LumoSQL First Results

Dan Shearer, Claudio Calvelli, Ruben De Smet

18th February 2022

Abstract

This is a limited-circulation summary of initial benchmarking results from the LumoSQL project, intended for authors with commit access to trees related to SQLite and LMDB, or others with intimate interest in these trees. We present data from running multi-way comparisons of various databases running on different hardware, as recorded/detected by the benchmarking code. We do not believe there is any existing comprehensive benchmarking system for SQLite or LMDB. We have provided all our data, and given instructions for either replicating or refuting our results as may be. If the SQLite and/or LMDB teams wish to make any code changes in response to these results we will be very pleased to include those changes in our benchmarking runs before releasing another version of this document. For those few readers who do not know of us already, further details about the LumoSQL authors are easily found online. We are *really* pleased… this is a clusterable, repeatable, somewhat-concurrent framework for generating test data that can be used collaboratively by people who do not know each other, and yet which conveys meaningful information about their disparate environments.

Contents

About this document 3

Highlights from our results 4

LumoSQL context and expectations 4

Replicating benchmarking results from our data 5
- Schema of benchmarking data . . . 6
- Trying it out . . . 6

DATASIZE=r,w option and implications 7

Replicating results ab initio 8
- Refining your benchmarking . . . 8

Limitations 8

What's next 9

About this document

This limited-circulation document is © the LumoSQL Authors 2022. Version 0.6.

Responses are welcome via any or all of:

- Contacting Dan Shearer on dan@shearer.org or +44 74 13 16 85 28
- Dropping in to #lumosql on Libera irc chat
- Requesting commit access to the LumoSQL Fossil repository to fix our bugs or misunderstandings

Highlights from our results

- SQLite becomes increasingly slow as storage hardware gets slower, but LMDB seems to remain constant. LMDB is very fast on a ramdisk.
- SQLite (compared only against other released versions of SQLite) has had some performance regressions across some releases.
- LMDB has had consistent performance regressions across some releases, although less so recently.
- For read operations such as reading an index, SQLite is twice as slow for twice the number of reads while LMDB is about four times as slow.
- As data size increases, LMDB write performance degrades faster than SQLite's write performance.
- A point problem: LMDB has a serious write performance issue with LumoSQL test 8, which is slower than SQLite and the time to run grows rather more than linearly with the number of SELECT operations (while the time to run the same with SQLite grows linearly, as one would expect). We do not know why.

LumoSQL context and expectations

This document is about performance of the main components of LumoSQL. We present the first detailed demonstration that LumoSQL's work over the last two years is repeatable. Originally about updating and reproducing Howard Chu's SQLightning project, in 2022 LumoSQL is an experimental testbed for asking questions about SQLite and LMDB that have not previously been easy or possible to ask. LumoSQL is not a fork of either SQLite or LMDB, thanks to the Not Forking system.

To set expectations, this document relates to the LumoSQL build tree as of Febrary 2022, which:

- will build without error on many versions of Linux, NetBSD and FreeBSD, with minimal dependencies in addition to those already required by SQLite and LMDB.
- will modify any version of LMDB since 2016 to be the storage backend for any version of SQLite since 2018. This work isn't finished, but it is quite functional today and we have no question it can be finished.
- has a generalised build system designed to cope with many more variables than we have tried or even thought about: backends, hardware/OS architectures, and modifications up and down the SQLite stack.
- has a generalised benchmarking system that currently handles more than 10 dimensions without being difficult to drive, stores its data in a simple and extensible SQLite database, and most importantly is intended to be used collaboratively.

The inspiration here is not just about the SQLightning experiment. Among some dozens we have found, Bloomberg LLP's Combdb2 and Expensify's Bedrock are projects that integrate with SQLite source in very different ways. Comdb2 continues to develop a pre-Oracle version of BDB, which is interesting in that LumoSQL includes a post-Oracle BDB for comparison purposes.

You will *not* find here:

- A fully-functional SQLite+LMDB, eg suitable for compiling into Fossil, Firefox, Nix or any other of thousands of apps. That is our goal, but first we need a general solution for some technical problems including long index keys. These are likely part of a more general backend API, which involves more work. We welcome design contributions.
- An SQLite+LMDB which will run SQLite's testsuite. The tests we supply are much more limited and specific, and we would love a hand to go through this and squash problems as we finish off the LMDB port.
- Any-on-any SQLite and LMDB versions. We could carry patches for more versions due to interface definitions changing, but the main point is amply proved and we are focussed elsewhere.
- A tree that builds on, or has cross-compiling support for, targets such as Android, iOS, Windows, MacOS etc. These targets may not be technically difficult given that SQLite runs natively on them already, but they have been excluded from the scope so far.

Things we are *not* discussing in this document, but which are important parts of the LumoSQL discussion to be covered another time:

- Details of interfacing SQLite to a backend. Richard Hipp has said he is open to the idea of a general API to the SQLite key-value store.
- Improved mechanism for triggering functions (such as a checksum) per SQLite database row.
- The Lumions RFC which is where our work on checksums, encryption, signing etc comes together. If each row in a database is a Lumion, some things will be slower but some will be faster, and everything will be of known integrity.
- Benchmarking all of the above.
- Detailed performance envelope for SQLite. We know we *can* do this, including incorporating the work in the existing SQLite public test suite, plus other likely SQL. However there are people on the distribution list for this document who are very good at designing tricky and demanding SQL and we'd be very pleased to hear from you.

Replicating benchmarking results from our data

There are 4 minimum requirements for you to display the data we have collected from our benchmark runs:

- tclsh in the path
- sqlite3 in the path

- The Tcl script https://lumosql.org/src/lumosql/file?name=tool/benchmark-filter.tcl
- The data: wget https://lumosql.org/dist/benchmarks-to-date/all-lumosql-benchmark-data-combined.sqlite

Schema of benchmarking data

There are two tables: `run_data` for information about the setup of a particular run (hardware, versions of software being tested, etc) and `test_data`, for the timing or other results from the run.

Every benchmarking run in `run_data` has an SHA3 runid allocated, stored as (run_id, key, value) tuples.

```
CREATE TABLE run_data (
        run_id VARCHAR(128),
        key VARCHAR(256),
        value TEXT
    );
CREATE UNIQUE INDEX run_data_index ON run_data (run_id, key);
CREATE TABLE test_data (
        run_id VARCHAR(128),
        test_number INTEGER,
        key VARCHAR(256),
        value TEXT
    );
CREATE UNIQUE INDEX test_data_index_1 ON test_data (run_id, test_number, key);
CREATE UNIQUE INDEX test_data_index_2 ON test_data (run_id, key, test_number);
```

Trying it out

Here are some commands to get a feel for benchmark-filter.tcl:

```
tclsh benchmark-filter.tcl -help
# The following command tells us there are 8140 tests in this data file
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -count
# This one shows the test result from the most recent 20 runs, with 20
# rows of data, one test timing per column
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -summary -column test
# We can select for hardware types, for example two types of autodetected disk:
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -column test -disk %ramdisk%
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -column test -disk %wdc%
# Or for example an autodetected processor:
```

```
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -column test -cpu %ryzen%
# The same, but showing one column per run, one row per test, with more
# details about runs, but also getting unreadable if showing more runs)
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -cpu %ryzen%
# This compares SQLite against other versions of itself on a particular
# hardware combination (-no-backend means only to show an unmodified SQLite,
# as opposed to one with an alternative storage backend provided)
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -column test -no-backend -datasize 1 -cpu %ryzen% -disk '%ssd%'
# This compares one version of SQLite with all versions of LMDB on the same
# hardware combination (-no-backend shows unmodified SQLite, and
# -backend lmdb adds to that runs with LMDB as alternative storage backend):
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -column test -version 3.37.2 -datasize 1 -no-backend -backend lmdb \
      -cpu %ryzen% -disk '%ssd%'
# This compares all versions of SQLite with one version of LMDB again on
# the same hardware combination:
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -column test -datasize 1 -backend lmdb-0.9.29 -no-backend \
      -cpu %ryzen% -disk %ssd% -limit 0
# This compares the last 200 benchmarks we ran, _unreadably_:
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -ignore-numbers -summary -limit 200
# same, but swapping rows and columns for an easier to read output:
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -ignore-numbers -summary -limit 200 -column test
# And finally, for humour value, this compares native SQLite with the copy
# of SQLite 3.18.2 included as "SQL option" in BDB 18.1.32 (we filter by
# -version 3.18.2 which selects both the native and the modified SQLite):
tclsh benchmark-filter.tcl -db all-lumosql-benchmark-data-combined.sqlite \
      -version 3.18.2 -quick
```

DATASIZE=r,w option and implications

These are some of the more interesting results.

Most tests include a number of SQL statements executed in sequence; `DATASIZE=r,w` allows to multiply the number of statements which read data by `r` and the number of statement which write or update data by `w` (if `r` = `w`, we use `DATASIZE=r` as an abbreviation, and this is how it is displayed by `benchmark-filter.tcl`)

Note: when results include different data sizes, the test names will differ so it refuses to show them all in one combined output; however the `-ignore-numbers` option replaces all numbers in the test names by a single `#`, so they can be shown side by side.

(show some examples of comparing for example 1,2 2,1 and 2 and comment on this, using -ignore-numbers)

Replicating results ab initio

We expect quite a few of you might want to see these results for yourself, on your systems. Our install instructions are fairly well tested, after which you should be able to continue to the next section, Quickstart Build and Benchmarking. This is about generating benchmark run data for *your* system. You will then be able to run all the `benchmark-filter.tcl` commands listed earlier in this document on *your* benchmarks.sqlite file.

Refining your benchmarking

- Add runs to a combined file, detecting duplicates: `sh tool/add-results-to-combined tool/benchmark-filter.tcl combined.sqlite *.sqlite`
- The -completed status on benchmark-filter.tcl is about selecting runs which ran successfully to the end
- If you are suspicious about corruption: `for n in *.sqlite; do echo "$n"; echo 'pragma integrity_check;' | sqlite3 -readonly "$n"; done`
- To see what the benchmarking will guess for DISK_COMMENT and CPU_COMMENT: `tclsh tool/hardware-detect.tcl $PATH-THAT-WOULD-BE-VALID` and `tclsh tool/hardware-detect.tcl`, respectively. PATH-THAT-WOULD-BE-VALID means to detect the disk that would contain the supplied path, whether or not it exists at the moment. Sometimes there will be a null response, but that is because we can't guess on that system yet, and is not an error. Do please report any detection tips for us to add. We are trying to avoid a dependency on Tclx, which not all systems have and which is not in any case so brilliant at hardware detection.
- We have some hints for running benchmarking on a cluster
- Do please send us your benchmarking sqlite files with a suitable filename, so we can add them to our growing collection

Limitations

- The LumoSQL tooling works with whole, human-readable version numbers rather than SCM commit hashes. It would be good to add support for SCM hashes.
- Some of the benchmark runs were done before all the fields were fully populated. Therefore, some comparisons have less source data available than others (eg the disk media detection is relatively new.)

- Nearly all benchmarking has been done on IA-64 architectures, with some on ARM32.
- All testing so far has been conducted by the three authors. The LumoSQL benchmarking system is intended to be used by anyone and to have output databases easily mergable, however we have not yet had external contributions.
- The SQL queries in benchmark-filter are not optimal, and we expect SQL specialists will be able to improve them. This won't change the benchmark results but will change how quickly we can get summary results.

What's next

- Incorporate feedback from the good recipients of this document!
- Use R and various graphs to supplement the output of benchmark-filter.tcl. This is being worked on now.
- Upload facility so (a) trusted people can upload benchmarks and (b) untrusted people can upload benchmarks, with suitable checks and suspicions.
- Carry on with improving LumoSQL and the privacy/security enhancements enabled by Lumions.

ENDS