Towards Database and Serverless Runtime Co-Design

Pekka Enberg, Sasu Tarkoma, Ashwin Rao
University of Helsinki
CoNEXT-SW 2023

Problem

- Serverless runtimes are stateless, but many applications need to manage state to be useful.

- Key-value stores are fast for keeping state, but restricted whereas traditional databases are but slow but more expressive.

- Embedded databases such as SQLite combine best of both worlds, but integrating with serverless runtime requires rethinking the architecture.

Outline

- Serverless & multitenancy
- SQLite architecture and limitations
- Our proposed solution
- Related work & discussions

Serverless & multitenancy

- Serverless runtime executes stateless functions, which typically take HTTP requests as input and output HTTP responses.

- Massive multitenancy is critical for serverless edge computing because service provider needs to pack lots of tenants on machines to reduce cost.

- For example, Cloudflare Workers uses V8 isolates instead of virtual machines or containers to fit 10 million customers on a dozen of machines [Varda, 2019].

Kenton Varda. 2019. Fine-Grained Sandboxing with V8 Isolates. Web page.

Example: Cloudflare Workers

The Cloudflare global network

Our vast global network, which is one of the fastest on the planet, is trusted by millions of web properties.

With direct connections to nearly every service provider and cloud provider, the Cloudflare network can reach about 95% of the world's population within approximately 50 ms.

"Deploy once, run everywhere"

Source: https://www.cloudflare.com/en-gb/network/

SQLite architecture

- SQLite is a SQL database engine library that applications can embed.

- The high-level interface is `sqlite3_prepare()` for preparing SQL statements and `sqlite3_step()` for executing them.

- SQL statements are transformed into bytecode programs that are executed by the SQLite virtual machine.

Kenton Varda. 2019. Fine-Grained Sandboxing with V8 Isolates. Web page.

SQLite architecture

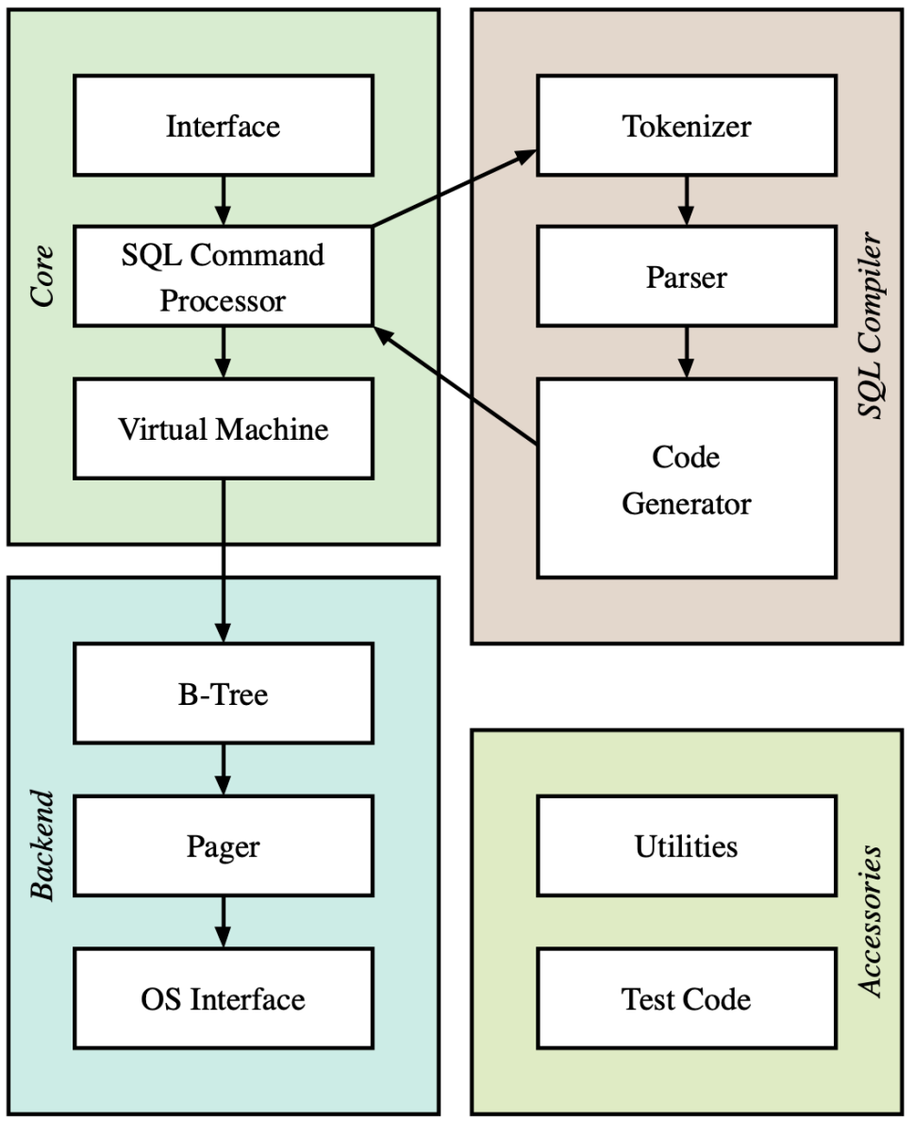


Figure 1: Architecture of SQLite

Table 1: Bytecode program for SSB Q1.1.

Address	Opcode	P1	P2	P3	P4	P5
0	Init	1	23	0		00
1	Null	0	1	3		00
2	OpenRead	0	7	0	12	00
3	OpenRead	1	6	0	5	00
4	Rewind	0	19	0		00
5	Column	0	11	4		00
6	Lt	6	18	4	BINARY-8	54
7	Gt	7	18	4	BINARY-8	54
8	Column	0	8	4		00
9	Ge	8	18	4	BINARY-8	54
10	Column	0	5	9		00
11	SeekRowid	1	18	9		00
12	Column	1	4	4		00
13	Ne	10	18	4	BINARY-8	54
14	Column	0	9	5		00
15	Column	0	11	11		00
16	Multiply	11	5	4		00
17	AggStep1	0	4	1	sum(1)	01
18	Next	0	5	0		01
19	AggFinal	1	1	0	sum(1)	00
20	Copy	1	12	0		00
21	ResultRow	12	1	0		00
22	Halt	0	0	0		00
23	Transaction	0	0	6	0	01

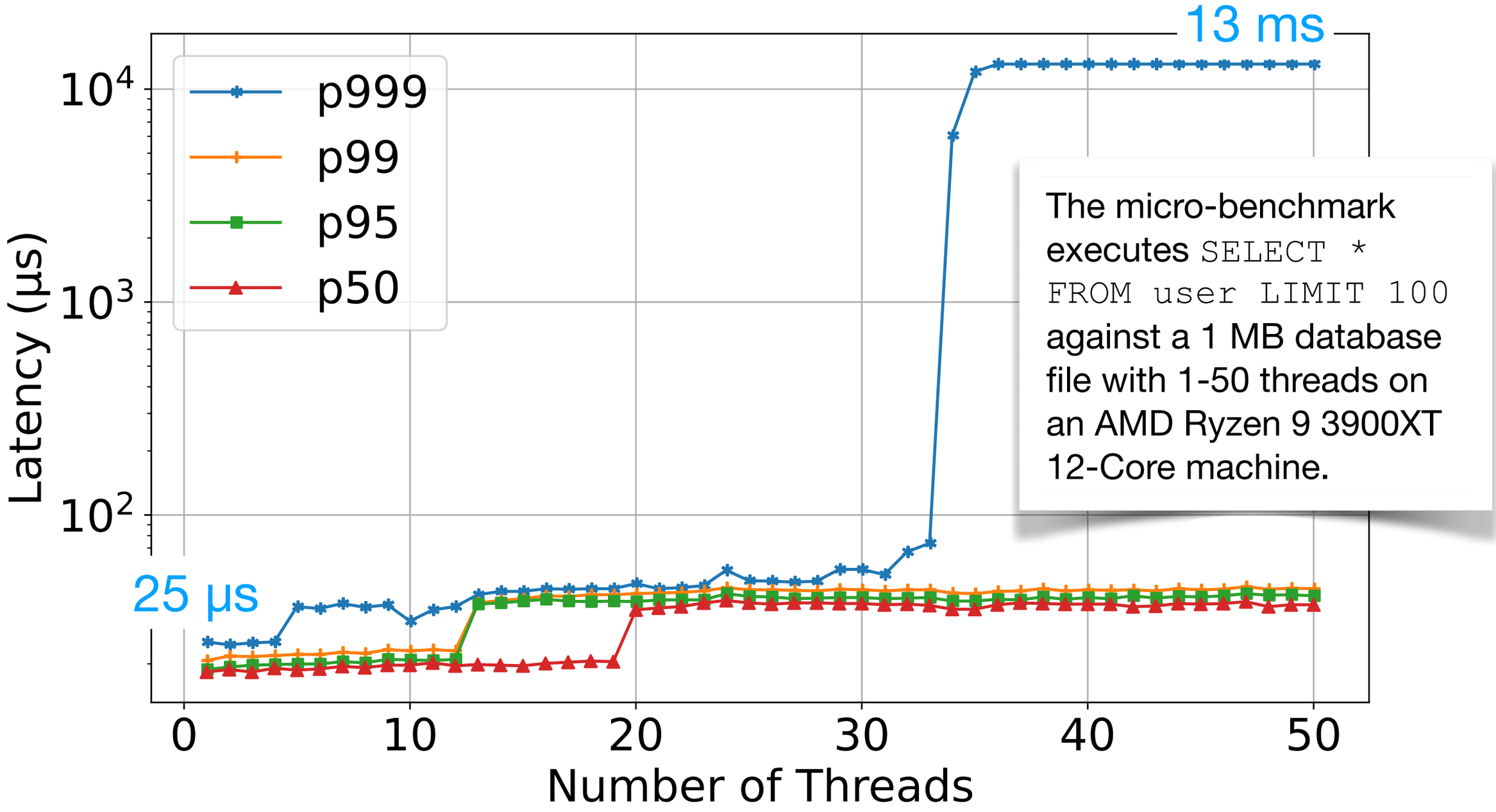
Kevin P. Gaffney *et al*. 2022. **SQLite: Past, Present, and Future**. *In* VLDB '22.

SQLite limitations

- **SQLite virtual machine is synchronous**, which means performing I/O blocks the thread.
    - However, the `sqlite3_step()` function could return a `SQLITE_IO` status code.

- **SQLite is filesystem centric**, which limits deployment options.

SQLite query latency distribution



Our solution

- Asynchronous bytecode instructions

- Decouple compute and storage

Our solution: Asynchronous bytecode instructions

- SQLite's virtual machine bytecode instructions that perform I/O are blocking.

- To support asynchronous I/O, let's introduce instructions that don't block using async/await.

Our solution: Asynchronous bytecode instructions

```
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
```

Our solution: Asynchronous bytecode instructions

```
> 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     Column          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
```

Our solution: decoupling query and storage

- We want the query engine integrated in the serverless runtime, but we can't necessarily host all databases on the serverless node because of storage constraints.

Related work: special purpose filesystems

- In-memory transactional caching buffering layer between SQLite and the cloud provider's network filesystem achieves 10M transactions per minute (tpm) reads, but only 100 tpm writes [Jonas *et al.*, 2019].

- FaaS filesystem although 30x faster than NFS, achieves 10k tpm [Schleier-Smith *et al.*, 2020].

- Filesystem is not the right abstraction for serverless?

Eric Jonas *et al*. 2019. **Cloud Programming Simplified: A Berkeley View on Serverless Computing**. Tech report.

Johann Schleier-Smith *et al.* 2020. **A FaaS File System for Serverless Computing.** Tech report.

Related work: SQLite logical logging

- Logical logging to reduce the write amplification of WAL in SQLite, which is a pain-point for mobile applications [Park *et al.*, 2017].

Jong-Hyeok Park *et al*. 2017. **SQL Statement Logging for Making SQLite Truly Lite**. *In* VLDB '17.

Related work: compute and storage decoupling

- Architecture for decoupling compute and storage, and backing up to S3 [Verbitski *et al.*, 2017].

Alexandre Verbitski *et al*. 2017. **Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases**. *In* SIGMOD '17.

Discussion

- Can multiple serverless runtimes access the same database? Or will we need smart request routing + sharding?
- Where does the page cache live? In the guest (the application) or the host (the runtime)?
- What is the strongest transaction isolation level we can guarantee? How does recovery work?
- Can we leverage compute offload on NICs, storage devices, etc.?