WebAssembly serverless join: A Study of its Application

Chanattan Sok, Laurent d'Orazio, Reyyan Tekin, Dimitri Tombroff

WebAssembly serverless join: A Study of its Application

Chanattan Sok and Laurent d'Orazio
chanattan.sok@ens-rennes.fr
laurent.dorazio@univ-rennes.fr
Univ Rennes, CNRS, IRISA, France

Reyyan Tekin and Dimitri Tombroff
firstname.lastname@thalesgroup.com
Thales Group, France

ABSTRACT

Big data's impact is driving research into efficient solutions for managing growing datasets, with a focus on distributed systems. Recent advancements in query processing, particularly the join operator, have been significant. WebAssembly (Wasm), known for its efficiency, is increasingly adopted. This project aims to evaluate the efficiency of Wasm in serverless environments, addressing challenges posed by serverless architectures and comparing Wasm-based joins against native in performance. We propose Blossom, a Rust-based experimental platform, to give insights into Wasm-based joins. Our initial results reveal trade-offs between Wasm performance and its generality.

KEYWORDS

WebAssembly, serverless computing, join, benchmarking

ACM Reference Format:
Chanattan Sok and Laurent d'Orazio and Reyyan Tekin and Dimitri Tombroff. 2024. WebAssembly serverless join: A Study of its Application. In *36th International Conference on Scientific and Statistical Database Management (SSDBM 2024), July 10–12, 2024, Rennes, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3676288.3676305

1 INTRODUCTION

Recent advances in cloud and big data processing [1] have led to challenges in network operations, particularly in performance, privacy, and energy consumption. The exponential growth of data centers in both usage and capacity has raised critical concerns about their CO2 footprint [9]. Serverless computing, a cloud computing model where the cloud provider dynamically manages the allocation of server-side machine resources, may offer a promising solution [2]. By allowing developers with Function as a Service (FaaS) to focus on operator, it enhances resource management and potentially reduces energy consumption, making it an active point of research.

Join is an extensively used operation, e.g. to filter data, in deployed functions over the cloud which justifies a concern in its study. However, serverless nodes may turn off and lose data, which poses challenges in managing ephemeral data with stateful operations like complex joins requiring intermediate storage. Nonetheless, other challenges arise in this distributed context such as the deployment of quick, both bandwidth-wise and performance-wise, operators while being less energy-consuming. Native operators induce a well-known time overhead called cold-start upon deployment, e.g., in Spark serverless [11]. That gives an inside to the re-usability challenge of operators taking into account ephemeral nodes. In edge/fog systems, data-privacy is also a concern [10] in transiting data, especially involved in joins.

Lots of papers on joins, for example in MapReduce or in-memory systems [12] [13] have dealt with joins optimization. To the best of our knowledge, none of the above-mentioned works has addressed efficient deployment of joins in serverless, edge or fog computing in particular to mitigate the cold-start effect. Recent works focus on local database systems with close objectives like Mutable [5] but none have specifically tackled serverless joins.

Our main goal is to generally improve joins in serverless computing. In this paper, we analyze stateless joins, e.g., Sort-Merge join for ease of deployment. Therefore, we propose leveraging a potential solution to tackle above limitations with a recent and emerging technology: WebAssembly (Wasm) [15]. Wasm is a portable binary instruction format with near-native performance, providing possible solutions to our problems. It is initially developed for web browsers enabling high-performance execution of code written in languages like C, C++, and Rust, alongside JavaScript. The use of the WebAssembly System Interface[1] (WASI) for Wasm pushes this technology beyond web-browsers. It allows standalone execution coupled with a Wasm runtime, giving opportunities in serverless computing [18] [3]. The adoption of this technology poses some new challenges as seen in [7] and we aim to point out Wasm pros over its implementation, specifically in serverless joins. As far as we know, there is not yet an existing benchmark tool to assess Wasm joins. Hence, we propose to study three join algorithms in Wasm and native, performance-wise. Our study relies on three environments to highlight Wasm features in serverless computing. Experiments are conducted using BLOSSOM, a Rust-based benchmarking tool, to assess Wasm's suitability for serverless joins compared to native. The goal of our work here is to get insights into Wasm's potential for serverless computing before its adoption, in particular with widely used join operations.

The remainder of this paper is organized as follows: Section 2 presents WebAssembly in serverless computing by outlining its pros, cons and central challenges. In Section 3 we introduce Blossom, our experimental platform to assess Wasm-based joins, along with our experimental protocol and technical challenges. The implementation details and preliminary results are given in Section 4 with a global overview, and Section 5 concludes the paper.

We would like to express our gratitude to Pr. Anne-Cécile Orgerie for her invaluable assistance with Grid'5000 and the experiments.

SSDBM 2024, July 10–12, 2024, Rennes, France

ACM ISBN 979-8-4007-1020-9/24/07
https://doi.org/10.1145/3676288.3676305

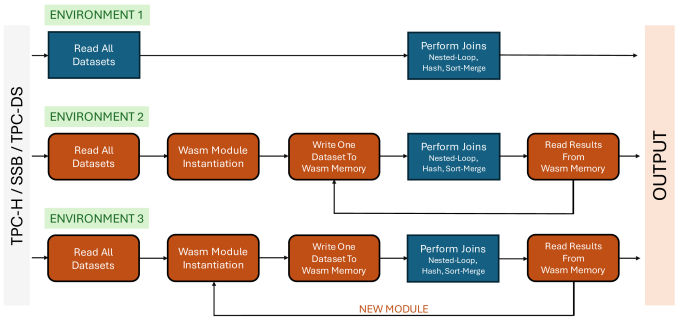

Figure 1: Experimental protocol with three environments, workloads, and joins. Rounded-boxes are steps executed by the wrapper program, and Straight-boxes by the operator.

2 WEBASSEMBLY IN SERVERLESS

This section outlines Wasm traits in serverless computing and addresses rising challenges with its potential adoption. Especially, we think Wasm may present several advantages addressing the aforementioned limitations:

1. **Re-usability**: Wasm operators compared to native ones may offer easier deployment in serverless environments [4], as standalone programs to be run on a runtime. They would be lightweight alternatives [3] and thus easier to dispose off.
2. **Near-native speed**: Wasm provides competitive speed compared to native programs, balancing generality and performance [15].
3. **Less energy-consuming**: We believe from point 1 that Wasm enables lower cold-start times [4] [3] and less energy-consuming operators [14] [16], consequently improving serverless computing.
4. **Language-interoperability with a portable binary format**: Wasm supports widely used languages like C++, C and Rust, simplifying developers choices. Easier architecture maintenance for cloud providers is also possible with a portable format only requiring a Wasm runtime such as Wasmtime[2] [17].
5. **Sandboxed environment**: Wasm programs have enforced security as they run in isolated environments, disallowing direct resource access. This enhances data-privacy in transiting data and resilience against cyber-attacks. Such property can be a direct alternative to containerization for classic operators, e.g., with Docker [3].

Nonetheless, the adoption of Wasm presents three central challenges in serverless computing with its recent portability out of the web. We identify the three following challenges:

1. **Limited memory**: As of the current Wasm32 version, a Wasm's module is limited to 4GB of linear memory with 32-bit addressing. This highlights a challenge in a big data context to handle large workloads. Although, some work is already being done towards Wasm64, which would remove this memory-limit barrier.
2. **Early state**: Wasm and WASI are still in development, with WASI being in early state. Several technical details need to be known, and a few limitations are to be considered when dealing with simple programs. For instance, memory-communication from user to a Wasm module would often need an intermediate program wrapper. This program would instantiate the Wasm module and allow classic native IPC-like/IO standard communications.
3. **Performance**: A central point is that Wasm was originally developed to run web applications, Wasm promises native-close performance in this initial context: the performance comparison lies between interpreted JavaScript and Wasm. Traditionally, web applications are run with interpreted JavaScript, whereas Wasm-applications are applications that have been compiled to Wasm to produce Wasm byte code. Afterward, Wasm-applications are executed directly on a Wasm runtime part of web browsers or standalone Wasm virtual machines (e.g., V8[3]), often using JIT or other optimization techniques. In our context, we consider Wasm outside the web to execute sandboxed modules with WASI: the performance is compared between a given Wasm runtime and raw native.

3 BLOSSOM

We propose BLOSSOM[4], an experimental platform to analyze and compare Wasm-based joins. To address above-mentioned challenges, we consider the experimental protocol illustrated in Figure 1 presenting the join operator execution in three environments:

Env. 1) No-serverless native: The join operator runs in native Rust and performs join algorithms on all loaded datasets.

Env. 2) No-serverless Wasm: Similar to 1. but with Wasm; a developed wrapper program acts as an intermediate to give progressive data throughput to the Wasm operator. There is only one Wasm module instantiation, i.e., execution.

Env. 3) Serverless Wasm: For each loaded dataset, a new Wasm module is instantiated and then executed. This mimics a serverless node with ephemeral data.

A fourth environment would be the Serverless native one, similar to Env. 3 but instead considering a native-compiled operator that would be run over again for each dataset. It has not been explored initially, given its lesser relevance regarding performance and known cold-start aspects compared to lightweight Wasm operators. A notable point lies in Wasm overheads with its supplementary steps. Our experimental protocol is:

(1) Load the datasets: directly in the operator in native and in the wrapper program passed in the module for Wasm.
(2) Only for Envs. 2 and 3, write data to the Wasm module. This step is necessary considering current Wasm32 memory limitation.
(3) Wasm or native operator runs the three join algorithms for a given dataset.
(4). For Envs. 2 and 3, read join results in Wasm memory from the wrapper program.

We measure time for data loading and results retrieval ((2), (4)) and operator execution ((3)) for raw performance comparison. Step (1) has no measuring, as it is similar for all Envs. We emphasize Env. 1 executes the join program directly, while Envs. 2 and 3 require a native wrapper program, here embedded in BLOSSOM, for handling large data as native programs are only limited in memory with the RAM. This is one of the required workarounds due to Wasm's limited memory and early state challenges. There are a few ways to communicate to Wasm modules, in particular the following three:

i. **Linear allocation**: Manipulate Wasm's linear memory object, which is a contiguous array shared amongst Wasm modules. This is possible through the use of the associated Wasm runtime library.

[1]https://wasi.dev/
[2]https://wasmtime.dev/
[3]https://v8.dev/
[4]The project is available on GitHub: https://github.com/chanattan/Blossom

ii. **Pipe communication**: As WASI was originally designed to be POSIX-like, we may communicate with Wasm modules through command line arguments, environment variables or pipes. Especially, stdin and stdout streams to pass data in and out.
iii. **Libraries**: Some tools like Wasm-bindgen[5] are being developed to facilitate high-level interactions between Wasm modules and JavaScript.
In this first study, we focused on approach i. with linear allocation. The main pro of this technique is that linear memory allows fast data communication. This is a possible solution to the Wasm performance challenge. However, this solution incurs notable development complexity in managing memory on the runtime because of Wasm's early state. The other approaches were not yet fully studied, comparatively, pipe communication lacks of flexibility: input is passed in the program with stdin, but the output is only received from stdout once the program is done computing. In order to avoid potential performance slowdown or another layer of complexity with JavaScript, approach i. was considered. In our experiments, all data is read raw as strings and manually serialized given hard-coded table structures, with future support for optimal serialization envisioned. A central hypothesis of our protocol lies in a disaggregated-database-like setup with an independent data storage layer (as in S3, MinIO or HDFS) that is close to the compute layer of a node: we assume close available datasets in experiments.

We study three join algorithms — Nested-Loop, Hash and Sort-Merge [13] — implemented directly within the join operator. The conducted benchmarks are based on relational data with SQL join-queries. For starters, we use the TPC-H bench test, but others like TPC-DS or SSB could be used with minor changes. Similar to previous studies [12], we focus on equi-join queries involving a single key attribute join from two relations. The queries are based on TPC-H Q-queries (2,3,5,8,9) simplified to only relevant join parts.

We consider Wasm join operators as a potential solution for serverless joins based on the outlined advantages above. Nonetheless, this prompts further questions with Wasm central challenges: What is the development and maintenance burden of Wasm operators? To which extent are these operators re-utilizable? Is the trade-off between generality and performance justified with out-of-the-web Wasm? How effectively Wasm impacts cold-start times and energy use? Our experimental protocol is still in development (Figure 1) to give possible solutions, with some aspects slated for future research due to space constraints.

4 PRELIMINARY RESULTS

All code is written in Rust, the join operator is compiled to both native and Wasm32-WASI. We use the Wasmtime runtime as it is lightweight and has competitive performance, but any other WASI-supported Wasm runtime, e.g., Wasmer, [6] [8] could be used with minor revisions, given the runtime has a Rust library. Experiments are conducted on a Grid'5000[6] cluster equipped with an Intel Xeon Gold 6254 (18 cores) with 384 GiB of RAM.

[5]https://rustwasm.github.io/wasm-bindgen/
[6]Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.Grid5000.fr)

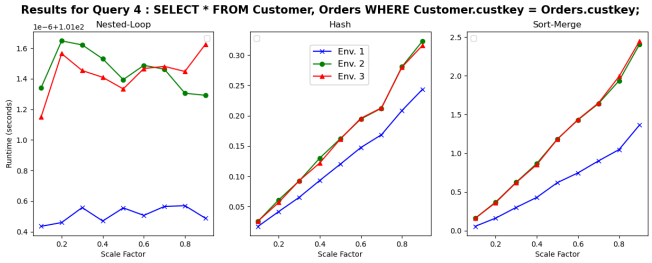

Figure 2: Performance results for Nested-Loop, Hash and Sort-Merge in the three environments for a fixed query.

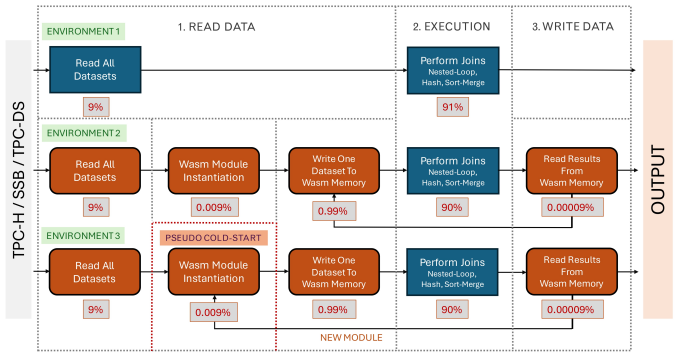

Figure 3: Experimental protocol with highlight on reading, execution and results writing big steps. Each detailed step on each environment shows its part of time taken on the global process (in percentage) from left to right. Each block column corresponds to a step in the join operator execution, and each line to the execution timeline for an environment.

The joins are performed on the TPC-H datasets with different Scale Factors (SF). Because of Wasm limited memory, we first restricted SF values from .1 (~100MB) to 1 (~ 1GB) with a step of .1. We use the join throughput metric from [13], calculated as $\frac{|R|+|S|}{|runtime|}$, where $|R|$ and $|S|$ denote the tuple counts of two relations, and $|runtime|$ represents the join operation time in seconds.

Figure 2 shows the results of native and Wasm joins raw performance on the cluster for the three join algorithms: Nested-loop, Hash and Sort-Merge joins on different SF values. It is important to note that drawing reliable conclusions requires higher SF and more consolidation due to significant execution runtimes, this presents preliminary results. Moreover, Wasm and Wasm runtimes' early state gave us technical challenges in order to implement our experiments. On top of that, we encountered difficulties with LLVM-based tool chain for wasm32, e.g. wasi-sdk or emscripten, and in our case with Rust to allocate a single linear memory object larger than 1GiB. This problem restricted the SF values for first results with our experiments. A considered solution is to implement a custom allocator with four independent linear memory objects less than 1GiB, in order to make the most use of Wasm32 4GiB linear memory.

Nonetheless, these suggest that Wasm exhibits slower performance compared to native, with a performance gap widening at higher SF values. This was expected given Wasm performance central challenge in section 2, where Wasm-WASI performances are being compared to raw native instead of Wasm against JavaScript on the web. Similar performance can be seen for both Envs. 2 and 3 as Figure 2 only focuses on the join. On average, Wasm shows

performance gaps of 100%, 25%, and 47% below native for Nested, Hash, and Sort-Merge algorithms, respectively. It can also be noted that tested Wasm modules have been produced with by-default JIT compilation, but not with AOT compilation. Preliminary analysis indicates that performance gaps may persist even with the potential benefits of Wasm64 with higher memory, emphasizing particular attention of out-of-the-web Wasm performance central challenge.

Figure 2 provides a comparison of the execution part, as depicted in Figure 3. For preliminary results, only the orders of magnitude of other steps are displayed. For instance, at the same SF values, Wasm demonstrates a magnitude of $10^{-4}s$ for module instantiation, $10^{-2}s$ for writing datasets, and $10^{-6}s$ for reading results. Figure 3 presents the same experimental protocol as Figure 1, emphasizing the three significant steps in join function execution: 1. Reading data, 2. Executing the join, and 3. Writing results. Using empirically collected mean values as magnitude orders, we can assign percentage values corresponding to each block column (each step) as part of the total time taken for the entire protocol execution, from step 1 to step 3, for each environment. These values provide insights into the distribution of time spent during the execution of a join operator in a given environment. Notably, Figure 3 reveals that the majority of time spent in executing a join operator is dedicated to the actual join execution across all three environments, in particular with Env. 3, hinting at serverless Wasm potentially being a considerable solution. In particular, we observe a pseudo cold-start in Env. 3 when instantiating a new Wasm module for each dataset, mimicking a serverless node with ephemeral data. However, the figure indicates that about 0.001% of the join operator execution time is consumed by this pseudo cold-start. Additionally, only about 1% of the time is allocated to memory communication of a Wasm module, which may become significant with larger workloads but remains minor compared to the execution time, which takes around 90% of the time. Regarding pseudo cold-start, only Env. 3 accounts for this aspect, as Env. 2 mimics a Wasm-based serverful node, and Env. 1 can be considered as having zero pseudo cold-start in our setup. While a startup time for Env. 2 can be seen, it occurs only once.

5 CONCLUSION

The current implementation of WebAssembly shows noticeable overheads in both technical implementation and raw performance. Although it was expected to be slower than native code, out-of-the-web Wasm still offers significant advantages like enhanced security, reusability and may ease developer choices and server-side maintenance. The potential for energy efficiency in serverless Wasm is still being explored. Our experiments suggest that some technical overheads in Wasm may be negligible compared to join execution aspects. Consequently, there is growing interest in improving the raw performance of Wasm programs, for a given Wasm runtime. Some specific Wasm runtimes with optimized out-of-the-web scenarios are also explored by peers. With the upcoming release of Wasm64, performance levels may remain consistent, albeit with larger memory allocation capabilities. This hints at potential solutions involving parallelism within Wasm, using parallel Wasm modules on a single Wasm runtime. It may require an orchestration of join operations for efficient task distribution, as seen with semi-joins. Our findings show necessary trade-offs between Wasm32 performance and versatility, with performance being a significant concern. However, ongoing work aims to optimize join execution and compilation in Wasm, making it a considerable solution.

REFERENCES

[1] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. 2011. Big data and cloud computing: current state and future opportunities. In *International Conference on Extending Database Technology (EDBT)*. 530–533.

[2] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*. 1–20.

[3] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. 2022. Pushing Serverless to the Edge with WebAssembly Runtimes. In *International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 140–149.

[4] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge. In *International Middleware Conference (Middleware)*. 265–279.

[5] Immanuel Haffner and Jens Dittrich. 2023. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. In *International Conference on Extending Database Technology (EDBT)*. 1–13.

[6] Devon Hockley and Carey Williamson. 2022. Benchmarking Runtime Scripting Performance in Wasmer. In *International Conference on Performance Engineering (ICPE*. 97–104.

[7] Cynthia Marcelino and Stefan Nastic. 2023. CWASI: A WebAssembly Runtime Shim for Inter-Function Communication in the Serverless Edge-Cloud Continuum. In *Symposium on Edge Computing (SEC)*. 158–170.

[8] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An Embedded Trusted Runtime for WebAssembly. In *International Conference on Data Engineering (ICDE)*. 205–216.

[9] Waqaas Munawar, Jian-Jia Chen, and Minming Li. 2014. Minimizing Environmental Footprints of Data Centers under Budget and Service Requirement Constraints. In *International Conference on Smart Grids and Green IT (SMARTGREENS)*. 222–232.

[10] Shalin Parikh, Dharmin Dave, Reema Patel, and Nishant Doshi. 2019. Security and Privacy Issues in Cloud, Fog and Edge Computing. In *International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN)*. 734–739.

[11] An-Truong Tran Phan, Laurent d'Orazio, and Le Gruenwald. 2024. GUESS: Monitoring Join Query Execution in Serverless and Serverful Spark. In *International conference on Database Systems for Advanced Applications (DASFAA)*.

[12] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1749–1762.

[13] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *International Conference on Management of Data (SIGMOD)*. 1961–1976.

[14] Merlijn Sebrechts, Tim Ramlot, Sander Borny, Tom Goethals, Bruno Volckaert, and Filip De Turck. 2022. Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI. In *International Conference on Cloud Networking (CloudNet)*. 195–202.

[15] Benedikt Spies and Markus Mock. 2021. An Evaluation of WebAssembly in Non-Web Environments. In *Latin American Computing Conference (CLEI)*. 1–10.

[16] Linus Wagner, Maximilian Mayer, Andrea Marino, Alireza Soldani Nezhad, Hugo Zwaan, and Ivano Malavolta. 2023. On the Energy Consumption and Performance of WebAssembly Binaries across Programming Languages and Runtimes in IoT. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 72–82.

[17] Wenwen Wang. 2022. How Far We've Come - A Characterization Study of Standalone WebAssembly Runtimes. In *International Symposium on Workload Characterization (IISWC)*. 228–241.

[18] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring the Edge up to Speed with A WebAssembly OS. In *International Conference on Cloud Computing (CLOUD)*. 353–360.