SQLite-presentation notes

**Serverless:** (Classical-Serverless)

The **database** engine **runs** **within** the **same** **process**, **thread**, and **address** **space** **as** the **application**. There is **no** **message** **passing** **or** **network** **activity**. [ neo-serverless runs on cloud]

Normal:

communication to server via intermediary process (typically tcp/ip) -> i.e., separate process

* Better protection from bugs (stray pointers can’t corrupt server memory)
* Single persistent process, => Any program that can access the disk is able to use an SQLite database.

SQLite:

processes read and write directly from db files on disk (no intermediary server process)

* Advantage:
* no separate server process to install, setup, configure, initialize, manage, and troubleshoot = zero-configuration =>  Any program that is able to access the disk is able to use an SQLite database
* Any program that is able to access the disk is able to use an SQLite database. (So, **no configuration**)

Source: https://www.sqlite.org/serverless.html

**Transactional (ACID): [ we did this in uni, so quickly over it, maybe we can test this ourself with the show database in presentation]**

<https://www.sqlite.org/transactional.html>

<https://sqlite.org/forum/info/0086fcd1e990c370>

<https://www.tutlane.com/tutorial/sqlite/sqlite-acid-transactions>

<https://www.sqlite.org/wal.html>

https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions

A transactional database is one in which all changes and queries appear to be Atomic, Consistent, Isolated, and Durable (ACID).

All changes within a single transaction in SQLite either occur completely or not at all, even if the act of writing the change out to the disk is interrupted by

* a program crash,
* an operating system crash, or
* a power failure.

The claim of the previous paragraph is extensively checked in the SQLite regression test suite using a special test harness that simulates the effects on a database file of operating system crashes and power failures.

**How to test:**

Atomic: Try making two changes and then either ROLLBACK or COMMIT. Look for not-ROLLBACKED and not-COMMITED changes. Check combinations of rows changed by 1 UPDATE, rows changed by 2 UPDATEs in the same table, different tables or different attached databases.

Consistency: Try inserting/updating rows that violate constraints (remember to enable foreign key constraints). What those are is up to the author of the schema.

Independance/Isolation: Try performing two transactions on the same row/table/database/across multiple attached databases in independent threads/processes. Check that data reread after both transactions are complete conform to expecations. In SQLite you will have to set a busy timeout to avoid the second transaction failing with "database is locked". Try to start the transactions simultaneously, so the changes occur in both orders.

Durability: Try killing threads/processes/power while transactions are in all states of progress (before/after first/last change, during commit), check to see if all comitted transactions are still present after the journal/WAL file has been processed.

Result:

Atomic: allowing a transaction to be rolled back if and part of the transaction fails

Consistency: enforces constraints, like unique and foreign keys, by raising an error and rolling back the violating transaction

Independence: supports independent transactions on the same row, table database, and across multiple attached databases in different threads or processes In case if two transactions executing concurrently only respective transaction results will appear to the client, not other clients transaction results.

Durability: if error before commit => rollback, ensures this via a rollback journal / write-ahead-log

**Cross Platform:**

<https://www.sqlite.org/different.html#:~:text=The%20SQLite%20file%20format%20is,use%20the%20same%20file%20format>. [this has all principles explained]

Cross Platform, since it uses .db, which all machines use. Backwards compatible

**.sqlite, .sqlite3, .db, .db3, .s3db, .sl3**

**SQL vs SQlite:**

<https://www.sqlitetutorial.net/sqlite-vs-other-sql-databases/>

<https://www.geeksforgeeks.org/difference-between-sql-and-sqlite/>

Language to manage & manipulate / communicate rel. db **vs** library that implements database engine (subset of sql, basic commands

Configuration **vs** no configuration

Large scale db **vs** small scale db

Network **vs** no network

Slow & resource intensive **vs** small, fast, (reliable)

Extensive functionality **vs** basic functionality

* In General different use cases, big / enterprise applications and projects **vs** small scale & embedded projects

**History:**

Jst asked gpt real quick about the history, cause I would maybe extend it later than 2010 and also functionality, when came acid and so on

* 2000: SQLite version 1.0.0 is released. It was designed as a lightweight, self-contained, and embeddable relational database management system.
* 2001: SQLite version 2.0.0 is released, which introduced a B-tree implementation to replace the original GNU database manager used in SQLite 1.x. This version also added support for transactions, triggers, and views.
* 2004: SQLite version 3.0.0 is released, which featured a number of improvements over version 2.0.0, including a redesigned virtual machine, support for additional data types, and a new query optimizer. This version also introduced the use of a single database file for both the schema and data.
* 2005: SQLite version 3.2.0 is released, which introduced support for the SQL-92 WITH clause, as well as new date and time functions.
* 2007: SQLite version 3.5.0 is released, which added support for full-text search and other new features.
* 2008: SQLite version 3.6.0 is released, which introduced support for atomic commit and rollback, as well as new features like the REPLACE statement and online backup API.
* 2010: SQLite version 3.7.0 is released, which introduced support for write-ahead logging (WAL), improved query planning, and other new features.
* 2013: SQLite version 3.8.0 is released, which introduced support for the JSON1 extension and other new features.
* 2016: SQLite version 3.14.0 is released, which introduced support for row values, enhanced the query planner, and added other new features.
* 2019: SQLite version 3.29.0 is released, which introduced support for incremental blob I/O, as well as new features like generated columns and window functions.

Today, SQLite remains a popular choice for embedded and client-server applications, due to its small footprint, low resource requirements, and high reliability. It is widely used in a variety of applications, including web browsers, mobile devices, and desktop software.