# Introducing a Query Acceleration Path for Analytics in SQLite3

Martin Prammer, Suryadev Sahadevan Rajesh, Junda Chen, Jignesh M. Patel
Department of Computer Sciences
University of Wisconsin - Madison
Madison, WI, USA
{prammer,jignesh}@cs.wisc.edu,{sahadevanraj,jchen693}@wisc.edu

## ABSTRACT

As large scale data processing becomes an ever more prominent component of modern computing tasks, databases now exist as a fundamental necessity of most computational platforms. However, in many cases there exists a disparity between the specializations of database management systems and the needs of the applications that run on them. The distinction between transactional and analytical workloads for databases has been well established, but not fully addressed within the space of the most widely used embedded database system, namely SQLite3. To overcome this shortcoming, we implement SQLite3/HE, an analytical database engine implemented as an alternative execution path for SQLite. Through the utilization of an additional, complementary storage layer, SQLite3/HE transforms SQLite into a hybrid database system, able to fully leverage the benefits of both row and columnar storage layouts. SQLite3/HE improves the performance of analytical queries in the 100x-1000x speedup range, at no cost to the existing transactional query performance. These results validate the decision to implement SQLite3/HE as an alternative execution path, enabling it to serve as a drop-in replacement for SQLite3 in existing systems.

## 1 INTRODUCTION

SQLite [5], a monumentally successful relational database management system (DBMS), serves as the database of choice for embedded systems. SQLite's high performance on these systems is driven by a number of deliberate design decisions, which together emphasize hardware compatibility and low software overhead. However, these decisions come at a significant cost: while SQLite functions well for transaction-focused workloads in a lightweight environment, these same optimizations significantly impede its analytical query performance.

One can imagine a small computer placed inside a wind turbine generator ("windmill"), connected to an array of sensors that monitor performance, maintenance, and other concerns. Similar to many other real world applications that benefit from the local availability of compute resources, this example of a windmill is a member of a broad class of edge devices applications [13]. Naturally, the sensor data would benefit from being stored in a query-accessible format, allowing seamless integration of both the recording of sensor readings and metadata. This in turn would facilitate real-time analysis of the data, enabling the tracking of long-term electricity generation trends. However, edge computing cases have three key differences when compared to traditional SQLite-powered embedded systems.

First, the data analysis tasks in the edge computing domain require highly efficient performance for both transaction- and analytics-focused workloads, as the data logging and analysis tasks are usually performed concurrently. This change in use case reflects a significant divergence from the historical needs of embedded systems: edge devices usually need to perform a number of concurrent tasks, usually through the cooperation of an operating system scheduler and hardware parallelism.

Next, the compute resources available to the devices in this domain have grown significantly over the last decade, blurring the boundary between a small personal computer and a traditional embedded device. These devices are now regularly equipped with multiple processing cores and relatively large amounts of memory. Likewise, each of these cores comes equipped with hardware support for vectorized instructions. Together, these advancements have drastically improved the computing power available to even the smallest of devices in this domain.

Finally, many edge devices are powered by a traditional computational stack, complete with an operating system and utility software. While these devices do share many constraints with the older class of traditional embedded devices, other factors such as executable image size are significantly less of a problem. These edge devices demonstrate the shifting of computational paradigms; for example, they prioritize portability and maintainability, to the point where a performance benefit may not be worth the cost of sacrificing either.

Still, SQLite has demonstrated that it deserves a place in modern DBMS offerings, usually as the database of choice for transactional workloads run on embedded devices. Thus, we find it highly relevant to provide a mechanism for SQLite to perform comparably to the state-of-the-art when processing analytical queries. To this end, we have developed SQLite3 Hustle Edition (SQLite3/HE), a query acceleration path that is able to maximize machine resource utilization leading to significant improvements in SQLite's analytical query processing performance [15]. This performance improvement is driven by the usage of both an in-memory, column-store and a set of analytics-optimized query operators. Most notably, the analytical query acceleration path is only taken by queries that benefit from such acceleration. Thus, SQLite3/HE is able to function as a drop-in augmentation for SQLite3 without fear of performance degradation in existing workloads.

In this paper, we make the following contributions:

1. An analysis of the technical behavior of SQLite3. This analysis explores the deliberate design decisions that, while appropriate for the older class of traditional embedded devices, have created the need for SQLite3/HE.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well as allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2022.
*12th Annual Conference on Innovative Data Systems Research.*
*CIDR '22, January 9–12, 2022, Chaminade CA, USA*
.

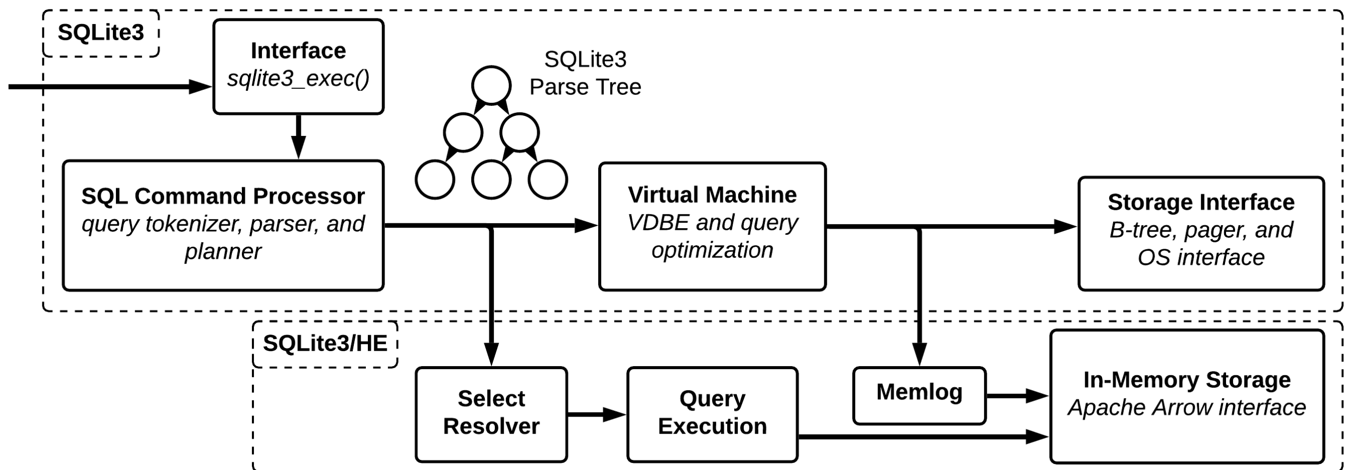


**Figure 1: The SQLite3/HE Acceleration Path. After SQLite3 has generated the query parse tree, analytical queries are branched off from the existing SQLite3 execution path while transactional queries continue to be executed by SQLite3.**

(2) An exploration into the implementation of SQLite3/HE, detailing the mechanisms that support both its in-memory database and its analytics-optimized query operators.
(3) An analysis of SQLite3/HE's performance, both in analytical and transactional workloads, which demonstrates its effectiveness as a drop-in add-on for existing SQLite3 implementations.

## 2 BACKGROUND

SQLite3/HE utilizes many existing, state-of-the-art techniques to improve analytical query performance. Most notably is its usage of a secondary storage layer, facilitating a fundamentally different set of performance behaviors than that of SQLite. We explore this and related topics in this section.

The practice of optimizing a database for read-focused queries has been well established in recent years [7, 14]. One of the most prominent mechanisms to achieve high read performance is to "transpose" the database, a process in which fields of a single column are placed adjacent in the storage layer instead of preserving locality within a single record. This storage format provides two major optimizations:

(1) The tuple members that are unrelated to the current query are not accessed (which would be the case if an entire row needs to be read, e.g., due to hardware limitations).
(2) The tuple members that are related to the current query are placed consecutively in storage, naturally aligning their sequential access with existing hardware-based optimizations.

Given these advantages, this storage format is more amenable towards calculating aggregate values from a subset of the data. Likewise, this storage format is also highly compatible with machine specific optimizations, such as vector instructions and hardware prefetching.

However, the two previously mentioned optimizations are inverted when discussing either writes or accessing entire tuples. Further, when considering the writing of tuples, we find that we achieve these optimizations by utilizing a row-based storage method, allowing for a write to act similarly to a simple append operation. Thus, we realize that for our particular windmill example, there is a need for both kinds of storage formats. To this end, and as part of a larger body of research, we see the benefit of, broadly, "hybrid stores" [10]. Of course, preserving the benefits of each storage solution simultaneously is both a difficult and an application-specific feat, especially without causing a significant amount of software-related overhead. To this end, only recently have these systems begun to enter the embedded device space, which imposes not only the aforementioned limitations but also further restrictions by nature of limited machine resources.

## 3 SQLITE

In order to fully explore the purpose of specific design choices within SQLite3/HE, we must first examine SQLite.

### 3.1 SQLite3 Implementation

As SQLite3/HE is implemented as an acceleration path for specific queries, it is important to understand the behavior of SQLite and how SQLite3/HE fits into the existing query processing pipeline. To accomplish this, we explore specific SQLite3 components related to SQLite3/HE's acceleration path.

*3.1.1 B-Tree Storage Format.* SQLite primarily utilizes a single, monolithic database file; while this approximation is not entirely accurate, SQLite can be thought of as a sort of "SQL Interface" for a flat file database. This design paradigm benefits resource-constrained environments, allowing SQLite to fine-tune its data storage mechanisms without placing a significant burden on the operating system's file management system. However, given the unwieldiness of using a single file as a database, SQLite3 utilizes a page-based, B-tree structure: rows are stored as individual pages at the leaf level, which are then organized into a B-tree. This implementation functions incredibly well for the core set of applications that SQLite targets. However, as scanning a column requires a large

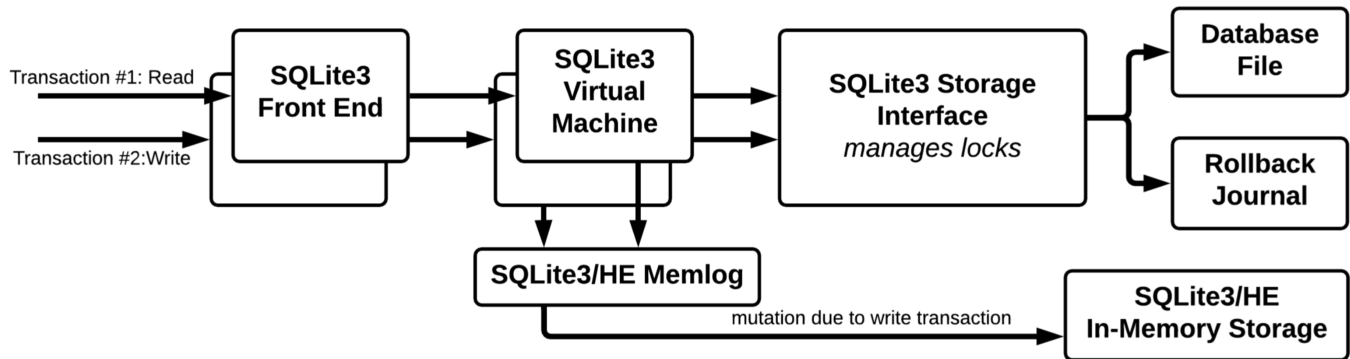


**Figure 2: Transaction Flow in SQLite3/HE, depicting the movement of data during the processing of multiple concurrent queries.**

number of page walks, this fundamental design choice creates a significant performance bottleneck for the analytical query execution path.

*3.1.2 The Query Pipeline.* Similar to other DBMS solutions, SQLite utilizes a query pipeline structure composed of a number of familiar steps: specifically, the tokenizer, parser, planner, optimizer, and executor. Internally, the Lemon parser generator [6] creates a parser that performs both the parsing and the planning tasks, receiving tokens from the tokenizer to construct a parse tree. However, the query optimizer is heavily connected to the translation unit for the Virtual Database Engine, and thus is discussed separately. In general, this compartmentalization of work creates the opportunity to intercept queries as they progress through the query processing pipeline. SQLite3/HE intercepts queries once the parse tree has been constructed and before they reach the optimization-translation unit.

*3.1.3 Virtual Database Engine.* SQLite transforms queries into a format reminiscent of assembly instructions. Virtual Database Engine (VDBE) instructions consist of an opcode and a number of operands (5 in SQLite3, though previous versions had fewer). These VDBE instructions assist in maintaining a high degree of portability for SQLite, as a system-specific implementation need only implement the machine instructions for each VDBE instruction. Unfortunately, this design creates a number of problems when integrating modern high-performance query optimization methods into SQLite3.

SQLite3 performs major query optimizations during the process of query translation, as the "code generator" component handles both tasks simultaneously. While the blurring of program responsibilities usually does not present a problem, it prevents other components from accessing the optimized query plan; the code generator only outputs the optimized VDBE instructions, preventing access to an optimized version of the input query plan. Thus, SQLite3's compartmentalization of query steps posed a significant design challenge for SQLite3/HE, and was one of the most important considerations for how SQLite3/HE works in tandem with SQLite3. Furthermore, as the instructions have already broken down the query plan into a number of discrete steps, it is difficult to recombine these components into a less optimized query plan. There is no feasible way to improve or reorient the existing level of query optimization, which prevents the application of analytics targeted optimizations after the query has been translated to VDBE instructions.

Additionally, optimizations are applied throughout the VDBE instruction generation process. Traditional optimizations such as constant propagation and push-down mechanisms are introduced before and during the VDBE instruction translation process. However, index-based optimizations occur later on, after the VDBE instructions are generated. Most notably is the usage of indices during nested loop joins, the join algorithm of choice for SQLite3. Of course, the decision to exclusively use nested loop joins creates additional problems for analytical queries. In general, while SQLite's optimizations are suitable for the general use case of embedded applications and related systems, some decisions that SQLite makes are contrary to the existing research involving high performance analytical query optimizations. These optimizations generally favor columnar methods with access patterns that are amenable to vectorization, further establishing the benefits of the presence SQLite3/HE's secondary columnar storage layer.

## 3.2 SQLite Optimizations

In addition to the aforementioned design choices, SQLite3 also utilizes a number of smaller optimizations that are relevant to analytical query acceleration. For example, the "record format" used to store each row makes a number of optimizations. One such optimization involves the header specifying the most efficient type possible for each value in the body of the record. Not only does SQLite3 include 24-bit and 48-bit integer types to assist in saving space, some types represent a field having an integer value of 1 or 0. In the value-specified type case, the body does not contain a value; thus, it is possible for some records to have a header and no body.

SQLite's row format is part of a larger set of optimizations that reduce the size of values stored. Another example is the usage of variable sized integers elsewhere, encoded as the "varint" type. In general, SQLite3 makes a significant effort towards compressing the values it stores. This storage paradigm is effective in append-only, transaction-recording workloads; however, the added levels of interdiction and conversion causes significant amounts of overhead for reads. Thus, a new storage mechanism must be introduced to bypass this fundamental incompatibility with state-of-the-art analytical query optimizations.

# 4 SQLITE3/HE ACCELERATION LAYER

Given the previous exploration of SQLite3's design, we now have the context to understand the existing obstacles towards using SQLite3 as a high-performance database platform for analytical queries.

## 4.1 SQLite3/HE Integration

SQLite3/HE is implemented as an alternate query execution path, escaping the existing execution path between the SQLite3 Query Planner and VDBE Code Generator (Figure 1). SQLite3/HE functions primarily by providing an in-memory columnar database that is used as the foundation for analytical query acceleration. Due to this separate storage layer, a new set of operators were developed to facilitate query execution on a secondary query execution path, which we explore in detail in the following subsections.

*4.1.1 Optional Acceleration Path.* While Figure 1 depicts the acceleration path, the mechanisms that fully enable this behavior must be detailed at a finer degree. The SQLite3 query parser and planner behave as expected, with the addition of a limited set of optimizations that they apply such as nested sub-query flattening. These optimizations are all performed in-place on the parse tree, which can then be intercepted by SQLite3/HE before it is transformed into VDBE instructions. SQLite3/HE intercepts all queries before they are sent to the SQLite3 code generator unit. If the query is determined to be SQLite3/HE compatible (generally, a read/analytics-focused query), the query will instead be routed through the SQLite3/HE query acceleration path, and will no longer execute in SQLite3.

In particular, there are three rules that dictate if a query will be accelerated by SQLite3/HE:

(1) The query is a supported SELECT or JOIN query.
(2) All source tables are present in memory.
(3) All operators used in the parse tree are supported by SQLite3/HE.

These rules allow for common analytical queries to be processed by SQLite3/HE. Parser flags for the SELECT query and the structure of the JOIN query determine if SQLite3/HE supports accelerating each respective query type. Specifically, JOIN queries must fall under the general structural archetypes of "star-join" or "chain-join" based on how the tables are joined together. We utilize existing methods to both categorize and optimize these kinds of joins [4]. Once a JOIN query has been placed in the acceleration path, we utilize Lookahead Information Passing (LIP) [16] to optimize the JOIN query plan beyond the optimizations of SQLite3, resulting in a more robust and efficient JOIN query implementation.

Write queries are not supported by the SQLite3/HE acceleration path, and thereby fall back to the original SQLite3 query execution path.

*4.1.2 In-Memory Columnar Storage Layer.* SQLite3/HE utilizes Apache Arrow [3] as an in-memory columnar storage layer. This external dependency enables future cross-application compatibility, but is chiefly motivated due to Apache Arrow's strong implementation of efficient memory access and vectorization-friendly compute kernels. Thus, SQLite3/HE is able to focus on query optimization while still preserving all the benefits of a machine-optimized set of compute kernels.

Apache Arrow arrays are implemented as "chunked arrays," —arrays partitioned into a number of chunks of contiguous memory. This design choice enables a fairly simple process for resizing, either by deleting excess chunks or appending additional chunks. While the chunking of an array is accompanied by additional overhead, it also bolsters overall scalability and is an acceptable trade-off for the robust set of features that Apache Arrow brings. In addition, the chunked array approach facilitates an extra layer of parallelism from the perspective of operators, as allocating work based on chunks presents simple and efficient opportunities for parallelism.

*4.1.3 Query Operators.* SQLite3/HE's utilization of Apache Arrow for its storage layer allows for a significantly expedited engineering effort. Apache Arrow is primarily concerned with high-performance data storage and retrieval, providing a number of performant compute kernels. By utilizing these kernels and the Apache Arrow API as a whole, SQLite3/HE capitalizes on the performance and long term support guarantees that Apache Arrow provides. Likewise, because of this existing body of work, the engineering effort behind SQLite3/HE has been able to focus on optimizing query operators, further improving SQLite3/HE as a whole.

## 4.2 Data Consistency

The implementation of a secondary storage layer requires mechanisms to maintain coherence between the SQLite3 and SQLite3/HE storage layers, especially in the case of the SQLite3 database receiving a write transaction. The primary mechanism in SQLite3/HE that facilitates this behavior is the "memlog."

The memlog records the writes that are made to the SQLite database file whenever transactions are committed by the SQLite3 query execution path. The memlog structure is depicted in Figure 2, within the context of two concurrent queries being processed at the same time. Note that due to SQLite3's database file-locking mechanisms, the memlog is presented with an ordered collection of changes to propagate from SQLite3 to the SQLite3/HE storage layer. Thus, SQLite3/HE is able to exploit the already existing ordering of mutations as they are made to the SQLite3 database.

The memlog is constructed as a collection of ordered write logs that must be propagated to SQLite3/HE columns, where each table is linked with a queue-like structure to contain its write logs. Each write log includes information regarding the particular mode of the write (INSERT, UPDATE, or DELETE), the targeted row ID, and the relevant data if appropriate. These writes are accumulated over time, to eventually be processed in bulk depending on the mode of write propagation used by SQLite3/HE. We define these modes as "eager," "lazy," and "async background," each of which behaves as follows: eager writes perform updates during the transaction, lazy writes only make changes when the table is next read by a query, and async background writes use a secondary thread to perform a lazy write (as the worker threads also process SQLite3 lock contention, these background writes are considered asynchronous). We evaluate the behavior of each respective write mode as part of the evaluation of SQLite3/HE's transactional performance.

In the case of a crash or some other failure, the SQLite3/HE tables are regenerated from the SQLite3 database. This behavior is unavoidable due to SQLite3/HE's existence as an in-memory storage layer. That said, the reliance on SQLite3's error recovery

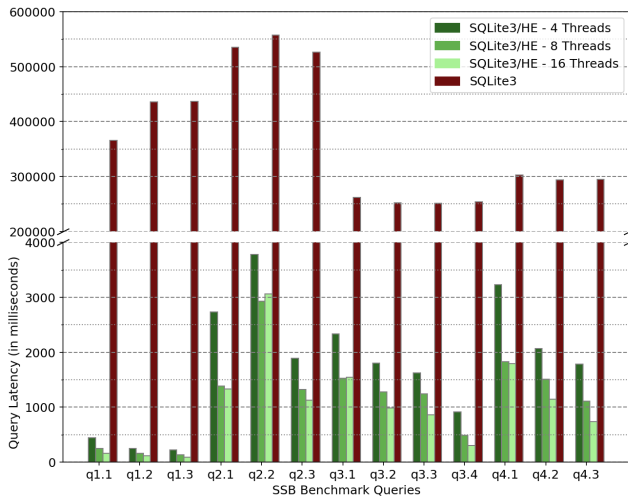


**Figure 3: Analytical workload performance improvement due to Hustle, as compared between different amounts of thread-level parallelism.**

mechanisms to regenerate data does provide the standard set of benefits which accompany a well-supported library.

## 5 EVALUATION

Our evaluation of SQLite3/HE focuses on three primary questions:

(1) What is the performance gain of SQLite3/HE for analytical queries?
(2) Do any of the optimizations implemented carry over to transactional workloads? In particular, does the write propagation strategy have a significant impact on performance?
(3) The secondary acceleration path incurs additional data movement costs when writing data to the SQLite3/HE columns –what is the overhead?

We utilized two different benchmarks as part of our evaluation: TATP [8] for write-focused, transactional workloads, and SSB [9] for read-focused, analytical workloads. All experiments were performed using SQLite3's default configuration parameters.

The experiments were run by a CloudLab "c220g5" machine [1]. This node consists of two Intel Xeon Silver 4114 processors each with a clock speed of 2.2 GHz. Each processor has 10 cores with two threads, for a total of 20 cores and 40 threads.

### 5.1 Analytical Workload Evaluation

We depict the SQLite3/HE results for each SSB query in Figure 3. SQLite3/HE completes all SSB (scale factor 10) queries in under 4 seconds, as compared to SQLite3 which requires at least 250 seconds at best. Query to query, SQLite3/HE boasts a speedup consistently over 100x, though some queries are closer to a 1000x improvement. The previously mentioned bottlenecks of SQLite3 have a significant influence on the results. Likewise, these queries are where SQLite3/HE is able to fully utilize the cutting edge of analytical query optimization, such as its in-memory columnar store, operation vectorization, LIP, and intra-operator parallelism. As these features are not supported by SQLite3, SQLite3/HE's demonstrated performance gain is a reflection of its employment of state-of-the-art techniques. As SQLite3/HE supports thread-level parallelism, we evaluate SSB performance using 4, 8, and 16 threads. Unsurprisingly, the execution time of the queries is significantly reduced as more threads are utilized, due to large amounts of intra-operator parallelism. However, the speedup from additional threads has diminishing returns. As the provisioned CloudLab machine is a two-socket, 10 cores (20 threads) per socket system, we begin to see the impact of increased coordination and data movement between cores, exacerbated by the need for the CPUs to more regularly utilize L3 cache coherence mechanisms. This performance behavior is perfectly reasonable, as using a fixed thread count to divide the work into a number of partitions is expected to require some amount of optimization specific to both the workload and the underlying machine.

### 5.2 Transactional Workload Evaluation

To evaluate our integration with SQLite3 for transactional workloads, we use the TATP benchmark. In both TATP experiments, we use a database containing 2 million subscriber records and the recommended default parameters. We vary the percentage of write transactions present to generate a read-heavy workload (80% reads, 20% writes) and a write-heavy workload (50% reads, 50% writes). These results are depicted in Figures 4 and 5 respectively. Once again, we use SQLite3's query throughput as a performance baseline. As previously mentioned, all write transactions primarily interact with the SQLite DB file. After this initial write, we perform a secondary write to the SQLite3/HE storage at some (write mode-determined) point preceding the next read to the modified field. Due to the secondary write being performed, we expected the writes to have slightly higher latency compared to the naive SQLite3. However, given that reads in SQLite3/HE will be significantly faster than that in SQLite3, overall performance is expected to increase.

These expectations were justified, as SQLite3/HE offers a reasonable improvement over baseline SQLite3. While dependent on the number of workers and the particular write mode used, an improvement of up to 20% is possible. Most configuration and workload combinations will experience a 5%-10% improvement. Due to significant lock contention issues inherent in the SQLite3 write process, there is little room for performance gain in the presence of this heavy bottleneck, and thus the proportion of reads to the overall workload dictates the maximum expected performance improvement.

## 6 DISCUSSION

Our evaluation demonstrates that SQLite3/HE accomplishes all performance improvements set forth. SQLite3 is purpose-built to handle transactional-query workloads in the resource constrained environment of an embedded device, and the SQLite3/HE acceleration path augments SQLite3 with state-of-the-art analytical query processing capabilities. By relaxing many of SQLite3's hardware-related constraints, SQLite3/HE is able to fully leverage the underlying hardware capabilities. This hardware-conscious behavior has become increasingly important as modern edge devices continue to diverge from the older class of traditional embedded devices.

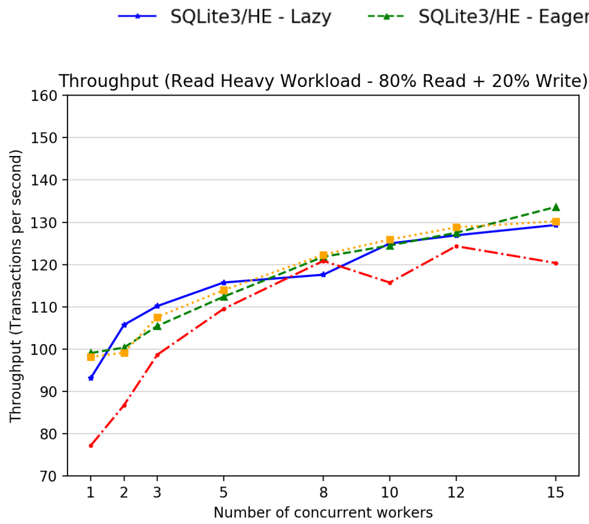


Figure 4: Transactional workload performance improvement due to Hustle, as compared between different write modes. This experiment utilized TATP in the default configuration.

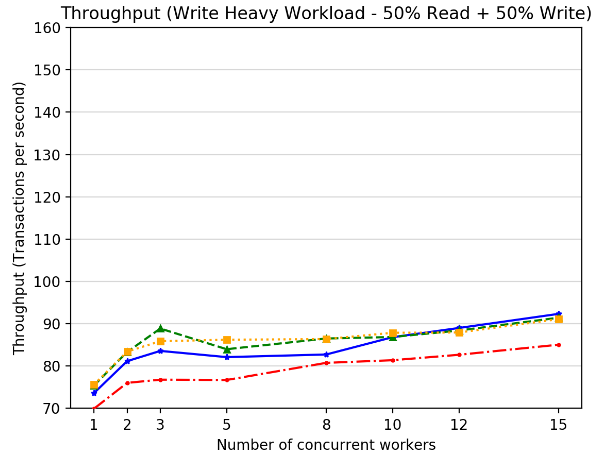


Figure 5: Transactional workload performance improvement due to Hustle, as compared between different write modes. This experiment altered the ratio of reads and writes in TATP, increasing the number of writes per read.

A critical aspect of our design is the ability for SQLite3/HE to function as a drop-in replacement for SQLite3. As demonstrated by the TATP experiments, the combination of careful write management and significant read-performance improvements allow SQLite3/HE to be used as an analytical query accelerator in existing SQLite3 implementations at no performance cost. Likewise, SQLite3/HE's impressive analytical workload performance facilitates a wider variety of database-driven applications, such as on-site, live data analytics (as in the original windmill example).

Many modern databases have both transactional and analytical query processing needs. Significantly optimizing one category "for free" by more efficiently leveraging existing machine resources demonstrates SQLite3/HE's widespread applicability. A true cost that SQLite3/HE does impose on the host system is a slightly larger executable image size. While this is a valid consideration, the SQLite3/HE does not target applications and devices where minimizing executable image size is a primary concern.

## 7 RELATED WORK

The use of secondary acceleration paths has become increasingly relevant in recent years across the breadth of computing research. These acceleration paths exist at all levels, driven by advancements in both software and hardware [7, 11]. These paths consistently demonstrate the benefits of acceleration add-ons for existing systems, leveraging specialization to improve performance.

Improving the performance of an existing database structure to handle analytical queries has become a common "growing pain" within the field of database research. Significant amounts of prior work surround SQLite3/HE, each focusing on their own unique innovations [7, 12, 14]. Within this broad space, DuckDB [12] and SQLite3/HE target a similar need. While SQLite3/HE introduces an alternate query execution path, DuckDB implements much of the DBMS architecture "from scratch," opting to recreate or rely on external modules to facilitate their own query processing pipeline (such as *libpg_query* [2], the PostgreSQL Parser). DuckDB is an alternative approach to the overall problem that both SQLite3/HE and DuckDB address.

## 8 FUTURE WORK

While the benchmarks presented demonstrate wide-spread feasibility of SQLite3/HE, a larger discussion could be had on the inner workings of the implementation of SQLite3/HE's in-memory, columnar database. An examination into the particulars of how data consistency is achieved between the SQLite3 and SQLite3/HE databases would provide a deeper understanding of the true performance characteristics of SQLite3/HE's acceleration path. Specifically, while "write" and "read" are sufficient for the general case, evaluating the differences between INSERT, UPDATE, and DELETE operations would assist in the exploration of SQLite3/HE's low-level behavior.

## 9 CONCLUSION

Through the implementation of an alternative query execution pathway, SQLite3/HE augments SQLite3 with state-of-the-art analytical query processing capabilities, generating speedups at the 100x-1000x scale. These performance benefits come at no cost to the transactional query performance, enabling SQLite3/HE to function as a drop-in replacement for existing SQLite-powered applications. We view SQLite3/HE as a strong candidate for consideration in existing database platforms. As the technological needs of computing applications ever grows, so too do the performance demands placed on the underlying database systems. SQLite3/HE's implementation of state-of-the-art techniques satisfies these demands, advancing the ability for embedded systems to perform high-performance data analytics.

## 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.

The “CloudLab c220g5” machine utilized was provided as part of the CloudLab service [1].

## REFERENCES

[1] 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

[2] Lukas Fittl. 2017. libpg_query. https://github.com/pganalyze/libpg_query.

[3] Apache Software Foundation. 2019. Apache Arrow. https://arrow.apache.org.

[4] Cesar A. Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. 2008. Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server. *2008 IEEE 24th International Conference on Data Engineering* (2008), 1190–1199.

[5] Richard D Hipp et al. 2021. SQLite. https://www.sqlite.org/index.html

[6] Richard D Hipp et al. 2021. The Lemon LALR(1) Parser Generator. https://www.sqlite.org/lemon.html

[7] Yinan Li and Jignesh M. Patel. 2014. WideTable. *Proceedings of the VLDB Endowment* 7, 10 (June 2014), 907–918.

[8] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. [n.d.]. Telecommunication Application Transaction Processing (TATP) Benchmark. http://tatpbenchmark.sourceforge.net/.

[9] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[10] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM.

[11] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. 2015. Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM.

[12] 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.

[13] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct. 2016), 637–646.

[14] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.

[15] Hustle Development Team. 2021. Hustle. https://github.com/UWHustle/hustle/.

[16] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking ahead makes query plans robust. *Proceedings of the VLDB Endowment* 10, 8 (April 2017), 889–900.