Towards Database and Serverless Runtime Co-Design

Pekka Enberg
University of Helsinki
Finland

Sasu Tarkoma
University of Helsinki
Finland

Ashwin Rao
University of Helsinki
Finland

ABSTRACT

In serverless computing, minimizing database access latency is crucial, but databases predominantly reside in cloud environments, necessitating 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 changing the SQLite virtual machine to provide asynchronous bytecode instructions to avoid blocking in the library and decoupling the query and storage engines to facilitate database and serverless runtime co-design.

ACM Reference Format:
Pekka Enberg, Sasu Tarkoma, and Ashwin Rao. 2023. Towards Database and Serverless Runtime Co-Design. In *Proceedings of the CoNEXT Student Workshop 2023 (CoNEXT-SW '23), December 8, 2023, Paris, France.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3630202.3630225

1 PROBLEM STATEMENT

With the emergence of serverless computing, the co-location of compute and data is becoming critical, especially with edge computing, where low latency is a critical requirement. Several in-memory key-value stores are available to address the low latency requirement. However, databases that support SQL are deployed in traditional, non-serverless cloud environments, partly because stateful workloads are challenging for serverless computing [5].

One practical approach to mitigate latency is multi-region support in databases, which 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. SQLite [3], an in-process database management system, has 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 [5], its architecture makes that challenging.

The SQLite architecture has two main parts: the core and the backend [2]. The core consists of the SQLite C API interface for applications, an SQL compiler, and a virtual machine for executing SQL queries. Execution of a SQL query when using SQLite begins with the `sqlite3_prepare()` function that transforms SQL statements such as SELECT into sequences of bytecode instructions. This bytecode is executed in the sqlite3_step() function that internally calls into the backend pager for traversing the database B-trees representing tables and rows. The pager caches the pages containing the relevant B-tree nodes, and performs the required disk I/O using the OS filesystem interface.

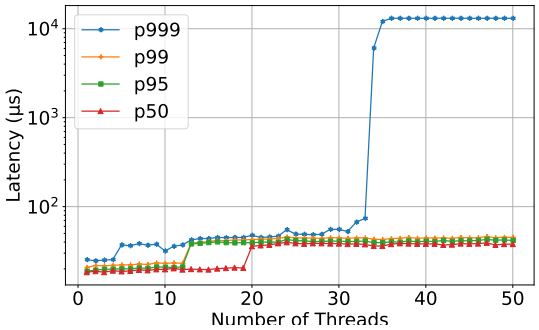

Figure 1: SQLite query latency (logarithmic scale). *Increasing the number of tenants (threads) beyond the number of cores significantly affects the tail latency.*

In contrast to SQLite, serverless applications are composed of functions that execute within a serverless runtime environment. Platform providers are vested in maximizing the density of functions per CPU and DRAM to achieve cost optimization. Consequently, maximizing the number of tenants running concurrently on a given physical machine is imperative to embedding SQLite into the serverless runtime. Furthermore, to maintain a high density of functions per DRAM, it is crucial to efficiently offload data storage operations to devices such as NVMe and SSDs while preserving low query latency. However, the `sqlite3_step()` function is inherently synchronous.

To highlight the impact of SQLite's synchronous architecture on multitenancy, we run a microbenchmark that creates a thread per tenant representing a resident and executable function in a serverless runtime. Each thread opens a connection to a separate, 1 MiB SQLite database file, and we increment the number of threads from 1 to 50. In each thread, we execute the query `SELECT * FROM user LIMIT 100` 1 million times and we measure the query execution time using HdrHistogram [7]. We ran our microbenchmark on an AMD Ryzen 9 3900XT 12-Core Processor machine. In Figure 1 we present the median, 90^{th}, 99^{th}, and 99.9^{th} percentiles of the query latency for a given number of threads. We observe that the query latency does not degrade gracefully with the number of threads when the number of threads exceeds the number of processing

CoNEXT-SW '23, December 8, 2023, Paris, France

ACM ISBN 979-8-4007-0452-9/23/12.
https://doi.org/10.1145/3630202.3630225

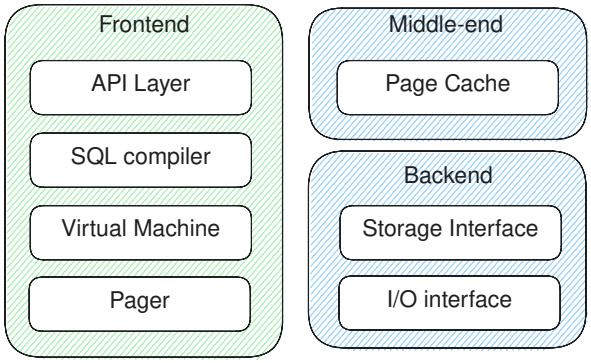

Figure 2: Proposed Database architecture.

cores. Instead, we observe jumps in the latency percentiles when the number of threads is close to a multiple of the number of cores.

Our preliminary results highlight that SQLite can limit multi-tenancy. Specifically, 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.

2 OUR SOLUTION

We propose the following two building blocks to integrate the database library into a serverless runtime.

- 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.

Asynchronous bytecode instructions. 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. For example, the `Next` instruction, which advances a cursor to point to the next available row, may need to read b-tree pages from the disk. We propose that the SQL compiler generates a pair of instructions `NextAsync` and `NextAwait`, where the first instruction submits I/O asynchronously and returns from the `sqlite3_step()` function and returns 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.

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

Figure 2 shows the proposed architecture that consists of a front end, a middle-end, and a back end. Unlike in SQLite's architecture, the front end includes the pager module. The backend part is a pluggable storage engine to provide flexibility for integrating with the serverless runtime. The middle-end contains the page cache. The frontend is either compiled into WebAssembly and runs as part of the application or is embedded into the runtime and exposed to applications separately. The backend is integrated into the serverless runtime or as a separate server process to facilitate elastic scaling independent of the compute layer. The middle-end can either run as part of the application, as part of the serverless runtime, or in a remote server, depending on the query latency and multitenancy trade-offs needed.

3 KEY RELATED WORKS

There are many workarounds to address the limitation of SQLite's architecture, and key examples include LiteFS [4] and LibSQL [6]. LiteFS is a user-space filesystem that intercepts writes to the SQLite database WAL for replicating SQLite databases in clusters. LibSQL [6] is a SQLite fork that uses a similar replication strategy but implements it using WAL extension hooks. In contrast, Cloudflare D1 [8] is a proprietary SQLite-based database integrated into the Cloudflare Workers [1] serverless runtime, which works by intercepting writes to the SQLite database file. Regardless, the synchronous architecture of SQLite is a fundamental limitation in these solutions.

4 SUMMARY

Asynchronous bytecode instructions and decoupling query and storage engine facilitates database and runtime co-design, which results in low-latency database access for serverless applications. Because the synchronous model is fundamental to SQLite, we propose to build a new database library prototype compatible with SQLite whose tail-latency increases gracefully (linearly) with the number of tenants. This approach paves the way to integrate general-purpose database systems into serverless computing environments.

REFERENCES

[1] Cloudflare. 2023. Workers. https://workers.cloudflare.com.
[2] 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).
[3] D. Richard Hipp. 2023. SQLite. https://www.sqlite.org/index.html.
[4] Ben Johnson. 2023. LiteFS. https://github.com/superfly/litefs.
[5] 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.
[6] libSQL. 2023. libSQL. https://github.com/libsql/libsql.
[7] Gil Tene. 2023. HdrHistogram: A High Dynamic Range Histogram. http://hdrhistogram.org/.
[8] Kenton Varda. 2023. D1 storage engine. https://x.com/KentonVarda/status/1659551757796515846?s=20.