sqlite-zstd: Transparent dictionary-based row-level compression for SQLite

An sqlite extension written in Rust to reduce the database size without losing functionality

JUL 31, 2022

repo github.com/phiresky/sqlite-zstd

Motivation (or "side side projects")

While working on my startup (DishDetective) I started a side project of an automatic time-tracking tool (*timetrackrs*), where I try to collect a lot of information about my habits for later analysis. The main component collects data from my computer about what programs are open every 30 seconds, but another component for example imports usage data from my phone (via *App Usage*).

I store all this data as "events" in an SQLite database, but each event is pretty redundant - for example the open program doesn't change that much. But I still want to store detailed, raw snapshots with the aggregation happening later so I can change the parameters of the analysis without losing any information.

The main events-log SQLite table looks somewhat like this:

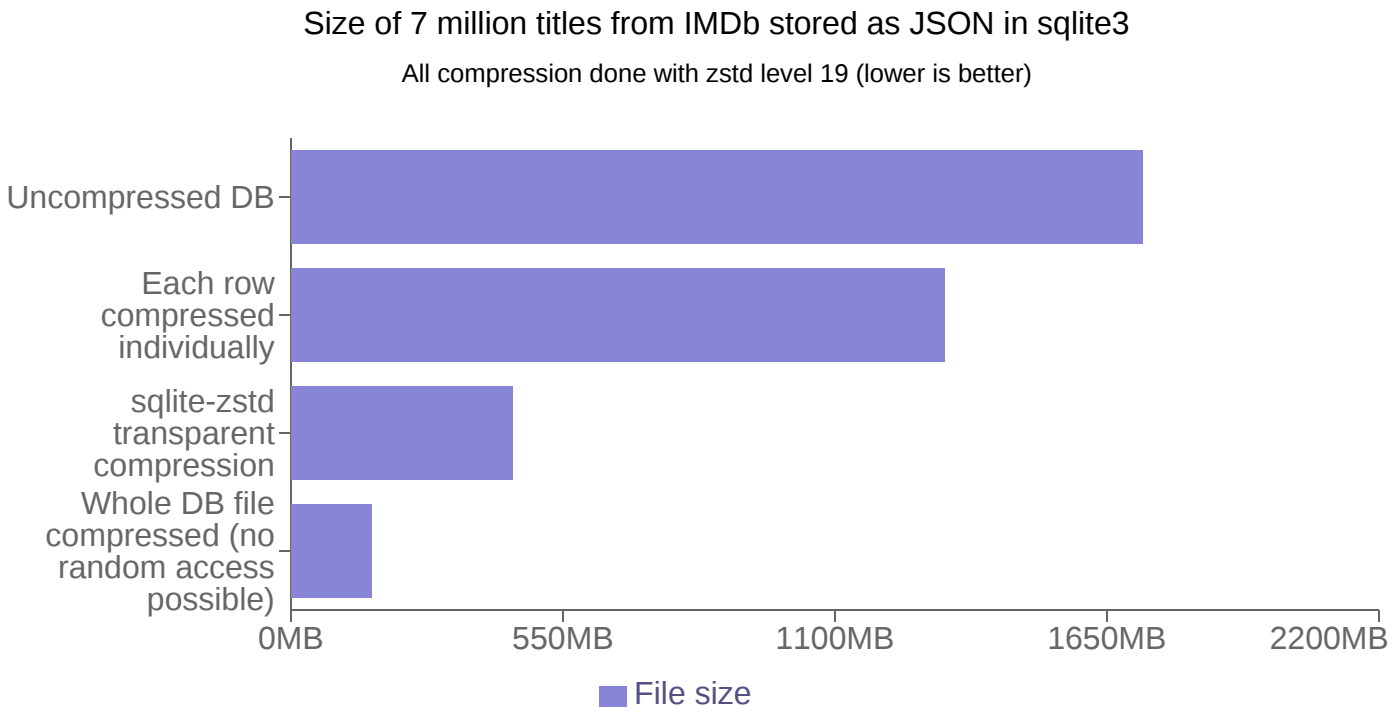
id	timestamp_unix_ms	data_type	data
52140	1616108552527	app_usage_import_v1	`{"device_type":"Smartphone","app":{"pkg_name":"org.telegram.messenger", "app_name":"Telegram", "app_type":3}, "act_type":200, "act_type_flag":0, "pid":35,...}`
52141	1616108009206	x11_v2	(30kByte json) `{"current_desktop_id": 9, "focused_program": 5, "open_programs": [...}`

id	timestamp_unix_ms	data_type	data
52142	1616107978795	x11_v2	(30kByte json) `{"current_desktop_id": 2, "focused_program": 17, "open_programs": [...}`

`data` is a json blob from the perspective of SQLite, but it is parsed into a strictly typed structure in the Rust application code.

After a year of collecting data, the SQLite database is already **7.6 GByte** in size.

Of course this is an extreme example of redundancy, but storing similar redundant data in databases is pretty common. SQLite doesn't have any compression features, so of course I had to start a *side-side-project* and build my own. Here's a teaser of the results:



Compressing the whole database file reduces the size by *90%*, but that way the data isn't actually queryable. Compressing the data in the application code before storing it in the database reduces the size by only *23%*.

My sqlite extension reduces the size by *75%* without modification to the application code and while retaining all normal querying capabilities.

Other options for compressing data in a database

There's a few solutions I could think of when you have compressible data in your database:

1. Normalize your data as much as possible, bringing it into a strict normal form. In theory, my above table could be split into dozens of smaller tables that each have no redundancy and perfectly clean relations with each other. This is how you traditionally "should" use a relational database.

My data is mostly just compressible because of the way its stored. JSON stores all the same keys every time, and many adjacent events have similar contents. All this could be fixed by normalizing the data. I think the same would apply for a lot of use cases of compressible data in databases such as event logs / analytics. But there's a reason ~~NoSQL~~ NoRelational databases have become popular:

- I'd have to duplicate all the types and data structure I already have in my application code in SQL code for little benefit - Here I always just want to fetch the whole data of one event, and not some smaller parts filtered by some criteria. I also don't want to bother with complex join queries just to get the same information every time.
- My schema would become very inflexible. Whenever I add more options or convert some types I'd have to write SQL migrations for it and think about the best way to store the new data.
- For extensibility, I might not actually know the exact structure of the data before storing it, for example if it is imported from an external tool.

I personally prefer to have a mix of a somewhat simple relational structure in the database (where it makes sense) together with denormalized data stored in a json column. This gives me the best mix of flexibility and structure. The popularity of the JSON column type in PostgreSQL shows that I'm not the only one. It's has a lot of functionality, for example you can query by individual members of the JSON with indexes on expressions. You could maybe even reimplement MongoDB using just the PostgreSQL JSON column type.

2. Just store the data compressed individually. On insert, `compress(data)`, on every select decompress it.

   This requires some application code change but is easy to implement and easy to use. But it's also not that effective, especially if your column contains JSON data since JSON stores the dictionary keys together with the values. Even with compression, these keys will still needs to be stored in every row.

   PostgreSQL does something similar for large blobs with its TOAST storage.

3. Split the database into separate files / partitions (for example weekly), then compress the less used partitions.

   If you need to access older data, simply decompress the whole chunk of data to RAM, then read from that database. This would work okay for my use case since it's pretty much append-only time-series data; older data is not read often, and when it is it is read pretty sequentially.

   In SQLite you can attach multiple database files into a single instance using `ATTACH DATABASE`, and then join across tables of the different files. In PostgreSQL you have table partitioning and tablespaces. You could also put the older partitions on a slower storage device like an HDD with file system level compression enabled.

4. Store the data in a column-oriented fashion. That way you can compress blocks of data much better. This is done in many database systems used for larger scale time-series data with less

need for a complex schema, such as InfluxDB or ClickHouse.

5. And finally: **Do some weird mix of the above**. My solution falls into this category. Another example of a weird mix would be timescaledb, which from what I understand does compression by converting data to a semi column-oriented format in chunks, then storing those chunks inside single rows in the row-oriented PostgreSQL table.

Introducing sqlite-zstd

My approach here is kind of a mix of partitioning and individual row compression with some of the benefits of column-oriented storage. I don't know of this method being used anywhere else (please let me know), so it might be interesting to other database systems as well. The whole thing should work just as well for e.g. PostgreSQL.

The idea is to define a way to split the rows of the table into chunks, then train a `zstd` dictionary for each chunk and compress each row in the chunk with it.

zstd is pretty much the state of the art for many kinds of data compression. The "training" feature works by taking a set of data samples and finding the most useful common strings in these samples. The resulting file is called a "dictionary" and can be used to compress other values similar to the seen samples to even smaller sizes.

In sqlite-zstd it's used like this: When selecting a row, the dictionary is loaded first and the data is decompressed with the dictionary. To improve performance, the least recently used dictionaries are kept instantiated, and the compression / chunking happens in a lazy fashion when the database is not busy by using the existing rows in the database for the training.

The whole thing is implemented by replacing the table with an updatable view so most of the application code doesn't need any changes.

How to use it

First, you take your database and load SQLite with the extension:

```
$ ls -lh imdb.sqlite3
-rw-r--r-- 2.0G imdb.sqlite3
$ sqlite3 -header imdb.sqlite3 'select * from title_basic limit 1'
id|data
 1|{"tconst":"tt0000001","titleType":"short",
   "primaryTitle":"Carmencita","originalTitle":"Carmencita",
   "isAdult":0,"startYear":"1894","endYear":null,
   "runtimeMinutes":"1","genres":["Documentary", "Short"]}
```

Then you enable transparent compression:

```
$ sqlite3 -header imdb.sqlite3 -cmd ".load libsqlite_zstd.so"
select zstd_enable_transparent('{"table": "events",
    "column": "data", "compression_level": 19,
    "dict_chooser": "''i.'' || (id / 1000000)"}');
select zstd_incremental_maintenance(null, 1);
vacuum; -- clean up space
```

And your database is smaller!

```
$ ls -lh imdb.sqlite3
-rw-r--r-- 528M imdb.sqlite3
```

While querying still works just fine:

```
$ sqlite3 -header imdb.sqlite3 -cmd '.load libsqlite_zstd.so' \
    'select * from title_basic limit 1'
id|data
 1|{"tconst":"tt0000001","titleType":"short",
   "primaryTitle":"Carmencita","originalTitle":"Carmencita",
   "isAdult":0,"startYear":"1894","endYear":null,
   "runtimeMinutes":"1","genres":["Documentary", "Short"]}
```

How it works

The `zstd_enable_transparent(config)` converts the table into a view. `config.dict_chooser` is an SQL expression that decides how to partition the data. Example partitioning keys:

Use case	Partitioning SQL expression	Explanation
Generic data	`rowid/100000`	Compress every 100k inserted rows with the same dictionary
Generic time-series data	`strftime(created, '%Y-%m')`	Compress every month of data together
A limited amount of different data formats	`source || '.' || strftime(created, 'weekday 0')`	Compress every `source` separately per week (useful if they have very different structure)
A write-once, read-often database	`'a'`	all the rows are compressed with the same dictionary

For real world use cases you might want a more complex key, for example one that keeps the "hot set" of data (e.g. the current week) uncompressed.

Internally, this will do the following:

1. It validates the given configuration including the dict_chooser expression, then stores it in a table `_zstd_config`
2. It renames the table and adds a dictionary_id column somewhat like this:

```
alter table events rename to _events_zstd;
alter table _events_zstd add column _data_dict integer
    default null references _zstd_dicts(id);
create index on _events_zstd (_data_dict);
```

3. It creates a view to replace the original table somewhat like this:

```
create view events as
    select id, col2, ...,
        zstd_decompress_col(data, 1, _data_dict, true) as data
    from _events_zstd;

create trigger events_insert_trigger
    instead of insert on events
    for each row
    begin
        insert into _events_zstd(id, ..., _data_dict, data) select
            new.id,
            ...,
            null as _data_dict,
            new.data;
    end;
create trigger events_update_trigger ...;
create trigger events_delete_trigger ...;
```

   The view calls the `zstd_decompress_col()` function, which decompresses the the given data with a dictionary stored in the _zstd_dicts table. The triggers basically just forward the UPDATE / INSERT queries to the backing table, addingt he _data_dict column as NULL (meaning uncompressed).

The `zstd_enable_incremental` function doesn't actually compress anything though, it just sets everything up.

To actually compress the data, you start the compression while your database is not too busy:

```
select zstd_incremental_maintenance(60, 1);
```

The first parameter is the maximum duration of the maintenance in seconds, the second is the maximum average database load (0 to 1).

The maintenance does the following:

1. Check all tables with enabled compression for data that is not compressed (_data_dict is NULL)
2. For every found row, group the data by the chosen dictionary key, then for every chosen dictionary key:
    1. If the dictionary with this key doesn't exist yet, train a new dictionary based on a sample of the data and insert it into `_zstd_dicts`.
    2. Split the rows into a chunk with a heuristic so each chunk takes will take less than one second to process.
    3. For each chunk, compress the data in the chunk in one transaction. This ensures the function can be interrupted / killed at any time.
    4. If the given database load is reached, wait. If the given time limit is reached, stop. This ensures the database stays functional during the maintenance.

Data size benchmarks

For the example above as well as my use case with timetrackrs, this extension is pretty effective. On GitHub someone mentioned they were using it for a private dataset within an Android app to great success:

> *I can't say it in detail, but I have a 800M database, after using this extension, the database's size shrink to 72M. That's enough for us to find a way to using this extension on mobile device. (source)*

It's kind of hard to give any guarantees because how well it works depends on many factors. Please let me know if you have another use case or statistic ;)

Why it works well

There's a few design choices I made to make this transparent compression work well:

- I'm making the compressed data as small as zstd allows by skipping the magic header (saves 4 bytes), skipping the checksum (saves 4 bytes), and not storing the data size.
- Every row knows which dictionary it was compressed with. This reduces the efficiency of the storage, but it also means that the whole thing is very robust: You can swap out the dictionary chooser / configuration however you want and incrementally (re)compress the data or change the dictionaries. The overhead is not too bad, because SQLite internally does some optimizations to store small integers very compactly.
- Newly inserted/updated data is not compressed. zstd has very fast decompression but somewhat slow compression. This way the performance (theoretically, see below) of INSERT and UPDATE queries stays the same. Compression can then happen outside of transactions and during low DB load times.
- You can exclude your "hot set" of data from compression to reduce the performance impact further. You just need to return null from the dict_chooser expression, see this example.
- You can compress any number of tables and columns in one table. They can use the same dictionaries or separate dictionaries.
- The used dictionaries are cached. zstd needs some time to parse dictionaries, so the instantiated version is kept in memory. After the first row is selected with one dictionary, the next rows decompress very fast.
- The `incremental_maintenance()` function is made for parallel operation. It doesn't affect read operations at all (with WAL mode enabled). You can choose for how much time it runs, as well as make it run for only part of the time to allow other write operations to take place. It does the compression in chunks, so it can be interrupted at any time without losing much progress, while still keeping the overhead low (the chunk size is estimated so each chunk always takes around 0.5 seconds).
- The dictionary size is configurable. By default the dictionary is limited to 1% of the total data size (e.g. with 1GB of data the dictionary will be 10MB). The training can use only a sample of all rows. By default it uses the recommended 100x dictionary size amount of data (e.g. to train a 1MB dictionary it will sample the rows to get a total of 100MB of data).
- Did I mention it's written in Rust?

Why it doesn't work well

There's also a few limitations and reasons why sqlite-zstd doesn't work as well as it maybe could.

- Zstd is not optimized for tiny data. In theory if we have an enum-like column in our database (e.g. all rows have one of the values `in_progress`, `cancelled`, `success`) then the dictionary should contain these three values and the resulting data should be a single byte, creating a

compression ratio of around 7x. As is though, the compression only really works if the average row size is at least 15-20 bytes.

- Dictionary-based compression doesn't really help for huge data. If each row has more than maybe 30-50kB of data, then usually the dictionary does not improve the compression much. The reason is that the redundancy within one sample makes the redundancy between multiple samples less important. You can still use this extension to get the transparent-compression goodness though with a dict_chooser of `[nodict]` though.

- Indexes into compressed data are not really supported (yet).

  In theory you can index values within a JSON data structure using

  ```
  CREATE INDEX startyearidx on _title_basics_zstd(
      zstd_decompress_col(data, 1, _data_dict, true)->>'startYear'
  );
  ```

  And this index is used fine when querying _title_basics_zstd directly. But when querying the `data` column in the `title_basics` view, the index is not used:

  ```
  sqlite> explain query plan select * from title_basics
              where data->>'startYear' == '1992' limit 1;
  QUERY PLAN
  `--SCAN _title_basics_zstd
  ```

  This sadly seems to be a limitation of SQLite of not combining expressions when figuring out index usability, even though it does pass down other normal indexable WHEREs. There might be some solutions using `GENERATED` columns instead of views.

  Right now, the workaround would be to store the columns you want to index in real columns.

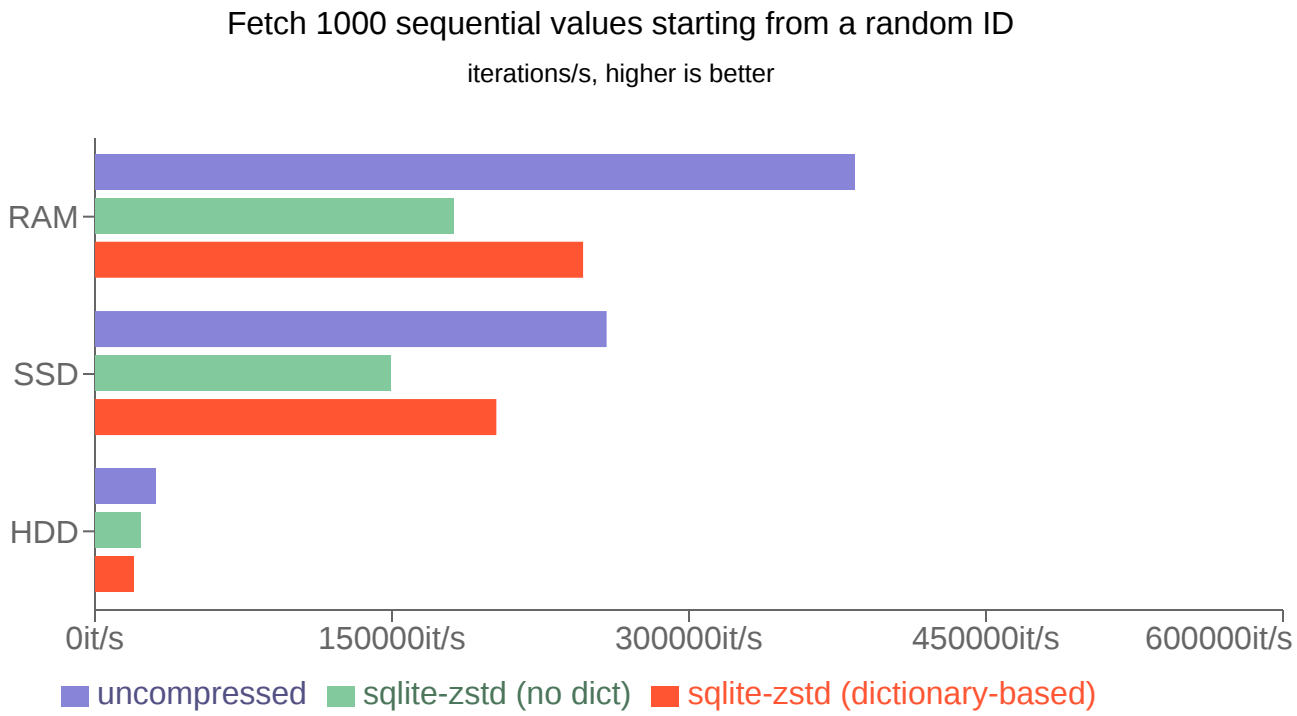
Performance Benchmarks

Setup

For testing (including the teaser chart at the beginning), I used the imdb `title.basics.tsv` file https://datasets.imdbws.com/title.basics.tsv.gz imported such that all the info is stored in a single JSON column. The database has 9 million rows and the average length of the JSON string is 215 Bytes. note.

The Rust code to create the test databases and run the benchmark is here. Before every benchmark, the operating system cache is cleared. I ran the benchmarks on Arch Linux with Ryzen 3900X CPU.

Selecting data sequentially

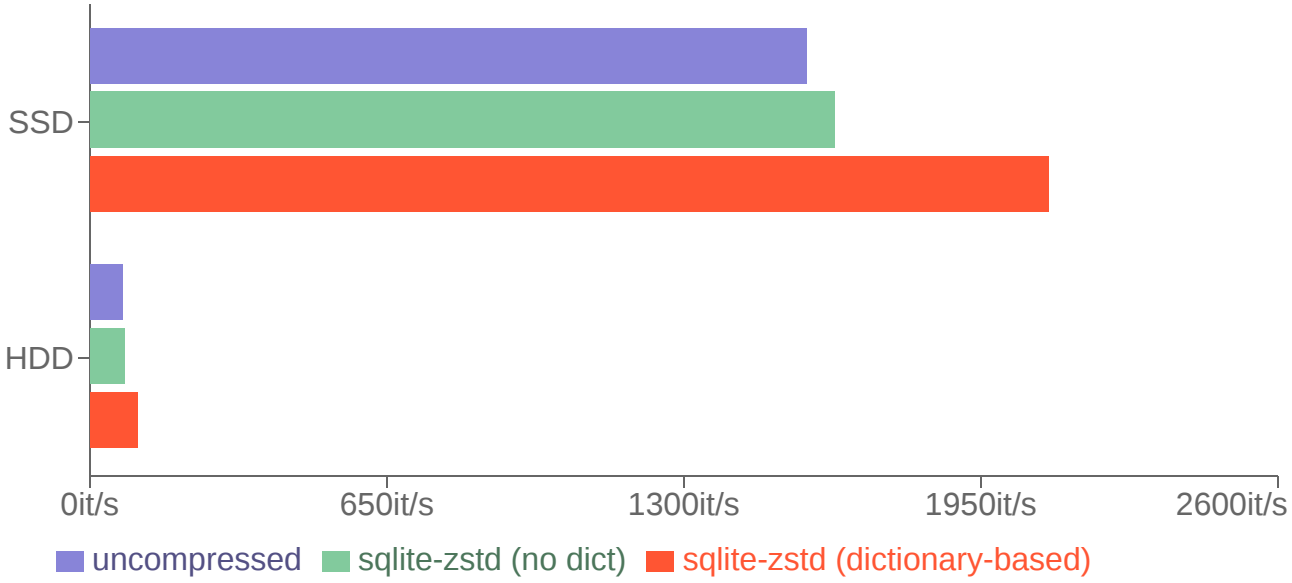


On an SSD with this database, sqlite-zstd can handle 250k SELECTs per seconds. (did I mention SQLite is fast?)

For my use case and probably many other use cases, you have a "hot set" of data that is queried very often, while the rest stays mostly untouched. When fetching sequential data the OS will start prefetching more data and then caching most of it, which is why the performance here is very high. sqlite-zstd reduces the performance a fair bit. The dictionary based decompression is faster than without dictionary, probably because there's less data to process.

Interestingly the performance is still reduced a bit when the storage device is the bottleneck, probably because of the way SQLite (and sqlite-zstd) is fully synchronous.

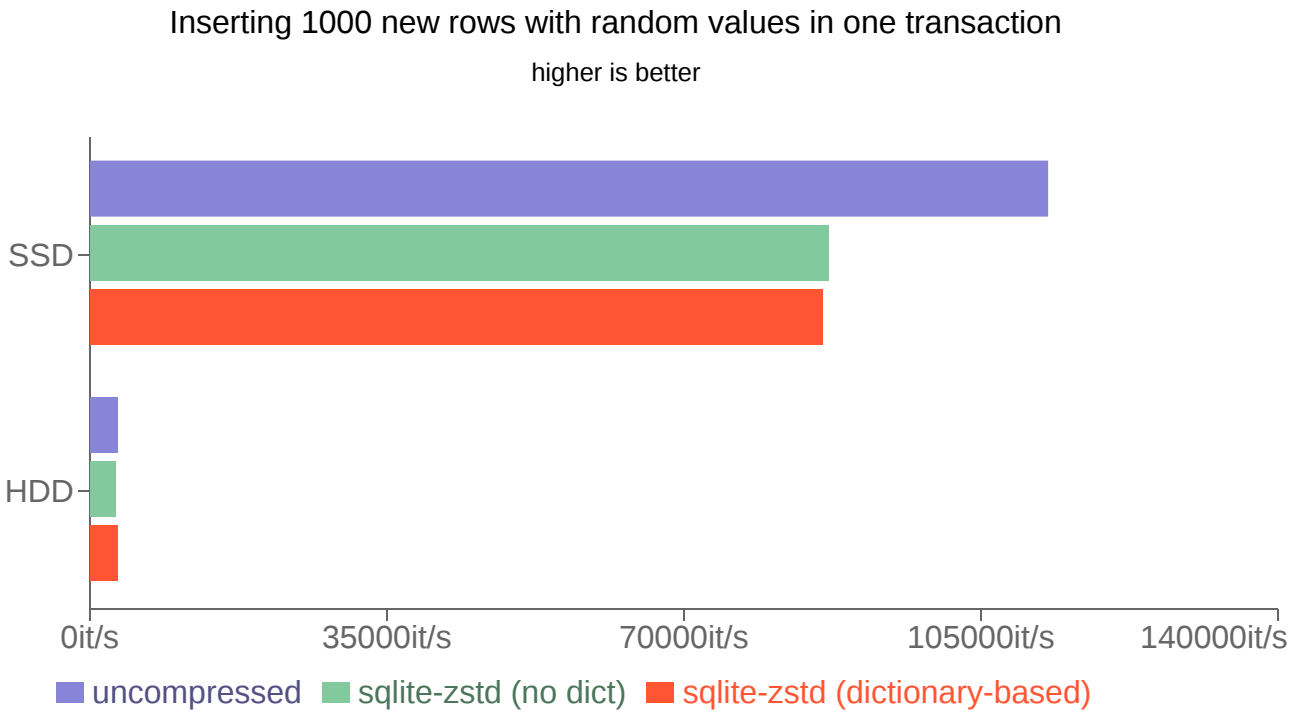
Selecting data randomly



This is a bit surprising: When data is selected randomly, the performance of sqlite-zstd is actually better than the uncompressed database. I'd assume this is because there's less data to read and thus the B-tree is smaller and more of the data cached more quickly by the operating system.

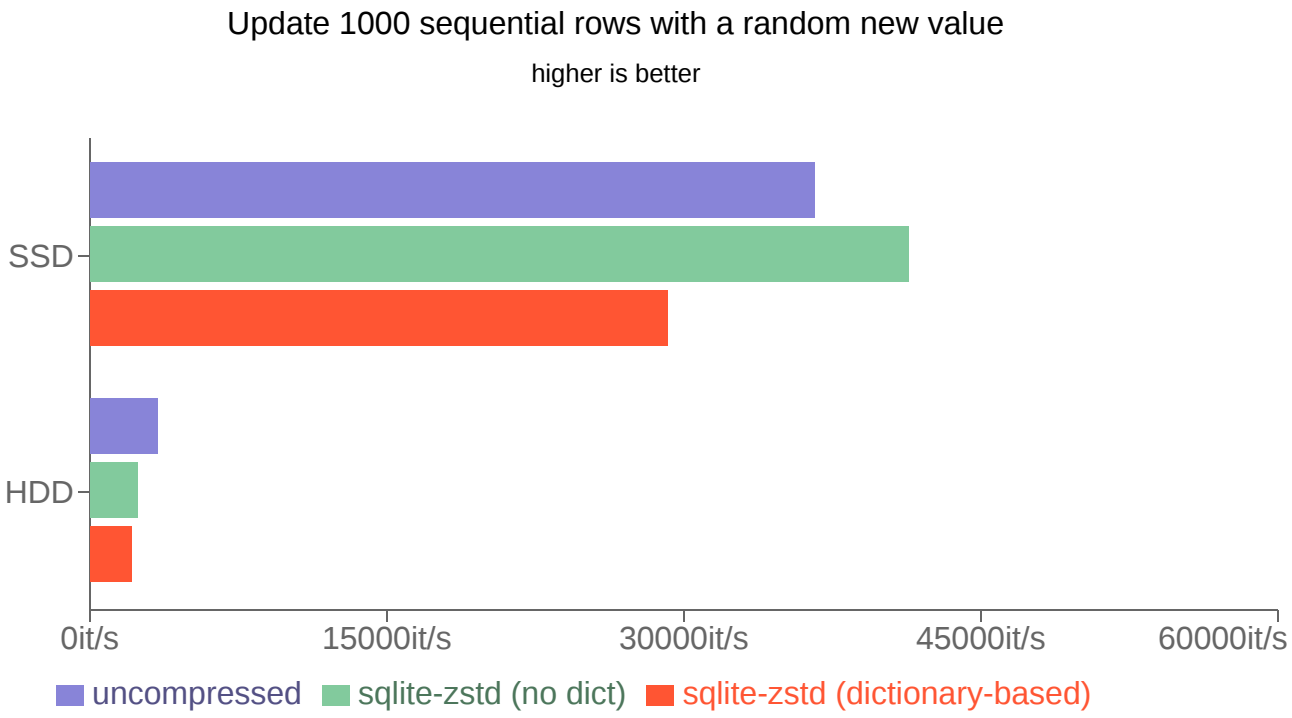
Note that the example database only has four dictionaries since I used a dict_chooser of `id/3000000`. With more different dictionaries, there would be more overhead because random access means sqlite-zstd probably has to load and parse all of them once.

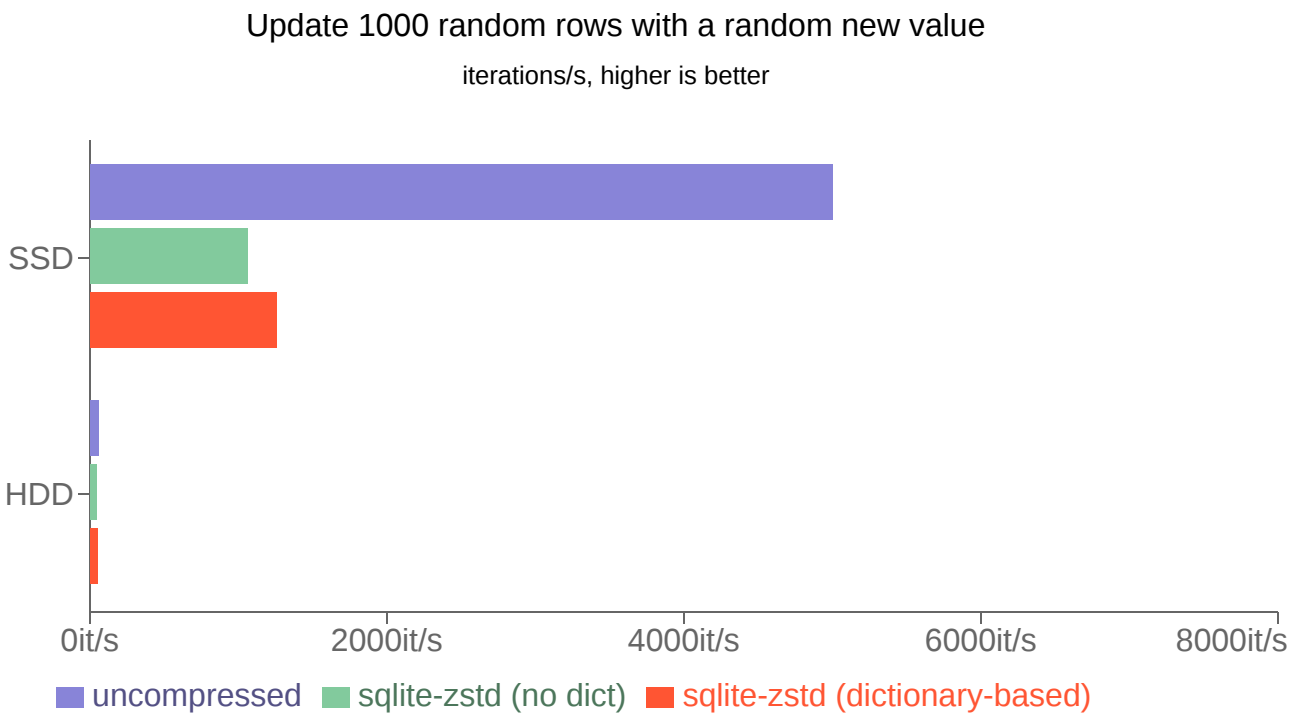
Inserting new data



The performance of inserting data is reduced a bit. Since the data is not compressed on insert, this overhead probably just comes from the way SQLite triggers work.

Updating rows





Honestly I'm not sure what to make of this one. For some reason the performance of updating random rows is much lower with sqlite-zstd, even though no compression is taking place. This might be because of the way the `UPDATE` trigger has a `SELECT` inside so there's multiple b-tree lookups, but I don't know.

Conclusion

For some use cases, sqlite-zstd is great. It can reduce the size of your database by 50 to 95%. The performance impact is there, but considering most operations still run at over 50k per seconds you'll probably have other bottlenecks. There's other optimizations to be done

The same method should work for other databases, with barely any modifications required for e.g. PostgreSQL. I'm not sure why no one has done this before or maybe I just couldn't find it.

Check out the GitHub repo: https://github.com/phiresky/sqlite-zstd

View post source on GitHub