Aalto University
School of Science
Bachelor's Programme in Science and Technology

Adapting SQLite to the Distributed Edge: A Comparative Study of Different Adaptations

Bachelor's Thesis

March 28, 2025

Kalle Ahlström

Aalto University
School of Science
Bachelor's Programme in Science and Technology

ABSTRACT OF
BACHELOR'S THESIS

Author:	Kalle Ahlström
Title of thesis:	Adapting SQLite to the Distributed Edge: A Comparative Study of Different Adaptations
Date:	March 28, 2025
Pages:	37
Major:	Computer Science and Engineering
Code:	SCI3027
Supervisor:	Professor Lauri Savioja
Instructor:	Doctoral Researcher Juho Vepsäläinen, Doctoral Candidate Pekka Enberg (Helsinki University) (Department of Information and Computer Science)

This thesis is a comparative study of different SQLite adaptations and their viability for Distributed Edge Computing environments. The four adaptions chosen for closer analysis are FaaSFS, LiteFS, libSQL, and rqlite. An ideal architecture is presented and that is used as a basis for the comparison between the adaptations.

None of the solutions studied implement the ideal architecture. Each has different trade-offs, making the selection process of what to use use-case specific. The main trade-offs highlight significant differences between write performance, stronger consistency guarantees, and/or increased read latency.

According to the CAP theorem, distributed systems can't fully guarantee consistency, availability, and partition tolerance simultaneously. Empirical analysis of the chosen adaptations indicates that implementation designs inherently favor either the CA or AP model, thereby substantiating the theorem.

Keywords:	edge, edge computing, sqlite, distributed systems, CAP theorem
Language:	English

Aalto-yliopisto Perustieteiden korkeakoulu Teknistieteellinen kandidaattiohjelma	KANDIDAATINTYÖN TIIVISTELMÄ

Tekijä:	Kalle Ahlström
Työn nimi:	SQLite:n sovittaminen hajautettuun reunalaskentaan: vertaileva tutkimus eri sovituksista
Päiväys:	28. maaliskuuta 2025
Sivumäärä:	37
Pääaine: **Koodi:**	Tietotekniikka SCI3027
Vastuuopettaja: **Työn ohjaaja(t):**	Professori Lauri Savioja Tohtorikoulutettava Juho Vepsäläinen, väitöskirjatutkija Pekka Enberg (Helsingin yliopisto) (Tietojenkäsittelytieteen laitos)

Tämä opinnäytetyö vertailee eri SQLite-sovituksia ja niiden soveltuvuutta hajautettuihin reunalaskentaympäristöihin. Lähempään tarkasteluun valitut neljä sovitusta ovat FaaSFS, LiteFS, libSQL ja rqlite. Vertailun pohjana käytetään esitettyä ideaalista arkkitehtuuria.

Yksikään tutkituista ratkaisuista ei täysin toteuta ideaalista arkkitehtuuria, vaan jokainen tekee erilaisia kompromisseja. Ratkaisujen valinta riippuu siten käyttötapauksesta. Keskeisimmät erot ilmenevät kirjoitusnopeuden, vahvan konsistenssin takaamisen sekä kasvaneen lukulatenssin välillä.

CAP-teoreeman mukaisesti hajautettu järjestelmä ei voi samanaikaisesti täysin taata konsistenssia, saatavuutta ja osituksensietoa. Empiirinen analyysi osoittaa, että tutkitut sovitukset painottavat toteutuksessaan joko CA- tai AP-mallia, mikä tukee kyseistä teoreemaa.

Avainsanat:	reunalaskenta, sqlite, hajautetut järjestelmät, CAP-teoreema
Kieli:	Englanti

Contents

1 Introduction

With the emergence of global software applications that need to serve users in different geological locations with varying connection capabilities, new deployment, and computational models are needed. This is especially apparent in web applications requiring high availability, fast response times, or both. The raised capabilities of edge computing have started the work towards solving these problems, but it still encounters the challenge of managing the application state in a globally distributed system. This forces applications to store and retrieve data remotely, often from a geographically centralized cloud data center, resulting in inefficiencies due to data transfer latency and throughput limitations. More traditional cloud platforms also struggle to provide the fine-grained resource provisioning and scalability needed at the edge, reducing the performance gains that edge computing promises [72; 44].

1.1 Embedded Edge Databases

One of the potential solutions to this Edge latency bottleneck could be a distributed database embedded within the application process, enabling significantly reduced database latencies. Given SQLite's' [36] inherent characteristics as an embedded database, it allows the integration of the database directly within the application itself. This embedding capability is a significant advantage, as it simplifies both the development experience and the application's performance by eliminating the need for separate database servers and reducing the overhead associated with managing external database connections. This feature enables the effective deployment of SQLite within distributed edge contexts when combined with mechanisms for replication synchronization.

1.2 Replication and Performance

A replication synchronization mechanism allows for consistent data across multiple instances of a database in a distributed setting, enhancing data reliability and accessibility [64]. By utilizing embedded SQLite databases, edge networks can leverage localized data processing and storage, which is crucial for applications requiring low latency and high availability. This configuration optimizes performance by minimizing dependency on centralized servers and increases resilience and scalability across the network infrastructure.

Reading data directly from a file system is typically faster than accessing it via network operations, thereby markedly enhancing efficiency, particularly in sequential reads scenarios. Sequential reads refer to operations where each query is contingent upon the results of preceding queries, necessitating multiple database accesses. This rapid access facilitated by the file system improves the feasibility of employing sequential queries without substantial performance penalties. Consequently, developers can sequentially structure their queries without overly concerning themselves with performance degradation. This approach not only enhances the readability of the code but also contributes to the overall maintainability of the application by simplifying data access patterns and reducing the complexity inherent in handling interdependent data retrieval tasks.

1.3 Research problem

This study explores how SQLite and some adaptations effectively support Edge Computing platforms where resources are constrained as distributed databases. Edge Computing environments, often characterized by limited processing power, storage, and network capabilities, require lightweight, efficient database solutions to manage and process data closer to the source. The research question of this study is: What modifications, benefits, and limitations characterize the use of SQLite in distributed Edge computing environments? By examining the current state of SQLite and its adaptations, the study aims to identify these factors.

1.4 Thesis structure

The thesis is structured in the following manner: Section 2 outlines SQLite and fundamental database concepts, Section 3 motivates Edge Databases by giving a brief background on Edge computing, whereas Section 4 introduces distributed systems theory and related challenges. Section 5 ties these concepts together, comparing various SQLite adaptions for Edge computing. Section 6 highlights key observations: the CAP theorem's applicability to a globally distributed SQLite and the embedding of SQLite in Serverless Edge Computing environments. Finally, Section 7 concludes the thesis by summarizing the main findings and discussing future work.

2 SQLite: an Embedded Database

This section focuses mainly on SQLite, highlighting its key features for serving as an embedded database within resource-constrained environments, such as edge computing. After briefly introducing basic concepts, we take a deeper dive into SQLite itself, looking into the architecture and its ACID compliance and transaction handling via either the Rollback Journal or the Write-Ahead Log.

2.1 Database

A database is a data collection designed to store large amounts efficiently, enabling data integrity and security.

Relational databases do this by storing data in tables. A table consists of rows that all share the same columns, defined during table creation. Relational Databases use variations of Structural Query Language (SQL) as their interface for data manipulation and database queries.

2.2 Transactions & ACID

A transaction can be considered a unit of work within a database system that ensures data consistency, isolation, and durability [53]. A transaction is a set of sequential interactions with a database executed indivisibly: either all interactions or none are executed [46]. A typical example of such a transaction is transferring money (see Figure 1). The balance change in both accounts is visible if and only if the `COMMIT TRANSACTION` is reached without errors. Ending a transaction is done by committing the transaction.

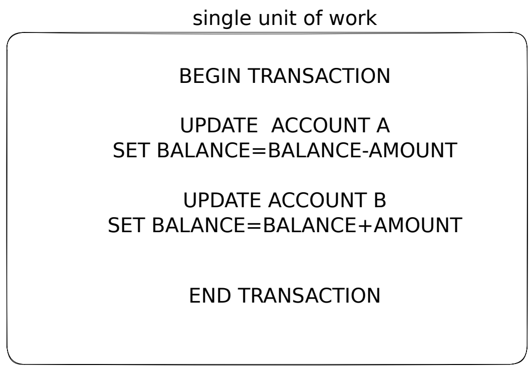

Figure 1: SQL transaction code illustrating the funds transfer between Account A and Account B. Although Account A is updated before Account B, modifications become visible only after executing `END TRANSACTION`, ensuring an all-or-nothing update.

A database is considered transactional if it upholds the transaction paradigm, characterized by each transaction in the system having these four properties forming the acronym ACID: atomicity, consistency, isolation, and durability [53].

Atomicity refers to the all-or-nothing nature of a transaction described previously.

Consistency means that all completed transactions transform the database from one consistent state to another.

Isolation ensures that all changes within a single transaction must be hidden from all other transactions that are run concurrently.

Durability guarantees that after a transaction is committed, the effects of the transaction are permanently stored in the database, regardless of whether a system failure occurs later on.

2.3 SQLite

SQLite is an embedded, cross-platform, and compact SQL database engine [51]. By being embedded, SQLite can run directly within the application without needing a separate server process and inter-process communication. SQLite is an open-source database system that permits unlimited copying and modification of its source code without restriction. However, it is important to note that SQLite does not operate under an open-contribution model [35].

The codebase of SQLite is self-contained, meaning there are no external dependencies and only a few functions are required from the C standard library. When optimizing for binary size, the compiled SQLite library with all features enabled can be less than 750 KiB [37].

SQLite has become the most widely deployed database engine [51]. This widespread deployment can be attributed to several factors, such as its embedded design, cross-platform compatibility, reliability [38], and speed [32]. Every iOS and Android device, most personal computers, and even some automobiles have embedded SQLite in their software. However, having this wide range of deployments requires SQLite to be quite flexible, as workloads and hardware specifications can vary significantly. A study [57] analyzed the SQLite activity of both internal and user-installed services of 11 smartphones running the Android operating system. According to the study, the queries and access patterns produced differed from typical database server workloads.

2.3.1 Architecture

SQLite has a modular architecture design [31]. The following section briefly covers the internals of SQLite and their functions.

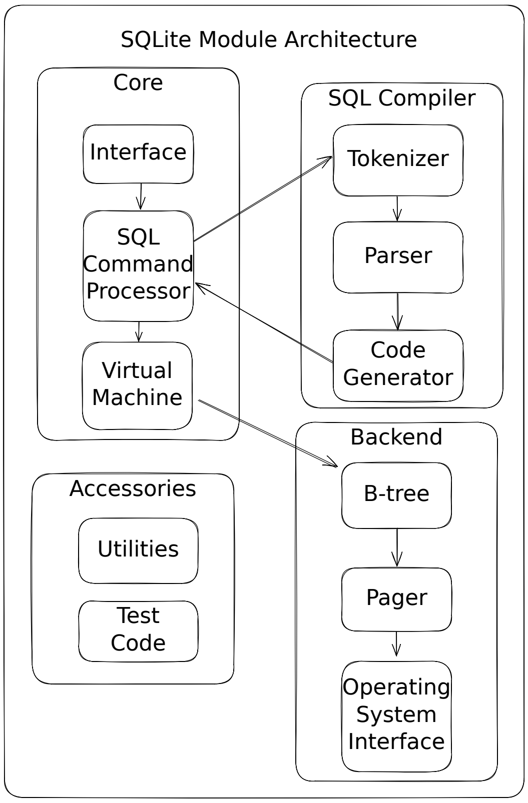

Figure 2: SQLite's module architecture.

The *core* modules consist of the interface, the SQL Command Processor, and the Virtual Machine, also known as the Virtual Database Engine (VDBE). The Interface module is the C-language interface used to communicate with the SQLite. When an SQL statement is passed in through a prepare interface, such as `sqlite3_prepare_v2()`, the SQL Command Processor module passes it to the *SQL Compiler* modules. The SQL Compiler modules tokenize, parse, and generate a bytecode program that implements a single SQL statement in a `sqlite3_stmt` object. This object is what the `sqlite3_step()` interface takes as input and passes the contained program to the VDBE. The VDBE executes the program, running it until completion, forming a results row to return, or experiencing a halt due to a fatal error or interruption.

The VDBE then interacts extensively with the *backend* modules, particularly with the B-tree module [51]. SQLite's database file consists primarily of different B-trees, one for each table and one for each index. A B-tree is a data structure that consists of a sorted collection of values sorted by a B-tree key. This key is derived from the underlying table or index. The sorted nature of the B-tree enables efficient traversal and locating of specific data rows. However, the gained efficiency depends on using the key of the B-tree for the search.

The B-trees are stored in fixed-size pages that the B-tree module requests from the Pager module. The Pager module is a page cache between the B-tree and the Operating System Interface modules. The Pager also ensures that modified pages are synced to stable storage. The Operating System interface interacts with the underlying file system implementation via an abstract object called a virtual file system, VFS for short. These objects have already been implemented for Unix and Windows operating systems in SQLite, but the presence of the abstract layer enables support extension to different systems.

2.3.2 Transactions

SQLite provides ACID guarantees for transactions, making it a transactional database [51]. The Pager module ensures the ACID guarantees hold for transactions [33]. The Rollback Journal and the Write-Ahead Log (WAL) are the two implementations that achieve these guarantees. Selecting which implementation, more commonly referred to as mode, SQLite should use is defined in the Database Header section of the database file. Modes other than Rollback Journal and WAL are currently not supported.

The Rollback Journal operates by writing changes directly to the database file and storing the original content in a separate file called the rollback journal file. When a transaction starts, it acquires a shared lock that allows the database to be read. The shared lock allows for the reading of the database, and many processes can hold it at once. The writing process, henceforth referred to as the writer, acquires a reserved lock if an `INSERT`, `UPDATE`, or `DELETE` statement is executed. Only a single process can hold the reserved lock at once. The reserved lock indicates that this transaction will eventually change the database. Eventually, when the transaction is to be committed, or the memory cache has filled up, the writer must acquire exclusive access to the database. Before this, the writer ensures all rollback journal data is written on disk to ensure that data is adequately rolled back on system failure. The writer then obtains a pending lock that prevents new readers from getting the shared lock and waits for all current shared locks to clear to obtain the exclusive lock. After ensuring the rollback journal's data is safely written to disk, the writer secures an exclusive lock, applies changes to the database, flushes these changes to disk, and then deletes the rollback journal file. The deletion of the rollback journal file is the instant when the commit happens. After this, the exclusive and pending locks will be released.

The WAL mode writes the new changes to a different file and keeps the original database file unchanged to improve read and write concurrency. When new changes are to be committed to the database, a special `COMMIT` record is appended to the WAL file. When reading the database, the reader must first check if the page it is trying to access is in the WAL file until reading the original. Each database reader stores a location of the last valid commit so that they can read the WAL file until that specific commit they originally started reading at. As the original database file hasn't changed and the WAL file has only been appended, the separate transactions are kept isolated. If other changes are to be made, they can be appended to the WAL file. When the WAL file grows to a certain point, usually to about 1000 pages, a checkpoint operation is triggered that transfers all the current transactions from the WAL file to the original database file.

The WAL mode provides significant improvements to the performance. This is achieved by reducing the amount of writes to stable storage, making the disk writes more sequential, and removing exclusive locks needed for each write operation. However, the WAL mode does come with some disadvantages. To significantly speed up readers' search for the WAL file, a separate WAL index is created and held in shared memory for processes to access. This speeds up reading transactions but requires all readers to run on the same system; thus, network file systems are unsupported.

3 Why edge databases are important

To understand why edge databases are important and what motivates their development, it is essential to understand the intersection of edge computing and databases, including their historical development. This section briefly covers edge computing and problems that arise with distributed systems before showing how they come together, specifically regarding edge databases in the next section.

3.1 Edge computing – an evolution of cloud computing

Edge computing differs from traditional cloud computing, representing a new computing model that processes data at the network's edge [65]. Edge networks are globally distributed systems, leveraging decentralized resources to bring computing tasks nearer to where the data is requested.

In essence, edge computing delivers services and performs calculations at the network's edge, where data is generated [48]. It brings the cloud's network, computing, and storage capabilities to the edge, providing intelligent services to meet the IT industry's needs for agile connectivity, real-time processing, data optimization, application intelligence, security, privacy, low latency, and high bandwidth.

The advantages and disadvantages of Edge computing manifest in several distinct ways:

Content delivery networks, an example use case and one of the earliest implementations of an edge computing network, are globally distributed networks of nodes with limited computation and storage capabilities focused on content delivery.

3.2 Content Delivery Networks

Revisiting traditional cloud computing and on-premise solutions is essential to comprehend the rationale behind Content Delivery Networks (CDNs). In terms of storage of static content, the current web ecosystem is dominated by providers operating within centrally controlled, public cloud infrastructures [56].

The centralization of these infrastructures means they are located in only a few places on the Internet. As a result, even simple network failures can cause these providers to become unavailable. Another consideration is that the Internet and its services operate globally, leading to significant variability in application latency and responsiveness depending on the user's geographical location.

CDNs represent a specific application of edge networks and computing. By strategically deploying storage and computational resources at the network's edge, CDNs aim to

Table 1: Edge computing advantages & disadvantages

Advantage/Disadvantage	**Description**
Lower latency	The proximity of data processing to the source reduces the distance data must travel over the network, decreasing latency. Charyyev et al. [49] observed that 58% of end-users can reach an edge-server location in less than 10 ms, with the same metric for cloud data centers being only 29%.
Reduced network bandwidth	By having data processing closer to the end user, the overall bandwidth usage for the global network infrastructure is reduced [65].
Increased availability & resilience	relying on a decentralized & highly distributed network of edge nodes can increase fault tolerance and availability of the application [68].
Management complexity	By being inherently distributed, edge computing infrastructures incur higher management complexity than centralized cloud systems [65].
Security challenges	Distributed edge infrastructures have variable perimeter security and a larger attack surface than centralized clouds, complicating uniform security management [71; 61].

address these issues by bringing content closer to end users. This globally distributed approach enhances availability, reduces latency, and improves application responsiveness.

A CDN has traditionally functioned as a low latency content cache for the end-user and the rest of the Internet [73]. This is achieved by having a global, distributed network of surrogate servers, i.e., proxy servers forwarding the requests to the origin servers and responding to the network request as close to the end-user device as possible (see Figure 3.

Depending on the rules configured during deployment or based on the response received from the origin server, the CDN will determine if and for how long the response should be stored in the local cache. Sending consequent requests that ”hit” the specific node will serve the locally cached content, with the request never reaching the server. This logic

Data centers, while fewer in number, possess significantly greater computing power compared to CDNs. For example, Amazon Web Services (AWS) operates slightly more than one hundred availability zones [1]. In contrast, Akamai’s CDN maintains 6100 explicit peering connections [70]. This disparity highlights the difference in infrastructure

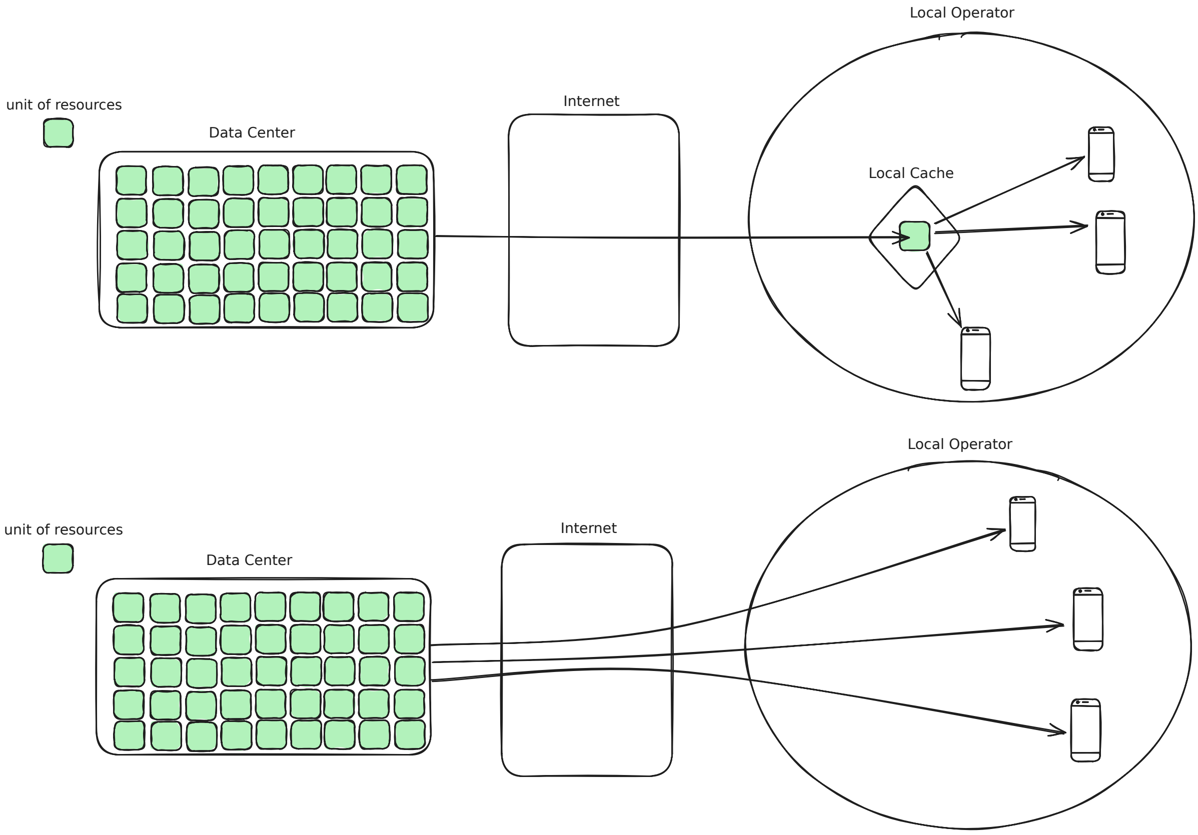

Figure 3: Information flow with and without a CDN [73]. After the initial request, the following requests hit the local cache, avoiding the full round-trip to the data center. Green boxes are used to give a sense of scale in resources available to a typical cloud data center, compared to a smaller, more local CDN.

scale and operational focus between centralized data centers and distributed CDNs. Certain cloud providers, these offerings utilize the underlying CDN infrastructure to enable general-purpose computing capabilities at the network edge [2; 5].

3.3 Serverless Edge computing

Although still modest compared to cloud data centers, global edge networks' computing and storage capabilities have progressively advanced, leading several cloud providers to offer Serverless Edge computing resources [67]. Serverless in this context refers to the cloud computing model where computation and storage are decoupled, code execution is automated without manual resource management, and billing is based on actual resource usage rather than pre-allocated resources [55]. Serverless platforms can have a wide variety of service offerings [3]. Serverless function platforms, termed Function-as-a-Service (FaaS), enable code execution without requiring server management.

The decoupled computing and storage architecture in serverless computing increases the complexity of embedded state management within the Function-as-a-Service (FaaS) paradigm [55]. Research efforts to address these challenges include the development of serverless file systems [66], moving computation closer to data storage [72], and improving serverless runtimes with native I/O module support [50].

4 Distributed Systems

This section describes theoretical constructs in distributed database management systems. The principal challenge is maintaining data consistency between nodes.

4.1 CAP Theorem meets ACID

Brewer [47] proposed a conjecture stating that it is impossible for a web service to simultaneously guarantee Consistency, Availability, and Partition tolerance (collectively known as CAP). Despite these properties being desirable and expected in real-world web services, Brewer's conjecture suggests that achieving all three at once may not be feasible [52]. The CAP theorem sparked the NoSQL movement and fueled a debate about trade-offs in data systems, challenging the prevailing belief that strong consistency is essential for databases. However, the CAP theorem faces criticisms and limitations, including ambiguous and conflicting interpretations, and it lacks a comprehensive framework for detailing these trade-offs [54].

While ACID properties ensure reliability within a database (see Subsection 2.2), distributed systems like Edge Computing networks encounter challenges when it comes to managing application state while balancing between consistency, availability, and partition tolerance between nodes, in 2015, Kong et al. [58] observed that, in conjunction with the CAP theorem, strict adherence to ACID properties in distributed systems is infeasible, requiring a trade-off that balances consistency, availability, and partition tolerance based on system priorities.

The CAP theorem shows that distributed systems can't fully guarantee consistency, availability, and partition tolerance simultaneously. To manage this, different consistency models offer ways to balance these limits based on the system's needs.

4.2 Consistency Model

A consistency model characterizes the behavior of data synchronization across nodes in a distributed system [69]. Various consistency models are utilized to manage data synchronization, each tailored to meet specific consistency requirements dictated by the system's operational needs.

On the other hand, an isolation level is a database concept that determines how transactions are managed to prevent conflicts or anomalies (like dirty reads or phantom reads) within a single database system. Common isolation levels include Read Uncommitted, Read Committed, Repeatable Read, and Serializable [20]. For instance, PostgreSQL, an open-source and widely used SQL database system, defaults to the read committed isolation level but can be configured to employ more stringent consistency modes [22]. Similarly, SQLite implements Snapshot Isolation for transactions in Write-Ahead Logging (WAL) mode [34].

4.3 Distributed Consensus

Distributed Consensus is a fundamental challenge in distributed systems. Multiple nodes must agree on the absolute state of the application while considering things such as software failures or network partitions, which are network disconnects between nodes. Exemplified by the Byzantine Generals Problem [60], The problems to consider include asynchronous communication, messaging delays, and node failures.

Algorithms like Paxos [59] and Raft [62] address these challenges by providing mechanisms for achieving consensus, prioritizing consistency, and fault tolerance. Nevertheless, as indicated by the CAP theorem, consensus often necessitates a trade-off, potentially reducing system availability during network partitions, with certain nodes becoming unable to respond.

4.3.1 Raft Consensus Protocol

The Raft consensus protocol is a distributed algorithm that manages a replicated log across multiple servers, ensuring consistency and fault tolerance in distributed systems. It simplifies reaching agreement among a group of nodes by dividing responsibilities into three roles: leader, follower, and candidate[1] [62]. The leader handles client requests and log replication, while followers accept and apply updates from the leader. Candidates emerge during leader elections when the current leader fails, ensuring continuous operation. Raft prioritizes understandability and decomposes consensus into leader election, log replication, and safety mechanisms, making it easier to implement and verify than comparable protocols like Paxos.

4.4 Synchronization of Distributed State

Different levels of consistency can implement the synchronization of state changes between replicated nodes of the same state machine. For this study, we introduce and focus on two different replication levels for a single primary node configuration: asynchronous and synchronous replication. Choosing between these consistency models constitutes the difference between the applied CAP theorem paradigm.

[1] A good visualization of Raft protocol https://thesecretlivesofdata.com/raft/

4.4.1 Asynchronous replication

Asynchronous replication is a data replication technique in distributed systems where the primary node, responsible for handling write operations, does not wait for the acknowledgment from secondary nodes regarding the successful commitment of those writes. Instead, the primary node processes subsequent transactions without ensuring the replicas have fully synchronized. In these cases, secondary nodes are usually left responsible for synchronizing the state by polling the primary node for updates on an interval or during other specified circumstances.

This method improves write throughput by reducing the latency of waiting for acknowledgment from secondary nodes. However, it compromises both durability and consistency. A temporal gap emerges wherein the state of the secondary nodes lags behind the primary, reflecting an outdated version of the database [13]. This introduces the risk of data loss in scenarios where the primary node experiences a failure before the secondary nodes have completed the replication process. Additionally, asynchronous replication does not inherently provide real-time guarantees regarding the database state before initiating a failover mechanism (if such a mechanism is supported). This creates challenges in maintaining a consistent and durable system state during failover or recovery events.

4.4.2 Synchronous replication

Synchronous replication, in contrast, is a replication strategy in which write operations are committed to all participating nodes in a distributed system before the primary node advances to subsequent transactions. This guarantees that all replicas reflect the same consistent state upon the successful completion of each transaction. Specifically, the primary node must receive an acknowledgment from all secondary nodes, confirming that the data has been successfully written and committed across the system.

This approach significantly enhances data durability and consistency by ensuring that no data is lost in the event of node failures and that all nodes maintain a consistent view of the data. However, it introduces notable trade-offs, including increased latency and decreased write throughput. These drawbacks arise from the additional communication overhead and the need for coordination and acknowledgment between nodes, which are inherently time-consuming processes, particularly in geographically distributed systems.

These replication methods — synchronous and asynchronous — are not mutually exclusive and can coexist within the same system or software, depending on the implementation and the level of configurability provided by the system, such as transaction control in a database.

5 SQLite on the Edge

SQLite on the Edge provides a potentially highly distributed, relational database that can be near to end-users.

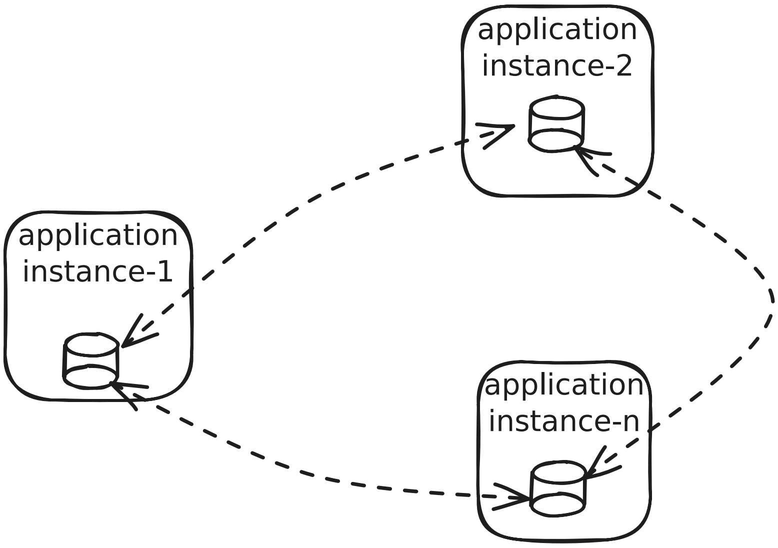

Figure 4: Architecture for an ideal, embedded SQLite application [10]. The application instance nodes are aware of each other to handle the leader election and data synchronization.

It would be ideal to run SQLite as intended, embedded within the process of our business application (see Figure 4). As highlighted earlier in Subsection 2.3, SQLite is open-source, not open-contribution. To maintain its public domain status, SQLite enforces a strict contribution policy. This policy has led to multiple adaptations [2] [3] that are not official SQLite but have different names and are based on it.

Compared to traditional, non-distributed databases, achieving these objectives introduces several trade-offs. High tolerance encompasses the development of infrastructure necessary for orchestrating communications between nodes. Additionally, it involves managing backup data for these primary nodes and synchronizing data and schema across subsidiary nodes. This complexity underscores the intricacies of maintaining system integrity and operational continuity in distributed database environments.

Below, available implementations are analyzed in the context of the following research questions:

- **Consistency guarantees**: Examine how each implementation ensures data consistency under different replication modes and their trade-offs.

[2] https://github.com/tursodatabase/libsql
[3] https://github.com/superfly/litefs

- **Performance considerations**: Assess the impact of various implementations on performance, such as latency and throughput.
- **Embeddability**: Investigate the ease of embedding these implementations in different environments and applications.
- **Viability for Serverless environments**: Analyze how well these implementations align with the characteristics and requirements of serverless architectures.
- **Suitability for global edge deployments**: Explore their effectiveness in scenarios requiring distributed edge computing.
- **SQLite compatibility**: Compare their levels of compatibility with SQLite's API and the implications for existing applications.

A traditional SQLite database deployment does not support distributed synchronization. However, as SQLite has a VFS layer, implementing a distributed file system underneath SQLite enables out-of-the-box support for distributed data synchronization. This means that for distributed SQLite to be possible directly for the official version SQLite, one needs to provide their own distributed filesystem and have that manage the synchronization of the disk write and read operations. Existing solutions implementing this already exist.

Among the available solutions, four candidates were selected for further investigation: FaasFS, LiteFS, libSQL, and rqlite. Detailed analyses of these approaches will follow.

5.1 FaaSFS

Schleier-Smith et al. [66] investigated the design and implementation of a POSIX-compliant file system API for FaaS platforms, leveraging cloud functions' fault tolerance to use local state with restart mechanisms for failed commits optimistically. Function boundaries enable implicit commit and rollback, allowing transaction processing without altering APIs, thus extending serverless scalability and simplicity to stateful applications with minimal modification.

FaaSFS achieves a strict serializable consistency model for the file system, ensuring that all operations appear to execute atomically in a total order consistent with real-time. This is achieved through a combination of mechanisms, including speculative local execution with optimistic lock elision and commit-time validation to enforce transaction isolation. Fine-grained cache updates ensure precise consistency at the block level, leveraging the transactional backend to track and manage changes efficiently. These mechanisms collectively ensure that FaaSFS maintains strong consistency while optimizing performance.

As FaaSFS implements a distributed file system, it is fully SQLite-compatible and is not limited to only SQLite-specific use cases.

Using SQLite as a benchmark, their prototype solution demonstrated throughput gains of up to 30x compared to Network File System (NFS). However, increased database transaction abort rates were observed under high concurrency and write-heavy situations. In contrast, the solution displayed promising results in read-intensive workloads, underscoring its effectiveness for read-dominant applications.

The prototype utilized a monolithic, in-memory backend server to manage the storage layer, which the authors identified as a key area for future development to enhance scalability and robustness [66]. This limits the suitability of using FaaSFS in a production setting. As this was a proof of concept, no considerations regarding a globally distributed deployment was not discussed, as it was out of scope.

5.2 LiteFS

LiteFS is a distributed file system that transparently replicates the underlying file system of SQLite databases. This enables applications to interact with SQLite databases as local on-disk instances, while LiteFS replicates the entire file system across all nodes in the cluster [12]. It functions as a passthrough file system, intercepting writes to SQLite databases to detect transaction boundaries [18]. Changes are recorded at the transaction level in Lite Transaction File (LTX) files, ensuring precise replication and consistency across the cluster.

Due to the usage of the distributed file system FUSE, LiteFS is limited to approximately 100 write transactions per second [14]. This level of performance is less than desired, as SQLite supports transactions per second in the magnitude of tens of thousands [51]. LiteFS is a valid option for globally distributed edge storage, read-heavy applications with weaker consistency guarantees. Similar to FaaSFS, LiteFS could, in theory, be used in Serverless environments, but there is currently no support for it [45].

Data tolerance with a distributed file system can be achieved by taking snapshots of the entire file system or incremental snapshots of the changes made to the file system. LiteFS uses the second option by using the LTX file format to act as a packaging format for the SQLite transaction changesets, similar to the WAL-file, but optimized for use in replicated systems [18].

LiteFS employs a single primary node architecture for handling write operations, with the primary node replicating data to replica nodes to optimize read performance. This setup introduces two application challenges: write requests must be directed to the primary node to maintain data consistency, and read requests must wait until prior writes have been fully replicated to ensure accurate data retrieval from replica nodes.

Configuring write-forwarding to the primary node responsible for the writing is not seamless and requires additional setup on the application side [11]. For web applications that follow HTTP method semantics, this means setting up proxying for all modifying operations, i.e., POST, PUT, PATCH, and DELETE. For self-hosted LiteFS implementations, read-your-writes semantics are not guaranteed.

By directly designating the primary node to an application instance, LiteFS eliminates the need for data synchronization and write operations over the network for that specific instance. This increases the load applied to this instance, as it has now become the "source of truth" for other nodes responsible for both synchronization and write operations coming from the other nodes.

5.3 libSQL

libSQL [15], a fork of SQLite, implements an alternate approach of replication by replicating the Write-Ahead log by virtualizing the Write-Ahead log, similar to the VFS layer in SQLite itself. It remains fully compatible with SQLite's file format, API, and embeddability while adding features like client-server support.

5.3.1 Replicating the WAL

As outlined in subsubsection 2.3.1, SQLite employs two mechanisms for handling write operations: the Rollback Journal and the WAL. The WAL is particularly critical for enabling efficient database replication. The WAL system defers the immediate recording of write operations directly to the database file, resulting in a predominantly static dataset with minimal changes. This approach of appending data exclusively to the write-ahead log ensures transaction isolation, as transactions consistently begin by reading from the same starting point.

Moreover, the append-only nature of the write-ahead log facilitates the synchronization of the database across distributed systems. This synchronization is achieved with minimal communication overhead, as the only data transmitted between systems comprises the new entries added to the log. This efficient synchronization mechanism reduces database replication's complexity and bandwidth requirements while enabling fully featured SQLite replication.

libSQL implements replication synchronization with a separate, centralized database instance (see Figure 5) as the primary node responsible for writing and synchronization by providing WAL updates. The synchronization is asynchronous, with no current methods of configuring this further. Unlike LiteFS, which is constrained by the limitations of

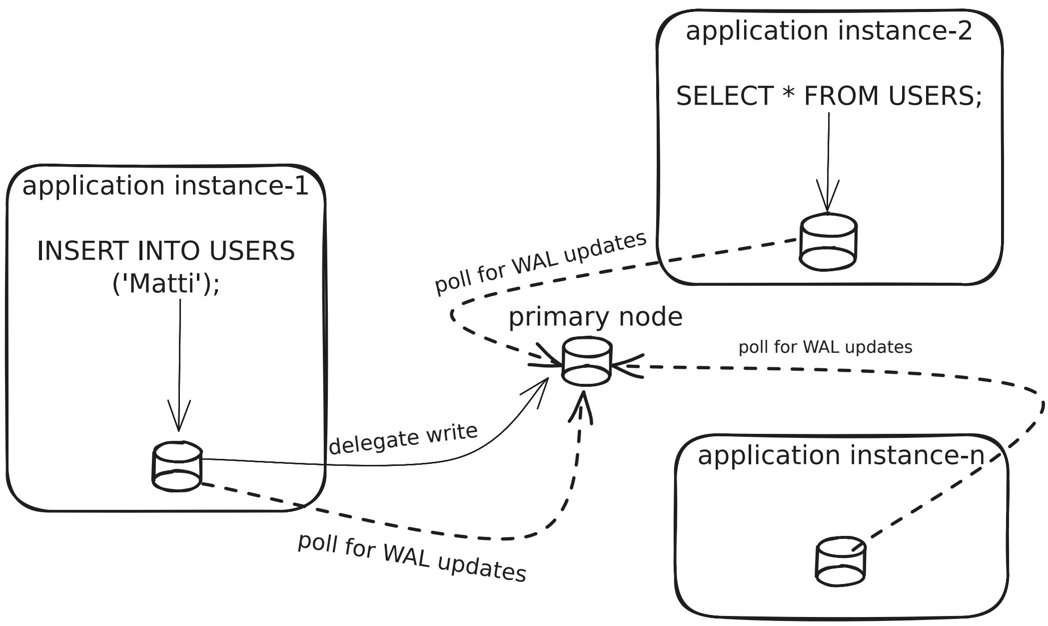

Figure 5: libSQL architecture [17] adapted to use the Embedded Replicas [43] feature. The embedded databases within application instances will execute all reads against the local database, while delegating the writes to the primary node over the network. The local databases are updated by polling the primary node for WAL updates.

FuseFS due to its reliance on a custom file system implementation, libSQL operates independently of such constraints. This architectural distinction suggests that libSQL may achieve substantially higher write performance.

5.3.2 libSQL Architecture

libSQL supports running geographically distributed, replicated secondary nodes with write-forwarding or embedding the entire database within the application process, usually done during the startup phase of the application [43] (see Figure 5). The embedded option is designed only for long-running tasks. In contrast, short-lived compute instances, such as serverless functions, require a remote connection to the primary or a replicated node.

For both embedded and replicated secondary nodes, write-forwarding to the primary node is seamless [16]. It provides read-after-write consistency [42], meaning that successful writes are immediately visible to the same database connection that initially sent the writes.

5.4 rqlite

rqlite [27] is an open-source distributed Relational Database Management System (RDBMS) with a client-server architecture. It leverages SQLite as its storage engine and employs the Raft Consensus Protocol (see subsubsection 4.3.1) to ensure data replication and consistency [30]. The system was initially designed to enhance the reliability of SQLite databases while preserving their simplicity and lightweight characteristics, in contrast to the greater complexity of relational database management systems (RDBMS) such as MySQL and PostgreSQL.

Unlike SQLite, rqlite almost exclusively[4] offers a custom HTTP-based API for database queries and modifications, thereby not supporting embeddability [24]. This API-centric design does, however, make it suitable for use with FaaS environments, provided the database itself is deployed on long-running tasks. The HTTP-API also supports several read consistency options, configurable separately for each query, providing varying levels of read consistency [25].

The replication of a rqlite cluster is facilitated by its Raft consensus layer, which generates and maintains a Raft log [28]. This log is an authoritative, sequential record of all committed SQLite commands executed within the system. It ensures consistency by storing the order of these operations and, depending on read consistency configurations, may also include entries for read-only queries. As every node in the rqlite cluster applies the log entries identically, this mechanism guarantees that the SQLite database remains synchronized and consistent across all nodes in the cluster.

The Raft Consensus Protocol enforces strong consistency by requiring all writes to achieve consensus before being committed [24]. This process introduces substantial overhead for write operations, as consensus demands two network round trips between the leader and other nodes to reach quorum. However, these communications can occur in parallel. When network latency increases, such as in a globally distributed system, write performance degrades significantly, with throughput dropping to approximately ten transactions per second [19]. Performance improvements are possible through techniques such as queuing writes, but these come at the cost of reduced durability [63].

rqlite supports read-only nodes to enhance read scalability or distribute data closer to clients [26]. These nodes are non-voting and do not participate in Raft consensus or leader elections. They receive committed log entries from the Leader and update their SQLite databases accordingly. Lower read consistency levels enable low-latency data retrieval, which is particularly beneficial for serverless edge functions. This approach balances the need for rapid access with the flexibility to enforce stricter consistency mechanisms, depending on the specific operational demands and data integrity needs.

[4] Reading the SQLite file as read-only is technically possible, but it is not an intended use case [23]

SQLite's transaction API is not supported in rqlite due to undefined behavior during a cluster failure in an ongoing manual transaction [29]. However, the HTTP API includes a custom `transaction` query parameter that ensures atomicity for transactional operations.

5.5 Summary

None of the selected implementations completely meet the ideal architecture described in Figure 4. The feature sets of each adaption concerning our research questions are summarized in Table 2.

Table 2: Summary of Distributed SQLite Systems

Feature	**LiteFS**	**libSQL**	**rqlite**	**FaaSFS**
Consistency	Async (Eventual Consistency)	Async (Eventual Consistency)	Strong (Raft) with flexibility	Strict Serializability
Performance	100 writes/sec	High	High reads, 10 writes/sec	High reads, increased abort rates with concurrent writers
Embeddability	Yes	Yes	No	Yes
Serverless	No	Client-Server	Client-Server	Yes
Edge Suitability	Read-heavy	Yes	No	N/A
SQLite compatibility	Full	Full	Limited	Full

LiteFS aligns closely with the ideal architecture. However, its foundational implementation limits its write performance. Additionally, LiteFS lacks strong consistency guarantees, does not support synchronous replication, which is critical for certain applications, is unsuitable for serverless use cases, and relies partly on HTTP semantics for synchronization.

In contrast, libSQL provides performance similar to SQLite but diverges from the ideal architecture. It necessitates an external primary node and relies on replicated nodes for serverless deployments, which introduces complexity and deviates from the architectural goals.

On the other hand, rqlite offers robust and flexible consistency guarantees. Despite this advantage, its write performance suffers significantly, especially in globally distributed environments. The performance degradation stems from the overhead introduced by raft-based replication, which can be substantial.

Lastly, FaasFS demonstrates some potential for addressing the issues regarding serverless environments. However, it currently lacks a backend implementation. Moreover, its suitability for edge environments has not been evaluated, as this consideration was outside the scope of the study it is based on.

Regarding database backup: libSQL, LiteFS, and rqlite support database file backups with either directly setting up or an implementation similar to Litestream [21], which copies write-ahead log pages from disk to one or more replicas to network storage, e.g., AWS S3, Google Cloud Storage, or NFS. As FaaSFS has no proper backend storage solution, it is arguably not applicable.

6 Discussion

The adaptations previously analyzed have made different design decisions regarding the CAP theorem, which affect the behavioral characteristics of these solutions when deployed and also impact their suitability to serverless use cases.

6.1 CAP Theorem applied to Globally distributed SQLite

The CAP theorem discussed in Subsection 4.1 states that distributed systems cannot simultaneously achieve consistency (C), availability (A), and partition tolerance (P). This trade-off is particularly significant for global edge deployments in distributed SQLite, where performance and low latency are critical.

Raft-based systems like rqlite prioritize consistency and partition tolerance (CP). However, this comes at the expense of availability and performance. For globally distributed setups, the severe write latency observed in rqlite raises questions about the practicality of the CP model for edge deployments, where fast and reliable access is essential.

On the other hand, solutions like LiteFS and libSQL favor availability and partition tolerance (AP), providing better write performance by relaxing strict consistency guarantees. This approach aligns more effectively with the needs of read-heavy edge applications, where high availability and low-latency reads are critical.

Enhancing LiteFS and libSQL with support for stricter isolation levels at the transaction level could be a viable solution to address the need for stronger consistency in specific scenarios. By enabling configurable synchronous replication for critical operations, these systems could offer a flexible balance between performance and consistency tailored to application requirements.

An alternative approach could involve an rqlite-based AP-model, where Raft nodes are localized to improve write performance, while globally distributed read-only nodes serve edge locations. These replicas, synchronized via the Raft log, would offer low-latency access and could be integrated with serverless edge functions using HTTP. Query-level consistency controls would allow a flexible balance between performance and data integrity in edge deployments.

6.2 Embedding SQLite to Serverless Edge

The limitations of serverless runtimes currently constrain the viability of embedding SQLite in serverless edge environments. To achieve practicality, these runtimes need substantial enhancements. A client-server architecture with replicated databases offers a more realistic approach. Options like rqlite and libSQL can be selected based on specific use cases. However, rqlite struggles with write performance under high-latency conditions, making it less suitable for write-intensive, globally distributed systems. libSQL improves performance but presents trade-offs in configurability for consistency and deployment requirements.

LiteFS shows potential for embedding SQLite in Function-as-a-Service (FaaS) platforms, but it is not currently supported, and its development does not seem fully aligned with this goal. Similarly, FaaSFS faces comparable limitations to LiteFS, with the additional drawback of lacking a fully implemented storage backend, further hindering its usability in this context.

6.3 Other considerations

While doing research, other adaptations of SQLite with similar goals were encountered: Dqlite [9], Cloudflare D1 [6], and SQLite Cloud [39] that were discarded due to the following reasons:

Dqlite is an open-source C library enabling distributed SQLite [9]. Like rqlite 5.4, it uses the Raft Consensus Protocol but employs the Write-Ahead Log (WAL) as its replication log, enhancing performance and supporting SQLite's transaction API and deterministic functions. This brings it closer to the ideal architecture than LiteFS, with stronger consistency guarantees. However, it was excluded from this study to limit its scope.

For the initial implementation method, Cloudflare D1 used the distributed file system approach by utilizing Cloudflare Durable Objects [7], proprietary technology for achieving distributed consistent storage available on the Cloudflare platform, under-the-hood [4]. The underlying implementation has changed, and current details are unknown; most notably, it does not seem to be global read replication support yet, but only as a promised feature to come [8]. Missing the global read replication makes it unready as an alternative to consider during this time. Regarding write performance, evaluations were conducted solely compared to the earlier implementation, without specific quantitative metrics.

SQLite Cloud seems to implement synchronization that relies on a customized version of the Raft Consensus Protocol [41; 40]. Still, not much else is known as the code and architecture design are not open-source or documented. Considering the limited write performance of the Raft-based solution like rqlite discussed in 5.4, the write performance of SQLite Cloud in a globally distributed setting remains an open question, warranting further investigation to evaluate its efficiency under similar constraints.

7 Conclusion

This study proposed an ideal architecture for running embedded, distributed SQLite databases within edge applications. Existing SQLite adaptations were evaluated with the compatibility with edge and serverless environments. The analysis highlighted differences between current solutions with their trade-offs, with notice of different behavioral characteristics based on their consistency models, as dictated by the CAP theorem.

7.1 Main findings

Systems such as rqlite that utilize distributed consensus protocols prioritize consistency and partition tolerance. They are well suited for scenarios and applications that require the highest data integrity while making significant sacrifices for performance. This contrasts libSQL and LiteFS, which apply looser consistency guarantees to achieve higher performance. Depending on application requirements, both models might be applicable.

For edge applications, architectures that enforce strong consistency and partition tolerance tend to incur significant performance overhead due to the challenges of maintaining strict data integrity across distributed nodes. In contrast, systems with relaxed consistency models often achieve better performance but may fall short when workloads demand robust consistency guarantees. Balancing and understanding these trade-offs is crucial when designing and developing edge applications.

7.2 Future work

The current body of performance studies is limited, with many observations lacking sufficient support, especially regarding concrete write performance. Conducting detailed studies on identical hardware could provide concrete evidence to validate findings, enabling accurate performance comparisons and more reliable benchmarks. This would strengthen current hypotheses and offer actionable insights for optimizing deployment strategies.

Further exploration of consistency and partition tolerance as a model for global edge deployments is also essential. Research into its real-world feasibility and the infrastructure requirements and complexity of different setups could identify key trade-offs affecting scalability, reliability, and cost. These insights would guide more effective edge computing architectures.

Investigating the possibilities of increasing the configurability of consistency models for LiteFS and libSQL could improve their usability in specific use cases and help adaptability.

References

[1] Availability Zones - AWS Fault Isolation Boundaries — docs.aws.amazon.com. https://docs.aws.amazon.com/whitepapers/latest/aws-fault-isolation-boundaries/availability-zones.html, . [Accessed 27-05-2024].

[2] Edge Computing| CDN, Global Serverless Code, Distribution | AWS Lambda@Edge — aws.amazon.com. https://aws.amazon.com/lambda/edge/, . [Accessed 22-11-2024].

[3] Serverless Computing – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/serverless/, . [Accessed 28-11-2024].

[4] Announcing D1: our first SQL database — blog.cloudflare.com. https://blog.cloudflare.com/introducing-d1/, . [Accessed 26-06-2024].

[5] How the Cache works | Cloudflare Workers docs — developers.cloudflare.com. https://developers.cloudflare.com/workers/reference/how-the-cache-works/, . [Accessed 22-11-2024].

[6] Build a natively serverless SQL database with Cloudflare D1 — cloudflare.com. https://www.cloudflare.com/developer-platform/d1/, . [Accessed 26-06-2024].

[7] Durable Objects from Cloudflare — cloudflare.com. https://www.cloudflare.com/developer-platform/durable-objects/, . [Accessed 26-06-2024].

[8] D1: We turned it up to 11 — blog.cloudflare.com. https://blog.cloudflare.com/d1-turning-it-up-to-11, . [Accessed 26-06-2024].

[9] Canonical Dqlite | Canonical Dqlite — dqlite.io. https://dqlite.io/, . [Accessed 01-12-2024].

[10] Canonical Dqlite | Canonical Dqlite — dqlite.io. https://dqlite.io/docs/explanation/architecture, . [Accessed 26-11-2024].

[11] Built-in HTTP Proxy — fly.io. https://fly.io/docs/litefs/proxy/#how-it-works, . [Accessed 01-12-2024].

[12] LiteFS - Distributed SQLite — fly.io. https://fly.io/docs/litefs/, . [Accessed 25-06-2024].

[13] How LiteFS Works — fly.io. https://fly.io/docs/litefs/how-it-works/#handling-split-brain, . [Accessed 26-06-2024].

[14] LiteFS FAQ — fly.io. https://fly.io/docs/litefs/faq/#what-are-the-tradeoffs-of-using-litefs, . [Accessed 26-06-2024].

[15] GitHub - tursodatabase/libsql: libSQL is a fork of SQLite that is both Open Source, and Open Contributions. — github.com. https://github.com/tursodatabase/libsql, . [Accessed 25-06-2024].

[16] libsql/docs/DESIGN.md at main · tursodatabase/libsql — github.com. https://github.com/tursodatabase/libsql/blob/main/docs/DESIGN.md, . [Accessed 01-12-2024].

[17] libsql/docs/sqld-overview.png at main · tursodatabase/libsql — github.com. https://github.com/tursodatabase/libsql/blob/main/docs/sqld-overview.png, . [Accessed 25-06-2024].

[18] litefs/docs/ARCHITECTURE.md at main · superfly/litefs — github.com. https://github.com/superfly/litefs/blob/main/docs/ARCHITECTURE.md, . [Accessed 25-06-2024].

[19] rqlite: The lighweight, distibuted database built on Raft and SQLite — docs.google.com. https://docs.google.com/presentation/d/1Q8lQgCaODlecHa2hS-OeMGoij22oMZ_xhyeRTrQnn6o/edit#slide=id.gec8333eb5a_0_0. [Accessed 01-12-2024].

[20] ISO/IEC 9075-1:2023 — iso.org. https://www.iso.org/standard/76583.html. [Accessed 01-11-2024].

[21] How it works - Litestream — litestream.io. https://litestream.io/how-it-works/. [Accessed 26-06-2024].

[22] 13.2. Transaction Isolation — postgresql.org. https://www.postgresql.org/docs/current/transaction-iso.html#XACT-READ-COMMITTED. [Accessed 17-06-2024].

[23] Directly accessing SQLite — rqlite.io. https://rqlite.io/docs/guides/direct-access/, . [Accessed 01-12-2024].

[24] FAQ — rqlite.io. https://rqlite.io/docs/faq, . [Accessed 01-12-2024].

[25] Read Consistency — rqlite.io. https://rqlite.io/docs/api/read-consistency/, . [Accessed 01-12-2024].

[26] Read-only nodes — rqlite.io. https://rqlite.io/docs/clustering/read-only-nodes/, . [Accessed 01-12-2024].

[27] rqlite — rqlite.io. https://rqlite.io/, . [Accessed 28-11-2024].

[28] rqlite Design — rqlite.io. https://rqlite.io/docs/design/, . [Accessed 01-12-2024].

[29] Accessing rqlite — rqlite.io. https://rqlite.io/docs/api/api/#transactions, . [Accessed 01-12-2024].

[30] https://philipotoole.com/replicating-sqlite-using-raft-consensus/, . [Accessed 01-12-2024].

[31] Architecture of SQLite — sqlite.org. https://www.sqlite.org/arch.html, . [Accessed 17-03-2024].

[32] 35% Faster Than The Filesystem — sqlite.org. https://www.sqlite.org/fasterthanfs.html, . [Accessed 17-03-2024].

[33] File Locking And Concurrency In SQLite Version 3 — sqlite.org. https://www.sqlite.org/lockingv3.html, . [Accessed 19-03-2024].

[34] Isolation In SQLite — sqlite.org. https://www.sqlite.org/isolation.html, . [Accessed 17-06-2024].

[35] SQLite Copyright — sqlite.org. https://www.sqlite.org/copyright.html, . [Accessed 26-06-2024].

[36] SQLite Home Page — sqlite.org. https://www.sqlite.org/, . [Accessed 26-06-2024].

[37] SQLite Library Footprint — sqlite.org. https://www.sqlite.org/footprint.html, . [Accessed 17-03-2024].

[38] How SQLite Is Tested — sqlite.org. https://www.sqlite.org/testing.html, . [Accessed 17-03-2024].

[39] SQLite Cloud - Share, scale and deploy SQLite databases. — sqlitecloud.io. https://sqlitecloud.io/, . [Accessed 26-06-2024].

[40] Scaling | SQLite Cloud Docs — docs.sqlitecloud.io. https://docs.sqlitecloud.io/docs/scaling, . [Accessed 25-11-2024].

[41] Strong Consistency with Raft and SQLite — blog.sqlitecloud.io. https://blog.sqlitecloud.io/strong-consistency-with-raft-and-sqlite, . [Accessed 26-06-2024].

[42] Data & Connections - Turso — docs.turso.tech. https://docs.turso.tech/data-and-connections, . [Accessed 01-12-2024].

[43] Introducing Embedded Replicas: Deploy Turso anywhere — turso.tech. https://turso.tech/blog/introducing-embedded-replicas-deploy-turso-anywhere-2085aa0dc242, . [Accessed 25-06-2024].

[44] x.com — x.com. https://x.com/leeerob/status/1780705942734331983?lang=en. [Accessed 26-06-2024].

[45] Thanks! I thought DonutDB was an awesome approach too. We do have plans for supp... | Hacker News — news.ycombinator.com. https://news.ycombinator.com/item?id=36605842. [Accessed 26-11-2024].

[46] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade and V. Watson. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976. ISSN 0362-5915, 1557-4644. doi: 10.1145/320455.320457. URL https://dl.acm.org/doi/10.1145/320455.320457.

[47] Eric A Brewer. Towards robust distributed systems. *PODC*, volume 7, pages 343–477. Portland, OR, 2000.

[48] Keyan Cao, Yefan Liu, Gongjie Meng and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020. doi: 10.1109/ACCESS.2020.2991734.

[49] Batyr Charyyev, Engin Arslan and Mehmet Hadi Gunes. Latency comparison of cloud datacenters and edge servers. *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–6, 2020. doi: 10.1109/GLOBECOM42002.2020.9322406.

[50] Pekka Enberg, Sasu Tarkoma, Jon Crowcroft and Ashwin Rao. Serverless runtime / database co-design with asynchronous i/o. *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '24, page 49–54, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705397. doi: 10.1145/3642968.3654821. URL https://doi-org.libproxy.aalto.fi/10.1145/3642968.3654821.

[51] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy and Jignesh M. Patel. SQLite: Past, present, and future. *Proceedings of the VLDB Endowment*, 15(12):3535–3547, August 2022. ISSN 2150-8097. doi: 10.14778/3554821.3554842.

[52] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL https://dl.acm.org/doi/10.1145/564585.564601.

[53] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. ISSN 0360-0300, 1557-7341. doi: 10.1145/289.291.

[54] Rajat Jain, Pradnya Borkar, Priyanshu Deshmukh, Sagarkumar Badhiye, Kritika Nimje and Kapil Gupta. Choosing a Suitable Consensus Algorithm for Blockchain Applications: A Review of Factors and Challenges. *International Journal of Intelligent Systems and Applications in Engineering*, 12(10s):333–341, January 2024. ISSN 2147-6799. URL https://ijisae.org/index.php/IJISAE/article/view/4381. Number: 10s.

[55] Eric Jonas, Anurag Khandelwal, Karl Krauth, Johann Schleier-Smith, Qifan Pu, Neeraja Yadwadkar, Ion Stoica, Vikram Sreekanti, Vaishaal Shankar, Joseph E Gonzalez, David A Patterson, Chia-Che Tsai, Joao Carreira and Raluca Ada Popa. Cloud Programming Simplified: A Berkeley View on Serverless Computing.

[56] Navin Keizer, Onur Ascigil, Michal Król, Dirk Kutscher and George Pavlou. A survey on content retrieval on the decentralised web. *ACM Comput. Surv.*, 56(8), apr 2024. ISSN 0360-0300. doi: 10.1145/3649132. URL https://doi.org/10.1145/3649132.

[57] Oliver Kennedy, Jerry Ajay, Geoffrey Challen and Lukasz Ziarek. Pocket Data: The Need for TPC-MOBILE. In *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things*, Raghunath Nambiar and Meikel Poess, editors, volume 9508, pages 8–25. Springer International Publishing, Cham, 2016. ISBN 978-3-319-31408-2 978-3-319-31409-9. doi: 10.1007/978-3-319-31409-9_2.

[58] Chao Kong, Ming Gao, Weining Qian, Minqi Zhou, Xueqing Gong, Rong Zhang and Aoying Zhou. Acid encountering the cap theorem: Two bank case studies. *2015 12th Web Information System and Application Conference (WISA)*, pages 235–240. IEEE, 2015.

[59] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2): 133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi-org.libproxy.aalto.fi/10.1145/279227.279229.

[60] Leslie Lamport, Robert Shostak and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. ISSN 0164-0925. doi: 10.1145/357172.357176. URL https://doi-org.libproxy.aalto.fi/10.1145/357172.357176.

[61] Aleksandr Ometov, Oliver Liombe Molua, Mikhail Komarov and Jari Nurmi. A survey of security in cloud, edge, and fog computing. *Sensors*, 22(3):927, 2022. doi: 10.3390/s22030927.

[62] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[63] Philip O'Toole. rqlite 7.5.0: Trading durability for write performance. `https://philipotoole.com/rqlite-trading-durability-for-performance/`. [Accessed 01-12-2024].

[64] Sang Hyuk Son. Synchronization of replicated data in distributed systems. *Information Systems*, 12(2):191–202, 1987. ISSN 0306-4379. doi: https://doi.org/10.1016/0306-4379(87)90043-3. URL `https://www.sciencedirect.com/science/article/pii/0306437987900433`.

[65] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1): 30–39, 2017. doi: 10.1109/MC.2017.9.

[66] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton and Joseph M. Hellerstein. A FaaS File System for Serverless Computing, September 2020.

[67] Mikael Siidorow. Survey of serverless edge computing for web applications. Bachelor's thesis, Aalto University, 05 2024.

[68] Maria Trigka and Elias Dritsas. Edge and cloud computing in smart cities. *Future Internet*, 17(3), 2025. ISSN 1999-5903. doi: 10.3390/fi17030118. URL `https://www.mdpi.com/1999-5903/17/3/118`.

[69] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1), jun 2016. ISSN 0360-0300. doi: 10.1145/2926965. URL `https://doi-org.libproxy.aalto.fi/10.1145/2926965`.

[70] Florian Wohlfart, Nikolaos Chatzis, Caglar Dabanoglu, Georg Carle and Walter Willinger. Leveraging interconnections for performance: the serving infrastructure of a large cdn. *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 206–220, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674. doi: 10.1145/3230543.3230576. URL `https://doi.org/10.1145/3230543.3230576`.

[71] Yinhao Xiao, Yizhen Jia, Chunchi Liu, Xiuzhen Cheng, Jiguo Yu and Weifeng Lv. Edge computing security: State of the art and challenges. *Proceedings of the IEEE*, 107(8):1608–1631, 2019. doi: 10.1109/JPROC.2019.2918437.

[72] Tian Zhang, Dong Xie, Feifei Li and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, Santa Cruz CA USA, November 2019. ACM. ISBN 978-1-4503-6973-2. doi: 10.1145/3357223.3362723. URL `https://dl.acm.org/doi/10.1145/3357223.3362723`.

[73] Behrouz Zolfaghari, Gautam Srivastava, Swapnoneel Roy, Hamid R. Nemati, Fatemeh Afghah, Takeshi Koshiba, Abolfazl Razi, Khodakhast Bibak, Pinaki Mitra and Brijesh Kumar Rai. Content Delivery Networks: State of the Art, Trends, and Future Roadmap. *ACM Computing Surveys*, 53(2):1–34, March 2021. ISSN 0360-0300, 1557-7341. doi: 10.1145/3380613. URL `https://dl.acm.org/doi/10.1145/3380613`.