Evaluation of Wasm for Computation Pushdown in Cloud Database

Suhas Shripad Hebbar, Sadman Sakib, Varun Kaundinya, Ashutosh Parida

Abstract

Wasm on the web provides a compilation target with fast and predictable performance for web applications. To attain similar benefits on servers, Wasm on server side is gaining traction due to various properties such as architecture independence, portability, sandboxing and an instruction format with good mechanical sympathy, making it a good compilation target for systems languages. The sandboxing properties makes it an attractive target for cloud providers as untrusted user queries need to be run in isolation for performing tasks such as computation pushdown in multi-tenant cloud systems. The use of Wasm on the server is still in its infancy and this project evaluates and benchmark the existing support in multiple languages (C++, Rust, Java, Go) to implement database operations such as projection, filter and other UDF operation.

1 Introduction

WebAssembly, or Wasm, is a binary instruction format for secure execution of code in a sandboxed environment. Wasm allows programs written in various languages to run fast in a platform-independent manner, making it a suitable platform for performance-oriented systems, including databases [5].

Many SQL databases offer users the capability to create functions, aggregates, and other extensions to enhance extensibility. For instance, in PostgreSQL, extensions can incorporate scripts written in SQL and C code, which is then compiled into a shared library. However, the shared library lacks a secure sandbox, making it challenging to prevent excessive usage of memory and other resources by the extension. Additionally, the compiled code is not portable, being dependent on the platform where the database is running. Therefore, there is a need for a multi-language, platform-independent, and secure approach to execute these extension functions, commonly referred to as user-defined functions (UDFs).

In this project, we study the performance of UDFs exported as Wasm in different execution environments. We look at both relational operations and complex functions to understand how they perform when exported as Wasm. We explore the current support for Wasm in different programming languages and in different Wasm runtimes.

The rest of the paper is organized as follows. Section 2 covers background materials on Wasm, its use in cloud database and related works. Section 3 describes our experimental design. Section 4 contains our evaluation results. Finally, Section 5 concludes the report.

2 Background

2.1 Wasm for the Web

The first question is, why was WebAssembly created in the first place? Browsers aimed to empower developers to compile code from languages such as C++ and Rust into a single file. The goal was to enable the execution of this file at nearly native speeds within the browser while maintaining a high level of security in a well-isolated sandbox. The necessity for a secure environment arises when dealing with potentially untrusted code downloaded from the internet. Achieving near-native speed required the WebAssembly bytecode to closely resemble native instruction set architectures (ISAs), such as x86 or ARM, without being tied to any specific ISA. Consequently, a low-level abstraction was created to cover various ISAs, facilitating the seamless execution of the same binary across diverse machines with different architectures. This innovation generated interest not just from developers focused on browsers but also from those working outside the browser environment.

2.2 Wasm outside the Web

WebAssembly System Interface (WASI) aims to establish a highly modular collection of system interfaces, encompassing both fundamental low-level interfaces typical of a system interface layer and some higher-level interfaces, such as those related to neural networks and cryptography. Anticipating the incorporation of additional high-level APIs, the design adheres to capability-based security principles to uphold the integrity of the sandbox. These interfaces also strive for portability across major operating systems, ensuring broad compatibility and seamless functionality.

2.3 Wasm for Cloud Database

Some complex data queries cannot be expressed in SQL and require additional expressive power as shipped via user defined functions (UDFs). Typically applications employ SQL queries containing a combination of standard relational operations and multiple calls to UDFs often in a pipelined fashion. However, this affects query performance. Therefore, efficiently integrating UDFs with SQL requires addressing challenges such as context switching, data conversion, data copies, opening up UDFs to query optimization etc.

Modern approaches to optimizing UDF queries are in three directions: integrating UDFs with data engine, translating UDF code into SQL and translating UDF code into an internal representation [4]. For integrating UDF with data engine, some databases have in-engine support for UDFs. Some exploit various compilation optimizations to execute UDFs efficiently. Several project translate UDFs written in various languages to semantically equivalent SQL. Others translate UDFs into an intermediate representation such as array-based IR and DAG-based IR. Using Wasm for UDF engine lies in this category.

Wasm has been deployed or being considered for UDF engine of cloud databases. SinglestoreDB Code Engine supports creating UDFs and TVFs (Table-valued functions) using code compiled to Wasm [12]. The code engine uses the wasmtime runtime to compile and run wasm code. Each wasm function instance runs in its own in-process sandbox. It uses a linear memory model and provides hard memory limit. The function is allocated 16 MB of memory by default and this is configurable. To ensure security, the UDFs is not allowed to use runtime capabilities that database does not provide such as system calls, opening files, opening sockets, creating threads etc. Currently, UDFs written in any language can accept and return numeric data types. However, SingleStoreDB supports passing and returning complex data types in UDF only in C/C++ and Rust languages. A recent project has created a fast and flexible UDF engine for TiDB, an open source database, based on Wasm [11].

Another use case of Wasm is as an execution environment for serverless functions. In serverless computing, functions run in an execution environment. The most popular execution environments today are micro-virtual machines and containers. However, both suffer from large delay during initial startup, known as cold start. One solution to this problem is the introduction of more lightweight runtime environments like Wasm. Several server-side runtime for Wasm exists such as Wasmer and WasmTime. A study [8] has evaluated performance of different Wasm runtime for serverless computing.

2.4 Wasm Concepts

Some key concepts defined by Wasm specification are-

- **Module** : A module is a binary file in bytecode form that contains a sequence of sections. Each section has a unique identifier and a payload. A module contains imports and exports.
- **Memory** : The memory is represented as a mutable linear bytearray that can dynamically grow in size. It can either be created within the module or imported from the host environment.
- **Table**: A table is a data structure that holds a list of function references. The references can be used to implement indirect function calls.
- **Instance**: An instance is a stateful, executable representation of a Wasm module. It contains modules, imports, exports, memory, functions and table.

2.5 Wasm Runtimes

In this project, our evaluation focused on language support and operations across various Wasm runtimes. The crucial criterion was finding a runtime that supports multiple languages through embedding. The combination of this requirement and comprehensive documentation for the use case led us to choose the following Wasm runtimes.

1. **Wasmer**
   The Wasmer Runtime [13] serves as the engine for executing WebAssembly modules and Wasmer packages across diverse environments. It is developed in Rust and provides support for **WASI** [1]. It offers flexibility by functioning either as a library accessible from any programming language or as a standalone runtime through the Wasmer CLI. This runtime is Pluggable as in it is designed to be embedded in various programming languages, making its use case versatile and adaptable. Another key reason for this choice for runtime is that Wasmer keeps upto date with the latest WebAssembly Proposals such as SIMD, Reference Types, Threads, etc. The Wasmer Runtime also supports multiple backends which can customized as per requirement. It supports **Singlepass** backend, this focuses on low memory usage and quicker compilation times but may result in slower runtime performance . The **LLVM** backend uses LLVM for optimization and code generation which leads to higher compile times but faster runtimes. The **Cranelift** provides balance compared to the other 2 providing fast compiler backend with a focus on compile times, and

runtime performance. Wasmer Runtime also optimizes performances by **caching**, i.e WebAssembly modules reused on subsequent runs have minimal startup times.

2. **Wasmtime**
   The Wasmtime runtime [3] is another fast and secure WebAssembly runtime. It utilizes the **Cranelift** code generator for high-quality machine code generation at runtime. Wasmtime is developed in Rust prioritizing correctness and security. Wasmtime is configurable and can provide fine-grained control over CPU and memory consumption. It supports the WASI standard, providing a rich set of APIs for host environment interaction. Wasmtime's uses JIT (Just In time) compilation , with additional support for AOT (Ahead-of-time) compilation. The runtime supports both **Cranelift** and **LLVM** backends for JIT and AOT compilation.

3 Experiment Design

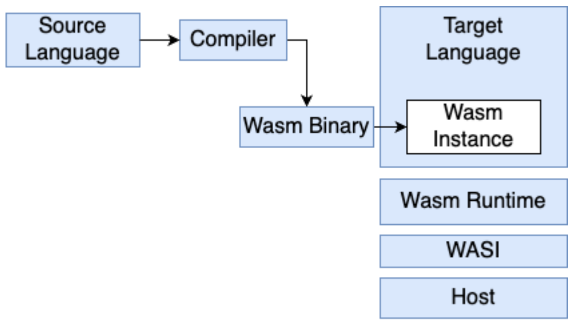

Figure 1: Compiling and running Wasm bytecode

In our project, we measured the performance of relational operations and complex functions when executed from Wasm form. Figure 1 shows the procedure for compiling and running UDFs in Wasm binary form. The Wasm runtime reads the Wasm module and compiles it into native code that can then be executed. All this is invoked in the target language through language bindings.

We used projection and selection UDFs as relational operations. The motivation is that these two functions serve as the base for more complex queries, and so understanding the performance of these operations is important. We also evaluate Wasm on sorting and on a sudoku solver benchmark from Langbench to observe its behaviour on more complex operations. Sort function takes a good amount of memory and uses a deep function stack that lets us understand its impact on performance. The sudoku function also uses a deep function stack, but performs computation on a small memory region. The UDFs were written in different source languages following a general procedure.

In the source language, relational operations are executed on a column-oriented table represented as a 2D array with two integer-type columns. This table is globally accessible via a variable. The projection function doesn't require any arguments. Within this function, a new 2D array is allocated with two columns, and a loop iterates through each row, populating the columns. One column is filled with the sum, while the other is filled with the product of the two columns in the global table.

The selection function takes two arguments: a column number, c, and a value, v. It selects a row from the global table if the value in its column c is less than v. Similar to the projection function, a new 2D array is allocated with two columns, and a loop iterates through each row to perform the selection.

In the target language, a Parquet file is read through the Arrow library. A Wasm runtime library in the target language is utilized to load and execute the Wasm module. An instance is created using the module and the necessary imports. The projection and selection functions can be looked up via their export name in the Wasm ABI. A reference to the Wasm instance memory is obtained from the instance exports. Finally, buffer data is copied from the host to the table in the instance memory. Projection and selection operations are then called, and their execution times are measured separately.

Among the two complex UDFs, the sort UDF implements a variation of merge-sort algorithm. Its time-complexity is $O(nlogn)$ and memory-usage complexity is $O(n)$. The array to merge contains small-length strings as elements. The average length of the elements is about $log(array\_length)$. First, the array is generated from a permutation process and then sorted with the merge-sort algorithm. The array length is passed from command line argument. In the Sudoku UDF, a 9x9 sudoku board is solved using a recursive backtracking algorithm. The algorithm repeatedly tries a new value for an empty field and verifies if the board is correct according to sudoku rules.

4 Evaluation Results

4.1 Testbed Setup

We ran the experiments on a Cloudlab server with x86-64 Intel CPU and 130GB RAM. Turbo boosts were disabled in the server. Table 1 shows the package versions we used for different languages as source and target for Wasm bytecode.

Language	As Source	As Target (Wasmer)	As Target (Wasmtime)
C++	Wasienv 0.5.4	Wasmer 4.2.3	Wasmtime 15.0
Python	Not used	Wasmer 2.1.1	Wasmtime 15.0
Go	Tinygo 0.26	Wasmer 2.0.0	Wasmtime 15.0
Rust	Native	Wasmer 4.2.3	Wasmtime 15.0
Java	TeaVM 0.9	Not used	Not used

Table 1: Software packages for experiment

4.2 Complex UDFs

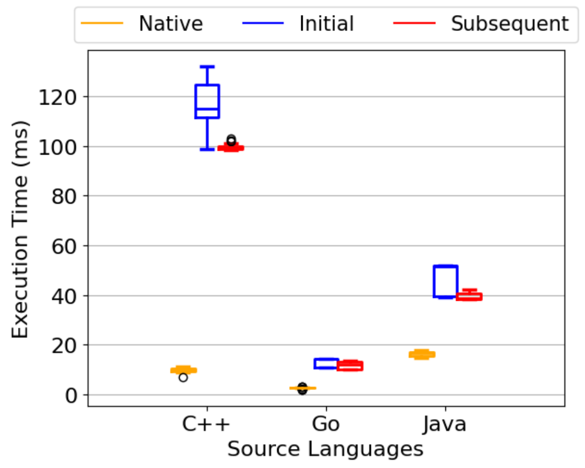

Figure 2: Execution time of sort UDF in native and Python target language

We compare the execution time of the UDFs in the source language (native) and initial and subsequent execution time in target languages for complex UDFs (sort and sudoku). The initial execution time is taken after the exported function is executed first time in target language. The subsequent execution time is taken as average time of next 10 runs of the exported function. Figure 2 and 3 shows the comparison for sort UDF in different source languages. For Figure 2, Python target language (version 3.10) with Wasmtime runtime (version 15.0) was used. For Figure 3, Rust target language (version 1.69) with Wasmer runtime (version 3.0) was used. The UDF was tasked to sort 10000 numbers. It is evident that the target language effect does not affect execution time much. However, the performance gap of native vs wasm-exported functions can depend on the source language.

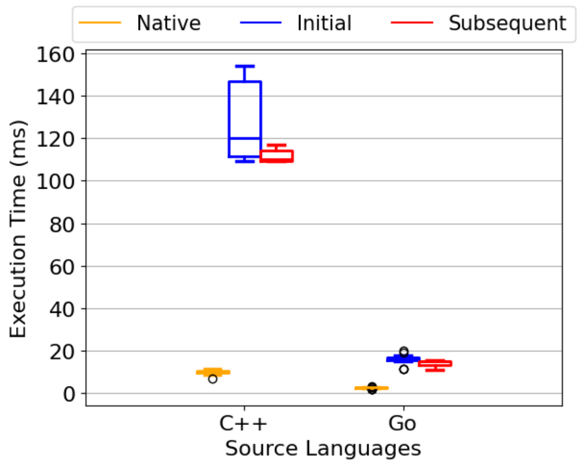

Figure 3: Execution time of sort UDF in native and Rust target language

guage. Go and Java sort UDF show 2-3 times higher execution time than native implementation. In contrast, C++ sort takes about 10 times more time than the native implementation which needs further investigation since the benchmarks for UDFs do not show this gap.

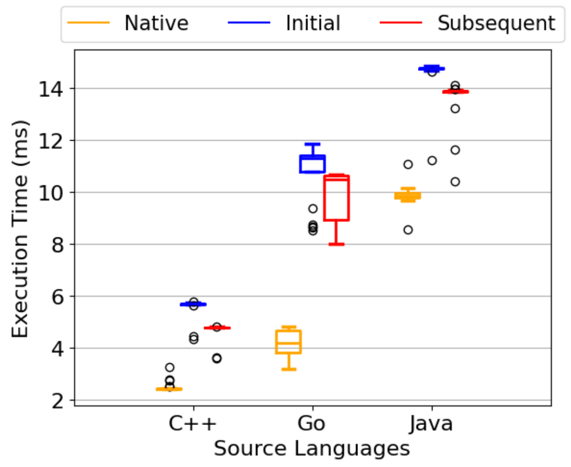

Figure 4: Execution time of sudoku UDF in native and Python target language

Figure 4 and 5 shows the performance comparison of execution of sudoku UDF. Figure 4 contains measurements with Python target language and wasmtime runtime. Figure 5 contains measurements with Rust target language And wasmer runtime. The versions are same as sort UDF measurement. Java sudoku takes the highest native time among source languages which is similar to sort UDF, but the wasm-exported function executes close to native performance. Similar to sort function, Go language UDF takes about 3x time when running as wasm-

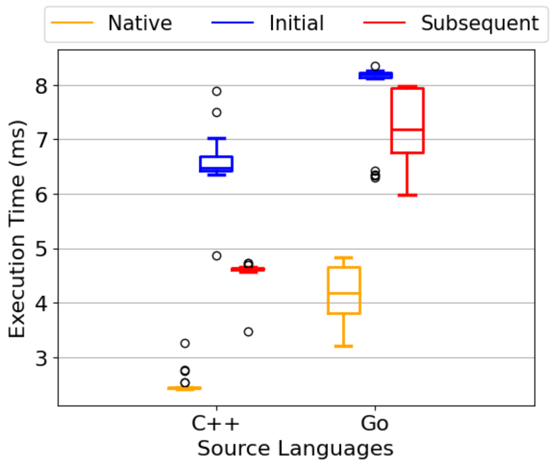

Figure 5: Execution time of sudoku UDF in native and Rust target language

exported function compared to native. The performance gap of C++ implementation is similar to Go implementation.

4.3 Pushdown UDF performance relative to Native Execution

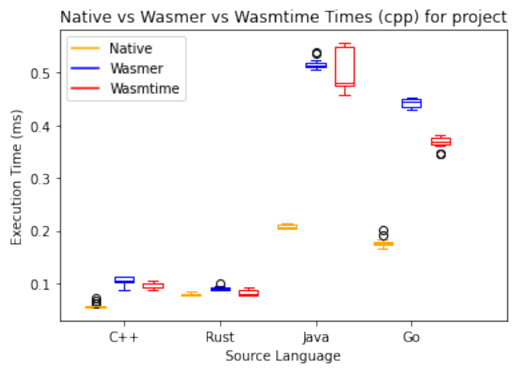

Figure 6: Comparison of performance of Project UDFs run natively vs Wasm runtimes)

Now, to consider scenarios that are representative of UDFs run in pushdown services, we consider Project and Filter UDFs which we run with a batch size of 10000 records that are fetched from parquet files.

Wasm can introduce significant overhead over native execution due to the additional indirection because of using offsets into Wasm memory over native pointers as we've seen in the **Langbench** benchmarks. We observe this overhead with Java and Go source languages for Project UDFs in 6. With C++ and Rust, Wasmer, Wasmtime and native are neck to neck within 10% of each other. This may be due to the UDFs being relatively simple with few branches and sequential memory access patterns that can be easily compiled by Wasm runtimes to efficient code.

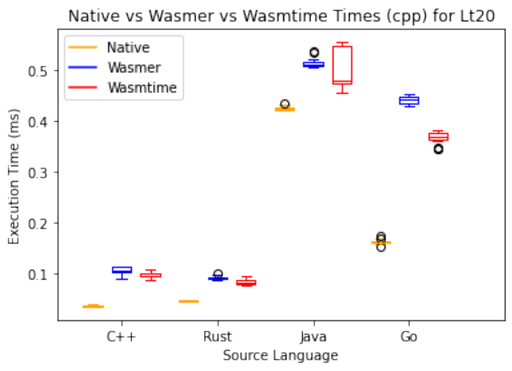

Figure 7: Comparison of performance of Filter UDFs run natively vs Wasm runtimes)

Filter UDFs as shown in 7 also have similar access patterns and codepaths to Projection and hence they also show similar performance numbers to the project UDFs with greater overhead for higher level languages with a more extensive runtime such as Go and Java when run via Wasm.

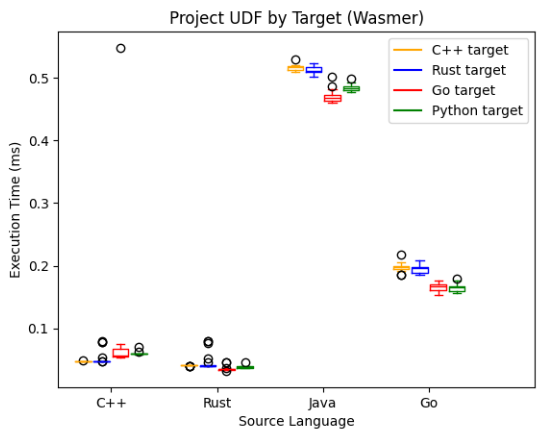

Figure 8: Comparison of execution times Wasmer execution times from different target languages)

4.4 Wasm performance from different language bindings

Given that the pushdown layer may be implemented in a different language than the Wasm runtime, inter-language FFI overhead can also be a considerable factor. The C++, Rust APIs and their bindings for both Wasmer and Wasmtime are excellent both in terms of tracking the

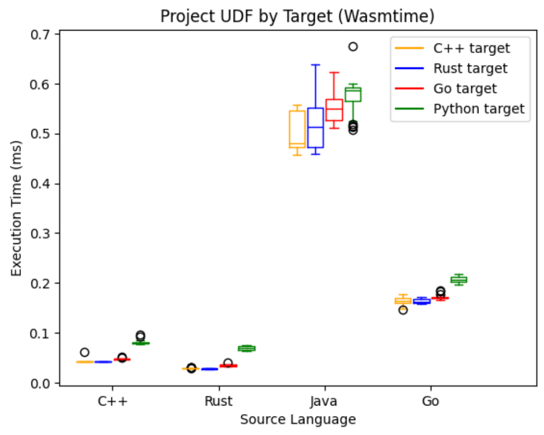

Figure 9: Comparison of execution times Wasmtime execution times from different target languages)

latest stable version of the runtimes as well as documentation. Support in other languages can lag behind stable in Wasmer while Wasmtime is excellent in this regard. The bindings from all four target languages (C++, Rust, Python and Go) we evaluated introduce minimal overhead in execution and show similar results.

5 Conclusion

Overall the ecosystem for compiling to Wasm and running Wasm is excellent around C++ and Rust while the support for compiling higher level languages may be incomplete as in the case of Java. Calling Wasm from other target languages introduces minimal overhead hence the implementation language of the pushdown layer need not be an impediment for supporting Wasm UDFs.

In terms of performance while Wasm can introduce overhead up to 10x of native it is still within an order of magnitude of native and in some cases can be even as fast as native execution.

As per our preliminary results Wasm seems to be promising as a platform for implementing pushdown computation, but further investigation on benchmarks comparable to production workloads that may have UDFs with different memory access patterns and implmentation complexity is necessary to give further justification for its use.

References

[1] Wasi, the webassembly system interface.

[2] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-Safe multilingual software sandboxing using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1975–1992, Boston, MA, August 2022. USENIX Association.

[3] bytecodealliance. Wasmtime a standalone runtime for webassembly. https://github.com/bytecodealliance/wasmtime, 2019.

[4] Yannis Foufoulas and Alkis Simitsis. User-defined functions in modern data engines. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3593–3598, 2023.

[5] Immanuel Haffner and Jens Dittrich. Fast compilation and execution of sql queries with webassembly, 2021.

[6] Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R. Lyu. Revealing performance issues in server-side webassembly runtimes via differential testing, 2023.

[7] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. Wave: a verifiably secure webassembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2940–2955, 2023.

[8] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly as an enabler for next generation serverless computing. *Journal of Grid Computing*, 21(3):34, 2023.

[9] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 205–216, 2021.

[10] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450. USENIX Association, August 2021.

[11] PingCAP. How webassembly powers databases: Build a udf engine with wasm. https://pingcap.medium.com/how-webassembly-powers-databases-build-a-udf-engine-with-wasm-1384d28342f0, 2021. Accessed: 2023-10-23.

[12] SingleStoreDB. Code engine - powered by wasm. https://docs.singlestore.com/cloud/reference/code-engine-powered-by-wasm/, 2023. Accessed: 2023-12-17.

[13] wasmer.io. Wasmer the leading webassembly runtime supporting wasix, wasi and emscripten. https://github.com/wasmerio/wasmer, 2018.