Performance/Avoid SQLite In Your Next Firefox Feature

From MozillaWiki
< Performance

Contents

Introduction

Many Firefox developers see SQLite as a default choice for storing any non-trivial amount of data, this wiki explains why that view is incorrect.

SQLite may seem like a very appealing option when first designing a feature. It offers the familiar and powerful SQL language, our codebase has nice C++ and JavaScript APIs for it, and it's already used in many places in the codebase. However, when choosing storage formats, we have to keep in mind that SQLite is a powerful and reliable database and that it comes with a lot of **hidden complexity**. It's very tempting to take the abstractions it offers at face value and to ignore the complexity happening under the hood. As a result, we have repeatedly seen SQLite become a source of performance problems. This isn't an indictment of SQLite itself -- any other relational embedded DB would pose the same challenges.

SQLite DBs are simply too complex to be used for relatively simple data storage needs. The list below outlines some of the ways SQLite can backfire on developers.

SQLite Pitfalls

Storage Footprint

- Features that regularly add data to the DB (e.g. visited URIs), but have no expiration policy for the data, can easily cause the DB to balloon into tens or hundreds of MBs. This is particularly undesirable on mobile devices

- WAL journals have been known to grow very large, e.g. bug 609122 (https://bugzilla.mozilla.org/show_bug.cgi?id=609122), bug 608422 (https://bugzilla.mozilla.org/show_bug.cgi?id=608422), their size must be limited at a performance cost
- Large transactions cause large journals, so transactions should be kept as small as possible
- Every index contains a complete copy of the indexed data. Creating indexes on lengthy text fields (like URIs) will cause very large indexes to be stored on disk
- Using the default microsecond precision for timestamps causes unnecessary bloat

I/O Patterns

- Your database file can suffer from logical/internal fragmentation due to common SQL patterns
    - Eg appending data to columns with indexes causes interleaved index and table data. This means both table and index scans are non-sequential until a VACUUM is performed.
    - Appending data to multiple tables interleaves tables too.
    - Deleting data does not shrink the database, just marks pages as unused.

- Physical/external fragmentation can also happen due to SQLite using inappropriate defaults
    - SQLite does not preallocate files by default. Appending data to an SQLite in multiple sessions often means the OS has to start a new data block that's not adjacent to the previous one.
    - By default sqlite will grow/shrink the database file by |page_size|. This behavior causes new data blocks to be allocated too. This is especially problematic on OSX, Linux XFS.
    - SQLite CHUNK_SIZE (https://www.sqlite.org/c3ref/c_fcntl_chunk_size.html#sqlitefcntlchunksize) feature is a way to minimize this problem

Factory-default SQLite page size (http://www.sqlite.org/pragma.html#pragma_page_size) is 1024 bytes. When the Mozilla default page size was changed to 32KB in bug 416330 (https://bugzilla.mozilla.org/show_bug.cgi?id=416330), there was a 4x reduction in SQLite IO waits (http://taras.glek.net/blog/2013/06/28/new-performance-people/) according to Telemetry. This is likely due to a reduction in syscalls and OS readahead.

- Performance can significantly degenerate over time (http://sqlite.1065341.n5.nabble.com/Coping-with-database-growth-fragmentation-td44781.html#a44782); scheduling periodic rebuilds of the DB (https://sqlite.org/lang_vacuum.html) is necessary.
    - Note that this "vacuuming" only deals with internal fragmentation, not the external [filesystem] fragmentation
    - VACUUM empties the free pages space of the database file, that means after a vacuum, until there's new free space, inserts will be a little bit more expensive cause they will have to grow the file

- SQLite uses fsync's to guarantee transaction durability and enable recovery from crashes. fsyncs can be very expensive, and happen relatively frequently in the default rollback-journal mode. This performance/reliability trade-off might not be necessary for simple storage needs, especially if the important data can be rebuilt at any time.
- JSON files or log files will show better I/O patterns almost every time, especially if they're compressed and read/written in entirety each time

Memory Usage

- By default, our SQLite wrapper uses a max of 2MB memory cache per connection (http://hg.mozilla.org/mozilla-central/file/444714c3820a/storage/src/mozStorageConnection.cpp#l48).
    - This cache size may be too large for Fennec and B2G, especially if there are multiple connections. The cache size should be adjusted with PRAGMAs
    - Queries unable to use an index will use SCAN TABLE to go through all of the rows, this will quickly grow the page memory cache to the maximum value, this space is never called back until PRAGMA shrink_memory is used, or the connection is closed.

CPU Usage

- Users have found certain Firefox features using 15% of their CPU time because SQL statements were constantly being executed in the background
- Lock contention can occur when using the DB from multiple threads
    - e.g. VACCUM-ing the DB on one thread while executing SQL queries on another

Battery Life

- Same as CPU, important on mobile

Unintended Main-Thread I/O

Main-thread SQL is a known evil, and luckily there are only a few major sources of it left in the codebase, but the Slow SQL dashboard (http://telemetry.mozilla.org/slowsql/) shows that developers are still getting bitten by main thread I/O from unexpected sources:

- PRAGMA statements to set the cache size are done on the main thread, but if this is the first transaction of the session, it can trigger SQLite crash recovery operations. This is particularly bad if using the default WAL journal size. Also note that "crashes" are particularly common on mobile where we never have clean exits.
- Some addons access Firefox DBs directly using main thread SQL

Contending for Access to Storage with the Main Thread

Bad SQLite performance off the main thread also degrades Firefox responsiveness because it can contend with the main thread for access to storage. Sadly our code still does main-thread I/O, and additionally, there will always be main-thread IO from add-ons and swapping.

- Even seemingly simple SQL queries can take multiple seconds to execute
    - e.g. in bug 966469 (https://bugzilla.mozilla.org/show_bug.cgi?id=966469) a user reports the following statement taking 22 seconds to execute: `DELETE FROM moz_pages WHERE id NOT IN (SELECT DISTINCT(pid) FROM moz_subresources);`

UX Impact of Slow Queries

Slow DB operations hurt UX. For example, you might have noticed that the AwesomeBar sometimes takes a very long time to fetch any auto-complete suggestions.

- The schemas have to be designed carefully and the DB has to be maintained periodically
- Fragmentation causes extra seeks, and the problem is made worse by magnetic hard disks and

cheap or old SSDs. Vacuuming DBs helps, but vacuums are resource-intensive and scheduling them can be tricky. See the Tips section

How to Store Your Data

- If you need to store a small amount of data (less than 1MB), you should use JSON files, and do all your I/O off the main thread. Simple solutions with linear worst-case performance are ideal.
    - If you're working with larger amounts of data (roughly on the order of 1MB), you should compress the JSON data with Snappy or lz4 before writing it to disk
    - You can use OS.File's writeAtomic() (https://developer.mozilla.org/en-US/docs/JavaScript_OS.File/OS.File_for_the_main_thread#OS.File.writeAtomic%28%29) method. More information on writeAtomic in this blog post (http://dutherenverseauborddelatable.wordpress.com/2014/02/05/is-my-data-on-the-disk-safety-properties-of-os-file-writeatomic/)
    - If your workload involves a lot of strings, don't use SQLite. Store your data in external files.
    - If your workload involves a lot of blobs, don't use SQLite. Store your data in external files.

- If you have a large dataset, but you don't need to run complex queries over it, evaluate a JSON files structure on disk
- If you must read and write most of your data every time, evaluate using a JSON file
- For larger datasets or when SQL is absolutely necessary, use SQLite.
    - Make sure you understand how SQLite works (https://sqlite.org/docs.html), carefully design your schemas and then profile your implementation.
    - IndexedDB is implemented on top of SQLite and has additional issues

NOTE: *We're currently working on an intermediate solution based on log-storage which will reduce the amount of data written for small modifications to data already on disk.*

How to Best Use SQLite If You Really Need To

Important Pragmas

See the list of SQLite pragmas (http://www.sqlite.org/pragma.html), the defaults are probably not OK. We already over-ride some defaults in the moz.build file (http://mxr.mozilla.org/mozilla-central/source/db/sqlite3/src/moz.build) and mozStorageConnection.cpp (http://mxr.mozilla.org/mozilla-central/source/storage/src/mozStorageConnection.cpp#622).

- Use a write-ahead log (https://www.sqlite.org/wal.html) for fewer fsyncs
    - `PRAGMA journal_mode = WAL;`
    - Though limit the size of the journal, by using a small autocheckpoint and a journal_size_limit about thrice that size
        - `PRAGMA wal_autocheckpoint = 16; /* number of 32KiB pages in a 512KiB journal */`
        - `PRAGMA journal_size_limit = 1536; /* 512KiB * 3 */`

    - If it's not possible to use WAL journaling, consider using a TRUNCATE journal

- Set the DB cache size based on the device specs, and consider adjusting it dynamically based on memory pressure on mobile devices
    - `PRAGMA cache_size = X;`

- For large DBs prone to fragmentation, pre-allocate a large empty DB initially by increasing the DB's growth increment
    - This can be done by calling Connection::SetGrowthIncrement() which changes the value of SQLite's `SQLITE_FCNTL_CHUNK_SIZE`. See bug 581606 (https://bugzilla.mozilla.org/show_bug.cgi?id=581606) for reference.

- Use a memory temp store if the amount of data in memory entities is acceptable, otherwise temp data will be written to disk, causing unexpected I/O
    - `PRAGMA temp_store = MEMORY`

More Tips

- It should go without saying that you should **never execute SQL on the main thread**
    - Use the SQLite.jsm wrapper (https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules/Sqlite.jsm) for DB operations from JavaScript, it's fully asynchronous and off-main-thread
    - For native code, use the async methods (https://mxr.mozilla.org/mozilla-central/source/storage/public/mozIStorageAsyncConnection.idl)

- Model how large your DB will grow on desktop/mobile. Report the DB size & memory use to Telemetry (https://developer.mozilla.org/en-US/docs/Performance/Adding_a_new_Telemetry_probe) to confirm your estimates were accurate. **Implement an expiration policy!**
- Profile your implementation and measure the number of SQL operations performed over the course of a typical session if it’s an “always-on” feature and you suspect it might cause performance issues
- Don’t index on text fields, index on hashes of text fields. Indexes can become very large.
- Don't use the default precision for timestamps (microseconds). Desktop apps are unlikely to ever require this level of precision
- Maintain your DB with VACUUMs (https://sqlite.org/lang_vacuum.html) during Firefox version upgrades or during idle times. Run ANALYZE (http://www.sqlite.org/lang_analyze.html) periodically if necessary
    - You should have a vacuum plan from the get-go. **This is a requirement for all new code**.

- If working on the same data from multiple connections, consider using an unshared memory cache (http://www.sqlite.org/sharedcache.html), to avoid memory contention. This will improve concurrency.
    - Though, it will also multiply the memory cache per the number of connection, so it's a memory/performance choice you should do carefully

Retrieved from "https://wiki.mozilla.org/index.php?title=Performance/Avoid_SQLite_In_Your_Next_Firefox_Feature&oldid=962016"
