running task can block the only thread. This behavior gets only exaggerated by DuckDB running in a different Worker, making the communication between the main thread and the Web Worker more complicated than a simple function call, as every communication request becomes a task in the JavaScript event loop and can only be executed once all other previously enqueued tasks have completed running. This prevents the DuckDB worker from receiving/sending messages while actively executing a query. The custom background execution mechanism built to enable AAV building further increases this problem, as this mechanism adds more tasks to the JavaScript event loop, further delaying communication between the main loop and the worker. However, this

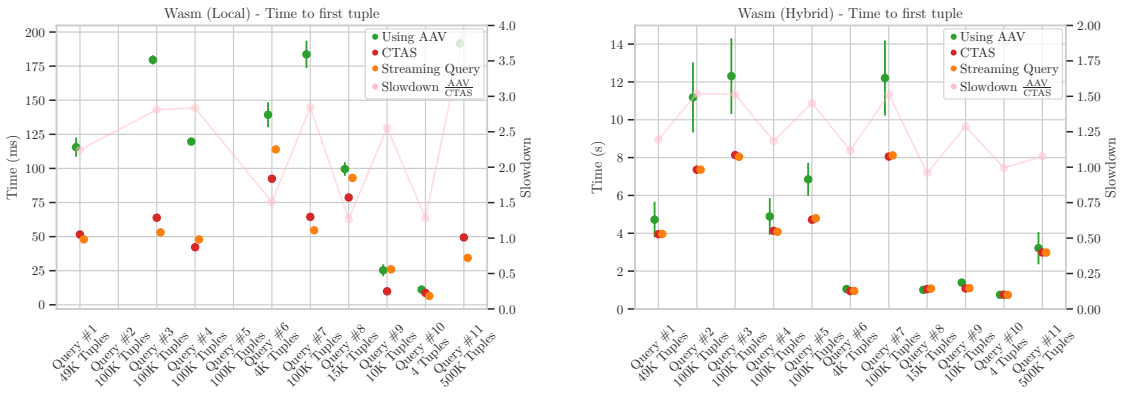


Figure 6.4: Time until the first tuples have been read in Wasm. Streaming queries measure the time until the first DataChunk of 1024 tuples was retrieved. CTAS and AAV measure the time starting before the creation of the AAV/table ending after the first tuples were read from the relation.

communication is necessary to execute queries and receive the results of a completely executed query.

This execution mechanism also explains the more significant outliers and the worse performance observed in Wasm. Following the lifecycle of our AAV benchmark, we start by creating the AAV, followed by a query accessing the accelerated view. Because the AAV's cache is not large enough for the accessing query, it is blocked until enough data has been cached. While the accessing query is blocked, the cache-building query is executed periodically and will eventually write enough data to the cache table, unblocking the accessing query. However, it will not be executed immediately, as the cache creation task or any other task queued before it must be completed first, preventing the immediate execution of the reading operation.

Resulting from this process, we can determine two parts which mainly contribute to the observed overhead:

1. The continuous re-scheduling of the background cache filling.
2. The delay resulting from the delta between the moment the background cache has enough data and the moment the query reads said data from the cache.

Opportunity 5. The current execution mechanism of Wasm is sub-optimal, as it does not allow for the execution of query-unrelated background tasks, forcing us to create a custom background execution driver, side-stepping DuckDB, to create the background cache. Next to being unintuitive and a maintenance burden, this exaggerates the existing communication latency. I propose a shift in the execution paradigm of DuckDB Wasm, which, while enabling native background execution, would also improve general query execution speed and communication latency.

As seen in Section 2.5, the execution of queries is triggered from the main event loop and executes a query in one go. What I propose here is an event-based, and therefore also more JavaScript-conform, design removing the current semi-blocking query execution mechanism and replacing it with the following:

The main event loop and the Web Worker would still communicate through an asynchronous communication channel. However, after receiving the query, the Worker does not execute the query in one go. Instead, the worker process has a C++ internal loop that works on query tasks. Every n milliseconds, it yields to the JavaScript event loop, allowing for exchanging messages with the main event loop (e.g., start/abort a query, etc.). As soon as this is done, the C++ loop continues executing query(-unrelated) tasks. Whenever a query finishes, the Web Worker dispatches a `query-finished` event, which either already contains the query result or functions as a notification for the main event loop, stating that the query result can be retrieved by communicating with the web worker.

These changes would remove times when no query is executed, decrease query latency, and allow for the creation and execution of background tasks, eliminating many of the observed problems.

Using the changes proposed in Opportunity 5, it would be possible to eliminate both main factors, resulting in high time to first byte times for Wasm, namely (1) the overhead resulting from having to re-schedule the cache building task and (2) the delay between having enough data and being able to read the data. (1) is fixed by natively being able

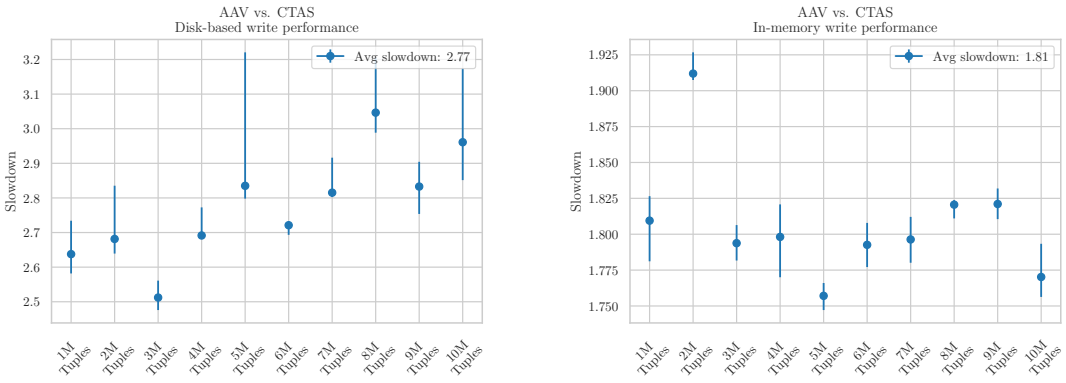


Figure 6.5: Slowdown of an AAV compared to a CTAS statement when running a query with almost no computational overhead.

to schedule background tasks, while (2) is also fixed by the new scheduler as it is now viable to schedule short-running tasks, as opposed to the current situation, where shorter background tasks result in higher context switching overhead and ever lower performance.

6.3.2 Write Speed

In the following, I will investigate the write performance of an AAV by comparing its time to create a fully materialized cache with the table creation speeds of a CTAS statement. This is especially relevant for hybrid caches, which are caches built on the server and the client (see Section 7.1 for more details). This enables the creation of larger server-side caches, where more memory/storage is available. Hybrid caches are primarily useful for scenarios where the cache exceeds the client resources or network bandwidth. As caches become much larger on the server, server-side caches will likely be on disk to preserve the server's RAM for query computation. Therefore, it is crucial to investigate the on-disk write overhead of an accelerated view compared to a CTAS.

To benchmark this, a query generating data with almost no computational overhead is used to avoid influencing the measured write performance too much. This query can be seen in Listing A.13 and utilizes a `generate_series` statement to create tuples containing different data types, allowing DuckDB to use different compression techniques.

6.3.2.1 Results

The result of this benchmark can be seen in Figure 6.5. The benchmarks are run with the previously mentioned query with two varying parameters: Firstly, the number of tuples

that the query should generate (between 1 000 000 and 10 000 000 on the X-Axis), and secondly the location of the cache, which is either on disk (left graph) on in memory (right graph). We then compare the time it took to run the query using the accelerated view t_{AAV} with the time it took to run the query using an CTAS statement t_{CTAS}. These two are then combined to calculate the slowdown, resulting from the use of an AAV compared to a CTAS statement Slowdown $= \frac{t_{\text{CTAS}}}{t_{\text{AAV}}}$ and are represented on the Y-Axis.

Looking at the slowdown graph, it becomes apparent that there is a significant write overhead when using an AAV compared to a CTAS. This can partially be attributed to the overhead that exists when using a AAV in general, which would be:

1. Writing `DataChunk`s of max 1024 tuples one `DataChunk` at a time instead of batching them.

2. Maintaining cache metadata like the row count

3. Not utilizing the query execution capabilities of the calling thread. In C++ , DuckDB queries are executed by the calling thread and the DuckDB thread pool (one worker thread for each Central Processing Unit (CPU) core, which in our case are 16vCPUs). However, when using AAVs, the calling thread is not used. As a result, we are missing $\frac{1}{17}$ of the processing power of a "normal" query.

However, a substantial part of the overhead also stems from accelerated views using the DuckDB `Appender` to write to the cache table. Writing data to a table using the DuckDB `Appender` is single-threaded [1], as the appended is a high-level utility not intended for larger insertions while writing data to a table using a CTAS statement uses multiple threads. This difference becomes especially apparent when writing to disk, where the data has to be compressed first, further highlighting the importance of having multiple threads to persist the table data.

The overhead is less pronounced for an in-memory AAV than for persisted Accelerated Approximate Views (AAVs). The difference in performance between disk-based and in-memory can at least partially be attributed to the absence of compression for in-memory catalogs.

[1] Client context append is called by FlushInternal of the Appender `https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/main/client_context.cpp#L1132` which then performs single-threaded table appends `https://github.com/duckdb/duckdb/blob/1f98600c2cf8722a6d2f2d805bb4af5e701319fc/src/storage/data_table.cpp#L852-L854`

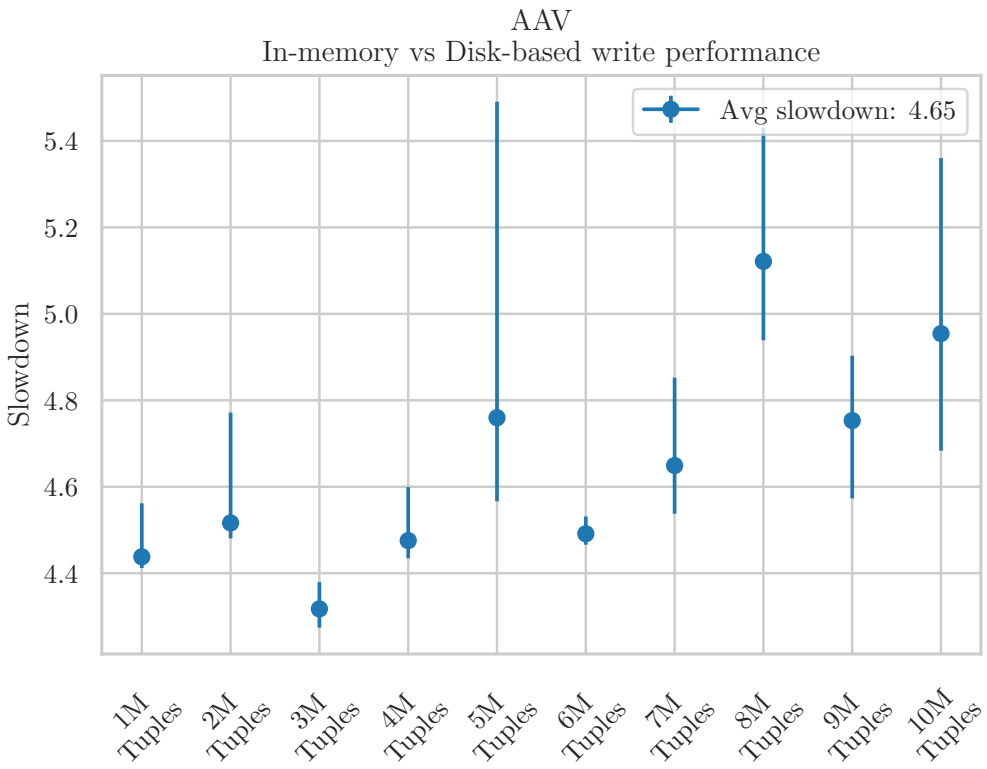


Figure 6.6: Comparison of the write speeds of in-memory and disk-based accelerated views. We plot the slowdown resulting from disk-based storage compared to in-memory storage.

Finally, a comparison of in-memory and on-disk accelerated approximate views (see Figure 6.6) shows that persisting accelerated views come with a penalty of a factor of roughly 4.65. However, this is for queries that take almost no time to execute but still generate much data. The overhead would be less pronounced for queries where the processing time dominates the writing time.

> **Opportunity 6.** One way to improve the current situation would be to imitate DuckDB's approach to writing data. This can be done by replacing the accelerated views cache table writing logic with the functionality used by CTAS. We need to extend the CTAS parallel writer to support the following features: the collection of cache statistics and the ability to pause cache insertions when the cache building has been paused.
>
> This can be done by extending DuckDB's **PhysicalInsert** operator to enable it to also collect cache statistics and support the pausing of its cache building by suspending its pipeline. This change, however, comes with multiple challenges.
>
> Firstly, because data is inserted in parallel, we cannot be sure that the **first** n tuples have been inserted because n tuples have been saved to disk. This, however, can potentially be solved by only considering tuples to be persisted if the entire batch they are associated with has been completely written to disk. Secondly, we have to change the way the cache data is read. While currently, each DataChunk is written in a different transaction, this will no longer be the case for the new implementation. Therefore, a new table reading operator has to be implemented, which also supports the reading of table data that a transaction has not committed yet (see Section 4.3.2 for a rough outline).

6.4 Query Materialization

In the following, we will investigate the overhead of executing a query and receiving/saving its result using different methods:

1. **Materialized Query**: This will run a statement until completion and collect all data in a buffer before returning to the user.
2. **CTAS:** Uses a CTAS statement to execute a query and save its result in a table.
3. **Accelerated Approximate Views:** use an AAV to run a query and cache its result.

4. **Streaming Query + Appender:** This is intended to model the internal cache creation behavior of AAVs, where an `Appender` is used to persist query data to a table. However, in this case, the calling thread also executes the query and inserts the query results into a table. Additionally, we do not flush the `Appender` after every DataChunk and do not collect cache metadata.

The previously defined benchmarking queries are then executed using these methods. For execution methods returning data to the caller (streaming/materializing queries), we measure the time starting from the query execution and ending after the caller consumes all data. For AAV and CTAS, we measure the time it takes to create the cache/table.

6.4.1 C++ Results

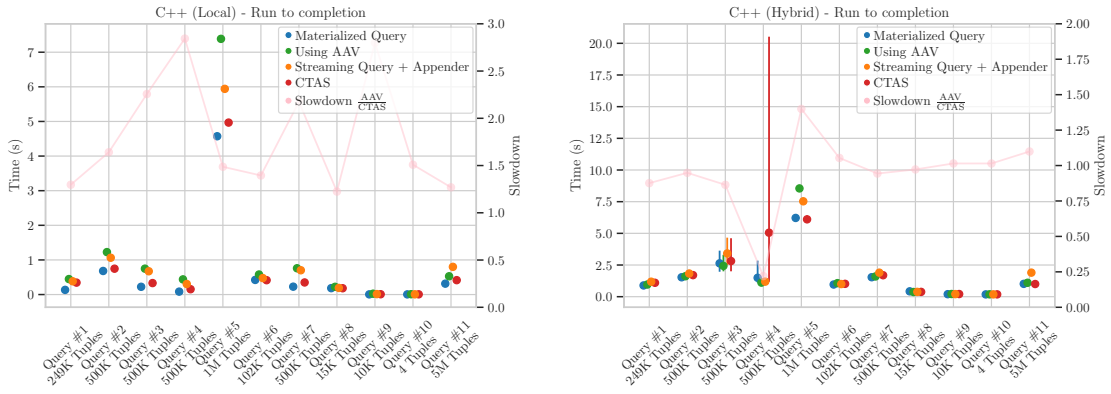


Figure 6.7: Execute queries until the query has completed running or the cache has completed building.

A general trend shows materialized queries completing quicker than the other access methods, which is expected, as this method has the least processing overhead (see Figure 6.7). This is followed by CTAS because CTAS supports inserting data using multiple threads, as opposed to AAVs and the *streaming query + Appender* execution method.

Query #5 takes the most time and is a query that produces significantly more data and processing power. Here, you can see a more considerable difference in processing speed when comparing DuckDB native execution methods (CTAS, materialized query) with AAV-associated execution methods (AAV, Appender). The overhead introduced by the appended can be attributed to the single-threaded nature of the table insertions. Compared to the Appender, the overhead added from using an AAV results from flushing

after every DataChunk and cache metadata maintenance. Additionally, the query calling thread does not participate actively in query execution.

> **Opportunity 7.** The need to flush the Appender after every DataChunk stems from the desire to save the generated data in the cache table as quickly as possible. This is done, so queries reading the first n tuples from the AAV do not have to wait for the appended flush limit to be reached before being able to read.
>
> However, because queries accessing an accelerated view specify the `LIMIT` they want to read from, we know when it is necessary to flush. That way, we would avoid unnecessary Appender flushes and only write data to disk when we hit the Appender buffer or reach a limit specified by a query accessing the AAV.

However, the results differ when looking at Query #11, which produces even more tuples than Query #5. However, the query is less complex. Both for hybrid and local execution AAVs beat the streaming query combined with the Appender. I, unfortunately, have no explanation why an AAV would be faster than the Appender, as AAVs do not utilize the calling thread and do not support bulk insertions. This particular experiment has been repeated multiple times to ensure this is not an outlier, even though each benchmark is run 10 times; however, the results are consistent.

6.4.2 Wasm Results

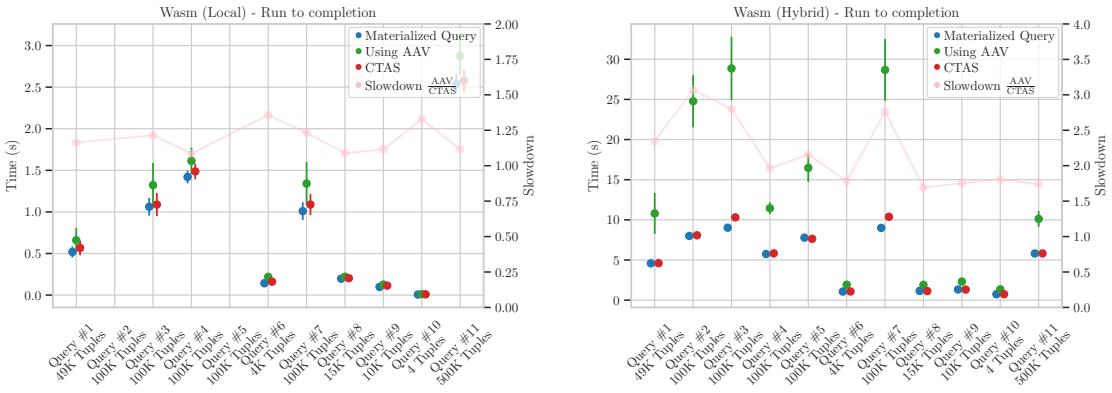


Figure 6.8: Execute queries until the query has completed running or the cache has completed building.

We can observe slightly better results for local-only execution in Wasm. However, Query #2 and Query #5 never finish executing. This has not been fixed because of time

constraints and a current limitation in the MotherDuck Wasm build, which results in linking errors when building its Wasm extension in debug mode. Generally speaking, we can see that AAVs are outperformed by both CTAS and the materialized query, however not by a considerable margin.

The results get worse for Wasm queries containing remote data, where 7 out of the 11 AAV queries are executed much slower than materialized queries and CTAS statements. This can be explained by the bridges that transfer remote/local data during query execution. Because sending data through a bridge blocks a pipeline and pauses query execution of the cache creation query, the AAV stops executing said query and yields control back to the event loop to execute other queries. However, by doing this, every time a bridge blocks, we lose the remaining allocated cache creation execution time, and the cache creation process can only continue once it has been re-scheduled and is executed again.

7

Limitations and Future Work

In this thesis, we implemented a new database operator that enabled the partial caching of views. While AAVs are in a working state, they can be improved. The following will go into some limitations of the current work and provide suggestions to remedy those. During this thesis, multiple contributions to DuckDB have been made to facilitate and or improve the experience of accelerated views [1] [2] [3] [4] [5] [6].

7.1 Hybrid Caching

The current implementation of AAVs is a client-side cache in a hybrid query environment. A hybrid cache, on the other hand, is a cache that is created both on the server and the client. Because server-side resources are less restricted than client-side resources, caches can become larger and satisfy more queries. This is especially true in Wasm environments, which currently cannot save catalogs on disk. Additionally, programs in Wasm cannot use more than 4GB of memory [29], reducing the available space for caching even further.

One possible approach for a hybrid cache implementation would be to build the cache on the server instead of building the cache on the client and then mirror the server's cache to the client. This allows falling back to the server whenever the client-side cache is not sufficiently large. Because all the planning is done client-side, the optimizer needs information about the server-side cache, which has to be synced to the client. This information entails, amongst others, the state the server-side cache is currently in (building,

[1] https://github.com/duckdb/duckdb/pull/10893
[2] https://github.com/duckdb/duckdb/pull/12436
[3] https://github.com/duckdb/duckdb/pull/13136
[4] https://github.com/duckdb/duckdb/pull/13161
[5] https://github.com/duckdb/duckdb/pull/13326
[6] https://github.com/duckdb/duckdb-node/pull/54

aborted because the cache got too large, etc.) and the number of already cached tuples. Only with this information can the optimizer make an informed decision. This decision should be made during the client-side optimization phase. There is an opportunity to restart an already running query using the *query restart* hook of DuckDB. However, this can only be done once for each query, and there would have to be a guarantee that the query is executed successfully the next time. In this context, we will assume that the decision on which cache to use can only be made during local planning.

The previously synced information is ultimately used to decide the following:

1. If the client-side cache is/will be large enough: use the *client's cache* (directly or delayed)
2. If the server-side cache is/will be large enough: use the *server's cache* (directly or delayed)
3. In all other cases: *inline* and *re-execute* the original query

7.2 Query Batching

The column explorer executes ≈ 3 queries to calculate statistics about the user-issued query (see Section 2.7 for more details). These three column explorer queries each read the result of the user query. If the user query could fit into the cache of the Accelerated Approximate View (AAV), accessing the query result is trivial. However, if that is not the case, the user query has to be re-executed once for each column explorer query. This could be avoided by using multi-query optimization [49] where queries are submitted and optimized in batches, allowing for finding a global optimal plan and identifying sharing opportunities (in this case, the same subquery used in multiple column explorer queries) with batches. This is similar to the automatic CTE materialization features introduced in DuckDB 1.1 [21] but would be less explicit and work across multiple statements. This would, however, most likely need to materialize the shared CTE, which introduces additional overhead.

7.3 Analyze non-MotherDuck UI workloads

While I collected statistics for queries executed in the MotherDuck UI, this covers only a part of the queries executed in MotherDuck and leaves out other APIs, like Python, Java Database Connectivity (JDBC), etc. It is not unreasonable to assume that the usage

patterns will vary between the different APIs. The MotherDuck UI, for instance, will likely be used more often for exploitative data analysis.

Collecting data from different APIs would allow me to draw more definitive conclusions about the queries run in MotherDuck and get a better-formed model of how declarative caching could improve user workloads. This is especially true when extending the current caching solution with a hybrid cache, allowing for the execution of complex queries generating more data.

Additionally, collecting logical and physical query plans would allow for more detailed investigations about the most commonly used database operators, enabling more targeted optimizations of said operators. Finally, the usage of DuckDB (table)functions and expressions has not been investigated. Investigating these and comparing the results with [66] would provide further insights into how the MotherDuck UI and DuckDB are used.

7.4 Persisting Accelerated Views in Wasm

Currently, it is impossible to persist accelerated views with DuckDB Wasm [3]. This prevents any caching of queries across browser tabs or after a tab reload. Adding support for Origin private file system (OPFS) [56], a browser native file system, in DuckDB, would enable cache (meta-)data persistence and would thereby allow for the creation of data apps that work offline and retain data even after reopening or reloading the app.

7.5 Technical limitations

Throughout the previous sections, I have identified multiple opportunities to improve the current implementation of AAVs, beginning with Opportunity 1, which explores cross-user query result caching. However, the viability and effectiveness of this have to be investigated further before pursuing anything in that direction.

AAV could be improved meaningfully by creating an optimistic runtime decision-making process, which delays query inlining as long as possible. Instead of deciding that the AAV's query should be inlined during optimization if accessed by a non-satisfiable query (see guaranteed satisfiability in Definition 1), the inlining is done during query execution and only after the AAV has run out of caching space. If that occurs, we fall back to inlining the query. Otherwise, we wait until the cache has completed building and read from the cache instead (see Opportunity 2 for more details).

Opportunity 4 is a MotherDuck UI-centric improvement that would be obsolete if Opportunity 2 is implemented. However, until then, pausing the cache creation process while executing queries that are not *guaranteed satisfiable* will allow these queries to finish quicker, as they are not interrupted by the background cache creation process.

The execution speed of the MotherDuck UI could be further improved by implementing Opportunity 3, which introduces the concept of *approximate count()* queries, which get more accurate the longer the AAV-building process has run and have the benefit of returning quicker than a `count(*)` query.

The main reason for AAVs performing poorly in Wasm is the lack of a central task executor for background tasks in DuckDB. By implementing a centralized task executor, similar to the one already present in C++, we could remove the currently used background execution driver and replace it with a fairer and more performant centralized solution supporting the execution of background tasks (see Opportunity 5 for details).

The last two opportunities concern improving accelerated views' (currently single-threaded) write operations. While Opportunity 7 would not use multiple threads for writing data to disk, it would batch appends and only flush if a query is interested in reading data saved solely in the `Appender`'s buffer. Opportunity 6, on the other hand, has the goal of reusing the multi-threaded table writing capabilities of DuckDB to improve the data insertion speed of accelerated views.

7.6 Query optimization improvements

As mentioned in Section 4.3.2, every time an AAV is accessed, the query accessing it references the AAV as a union of the view's underlying query and cache. This is done so the DuckDB optimizer optimizes both possibilities before we decide which side to keep. While having a union operator in every query plan accessing an accelerated view is not optimal, it is unavoidable. This is because the decision of which side to keep has to be performed after optimizations such as limit push-downs have been performed, as a limit push-down, for instance, can change whether a query accessing an AAV is considered *guaranteed satisfiable*. As a result, the union has to be kept, but this presents several challenges:

For instance, consider the AAV created in Listing 7.1. When accessing it with a limit of 10, the query will be re-written as a `UNION ALL` represented by the last statement in the code block.

```sql
-- Create result
CREATE RESULT ordered_ducks AS FROM migrated_ducks ORDER BY duck_id;
-- Access result
FROM ordered_ducks LIMIT 10;

-- Will be re-written as the following query
FROM
  (
    FROM cache.migrated_ducks
    UNION ALL
    (FROM migrated_ducks ORDER BY duck_id)
  )
LIMIT 10;
```

Listing 7.1: Union access with `LIMIT`

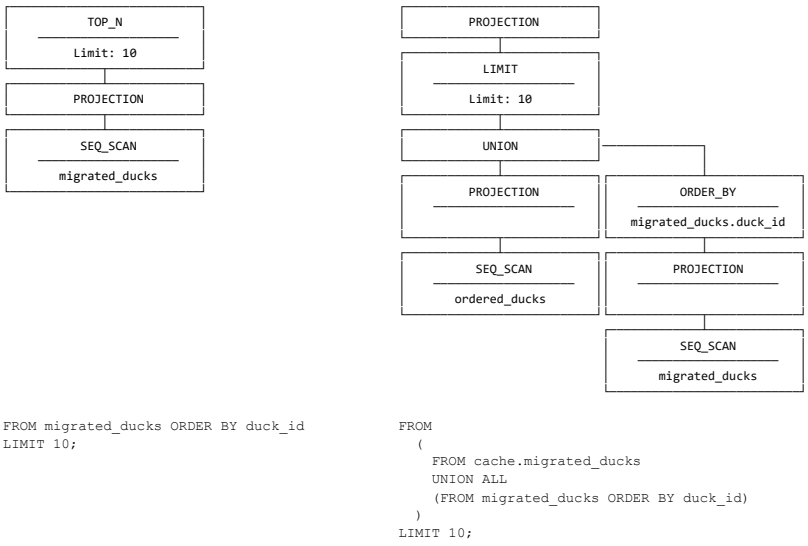


Figure 7.1: Optimization applied to the underlying query of Listing 7.1 (left), compared to the same query when used in a `UNION ALL` (right).

First, let's look at the optimized query plan of the query underlying the AAV if a `LIMIT 10` is applied to it. This can be seen on the left of Figure 7.1. Because the query contains an `ORDER BY` as well as a `LIMIT` operator, DuckDB optimizes this into a `TOP_N` operator. The same query is used as a subquery of a `UNION ALL` operator on the right hand, but the limit is applied to the result of the `UNION`. There is, however no reason not to copy the limit past the `UNION ALL` and apply it to both queries making up the union. This is because we are only interested in the top 10 tuples of each side of the union (all other

results beyond that will be discarded by the limit on top of the union anyway). Copying the same limit to the sub-queries of the union enables new optimizations like the `TOP_N` operator. However, DuckDB does not do this, making it less efficient to execute the same query within a `UNION ALL` compared to executing it alone. One way to work around this would be to lift the `ORDER BY` of the AAV's query out of the right side of the union and apply it to the union instead, thereby triggering the DuckDB optimization. While this is the only optimization I have found that does not work with `UNION` s, there might be more, making the current way of reading from a AAV and falling back to the original query less efficient.

7.7 Automatic cache evictions

While it is possible to give AAVs an enforced memory limit, automatically evicting cache entries has not been implemented yet. Different potential cache eviction strategies have been introduced in Section 4.3.4 and Section 3.2. The most promising approach is thereby an eviction strategy that takes both the access patterns (e.g., LRU or LFU) and the cache age into account.

7.8 Evaluation limitations

Due to time constraints and the overhead of having to implement each benchmark in two different environments (C++ and Wasm), the evaluation of the AAV operator is not as complete as I intended. The following section covers some unfinished benchmarking and evaluations.

7.8.1 AAV performance under varying network conditions

All benchmarking has been run on an AWS EC2 instance with a network speed of up to 10Gbps. However, the maximum network throughput possible was not necessarily utilized, as the benchmarking server is located in Frankfurt, and the MotherDuck servers are in the US-East region, which might have reduced the possible network throughput. Regardless of the network speed, this benchmarking setup does not express the "average" user situation.

Benchmarking the AAV under more network conditions by introducing latency and bandwidth restrictions would allow me to assess the background cache creation mechanism and its efficiency under different network conditions. I suspect that a slower network might lead to different benchmark results for especially Wasm workloads, as the background

materialization driver (see Section 4.3.1.2) in Wasm could potentially under-utilize it's assigned execution timeslots as the cache creation query is blocked, because of unfinished down/uploads.

7.8.2 AAV performance with different DuckDB settings

One of the hypotheses for the overhead introduced by AAVs is that AAVs write to disk using only a single thread. More benchmarks with varying thread counts in DuckDB would have been necessary to validate this hypothesis.

7.8.3 Bypassing AAVs

Creating an AAV has an overhead associated with it that is especially apparent when using a disk-based storage backend. Because of this overhead, it might be more efficient not to cache some queries because the overhead associated with caching is more significant than re-executing the query. Data on when this is true for remote and local queries would allow the optimizer to decide when to cache a query associated with a AAV and when not to cache the view query. This would be especially advantageous for hybrid caches, where users would have to pay for the server-side cache storage they use without benefitting from it.

7.8.4 Representative queries

While this thesis includes a user study that analyzed the queries run in the MotherDuck UI, it makes no claims to have created an unbiased, generalizable data warehouse benchmark. The queries used to evaluate AAVs are an approximation of the queries run in the MotherDuck UI but tend to be a bit more complex and return more tuples than the average MotherDuck UI query. Additionally, no workloads sent through any of the other MotherDuck APIs have been captured in the user study. The creation of a truly representative data warehouse benchmark would, amongst other things, entail analyzing the reported results and datasets of [66] [46] [47] [35] in combination with a more detailed user study that captures all MotherDuck APIs.

7.8.5 Standard Benchmarks

While it was decided not to use a standard benchmark to evaluate AAVs, this was, in hindsight, a mistake. While the SSB does not conform with the query types executed in MotherDuck, it makes the results of this thesis more comparable and guarantees a more thorough and unbiased coverage.

8

Conclusion

To answer **RQ1** on how to design an SQL extension that facilitates the creation and management of a cache-backed view, we introduced a SQL concept called `RESULT` s. The SQL extension enables users to create and manage results and result-adjacent configuration options like cache size cache and the cache building interval. Following **RQ1.a**, access to the cache is optimized so that queries can read cached results as early as possible. For this, the optimizer analyzes the query accessing an AAV for a `LIMIT` operator and uses this to decide how to answer the query. There are more, however not yet implemented, optimizations that would improve the current state of AAVs by creating an `approximate_count` or by creating an optimization rule that estimates the benefits of creating a cache for an AAV.

To facilitate the creation of a background cache, a custom background execution driver for AAVs has been created. This driver works in multi-threaded C++ environments by utilizing the DuckDB thread pool and single-threaded Wasm environments by periodically allocating time for the cache creation process in the JavaScript event loop.

The old cache-table-based caching solution used in the MotherDuck UI has been replaced with a newer AAV based solution and validated. Replacing the old caching approach simplified the application code, as no manual cache management must be done. Instead, all caching decisions are made by the AAV.

The time-to-first-tuple metric in C++ has been improved because of the purposefully buffer-less design of AAVs. Time-to-first-tuple measures the time taken after issuing a query until the first tuple is emitted. AAVs outperform not only CTAS statements but, most of the time, also DuckDB's Streaming query API which is designed to emit tuples as early as possible. Cases in which the Streaming API outperforms AAVs are queries that return fewer result tuples because the *query end* works as a preemptive buffer flush.

Response times in Wasm have not been improved because the granularity of the background execution driver was too high to improve the time-to-first-tuple metric. However, reducing the execution times increases the overhead of AAVs, as fewer tasks are executed without switching contexts. To remedy this, a new execution model for DuckDB Wasm has been proposed, and the first steps towards this have been taken by merging a more predictable scheduler for Wasm [1].

The general overhead of AAVs is still non-negligible and can mainly be attributed to non-parallel writes for C++ environments. This is especially true for disk-based AAVs where the single-threaded nature of the inserts is slowed down even more by compression. The already single-threaded Wasm environment does not suffer from this problem. However, the custom background execution driver necessary for single-threaded background execution introduces more overhead and idle times, thereby reducing Wasm performance.

Solutions for single-threaded inserts and a more inefficient Wasm execution model have been proposed. They would remedy the problems by (1) re-using and adapting DuckDB's multi-threaded writer and (2) creating a new Wasm tasks executor, allowing for fairer scheduling of queries and background tasks, thereby removing the need for a background execution driver and its associated overhead.

[1] `https://github.com/duckdb/duckdb/pull/13326`

References

[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. "Column-oriented database systems." en. In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1664–1665. URL: https://dl.acm.org/doi/10.14778/1687553.1687625 (visited on 07/17/2024) (10).

[2] Mehmet Altınel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. "Cache Tables: Paving the Way for an Adaptive Database Cache." In: *Proceedings 2003 VLDB Conference*. Ed. by Johann-Christoph Freytag, Peter Lockemann, Serge Abiteboul, Michael Carey, Patricia Selinger, and Andreas Heuer. San Francisco: Morgan Kaufmann, 2003, pp. 718–729. URL: https://www.sciencedirect.com/science/article/pii/B9780127224428500690 (visited on 03/15/2024) (36, 42, 43).

[3] Eiichi Arikawa. *Add OPFS (Origin Private File System) Support to the Latest Version of duckdb-wasm*. en. Sept. 2024. URL: https://github.com/duckdb/duckdb-wasm (visited on 09/17/2024) (24, 101).

[4] RJ Atwal, Peter Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves LeMaout, Boaz Leskes, Yao Liu, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, and Yannick Welsch. "MotherDuck: DuckDB in the cloud and in the client." en. In: (2024) (1, 20, 36, 44).

[5] AWS. *Compute – Amazon EC2 Instance Types – AWS*. en-US. 2024. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 09/23/2024) (82).

[6] AWS. *Four new EC2 High Memory instances with up to 12TB of memory are now available with On-Demand and Savings Plan purchase options*. en-US. May 2021. URL: https://aws.amazon.com/about-aws/whats-new/2021/05/four-ec2-high-memory-instances-with-up-to-12tb-memory-available-with-on-demand-and-savings-plan-purchase-options/ (visited on 07/17/2024) (8).

[7] Ilaria Battiston, Kriti Kathuria, and Peter Boncz. "OpenIVM: a SQL-to-SQL Compiler for Incremental Computations." In: *Companion of the 2024 International Conference on Management of Data*. arXiv:2404.16486 [cs]. June 2024, pp. 516–519. URL: http://arxiv.org/abs/2404.16486 (visited on 07/26/2024) (12).

[8] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. "How is the weather tomorrow? towards a benchmark for the cloud." In: *Proceedings of the Second International Workshop on Testing Database Systems*. DBTest '09. New York, NY, USA: Association for Computing Machinery, June 2009, pp. 1–6. URL: https://dl.acm.org/doi/10.1145/1594156.1594168 (visited on 10/11/2024) (28).

[9] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. "Efficiently updating materialized views." In: *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*. SIGMOD '86. New York, NY, USA: Association for Computing Machinery, June 1986, pp. 61–71. URL: https://dl.acm.org/doi/10.1145/16894.16861 (visited on 08/13/2024) (6).

[10] Martin Boissier, Carsten Alexander Meyer, Timo Djürken, Jan Lindemann, Kathrin Mao, Pascal Reinhardt, Tim Specht, Tim Zimmermann, and Matthias Uflacker. "Analyzing Data Relevance and Access Patterns of Live Production Database Systems." In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 2473–2475. URL: https://doi.org/10.1145/2983323.2983336 (visited on 10/12/2024) (28).

[11] P. Boncz, M. Zukowski, and N. Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: 2005. URL: https://www.semanticscholar.org/paper/MonetDB-X100%3A-Hyper-Pipelining-Query-Execution-Boncz-Zukowski/72c78448fd8630545a3582c8e24cd7b88 (visited on 08/05/2024) (10).

[12] Chungmin Melvin Chen and Nicholas Roussopoulos. "The implementation and performance evaluation of the ADMS query optimizer: integrating query result caching and matching." In: *Proceedings of the 4th international conference on extending database technology: Advances in database technology*. EDBT '94. Berlin, Heidelberg: Springer-Verlag, May 1994, pp. 323–336. (Visited on 08/28/2024) (37, 66).

[13] E F Codd. "A Relational Model of Data for Large Shared Data Banks." en. In: 13.6 (1970) (5, 6).

[14] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. "The

Snowflake Elastic Data Warehouse." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 215–226. URL: https://dl.acm.org/doi/10.1145/2882903.2903741 (visited on 10/22/2024) (44, 45).

[15] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. "Semantic Data Caching and Replacement." In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 1996, pp. 330–341. (Visited on 09/18/2024) (66).

[16] J. R. Dash and R. N. Ojala. "IBM Database 2 in an Information Management System environment." In: *IBM Systems Journal* 23.2 (1984). Conference Name: IBM Systems Journal, pp. 165–177. URL: https://ieeexplore.ieee.org/document/5387772 (visited on 10/15/2024) (42).

[17] Aaron Davidson and Andrew Or. "Optimizing shuffle performance in spark." In: *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep* (2013) (10).

[18] Denodo. *Cache Module - Virtual DataPort Administration Guide*. URL: https://community.denodo.com/docs/html/browse/latest/vdp/administration/cache_module/cache_module (visited on 07/29/2024) (50).

[19] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. "Caching multidimensional queries using chunks." In: *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. SIGMOD '98. New York, NY, USA: Association for Computing Machinery, June 1998, pp. 259–270. URL: https://dl.acm.org/doi/10.1145/276304.276328 (visited on 10/13/2024) (36, 41, 42).

[20] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. "Client-Server Paradise." In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 1994, pp. 558–569. (Visited on 10/22/2024) (41).

[21] DuckDB. *Announcing DuckDB 1.1.0*. en. Sept. 2024. URL: https://duckdb.org/2024/09/09/announcing-duckdb-110.html (visited on 10/19/2024) (100).

[22] *emscripten.h — Emscripten 3.1.65-git (dev) documentation*. July 2024. URL: https://emscripten.org/docs/api_reference/emscripten.h.html?highlight=emscripten_async_call (visited on 07/28/2024) (56).

[23] DB-Engines. *DB-Engines Ranking*. en. Aug. 2024. URL: https://db-engines.com/en/ranking (visited on 08/05/2024) (6).

[24] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. "Performance tradeoffs for client-server query processing." In: *SIGMOD Rec.* 25.2 (June 1996), pp. 149–160. URL: https://dl.acm.org/doi/10.1145/235968.233328 (visited on 08/05/2024) (6, 36, 40, 41).

[25] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: the complete book*. eng. 2. ed., internat. ed. Pearson international edition. Upper Saddle River, NJ: Pearson/Prentice Hall, 2009 (5).

[26] Goetz Graefe and William J McKenna. "Extensibility and search efficiency in the Volcano Optimizer Generator." In: 1993 (37).

[27] James R. Groff and Paul N. Weinberg. *SQL, the complete reference*. en. Berkeley, Calif: Osborne/McGraw-Hill, 1999 (5).

[28] Ashish Gupta and Inderpal Singh Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications." en. In: *Materialized Views*. Ed. by Ashish Gupta and Inderpal Singh Mumick. The MIT Press, May 1999, pp. 145–158. URL: https://direct.mit.edu/books/book/2853/chapter/77321/Maintenance-of-Materialized-Views-Problems (visited on 07/26/2024) (6).

[29] Andreas Haas, Jakob Kummerow, and Alon Zakai. *Up to 4GB of memory in WebAssembly · V8*. May 2020. URL: https://v8.dev/blog/4gb-wasm-memory (visited on 10/17/2024) (99).

[30] Theo Haerder and Andreas Reuter. "Principles of transaction-oriented database recovery." In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. URL: https://dl.acm.org/doi/10.1145/289.291 (visited on 08/13/2024) (5).

[31] Alex Hall. *Caching & Reuse of Subresults across Queries*. en. Sept. 2024. URL: https://www.firebolt.io/blog/caching-reuse-of-subresults-across-queries (visited on 10/11/2024) (36, 39).

[32] G. D. Held, M. R. Stonebraker, and E. Wong. "INGRES: a relational data base system." In: *Proceedings of the May 19-22, 1975, national computer conference and exposition*. AFIPS '75. New York, NY, USA: Association for Computing Machinery, May 1975, pp. 409–416. URL: https://dl.acm.org/doi/10.1145/1499949.1500029 (visited on 10/23/2024) (45).

[33] Susan Horwitz and Tim Teitelbaum. "Relations and attributes: A symbiotic basis for editing environments." In: *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*. SLIPE '85. New York, NY, USA: Association for Computing Machinery, June 1985, pp. 93–106. URL: https://dl.acm.org/doi/10.1145/800225.806831 (visited on 08/13/2024) (6).

[34] IBM. *Linux on IBM Systems*. en-US. Nov. 2023. URL: https://www.ibm.com/docs/en/linux-on-systems?topic=hesedcpt-workload-description (visited on 10/23/2024) (42).

[35] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. "SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 281–293. URL: https://dl.acm.org/doi/10.1145/2882903.2882957 (visited on 10/11/2024) (27, 33–36, 105).

[36] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. "Not so fast: analyzing the performance of webassembly vs. native code." In: *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '19. USA: USENIX Association, July 2019, pp. 107–120. (Visited on 08/15/2024) (18, 82).

[37] Nodira Khoussainova, Magda Balazinska, Wolfgang Gatterbauer, YongChul Kwon, and Dan Suciu. *A Case for A Collaborative Query Management System*. arXiv:0909.1778. Sept. 2009. URL: http://arxiv.org/abs/0909.1778 (visited on 10/12/2024) (28).

[38] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. "DuckDB-wasm: fast analytical processing for the web." en. In: *Proceedings of the VLDB Endowment* 15.12 (Aug. 2022), pp. 3574–3577. URL: https://dl.acm.org/doi/10.14778/3554821.3554847 (visited on 07/12/2024) (1).

[39] Yannis Kotidis and Nick Roussopoulos. "DynaMat: a dynamic view management system for data warehouses." In: *SIGMOD Rec.* 28.2 (June 1999), pp. 371–382. URL: https://dl.acm.org/doi/10.1145/304181.304215 (visited on 10/22/2024) (37).

[40] Fabian Nagel, Peter Boncz, and Stratis D. Viglas. "Recycling in pipelined query evaluation." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. ISSN: 1063-6382. Apr. 2013, pp. 338–349. URL: https://ieeexplore.ieee.org/document/6544837 (visited on 03/04/2024) (36, 38, 39, 66).

[41] Pat O'Neil, Betty O'Neil, and Xuedong Chen. "Star Schema Benchmark." en. In: (June 2009) (43, 81).

[42] Mosha Pasumansky and Benjamin Wagner. "Assembling a Query Engine From Spare Parts." en. In: *CDMS 2022, 1st International Workshop on Composable Data Management Systems* (Sept. 2022) (39).

[43] Mark Raasveldt. *Move to push-based execution model · Issue #1583 · duckdb/duckdb*. en. URL: https://github.com/duckdb/duckdb/issues/1583 (visited on 08/16/2024) (11).

[44] Mark Raasveldt. “Push-Based Execution in DuckDB.” en. In: (2021) (11).

[45] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database.” en. In: *Proceedings of the 2019 International Conference on Management of Data*. Amsterdam Netherlands: ACM, June 2019, pp. 1981–1984. URL: https://dl.acm.org/doi/10.1145/3299869.3320212 (visited on 03/04/2024) (1, 8, 10).

[46] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. “Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet.” en. In: *VLDB 2024* (2024) (10, 27, 29, 30, 32, 33, 35, 36, 69, 71, 72, 74, 105).

[47] Alexander van Renen and Viktor Leis. “Cloud Analytics Benchmark.” In: *Proc. VLDB Endow.* 16.6 (Feb. 2023), pp. 1413–1425. URL: https://doi.org/10.14778/3583140.3583156 (visited on 08/28/2024) (28–33, 35, 69, 105).

[48] Prasan Roy, Krithi Ramamritham, S. Seshadri, Pradeep Shenoy, and S. Sudarshan. *Don’t Trash your Intermediate Results, Cache ’em*. arXiv:cs/0003005. Mar. 2000. URL: http://arxiv.org/abs/cs/0003005 (visited on 03/04/2024) (36–38).

[49] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. *Efficient and Extensible Algorithms for Multi Query Optimization*. arXiv:cs/9910021. Oct. 1999. URL: http://arxiv.org/abs/cs/9910021 (visited on 06/21/2024) (100).

[50] Peter Scheuermann, Junho Shim, and Radek Vingralek. “WATCHMAN: A Data Warehouse Intelligent Cache Manager.” In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB ’96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 1996, pp. 51–62. (Visited on 10/15/2024) (37, 46).

[51] Itamar Sher. *Wasm cross-platform support · Issue #24166 · grpc/grpc*. en. Sept. 2020. URL: https://github.com/grpc/grpc/issues/24166 (visited on 08/06/2024) (20).

[52] J. Shim, P. Scheuermann, and R. Vingralek. “Dynamic Caching of Query Results for Decision Support Systems.” In: *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*. July 1999, pp. 254–263. URL: https://ieeexplore.ieee.org/document/787641 (visited on 08/29/2024) (36, 45, 46, 66).

[53] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. en. Seventh edition. New York, NY: McGraw-Hill, 2020 (7).

[54] Vik Singh, Jim Gray, Ani Thakar, Alexander S. Szalay, Jordan Raddick, Bill Boroski, Svetlana Lebedeva, and Brian Yanny. *SkyServer Traffic Report - The First Five Years*. arXiv:cs/0701173. Jan. 2007. URL: http://arxiv.org/abs/cs/0701173 (visited on 10/12/2024) (28, 30, 31, 33).

[55] Snowflake. *Using Persisted Query Results | Snowflake Documentation*. 2024. URL: https://docs.snowflake.com/en/user-guide/querying-persisted-results (visited on 07/29/2024) (50).

[56] SQLite. *Persistent Storage Options*. URL: https://sqlite.org/wasm/doc/trunk/persistence.md (visited on 10/10/2024) (101).

[57] Michael Stonebraker. "Implementation of integrity constraints and views by query modification." In: *Proceedings of the 1975 ACM SIGMOD international conference on Management of data*. SIGMOD '75. New York, NY, USA: Association for Computing Machinery, May 1975, pp. 65–78. URL: https://dl.acm.org/doi/10.1145/500080.500091 (visited on 08/13/2024) (5, 6).

[58] Michael Stonebraker, Jeff Anton, and Eric Hanson. "Extending a database system with procedures." In: *ACM Trans. Database Syst.* 12.3 (Sept. 1987), pp. 350–376. URL: https://dl.acm.org/doi/10.1145/27629.27631 (visited on 08/29/2024) (36, 55).

[59] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. "The design and implementation of INGRES." In: *ACM Trans. Database Syst.* 1.3 (Sept. 1976), pp. 189–222. URL: https://dl.acm.org/doi/10.1145/320473.320476 (visited on 10/23/2024) (45).

[60] Streamlit. *Connecting to data - Streamlit Docs*. URL: https://docs.streamlit.io/ (visited on 07/12/2024) (1).

[61] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. "Cache-on-demand: recycling with certainty." In: *Proceedings 17th International Conference on Data Engineering*. ISSN: 1063-6382. Apr. 2001, pp. 633–640. URL: https://ieeexplore.ieee.org/document/914878 (visited on 08/29/2024) (36, 38).

[62] Jordan Tigani. *Big Data is Dead*. en. Feb. 2023. URL: https://motherduck.com/blog/big-data-is-dead/ (visited on 08/20/2024) (10).

[63] Jordan Tigani. *Redshift Files: the Hunt for Big Data*. en. Aug. 2024. URL: https://motherduck.com/blog/redshift-files-hunt-for-big-data/ (visited on 08/20/2024) (10, 29–31, 71).

[64] TPC. *TPC-DS Homepage*. URL: https://www.tpc.org/tpcds/default5.asp (visited on 09/06/2024) (81).

[65] TPC. *TPC-H Homepage*. URL: https://www.tpc.org/tpch/ (visited on 09/06/2024) (81).

[66] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. "Get Real: How Benchmarks Fail to Represent the Real World." In: *Proceedings of the Workshop on Testing Database Systems*. DBTest '18. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 1–6. URL: https://dl.acm.org/doi/10.1145/3209950.3209952 (visited on 10/11/2024) (27, 29, 32–34, 101, 105).

[67] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. "Building an elastic query engine on disaggregated storage." In: *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*. NSDI'20. USA: USENIX Association, Feb. 2020, pp. 449–462. (Visited on 08/28/2024) (28, 35, 36, 44, 45).

[68] W3C. *WebAssembly Core Specification*. May 2019. URL: https://www.w3.org/TR/wasm-core-2/ (visited on 07/26/2024) (18).

[69] Weihang Wang. "Empowering Web Applications with WebAssembly: Are We There Yet?" In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ISSN: 2643-1572. Nov. 2021, pp. 1301–1305. URL: https://ieeexplore.ieee.org/abstract/document/9678831 (visited on 08/15/2024) (18).

[70] *WasmEdge/WasmEdge*. original-date: 2019-11-29T19:00:17Z. Aug. 2024. URL: https://github.com/WasmEdge/WasmEdge (visited on 08/15/2024) (18).

[71] *wasmerio/wasmer*. original-date: 2018-10-11T10:15:53Z. July 2024. URL: https://github.com/wasmerio/wasmer (visited on 07/26/2024) (18).

[72] whatwg. *DOM Standard*. Aug. 2024. URL: https://dom.spec.whatwg.org/ (visited on 08/15/2024) (18).

[73] whatwg. *HTML Standard - Web workers*. Aug. 2024. URL: https://html.spec.whatwg.org/multipage/workers.html (visited on 08/05/2024) (18).

[74] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. "FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS." In: *Proc. VLDB Endow.* 14.11 (July 2021), pp. 2101–2113. URL: https://dl.acm.org/doi/10.14778/3476249.3476265 (visited on 10/16/2024) (36, 43, 44).

[75] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. "Apache Spark: a unified engine for big data processing." en. In: *Communications of the ACM*

59.11 (Oct. 2016), pp. 56–65. URL: https://dl.acm.org/doi/10.1145/2934664 (visited on 07/17/2024) (10).

[76] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. "Answering complex SQL queries using automatic summary tables." In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. SIGMOD '00. New York, NY, USA: Association for Computing Machinery, May 2000, pp. 105–116. URL: https://dl.acm.org/doi/10.1145/342009.335390 (visited on 10/15/2024) (42).

[77] Marcin Zukowski, Mark Van De Wiel, and Peter Boncz. "Vectorwise: A Vectorized Analytical DBMS." en. In: *2012 IEEE 28th International Conference on Data Engineering*. Arlington, VA, USA: IEEE, Apr. 2012, pp. 1349–1350. URL: http://ieeexplore.ieee.org/document/6228203/ (visited on 07/17/2024) (10, 38).

Appendix A

Benchmarking Queries

Variable	Description	Wasm	C++
___SF___	The scale factor of the tables created for the benchmark.	1	5
___LIMIT___	The maximal number of tuples that should be returned for a query. Used as a save-guard against OOM errors	500 000	100 000 000
___ROW_COUNT___	Maximal number of written tuples used to measure the write performance of AAV and CTAS.	Not applicable, as Wasm does not support saving catalogs to disk	10 000 000

Table A.1: Variables used in the following queries

A.1 Set-Up

```
CREATE TABLE table_all_data_types AS
SELECT
    random() as rand,
    hash(random()) as int_hash,
    hash(range)::string as string_hash,
    round(random() * 100, 2) as rounded_random,
    (random() * 10000)::int as int_random,
    substr(hash(random())::string, 1, 8) as short_hash,
```

```
    CASE
        WHEN random() > 0.5 THEN 'Above 0.5'
        WHEN random() > 0.3 THEN 'Between 0.3 and 0.5'
        ELSE 'Below 0.3'
    END as random_category,
    'constant_value' as constant_string,
    current_date as current_date,
    date_add(current_date, to_days((random() * 1000)::int)) as future_date,
    CASE
        WHEN random() > 0.5 THEN TRUE
        ELSE FALSE
    END as random_bool
FROM generate_series(1, 100000 * __SF__) AS range;

CREATE TABLE str_big_double AS
SELECT
    (random() * 10000)::bigint as int_random,
    hash(range)::string as string_hash_1,
    md5(random()::text)::string as string_hash_2,
    repeat(substring(md5(random()::text) from 1 for 8), 50) as
    string_hash_3,
FROM generate_series(1, 100000 * __SF__) AS range;


CREATE TABLE str_date_datetime_bool AS
SELECT
    (random() * 10000)::bigint as int_random,
    hash(range)::string as string_hash_1,
    md5(random()::text)::string as string_hash_2,
    repeat(substring(md5(random()::text) from 1 for 8), 50) as
    string_hash_3,
    current_date() + (FLOOR(random() * 2001) - 1000)::INT as current_date,
    make_timestamp((random() * 1000000000000000)::INT64) as make_timestamp,
    CASE
        WHEN random() > 0.5 THEN TRUE
        ELSE FALSE
    END as random_bool
FROM generate_series(1, 100000 * __SF__) AS t(range);
```

Listing A.1: Setup

A.2 Queries

```
SELECT * FROM remote_benchmark.str_date_datetime_bool WHERE random_bool IS
    TRUE LIMIT __LIMIT__;
```

Listing A.2: Query #1

```
SELECT * FROM remote_benchmark.str_big_double ORDER BY int_random LIMIT
    __LIMIT__;
```

Listing A.3: Query #2

```
SELECT * FROM remote_benchmark.str_date_datetime_bool LIMIT __LIMIT__;
```

Listing A.4: Query #3

```
SELECT * FROM remote_benchmark.table_all_data_types LIMIT __LIMIT__;
```

Listing A.5: Query #4

```
SELECT
  *
FROM
  (FROM remote_benchmark.table_all_data_types LIMIT 200000) a
    JOIN
  (FROM local_benchmark.table_all_data_types LIMIT 200000) b
  ON
    a.int_random = b.int_random
WHERE
  a.rand > b.rand
ORDER BY
  a.int_random, a.rand
LIMIT __LIMIT__;
```

Listing A.6: Query #5

```
SELECT
    t1.rand,
    t1.int_hash,
    t1.string_hash,
    t2.string_hash_1 AS string_hash_big,
    t2.string_hash_3,
    t3.make_timestamp,
    t1.random_category,
    t3.random_bool AS random_bool_date,
    CASE
        WHEN t1.rounded_random > 75 THEN 'High'
        WHEN t1.rounded_random > 50 THEN 'Medium'
        ELSE 'Low'
    END AS random_classification
FROM (FROM remote_benchmark.table_all_data_types LIMIT 200000) t1
JOIN (FROM local_benchmark.str_big_double  LIMIT 200000) t2 ON t1.
    int_random = t2.int_random
JOIN (SELECT DISTINCT * FROM remote_benchmark.str_date_datetime_bool LIMIT
    1000) t3 ON t1.int_random = t3.int_random
WHERE t1.rounded_random > 50 AND t1.random_bool = TRUE
```

```
ORDER BY t1.rand
LIMIT __LIMIT__;
```

Listing A.7: Query #6

```
SELECT * FROM remote_benchmark.str_date_datetime_bool LIMIT __LIMIT__;
```

Listing A.8: Query #7

```
SELECT
    t1.rand,
    t1.int_hash,
    t1.string_hash,
    t1.random_category,
    t2.string_hash_1,
    t2.current_date,
    t2.random_bool
FROM
    remote_benchmark.table_all_data_types t1
JOIN
    local_benchmark.str_date_datetime_bool t2
    ON t1.int_random = t2.int_random
WHERE
    t1.random_bool = TRUE
    AND t2.current_date > CURRENT_DATE() - INTERVAL '30 days'
    AND t1.random_category IN ('Above 0.5', 'Between 0.3 and 0.5')
ORDER BY
    t1.rand
LIMIT 15000;
```

Listing A.9: Query #8

```
SELECT * FROM remote_benchmark.str_date_datetime_bool LIMIT 10000;
```

Listing A.10: Query #9

```
SELECT
    random_bool,
    COUNT(*) AS count_rows,
    AVG(int_random) AS avg_int_random,
    MIN(current_date) AS min_current_date,
    MAX(current_date) AS max_current_date,
    CASE
        WHEN int_random > 5000 THEN 'High'
        ELSE 'Low'
    END AS int_random_category
FROM
    remote_benchmark.str_date_datetime_bool
GROUP BY
```

```
    random_bool,
    CASE
        WHEN int_random > 5000 THEN 'High'
        ELSE 'Low'
    END;
```

Listing A.11: Query #10

```
SELECT
  subquery.string_hash_1,
  subquery.int_random,
  local_benchmark.str_date_datetime_bool.current_date,
  local_benchmark.str_date_datetime_bool.random_bool
FROM
  (
    SELECT * FROM local_benchmark.str_big_double
    WHERE int_random > 1000
    LIMIT 10
  ) AS subquery
  JOIN local_benchmark.str_date_datetime_bool ON 1 = 1
LIMIT __LIMIT__;
```

Listing A.12: Query #11

A.3 Write Benchmark Query

```
SELECT
    random() as rand,
    hash(random()) as int_hash,
    hash(range)::string as string_hash,
    round(random() * 100, 2) as rounded_random,
    (random() * 10000)::int as int_random,
    substr(hash(random())::string, 1, 8) as short_hash,
    CASE
        WHEN random() > 0.5 THEN 'Above 0.5'
        ELSE 'Below 0.5'
    END as random_category,
    'constant_value' as constant_string,
    current_date as current_date,
    date_add(current_date, to_days((random() * 1000)::int)) as future_date,
    CASE
        WHEN random() > 0.5 THEN TRUE
        ELSE FALSE
    END as random_bool
FROM generate_series(1, __ROW_COUNT__) AS range
```

Listing A.13: Write benchmark