# Serverless Runtime / Database Co-Design With Asynchronous I/O

Pekka Enberg
University of Helsinki

Sasu Tarkoma
University of Helsinki

Jon Crowcroft
University of Cambridge

Ashwin Rao
University of Helsinki

## ABSTRACT

Minimizing database access latency is crucial in serverless edge computing for many applications, but databases are predominantly deployed in cloud environments, resulting in costly network round-trips. Embedding an in-process database library such as SQLite into the serverless runtime is the holy grail for low-latency database access. However, SQLite's architecture limits concurrency and multitenancy, which is essential for serverless providers, forcing us to rethink the architecture for integrating a database library.

We propose rearchitecting SQLite to provide asynchronous bytecode instructions for I/O to avoid blocking in the library and decoupling the query and storage engines to facilitate database and serverless runtime co-design. Our preliminary evaluation shows up to a 100x reduction in tail latency, suggesting that our approach is conducive to runtime/database co-design for low latency.

## CCS CONCEPTS

• **Information systems** → **Database management system engines**; • **Networks** → *Cloud computing*.

## KEYWORDS

Edge computing, serverless computing, in-process databases

**ACM Reference Format:**
Pekka Enberg, Sasu Tarkoma, Jon Crowcroft, and Ashwin Rao. 2024. Serverless Runtime / Database Co-Design With Asynchronous I/O. In *7th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3642968.3654821

## 1 INTRODUCTION

> *If at least one of these attributes [bandwidth scalability, low end-to-end latency, data privacy, high availability, data export compliance] is crucial for the application, then it is an edge-native application. If none of them are really needed, none of them is essential, then edge computing is an expensive luxury.*
>
> Mahadev Satyanarayanan [10].

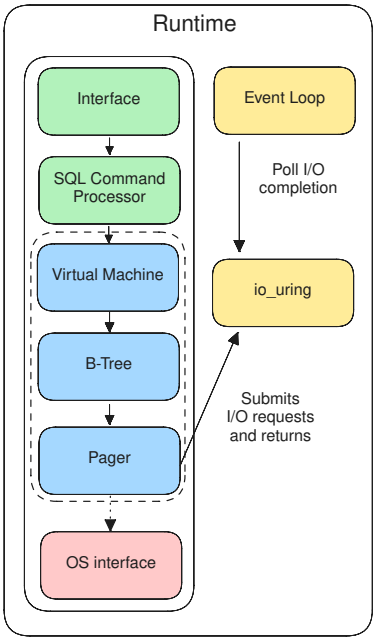



**Figure 1: Rearchitecting SQLite for asynchronous I/O.** *The SQLite architecture is synchronous, i.e., application threads block on I/O at the OS interface. To support asynchronous I/O, we propose changing SQLite—at the virtual machine layer and below—to asynchronously submit I/O using an interface such as* `io_uring` *and return control to the application thread. This enables each application thread to concurrently execute computational and I/O tasks.*

Applications built to run on cloud and edge/fog platforms are increasingly being developed as serverless functions. The serverless paradigm aids application development by offloading the problems of scaling and deployment of computing resources to the platforms. For instance, serverless run-times such as Cloudflare Workers [4] allow application developers to deploy applications in a location-agnostic manner. A key benefit of developing location-agnostic applications is that they can be easily migrated from one data center to another and, thus, from the cloud to the edge. The *serverless paradigm is therefore beneficial for creating and deploying location agnostic applications that can be deployed at the edge.*

Along with location agnostic applications, serverless runtimes are increasingly leveraged for developing latency-critical applications. This was not always the case. Legacy serverless runtimes suffered from high run-time overheads because their underlying control and data planes were not designed to serve the short-lived functions [2, 8, 12]. We therefore believe that, along with location


*EdgeSys '24, April 22, 2024, Athens, Greece*

ACM ISBN 979-8-4007-0539-7/24/04
https://doi.org/10.1145/3642968.3654821

agnosticism, *serverless run-times will be useful for latency-critical edge-native applications.*

Applications that require low end-to-end latency are good candidates to be deployed as edge-native applications. Latency-critical edge-native applications require co-locating the data used by the application with the computing resources that process the data.

One practical approach to co-locate data with the computing resources deployed at the edge is to leverage the multi-region support in databases. This involves deploying multiple database replicas across different geographical regions. However, even with replication, database access still requires a network round-trip from the serverless runtime to the database.

The holy grail for achieving minimal database access latency is to integrate the database into the application itself. This approach eliminates the need for network round-trips, thereby reducing query execution to efficient in-process function calls. Light-weight data stores such as in-memory key value stores are typically deployed within serverless runtimes when data can be moved closer to the compute. For instance, Cloudflare Workers KV [5] exemplifies in-memory key-value stores that are optimized to run in the serverless runtimes. However, databases that support SQL are deployed in traditional, non-serverless cloud environments, partly because stateful workloads are challenging for serverless computing [9]. SQLite [7], an in-process database management system, has therefore gained widespread adoption across diverse use cases, solidifying its position as the most widely deployed database solution. While embedding SQLite into the serverless runtime seems obvious for co-locating computing and data [9], its architecture makes that challenging.

In this paper, we first delve into the internals of SQLite and highlight that synchronous I/O is deeply rooted in its architecture (Figure 1). This makes SQLite a bottleneck when deploying latency critical serverless applications that use it. We propose changing the SQLite virtual machine to provide asynchronous bytecode instructions. We show that our modifications help serverless runtimes avoid blocking in the library and decoupling the query and storage engines to facilitate database and serverless runtime co-design. Our preliminary evaluation shows up to a 100x reduction in tail latency, suggesting that our approach is conducive to runtime/database co-design for low latency.

## 2 BACKGROUND AND MOTIVATION

### Serverless computing and data

Serverless computing is a paradigm where applications can execute code without dealing with resource allocation, users pay for resources used instead of the resources allocated, and computation and storage are decoupled [9]. For example, Cloudflare Workers [4] is a serverless platform that internally uses V8 isolates, a light-weight execution environment to run JavaScript applications in isolation from each other within a single OS process. With Cloudflare Workers, an application developer writes JavaScript functions to perform application logic, and this code can run across all of Cloudflare's locations. Cloudflare routes user requests automatically to the closest location, providing location-agnostic serverless computation. Serverless computing platforms, such as Cloudflare Workers, offer location-agnostic computing which can be leveraged for developing and deploying edge-native applications.

Most real-world applications need to manage state and access data, but serverless state management continues to be an unsolved problem [9, 16]. Many serverless platforms offer in-memory key-value stores for low latency data access. However, they may be unsuitable for many tasks. For instance, their domain-independent interface pushes complexity to the application, they may require network round-trips, and they incur (un)marshaling overheads [1]. To address these limitations of in-memory key-value stores, many applications end up externalizing data access to a cloud database [16] which offers transactionality and a query language such as SQL. However, accessing these cloud databases can incur costly round-trips to the database. One approach to reduce the database access latency is to leverage multi-region support, i.e., deploying multiple database replicas across different geographical regions. However, even with replication, database access still requires a network round-trip from the serverless runtime to the database.

The holy grail for achieving minimal database access latency is to integrate the database into the application itself. This approach eliminates the need for network round-trips, thereby reducing query execution to efficient in-process function calls.

### Synchronous I/O in SQLite architecture

> *Think of SQLite not as a replacement for Oracle but as a replacement for fopen() [13].*

SQLite is an in-process database management system [7]. Unlike traditional databases that have a client-server architecture, SQLite embeds within an application. Applications access databases by invoking library functions provided by SQLite, which use the filesystem to store data and write-ahead logs (WAL). Query processing happens in the context of an application thread; there are no dedicated database-specific processes or threads that execute SQL queries. Consequently, if SQLite blocks during I/O operations, the application thread also blocks.

SQLite has gained widespread adoption across diverse use cases, solidifying its position as the most widely deployed database solution [6]. As shown in Figure 1, its architecture has two main parts: the core and the backend. The core consists of the SQLite C API interface for applications, an SQL command processor, and a virtual machine that executes SQL queries. The backend consists of a B-tree access module, a pager that brings pages from disk to memory, and an OS interface for performing I/O. While embedding SQLite into the serverless runtime seems obvious for co-locating computing and data [9], its blocking I/O architecture makes it challenging.

We now present an overview of these challenges by taking a shallow dive into what happens when an application executes the query `SELECT * FROM users`. This query performs a full table scan on a table `users`. An application first opens a database with the `sqlite3_open()` function. As SQLite uses files for storing the data, the `sqlite3_open()` function invokes the low level OS I/O, such as POSIX `open`, for opening the file. The application then prepares a SQL statement using the `sqlite3_prepare()` function that transforms SQL statements such as `SELECT` and `INSERT` into

sequences of bytecode instructions. The application then executes the statement with the `sqlite3_step()` function.

The `sqlite3_step()` function executes the sequence of bytecode instructions until the query produces a row to read, or it completes. When there is a row, the function returns the `SQLITE_ROW` value and when the statement is complete, the function returns `SQLITE_DONE`. The `sqlite3_step()` function internally calls into the backend pager, traversing the database B-trees representing tables and rows. If a B-Tree page is not in the SQLite page cache, the page has to be read from disk. SQLite uses synchronous I/O such as the `read` system call in POSIX to read the page contents from disk to memory, which means the `sqlite3_step()` function blocks the kernel thread, requiring applications to utilize more threads to perform work concurrently to the I/O wait.

Synchronous I/O can lead to increased contention among applications for computing and I/O resources, and also incur increased latency. The increased latency can make serverless environments unsuitable for latency-critical edge-native applications, while the increased contention hampers multi-tenancy. This can lead to inefficient resource utilization and consequently higher operational costs to the customers of serverless platforms. Consequently, the costs incurred by SQLite's synchronous I/O are elevated in serverless environments.

### Towards asynchronous I/O with `io_uring`

The traditional POSIX I/O system calls `read()` and `write()` are synchronous, which means that the thread making the system call will block until I/O is complete. This blocking is detrimental to concurrency and parallelism because it causes the application to wait, thereby reducing the overall utilization of computing resources.

The `io_uring` subsystem in the Linux kernel provides an interface for asynchronous I/O. Its name has its roots in the ring buffers shared between user space and kernel space that alleviate the overheads of copying buffers between them [3]. It allows the application to submit an I/O request and concurrently perform other tasks until it receives a notification from the OS on the completion of the I/O operation. With `io_uring`, the application first calls the `io_uring_setup()` system call to set up two memory regions: the submission and completion queues. The applications then submit I/O requests to the submission queue and call the `io_uring_enter()` system call to tell the OS to start processing the I/O requests. However, unlike the blocking `read()` and `write()` calls, `io_uring_enter()` does not block the thread by returning control to the userspace. The application can now perform other work concurrently and periodically poll the completion queue in userspace for I/O completion.

Replacing the POSIX I/O calls with `io_uring` is not trivial, and applications that use blocking I/O must be re-designed for the asynchronous I/O model of `io_uring`. Specifically, applications now need to handle the situation of I/O submission in their control flow. In the case of SQLite, the library needs to return control to the application when I/O is in flight.

*In summary, serverless edge platforms solve the problem of location-agnostic computing. However, as many real-world applications need to manage state and access data, serverless runtime and database co-design is needed to retain the latency advantage. SQLite as an embedded database management system is an excellent fit, but its synchronous architecture makes it suboptimal for integrating with the serverless runtime. We, therefore, need to rearchitect SQLite for asynchronous I/O to allow high concurrency and multitenancy, which we discuss in the next section.*

## 3 OUR SOLUTION: LIMBO

We posit refactoring SQLite's architecture for asynchronous I/O to facilitate serverless runtime co-design. As shown in Figure 1, our solution, *Limbo*, removes interaction with the OS interface for I/O. Instead, the application provides an I/O module to the library, which the library calls to perform I/O such as read or write B-tree pages. We have implemented an I/O module using `io_uring`, which provides asynchronous I/O support on Linux, but the mechanism is agnostic to the I/O interface. The virtual machine, b-tree, and pager components are changed to accommodate for asynchronous I/O with two building blocks:

- Asynchronous bytecode instructions to avoid blocking in the library.
- Decoupling query and storage engines to facilitate the co-design of the database and serverless runtime.

*Limbo* retains the interface and SQL command processor components of SQLite. In the current version of *Limbo*, they are conceptually the same; we reimplemented them in Rust using the `sqlite3-parser` package, which is a translation of the SQLite parser from C to Rust. However, in the future, we are considering reusing more SQLite components for compatibility.

### 3.1 Bytecode instructions for asynchronous I/O

When the SQLite virtual machine executes instructions to traverse the B-tree, the pager module may have to perform I/O to read the pages into memory. Similarly, when the virtual machine commits a transaction, it must write pages to the disk. To avoid blocking in the library, we propose to change the SQLite bytecode instruction set to provide asynchronous variants for instructions that perform I/O. In the example presented in Figure 2, the blocking Open, Rewind, and Next instructions are replaced by their non-blocking counterparts: OpenRead is replaced by OpenReadAsync and OpenWaitAsync, Rewind is replaced by RewindAsync and RewindWait, and Next is replaced by NextAsync and NextWait.

For brevity, we delve into the details of only one instruction, `Next`. The `Next` instruction advances a cursor to point to the next available row, and may need to read b-tree pages from the disk. In *Limbo*, the SQL compiler generates a pair of instructions `NextAsync` and `NextAwait`, instead of `Next`. The `NextAsync` instruction submits I/O asynchronously and returns from the `sqlite3_step()` function with a `SQLITE_IO` result to indicate that I/O was submitted. The application can then either call into `sqlite3_step()` again to execute `NextAwait` to block on the I/O, or keep performing other operations until the I/O is complete. An external I/O dispatch loop such as `io_uring` notifies the application on I/O completion.

### 3.2 Decoupling query and storage engines

The asynchronous architecture eliminates blocking and improves concurrency in the serverless runtime, allowing for more tenants. However, it is also essential to decouple the query and storage

```
sqlite> EXPLAIN SELECT * FROM users;
addr  opcode         p1    p2    p3    p4             p5
----  -------------  ----  ----  ----  -------------  --
0     Init           0     8     0                    0
1     OpenRead       0     2     0     2              0
2     Rewind         0     7     0                    0
3       Column         0     0     1                    0
4       Column         0     1     2                    0
5       ResultRow      1     2     0                    0
6     Next           0     3     0                    1
7     Halt           0     0     0                    0
8     Transaction    0     0     1     0              1
9     Goto           0     1     0                    0
```

**(a) SQLite bytecode output for** `SELECT * FROM users`

```
> EXPLAIN SELECT * FROM users;
addr  opcode         p1    p2    p3    p4             p5
----  -------------  ----  ----  ----  -------------  --
0     Init           0     11    0                    0
1     OpenReadAsync  0     2     0                    0
2     OpenReadAwait  0     0     0                    0
3     RewindAsync    0     0     0                    0
4     RewindAwait    0     10    0                    0
5     RowId          0     0     0                    0
6     Column         0     1     1                    0
7     ResultRow      0     2     0                    0
8     NextAsync      0     0     0                    0
9     NextAwait      0     4     0                    0
10    Halt           0     0     0                    0
11    Transaction    0     0     0                    0
12    Goto           0     1     0                    0
```

**(b) Limbo bytecode output for** `SELECT * FROM users`

**Figure 2: Bytecodes for Asynchronous I/O.** *Limbo retains the registers used by the SQLite bytecode, and the key differences are that the I/O operations have their Async and Wait counterparts. In this example, OpenRead is replaced by OpenReadAsync and OpenWaitAsync; Rewind is replaced by RewindAsync and RewindWait; Next is replaced by NextAsync and NextWait. The above output is obtained for a table created using* `CREATE TABLE users (id INT PRIMARY KEY, username TEXT).`

engines to minimize query latency and maximize the density of functions per CPU and DRAM.

In *Limbo*, I/O is external to the database library with a virtual I/O module. The application provides and implements this virtual I/O module. In Figure 1, the serverless runtime provides an I/O module that is based on `io_uring`. The database library performs I/O by submitting an I/O request to the `io_uring` submission queue via this I/O module, and it then returns control to the runtime. The runtime then polls `io_uring` completion queue as part of its event loop. When I/O completes, the runtime can resume control to the database library. The decoupling of the query engine, which is implemented in the database library, and storage, which is handled external to the library, is the key element in enabling co-design between the runtime and the database. This decoupling is essential because it eliminates blocking and allows the runtime to efficiently implement multitenancy.

# 4 PRELIMINARY EVALUATION

The goal of our evaluation is to answer the question: *what is the impact of synchronous and asynchronous I/O in query tail latency?*

## 4.1 Methodology

To answer this question, we built a microbenchmark to experimentally compare the performance of *Limbo*, our prototype, with *SQLite*. For both systems, we benchmark a scenario where we simulate a multi-tenant serverless runtime where each tenant is a serverless function that accesses their own database, and executes a `SELECT * FROM users LIMIT 100` SQL query 1000 times. For the benchmark, we vary the number of tenants from 1 to 100 in increments of 10, and measure the latency distribution using HdrHistogram [14] on an AMD Ryzen 9 3900XT 12-Core Processor machine.

For SQLite, we used the `rusqlite` library that provides Rust bindings for SQLite. To simulate a multi-tenant serverless runtime, the microbenchmark creates a thread per tenant to represent a resident and executable function in the runtime. In each thread, we open a connection to a separate 1 MiB SQLite database file, and we execute the SQL query.

For *Limbo*, we use the same basic approach as for SQLite and perform the SQL query 1000 times, recording the latency distribution with HdrHistogram. However, as I/O is asynchronous, we use Rust coroutines instead of threads for tenants. Furthermore, we simulate an I/O loop by limiting the number of iterations one coroutine can run to ten and when a query returns an I/O status, we schedule another coroutine instead of blocking.

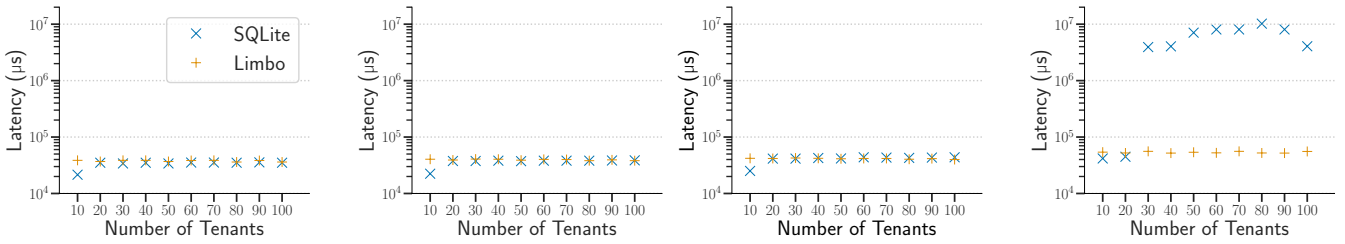
## 4.2 Results

In Figure 3 we present the median, $90^{th}$, $99^{th}$, and $99.9^{th}$ percentiles of the query latency for a given number of threads. For *SQLite*, we observe that the query latency does not degrade gracefully with the number of threads. Instead, we observe jumps in the latency percentiles when the number of threads is close to a multiple of the number of cores, which highlights that synchronous I/O limits multi-tenancy. For instance, in Figure 3(d) we observe jumps in the latency when the number of tenants increases from 10 to 20, from 20 to 30, from 40 to 50, and from 60 to 70; the jump is the latency when the number of tenants increases from 10 to 20 is seen in all the plots in Figure 3. We plan to investigate the reason for the fall in latency, observed in Figure 3(d), when the number of tenants increases beyond 80.

Our results show that the number of tenants that concurrently use SQLite may be limited by the number of processing cores, and increasing the number of tenants beyond the number of cores can increase the tail latency experienced by the tenants. In contrast, our preliminary evaluation for *Limbo* shows up to a 100x reduction in tail latency, suggesting that our approach is conducive to runtime/database co-design for low latency.

# 5 DISCUSSION AND RELATED WORK

**SQLite in serverless computing.** We are not the first to consider bringing SQLite to the serverless environments. Jonas et al. [9]




(a) Percentile = 50 (b) Percentile = 90 (c) Percentile = 99 (d) Percentile = 99.9

**Figure 3: Query latency (logarithmic scale).** *Each tenant executed* '`SELECT * FROM users LIMIT 100`' *1000 times. We log the latency for each query executed by each tenant. We observe that SQLite can limit multi-tenancy, and Limbo shows up to a 100x reduction in tail latency.*

explored integrating stateful databases with serverless computing, highlighting the mismatch between the ephemeral nature of serverless functions with a requirement for persistent storage, connection-oriented protocols, and reliance on the shared memory of traditional databases such as PostgreSQL, Oracle, and MySQL. They explored solving the issue by running the SQLite embedded database as part of a serverless runtime using an in-memory transactional caching layer to interpose between SQLite and a cloud provider network filesystem, overcoming the issues with traditional databases. The approach enabled them to achieve over 10 million transactions per minute but only 100 transactions per minute for writes on a modified TPC-C benchmark, which made the approach unpalatable as a general-purpose database.

Schleier-Smith et al. [11] explored using unmodified SQLite in the serverless environment by running it on top of a network filesystem specifically targeted for function as a service environment, FaasFS. Although their approach showed significant performance improvements over traditional network file systems (NFS), achieving nearly 30x higher transactions per minute in some configurations, it also revealed limitations in increased transaction abort rates due to concurrent modifications and the inefficiency of layering transactional systems. Their results suggest that SQLite on FaaSFS is a viable solution for read-heavy workloads.

Although our primary focus is on low latency, we believe we can improve the write throughput by co-designing the serverless runtime and database with massive multitenancy using asynchronous I/O, making SQLite a viable option for serverless computing.

**Moving data to code instead of code to data.** The problem with stateless serverless applications and data access, sometimes referred to as *data shipping problem* [16], is an open problem for serverless computing. The Shredder system solves the problem by allowing compute on storage nodes, essentially shipping code to data [16].

In contrast, we aim to address an opposite problem: shipping data to code. Our solution allows application logic to remain the same as we provide the illusion of the database being in the same memory space as the application, although I/O can still be external to the serverless runtime. To do that, we follow the same approach as Aurora to decouple storage and compute by decoupling query processor, transactions, and caching from logging and storage [15]. However, we virtualize logging and storage via an I/O module, which the serverless runtime implements in whatever way it sees fit. Our solution, *Limbo*, is built using `io_uring`, which assumes a volume, and our approach can be leveraged to offload storage to an external storage node for scalability. Furthermore, our approach allows the same architecture to be used outside serverless runtimes in, for example, web browsers.

**Benefits of asynchronous I/O.** Making database library query execution asynchronous, enables application to multitask while I/O is in progress. This opens up opportunities for serverless runtimes to efficiently implement multitenancy. Furthermore, from the SQL query engine perspective, asynchronous I/O also enables efficient remote I/O operations. For example, this approach allows a serverless architecture similar to Aurora where storage is offloaded to a remote server for scalability [15], but with the low latency of an embedded database when data is cached.

**Privacy and security.** Integrating a database in a serverless runtime introduces some new privacy and security challenges. Whereas we can secure a traditional database in a cloud environment by limiting the set of data a tenant can access, integrating the database to a runtime with security vulnerabilities makes much more data potentially accessible to malicious attackers. Therefore, co-designing databases and serverless runtimes requires mechanisms for ensuring data integrity and confidentiality. Serverless runtimes have building blocks to address this problem, and they already isolate tenants. For instance, they offer massive multi-tenancy especially in the edge deployments using lightweight isolation mechanisms such as V8 isolates. This isolation approach can be extended to data residing in the memory space of the runtime by leveraging memory enclaves or memory protection at the virtual machine layer, such as the JavaScript or WebAssembly JIT compiler or interpreter.

## 6 FUTURE WORK

**SQLite compatibility.** Compatibility with SQLite can be important for some applications to adopt our proposed solution. Our current prototype is a clean-slate implementation of SQLite, except for the SQL parser that we reuse from SQLite. A more compatible solution would be to reuse the SQLite front-end, including the code generator, and reimplement the backend starting from the virtual machine layer. However, our experience so far is that SQLite is only partially modular, which can make reusing the front end hard. Furthermore, the backend needs to propagate I/O status to `sqlite3_step()` API function for callers to take advantage of the

non-blocking interface, which may further complicate reuse. We, therefore, believe a clean-slate implementation may be needed. Compatibility with SQLite has to be achieved through a test suite that runs both on SQLite and Limbo.

**Evaluation methodology.** We have so far shown that moving to an asynchronous I/O model can improve the tail latency of SQL queries using a micro-benchmark. We plan to evaluate tail latency impact using more realistic workloads. We also plan to measure the impact on throughput. These evaluations will be done using the TPC-H benchmark suite, which consists of ad-hoc business oriented queries with concurrent data modifications, to measure both the throughput and latency.

Furthermore, in our micro-benchmark, there is only a single reader. We will also investigate what happens to tail latency and throughput when there are writers, and also when there are multiple readers. This evaluation will also be done to study the transaction isolation and transaction coordination. For example, we need transaction coordination when there are multiple tenants writing to the same database from a single instance of a serverless runtime, and we need distributed transaction coordination when there are writers from multiple runtime instances.

**Impact on serverless runtime architecture.** The stateless nature of serverless functions implies that each invocation of the function can be executed on any available instance within the provider's serverless runtime environment. Integrating an in-process database to the serverless runtime implies either ensuring the database is accessible from every runtime instance or the client requests are routed to run on a subset of the runtime instances. With serverless edge computing in particular, data locality is important for low latency. Therefore, we anticipate that for optimal integration of an in-process database, the serverless runtime architecture needs to address these issues.

## 7 CONCLUSION

Serverless runtimes need to be co-designed with databases to retain their latency advantage for edge-native applications. Embedding in-process databases such as SQLite into the serverless runtime eliminates the need for network round-trips, however SQLite's synchronous architecture can make the embedding counter-productive.

In this article, we posit changing the SQLite virtual machine to provide asynchronous bytecode instructions. We show that our modifications can help serverless runtimes towards facilitating database and serverless runtime co-design.

## REFERENCES


[1] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3317550.3321434

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[4] Cloudflare. 2023. Workers. https://workers.cloudflare.com.

[5] Cloudflare. 2024. Workers KV. https://developers.cloudflare.com/kv/.

[6] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* (2022).

[7] D. Richard Hipp. 2023. SQLite. https://www.sqlite.org/index.html.

[8] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[9] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf.

[10] Mahadev Satyanarayanan. 2023. Sinfonia: Cross-Tier Orchestration for Edge-Native Applications. https://www.youtube.com/watch?v=QQKjjf3veDA.

[11] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton, and Joseph M Hellerstein. 2020. A FaaS File System for Serverless Computing. *arXiv preprint arXiv:2009.09845* (2020).

[12] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 138–152. https://doi.org/10.1145/3472883.3486981

[13] SQLite. 2023. About SQLite. https://www.sqlite.org/about.html.

[14] Gil Tene. 2023. HdrHistogram: A High Dynamic Range Histogram. http://hdrhistogram.org/.

[15] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[16] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3357223.3362723