SQLite: Past, Present, and Future

Kevin P. Gaffney
University of Wisconsin-Madison
kpgaffney@wisc.edu

Martin Prammer
University of Wisconsin-Madison
prammer@cs.wisc.edu

Larry Brasfield
SQLite
larrybr@sqlite.org

D. Richard Hipp
SQLite
drh@sqlite.org

Dan Kennedy
SQLite
dan@sqlite.org

Jignesh M. Patel
University of Wisconsin-Madison
jignesh@cs.wisc.edu

ABSTRACT

In the two decades following its initial release, SQLite has become the most widely deployed database engine in existence. Today, SQLite is found in nearly every smartphone, computer, web browser, television, and automobile. Several factors are likely responsible for its ubiquity, including its in-process design, standalone codebase, extensive test suite, and cross-platform file format. While it supports complex analytical queries, SQLite is primarily designed for fast online transaction processing (OLTP), employing row-oriented execution and a B-tree storage format. However, fueled by the rise of edge computing and data science, there is a growing need for efficient in-process online analytical processing (OLAP). DuckDB, a database engine nicknamed "the SQLite for analytics", has recently emerged to meet this demand. While DuckDB has shown strong performance on OLAP benchmarks, it is unclear how SQLite compares. Furthermore, we are aware of no work that attempts to identify root causes for SQLite's performance behavior on OLAP workloads. In this paper, we discuss SQLite in the context of this changing workload landscape. We describe how SQLite evolved from its humble beginnings to the full-featured database engine it is today. We evaluate the performance of modern SQLite on three benchmarks, each representing a different flavor of in-process data management, including transactional, analytical, and blob processing. We delve into analytical data processing on SQLite, identifying key bottlenecks and weighing potential solutions. As a result of our optimizations, SQLite is now up to 4.2X faster on SSB. Finally, we discuss the future of SQLite, envisioning how it will evolve to meet new demands and challenges.

PVLDB Reference Format:
Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. SQLite: Past, Present, and Future. PVLDB, 15(12): 3535 - 3547, 2022.
doi:10.14778/3554821.3554842

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/UWHustle/sqlite-past-present-future.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554842

1 INTRODUCTION

SQLite was initially released in August 2000 as a small library of data management functions [29]. Originally packaged as an extension to the Tcl programming language, SQLite was born out of the frustration of debugging a database server running in a separate process [53]. Unlike client-server database systems, which typically occupy dedicated processes and communicate with applications via shared memory primitives, SQLite is embedded in the process of the host application [33]. Instead of communicating with a database server across process boundaries, applications manage a SQLite database by calling SQLite library functions.

In the decades that followed its initial release, SQLite grew to become the most widely deployed database engine in existence [27]. SQLite is embedded in major web browsers, personal computers, smart televisions, automotive media systems, and the PHP and Python programming languages. Furthermore, SQLite is found in every iOS and Android device, which currently number in the billions. There are likely over one trillion SQLite databases in active use. It is estimated that SQLite is one of the most widely deployed software libraries of any type.

No single factor is likely responsible for SQLite's popularity. Instead, in addition to its fundamentally embeddable design, several characteristics combine to make SQLite useful in a broad range of scenarios. In particular, SQLite strives to be:

- **Cross-platform.** A SQLite database is stored in a single file, which can be freely copied between 32-bit and 64-bit machines and little-endian and big-endian architectures [30]. SQLite can run on any platform with an 8-bit byte, two's complement 32-bit and 64-bit integers, and a C compiler. Due to its stability and portability, SQLite's file format is a US Library of Congress recommended storage format for the preservation of digital content [1].
- **Compact and self-contained.** The SQLite library is available as a single C file, consisting about 150 thousand lines of source code [31]. With all features enabled, the compiled library size can be less than 750 KiB [17]. SQLite has no external dependencies and requires only a handful of C standard library functions to operate. SQLite requires no installation or configuration.
- **Reliable.** There are over 600 lines of test code for every line of code in SQLite [25]. Tests cover 100% of branches in the library. The test suite is extremely diverse, including fuzz tests, boundary value tests, regression tests, and tests that simulate operating system crashes, power losses, I/O errors, and out-of-memory errors. Due to its reliability, SQLite

is often used in mission-critical applications such as flight software [36].

- **Fast.** SQLite can support tens of thousands of transactions per second. In some cases, SQLite reads and writes blob data 35% faster and uses 20% less storage space than the filesystem [16]. SQLite's query planner produces efficient plans for complex analytical queries [28].

While distinct in many ways, SQLite shares several characteristics with traditional database systems designed for online transaction processing (OLTP). SQLite uses a row-oriented storage format, where all the columns of a given record are stored in a contiguous memory region [23]. SQLite's operators act on individual rows, rather than batches of rows as in vectorized query execution [32]. Finally, SQLite provides full ACID guarantees: transactions are atomic, consistent, isolated, and durable [34].

However, SQLite is used in scenarios that are well outside the boundaries of conventional OLTP. Well-known uses of SQLite include processing data on embedded devices and the internet of things, managing application state as an application file format, serving website requests, analyzing large datasets, caching enterprise data, and transferring data from one system to another [19]. A previous study, which traced SQLite activity on mobile phones, found that SQLite was used for a diverse range of tasks [39]. During the study, the majority of operations performed by SQLite were single-table scans and key-value lookups. However, the trace included much more complex analytical queries that involved joins between several tables. In addition, a small but significant portion of the workloads consisted of OLTP operations.

Recently, the explosive growth of edge computing and interactive data analysis has created a need for efficient in-process online analytical processing (OLAP). Several database systems have already been created or adapted for OLAP, including MonetDB [37], Oracle OLAP [10], and SAP HANA [11]. However, these efforts have largely focused on client-server architectures. Despite this focus, efficient OLAP is often desired in situations where a client-server database system is unwieldy or even impossible to use.

In-process OLAP is an important component of edge computing. In the internet of things, data analysis is increasingly being pushed to the edge in order to reduce network traffic and server load [48]. In addition, there are often privacy concerns associated with sending sensitive data across a network. While OLAP database systems are well-suited for the task of data analysis, edge devices may not possess the energy or computational resources necessary to host a client-server database system.

As another example, data scientists frequently perform in-process OLAP to interactively explore a dataset prior to building a model. Significant portions of data science workflows involve relational algebraic operations, such as selection, projection, join, and aggregation. These operations can be concisely scripted using a dataframe library such as pandas [52]. However, despite their ease of use, dataframe libraries provide limited query optimization and often materialize large intermediate results. Furthermore, datasets often exceed the capacity of memory, which may force the data scientist to implement hand-rolled buffer management and use inefficient storage representations, such as CSV or JSON. An embeddable database engine is better equipped to handle these workloads. For this reason, SQLite is already a popular tool in data science. Due to its stability, portability, and space-efficiency, the SQLite database file format is commonly used for sharing datasets. For example, SQLite is one of the primary file formats used by the popular Kaggle data science platform [38]. The Python `sqlite3` module [15] is often used to carry out SQL operations in data science notebooks. While SQLite produces efficient query plans and handles datasets much larger than memory, it is less optimized for analytics compared to OLAP-specific database systems such as those mentioned above.

These areas are prime targets for a powerful, embeddable OLAP database engine. DuckDB [47] recently emerged to meet this demand. Nicknamed "the SQLite for analytics", DuckDB is built from the ground up for in-process OLAP, employing columnar storage, parallel and vectorized query processing, and multi-version concurrency control optimized for extract-transform-load (ETL) operations. While still in a pre-release development phase, DuckDB has already produced competitive performance on OLAP benchmarks [8]. We believe that DuckDB fills a much needed gap in embeddable data processing.

While SQLite and DuckDB have both been evaluated separately, a well-rounded comparison of the two systems is missing from the literature. As described above, SQLite shares many design elements with OLTP database systems, so one might expect it to excel on OLTP benchmarks. However, SQLite aims to be as general-purpose as possible, so competitive performance could reasonably be expected on a variety of workloads. In contrast, DuckDB is purpose-built for analytics, so one might expect it to outperform SQLite on OLAP benchmarks.

In this paper, we provide experimental support for these expectations and quantify their magnitude. For OLAP, we go one level deeper—we identify specific characteristics of SQLite responsible for its OLAP performance and then present optimizations that substantially increase its speed.

The key contributions of this paper are:

- **We present a historical perspective of SQLite.** We also present a concise description of its architecture.
- **We provide a thorough evaluation of SQLite on characteristic workloads, using DuckDB as a baseline.** Our evaluation includes benchmarks that represent diverse flavors of in-process data management.
- **We optimize SQLite for analytical data processing.** We identify key bottlenecks in SQLite and discuss the advantages and disadvantages of potential solutions. We integrate our optimizations into SQLite, resulting in overall 4.2X speedup on SSB.
- **We identify several performance measures specific to embeddable database engines**, including library footprint and blob processing performance.
- **We provide a high-level description of potential future directions for further performance improvement in SQLite.**

The rest of this paper is organized as follows. Section 2 provides an architectural overview of SQLite and rationale for its design. Section 3 discusses SQLite in the context of evolving workloads and hardware. Section 4 evaluates SQLite and DuckDB on a variety

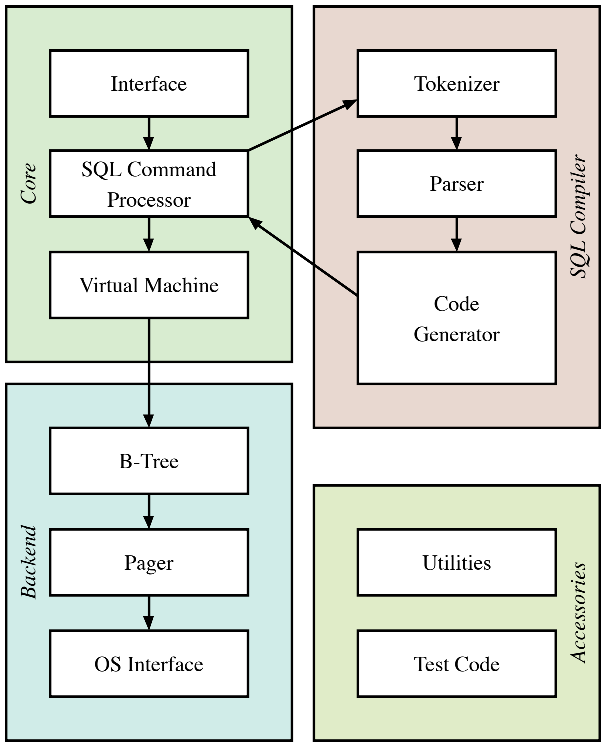


Figure 1: Architecture of SQLite

of performance measures and presents our optimizations. Section 5 discusses future directions, and Section 6 concludes.

2 ARCHITECTURE

In this section, we present a brief overview of SQLite's architecture. We include this information to facilitate understanding of subsequent sections and keep this paper self-contained. The finer details of SQLite's architecture can be found in the documentation [20]. We also discuss some of the rationale behind core elements of SQLite's design.

2.1 Modules

SQLite follows a modular design. Its architecture consists of the four groups of modules show in Figure 1. The *core* modules are responsible for ingesting and executing SQL statements. The *SQL compiler* modules translate a SQL statement into a bytecode program that can be executed by the virtual machine. The *backend* modules facilitate access database pages and interact with the operating system to persist data. SQLite also includes several *accessory* modules, including a large suite of tests and utilities for memory allocation, string operations, and random number generation. We describe some of these modules in greater detail below.

2.1.1 SQL compiler modules. Conceptually, each SQL statement can be thought of as an executable program. Extending the analogy further, SQLite's tokenizer, parser and code generator act like a compiler, translating SQL into executable code. The output of the code generator is a bytecode program. An example bytecode program, which was compiled from Star Schema Benchmark (SSB) Q1.1, is shown in Table 1. A bytecode program consists of one or more virtual instructions. Each virtual instruction includes a unique opcode and several operands. These virtual instructions are the basic building blocks of data processing in SQLite. While they can be combined into complex programs, the instructions themselves are rather low-level. For example, the `Column` instruction extracts the P2th column from the current row and stores it in register P3. The `Lt` instruction jumps to address P2 if the value in register P3 is less than the value in register P1. A complete description of SQLite's virtual instruction set is available in the documentation [32].

Table 1: Bytecode program for SSB Q1.1.

Address	Opcode	P1	P2	P3	P4	P5
0	`Init`	1	23	0		00
1	`Null`	0	1	3		00
2	`OpenRead`	0	7	0	12	00
3	`OpenRead`	1	6	0	5	00
4	`Rewind`	0	19	0		00
5	`  Column`	0	11	4		00
6	`  Lt`	6	18	4	BINARY-8	54
7	`  Gt`	7	18	4	BINARY-8	54
8	`  Column`	0	8	4		00
9	`  Ge`	8	18	4	BINARY-8	54
10	`  Column`	0	5	9		00
11	`  SeekRowid`	1	18	9		00
12	`  Column`	1	4	4		00
13	`  Ne`	10	18	4	BINARY-8	54
14	`  Column`	0	9	5		00
15	`  Column`	0	11	11		00
16	`  Multiply`	11	5	4		00
17	`  AggStep1`	0	4	1	sum(1)	01
18	`Next`	0	5	0		01
19	`AggFinal`	1	1	0	sum(1)	00
20	`Copy`	1	12	0		00
21	`ResultRow`	12	1	0		00
22	`Halt`	0	0	0		00
23	`Transaction`	0	0	6	0	01
24	`Integer`	1	6	0		00
25	`Integer`	3	7	0		00
26	`Integer`	25	8	0		00
27	`Integer`	1993	10	0		00
28	`Goto`	0	1	0		00

2.1.2 Core modules. SQLite's execution engine is structured as a virtual machine. The virtual machine, also known as the virtual database engine (VDBE), is the heart of SQLite. The VDBE is responsible for executing the logic of the bytecode program produced by the code generator. The VDBE begins with the instruction at address 0 and continues until it sees a `Halt` instruction, encounters an error, or reaches the end of the bytecode program. The instruction logic is implemented as a large switch statement in the VDBE, where each instruction is processed as a unique case. When the VDBE exits, it frees any memory it may have allocated and closes any cursors it may have opened. If an error was encountered, the

VDBE rolls back any pending changes to the database to leave it in a clean state.

2.1.3 Backend modules. The VDBE interacts heavily with SQLite's backend, particularly the B-tree module. An SQLite database file is essentially a collection of B-trees [23]. A B-tree is either a *table* B-tree or an *index* B-tree. Table B-trees always use a 64-bit signed integer key and store data in the leaves. Index B-trees use arbitrary keys and store no data at all. Each table in the database schema is represented by a table B-tree. The key of a table B-tree is the implicit *rowid* column of the table. For `INTEGER PRIMARY KEY` tables, the primary key column replaces the rowid as the B-tree key. Tables declared with the specification `WITHOUT ROWID` are a special case; these tables are stored entirely in index B-trees. The B-tree key for a `WITHOUT ROWID` table is composed of the columns of the primary key followed by all remaining columns of the table. There is one index B-tree for each index in the database schema, unless that index is already represented by a table B-tree, as in the case of `INTEGER PRIMARY KEY` tables.

The page cache is responsible for providing pages of data requested by B-tree module. The page cache is also responsible for ensuring modified pages are flushed to stable storage safely and efficiently. Finally, the OS interface is the gateway to the underlying file system. SQLite uses an abstract object called the virtual file system (VFS) to provide portability across operating systems. SQLite comes with several existing VFSes for Unix and Windows operating systems. A VFS can be created to support a new operating system or extend the functionality of SQLite.

2.2 Transactions

SQLite is a transactional database engine. It provides the ACID guarantees of atomicity, consistency, isolation, and durability. SQLite has two primary modes by which these guarantees are achieved: rollback mode and write-ahead log mode.

2.2.1 Rollback mode. In rollback mode, SQLite begins a transaction by acquiring a shared lock on the database file. After the shared lock is acquired, pages may be read from the database at will. If the transaction involves changes to the database, SQLite upgrades the read lock to a reserved lock, which blocks other writers but allows readers to continue. Before making any changes, SQLite also creates a rollback journal file. For each page to be modified, SQLite writes its original content to the rollback journal and keeps the updated pages in user space. When SQLite is instructed to commit the transaction, it flushes the rollback journal to stable storage. Then, SQLite acquires an exclusive lock on the database file, which blocks both readers and writers, and applies its changes. The updated database pages are then flushed to stable storage. The rollback journal is then invalidated via one of several mechanisms, depending on the *journal mode*. In DELETE mode, SQLite deletes the rollback journal. Because deleting a file is expensive on some systems, SQLite also provides alternative journal modes. In TRUNCATE mode, the rollback journal is truncated instead of deleted. In PERSIST mode, the header of the rollback journal is overwritten with zeros. The act of invalidating the rollback journal effectively commits the transaction. Finally, SQLite releases the exclusive lock on the database file.

2.2.2 Write-ahead log mode. Conceptually, write-ahead log (WAL) mode is an inversion of rollback mode. In rollback mode, SQLite writes original pages to the rollback journal and modified pages to the database file. In contrast, WAL mode maintains original pages in the database file and appends modified pages to a separate WAL file. In WAL mode, SQLite begins a transaction by recording the location of the last valid commit record in the WAL, called the *end mark*. When SQLite needs a page, it searches the WAL for the most recent version of that page prior to the end mark. If the page is not in the WAL, SQLite retrieves the page from the database file. Changes are merely appended to the end of the WAL. A commit that causes the WAL to grow beyond a specified size will trigger a *checkpoint*, in which updated pages in the WAL are written back to the database file. The WAL file is not deleted after a checkpoint; instead, transactions overwrite the file, starting from the beginning.

There are two main advantages to WAL mode. First, WAL mode offers increased concurrency as readers can continue operating on the database while changes are being committed into the WAL. While there can only be one writer at a time, readers can proceed concurrently with the one writer. Second, WAL is often significantly faster as it requires fewer writes to stable storage, and the writes that do occur are more sequential.

However, WAL mode has notable disadvantages. To accelerate searching the WAL, SQLite creates a *WAL index* in shared memory. This improves the performance of read transactions, but the use of shared memory requires that all readers must be on the same machine. Thus, WAL mode does not work on a network filesystem. It is not possible to change the page size after entering WAL mode. In addition, WAL mode comes with the added complexity of checkpoint operations and additional files to store the WAL and the WAL index.

3 EVOLVING WORKLOADS AND HARDWARE

Far from its humble beginnings as a small library of data management functions, SQLite is now the most used database engine in the world. However, just as SQLite's usage has grown, so too have the demands that users place on SQLite; modern use-cases of SQLite have developed significantly more complex needs than what can be satisfied by a simple data storage platform. Many applications utilize SQLite for its properties as a platform-independent storage format, while others are more concerned with its robustness and reliability guarantees [35]. These changing and growing needs are not only a reflection of the constantly increasing complexity of modern software, but also a reflection of hardware advancements made since SQLite's inception. In this section, we examine both the changes in hardware and workloads to better understand how SQLite fits in the modern landscape of database engines.

3.1 Hardware Advancements

While the improvement of computing hardware over time is well understood, it is important to contextualize how quickly some of these changes have come about. One of the earliest devices to successfully run SQLite was a Palm Pilot, a personal digital assistant powered by a Motorola MC68328, a 16MHz, single core processor [40, 45]. While in this case it was an independent user that deployed SQLite to their personal phone, the trend of SQLite being

Table 2: Hardware configurations used for evaluation.

Name	Processor	Processor speed	Cores	Memory	Storage
Cloud server	Intel Xeon Silver 4114	2.2 GHz	10	192 GB ECC DDR4-2666	Intel DC S3500 480 GB 6G SATA SSD
Raspberry Pi	ARM Cortex-A72	1.5 GHz	4	8 GB LPDDR4-3200	Micro-SD card

used in resource constrained environments would continue. Nokia and Motorola were two of the earliest companies to adopt SQLite into their mobile phones. Eventually, Google would follow as well, integrating SQLite into its Android platform where it continues to be used to this day [27, 36, 45].

However, mobile compute has fundamentally changed since the early 2000s. For example, the Raspberry Pi 4 Model B, which we use as part of our evaluation in section 4, was released in July 2019 as an inexpensive yet powerful single-board computer [12, 13]. The Raspberry Pi 4 Model B uses an ARM Cortex-A72, a 1.5GHz 4-core processor [14], which is a significant performance improvement compared to the Motorola processor mentioned above. Furthermore, the Raspberry Pi 4 Model B is powered by computing hardware that supports single instruction, multiple data (SIMD) and hardware-level parallelism. As illustrated by the dramatic differences between the Motorola processor and the Raspberry Pi 4 Model B, the capabilities of mobile computing hardware have grown at a rapid pace.

3.2 Workload Changes

In addition to the aforementioned hardware advancements, the software that uses SQLite has evolved as well. We emphasize that, fundamentally, SQLite is an OLTP-focused database engine that is significantly optimized for use in resource constrained environments. However, SQLite is often used for workloads that are considerably different than those for which it was originally designed. For example, a month-long trace of SQLite usage on mobile phones observed a broad range of workloads with varying query complexity and read/write mix [39]. A large proportion of operations were simple key-value lookups, suggesting that SQLite is often used simply as a key-value store. However, the trace included a significant tail of complex OLAP queries. These queries involved multiple levels of nesting or joins between 5 or more tables. In addition, about 25% of all observed statements involved writes to the database. Many of these writes were `UPSERTS` (insert or replace operations), providing further evidence that SQLite is often used as a key-value store. The trace also included a significant proportion of `DELETES`, which were much more expensive than other statements, averaging about 4 ms per statement. Several `DELETE` statements included predicates with nested `SELECT` queries. This study suggests that usage of SQLite is extremely varied. Furthermore, this study was limited to mobile phone usage; we expect even greater workload diversity when considering the range of devices on which SQLite runs. Broadly, we find that these observations represent the continually expanding demands of real-world applications.

3.3 SQLite in the Modern World

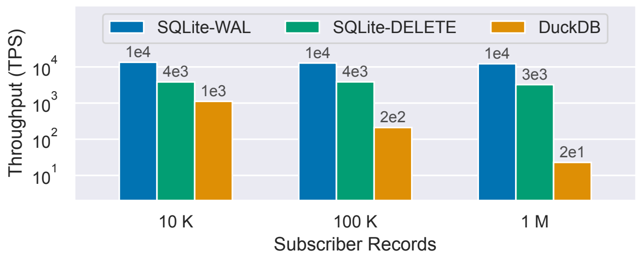
Advancements in computing hardware and application software over time have placed SQLite in a unique position. While it continues to be the most widely used database engine in the world, the drastic changes in both hardware capabilities and software demands have exposed SQLite to a unique set of challenges.

The expansion of hardware capabilities calls for a deeper evaluation into the underlying implementation of SQLite. Notably, SQLite generally does not use multiple threads, which limits its ability to take advantage of the available hardware parallelism. For sorting large amounts of data, SQLite uses an optional multithreaded external merge sort algorithm. For all other operations, SQLite performs all work in the calling thread. This design minimizes resource competition with other processes running on the device. However, it is likely that certain workloads, particularly those that include complex OLAP, would benefit from multithreading. Furthermore, SQLite's row-oriented storage format and execution engine are suboptimal for many OLAP operations. In general, SQLite is considered not to be competitive with state-of-the-art OLAP-focused database engines, especially in the context of its limitations. In contrast, DuckDB [47] has poised itself as "the SQLite for analytics" through a number of features modeled after SQLite, such as its embeddable design, single-file database, and self-contained code. However, DuckDB brings many state-of-the-art OLAP techniques to the SQLite-like environment, such as a vectorized engine and parallel query processing. Together, these features have enabled DuckDB's strong OLAP performance. We question which OLAP-focused optimizations can be incorporated into SQLite without sacrificing its portability, compactness, reliability, and efficiency on diverse workloads.

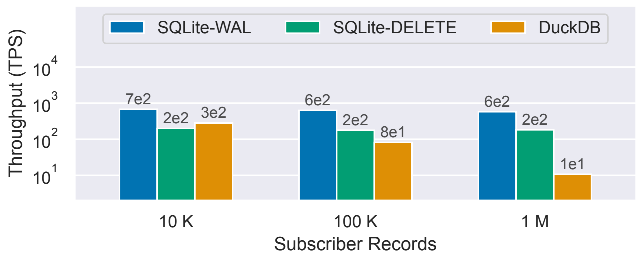
4 EVALUATION AND OPTIMIZATION

In this section, we present an extensive performance evaluation of SQLite. We employ three benchmarks, each of which simulates a different flavor of in-process data management, namely OLTP, OLAP, and blob I/O. While these benchmarks do not cover all existing use cases of embeddable database engines, they are representative of the most common workloads. We identify key bottlenecks in SQLite's OLAP performance, discuss the tradeoffs of potential solutions, and present the performance impact of our optimizations. Finally, we discuss the "footprint" of SQLite: the amount of memory and time it takes to compile, the size of the resulting binary, and the space it requires to store benchmark datasets.

We use DuckDB [47] as a reference point for comparisons. DuckDB is an embeddable database engine optimized for analytics. Similar to the design of SQLite, DuckDB is built entirely from two files, a header file and an implementation file, with no external dependencies. DuckDB is embedded in the process of the host application and manages a database stored in a single file. However, in contrast



(a) Cloud server



(b) Raspberry Pi

Figure 2: TATP throughput (logarithmic scale, higher is better).

to SQLite, DuckDB uses columnar data organization and vectorized query execution. DuckDB also uses a variant of multi-version concurrency control optimized for bulk operations. Our use of DuckDB as a baseline is not to imply that one system is definitively better than the other. Rather, we observe that there is a substantial set of tasks on which it would be appropriate to use either SQLite or DuckDB. We evaluate DuckDB alongside SQLite to provide a more well-rounded picture of the performance of both systems.

The details of our experiments are the following.

- **Hardware.** We use two hardware configurations: a cloud server, provisioned on CloudLab [9], with an Intel Xeon Silver 4114 CPU; and a Raspberry Pi 4 Model B with an ARM Cortex-A72 CPU [14]. More details are in Table 2.
- **Versions.** We used SQLite version 3.38.0 and DuckDB version 0.3.2. These were the most recent available software versions at the time this paper was written.
- **Options.** SQLite and DuckDB were built with `gcc -O3`. SQLite was built with all the recommended compile-time options [22] except `SQLITE_DEFAULT_WAL_SYNCHRONOUS` to maintain durability in WAL mode. We added the option `SQLITE_OMIT_LOAD_EXTENSION`. DuckDB was restricted to a single thread. Unless otherwise noted, both SQLite3 and DuckDB were allowed 1 GB of memory. All other options were left as default.
- **Reporting.** For each experiment, we plot the mean outcome of three trials. We observed negligible variance among the three trials. Each experiment is annotated with the mean outcome rounded to one significant figure. Due to extreme differences in performance, some plots use a logarithmic scale.

4.1 Online transaction processing

To evaluate OLTP performance, we use the TATP benchmark [42]. TATP is designed to measure the performance of a database system in a typical telecommunications application. The benchmark consists of seven transaction types that are randomly generated according to fixed probabilities; 80% are read-only, and 20% involve updates, inserts, or deletes. TATP has been extensively used in related work and thoroughly compared to other benchmarks. Because its transactions are relatively lightweight, TATP is better suited for modeling a SQLite workload than the more complex TPC-C [49] and TPC-E [50] benchmarks, which are often used to evaluate client-server OLTP database systems.

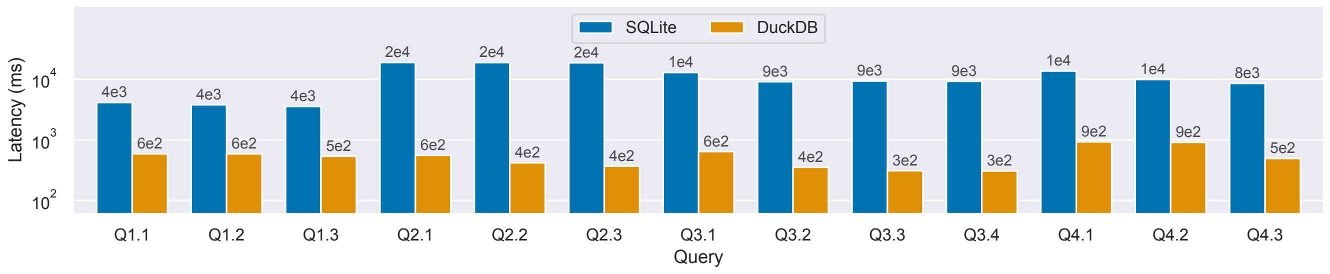
Our experiments follow the TATP specification. To measure the effect of database size on performance, we vary the number of records in the `subscriber` table and scale other tables proportionally. We simulate a single TATP client in all experiments. Each experimental trial consists of a 10 second warmup period followed by a 60 second evaluation period. We report the throughput of each system in transactions per second (TPS).

We evaluate three SQLite journal modes: DELETE, TRUNCATE, and WAL. These journal modes are briefly introduced in section 2. More detailed descriptions of SQLite's journal modes are available in the documentation [21]. We observed negligible performance difference between DELETE and TRUNCATE modes, so we exclude TRUNCATE results for brevity.

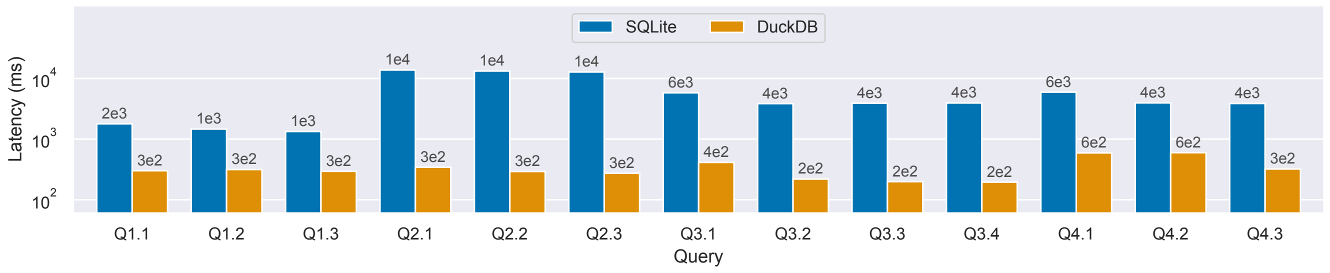
It is worth noting that TATP is outside DuckDB's "comfort zone". DuckDB is designed for workloads consisting of mainly OLAP and extract-transform-load (ETL) operations [47]. To support concurrent OLAP and ETL, DuckDB offers transactional guarantees via multi-version concurrency control. However, DuckDB is optimized for bulk updates, such as adding a column to a table or appending a large batch of rows, rather than the fine-grained operations typically present in OLTP workloads. Nevertheless, DuckDB's performance on TATP is an informative baseline for SQLite.

Results are shown in Figure 2. On both hardware configurations, SQLite-WAL produces the highest throughput by a wide margin. On the cloud server, SQLite-WAL reaches a throughput of 10 thousand TPS, which is 10X faster than DuckDB for the small database and 500X faster for the large database. On the Raspberry Pi, the performance gap is smaller yet still significant. SQLite-WAL is 2X faster for the small database and 60X faster for the large database. SQLite-DELETE is slower than SQLite-WAL, but substantially faster than DuckDB on the cloud server. On the Raspberry Pi, DuckDB has a slight edge over SQLite-DELETE for small databases, but SQLite-DELETE is faster for large databases. Both SQLite-WAL and SQLite-DELETE produce generally consistent performance regardless of database size, whereas DuckDB is adversely affected by database size.

These results are generally consistent with our expectations. SQLite's transaction processing machinery, which has been finely tuned over many years, produces strong performance on TATP. Meanwhile, DuckDB, which is designed primarily for OLAP and



(a) Cloud server



(b) Raspberry Pi

Figure 3: SSB latency (logarithmic scale, lower is better).

ETL workloads, performs significantly worse. A resulting question is whether this performance gap is due to implementation details or more fundamental differences in system architecture. We leave this question to future work.

4.2 Online analytical processing

We now transition to OLAP performance. We present the results of our evaluation of SQLite and DuckDB, followed by an analysis of SQLite's performance bottlenecks. Finally, we discuss our optimizations and their impact on performance.

4.2.1 Benchmarking. To evaluate OLAP performance, we use the Star Schema Benchmark (SSB) [43]. SSB measures the performance of a database system in a typical data warehousing application. The benchmark uses a modified TPC-H [51] schema that consists of large *fact* table and four smaller *dimension* tables. SSB queries involve joins between the fact table and the dimension tables with filters on dimension table attributes. SSB is widely used in OLAP database system research.

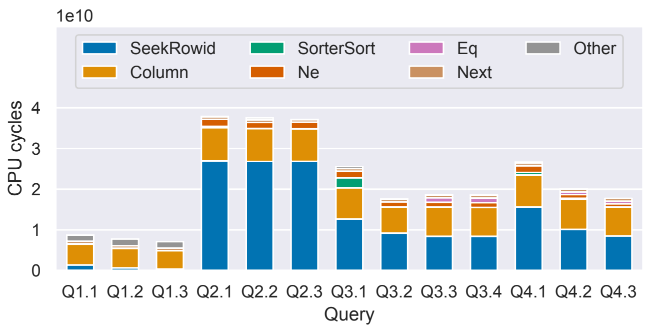
Our experiments follow the SSB specification with minimal modification. Our only departure from the specification is that before running the SSB queries, we scan each table with a `SELECT *` query, ensuring that the buffer pool is populated. We use scale factor of 1 on the Raspberry Pi and scale factor 5 on the cloud server. Both SQLite and DuckDB were allowed to gather statistics about the data prior to measurement.

Results are shown in Figure 3. For all queries, DuckDB is substantially faster than SQLite. The widest performance margin is on query flight 2, for which DuckDB is 30-50X faster, and the narrowest margin is on flight 1, for which DuckDB is 3-8X faster. SQLite's latency is highly variable across different query flights. On the Raspberry Pi, SQLite's fastest query is 10X faster than its slowest, whereas DuckDB's fastest query is only about 3X faster than its slowest. Interestingly, SQLite's fastest queries are in flight 1, whereas DuckDB's fastest queries are in flight 3.

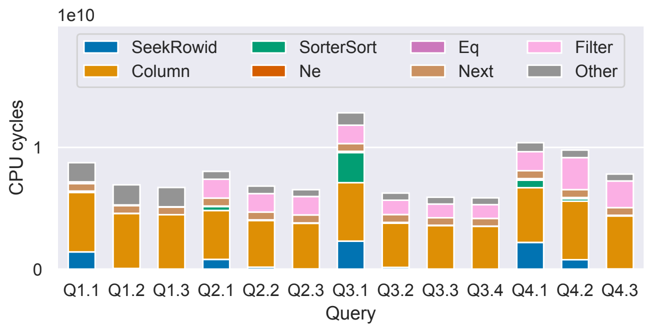
4.2.2 Performance profiling. To understand the fundamental reasons for our observations, we profiled SQLite's execution engine. As described in section 2, SQLite's query planner translates an SQL query into a bytecode program, which is then executed by the VDBE. The instructions that comprise the bytecode program are the building blocks of data processing in SQLite. This design naturally accommodates performance profiling. SQLite provides the compile-time option `VDBE_PROFILE`, which enables utilities that measure the number of CPU cycles the VDBE spends executing each instruction in the bytecode program. We run SSB on SQLite with this option turned on.

Profiling results are shown in Figure 4a. Surprisingly, only two instructions are responsible for the vast majority of cycles: `SeekRowid` and `Column`. The `SeekRowid` instruction searches a B-tree index for a row with a given row ID. For a table with `INTEGER PRIMARY KEY`, the row ID is equivalent to the primary key. When running SSB, SQLite uses the `SeekRowid` instruction to perform an index join between the fact table and a dimension table. The `Column` instruction extracts a column from a given record. While the `Column` instruction consumes a substantial amount of cycles for each query, the `SeekRowid` instruction is largely responsible for the dramatic differences in performance on query flight 1 and query flight 2. Based on these results, we identified two key optimization targets: avoiding unnecessary B-tree probes and streamlining value extraction.

4.2.3 Avoiding unnecessary B-tree probes. We first describe our approach to avoiding unnecessary B-tree probes during joins. SQLite



(a) Pre-optimization



(b) Post-optimization

Figure 4: SSB performance profiles.

```sql
SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder, date, part, supplier
WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
  AND lo_suppkey = s_suppkey
  AND p_category = 'MFGR#12'
  AND s_region = 'AMERICA'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1;
```

(a) SQL

```
QUERY PLAN
|--SCAN lineorder
|--SEARCH part USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH date USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH supplier USING INTEGER PRIMARY KEY (rowid=?)
`--USE TEMP B-TREE FOR GROUP BY
```

(b) Query plan pre-optimization

```
QUERY PLAN
|--SCAN lineorder
|--BLOOM FILTER ON part (p_partkey=?)
|--BLOOM FILTER ON supplier (s_suppkey=?)
|--SEARCH part USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH date USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH supplier USING INTEGER PRIMARY KEY (rowid=?)
`--USE TEMP B-TREE FOR GROUP BY
```

(c) Query plan post-optimization

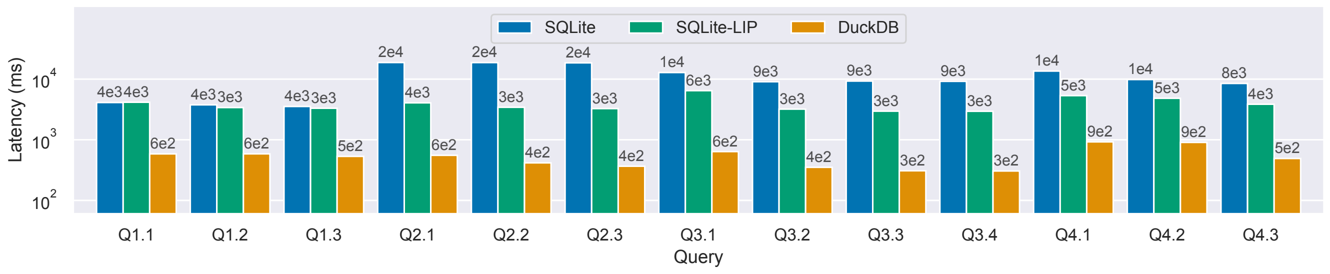
Figure 5: SSB Q2.1.

uses nested loops to compute joins. However, the inner loops in the join are typically accelerated with existing primary key indexes or temporary indexes built on the fly. In the following discussion, we use the term *outer table* to refer to the table that is scanned in the outermost loop. We use the term *inner table* to refer to a table that is searched, via scan or index, for tuples that join with those in the outer table.

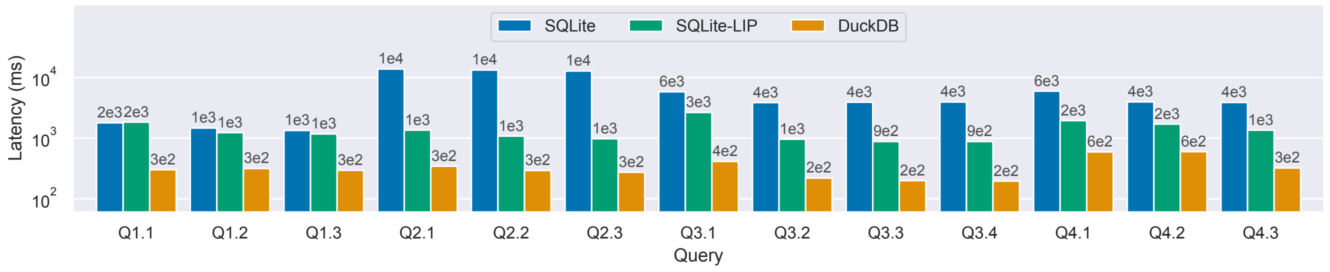
We use SSB Q2.1, shown in Figure 5a, as a working example. For this query, SQLite produces the query plan shown in Figure 5b. In this query plan, the outer table is the `lineorder` table, and the inner tables are the `part`, `date`, and `supplier` tables. For each tuple in the `lineorder` table, SQLite probes the primary key index on the `part` table for the joining tuple. If the restriction on `p_category` is satisfied, SQLite then probes the primary key index on the `date` table, followed by the primary key index on the `supplier` table, for the joining tuples. If the restriction on `s_region` is satisfied, SQLite adds the `lo_revenue` to the accumulator for the current `d_year` and `p_brand1`. Note that SQLite probes the `part` table index for every tuple in the `lineorder` table. Because the `part` table is the largest dimension table, probes to its primary key index are expensive. Furthermore, only 0.8% of the `lineorder` tuples satisfy the restrictions on `p_category` and `s_region`. Other SSB joins are even more selective on `lineorder` table. As a result, a large portion of B-tree probes are for tuples that are ultimately excluded from the final result.

These observations led us to consider two potential optimizations: hash joins and Bloom filters [3]. Hash joins are attractive for their best-case linear time complexity. However, hash joins often require a significant amount of memory and may spill to storage if the available memory is exhausted. Moreover, adding a second join algorithm to SQLite would considerably increase the complexity of its query planner. In contrast, Bloom filters are memory-efficient and require minimal modification to the query planner. Bloom filters have well-studied theoretical properties [3, 41, 54] and demonstrated usefulness in constrained memory scenarios [2]. Bloom filters are very powerful for selective star joins, such as those in SSB. For these reasons, we integrated Bloom filters into SQLite's join algorithm.

We use Bloom filters to implement Lookahead Information Passing (LIP) [54], which was a key technique used in the Quickstep system [44] to speed up the execution of equijoin queries. Similar techniques have also been used in other commercial systems [4–7]. LIP optimizes a pipeline of join operators by creating Bloom filters on all the inner (dimension) tables before the join processing starts, and then passing the Bloom filters to the first join operation. A key change is made to the join processing, which is to probe the Bloom filters before carrying out the rest of the join. Applying the Bloom filters early in the join pipeline dramatically reduces the number of tuples that flow through the join pipeline, and thus improves performance. In addition, LIP has the advantage of being robust



(a) Cloud server



(b) Raspberry Pi

Figure 6: SSB latency of optimized SQLite (logarithmic scale, lower is better).

to poor choices of join order. Our implementation differs slightly from the approach described in the LIP paper [54] in that Bloom filters are probed in a fixed order determined by the query optimizer rather than an adaptive order influenced by hit and miss statistics.

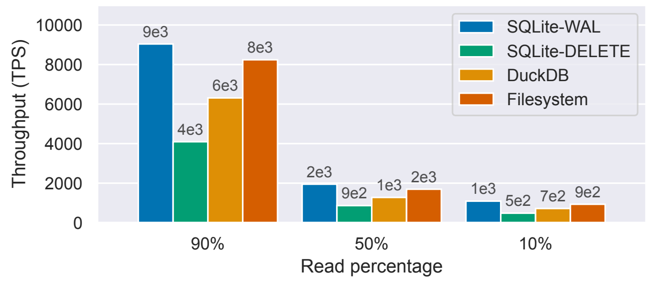
To implement LIP in SQLite, we add two new virtual instructions to the VDBE: `FilterAdd` and `Filter`. `FilterAdd` computes a hash on an operand and sets the corresponding bit in the Bloom filter. Using the same hash function, `Filter` computes a hash on an operand and checks the corresponding bit in the Bloom filter. If the bit is unset, the VDBE jumps to a specified address in the bytecode program. The remaining LIP logic can be constructed from SQLite's existing virtual instructions: Bloom filters are initialized as blobs via the `Blob` instruction, inner tables are scanned via the `Next` and `Rewind` instructions, and expressions are evaluated via SQLite's expression operators. When compiling the bytecode program, the query planner initially places each Bloom filter check immediately before the corresponding search of the inner table. However, if there are multiple Bloom filters, as may be the case when three or more tables are joined, SQLite pushes all Bloom filter checks to the beginning of the outer table loop. As a result, SQLite will check all Bloom filters prior to searching the inner tables.

SQLite's query planner uses a straightforward model to determine whether a Bloom filter should be constructed. For each inner table, the query planner generates the above Bloom filter logic if all of the following conditions are true:

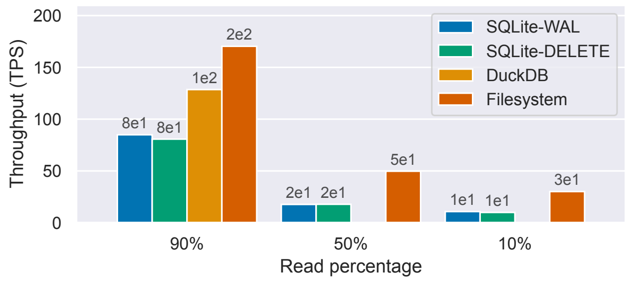
(1) The number of rows in the table is known by the query planner.
(2) The expected number of searches exceeds the number of rows in the table.
(3) Some searches are expected to find zero rows.

After the query planner has generated the bytecode instructions, the join proceeds in two phases: a *build* phase and a *probe* phase. During the build phase, SQLite constructs each Bloom filter by scanning the corresponding inner table, applying the relevant restrictions, hashing the joining column of surviving tuples, and setting the bits in the Bloom filter. SQLite also constructs any temporary indexes requested by the query planner. During the probe phase, SQLite scans the outer table. For each tuple in the outer table, SQLite hashes each joining column and checks the bit in the corresponding Bloom filter. If any of the bits are unset, SQLite immediately advances to the next tuple in the outer table *without* searching any of the inner tables. This design allows SQLite to take advantage of the combined selectivity of several inner table restrictions, substantially reducing the number of unnecessary B-tree probes. Figure 5c shows the updated query plan for SSB Q2.1 after implementing LIP in SQLite. Note that the Bloom filter checks occur prior to the searches of the inner tables.

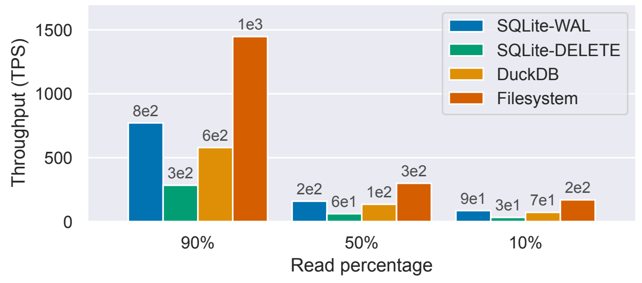
The performance impact of our optimizations is shown in Figure 6. On the Raspberry Pi, SQLite is now 4.2X faster on SSB. Our optimizations are particularly effective for query flight 2, resulting in 10X speedup. On the cloud server, we observed an overall speedup of 2.7X and individual query speedups up to 7X. Importantly, our changes cause no appreciable performance degradation. The performance profiles in Figure 4b further illustrate the benefits of the Bloom filter (note the difference in scale compared to Figure 4a). The VDBE spends substantially fewer CPU cycles on the `SeekRowid` instruction, reflecting the reduction in B-tree probes. The new `Filter` instruction, which probes a Bloom filter for membership, incurs little overhead compared to `SeekRowid` in Figure 4a.



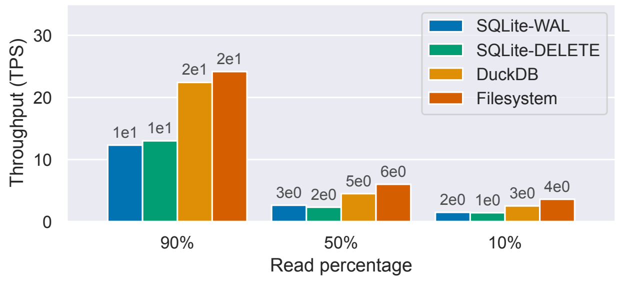
(a) Cloud server, 100 KB blob



(b) Cloud server, 10 MB blob



(c) Raspberry Pi, 100 KB blob



(d) Raspberry Pi, 10 MB blob

Figure 7: BLOB throughput (linear scale, higher is better).

4.2.4 Streamlining value extraction. We now discuss considerations related to streamlining value extraction in SQLite. By value extraction, we refer to the basic operation of obtaining the value of a specific row and column in a table. To aid in understanding SQLite's approach to value extraction, we provide a brief overview of how it stores and retrieves records. A more detailed description is available in the documentation [23].

In contrast to most other database engines, SQLite uses flexible typing. This entails that data of any type may be stored in any column of an SQLite table (except an `INTEGER PRIMARY KEY` column, in which case the data must be integral). Furthermore, columns can be declared without a data type; for example,

```
CREATE TABLE t (a, b, c);
```

is a valid statement in SQLite. Flexible typing is advantageous in several situations [18]. For example, applications that use SQLite as a key-value store can create a table with two columns, a key and a value, and store any type of data in the value column. In addition, flexible typing works well with dynamic programming languages and semi-structured data formats such as JSON.

Whereas statically-typed database engines typically associate a data type with each *column*, SQLite's flexible typing system associates a data type with each *value*. Accordingly, SQLite stores type information alongside each value in the database. An SQLite record is composed of two parts: a header and a body. The header contains *serial type codes* that encode the data type of each column in the record. The record body, which immediately follows the header, contains the actual values.

To extract a value, SQLite must obtain a pointer to the encoded value in the record body. SQLite begins this process by examining a pointer to the header. The first integer in the header is the size of the header in bytes. SQLite records this size as the initial offset from the beginning of the header. SQLite then walks through each serial type code in the header, until it reaches the desired column. For each serial type code, SQLite adds the size of the corresponding value to the total offset. The resulting sum is the offset of the desired value from the beginning of the header. Contrast this approach with a typical columnar database engine, in which successive column values are contiguous. In the columnar approach, value extraction is significantly more streamlined.

We explored several alternative approaches to value extraction in SQLite. However, we quickly encountered limitations surrounding changes to SQLite's database file format. The database file format is extremely stable, cross-platform, and backwards compatible. As noted earlier, the database file format is a US Library of Congress recommended format for the preservation of digital content [1], largely due to its stability, portability, and thorough documentation. It is straightforward to imagine new data formats, such as column-oriented, that would streamline value extraction. However, we were unwilling to sacrifice the stability and portability of the database file format for the added performance. We provide further discussion surrounding these tradeoffs in section 5.

4.3 Blob manipulation

A considerable number of applications use SQLite simply as a blob data store [19]. Such applications may prefer a custom data model over the relational data model SQLite provides. However, they may require stronger guarantees than those provided by direct calls to the filesystem. In contrast to basic file manipulation functions such as `fwrite`, SQLite offers attractive transactional guarantees. Modifications to a SQLite database are ACID-compliant: atomic, consistent, isolated, and durable. Furthermore, manipulating blob data in SQLite can be faster and more space-efficient than using the filesystem [16].

These observations motivated us to develop the BLOB benchmark, which simulates an application that uses a database engine to manage raw blob data. The benchmark is straightforward. A table is created in the database with a single row and a single column of blob data with a given size. Then, a connection to the database repeatedly either reads or writes the entire blob, based on specified probabilities. After a warmup period of 10 seconds, we measure the number of operations completed in 60 seconds. We report the throughput in transactions per second (TPS).

In addition to SQLite (with journal modes WAL AND DELETE) and DuckDB, we used the BLOB benchmark to evaluate filesystem calls. For blob reads, we used `fread`, For blob writes, we used `fdatasync`. While these functions provide lesser transactional guarantees than a database engine, they are an informative baseline for the benchmark.

Results are shown in Figure 7. For 100 KB blobs, SQLite-WAL produces the highest throughput of the transactional methods. On the cloud server, SQLite-WAL even has a slight edge over the filesystem for small blobs. This is likely due to SQLite's ability to serve read requests from its cache, whereas the filesystem serves read requests with calls to `fread`. SQLite-DELETE is slightly less than half as fast as SQLite-WAL. Because SQLite-DELETE writes its changes to the rollback journal and then to the database file, it incurs about twice as many writes as SQLite-WAL, which appends its changes only to the WAL (assuming no checkpoint is triggered by the transaction). For 10 MB blobs, DuckDB produces the highest throughput of the transactional methods. SQLite-WAL and SQLite-DELETE have mostly similar performance for blobs of this size. The default limit of the WAL is 1000 pages, which equates to about 4 MB. Because the blob size is 10 MB, a single write causes the size of the WAL to exceed this limit, which triggers an immediate checkpoint. As a result, SQLite-WAL incurs two writes for each 10 MB blob update. We observed increased throughput for greater WAL size limits, but we omit these results for brevity. All methods, regardless of hardware configuration and blob size, are strongly impacted by the percentage of read requests in the workload. In Figure 7b, certain DuckDB measurements are omitted because DuckDB encountered an error and could not complete the benchmark.

4.4 Resource footprint

An important consideration for an embeddable database engine is the footprint of its resource usage. In particular, SQLite strives to maintain a small compiled library size. SQLite has a simple type system with just a handful of data types [24]. It avoids the use of template metaprogramming to generate code. SQLite also provides many compile-time options that can be disabled to exclude certain features from the library. Furthermore, SQLite strives to be efficient in how it stores data. For example, integers are stored in 0, 1, 2, 3, 4, 6, or 8 bytes, depending on the magnitude of the value. Floating point values with no fractional component may be stored as integers to occupy less space.

We compiled SQLite and DuckDB on the cloud server, recording the compilation time, the maximum memory usage, and the size of the resulting library. We used GCC version 9.3.0 with either `-Os` or `-O3`. Results are shown in Table 3. SQLite requires little time and memory to compile. The resulting library is 900 KB when optimizing for size, and 1.5 MB when optimizing for speed. For applications that require only basic SQLite functionality, it is possible to further reduce the size of the SQLite library by disabling certain compile-time options, but we did not experiment with alternative option configurations. In contrast, compiling DuckDB requires 5-10 minutes and about 7.6 GB of memory, resulting in a library that is 32-37 MB in size. We note that pre-compiled libraries for both SQLite and DuckDB are available for download, so in some cases it may be unnecessary to build either library from source. However, applications that wish to embed DuckDB in their source code may find it problematic to compile in resource-constrained environments.

While the SQLite library is substantially smaller and faster to compile, DuckDB may require less space to store the same data. DuckDB associates type information with entire columns, rather than individual records as in SQLite, which eliminates the need for record headers. Furthermore, DuckDB's columnar storage format allows efficient representation and compression of data. For example, a Boolean value can be stored as a single bit in DuckDB. In contrast, SQLite represents a Boolean value with a 1-byte integer. In Table 4, we compare the space required by SQLite and DuckDB to store some of the datasets used in this paper. The TATP dataset contains 1 million subscriber records. The SSB dataset is scale factor 5. In addition, we report the time required to load an SSB dataset, which is organized as a collection of CSV files, into the database. Compared to DuckDB, SQLite requires 90% and 60% more space to store TATP and SSB, respectively.

Interestingly, SQLite is almost 20% faster at loading the SSB dataset from CSV. However, we were surprised by the time required to load CSV data into either system. DuckDB only takes about 7 seconds to run all of SSB, but requires 100 seconds to load the data. SQLite takes 51 seconds to run all of SSB, but requires 82 seconds to load the data. Given the popularity of the CSV format in the data science field, these results motivate further exploration into how CSV loading can be accelerated.

Table 3: Library footprints.

System	Library size	Compile time	Compile memory
SQLite (`-Os`)	900 KB	15 s	340 MB
SQLite (`-O3`)	1.5 MB	30 s	380 MB
DuckDB (`-Os`)	32 MB	5 m	7.7 GB
DuckDB (`-O3`)	37 MB	10 m	7.6 GB

Table 4: Database footprints.

System	TATP size	SSB size	SSB load time
SQLite	520 MB	2.8 GB	82 s
DuckDB	270 MB	1.8 GB	100 s

5 FUTURE DEVELOPMENT

The developers intend to provide support for SQLite through the year 2050, and design decisions are made accordingly. SQLite's code and database file format are fully cross-platform, ensuring that SQLite can run on any current or future platform with an 8-bit byte, two's complement 32-bit and 64-bit integers, and a C compiler. Every machine-code branch in the SQLite library is tested with multiple platforms and compilers, which makes the code robust for future migrations. SQLite is also extensively documented and commented, which helps new developers quickly understand SQLite's architecture. Finally, the developers work hard to evaluate new programming trends based on merit rather than popularity [26].

While the performance gap has narrowed as a result of this work, DuckDB is still considerably faster than SQLite on SSB. This is somewhat expected; SQLite is a general-purpose database engine, whereas DuckDB is designed from the ground up for efficient OLAP. Although SQLite's OLAP performance could be further improved in future work, there are several constraints that potential modifications to SQLite must satisfy. First, modifications should cause no significant performance regression across the broad range of workloads served by SQLite. Second, the benefit of an optimization must be weighed against its impact on the size of the source code and the compiled library. Finally, modifications should not break SQLite's backwards compatibility with previous versions and cross-compatibility with different machine architectures. Although SQLite's performance is a key priority, it must be balanced with these (sometimes competing) goals. We considered several means of improving value extraction in SQLite, but no single solution satisfied all the constraints above. For example, changing the data format from row-oriented to column-oriented would streamline value extraction, but it would also likely increase overhead for OLTP workloads. Moreover, drastic changes to the data format are at odds with SQLite's goal of stability for the database file format.

An alternative approach to improving SQLite's OLAP performance is a separate, yet tightly connected query engine that evaluates analytical queries on its own copy of the data, while SQLite continues to serve transactional requests, ensuring that the analytical engine stays up to date with the freshest data. If the extra space overhead is acceptable, the specialized analytical engine can provide substantial OLAP performance gains. This design has been successfully implemented in SQLite3/HE [46], a query acceleration path for analytics in SQLite. SQLite3/HE achieves speedups of over 100X on SSB with no degradation in OLTP performance. However, the current implementation of SQLite3/HE does not persist columnar data to storage and is designed to be used in a single process. Future work may explore similar approaches without these limitations.

SQLite's continued development is aided by informative profiling utilities and aggressive testing. In part, this paper can be viewed as a case study on adapting a database engine to a type of workload for which it was not originally and explicitly designed. Considering a conventional OLTP database engine as a starting point, there are several methods that are well-known to boost OLAP performance, including columnar storage, compression, operators over compressed data, vectorized execution, runtime code generation, and small materialized aggregates. However, without first understanding the bottlenecks of the system, it is difficult to estimate the magnitude of each method's benefit. As a result, it is unclear which method should be prioritized. For example, one might reasonably predict that SQLite would benefit from vectorized execution, which DuckDB uses to reduce the overhead of runtime query interpretation. However, our profiling analysis revealed that SQLite spends little time in query interpretation relative to B-tree probes and value extraction, and thus vectorization is unlikely to have an appreciable impact. As this work progressed, we grew to appreciate the ease of profiling SQLite's execution engine. Its architecture enabled us to pinpoint which virtual instructions were responsible for slowing down performance. We could then target our optimizations accordingly. Furthermore, SQLite's extensive test suite allowed us to quickly integrate our optimizations into a release build with little worry of breaking other components of the library. SQLite includes over 600 lines of test code for every one line of library code, and its tests cover 100% of machine code branches in the library [25]. Although SQLite cannot claim to be bug-free, its aggressive testing drastically reduces the chance of a bug appearing, which generally enables the developers to rapidly implement new features with high confidence.

6 CONCLUSION

The widespread deployment of SQLite is likely a result of its cross-platform code and file format, compact and self-contained library, extensive testing, and low overhead. While SQLite is a general-purpose database engine, it is primarily designed for efficient OLTP. Recognizing the potential for performance gains on in-process OLAP workloads, DuckDB recently emerged as an embeddable database engine specialized for analytics. In this paper, we presented a thorough evaluation of SQLite and DuckDB on a diverse set of benchmarks. Our analysis uncovered specific bottlenecks responsible for slowing SQLite's OLAP performance. We discussed the tradeoffs and feasibility of potential solutions. Our selected optimizations, which have been integrated into the current version of SQLite, resulted in up to 4.2X speedup on SSB. We also evaluated additional considerations important to embeddable database engines, including the resource footprint of the library. Finally, we discussed future directions that could further improve SQLite's performance without sacrificing its portability, compactness, and reliability.

ACKNOWLEDGMENTS

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA. Additional support was provided by the National Science Foundation (NSF) under grant OAC-1835446.

REFERENCES

[1] 2021. *Recommended Formats Statement.* Technical Report. United States Library of Congress.

[2] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-Efficient Hash Joins. In *Proceedings of the VLDB Endowment (PVLDB)*, Vol. 8. 353–364. https://doi.org/10.14778/2735496.2735499

[3] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[4] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1993. Applying Hash Filters to Improving the Execution of Bushy Trees. In *19th International Conference on Very Large Databases.* 506–516.

[5] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1997. On applying hash filters to improving the execution of multi-join queries. *The VLDB Journal* 6, 2 (1997), 121–131. https://doi.org/10.1007/s007780050036

[6] Ming-Syan Chen and Philip S. Yu. 1992. Interleaving a join sequence with semijoins in distributed query processing. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3. 611–621. https://doi.org/10.1109/71.159044

[7] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-Aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 2011–2026. https://doi.org/10.1145/3318464.3389769

[8] DuckDB. [n.d.]. Continuous Benchmarking. https://duckdb.org/benchmarks/.

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[10] William Endress. 2013. *On-line Analytic Processing with Oracle Database 12c*. Technical Report. Oracle Corporation. https://www.oracle.com/technetwork/database/options/olap/olap-wp-12c-1896136.pdf

[11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA Database - Data Management for Modern Business Applications. In *SIGMOD Record*, Vol. 40. 45–51. https://doi.org/10.1145/2094114.2094126

[12] Raspberry Pi Foundation. [n.d.]. About us. https://www.raspberrypi.org/about/.

[13] Raspberry Pi Foundation. [n.d.]. Raspberry Pi 4. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[14] Raspberry Pi Foundation. [n.d.]. Raspberry Pi 4 Tech Specs. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[15] Gerhard Häring. [n.d.]. sqlite3 — DB-API 2.0 interface for SQLite databases. https://docs.python.org/3.10/library/sqlite3.html.

[16] D. Richard Hipp. [n.d.]. 35% Faster Than The Filesystem. https://www.sqlite.org/fasterthanfs.html.

[17] D. Richard Hipp. [n.d.]. About SQLite. https://www.sqlite.org/about.html.

[18] D. Richard Hipp. [n.d.]. The Advantages Of Flexible Typing. https://www.sqlite.org/flextypegood.html.

[19] D. Richard Hipp. [n.d.]. Appropriate Uses For SQLite. https://www.sqlite.org/whentouse.html.

[20] D. Richard Hipp. [n.d.]. Architecture of SQLite. https://www.sqlite.org/arch.html.

[21] D. Richard Hipp. [n.d.]. Atomic Commit In SQLite. https://sqlite.org/atomiccommit.html.

[22] D. Richard Hipp. [n.d.]. Compile-time Options. https://www.sqlite.org/compile.html.

[23] D. Richard Hipp. [n.d.]. Database File Format. https://www.sqlite.org/fileformat.html.

[24] D. Richard Hipp. [n.d.]. Datatypes In SQLite. https://www.sqlite.org/datatype3.html.

[25] D. Richard Hipp. [n.d.]. How SQLite Is Tested. https://www.sqlite.org/test.html.

[26] D. Richard Hipp. [n.d.]. Long Term Support. https://sqlite.org/lts.html.

[27] D. Richard Hipp. [n.d.]. Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html.

[28] D. Richard Hipp. [n.d.]. The Next-Generation Query Planner. https://www.sqlite.org/queryplanner-ng.html.

[29] D. Richard Hipp. [n.d.]. Release History. https://www.sqlite.org/changes.html.

[30] D. Richard Hipp. [n.d.]. Single-file Cross-platform Database. https://www.sqlite.org/onefile.html.

[31] D. Richard Hipp. [n.d.]. The SQLite Amalgamation. https://www.sqlite.org/amalgamation.html.

[32] D. Richard Hipp. [n.d.]. The SQLite Bytecode Engine. https://www.sqlite.org/opcode.html.

[33] D. Richard Hipp. [n.d.]. SQLite Is Serverless. https://www.sqlite.org/serverless.html.

[34] D. Richard Hipp. [n.d.]. SQLite is Transactional. https://www.sqlite.org/transactional.html.

[35] D. Richard Hipp. [n.d.]. TH3. https://sqlite.org/th3.html.

[36] D. Richard Hipp. [n.d.]. Well-Known Users of SQLite. https://www.sqlite.org/famous.html.

[37] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, Vol. 35. 40–45.

[38] Kaggle. [n.d.]. Datasets. https://www.kaggle.com/docs/datasets.

[39] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. 2015. Pocket Data: The Need for TPC-MOBILE. In *Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*.

[40] Motorola Microprocessor and Memory Technologies Group. [n.d.]. Product Brief Integrated Portable System Processor - DragonBall (TM). https://www.nxp.com/docs/en/product-brief/MC68328P.pdf.

[41] Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press.

[42] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2011. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net

[43] Pat O'Neil, Betty O'Neil, and Xuedong Chen. 2009. Star Schema Benchmark. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF

[44] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-up Approach. In *Proceedings of the VLDB Endowment (PVLDB)*. 663–676. https://doi.org/10.14778/3184470.3184471

[45] CoRecursive Podcast. [n.d.]. 066: The Untold Story of SQLite. https://corecursive.com/066-sqlite-with-richard-hipp/.

[46] Martin Prammer, Suryadev Sahadevan Rajesh, Junda Chen, and Jignesh M. Patel. 2022. Introducing a Query Acceleration Path for Analytics in SQLite3. In *12th Annual Conference on Innovative Data Systems Research (CIDR '22)*.

[47] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984.

[48] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[49] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[50] Transaction Processing Performance Council (TPC). 2015. TPC Benchmark E. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-e_v1.14.0.pdf

[51] Transaction Processing Performance Council (TPC). 2021. TPC Benchmark H. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf

[52] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a

[53] Marianne Winslett and Vanessa Braganholo. 2019. Richard Hipp Speaks Out on SQLite. In *ACM SIGMOD Record*, Vol. 48. 39–46. Issue 2. https://doi.org/10.1145/3377330.3377338

[54] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. In *Proceedings of the VLDB Endowment (PVLDB)*, Vol. 10. 889–900. https://doi.org/10.14778/3090163.3090167