Thorough Analysis of The Database Issues

The primary factors of the database issues and strategies for effectively resolving them

I have conducted a thorough analysis of the database issues based on your requirements, identifying the root cause behind them. Subsequently, I have formulated strategic recommendations to effectively address and resolve these issues.

Thorough Analysis of the Database Issues

The primary factors of the database issues and strategies for effectively resolving them

The database issues

A series of pragmas will speed up the insertion of large amounts of data into the database.

As far as corruption of the database is concerned, since this was just a simple one time insertion of 110,000 records, it will not be observed.

It will take a lot of such insertions for the database to be corrupted.

The primary factors of the database issues

1. Transaction size that is too large
2. Table structure that is not optimized
3. Pragma statement that is unstable
4. The operations are conflicting with each other

Strategies for effectively resolving them

1. Minimize the size of transactions to avoid excessive bulk.
2. Optimize the database structure and performance.
3. Implement a partitioned database by the values of date field.
4. Ensure the stability and reliability of the SQLite database.
5. Create an index on the time field to improve query efficiency.

Performance Comparison

The primary factors of the database issues

I have tested the current status of the database using the file "PerformanceSamples.db" and identified some issues that need to be corrected.

1. Transaction size that is too big
   1. What will happen?

* DB Corruption
* Insertion speed lower
  1. Reason

In the table titled “AmdPmfSamples”, we can determine size of each record as following.

* + 1. Why the insertion speed is lower if the transaction size is too big?

According to the table design and queries we use, the minimum size of each record is determined to be at least 360 Bytes, calculated as follows:

21 \* 4 + 25 + 71 + 15 \* 2 + 5 \* 2 + 9 + 11 + 11 + 41 + 27 + 41.



In this specific test case (TelemetryStorage), it is observed that this module adds 110,000 rows in a single transaction.

Consequently, the total size of a transaction containing 110K rows would be at least 39.6M (360 \* 110K).

To follow this guideline, it is important to choose the right number of records to insert in each transaction.

* Let's call this number 'x'.
* If the next 'x' records are produced, a new transaction will be created and executed.
* It is crucial to make sure that the time it takes to complete a transaction is shorter than the time it takes to generate the next set of 'x' records if the execution and building transaction are done in separate thread.

To determine the optimal value for 'x', it is necessary to measure the performance of the insert operation.

Inserting data in smaller batches within a transaction sometimes often enhances performance compared to a single large transaction.

Large transactions in SQLite can be slower than smaller transactions due to a few reasons:

* Increased Memory Usage: Large transactions require SQLite to allocate more memory to track and store all the changes made. This increased memory usage can lead to performance issues, especially on devices with limited resources.
* Locking and Concurrency: SQLite uses a locking mechanism to ensure data consistency during transactions. When a large transaction is being executed, it may hold locks for an extended period, preventing other operations from accessing or modifying the database concurrently. This can result in delays and decreased overall throughput.
* Journaling and Disk I/O: SQLite maintains a write-ahead log or journal to support atomicity and durability of transactions. In large transactions, the journal file can grow significantly and involve more disk I/O operations, slowing down the transaction processing time.
* Rollback and Undo Operations: If a large transaction needs to be rolled back, SQLite must undo all the changes made within that transaction. The larger the transaction, the more work is required to perform the rollback, which can contribute to slower performance.

Therefore, careful measurement and analysis of the insert operation should be conducted to determine the ideal value for 'x'.

* + 1. Why db is sometimes corrupted when writing large transaction?

I believe that there is no corruption when the database engine executes only large transactions. However, it frequently occurs when vacuum operation starts while large transaction is being written.

The main reason for database corruption seems to be that insertion and vacuum processes may overlap in the free space of the database. The "free space" refers to the areas in the database where rows have been marked as deleted.

When a row is deleted, the space it occupied is marked as free and can be reused for new row inserts if the size of the new row is smaller than the free space segment size.

In the SQLite database, writing a new row always requires the data to be written to the hard disk. However, when the write operation is performed by the operating system, the data to be written is cached for a certain duration until the queued data is populated into the cache buffer or the flush() function is executed.

Repeated execution of insertions involves a significant number of hard disk writes and caching before the writes are handled by the operating system kernel.

During this period, if a vacuum operation is executed in a separate thread, there could be an unexpected scenario where insertion data and the removal of free space operations are completely mixed in the cached buffer.

When the operating system writes this cached buffer to the hard disk, the internal database file structure containing the mixed data is corrupted.

Therefore, I have attempted to test this corruption using a test project that I wrote and have found that there are occasional database crashes when these two operations occur simultaneously, albeit at a low frequency.

1. Table structure that is not optimized
   1. What will happen?

* Speed issue & Database size
  1. Reason

The structure of the table named AmdPmfSamples is as follows:

CREATE TABLE AmdPmfSamples (time TEXT, pmf\_version INT, device\_state INT, platform\_type INT, lid\_state INT, user\_engaged INT, user\_present INT, hpd\_prob BLOB, hpd\_dist BLOB, angle BLOB, display\_num INT, display\_info TEXT, power\_slider INT, policy\_id TEXT, policy\_priority TEXT, mouse\_num BLOB, keyboard\_num BLOB, ambient\_light BLOB, display\_brightness BLOB, fan\_id BLOB, ppm\_profile INT, bios\_input TEXT, bios\_output TEXT, power\_source INT, designed\_battery\_capacity BLOB, full\_battery\_capacity BLOB, battery\_charge\_rate BLOB, battery\_charge\_level BLOB, gfx\_clock\_freq INT, soc\_clock\_freq INT, mem\_clock\_freq INT, gfx\_activity INT, uvd\_activity INT, voltage TEXT, current TEXT, power TEXT, core\_frequency TEXT, core\_power TEXT, core\_temp TEXT, gfx\_temp INT, soc\_temp INT, skin\_temp INT, throttler\_status INT, current\_socket\_power INT, stapm\_current\_limit INT, apu\_power BLOB, d\_gpu\_power BLOB)

And the structure of the table named EnergyEMISamles is as follows:

CREATE TABLE EnergyEMISamples (time TEXT, absolute\_time BLOB, sensor\_name TEXT, sensor\_model TEXT, sensor\_oem TEXT, sensor\_revision INT, sensor\_channel INT, absolute\_energy BLOB)

Within the table structures, there exist certain fields that contain duplicated values across multiple records.

For instance, the field AmdPmfSamples.displayInfo does not have distinct values for all records.

["{\"displayType\":2147483648,\"displayEnabled\":true}","{}","{}","{}"]

This string value is stored repeatedly in every record of this table.

Thus, to reduce the size of the database file, we can create a new table specifically for this "displayInfo" and assign only the unique identifier (id) of the table to each record instead of storing the entire string value repeatedly.

Similarly, the fields EnergyEMISamples.sensor\_name, EnergyEMISamples.sensor\_model, and EnergyEMISamples.sensor\_oem also lack distinct values.

These redundant values are not necessary within the main table records.

This non-optimized table structure leads to increased record size and slower insertion speed.

In accordance with the principles of relational database design, these values should be stored in separate tables and linked to the main tables through relational identifiers.

1. Pragma statement that is unstable
   1. What will happen?

* DB Corruption
  1. Reason

SQLite implements transactional writes, and larger transactions may result in increased time for the database to execute essential disk I/O and locking operations.

However, in this scenario, disk I/O is not a concern since we have configured the journal\_mode pragma as MEMORY.

This configuration surely enhances the performance of DML query execution.

Nonetheless, it is important to note that in the event of a crash, journal information cannot be recovered as the journal data is stored in the process's memory.

If the program accidentally exits or is terminated by process manager, it means that the journal data will be missing, resulting in data loss.

Therefore, to ensure stability, it is advisable not to configure the journal\_mode pragma as MEMORY.

If we don't set the journal\_mode pragma as MEMORY, the default value is set to WAL.

WAL stands for Write-Ahead Logging. In SQLite, it is a journaling mode that allows concurrent read and write operations to the database. It achieves this by writing changes to a separate log file before modifying the actual database file. This ensures durability and reduces the possibility of data corruption in case of a crash or power loss.

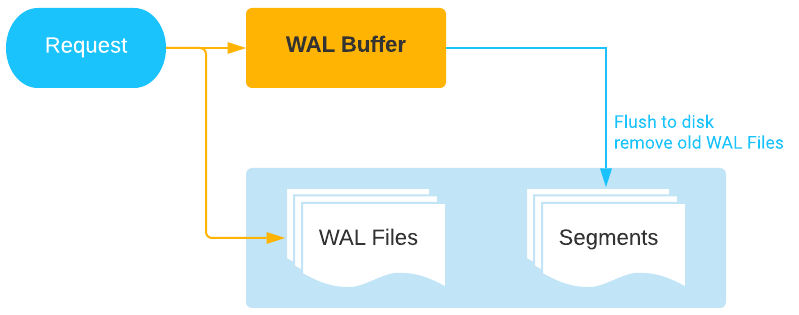


Figure 1. Standardized and reliable write-ahead log

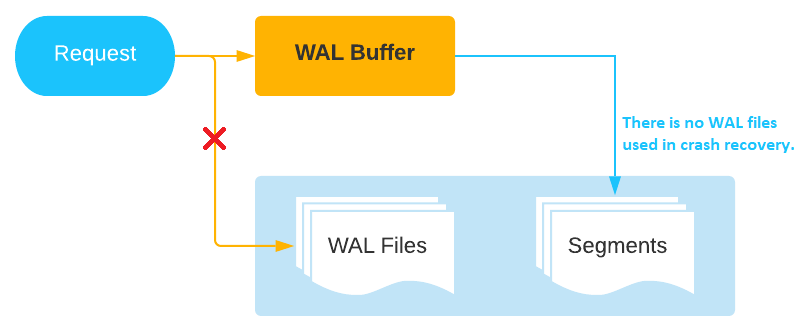


Figure 2. Improper utilization of the write-ahead log

1. The operations are conflicting with each other
   1. What will happen?

* DB Corruption
* Insertion speed lower
  1. Reason

After careful analysis, I have identified a collision problem within the program.

Currently, these operations are executed simultaneously, resulting in collision conflicts and subsequent delays.

For instance, when the vacuum operation is in progress, the insert operation is delayed until the former is completed, and vice versa.

Consequently, the insert operation appears to take a longer time than necessary.

Based on my assessment, I believe this is the primary underlying cause of the performance issues in your database.

To address this, it is crucial to establish a database structure that avoids collision among these operations.

One approach is to split the database file into multiple files, thereby mitigating collision issues and enhancing overall performance.

We will be describing the solution in next chapter.

Strategies for effectively resolving them

1. Minimize the size of transactions to avoid excessive bulk.

Based on my experience, it is advisable to set the transaction size to closely match the cache size of the hard disk.

This is because the operating system asynchronously sends I/O requests, and the hard disk buffers them in its cache until it is full and performs a write operation.

Therefore, it is recommended to divide the large transaction into smaller transactions using the following approach:

for(int i = 0; i < 110000; i += n) {

/\* start transaction \*/

for(int j = i; j < ((i+n) > 110000 ? 110000 : (i+n)); j++) {

/\* execute insert query \*/

}

/\* commit transaction \*/

}

It is advisable to perform measurements and tests in order to determine the value of 'n'.

The sizes of hard disk caches can vary from what is stated by the manufacturer.

However, a common example of a hard disk cache size is 64MB or 256MB.

1. Optimize the database structure and performance.

After analyzing the second issue, it is clear that there is potential for optimizing the structure of the table.

Therefore, I propose the following table structure, which is illustrated in the accompanying figure:

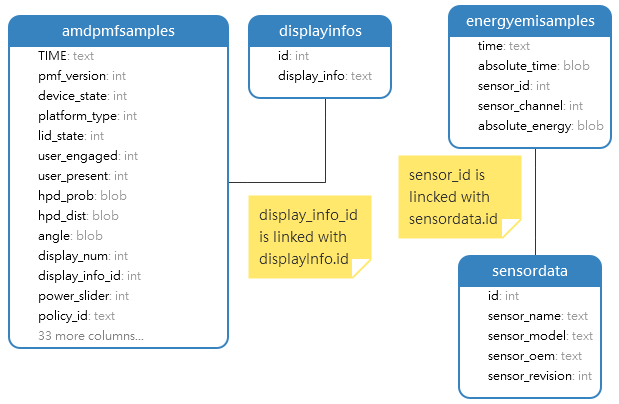


Figure 3. Suggested table structure in relationship

To achieve this, we can decrease the size of each record by a certain percentage.

1. Implement a partitioned database by the values of date field.

According to the experience of most developers, using a SQLite database larger than 200MB tends to generate overhead for DML operations.

Therefore, I propose implementing a partitioned SQLite database based on date.

This approach involves having a separate database file for each day, resulting in approximately 30 or 31 database files per month.

- Advantages of a partitioned database include:

* The collision issue will be permanently resolved, resulting in a significant performance upgrade overall.
* Ensuring that the size of each database file remains within limits, thereby avoiding issues related to insert speed limitations and database crashes.
* Eliminating the need for vacuuming the database. Instead, we can simply delete old database files to remove outdated data.

- However, there are some disadvantages to consider:

* Searching data from the partitioned database becomes more complex.  
  We need to open one database and attach all the database files to perform data searches.  
  Although this requires some complex coding, it is not an insurmountable challenge.  
  The search queries remain the same.
* Creating a new database file every day is necessary.  
  However, there is an efficient method to handle this.

When the program needs to create a new database file, we can simply copy an empty database file.

To enhance the performance and stability of the program, it is essential to implement a fully partitioned database.



Figure 4. Vacuuming, searching and insertion are   
conflicted each other in a database file

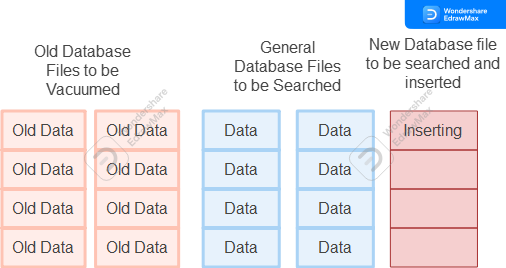


Figure 5. Partitioned databases prevent collisions among the   
vacuuming, searching and insertion.

1. Ensure the stability and reliability of the SQLite database.

I have discovered the presence of a pragma statement resembling this format:

PRAGMA synchronous=OFF;PRAGMA journal\_mode=MEMORY;PRAGMA temp\_store=MEMORY;

The primary cause of database crashes is the inability of SQLite to locate the journal file containing logs upon program restart.

Therefore, it is imperative to refrain from utilizing this pragma statement in order to maintain the stability of the SQLite database.

1. Create an index on the time field to improve query efficiency.

After conducting a thorough analysis, I have discovered that there is currently no index on the table. The significance of an index lies in its ability to enhance search operations within the database.

Without an index, search operations can become considerably slow, adversely impacting the overall performance of the database. However, with the presence of an index, the insert speed may be slightly affected, but the search speed on the index is significantly multiplied.

Considering this, I recommend creating an index specifically on the time column to optimize search operations.

The reason for the decreasing search speed in non-indexed databases is as follows:

When a search is performed on a database without an index, the database management system (DBMS) needs to scan all the records in the database file to find the matching search conditions. This means that all pages of the database need to be loaded into memory and compared to determine if they meet the search condition. In this case, the decreasing speed is dependent on the input/output (I/O) operations of the hard disk and the utilization of the CPU.

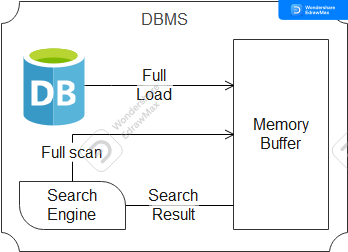


Figure 6. Full scan on the database file without an index

In contrast, when there is an index on a table, the DBMS does not need to perform a full scan of all the records. Instead, it can scan the index tree (typically a B-Tree) and retrieve only a few relevant records that match the search condition. It then compares these few selected records to determine if they meet the search condition. Significantly less Disk I/O and CPU utilization are required for search operations, resulting in a notable improvement in search speed.

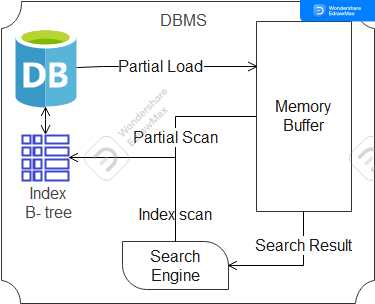


Figure 7. Partial index scan on the database file with an index

Performance Comparison

Device environment: Core i7 3.4GHz, 8GB RAM, SATA HDD 1TB

Record count: 5,000,000

Table 1. Performance comparation

|  |  |  |  |
| --- | --- | --- | --- |
|  | Insertion Speed | Vacuum Time | Search Time |
| Current method (Pragma JournalMode=MEMORY) | 110K/s | 75s | 10s |
| Suggested Method | 50K/s | 0s | 0s |

I have configured the row count within a transaction to be set at 500.

The vacuum time is 0 seconds since we only delete the old database file instead of executing a vacuum query.

Additionally, the search time is also 0 seconds due to the indexing of the time field.

Based on the table provided above, implementing the suggested method may potentially cause a decrease in speed due to the emphasis on ensuring stability. However, it results in a vacuum time and search time of 0 seconds.

Consequently, there is no collision among the operations, and the overall performance is expected to significantly improve.