POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Development and Optimization of a WebAssembly-Powered Browser Extension for Enhanced Performance and Functionality

Supervisors

Prof. Giovanni MALNATI

Candidate

Pouya HAKIMIFARD

March 2025

Abstract

As web technologies continue to evolve, web applications are becoming more sophisticated and demanding, making performance and security more critical than ever. WebAssembly (WASM) has emerged as a game-changer, offering a low-level, portable bytecode that enables near-native execution speeds, compact code representation, and a secure runtime environment. Unlike JavaScript, which wasn't designed for high-performance computing, WASM provides a powerful alternative, acting as an abstraction over modern hardware to support multiple programming languages and platforms. This thesis explores how WASM can enhance browser-based data processing and analytics by developing an optimized browser extension.

The goal of this research is to leverage WebAssembly to enable efficient, scalable, and offline data analysis, particularly in scenarios where large datasets—such as academic records or research data—need to be processed in real time without relying on a server. The browser extension integrates DuckDB, a high-performance analytics engine, compiled to WASM. This allows users to run complex SQL queries directly in their browser with minimal latency and without compromising security. The project extends and optimizes the WASM-powered version of DuckDB to handle datasets with millions of records, ensuring scalability and responsiveness.

A key challenge tackled in this work is optimizing how large JSON files are loaded and queried within the browser. Since JSON files are widely used in data analysis and research environments, the project implements advanced memory management and compression techniques to reduce load times and minimize memory consumption. Additionally, WASM's inherent security features ensure that high-performance data processing can be done safely within the browser environment.

This thesis also addresses the wider challenge of performing large-scale data analysis entirely on the client side, eliminating the need for backend infrastructure while maintaining consistent performance. By capitalizing on WASM's strengths in execution efficiency and security, the project presents a compelling case for using WASM in real-time, browser-based analytics applications. The result is a fully functional, responsive, and secure extension that enables seamless interaction with large datasets.

In summary, this research highlights the potential of WebAssembly as a powerful tool for high-performance web applications, particularly in handling large-scale JSON data within the browser. By leveraging WASM for efficient client-side data processing, this work demonstrates how browser-based computation can significantly enhance the speed and responsiveness of data queries. While not strictly real-time, these optimizations enable near-instantaneous interactions with large datasets, making complex data analysis more accessible and efficient within a web environment.

Acknowledgements

“first of all, I would like to express my greatest respect and appreciation to my supervisor, Professor Giovanni Malnati, for his invaluable guidance and support throughout this project. His advice and encouragement played a crucial role in the completion of my thesis. I also wish to extend my heartfelt thanks to my family and friends for their support and understanding during this journey. Their encouragement and patience provided me with the strength and motivation to persevere.”

Pouya HakimiFard, Turin , March 2025

Table of Contents

List of Tables

List of Figures

Acronyms

WASM
: WebAssembly

EMCC
: Emscripten Compiler

MVC
: Model-View-Controller

UI
: User Interface

CORS
: Cross-Origin Resource Sharing

DOS
: Denial-of-Service

NaCl
: Native Client

DOM
: Document Object Model

ORM
: Object Relational Mapping

CFI
: Control-Flow Integrity

LLVM

Low-Level Virtual Machine

ISA

instruction set architecture

Chapter 1

Introduction

1.1 Background and Motivation

Browser extensions have become an essential part of modern web applications, helping users automate tasks, improve their browsing experience, and add extra features beyond what browsers offer by default. However, most extensions are built using JavaScript, which, while versatile, struggles with performance when handling large datasets or complex computations. This becomes a major bottleneck for applications that require real-time data processing, analytics, or efficient memory management.

Over the years, developers have tried various approaches to bridge the performance gap between JavaScript and native code. Technologies like ActiveX, Native Client (NaCl), and asm.js attempted to improve execution speed, but each came with its own drawbacks—security risks, platform restrictions, or inefficiencies. Recognizing these challenges, the W3C introduced WebAssembly (Wasm) in 2017 as a game-changing solution. Wasm provides a highly efficient, low-level binary format that allows developers to write performance-critical code in languages like C, C++, and Rust, running it alongside JavaScript at near-native speeds [1].

This thesis explores the use of WebAssembly to develop a high-performance browser extension that integrates DuckDB, an in-memory analytical database, for efficient data processing directly within the browser. By leveraging Wasm's speed, memory efficiency, and security benefits, this project aims to demonstrate how WebAssembly can significantly enhance browser-based analytics and provide a scalable solution for handling large datasets more efficiently than traditional JavaScript-based approaches.

1.2 Key Aspects

This project focuses on several key aspects to enhance the performance and functionality of browser extensions:

Performance Optimization
Utilizing WebAssembly to achieve faster execution speeds compared to JavaScript, especially for computationally heavy tasks.
Efficient Data Processing
Implementing DuckDB to perform in-memory SQL queries, reducing reliance on cloud-based data processing.
Scalability
Ensuring that the extension can handle large datasets (millions of records) while maintaining speed and efficiency.
Seamless Browser Integration
Designing a WASM-powered extension that interacts efficiently with web applications without requiring excessive computational resources.
Memory Management
Optimizing the allocation and deallocation of memory in WASM to prevent memory leaks and improve overall performance.
Security
Running code inside a sandboxed execution environment within the browser this ensures that wasm modules cannot perform unauthorized actions, such as accessing files, network resources, or executing arbitrary system commands, unless explicitly permitted through browser APIs. this makes wasm both fast and secure, making it a significant improvement over previous native execution methods like Native Client (NaCl).[1]

1.3 Objectives

The main objectives of this project are:
Develop a WebAssembly powered browser extension capable of handling large datasets.
Integrate WebAssembly with DuckDB to perform in-memory analytics without relying on external servers.
Optimize data handling and memory management to improve execution time and efficiency.
Compare the performance of WASM vs. JavaScript-based solutions for browser-based analytics.
Ensure scalability and security while maintaining usability within a browser

extension.

1.4 Questions

This thesis aims to answer the following key questions:

1. How does WebAssembly improve the performance of browser extensions compared to JavaScript?
2. What are the benefits of integrating DuckDB with WebAssembly for in-browser data analytics?
3. How can memory management be optimized to support large datasets efficiently in a WASM environment?
4. What are the trade-offs between WebAssembly and traditional JavaScript-based implementations?

1.5 Structure of the Thesis

Chapter 2: Literature Review explores existing research on browser extensions, WebAssembly, and in-browser databases.

Chapter 3: Methodology details the architecture, design choices, and implementation of the WASM-powered extension.

Chapter 4: Implementation and Optimization presents the coding strategies, performance enhancements, and challenges encountered.

Chapter 5: Performance Analysis evaluates execution speed and scalability.

Chapter 6: Conclusion and Future Work summarizes key findings and proposes potential improvements.

Chapter 2

Literature Review

2.1 Browser Extensions

A browser extension is like a little add-on that boosts your web browser's functionality. It can personalize your browsing experience by offering new features, automating tasks, and connecting to other services, making your time online smoother and more efficient.

These extensions work by interacting with the webpage's structure (known as the Document Object Model, or DOM). This lets them change content, add scripts, or tap into the browser's built-in capabilities.[2] They're created using familiar web technologies like:

- **HTML:** Used to build the user interface for the extension.
- **CSS:** For styling and customizing the appearance of extension components.
- **JavaScript:** The core language that handles the logic, lets the extension talk to the browser, and changes webpage content.
- **React.js :**A JavaScript library used for building user interfaces. It allows you to structure the extension's frontend with reusable components, manage state efficiently, and create dynamic and interactive UIs. React can be particularly useful for building extension popups, options pages, or background interfaces that require frequent updates or complex interactions.
- **WASM:** To execute high performance codes in the browser that is used when the performance is essential to process and analyze of large datasets.

2.2 The Significans of WebAssembly

WASM is transforming the way we build web applications by bringing near-native performance directly to the browser. Unlike JavaScript, which can slow down when handling complex computations, WASM runs precompiled binary code, making it significantly faster and more efficient all within a secure, sandboxed environment[3].

2.2.1 Key Points of WebAssembly

- **WASM Performance**
  One of WASM's biggest strengths is speed. While WASM is optimized for speed, it may not always match fully optimized native machine code due to factors like memory access overhead and sandboxing security measures [3].

- **Use Cases**
  WASM is well-suited for compute-heavy applications like gaming, video editing, data analysis, and scientific simulations[1].

- **Supports Multiple Programming Languages**
  WASM isn't just for JavaScript developers. It allows developers to write code in languages like C, C++, Rust, and Go, then compile it into WebAssembly. This means existing high-performance code can be brought into the web without needing to be completely rewritten in JavaScript.

- **Memory Management**
  WebAssembly gives developers precise control over memory usage, which is crucial for handling large datasets efficiently. For example, in our project, we integrate WASM with DuckDB to optimize memory allocation while performing in-browser data analytics, ensuring smooth performance without unnecessary resource consumption.

- **Security and Privacy** Because WASM runs in a sandboxed environment,WASM itself does not have a built-in permissions system like JavaScript APIs do. Instead, it relies on the host environment (e.g., the browser) to enforce security restrictions. It can only interact with system resources through APIs provided by the host (e.g., Web APIs or imported JavaScript functions)[4].
  While WASM can enhance browser extensions, it still requires JavaScript to interact with browser APIs (e.g., storage, networking). WASM alone does not inherently have privileged access.

2.3 Porting C and C++ Code to WebAssembly

WebAssembly (Wasm) is designed to support C and C++ efficiently, making it easier to run high-performance applications on the web. This guide covers essential concepts for C/C++ developers, including porting existing code, platform features, language support, and best practices[5].

2.3.1 Understanding WebAssembly's Platform Features

WebAssembly follows a standard instruction set architecture (ISA), meaning that most portable C/C++ code can be compiled to Wasm with minimal changes. Some key features include:

- **8-bit bytes**, **two's complement integers**, and **little-endian format**, similar to many other platforms.
- **Two architecture variants:**
  - **wasm32 (32-bit)** – Uses an ILP32 model where `int`, `long`, and pointers are 32-bit, while `long long` is 64-bit.
  - **wasm64 (64-bit)** [**Future support planned**] – Uses an LP64 model where `long` and pointers are 64-bit, and `int` remains 32-bit.

Currently, only **wasm32** is supported, with **wasm64** planned for larger address spaces.

2.3.2 Floating-Point Support

WebAssembly follows IEEE 754-2019 standards for floating-point arithmetic:

- **float** and **double** are natively supported.
- **long double** is software-emulated since WebAssembly does not have a built-in quad-precision type.

For best performance and compatibility, using **float** and **double** is recommended.

2.4 C and C++ Language Support in WebAssembly

WebAssembly does not modify the C/C++ language itself, but compiler support plays a key role in its integration. Modern compilers like Clang/LLVM fully support WebAssembly[5]. Some upcoming features include:

- **Multi-threading with shared memory** (planned)
- **Zero-cost exception handling** for C++ (to reduce performance overhead)
- **128-bit SIMD support** for efficient vector operations

2.4.1 Using APIs in WebAssembly

WebAssembly allows C/C++ programs to use high-level APIs, including:

- **Standard C and C++ libraries**
- **OpenGL (via WebGL)**
- **SDL for multimedia applications**
- **pthreads for multi-threading** (once supported)

These APIs rely on WebAssembly's low-level capabilities, which interface with web-based APIs when running in a browser.

2.4.2 ABI Considerations

Currently, WebAssembly does not have a stable ABI for dynamically linked libraries. This means:[5]

- All linked code must be compiled with the same compiler and options.
- Future WebAssembly updates will introduce **dynamic linking** and stable ABIs for improved interoperability.

2.5 Undefined and Implementation-Defined Behavior

2.5.1 Undefined Behavior

WebAssembly does not change C or C++ undefined behavior. For example:

- Unaligned memory access is defined in WebAssembly but may still lead to optimization issues in C/C++.
- Compilers optimize based on the assumption that undefined behavior does not occur, which can lead to unpredictable behavior.

WebAssembly ensures memory safety and prevents sandbox escapes, maintaining security invariants.

2.5.2 Implementation-Defined Behavior

Most implementation-defined behavior depends on the compiler rather than WebAssembly. However, Wasm maintains standard features like:

- **8-bit bytes**
- **Two's complement integers (32-bit and 64-bit)**
- **IEEE-754 floating-point arithmetic**

2.5.3 Portability of Compiled Code

One of WebAssembly's greatest strengths is its **cross-platform consistency**. Due to its limited nondeterminism, Wasm applications behave consistently across different implementations and environments.

2.5.4 Conclusion

For C/C++ developers, WebAssembly offers an efficient way to run native applications in a secure, portable environment. While WebAssembly is still evolving, it already provides strong support for familiar language features and APIs. Future improvements, such as multi-threading, exception handling, and dynamic linking, will make WebAssembly even more powerful for C/C++ development.

Keywords: WebAssembly, C++, Portability, Compilation, API, ABI, Performance

2.6 Managing Memory in WebAssembly

WebAssembly (Wasm) introduces a different approach to memory management than most high-level languages. Unlike JavaScript, Java, or Python—which have built-in garbage collection—Wasm requires developers to handle memory explicitly. This makes it powerful but also introduces challenges, especially for those used to automatic memory management[6]. In this section, we explore how WebAssembly handles memory, how it interacts with host environments, and the best ways to manage it efficiently.

2.6.1 How WebAssembly Handles Memory

WebAssembly uses a **linear memory model**, meaning memory is a single, continuous block of bytes. This memory starts small but can grow in 64KB chunks (called pages) when needed. However, it remains strictly isolated—no program can

access memory outside its allocated space unless explicitly allowed.[7]This design ensures:

- **Security** – WebAssembly modules cannot read or write memory they shouldn't.
- **Stability** – Built-in checks prevent memory corruption.
- **Isolation** – Wasm modules do not interfere with each other or the host system unless given permission.

Even with these safeguards, WebAssembly is not immune to common memory issues like buffer overflows or dangling pointers. Some researchers have proposed additional safety features, such as stricter bounds checking, to further improve memory protection.

2.6.2 How WebAssembly Interacts with the Host Environment

A WebAssembly module does not have direct access to the host system's memory, files, or APIs. Instead, it interacts with the host (such as JavaScript in a browser or a Rust runtime) in two main ways:

1. **Function Calls** – Wasm functions can take numbers (integers or floats) as arguments and return numbers, but passing complex data like strings or arrays requires extra steps.
2. **Shared Memory** – The host can read and write directly into WebAssembly's memory, allowing data exchange through pointers and manual memory management.

Most applications use a mix of both, passing pointers to memory locations where data is stored. This approach keeps data transfers efficient while maintaining WebAssembly's security rules.

2.6.3 Allocating and Freeing Memory in WebAssembly

WebAssembly does not manage memory automatically; developers must allocate and free it manually. The host environment plays a key role in ensuring memory is handled correctly. If memory is not freed properly, programs may experience memory leaks.

2.6.4 Allocating Memory

Many WebAssembly modules provide functions to allocate memory, and tools like `wasm-bindgen` help simplify the process.

2.6.5 Freeing Memory

If memory is not released when it is no longer needed, the program could run out of memory. Languages like Rust solve this by using ownership rules that automatically free memory when it is no longer in use.

To avoid memory issues, developers often use debugging tools like Valgrind to check for leaks. Some projects even implement custom memory allocators to fine-tune performance and efficiency.

2.6.6 Conclusion

Managing memory in WebAssembly requires careful attention, especially compared to higher-level languages with automatic garbage collection. While Wasm ensures strong isolation, developers still need to allocate and free memory properly—especially when sharing data with the host environment. By following best practices and using the right tools, developers can keep their WebAssembly applications both efficient and secure.

2.7 Security in WebAssembly

WebAssembly (Wasm) is designed with two key security goals:[8]

1. **Protect users** from buggy or malicious modules.
2. **Provide developers** with safe programming primitives and mitigations.

2.7.1 Sandboxing and Isolation

- Wasm modules execute in a **sandboxed** environment, preventing direct access to the host system.
- In a browser, Wasm adheres to **same-origin policy** and other web security measures.
- On non-web platforms, security policies (e.g., POSIX model) enforce access controls.

2.7.2 Memory Safety

- **No arbitrary code execution**: Functions and memory addresses are validated before execution[4].

- **Buffer overflows are limited**: Fixed-size local/global variables prevent corruption, though linear memory regions can still be affected.
- **Bounds checking** prevents accessing memory outside allocated regions, reducing memory corruption vulnerabilities.
- **Traps** halt execution on errors like invalid memory access, division by zero, or exceeding stack limits.

2.7.3 Control-Flow Integrity (CFI)

- **Direct function calls**: Only valid, declared functions can be executed.
- **Indirect calls**: Type signatures must match, preventing unintended function execution.
- **Protected call stack**: Prevents return address corruption and stack-based attacks.
- **Code reuse attacks**: Limited due to function-level CFI but still possible against indirect calls.

2.7.4 Additional Security Considerations

- **No execution guarantees**: Potential for **race conditions** (e.g., TOCTOU vulnerabilities).
- **Side-channel attacks**: Possible, including timing attacks, though mitigations may improve over time.
- **Future protections**: Code diversification, memory randomization (ASLR), and bounded pointers could enhance security.

2.7.5 Fine-Grained CFI via Clang/LLVM

- Enabling `-fsanitize=cfi` in Clang/LLVM enhances security by checking function calls at the C/C++ type level.
- This provides stricter CFI protections beyond Wasm's built-in type-based enforcement, though with a small performance cost.

Overall, Wasm's security model significantly reduces traditional vulnerabilities in C/C++ applications, offering a safer execution environment while maintaining high performance.

2.8 Enhancing Browser Extensions with WASM

WASM can really boost the performance of browser extensions by handling heavy tasks more efficiently than JavaScript, especially when it comes to running complex SQL queries or doing resource-intensive computations. In our project, we use WASM alongside DuckDB to run these data queries directly in the browser, which means we don't need a server to process them. This makes everything faster and reduces the load on the server. In fact, there are still cases where server-side help is needed—like for storing data, ensuring security, or handling advanced analytics—but overall, this approach keeps the extension lightweight and super responsive.

2.8.1 Seamless Integration with JavaScript and Web APIs

WASM doesn't replace JavaScript—it works alongside it. In our project, we used emcc to compile C code into WASM, which we then call from JavaScript. This makes it easy to integrate high-performance WASM modules into our web application. For example, in a browser extension built with React, we offload resource-intensive tasks to WASM, ensuring the main app remains smooth and responsive without overloading the browser's main thread.

2.9 DuckDB: The Database Powering The In-Browser Analytics

DuckDB is a super-efficient, in-memory database designed to handle complex data analysis tasks. It's a columnar database, which means it stores data in a way that makes it much faster when you're running analytics queries—especially those that deal with large datasets. What's cool about DuckDB is that it's optimized for doing heavy analytical tasks directly in the browser, which means we can run all the analytics without needing a server and it is exactly what we need for our project[9].

In our research, DuckDB is key to running fast, in-browser data analysis without relying on a server to do the heavy lifting. Here's why it's a perfect fit for our needs:

- **Fast Data Queries:**
  DuckDB is built for heavy computations, so it can handle complicated queries in no time. This makes it perfect for our project, where we're working with large datasets and need fast results without relying on a server.

- **Works from Memory for Speed:**
  Since DuckDB is an in-memory database, it keeps data stored in the computer's RAM instead of loading it from a disk. This results in lightning-fast access to the data and much quicker processing.

- **Efficient Columnar Storage**
  DuckDB uses columnar storage, which makes it more efficient for analyzing large amounts of data. It's especially good for tasks like filtering, scanning, or aggregating data, which saves time and resources.

- **Works Well with WebAssembly:**
  DuckDB and WebAssembly (WASM) are a perfect match. WASM allows us to run DuckDB's queries directly in the browser at near-native speed. This means we don't have to rely on external servers to do the heavy lifting, making our extension much faster[10].

- **Lightweight and Scalable:**
  DuckDB is lightweight, so it's perfect for embedding in a browser extension. It can handle growing datasets without slowing down, keeping everything responsive even as the data size increases.

2.9.1 Web Filesystem for JSON and Parquet File Processing in DuckDB-Wasm

DuckDB-Wasm comes with a specialized filesystem designed for WebAssembly, making it easier to work with structured data formats like JSON and Parquet. Since DuckDB is built on a virtual filesystem, it abstracts high-level operations—such as reading JSON or Parquet files—away from the low-level filesystem interactions that vary across operating systems. This design allows DuckDB-Wasm to create custom filesystem solutions that are optimized for different WebAssembly environments, ensuring smooth data access and efficient processing.

When a user runs a SQL query on a JSON or Parquet file, the query is first sent to a web worker through a JavaScript API. From there, it passes through the WebAssembly module for execution. When the query reaches the json_scan or parquet_scan function, the system reads the file using a buffered filesystem, which processes data in smaller chunks. The web filesystem then fetches the necessary data from different sources, whether it's stored locally, on a remote server, or in memory as a preloaded buffer.

Treating JSON and Parquet files the same way, regardless of where they are

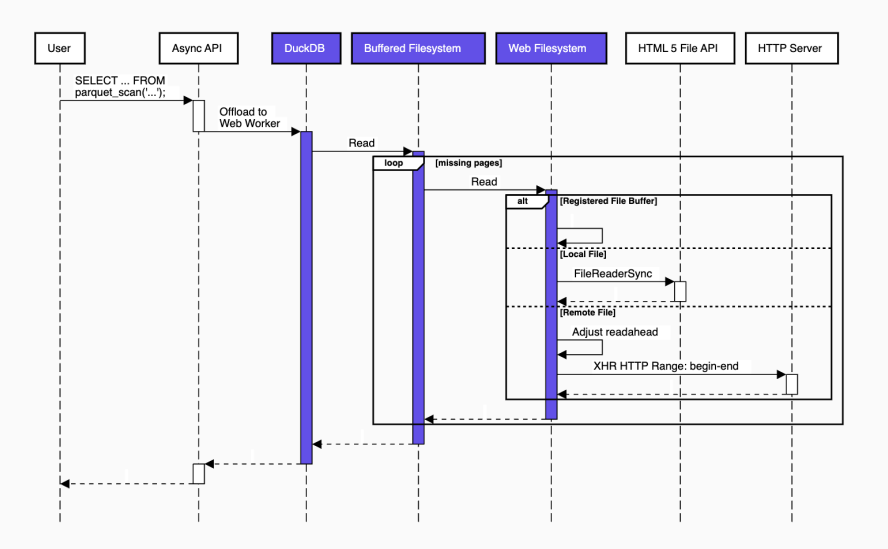

Figure 2.1: Web File System with DuckDB

stored, simplifies data retrieval and enables useful optimizations. JSON, in particular, is widely used for storing structured data, but its nested format can make queries slower. DuckDB-Wasm helps speed things up by applying techniques like schema inference and indexing to quickly locate and extract relevant data.

One major advantage of this approach is that DuckDB-Wasm doesn't need to load entire files into memory. Instead, it retrieves only the parts required for a given query. For example:

- A query like SELECT count(*) FROM json_scan('data.json') or SELECT count(*) FROM parquet_scan('data.parquet') can be executed using just the file's metadata, making it incredibly fast—even for very large files.
- Queries with LIMIT and OFFSET, such as SELECT * FROM json_scan('data.json') LIMIT 20 OFFSET 40, allow users to efficiently page through large datasets without unnecessary data loading.
- If a query includes filters, DuckDB-Wasm can use metadata to skip over irrelevant sections of the file, reducing read times and improving performance.

While JSON offers more flexibility than the columnar format of Parquet, DuckDB-Wasm applies similar performance-enhancing techniques to both formats, such as compression and batched data processing. By relying on SQL semantics rather than complex JavaScript logic, it makes querying structured data in the

browser more efficient and user-friendly.

That said, there are still some limitations. Since browser-based File APIs are somewhat restrictive, certain features—like saving DuckDB databases for long-term storage—are challenging to implement. While IndexedDB could be a workaround, it doesn't yet support the fast, synchronous read/write operations needed for seamless integration. Overcoming these limitations is still an ongoing area of research, and future improvements may help bridge these gaps, making web-based data processing even more powerful.

Chapter 3

Methodology

3.1 Introduction

In this chapter, we describe the practical approach we took to design and implement our WebAssembly-powered browser extension. Our goal was to build an extension that is both highly efficient and user-friendly by taking advantage of WebAssembly's speed for intensive computations and integrating it with a modern React.js front-end.

The architecture is split into two main parts:

- **Client-Side:** Here, we developed the user interface using React.js. This layer handles all user interactions and communicates with the WebAssembly module, which executes complex data operations directly within the browser. This combination ensures that the extension remains responsive and fast.

- **Server-Side:** On the backend, we use DuckDB for processing large datasets. DuckDB's in-memory analytics allow us to efficiently handle data operations that might be too heavy for the browser alone. This server-side component acts as a reliable data provider that can support the extension when dealing with very large or complex data queries.

By merging the near-native performance of WebAssembly, the robust in-memory querying capabilities of DuckDB, and the flexible, dynamic UI provided by React.js, this project demonstrates how modern web technologies can be combined to create a powerful browser extension. The result is an application that not only performs intensive data processing tasks with remarkable speed but also delivers a seamless and engaging user experience.

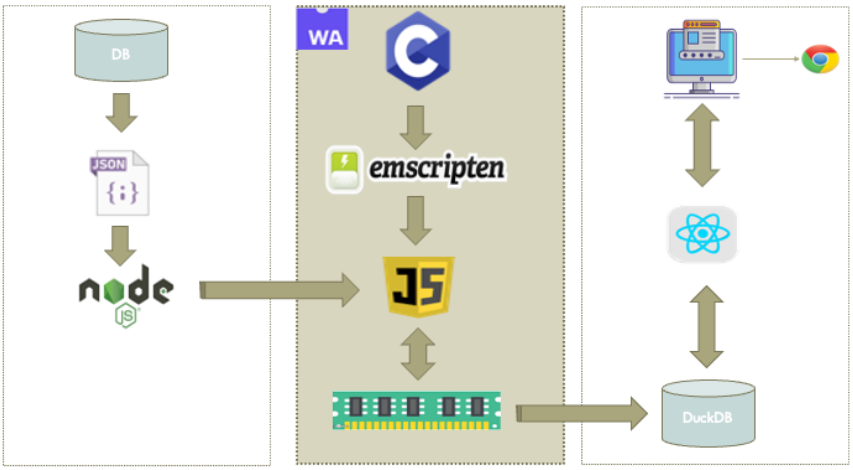

Figure 3.1: WASM Archirecture Diagram

3.2 Server-Side Architecture

3.2.1 MVC Pattern

For the server-side of the application, we've adopted the Model-View-Controller (MVC) architecture, a well-established approach that helps in keeping different parts of the system separated and organized. This not only makes the system easier to manage but also more scalable as it grows. The MVC pattern divides the application into three main components: the Model, the View, and the Controller, each with its own role: [11]

- **Model (Data and Business Logic):** The Model is responsible for managing and processing data. In this project, it deals with fetching data from PostgreSQL and performing tasks such as data formatting, pagination, and integration with DuckDB. DuckDB, an in-memory database, is used here to efficiently query and process large datasets. It works with the JSON files of students, marks, and courses, which were previously exported from PostgreSQL. The Model handles queries to paginate the data, ensuring smooth and efficient retrieval of large datasets. Once the data is retrieved and processed, it's passed to the View to be displayed.
- **View (User Interface):** The View is the part that users interact with directly. In this project, the View is the WebAssembly-powered browser extension. This extension manages the user interface, fetching data from the backend in a

paginated manner and displaying it to the user in an organized and visually appealing way. The View is responsible for ensuring a smooth user experience by rendering the data efficiently and interactively.

- **Controller (Request Handling and Data Flow):** The Controller serves as the bridge between the Model and the View. It takes incoming requests from the client, processes the required data through the Model (e.g., handling pagination, formatting, etc.), and sends the results to the View for display. The Controller is also responsible for managing errors, validating requests, and ensuring that the data is correctly structured before it's passed to the View.

3.2.2 Advantages of Using MVC in Server-Side Architecture

Using the MVC pattern offers several key benefits, especially for a project of this scale:

- **Separation of Concerns:** With the MVC pattern, the application is neatly divided into three components. This separation makes it easier to maintain and update the system. For example, if you need to update how the data is processed or change the design of the UI, these changes can be made independently without affecting the other parts of the system.

- **Scalability:** As the application evolves, it's easy to add new features without a major overhaul. If new data sources need to be integrated, they can be added in the Model without affecting the View or Controller. This makes scaling the application much more efficient.

- **Maintainability:** By keeping the components separate, the codebase becomes more organized and easier to understand. Each developer can focus on their specific area—whether it's the data, the UI, or the request handling—making the code more readable and reducing the risk of errors during development or debugging.

- **Testability:** Since each component is separate, it's easier to test them individually. For example, you can perform unit tests on the Model to check if the data manipulation works correctly, test the View to ensure the UI is rendering as expected, and test the Controller for request handling. This modularity ensures better robustness and reliability for the application.

3.2.3 Final Thoughts

Adopting the MVC pattern for the server-side architecture significantly contributes to the modularity, maintainability, and scalability of the system. By leveraging this architecture, the application can efficiently handle complex data processing, ensuring smooth interaction between the client-side (via the WebAssembly-powered extension) and the server-side components. This clean separation of concerns allows the system to grow and evolve without affecting the core functionality.

3.3 Client-Side Architecture

The client-side architecture of our WebAssembly-powered browser extension is designed to provide an efficient, fast, and responsive user experience. It is built using React.js for the user interface (UI), DuckDB for local data storage and querying, and WebAssembly (WASM) for performing computationally intensive tasks directly in the browser. This combination allows the extension to handle large datasets while maintaining high performance, low memory usage, and a smooth, interactive UI.

Key Responsibilities

The client side of the extension is responsible for several core tasks:

1. **User Interface Rendering**
    - The UI, built with React.js, offers a simple, interactive platform for users to enter queries, view results, and adjust settings.
    - React's component-based architecture allows us to efficiently manage the state and layout of the UI, providing flexibility and scalability.
2. **Data Storage and Querying Using DuckDB**
    - DuckDB, an in-browser SQL database, is used to store the data fetched from the backend. Once the data is retrieved, DuckDB handles local querying, which means we can filter and process large datasets right in the browser without repeatedly hitting the backend.
    - DuckDB ensures that the data is efficiently stored in memory and queried, allowing us to load only relevant data and avoid unnecessary network calls.
3. **WebAssembly Integration for Heavy Computation**

- WebAssembly (WASM) is used for handling complex calculations and data transformations. After data is processed in DuckDB, it is passed to WebAssembly to perform computational tasks that would otherwise be too resource-intensive for JavaScript.
- WebAssembly allows us to run high-performance operations directly in the browser, significantly speeding up data processing without overloading system resources.

4. **State Management and Smooth Data Flow**

   - React's built-in state management features, such as useState and useEffect, ensure that data is constantly in sync between the UI, DuckDB, and WebAssembly.
   - The state is dynamically updated, ensuring that the interface remains responsive and reflects any changes in the data in real-time.

3.3.1 WebAssembly Workflow on the Client Side

The client-side WebAssembly workflow follows a structured process to ensure efficient data handling and computation:

1. **Data Fetching and Storing in DuckDB**

   - The extension first fetches paginated data from the backend API. This includes data like students, marks, and courses.
   - This data is then stored in DuckDB, a local, in-browser database that allows fast access to data without needing to query the backend repeatedly.

2. **Local Data Processing with DuckDB**

   - DuckDB is used to perform initial queries on the data, including filtering, sorting, and aggregating. This ensures that only the relevant data is sent to WebAssembly for further processing, improving performance and reducing the workload of both the backend and the browser.

3. **WebAssembly Data Processing**

   - Once DuckDB processes the data, it is passed to WebAssembly for heavy computations such as statistical analysis, data transformation, and other resource-intensive tasks.
   - WebAssembly performs these operations at high speed, returning the processed results back to React for display.

4. **Displaying Results in the UI**

    - The React app dynamically updates the UI with the results returned from WebAssembly. Components like tables, charts, and forms are updated based on the latest data.[12]
    - The virtual DOM in React ensures that only the necessary components are re-rendered, maintaining fast and responsive interactions.

3.3.2 Enhancing User Experience

To ensure a seamless user experience, we have implemented several features to optimize performance and responsiveness:

1. **Efficient Data Querying with DuckDB**

    - By using DuckDB for local data storage and querying, we minimize the need for repeated backend API calls. This speeds up data retrieval and reduces the load on the backend, allowing the app to respond more quickly to user interactions.

2. **Asynchronous Processing with WebAssembly and React**

    - Communication between React and WebAssembly is asynchronous, meaning that long-running tasks do not block the user interface.
    - Promises and Web Workers are used to handle data processing without freezing the UI, ensuring that users can continue interacting with the extension while data is being processed in the background.

3. **Optimized Rendering with React's Virtual DOM**

    - React's virtual DOM minimizes unnecessary UI updates, ensuring that only the necessary components are re-rendered when the data changes.
    - This helps maintain smooth and fast interactions, even when large datasets are being processed or queried.

4. **User-Friendly Interface**

    - The UI is designed to be intuitive and easy to use, with dynamic components that automatically update based on the data returned from WebAssembly.
    - The interface includes tables, charts, and forms that are updated in real-time, providing users with an interactive and visually engaging experience.

3.3.3 Why This Architecture Works

The combination of **React.js**, **DuckDB**, and **WebAssembly** enables the extension to deliver high performance, scalability, and efficiency. Here's why this architecture is effective:

- **Fast Performance:** DuckDB handles data storage and querying locally, minimizing latency and reducing the need for network requests. WebAssembly performs heavy computations directly in the browser, allowing for faster data processing.
- **Memory Efficiency:** By using DuckDB to store and query data locally, we reduce the amount of memory required to hold large datasets. WebAssembly processes only the relevant data, ensuring minimal memory usage.
- **Reduced Backend Dependence:** DuckDB enables local data querying, which reduces the need to make frequent calls to the backend. This minimizes network overhead and speeds up the user experience.
- **Responsive UI:** React's state management and virtual DOM optimizations ensure that the UI remains smooth and responsive, even during intensive data processing tasks.

3.3.4 Final Thoughts

The client-side architecture of our WebAssembly-powered browser extension is designed to offer a fast, efficient, and responsive user experience. By combining React.js for the UI, DuckDB for local data storage and querying, and WebAssembly for high-performance computations, we are able to handle large datasets, reduce memory usage, and ensure that the extension operates smoothly in the browser.

This architecture not only delivers real-time data processing capabilities but also keeps the user interface lightweight and responsive. With asynchronous data fetching, optimized rendering, and local computation, the extension can handle complex tasks with ease while providing an enjoyable experience for the user.

Chapter 4

Implementation and Optimization

4.1 Introduction

In this section, we will explore the complexities of the implementation and optimization process for the WebAssembly-powered browser extension. The core objective behind this project was to develop a fast, efficient, and scalable extension capable of handling large datasets while ensuring that users experience minimal delays and smooth interactions with the interface. The combination of modern technologies such as React.js, DuckDB, and WebAssembly allowed us to achieve high performance and reduced memory overhead—key factors for building a responsive, robust application.

Creating a seamless and interactive user interface is paramount for browser extensions, especially when working with large and complex datasets. This requires a careful balance of data management, computational efficiency, and UI responsiveness. In the sections that follow, we break down the technical decisions made during the development, the integration of different technologies, and the optimization strategies employed to enhance performance, improve memory usage, and optimize data handling.

4.2 Server-Side Implementation

4.2.1 Server Architecture

Our server is designed with efficiency, scalability, and seamless data handling in mind. At its core, it utilizes Express.js, a lightweight and flexible Node.js framework,

to manage HTTP requests and define API routes. Express provides a robust foundation for handling client-server interactions while allowing for easy integration of middleware for security, data processing, and cross-origin communication.

To manage our database operations, we rely on Sequelize, a powerful Object-Relational Mapping (ORM) library for PostgreSQL. Sequelize simplifies complex SQL queries, ensures data consistency, and provides a secure way to interact with the database. This ORM enables us to perform database operations using JavaScript, reducing the need for raw SQL queries and making development more streamlined.

For high-speed data analysis, we incorporate DuckDB, a high-performance, in-memory analytical database. DuckDB is particularly well-suited for handling large datasets directly within the server, eliminating the need for external data warehouses. This allows us to perform complex queries efficiently, enhancing the speed and responsiveness of our application.

Additionally, we use JSON file storage to optimize data retrieval, especially for frequently accessed datasets. Storing structured data in JSON format allows for quick read/write operations, reducing the dependency on database queries for non-dynamic data.

To ensure smooth client-server communication, we implement CORS (Cross-Origin Resource Sharing), enabling secure access to our API from different domains. We also utilize JSON body parsing middleware to efficiently process incoming requests, ensuring that all JSON data sent from the client is correctly formatted and accessible within our application.

4.2.2 Key Components of the Server-Side Architecture:

- **Express.js** Handles routing, API endpoints, and middleware integration.
- **Sequelize** Manages database interactions and ensures efficient data transactions.
- **DuckDB** DuckDB optimizes analytical query processing by using in-memory execution for fast computations while also supporting disk-based storage for larger datasets. This hybrid approach ensures high performance when memory is available while maintaining scalability by seamlessly accessing data from disk when needed.[13]
- **JSON File Storage** Optimizes data retrieval by reducing unnecessary database queries.
- **CORS and JSON Body Parsing Middleware** Ensures secure and structured communication between the client and server.

By combining these technologies, our server achieves a balance between performance, flexibility, and security, making it well-equipped to handle real-time data processing and efficient client interactions.

4.2.3 API Implementation (Server-Side Design)

Key Optimizations

- **Increased Request Timeout:**
  Since data processing can be intensive, the request timeout is increased to 60 minutes.

```
app.use((req, res, next) => {
    req.setTimeout(60 * 60 * 1000); // 60 minutes
    res.setTimeout(60 * 60 * 1000); // 60 minutes
    next();});
```

- **Memory Management:**
  To ensure our server can efficiently handle large datasets, we configure the memory limit in the package.json file using the –max-old-space-size=8192 flag. [14]This setting increases Node.js' memory allocation to 8GB, allowing for better performance when processing large amounts of data. By modifying this configuration, we prevent potential memory-related crashes and ensure smooth execution of computationally intensive tasks.

```
"scripts":{
    "start":  "nodemon -ignore 'data/students.json' -ignore
'data/marks.json' -ignore 'data/courses.json' -exec 'node
-max-old-space-size=8192 server.js'",
    });
```

- **CORS Configuration:**
  To allow smooth interaction between the frontend and backend, we set up CORS (Cross-Origin Resource Sharing) to permit requests from http://localhost:3000|

This prevents the browser from blocking API requests due to security policies, ensuring seamless data exchange.

Here's how we set it up:

```
const corsOption = {
    origin:  'http://localhost:3000',
    optionsSuccessStatus:  200,
    credentials:  true
    };
    app.use(cors(corsOption));
```

- **Managing Large JSON Requests:**
  To ensure our server can handle large API requests smoothly, we've increased the JSON size limit to 100MB. This allows us to process bulk data uploads, large JSON responses, and complex API requests without hitting default size restrictions.

  Why Adjust the Limit?
  By default, Express.js sets a 1MB limit for incoming JSON data. For applications dealing with large datasets, this can be too restrictive, causing requests to fail. While we can define larger limits based on our needs, setting them too high can impact performance and expose the server to Denial-of-Service (DoS) risks if not managed properly.[15]

```
app.use(express.json({ limit:  '100mb' }));
```

4.2.4 Database Configuration for Large-Scale Data Handling

To handle large amounts of data efficiently, we optimized the backend to support high-volume transactions without slowdowns. PostgreSQL was chosen for its ability to manage complex queries and massive datasets, while Sequelize ORM helped streamline database interactions. To keep things running smoothly, we fine-tuned connection pooling and timeout settings, ensuring stable performance even under heavy workloads.

Key Improvements

- **Smart Connection Pooling:**Managed active database connections efficiently to prevent overload while keeping performance high.
- **Extended Timeout Settings:**Increased time limits for idle connections and data retrieval to avoid disruptions during bulk inserts.
- **Logging Optimization:**Disabled logging during large data inserts to minimize unnecessary processing and improve efficiency.

These adjustments allowed the system to **handle millions of records seamlessly** , ensuring that queries and inserts remained smooth even under stress.

4.2.5 Faster Data Retrieval with JSON Caching

To improve response times and reduce repeated queries to PostgreSQL, we implemented JSON caching as a hybrid approach for data retrieval. Frequently accessed data is stored in JSON files, allowing the system to fetch preprocessed results instead of repeatedly querying the database.

Benefits of JSON Caching:

- **Faster Data Access:**Queries are run once and stored in JSON, cutting down on redundant database calls.
- **Lower Server Load:**With frequently accessed data cached in JSON, PostgreSQL is freed up for more critical operations, improving scalability.
- **Optimized Pagination:**Large datasets are paginated using DuckDB, enabling efficient retrieval of segmented data without reloading the entire dataset.

4.2.6 Simulating Large-Scale Data for Performance Testing

To evaluate how the system performs under real-world conditions, we generated and inserted 2,000,000 records into the database using **@faker-js/faker**. This fake data generator dynamically created student, course, and marks, simulating realistic database activity.

- **Realistic Data Simulation:**By using localized Faker.js settings, we generated student names, course titles, and marks that match real-world patterns based on specific regions.

- **Bulk Data Insertion:**The system efficiently processed and inserted millions of records without slowdowns or crashes.
- **Scalability Check:**This approach confirmed that our database setup can handle high data volumes while maintaining fast performance.

These enhancements ensure that our system is scalable, realistic, and adaptable for various use cases. More details on performance benchmarks can be found in the Performance Analysis section.

4.3 Client-Side Implementation

4.3.1 Introduction

The client-side of our system is designed with a strong focus on efficiency, scalability, and responsiveness. We leverage **React.js** to create a dynamic and interactive UI, while **WebAssembly (WASM)** is used to optimize in-memory data operations, significantly reducing the need for frequent backend requests. By using WASM, we ensure that large-scale datasets—containing millions of records—can be processed and retrieved seamlessly without overloading the browser or causing performance bottlenecks.

This approach not only enhances the speed and responsiveness of the application but also minimizes server dependency, making the system more scalable and cost-effective.

4.3.2 Technologies Used

- **React.js** A component-based front-end library for building scalable UIs.
- **Fetch API** Handles asynchronous communication between the frontend and backend.
- **WebAssembly (WASM)** Speeds up large-scale in-memory computations.
- **C (WASM Module)** generated modules by emcc Used to manage memory-efficient data storage and retrieval.
- **DuckDB** A high-performance in-memory SQL database optimized for analytical queries.
- **Material-UI** Provides a clean and modern UI design for enhanced user experience.

Each of these tools plays a crucial role in ensuring the smooth performance of the client-side system, particularly when dealing with vast amounts of data.

4.3.3 Efficient Data Fetching Strategy

- **Pagination** Fetching data in smaller chunks rather than loading everything at once.[16]
- **In-Memory Storage with WASM** Storing fetched data in memory instead of making repeated backend calls.
- **Asynchronous Fetching** Ensuring smooth data retrieval without freezing the UI.
- **DuckDB Query Execution** Running analytical SQL queries directly in the browser for fast data analysis.
- **Batch Processing** Grouping API requests to reduce network overhead.
- **Pre-fetching Strategies** Anticipating the user's next actions and loading data in advance to enhance the browsing experience.
- **Lazy Loading** Loading only the necessary data initially and retrieving additional records when required.

4.3.4 Fetching Data from the API

To fetch large datasets efficiently, we implemented a pagination-based fetching mechanism:

```
const response = await fetch(BACKENDURL +
‘/allPagedJsonData?page=page&limit =limit‘);
```

How This Approach Improves Performance?

By implementing this intelligent data-fetching approach, we ensure that our system can handle millions of records smoothly while keeping the UI responsive and efficient.

4.3.5 Data Storage and Processing in WASM

Once the data is fetched, it is stored in WASM memory, allowing for high-speed access and efficient memory utilization. The C-based WebAssembly module manages structured storage, ensuring that data operations are performed without JavaScript-induced performance bottlenecks.

Memory Management in C : Optimizing Performance for Large Datasets

Managing memory efficiently is crucial when handling large datasets, especially when working with millions of records. Our system employs a structured memory allocation approach in C, ensuring that data is stored, retrieved, and processed quickly without unnecessary overhead. This setup is designed to work seamlessly within WebAssembly (WASM) to improve performance on the client side.[17]

Memory Configuration During the Trial Phase

Before integrating real-world data, we conducted a trial phase where we used fake and simulated datasets to fine-tune our memory management approach. This phase was essential for:

- **Testing different memory allocation sizes** to determine the best fit for performance.
- **Simulating real-world database interactions** to ensure smooth data retrieval.
- **Identifying memory bottlenecks** and addressing potential leaks.
- **Benchmarking query speeds** to optimize data access.

By configuring our memory handling based on trial data, we were able to predict system behavior under real load and make necessary adjustments before full deployment.

Efficient Memory Allocation and Initialization

To ensure smooth operations, memory is dynamically allocated for different entities—students, courses, and marks—only when needed. Each dataset has its own initialization function that:

- **Frees previously allocated memory** (if any) to prevent leaks.[17]
- **Allocates new memory blocks dynamically based on record count.**
- **Optimizes space usage** to prevent excessive memory consumption.

For example, the student memory initialization function follows this approach:

```
void std_init(int count){
if (students!=NULL){
free(students);
}
 num_students = count;
 students = (Student*)malloc(count * sizeof(Student));
}
```

This structure is **replicated for courses and marks**, ensuring that each dataset is handled independently while preventing memory conflicts.

Inserting Data into Memory

Once memory is allocated, data is directly inserted into pre-allocated memory blocks, ensuring **fast read/write operations** without the need for additional storage operations.

For example, when adding a new student record, we:

- **Ensure the index is within the allocated range** to prevent memory corruption.
- **Copy the student's name which is an string safely** using strncpy() to avoid buffer overflows.
- **Store the student's ID and age** which are integers in the structured memory block.

This approach ensures that student data is efficiently stored and easily retrievable. Similar functions are implemented for courses and marks, maintaining a structured and predictable data layout in memory.

Retrieving Data from Memory for Queries

A major advantage of our C-based memory management is the ability to retrieve data instantly without needing additional API calls. Since data is stored in contiguous memory blocks, queries are executed in real-time with minimal delay.

For instance, fetching a student record is as simple as:

```
Student* get_students(int index){
if (index < 0 || index >= num_students) return NULL;
return &students[index];
}
```

This design allows us to:

- **Retrieve records instantly** for UI rendering and analytics.
- **Optimize in-memory searching** for real-time performance.

The same logic applies to courses and marks, allowing for seamless access to all required data.

Preventing Memory Leaks with Proper Deallocation

Since we use dynamic memory allocation, proper memory deallocation is crucial to avoid leaks and performance degradation over time. To handle this, we implement cleanup functions that free allocated memory when it is no longer needed.
For example, to free student records, we use:

```
Student* students=NULL;
void free_students(){
free(students);
students = NULL;
}
```

This ensures that:

- **Allocated memory is properly released**,preventing memory buildup.
- **Dangling pointers are avoided** by setting them to NULL.

Similar deallocation functions exist for courses and marks, ensuring efficient memory management across the entire system.

Final Thoughts: High-Performance Memory Management

By structuring our memory allocation in C, we've built a fast, scalable, and efficient system capable of handling millions of records with minimal performance overhead. Our approach ensures that:

- **Data is stored and retrieved instantly** using in-memory storage.
- **Memory leaks are prevented** through structured allocation and deallocation.
- **Database load is reduced** improving overall system efficiency.
- **Query execution is near-instant** ensuring smooth UI performance.

The trial phase played a crucial role in fine-tuning our memory management, allowing us to predict real-world behavior and optimize the system accordingly. With this approach, we can efficiently**handle large datasets, reduce latency, and maintain a smooth user experience**—all while keeping memory usage under control.

4.4 Using Emscripten (emcc) to Bridge C and JavaScript in Our Project

4.4.1 Introduction to Emscripten (emcc)

Emscripten (emcc) is a powerful compiler that allows us to convert C code into WebAssembly (WASM) and JavaScript, enabling seamless integration of high-performance native code within web applications[18]. In our project, we use emcc to compile our C-based memory management and data handling logic into JavaScript-accessible functions, ensuring fast in-memory operations while keeping the flexibility of a React-based frontend.

By integrating emcc, we achieve:

- **Seamless execution of C functions within JavaScript**, reducing reliance on backend processing.
- **Optimized memory management through WebAssembly**, minimizing browser memory overhead.
- **Faster data retrieval and storage** by keeping frequently accessed data in memory.
- **Improved scalability**, enabling efficient handling of millions of records.

4.4.2 Breaking Down the emcc Compilation Script

To generate a WebAssembly module from our C code (allDataInMemory.c), we use the following emcc command[19]:

```
emcc allDataInMemory.c -o mainstudents4.js \
    -s MODULARIZE=1 \
    -s SINGLE_FILE=1 \
    -s EXPORTED_FUNCTIONS="['_std_init', '_courses_init', '_marks_init',
        '_insert_student', '_insert_courses', '_insert_marks',
        '_get_students', '_get_courses', '_get_marks',
        '_free_students', '_free_courses', '_free_marks']" \
    -s EXPORTED_RUNTIME_METHODS="['cwrap', 'UTF8ToString', 'HEAP32',
        'lengthBytesUTF8', 'stringToUTF8', '_malloc', '_free']" \
    -s INITIAL_MEMORY=2147483648 \
    -s ALLOW_MEMORY_GROWTH=1 \
    -s MAXIMUM_MEMORY=4GB
```

Each flag plays a critical role in **exposing our C functions to JavaScript while optimizing performance**. Let's break them down:

1. **Creating a Modular JavaScript Wrapper using** -s MODULARIZE=1
   This ensures that our compiled WebAssembly module is wrapped inside a JavaScript function instead of being a global script. This approach provides:

   - **Better maintainability** We can import the module as needed rather than polluting the global namespace.
   - **Dynamic loading** The module can be loaded asynchronously in our React application.
   - **Multiple instances** If needed, we can create different instances of the module for various tasks.

     With MODULARIZE=1, we import the module in JavaScript like this: import createMainModule from "./mainstudents";

2. **Outputting a Single File using** -s SINGLE_FILE=1 This forces the compiler to **bundle everything, including the WebAssembly code, into a single** .js file.[19] The advantages of this approach include:

   - **Easier deployment** No need to manage separate .wasm files.
   - **Reduced HTTP requests** The browser loads everything in one go.

- **Simpler project structure** All compiled code is self-contained.

3. **Making C Functions Usable in JavaScript**

```
-s EXPORTED_FUNCTIONS=
"['_std_init', '_courses_init', '_marks_init',
'_insert_student', '_insert_courses', '_insert_marks',
'_get_students', '_get_courses', '_get_marks',
'_free_students', '_free_courses', '_free_marks']"
```

This flag ensures that specific functions from our C code are exposed and callable in JavaScript.

For example, after compilation, we can initialize student data directly from JavaScript:

```
module._std_init(numberOfStudents);
```

By exporting these functions, we ensure that our **entire in-memory data management system—initialization, insertion, retrieval, and cleanup—is accessible from JavaScript.**

4. **Exporting Runtime Methods for Memory Management**

```
-s EXPORTED_RUNTIME_METHODS=
"['cwrap', 'UTF8ToString', 'HEAP32',
'lengthBytesUTF8', 'stringToUTF8', '_malloc', '_free']"
```

These runtime methods allow us to handle memory allocation and string conversion between JavaScript and C efficiently:

- **cwrap** Enables calling C functions from JavaScript without manual bindings.
- **UTF8ToString and stringToUTF8** Convert text data between JavaScript and C.
- **HEAP32** Provides direct access to WebAssembly's memory.

- **__malloc and __free** Manage dynamic memory allocation, preventing leaks.

For example, when inserting student names, we need to **convert JavaScript strings to C-style UTF-8 strings and store them in WebAssembly memory:**

```
const snamePtr = module._malloc(module.lengthBytesUTF8
(student.sname) + 1);
module.stringToUTF8(student.sname, snamePtr,
module.lengthBytesUTF8(student.sname) + 1);
module._insert_student(index,student.id,snamePtr, student.age);
module._free(snamePtr);
```

This method ensures **efficient memory use and prevents unnecessary reallocation.**

5. **Configuring WebAssembly Memory Allocation**[19]

```
-s INITIAL_MEMORY=2147483648
```

This pre-allocates **2GB of memory** for WebAssembly. Since our project deals with large datasets, having a sizable initial memory allocation:

- **Prevents memory fragmentation.**
- **Improves performance by reducing the need for memory expansion.**
- **Ensures stability when handling large-scale data.**

```
-s ALLOW_MEMORY_GROWTH=1
```

This allows WebAssembly memory to **dynamically expand** when needed, rather than being limited to a fixed size. It helps in:

- **Efficient memory usage**, growing only when necessary.

- **Handling varying dataset sizes** without pre-allocating excessive memory.

This sets a **hard cap on memory usage at 4GB**, ensuring that our

```
-s MAXIMUM_MEMORY = 4GB
```

application does not consume excessive system resources.

Why emcc Matters in Our Project?

By compiling our C code with emcc and integrating WebAssembly, we achieve:

- **Blazing-fast in-memory operations** All data queries run directly in memory, reducing API calls.
- **Seamless interaction between C and JavaScript** Complex data processing happens efficiently without slowing down the UI.
- **Optimized memory handling** With manual allocation and cleanup, we prevent memory leaks.
- **Smooth user experience** Even with millions of records, the frontend remains responsive.
- **Scalability** Our architecture supports increasing data volumes with minimal performance impact.

With this setup, our project can efficiently store, retrieve, and process large-scale data in WebAssembly memory, providing high performance without overloading the browser or backend.

4.4.3 Efficient Data Handling with WebAssembly in JavaScript

One of the key aspects of our project is how JavaScript interacts with WebAssembly (WASM) to efficiently store and retrieve structured data. Instead of relying on standard JavaScript objects, which can be memory-intensive and slow for large datasets, we use WASM to manage data in a compact, structured format. The functions that make this possible—such as **__std__init** ,**__insert__student** , and **__get__students** —were generated using **Emscripten (emcc)** and allow JavaScript to directly manipulate WASM memory.

Setting Up the WebAssembly Module

Before anything else, we need to initialize our WebAssembly module, which acts as a bridge between JavaScript and the compiled C functions. This is done using:

```
await createMainModule();
```

This function loads the precompiled WASM module, giving us access to the memory-efficient functions we defined in C. While createMainModule() is the function name in our implementation, it could be named anything—since it's just a wrapper around our compiled module.

Allocating Memory

Once the WASM module is ready, we need to allocate memory for the student records:

```
module._std_init(students.length);
```

This function, _std_init(), ensures that WASM has enough memory space to store all student records efficiently. Without this step, trying to insert students into memory would fail.

Storing Data in WASM

Since WASM does not support JavaScript strings directly, we need to manually allocate memory for each student's name before inserting the data. The process works as follows:

1. Allocate memory for the student's name using **_malloc()**.
2. Convert the name to UTF-8 and copy it into WASM memory using **_stringToUTF8()**.
3. Insert the structured student data using **_insert_student()**.
4. Free the allocated memory afterward using **_free()** to prevent memory leaks.

Here's how this is done in code:

```
students.map((student, index) => {
const snamePtr =
module._malloc(module.lengthBytesUTF8(student.sname) + 1);
 module.stringToUTF8(student.sname, snamePtr,
module.lengthBytesUTF8(student.sname) + 1);
module._insert_student(index, student.id, snamePtr,
student.age);
 module._free(snamePtr);
}
```

This method ensures that our student data is stored efficiently inside WASM without unnecessary memory overhead.

Retrieving Data from WASM

Once the data is inside WASM memory, we need a way to retrieve it. This is done using _get_students(index), which returns a pointer to the stored data. Since WASM memory is a continuous block, we extract individual values using pointer arithmetic:

```
    const stdproceeddata = students.map((_, index) => {
    const stdpointer = module._get_students(index);
    return {
        id:  module.HEAP32[stdpointer / 4],
        sname:  module.UTF8ToString(stdpointer + 4),
        age:  module.HEAP32[(stdpointer + 56) / 4]
    }
    });
```

- **module.HEAP32[stdpointer / 4] retrieves the student's ID.**
- **module.UTF8ToString(stdpointer + 4) extracts the student's name from WASM memory.**

- **module.HEAP32[(stdpointer + 56) / 4] retrieves the age.**

This approach allows JavaScript to fetch structured data directly from WASM's memory, bypassing the inefficiencies of traditional JavaScript objects.

By using WASM, we've created a highly efficient way to store and retrieve structured data within the browser. The functions handling this process—such as _std_init, _insert_student, and _get_students—were all generated using Emscripten (emcc), allowing JavaScript to interact seamlessly with our compiled C code. This setup significantly improves performance, especially when dealing with large datasets, and ensures optimal memory usage by avoiding JavaScript's usual memory overhead

4.5 Leveraging DuckDB for Efficient In-Browser Data Processing

Handling large datasets efficiently within the browser is a significant challenge, especially when dealing with millions of records. Traditional databases require constant communication with a backend server, leading to latency and performance bottlenecks. To address this, DuckDB, an in-memory analytical database, is used to process large datasets directly in the browser.

By integrating DuckDB with WebAssembly (WASM) and utilizing pagination techniques, we ensure high performance, scalability, and memory efficiency, preventing browser crashes due to excessive memory usage. This section explores how DuckDB enables efficient client-side data processing, eliminating the need for a backend database.

Integrating DuckDB in the Browser

To run DuckDB within a browser, we utilize WebAssembly (WASM) to create a high-performance execution environment. WebAssembly allows DuckDB to operate efficiently within the browser's memory space, providing near-native performance.

Setting Up WebAssembly Memory

To prevent memory fragmentation and ensure smooth execution, WebAssembly memory is explicitly allocated:[6]

```
    const wasmMemory = new WebAssembly.Memory({
     initial:  512, // 32 MB
     maximum:  32768, // 2 GB
     shared:  true
    });
```

This preallocates memory, preventing unexpected crashes when handling large datasets.

Once memory is allocated, DuckDB is initialized as a Web Worker, allowing database queries to run asynchronously without freezing the UI:

```
    const bundle = await duckdb.selectBundle(
duckdb.getJsDelivrBundles());
    const worker = await duckdb.createWorker(bundle.mainWorker);
    const db = new duckdb.AsyncDuckDB(new duckdb.ConsoleLogger(),
worker,  wasmMemory )
    await db.instantiate(bundle.mainModule,
bundle.pthreadWorker);
```

Using a dedicated worker ensures that all database operations run in the background, maintaining a responsive user experience.

Handling Large Datasets Efficiently

To prevent **out-of-memory errors**, we adopt a pagination technique that processes data in chunks instead of loading everything at once.

4.5.1 Creating Tables and Loading Data

The dataset consists of three main tables:

- **Students:** Stores student details (ID, name, age).
- **Courses:** Stores course information (ID, name, credits).

- **Marks:**Stores student grades with foreign key references. With foreign key constraints, the integrity of the relational dataset is maintained (Many to Many).

4.5.2 Pagination Technique to Prevent Out-of-Memory Errors

Instead of loading millions of records at once (which could crash the browser), we use a pagination technique to process data in smaller chunks:

```
while (true) {
    let offset = stdpage * limit;
    await c.query(
    'INSERT INTO students (id, sname, age)
    SELECT id, sname, age FROM read_json_auto('students_*')
    LIMIT limit OFFSET offset'
    );
   const result = await c.query(
    'SELECT * from students LIMIT ${limit} OFFSET  ${offset}');
   if (result.toArray().length === 0) break;
    stdpage++;
}
```

Why Pagination?

- Prevents Memory Exhaustion: Instead of inserting all records at once, data is processed in batches, ensuring that memory usage remains stable.
- Improves Performance: Queries execute faster since smaller datasets are processed at a time.
- Ensures Smooth User Experience: The browser remains responsive even when handling millions of records.

The same technique is applied when inserting marks and courses, ensuring efficient memory management across all tables.

Conclusion

By integrating DuckDB with WebAssembly and pagination techniques, we enable high-performance, in-browser data processing without requiring a backend database.

Key takeaways:

- **High Performance:** DuckDB's in-memory, columnar processing ensures fast query execution even for large datasets.
- **Scalability:** Using pagination, the system can handle millions of records without running out of memory.
- **No Backend Required:** All operations run locally in the browser, reducing network overhead and improving privacy.
- **Seamless Integration:** DuckDB works directly with WebAssembly memory, optimizing performance for large-scale data analysis.

This approach is ideal for client-side analytics, interactive dashboards, and offline data processing, making web applications more powerful and efficient.

4.6 User Interface

In modern web applications, having a well-designed user interface (UI) is crucial for providing a smooth and efficient experience. This React-based UI is built to handle large datasets efficiently, combining Material UI for styling, WebAssembly (WASM) for performance optimization, and DuckDB for fast and lightweight data processing.

Users can input and execute SQL queries on large datasets, retrieving results in real-time with minimal delay. The application also provides query performance tracking, displaying execution speed to help users understand the efficiency of their queries. Error handling and notifications ensure that users receive clear messages in case of success, failure, or warnings, making interactions more intuitive[9].

In addition to executing queries, the UI also allows users to upload JSON files for analysis. Through the navigation bar, users can generate and insert fake data into PostgreSQL for testing purposes and easily export database records as a JSON file with a single click. Since this project is centered around researching applicability and performance, we included fake data insertion to facilitate future testing and performance evaluations as the study progresses.

Query results are displayed in a searchable table, making it easy to filter, search, and navigate through the data. This enhances usability, especially when working

with large datasets. To prevent memory overflow issues, the application implements pagination, ensuring that only a manageable portion of data is processed at a time.

By integrating these features, the application creates a highly efficient and user-friendly environment for data querying, processing, and visualization.

Chapter 5

Performance Analysis

To evaluate the system's efficiency in handling large-scale data operations, we ran thorough tests on the **PostgreSQL database** and **Json file** with **30,000,000 records**. The objective was to determine its ability to process high data volumes while maintaining speed and stability.

5.0.1 Key Performance Metrics Evaluated

- **Bulk Insert Efficiency** Ensuring the system can handle massive data inserts without failures.
- **Query Execution Speed** Measuring how quickly the database retrieves and processes large datasets.
- **Server Load and Scalability** Evaluating how concurrent requests impact system performance.
- **Browser Memory Constrains** Limitations in data upload and analysis according to the browsers memory capacity constrain.

5.1 Bulk Data Insertion Performance

To simulate real-world conditions, we generated synthetic student, course, and marks data using Faker.js and inserted it into PostgreSQL. We optimized the bulk insertion process using:

- **Batch Inserts:** Instead of inserting records one by one, we grouped them into batches, reducing transaction overhead.
- **Optimized Connection Pooling:** Adjusting Sequelize's connection pool settings to keep database connections stable under heavy loads.

Results:

- **Successfully inserted 2,000,000 records in under 45 minutes** without errors or connection failures.
- **Memory consumption remained stable** due to optimized pooling and transaction handling.

5.2 Query Performance and Optimization

Working with large datasets requires careful query optimization. In this approach, we used a combination of JSON caching and pagination techniques to effectively manage datasets with around 30,000,000 records. These strategies helped us avoid common issues like memory overload and errors when dealing with such massive amounts of data.

5.2.1 Key Optimizations

Caching Data with JSON Files:

- Instead of constantly querying PostgreSQL, we preprocessed the data and saved it in JSON files.
- This helped lighten the load on the PostgreSQL database, as we could quickly serve frequently accessed data directly from the JSON cache, reducing the strain on the system.

Using Pagination with JSON Files:

- For large datasets, we employed JSON data retrieval along with a pagination method.
- Pagination allowed us to break the query into smaller, more manageable chunks, which helped avoid issues like **"Out of Memory"** or **"Invalid string length"** errors that can occur when trying to pull large datasets all at once.
- By fetching smaller batches of data at a time, we could successfully manage datasets containing millions of records without running into system limitations.

Integrating DuckDB for Data Processing:

- DuckDB was used to process the data stored in JSON files, enabling efficient handling of large datasets by performing quick in-memory operations like filtering and aggregation.
- This integration helped us work with large amounts of data more easily, without the risk of overwhelming the system.

5.2.2 Results:

- **JSON-based retrieval** Caching the data in JSON files allowed us to retrieve large datasets without overloading the database.
- **Pagination Technique:** The pagination strategy, which broke the data into smaller chunks, resolved issues like memory overload, "Out of Memory", and "Invalid string length" errors, ensuring smooth and successful data retrieval even with massive datasets.

5.3 Data Upload Performance and Benchmarking

In this section, we look at how we fetch data from the backend and upload it into DuckDB tables, which allows us to easily query large datasets from the frontend. We tested two methods for inserting data: adding records one by one and using a more efficient approach where data is uploaded in larger chunks.

5.3.1 Benchmark Results

We conducted several tests to compare the performance with diffrent data sizes. The results are shown in the table below:

Method	**Records Inserted**	**Time Taken (ms)**
Inserting data one by one	1,000,000	926,373
Inserting data in chunks (1M)	1,000,000	18,015
Inserting data in chunks (6M)	30,000,000	724,837

Table 5.1: Benchmark Results for Data Upload Methods

1. **Inserting Data One by One:** When we tried inserting data one record at a time, it took **926,373 ms** to upload **1,000,000 records**. This method was really slow, especially with such a large number of records.
2. **Inserting Data in Chunks (1M):** To improve this, we switched to inserting data in larger chunks. By uploading all **1,000,000 records** at once, the time dropped dramatically to **18,015 ms**, which was a huge improvement.
3. **Inserting Data in Chunks (6M):** For even larger datasets, we increased the chunk size to handle **6,000,000 records** in each batch. With this approach, we were able to upload **30,000,000 records** in just **724,837 ms**, showing even better performance.

These results clearly demonstrate that using chunking for data insertion significantly speeds up the process and makes it easier to handle large datasets without running into performance issues.

5.4 Data Querying Performance: WASM vs. Normal API Fetch

We compared the performance of querying data using WASM and the traditional API fetch method. For a dataset of 30,000,000 records, the normal API fetch took 283,339 ms. In contrast, querying the same data using WASM was much faster, taking only 11,492 ms— a **95.94% improvement in speed** .

Method	**Data Records**	**Time Taken (ms)**
API Fetch Report	28,000,000 =1.5 GB	283,339
API Fetch Querying	28,000,000 =1.5 GB	200,112
WASM Reprot	28,000,000 =1.5 GB	8,112
WASM Query	28,000,000 =1.5 GB	2,957

Table 5.2: Benchmark Results for Data Querying with WASM vs Normal API Fetch

5.5 Limitations in Data Upload and Analysis

While our WebAssembly (WASM)-powered approach significantly improves data querying performance, it is constrained by the memory limitations of web browsers. Unlike traditional server-based processing, where memory allocation can be more flexible, browser environments impose strict memory limits that affect the volume of data we can handle.

In our case, using Google Chrome (memory limits may vary between browsers and browser versions), we observed a maximum WebAssembly memory allocation of approximately **2GB**. This directly impacts the size of JSON files that can be uploaded and queried. Based on our testing, we found the following constraints:

5.5.1 Uploading Large JSON Files

- We were able to successfully upload and process JSON files up to **1.5GB** before reaching memory constraints.
- Beyond this limit, we encountered a memory allocation failure:

```
malloc of size 536870912 failed
```

This suggests that insufficient contiguous memory was available for further allocation.

5.5.2 Querying Large Joined Tables

Queries involving large joined tables, such as retrieving all students, their courses, and marks with the following SQL query:

```
SELECT s.id, s.sname, c.cname, m.marks
FROM marks m
JOIN students s ON m.sid = s.id
JOIN courses c ON m.cid = c.cid;
```

required **1GB of browser memory** when working with tables totaling **512MB each (1GB in total)**. Since Chrome's WASM memory cap is approximately **2GB**, this allowed us to handle up to **1GB of data** while reserving the remaining **1GB for retrieval and processing**.

5.5.3 Querying Aggregated Data

Queries with aggregations, such as counting students with a specific mark:

```
SELECT c.cname, COUNT(DISTINCT s.id) AS Number_of_Students
FROM marks m
JOIN students s ON m.sid = s.id
JOIN courses c ON m.cid = c.cid
WHERE m.marks = 30
GROUP BY c.cname, c.cid
ORDER BY c.cname;
```

required less memory for result retrieval, allowing us to handle up to **1.5GB of data**. This is because the final query result was smaller compared to large joined tables, which required storing a higher volume of intermediate data in memory.

5.5.4 Key Takeaways

- **WebAssembly memory constraints (2GB in Chrome) limit how much data can be processed in-memory.**
- **1GB datasets (when querying large joined tables) are feasible**, as an additional **1GB is needed for memory allocation.**
- **1.5GB datasets are possible for queries with smaller output sizes**, as they require less memory for retrieval.
- **Exceeding these limits results in memory allocation failures** (e.g., `malloc of size 536870912 failed`).
- **Optimizations such as streaming data or using binary formats (e.g., Parquet, Arrow) could mitigate these limitations.**

This highlights the necessity of memory-efficient query execution and potential future optimizations, such as **streaming data instead of loading it all at once** or exploring **alternative browsers with higher WebAssembly memory limits**.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis examined how integrating WebAssembly (WASM) with JavaScript can enhance data retrieval and processing in web applications. The research demonstrated that WASM significantly boosts performance, improves memory management, and strengthens security when handling large datasets like student records, course information, and academic performance. By leveraging WASM, our implementation achieved faster execution times and optimized resource utilization compared to traditional JavaScript-based methods.

6.1.1 Key Findings

Performance Optimization

- WASM accelerates execution speed by operating closer to native code performance, minimizing JavaScript-related slowdowns in data-intensive tasks.
- Using typed arrays and direct memory manipulation (HEAP32) reduces overhead and speeds up data retrieval.

Efficient Memory Management

- The application optimizes memory usage through explicit allocation (malloc) and deallocation (free), ensuring efficient resource management.
- String handling within the WASM module is optimized to prevent memory leaks and performance degradation.

Security Enhancements

- WASM's sandboxed execution model isolates code, preventing unauthorized memory access and mitigating common web security threats.
- While WASM improves security, careful management of pointers and memory access is necessary to avoid vulnerabilities like buffer overflows.

Optimized Query Execution

- Integrating DuckDB with WebAssembly enables high-speed SQL-like queries directly within the browser.
- The system efficiently processes and retrieves filtered datasets, reducing reliance on backend computations.

Enhanced User Experience

- Combining WASM with React and Material-UI results in a smooth, responsive interface for executing and visualizing queries.
- Users can seamlessly interact with large datasets without compromising performance, enhancing overall usability.

6.1.2 Future Work

While this implementation demonstrates the advantages of WASM in data-driven applications, a key improvement for the future would be optimizing memory management to more efficiently handle dynamic and continuously changing data.

6.1.3 Final Thoughts

This thesis confirms that WebAssembly is a powerful tool for web-based data processing. By integrating WASM with modern web technologies, we achieve significant performance gains, enhanced security, and an improved user experience. As WebAssembly grows, it's set to shape the future of fast web apps.

Appendix

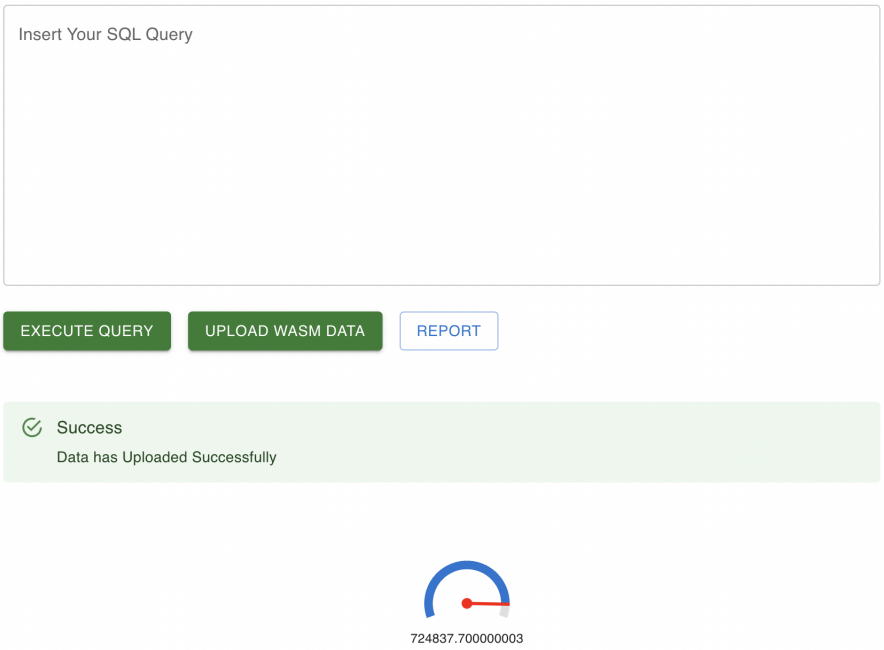

Figure 1: Upload performance

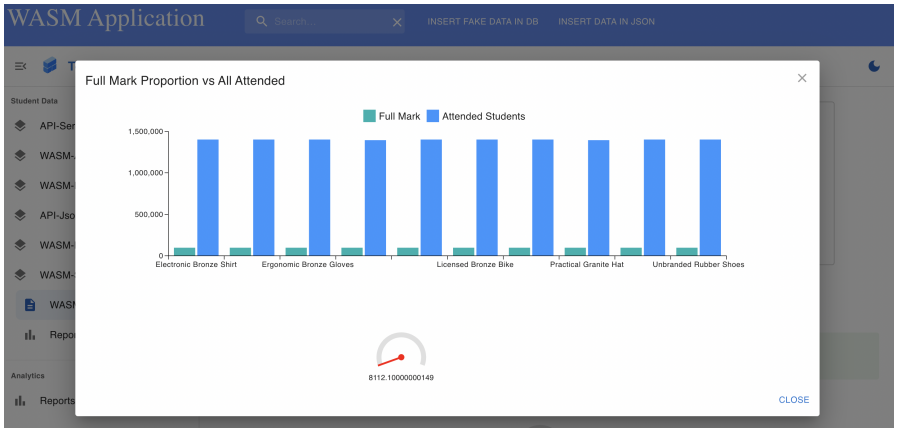

Figure 2: Report performance

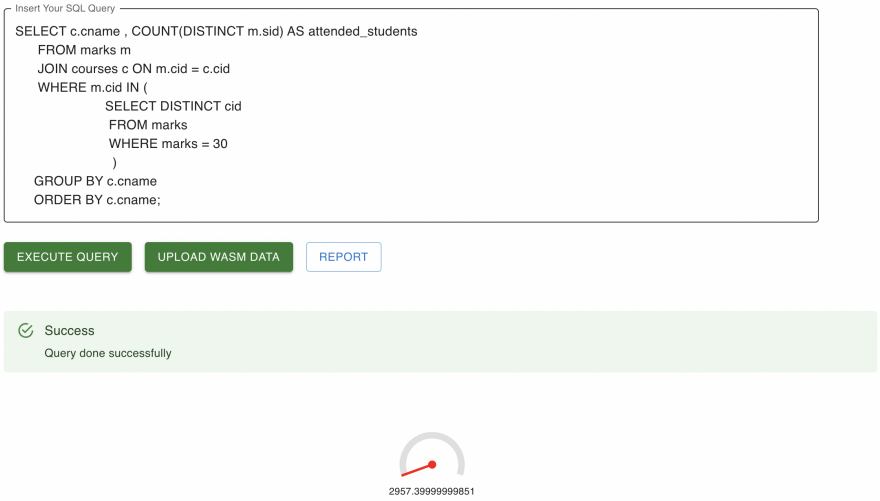

Figure 3: Query performance

Bibliography

[1] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, and Michael Holman. «Bringing the Web up to Speed with WebAssembly». In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)* (2017). URL: https://dl.acm.org/doi/10.1145/3062341.3062363 (cit. on pp. 1, 2, 5).

[2] *chrome extension to interact with DOM*. Accessed November 2024. 2024. URL: https://medium.com/%40divakarvenu/lets-create-a-simple-chrome-extension-to-interact-with-dom-7bed17a16f42 (cit. on p. 4).

[3] *Performance Comparison of WebAssembly vs. Native Apps*. Accessed November 2024. 2024. URL: https://blog.pixelfreestudio.com/webassembly-vs-native-apps-performance-comparison/?utm_source=chatgpt.com (cit. on p. 5).

[4] *WebAssembly Security*. Accessed February 2025. 2025. URL: https://webassembly.org/docs/security/?utm_source=chatgpt.com (cit. on pp. 5, 10).

[5] *Porting C and C++ Code to WebAssembly*. Accessed December 2024. 2024. URL: https://github.com/WebAssembly/design/blob/main/CAndC%2B%2B.md (cit. on pp. 6, 7).

[6] *WebAssembly.Memory() Constructor*. Accessed December 2024. 2024. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/JavaScript_interface/Memory/Memory (cit. on pp. 8, 41).

[7] Radu Matei. *A Practical Guide to WebAssembly Memory*. Accessed December 2024. 2024. URL: https://radu-matei.com/blog/practical-guide-to-wasm-memory/ (cit. on p. 9).

[8] Daniel Lehmann, Raphael Gawlik, Thorsten Holz, Dorothea Kolossa, and Christian Rossow. «Security of WebAssembly: State of the Art and Challenges». In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)* (2019). URL: https://dl.acm.org/doi/10.1145/3319535.3363192 (cit. on p. 10).

[9] *DuckDB-Wasm: Efficient Analytical SQL in the Browser*. Accessed February 2025. 2025. URL: https://duckdb.org/2021/10/29/duckdb-wasm.html?utm_source=chatgpt.com (cit. on pp. 12, 44).

[10] *Memory Management in DuckDB*. Accessed December 2024. 2024. URL: https://duckdb.org/2024/07/09/memory-management.html (cit. on p. 13).

[11] *MVC Framework*. Accessed December 2024. 2024. URL: http://geeksforgeeks.org/mvc-framework-introduction/?utm_source=chatgpt.com (cit. on p. 17).

[12] *Building Web Apps with React, WebAssembly*. Accessed December 2024. 2024. URL: https://dev.to/akshaysrepo/building-web-apps-with-react-webassembly-and-go-27f8?utm_source=chatgpt.com (cit. on p. 21).

[13] Anil Kumar Moka. *DuckDB Optimization: A Developer's Guide to Better Performance*. Accessed December 2024. 2024. URL: https://dzone.com/articles/developers-guide-to-duckdb-optimization (cit. on p. 24).

[14] *Increase the Max Memory for Node*. Accessed January 2025. 2025. URL: https://support.circleci.com/hc/en-us/articles/360009208393-How-Can-I-Increase-the-Max-Memory-for-Node?utm_source=chatgpt.com (cit. on p. 25).

[15] *Increase the Max Memory for Node*. Accessed January 2025. 2025. URL: https://expressjs.com/en/api.html?utm_source=chatgpt.com (cit. on p. 26).

[16] *Increase the Max Memory for Node*. Accessed January 2025. 2025. URL: https://dev.to/robertobutti/efficient-api-consumption-for-huge-data-in-javascript-1i72?utm_source=chatgpt.com (cit. on p. 29).

[17] *Understanding Memory Management*. Accessed February 2025. 2025. URL: https://educatedguesswork.org/posts/memory-management-1/ (cit. on p. 30).

[18] *Emscripten Compiler Frontend*. Accessed February 2025. 2025. URL: https://emscripten.org/docs/tools_reference/emcc.html?utm_source=chatgpt.com (cit. on p. 33).

[19] Iprosk. *Low-Level Code Using JavaScript*. Accessed November 2024. 2024. URL: https://dev.to/iprosk/cc-code-in-react-using-webassembly-7ka (cit. on pp. 34, 35, 37).