SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database

Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, Jakub Szymaszek
Microsoft
Redmond, WA, USA
{panant, skaushi, hanumak, srosales, rewong, janders, jaszymas}@microsoft.com

ABSTRACT

SQL Ledger is a new technology that allows cryptographically verifying the integrity of relational data stored in Azure SQL Database and SQL Server. This is achieved by maintaining all historical data in the database and persisting its cryptographic (SHA-256) digests in an immutable, tamper-evident ledger. Digests representing the overall state of the ledger can then be extracted and stored outside of the RDBMS to protect the data from any attacker or high privileged user, including DBAs, system and cloud administrators. The ledger and the historical data are managed transparently, offering protection without any application changes. Historical data is maintained in a relational form to support SQL queries for auditing, forensics and other purposes. SQL Ledger provides cryptographic data integrity guarantees while maintaining the power, flexibility and performance of a commercial RDBMS. In contrast to Blockchain solutions that aim for full integrity, SQL Ledger offers a form of integrity protection known as Forward Integrity. The proposed technology is significantly cheaper and more secure than traditional solutions that establish trust based on audits or mediators, but also has substantial advantages over Blockchain solutions that are complex to deploy, lack data management capabilities and suffer in terms of performance due to their decentralized nature.

CCS CONCEPTS

• Information systems → DBMS engine architectures; • Security and privacy → Database and storage security;

KEYWORDS

Database Security; Integrity Protection; Data Verifiability; Ledger; Blockchain; Cryptographic Verifiability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457558

ACM Reference format:

Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason M. Anderson, Jakub Szymaszek. 2021. SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database. In *Proceedings of SIGMOD '21, June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457558

1 INTRODUCTION

Establishing trust around the integrity of data stored in database systems has been a long-standing problem for all organizations that manage financial, medical or other sensitive data. Systems that maintain such data are also known as Systems of Record (SOR) and need to guarantee the fidelity of their data for legal and compliance reasons. This is extremely challenging since attackers or high privileged users, such as DBAs or system administrators who have full control of the system, can easily tamper with the data and erase any traces of their actions. Additionally, as more organizations move their data into the cloud, the cloud provider and any operators also need to be trusted to maintain the integrity of the data they manage, significantly expanding the trust boundary.

Traditionally, organizations have depended on complicated, time-consuming monitoring and auditing solutions to detect unauthorized operations and certify that their systems meet the expected security standards. Alternatively, they might involve trusted intermediaries to host their sensitive data who in turn have gone through similar processes to establish trust. These solutions are expensive but also incomplete, as audits cannot fully model a system and discover all its vulnerabilities.

Blockchain systems, which were originally established by Bitcoin [24], introduce a revolutionary technology for guaranteeing the integrity of data and business logic in an untrusted environment. They form a network of participants who execute transactions using a decentralized, Byzantine fault-tolerant [14] consensus protocol and validate the correct execution of every transaction. The data is stored in various forms of tamper-evident data structures, such as Blockchains (linked lists of hashed blocks) or Merkle Trees [18], to allow each participant to efficiently verify that their state is consistent with the other participants. The Blockchain technology is designed for environments where there is no or very limited trust and only a majority of participants can be trusted. However, due to its decentralized nature, it presents significant challenges when used for real-world, production

workloads. Even state of the art Blockchain systems, such as Hyperledger Fabric [1], lack the advanced data management capabilities of an RDBMS, use custom programming languages that do not integrate with the existing tooling and reporting ecosystem and, more importantly, provide more than an order of magnitude lower throughput and higher latency than a commercial RDBMS.

While there are scenarios that merit the strong security guarantees that the Blockchain technology offers, there are many scenarios, especially around SORs, where using a Blockchain solution is an overkill and does not justify the cost and performance overhead. SQL Ledger is a new technology that addresses the latter scenarios by offering a different point in the security-cost/performance spectrum. It maintains the power, flexibility and performance of a commercial RDBMS and can be centrally managed, like a traditional RDBMS. However, it still offers strong security guarantees that are based on the notion of Forward Integrity [7]. Briefly, Forward Integrity assumes that the RDBMS is trusted until the time a transaction is processed and only offers protection against future attacks. We believe that Forward Integrity is sufficient for a wide class of real-world applications where the organization hosting the data is generally trusted but must be able to attest to the authenticity of the data and prove that it has not been tampered with. SQL Ledger achieves that by leveraging the core data structures from the Blockchain technology to capture the state of the database in compact, cryptographic digests. These digests can be stored outside of the database or shared with any parties that need to validate the integrity of the data and used at a later point in time to verify that the data has not been tampered with.

This paper describes the overall design of SQL Ledger and is organized as follows: Section 2 outlines the architecture of our system and the functionality exposed. Section 3 covers the detailed design of the various components. Section 4 presents our experimental results regarding the performance of the system. Finally, Section 5 discusses our ongoing work to extend the functionality of the system.

2 OVERVIEW

This section provides an overview of SQL Ledger and describes the main components of the system, as demonstrated in Figure 1. We also discuss the threat model and integrity guarantees of our technology.

2.1 Ledger Tables

SQL Ledger introduces a new type of tables, called Ledger tables, that is designed to protect the integrity of the underlying data without requiring any application changes. Ledger tables are relational tables that can use all SQL features that are available for regular tables. In addition, they allow users to detect malicious data modifications and cryptographically verify the data integrity:

- All historical data of Ledger tables is transparently maintained in the system and exposed to users for auditing and forensic purposes. Malicious users can update the content of Ledger tables, using regular DML operations, and tamper with the data. The historical data can be used to analyze the operations executed on and detect unexpected or malicious modifications.
- All latest and historical data of Ledger tables is cryptographically hashed, using the SHA-256 algorithm. This allows the system to capture the state of the data in a compact digest and use that to cryptographically verify its integrity later. Cryptographic verifiability protects data against sophisticated attacks where the adversary manages to bypass the earlier logic that audits modifications and preserves historical data. For example, by compromising the RDBMS process or overwriting the data directly at the storage layer.

There are two types of Ledger tables: append-only and updateable. Append-only tables only support insertions and are designed for auditing scenarios where updates and deletions are not allowed. Updateable tables allow all operations and can be used as regular tables by any SQL application. In the case of updateable tables, all earlier versions of a row are preserved in a secondary table, known as the History table, that mirrors the schema of the Ledger table. When a row is updated, the latest version of the row remains in the Ledger table, while its earlier version is inserted into the History table by the system, transparently to the application.

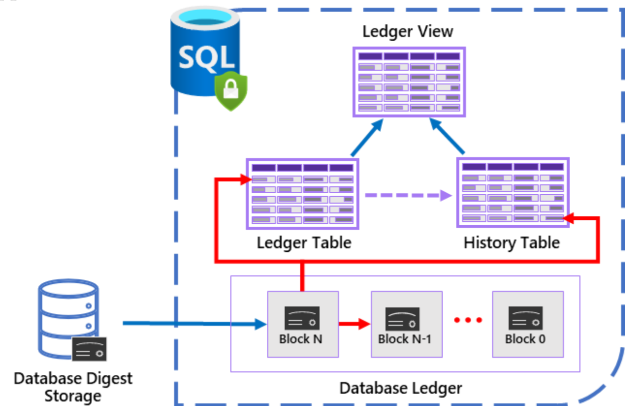

Figure 1. SQL Ledger architecture

For every Ledger table, the system automatically generates a view, called Ledger view, that reports all row modifications that have occurred on this table by leveraging the historical data in the History table. This enables users, their partners and auditors to analyze all historical operations and detect potential tampering. Each row operation is accompanied by the ID of the transaction that performed it, allowing users to retrieve more information about the time the transaction was executed, the identity of the user who executed it and correlate it to other operations performed by this transaction. Figure 2 provides an example of an updateable Ledger table with the corresponding History table and Ledger view.

Finally, all latest and historical row versions are cryptographically hashed, as they get updated, to capture the state of the Ledger

table. The corresponding digests are persisted in a tamper-evident, append-only data structure that is described in the next section.

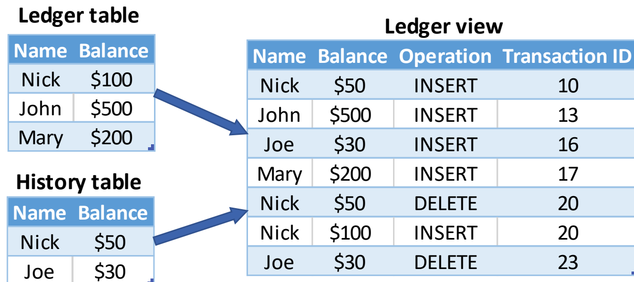

Figure 2. Sample updateable Ledger table, History table and Ledger view for a table storing account balances.

2.2 Database Ledger

A naïve approach for protecting the integrity of a Ledger table would be to periodically compute the SHA-256 hash of all its rows, including the rows of the corresponding History table to also protect any historical data. The generated digest would then be used at a later point to verify the integrity of the data by restoring the tables back to the time the digest was generated, recomputing the hash over their rows and, finally, verifying that the new hash matches the original digest. This might be practical for small tables but, when it comes to production workloads, with TBs of data that is continuously updated, the cost of computing the hash across the whole dataset frequently enough to provide actual protection is prohibitive.

To address this challenge, we leverage ideas from the Blockchain community and incrementally maintain the hashes of all Ledger tables in a Blockchain data structure, called Database Ledger. Since transactions are the unit of atomicity for any RDBMS, this is the unit of work being captured in the Database Ledger. Specifically, when a transaction commits, the SHA-256 hash of any Ledger table rows that were modified by this transaction, together with some metadata for the transaction, such as the identity of the user that executed it and its commit timestamp, are appended as a "transaction entry" in the Database Ledger. When several transactions get appended, we form a block that contains the entries of all transactions in this block, but also the hash of the previous block, forming a Blockchain. Using this technique, we only need to incrementally compute the hashes of rows being modified and not the overall dataset. The trade-off compared to the naïve approach is that our ledger needs to maintain information for all historical data and transactions; however, since our system maintains historical data to allow users to detect malicious modifications on Ledger tables, this is anyways necessary. Figure 3 presents the Database Ledger structure.

The hash of the latest block in the Database Ledger is known as the Database Digest and represents the state of all Ledger tables in the database at the time when this block was generated. Generating a Database Digest is extremely efficient since it only involves computing the hashes of the blocks that were recently appended. This allows users to extract a digest of the database state very frequently, even every second, and use it later for verifying the data integrity. SQL Ledger exposes an API for users to generate a Database Digest in the form of a JSON document that contains the hash of the latest block together with metadata regarding the block ID, the time the digest was generated and the commit timestamp of the last transaction in this block.

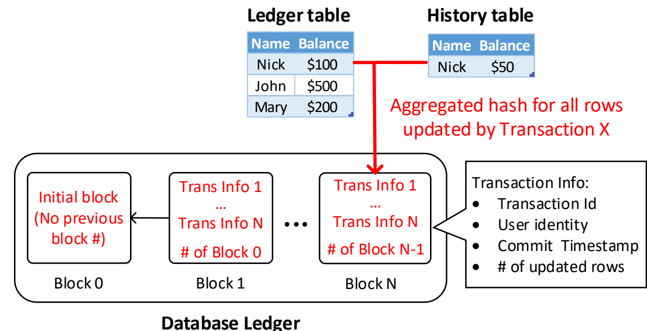

Figure 3. The Database Ledger.

2.3 Ledger Verification

Although our system does not allow users to modify the content of the ledger, an attacker or system administrator who has control of the machine can bypass all system checks and directly tamper with the data, for example by editing the database files in storage. SQL Ledger cannot prevent such attacks but guarantees that any tampering will be detected when the ledger data is verified. The Ledger Verification process takes as input one or more previously generated Database Digests and recomputes all the hashes stored in the Database Ledger based on the current state of Ledger tables. If the computed hashes do not match the input digests, the verification fails, indicating that the data has been tampered with, and reports all inconsistencies detected.

The verification process scans all Ledger and History tables and recomputes the SHA-256 hashes for their rows to verify the digests stored in the Database Ledger. It is, therefore, a resource intensive process that is expected to be executed only when users need to verify the integrity of their database. It can be executed hourly or daily, for cases where the integrity of the database needs to be continuously monitored, or only when the organization hosting the data goes through an audit and needs to provide cryptographic evidence regarding the authenticity of their data. To reduce the cost of verification, our system exposes options to verify individual Ledger tables or only a subset of the ledger.

2.4 Digest Management

The verification process and, therefore, the integrity of the database depends on the integrity of the input digests. For this purpose, Database Digests that are extracted from the database need to be stored in a trusted location that cannot be tampered with by the high privileged users or attackers of the RDBMS. SQL Ledger integrates with Azure Immutable Blob Storage [19] to periodically (every few seconds) upload Database Digests in immutable, append-only BLOBs that do not allow future modifications by any users or even Microsoft engineers. This

provides a simple and cost-effective way for users to automate digest management without having to worry about their availability and geographic replication. Additionally, our system allows users to generate a Database Digest on demand so that they can manually store it in any service or device that they consider trusted. Using this mechanism, users with stricter security requirements can store digests outside of the Microsoft cloud, removing Microsoft from the trust boundary. For example, Database Digests can be stored in an on-premise WORM device, uploaded to a Public Blockchain, such as Bitcoin or Ethereum [11], or even signed with the company's private/public key pair, to guarantee their authenticity, and shared with any customers, partners or auditors who can later use them to verify the corresponding data.

2.5 Integrity Guarantees and Threat Model

2.5.1 Integrity Guarantees

As discussed in Section 1, SQL Ledger offers a less well-known notion of integrity called Forward Integrity. The idea is that the system is trusted until the time when a transaction is processed, and data integrity is protected against *future* compromises of the system.

Let us use a real-world scenario to explain how Forward Integrity, although weaker than full integrity, can address a wide range of scenarios: Contoso is a car manufacturer that uses a database to track the parts manufactured and their lifecycle. Bob bought a car made by Contoso in 2018. In 2020, Bob has a collision and he soon finds out that Contoso had issued a recall for certain batches of brake parts that they manufactured. He, then, files a lawsuit against Contoso claiming that defective parts were used on his car and are responsible for the collision, but he was never notified as part of the recall process. Contoso now needs to prove that the parts used on Bob's car were not part of the defective batches that were recalled. Unless Contoso is using a technology to protect data integrity, one of their DBAs or system administrators can easily tamper with the part related data and provide false information, therefore making such evidence unreliable. In this scenario, Contoso is generally an honest company that tracks their parts lifecycle correctly since they do not have any motive to do otherwise. Therefore, all parts used in Bob's car are originally tracked correctly in their database. When the defective parts are identified later or, more importantly, when the lawsuit happens, Contoso or some of their employees might be motivated to tamper with the data to their advantage. Forward Integrity protects against such attacks where an adversary (internal or external) attempts to tamper with the data after it has been originally written in the system. This is a very common pattern in real-world scenarios as enterprises generally act honestly until a specific event might put their finances or reputation at risk, motivating them to act maliciously.

SQL Ledger guarantees Forward Integrity by leveraging cryptographically tamper-evident data structures under the following assumptions:

1) The user operation that originally updated the data in the Ledger table was processed correctly and remained valid until a Database Digest was generated (which occurs within a few seconds). This assumption is aligned with the definition of Forward Integrity which intends to protect data from future tampering.
2) The Database Digests used for verification are stored in a separate, trusted location that cannot be tampered with by the adversary.
3) The ledger verification process can be executed in a trusted environment. This can be achieved in the original environment itself if the users trust it, or by restoring the database in a different trusted environment.

2.5.2 Threat Model

We aim to protect data integrity against strong adversaries who have total control of the database, the operating system (OS) and the machine where the RDBMS is running. A strong adversary has full power over the system and can mount any attack, such as a) modify the data stored in the database using the supported database APIs, b) modify the data bypassing the database layer and directly updating it in storage or c) manipulate the execution of the RDBMS and the OS (for example by attaching a debugger) for their advantage.

3 DETAILED DESIGN

In this section, we describe the detailed design of the various components of SQL Ledger and how they interact with each other. We focus on updateable Ledger tables since they represent the more generic case, supporting all DML operations.

3.1 Ledger Table Schema Extensions

The schema of Ledger and History tables is automatically extended to include four additional columns that are internally populated by the system and track:

- The ID of the transaction that generated the row version.
- The sequence number of the operation that generated the row version.
- The ID of the transaction that deleted the row version.
- The sequence number of the operation that deleted the row version.

These columns are hidden from the users to guarantee transparency for the application using the table but are still exposed through the Ledger view. The transaction information is used to correlate the row versions modified by a transaction to the corresponding transaction entry in the Database Ledger. This allows users to retrieve metadata regarding the transactions that performed these operations but, more importantly, it is used by the verification process to identify all modifications performed by each transaction. The operation sequence numbers store the order within a transaction in which the row versions were updated. This is necessary because the row versions are hashed in the order in which they are updated and the verification process must use the same order when re-computing the hash for verification purposes.

3.2 DML Operations and Row Hashing

Any operations that update a Ledger table need to perform some additional tasks to maintain the historical data and compute the digests captured in the Database Ledger. Specifically, for every row updated, we must:

- Persist the earlier version of the row in the history table.
- Assign the transaction ID and generate a new sequence number, persisting them in the appropriate system columns.
- Serialize the row content and include it when computing the hash for all rows updated by this transaction.

SQL Ledger achieves that by extending the DML query plans of all insert, update and delete operations targeting Ledger tables. As part of inserting or updating a row, the transaction ID and newly generated sequence number are set for the new version of the row. Then, the query plan operator executes a special expression that serializes the row content and computes its hash, appending it to a Merkle Tree that is stored at the transaction level and contains the hashes of all row versions updated by this transaction for this Ledger table. The root of the tree represents all the updates performed by this transaction in this Ledger table. If the transaction updates multiple tables, a separate Merkle Tree is maintained for each table.

If the operation also deletes a row version, by doing a delete or update, the deleted row version is propagated to an additional operator that also assigns the transaction ID and new sequence number for the deleted version and inserts it into the corresponding History table. The deleted row version is also hashed and appended to the same Merkle Tree. When the transaction commits, it will compute the root for each Merkle Tree and store it as part of the transaction entry that is recorded in the Database Ledger in tuples of the form (ledger_table_id, merkle_root_hash).

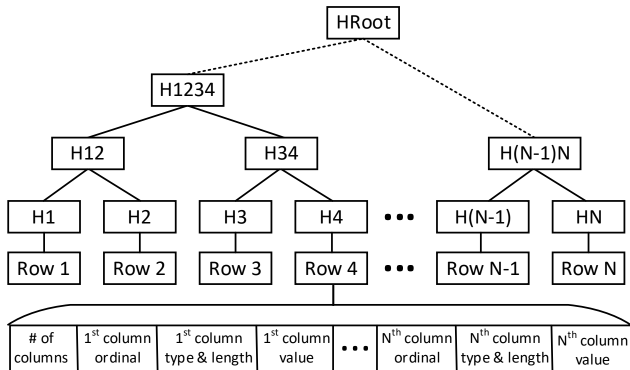

Figure 4. Sample Merkle Tree of updated rows.

Figure 4 shows an example of a Merkle Tree storing the updated row versions of a ledger table and the format used to serialize the rows. Other than the serialized value of each column, we include metadata regarding the number of columns in the row, the ordinal of individual columns, the data types, lengths and other information that affects how the values are interpreted. This is important because an attacker can tamper with the metadata of a Ledger table and cause the data to be interpreted incorrectly without ever tampering with the data itself. To better understand this, let us take a simple example of a table with two columns: "Column1" of type INT (4 bytes) and "Column2" of type SMALLINT (2 bytes). Now, let us assume that we have a row where Column1 = 0x12 and Column2 = 0x34. If our format only serialized the column values, the serialized form of this row would be: 0x000000120034. If the attacker tampers with the metadata and declares Columns1 as SMALLINT and Column2 as INT, the overall serialized value would not change but the data would now be interpreted as Column1 = 0 and Column2 = 0x120034, leading to incorrect results when querying the Ledger table. By including the metadata of the columns, we guarantee that the verification process will also detect attacks that affect the data interpretation.

3.2.1 Merkle Tree Algorithm

By appending the updated row versions in a Merkle Tree, we can represent all the versions updated by a transaction with the root of the Merkle Tree. This is very efficient since it reduces a potentially huge volume of data down to a single SHA-256 hash that can be stored in the transaction metadata. The challenge, however, is that the number of versions updated can be very large and is not known upfront. Additionally, it would be inefficient to re-process them after the operation has completed. For that purpose, we developed a streaming algorithm that computes the root of a Merkle Tree while the row versions get updated. Our algorithm works as follows:

Let $N_{L,i}$ be the ith node of level L. For each level L, we store the last node that was appended to this level: $N_{L,i}$, where i is the last node of the level. When a new node $N_{l,i+1}$ is appended to the level:

- If i+1 is odd, we simply store this node as the last node of this level.
- If i+1 is even, we compute the hash of the new node ($N_{L,i+1}$), combined with the previous node we have stored for the level ($N_{L,i}$), per the Merkle Tree definition, and append the resulting hash as a new node for the parent level (L+1).

This process is executed recursively as long as a new node needs to be appended to the parent level. When all leaf nodes have been appended to the Merkle Tree, if the last node does not have a sibling, we promote the node itself as its parent. This is also executed recursively until we reach the root of the tree.

The time complexity of this algorithm is O(N) and the space complexity O(logN), where N is the number of leaf nodes of the tree, allowing us to efficiently compute the root of the Merkle Tree as the row versions get updated. The small space required to maintain the intermediate state of the tree is also critical to enable partial transaction rollbacks [22] that are supported by most RDBMSs. Specifically, when a savepoint is created in the transaction, the current state of the Merkle Tree is copied and maintained as part of the savepoint information. As more operations occur, the transaction Merkle Tree gets updated. However, if the transaction rolls back to this savepoint, the earlier copy is used to bring the tree back to the state it had when the savepoint was created. The logarithmic space needed for

recording the Merkle Tree state allows us to support a large number of savepoint with minimal memory footprint and overhead.

3.3 Database Ledger Design

3.3.1 Data Structure

As described in Section 2.2, the Database Ledger incrementally captures the state of the database as it evolves over time while updates occur on Ledger tables. To achieve that, the Database Ledger stores an entry for every transaction capturing metadata about the transaction, such as its commit timestamp and the identity of the user that executed it, but also the Merkle Tree root of the rows updated in each Ledger table. These entries are then appended to a tamper-evident data structure to allow verifying their integrity in the future. The data structure used for the Database Ledger needs to meet the following requirements:

1) High append throughput – Since the Database Ledger records every transaction in the system, the data structure needs to be able to keep up with the high throughput rate of a commercial RDBMS which can exceed 100K transactions per second.
2) Low storage overhead – The number of transactions accumulated over time will be significant and the data structure must not incur significant overhead over the space required for storing the transaction information.
3) Efficient external verification of the integrity of the data structure. As described in the previous section, the Database Digest captures the state of the database at different points in time. It is important that our data structure allows users to verify that a digest generated at time t_1 can be derived from an earlier digest generated at time t_0 where $t_0 < t_1$. This allows users to confirm that the new digest accurately represents the state captured in the earlier digest so that they can only maintain the latest one. If this verification fails, it indicates that earlier data has been tampered with and the new digest represents a "forked" state that should not be accepted. This enables early detection of a critical class of attacks that would lead to generating invalid digests until the verification process is executed.
4) Ability to externally verify that a transaction T is included in the ledger in an efficient manner. This allows the system to return a proof that a transaction is part of the ledger and support non-repudiation even if the ledger is tampered with or destroyed. Section 5.1 discusses this process in more detail.

It is also important to state that requirements 3 and 4 should be supported without compromising the confidentiality of the transactions in the ledger since the goal is to allow external verification from users who might not have access to individual transaction information.

Figure 5 shows the design of the Database Ledger data structure. To satisfy the first and second requirements, we use a Blockchain data structure with a large block size (100K transactions per block) so that the cost of computing the block-level hashes and storing the block information is amortized over a large number of transactions. The large block size also helps us meet the third requirement since external verification would happen at the block level where the number of blocks will be orders of magnitude lower than the number of transactions in the system. However, the large block size poses difficulties around our fourth requirement since users would need to retrieve information for all other transactions in the block to successfully verify that a transaction of interest is contained there. To address that, instead of storing the transactions directly in the block, we create a Merkle Tree over the transactions in the block and store its root as part of the block information. This technique allows us to leverage Merkle Proofs to efficiently prove that a specific transaction is contained within a block. Users can simply verify the Merkle Proof for the transaction to confirm that the transaction was indeed inserted in the ledger. Finally, since the block only stores the root hash of the transactions Merkle Tree and Merkle Proofs only contain hashes, the third and fourth requirement can be satisfied without leaking any information about the transactions in the ledger.

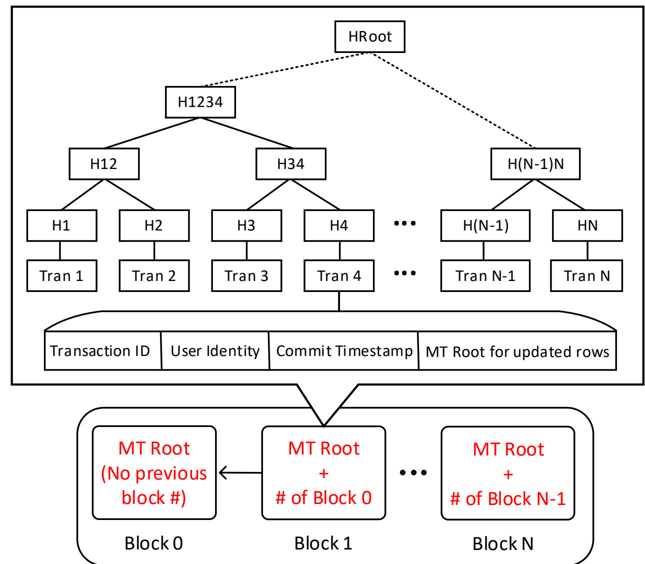

Figure 5. The Database Ledger data structure.

Although the Database Ledger logically leverages the Blockchain and Merkle Tree data structures, the actual information regarding transactions and blocks is physically stored as rows in two new system tables: "database ledger transactions" and "database ledger blocks" respectively. The former maintains a row with the information of each transaction in the ledger, including the ID of the block where this transaction belongs and the ordinal of the transaction within the block. The latter maintains a row for every block in the ledger, including the root of the Merkle Tree over the transactions within the block, as well as the hash of the previous block to form a Blockchain.

3.3.2 Atomicity and Durability

SQL Ledger needs to guarantee that a transaction and any data modified by it are captured in the ledger if and only if the transaction is successfully committed. The ledger data must also be recoverable in case of any system failures. Since the data stored in Ledger tables and their corresponding History tables is transactionally updated, similar to other user data, SQL Server's

transactional system automatically satisfies these requirements. However, the transaction entries stored in the Database Ledger cannot be directly inserted in a database table since this would be expensive and cause contention when appending to the last slot of the ledger structure, limiting the system throughput. SQL Ledger integrates the process of appending transactions to the Database Ledger with SQL Server's recovery subsystem to achieve high throughput, atomicity and durability.

SQL Server follows the ARIES [22] recovery model which depends on Write-Ahead-Logging (WAL) and Checkpointing to provide atomicity and durability. Specifically, when a transaction commits, it generates a COMMIT log record which, once written to the log, guarantees that the transaction is durably persisted in the system. SQL Ledger extends the transaction commit process to a) generate the ledger transaction entry, b) assign it to the latest block of the ledger data structure and c) append it to the Database Ledger in-memory queue, as part of generating the COMMIT log record. These steps only involve modifying in-memory state and require minimal synchronization, therefore not incurring any noticeable overhead to the commit process which is in the hot path of transaction processing. Additionally, the COMMIT log record tracks the block ID and ordinal of the transaction within the block to make this information recoverable. When a checkpoint occurs, any transactions accumulated in the in-memory queue are batched and inserted into the "database ledger transactions" system table that stores the transaction entries. In the event of a failure, the Analysis phase of recovery will process the COMMIT log records since the last successful checkpoint and reconstruct the state of the in-memory queue for any transactions that were not written to the system table before the failure. This process guarantees the recoverability of the transaction information in the Database Ledger.

When a block becomes full, it is marked as "closed" so that new transactions will be inserted in a new block. The block generation process then retrieves all transactions that belong to the "closed" block from both the in-memory queue and the "database ledger transactions" system table, computes the Merkle Tree root over these transactions and the hash of the previous block and persists the closed block in the "database ledger blocks" system table. Since this is a regular table update, its durability is automatically guaranteed by the system. To maintain the single chain of blocks, this operation is single-threaded, but it is also very efficient, as it only computes the hashes over the transaction information, and happens asynchronously, thus, not impacting the transaction performance.

3.4 Ledger Verification

3.4.1 Ledger Verification Invariants

The ledger verification process is responsible for verifying that the latest and the historical data of Ledger tables is consistent with the Database Digests that are provided as input. This allows detecting any tampering that might have occurred and guaranteeing Forward Integrity.

Although the data of Ledger tables is logically included in the Blockchain data structure used by the Database Ledger, the various elements of the ledger (latest and historical data, transactions and blocks) are physically stored in separate relational tables and linked together through their IDs and hashes. Additionally, the Ledger and History tables can have non-clustered indexes that duplicate the base table data and can be tampered with independently. Based on that, the ledger verification process involves multiple steps that verify the following invariants for the individual elements of the ledger:

Given a set of input Database Digests:

1) For every digest, the hash captured in the digest for the ith block of the Database Ledger Blockchain should be equal to the hash computed over the current state of the ith block.
2) For every block in the Blockchain, the hash recorded for the previous block should be equal to the hash computed over the current state of the previous block. The only exception is block 0 where the hash recorded for the previous block should be null.
3) For every block in the Blockchain, the transactions Merkle Tree root recorded in the block should be equal to the Merkle Tree root computed over the transactions that belong to this block in the current state of the database. Additionally, all transactions should belong to a block that is part of the Blockchain.
4) For every transaction in the system and Ledger table updated by this transaction (as recorded in the transaction information), the Merkle Tree root recorded in the transaction information should be equal to the Merkle tree root computed over the current state of the table for all rows that were updated by this transaction. Additionally, no Ledger or History table rows should reference transactions that are not recorded in the system.
5) The data of every non-clustered index of a Ledger or History table should be equivalent to the base table data.

The verification process can provide cryptographic guarantees for the data up to the highest block captured by the input digests. Data contained in later blocks can be verified to be consistent within the database, but, since it is not covered by a digest, it is still susceptible to advanced attacks that can overwrite the data and recompute the transaction and block hashes to make them consistent. This is the reason why our system has been optimized to allow for frequent digest generation so that it can cryptographically protect even the most recent data.

3.4.2 Ledger Verification Implementation

Since the various elements of the Database Ledger are stored in a relational form, in the Ledger and History tables, as well as the system tables that store the transaction and block information, SQL Ledger leverages SQL Server's query processing engine to execute the verification tasks through queries. This allows us to take advantage of the optimized query processing operators for performing Joins and Aggregations and leverage parallel query execution to minimize the verification time over the potentially large volume of data. To achieve that, we expose the row serialization, hashing and Merkle Tree computation logic, that is used to maintain the ledger during transaction processing, as intrinsic and aggregate functions and then generate queries that use these functions to verify each of the invariants we defined. Specifically, the serialization and hashing logic is exposed as an

intrinsic function, called LEDGERHASH, while the Merkle Tree computation logic as an aggregate function, called MERKLETREEAGG, that computes the Merkle Tree root over a set of rows.

The verification queries map one to one to the ledger invariants and are the following:

1) Using the OPENJSON function, the JSON array of input digests is converted into a relation. We then perform a LEFT JOIN with the "database ledger blocks" system table based on the Block ID. The query calls the LEDGERHASH function to compute the hash of each block from the system table and checks for cases where a) the hash of the digest does not match the computed hash for the block or b) there are digests representing a block that is not present in the ledger.
2) Using the LAG function of SQL Server that allows accessing the previous row of a dataset, we scan the "database ledger blocks" system table ordered by the Block ID. This enables us to access each block together with its previous block. We then use the LEDGERHASH function to compute the hash of the previous block and check whether the computed hash matches what is recorded in the current block as is expected for our Blockchain data structure.
3) Using the LEDGERHASH and MERKLETREEAGG functions, we scan the "database ledger transactions" system table, compute the hash of every transaction and then the Merkle Tree root for all transactions belonging to each block (GROUP BY the Block ID), ordered by their ordinal within the block. We then perform an OUTER JOIN of this dataset with the "database ledger blocks" system table on the Block ID to identify cases where a) the computed Merkle Tree root for the transactions of a block does not match what is recorded in the block or b) there are transactions that belong to a block that is not present in the system.
4) Similar to the previous query, we use the LEDGERHASH and MERKLETREEAGG functions to compute the hash of each row version in a Ledger Table (and the corresponding History table) and then compute the Merkle Tree root over these rows for each transaction (GROUP BY the Transaction ID), ordered by the sequence number of each row. We then perform an OUTER JOIN of this dataset with the "database ledger transactions" system table on the Transaction ID to identify cases where a) the computed Merkle Tree root for the row versions updated by this transaction does not match what is recorded in the transaction entry or b) there are rows in the Ledger table that belong to a transaction that is not recorded in the system. This process is then repeated for every Ledger table in the database.
5) For each Ledger table and History table, we use the LEDGERHASH and MERKLETREEAGG functions to compute the Merkle Tree root of all rows in the table after ordering them based on the clustered index key or Row Identifier (in the case of Heaps). We then apply the same logic to compute the Merkle Root of all rows in each of their non-clustered indices and check for cases where the hash computed over the base table does not match what is computed over the non-clustered index.

Once all data has been verified using these queries, the verification process confirms that the Ledger view definition for each Ledger table is valid. This is necessary since the Ledger view is also a database artifact that could have been tampered with, leading to incorrect results when users attempt to query the ledger to detect malicious row modifications.

3.5 Schema Changes

Since the data stored in Ledger tables is designed to be immutable, schema changes present certain challenges since they fundamentally affect the data stored in the table. We distinguish schema changes in two categories: physical and logical. Physical schema changes refer to operations that affect the physical design of the database, such as adding or dropping indexes, primary or foreign keys. Logical schema changes correspond to operations that logically affect the data stored in a table, such as adding, dropping or altering a column or a table.

Since the hashes captured in the ledger are computed over the logical data stored in Ledger tables and are not impacted by the indexes and keys defined, physical schema changes can easily be supported in our system. However, logical schema changes can impact the underlying data of the table and, therefore, require special handling to guarantee that they do not affect the data that has been recorded in the ledger. This section describes how SQL Ledger handles some common logical schema changes.

3.5.1 Adding Columns

Adding a nullable column is probably the most common schema change as applications evolve over time and need to store additional data in their tables. SQL Ledger is designed to handle this operation by ignoring NULL values when computing the hash of a row version. Based on that, when a nullable column is added, SQL Ledger will modify the schema of the Ledger and History tables to include the new column, however, this does not impact the hashes of existing rows. When the verification process re-computes these hashes, it will ignore the NULL values for the new column and compute a hash that matches what was originally recorded in the ledger.

Although this technique helps us support adding new columns, it opens a window for an attacker to tamper with the row information that defines which of the columns contain a NULL value. This would allow them to modify the way the column values are interpreted and return a NULL value for a different column instead. This attack is prevented by including the column ordinal for all non-NULL columns in our serialization format, described in Section 3.2, to explicitly define which columns contain non-NULL values.

3.5.2 Dropping Columns and Tables

Dropping a column or a table is also frequently used as old data becomes irrelevant or while developers are experimenting with the schema of their applications. Normally, dropping a

column/table completely erases the underlying data from the database and is, therefore, fundamentally incompatible with the ledger functionality that requires data to be immutable. Instead of deleting the data, SQL Ledger simply renames the objects being dropped so that they are logically removed from the user schema but physically remain in the database. Any dropped columns are also hidden from the Ledger table schema, so that they are invisible to the user application. However, the data of such dropped objects remains available for the ledger verification process, which can still access it based on their object IDs, and allows users to inspect any historical data through the corresponding Ledger views.

Despite its usefulness, allowing users to drop an object enables a class of attacks that could violate the Forward Integrity guarantees our system provides. An attacker can drop an existing table and create a new one with the same name but data that has been tampered with. The verification process will verify both tables (based on their IDs) but when the users query the table, they might not realize it is now referring to a different object that could have been introduced more recently. To mitigate this risk, SQL Ledger stores the metadata of all Ledger tables and columns in two updateable Ledger system tables, for tables and columns respectively, and exposes all operations that have modified this metadata through the corresponding Ledger views. The integrity of these tables is verifiable through the regular verification process and users can query the views to identify when their tables/columns were created/dropped to decide whether this was intentional or a potential attack. Figure 6 provides an example of the Ledger view tracking the operations that created or dropped various Ledger tables.

Table Name	Table ID	Operation	Transaction ID
Customers	1	CREATE	100
Orders	2	CREATE	150
LineItems	3	CREATE	160
MS_DroppedTable_Customers	1	DROP	190
Customers	4	CREATE	190

Figure 6. Ledger system view tracking table operations.

3.5.3 Altering Column Properties

SQL Server allows any properties of a column to be altered: data type, length, nullability, collation, etc. Any changes that do not impact the underlying data of a Ledger table, such as changing nullability, collation for Unicode strings or the length of variable length columns, are supported without any special handling as they do not impact the hashes being captured in the ledger. However, any operations that might affect the format of existing data, such as changing the data type, are handled by dropping the existing column, adding it back with the original name and, finally, re-populating it with the original data, including any conversion that is required for the type change. The logic to drop and add the column follows the semantics we presented in the previous sections.

3.6 Integration with Azure Immutable Blob Storage

As described in Section 2.4, SQL Ledger automates digest management by integrating with Azure Immutable Blob Storage and periodically uploading digests to append-only, immutable BLOBs that cannot be modified by users or even Microsoft engineers. During verification, these digests are automatically downloaded and used to verify the integrity of the database.

In the common case, this process is straightforward and simply requires accessing a user provided storage account to upload or download the JSON documents storing the digest information. However, Azure SQL Database supports certain operations that allow bringing the database state back to an earlier point in time. Although these operations move the database state back in time, which is normally considered an attack for SQL Ledger, it is important to support them since they are necessary for meeting the operational requirements of enterprise users. Specifically, our digest management solution needs to address the following scenarios:

- Failover across geographic regions.
  Replication across geographic regions is asynchronous for performance reasons and, thus, allows the secondary database to be slightly behind compared to the primary. In the event of a geographic failover, any latest data that has not yet been replicated is lost.
- Restoring the database back to an earlier point in time, also known as Point in Time Restore [20].
  This is an operation frequently used when a mistake occurs and users need to quickly revert the state of the database back to an earlier point in time.

In the case of geographic failovers, the replication delay is bounded and normally remains below one second. Based on that, SQL Ledger will only issue Database Digests for data that has been replicated to geographic secondaries to guarantee that digests will never reference data that might be lost in case of a geographic failover. This slightly increases the window of vulnerability but, given that the introduced delay is normally below 1 second and digests are generated every few seconds, the difference should be negligible. If replication starts falling further behind, significantly delaying the digest generation, SQL Ledger will trigger an alert and eventually stop accepting new requests until the secondaries are caught up and digests can get successfully generated.

In the case of restore, the database can be restored to any arbitrary point in time and, therefore, deferring digest generation is not an option. Instead, when uploading the generated digests to Azure Storage, we will capture the “create time” of the database that these digests map to. Every time the database is restored, it is tagged with a new create time and this technique allows us to store the digests across different “incarnations” of the database. SQL Ledger preserves the information regarding when a restore operation occurred, allowing the verification process to use all the relevant digests across the various incarnations of the database. Additionally, users can inspect all digests for different create times to identify when the database was restored and how far back

it was restored to. Since this data is written in immutable storage, this information will be protected as well.

3.7 Recovery from Tampering

SQL Ledger cryptographically guarantees Forward Integrity by leveraging tamper-evident data structures and verifying their integrity through an asynchronous verification process. Although this should deter any attackers from attempting to tamper with the data, which is not even possible through the official system APIs, a determined adversary can still do so by compromising the process or modifying the data directly in storage. Such attacks cannot be prevented in our system but will be detected when the verification process is executed. If the verification fails, bringing the database back to a consistent state can be challenging and depends on the type of data and how soon the attack gets detected. SQL Ledger does not currently automate this process, however, in this section, we discuss how users can manually achieve that.
For this discussion, we make the following assumptions:

- Earlier backups of the database are available and have not been tampered with. These backups can be restored and verified to be consistent through the ledger verification process. This allows users to recover to a consistent state before the tampering occurred.
- Any attack that attempts to "fork" the ledger Blockchain, by overwriting earlier blocks, is detected when a digest is generated, following the external verification process, described in Section 3.3.1, that confirms that each new digest can be derived from the previous one. This guarantees that all digests are correctly generated over a Blockchain that was never forked.

We then separate the data stored in Ledger tables in two categories:

1) Data that does not affect how future transactions are processed, such as the transaction details tracked for each financial transaction.
2) Data that is used for further transaction processing, such as the current account balances that define how much money a customer can withdraw.

When data of the first category is tampered with, this event does not impact the execution of future transactions which are correctly written to the ledger, computing the appropriate hashes and generating valid digests. In this case, users can simply restore the latest database backup that can be successfully verified and repair the data of the original database that has been maliciously modified (as reported by the verification process). All generated digests will remain valid since the chain was never forked and the verification process can succeed, proving the database integrity.
When data of the second category is tampered with, future transactions that use this data for their execution might have also been compromised. The outcome of these transactions can be invalid, resulting to an incorrect ledger and generated digests. In this case, users need to restore the latest backup that can be successfully verified and then use that as the basis to re-execute the transactions that occurred after this point in time. This process can be challenging and, thus, it is critical for the verification process to be executed frequently to minimize the number of transactions that need to be reprocessed. Additionally, any previously generated digests for this period of time need to be invalidated and any parties that have been using them, such as partners or auditors, should be made aware of this fork in the ledger.

4 PERFORMANCE EVALUATION

This section presents experimental results regarding the performance of SQL Ledger. All our experiments are executed on a workstation with 4 sockets, 72 cores (Intel® Xeon® Processor E7-8890, 2.50GHz) and 1TB of RAM. External storage consists of two 1TB SSDs for data and log respectively.

4.1 User Workload Performance

In our first set of experiments, we evaluate the performance of SQL Ledger when executing user transactions. SQL Ledger takes advantage of the optimized transaction processing engine of SQL Server but it must also preserve historical data in the History tables and compute the SHA-256 hashes of any modified rows. Based on that, we measure the overhead introduced by this additional logic and how it impacts the throughput and latency of the system.

4.1.1 Throughout

Since the overhead introduced by SQL Ledger is mainly around row modifications, we evaluate the throughput of the system using update intensive OLTP workloads. Specifically, we experiment with a TPC-C-like workload that is extremely update intensive and should be a worst-case scenario for SQL Ledger, and a TPC-E-like workload that represents a more common ratio between reads and writes.
The TPC-C workload simulates an order processing system of a wholesale supplier that receives and fulfils orders from customers. In this setting, the supplier would want to establish trust with their customers by protecting the integrity of the data that tracks the order details and, especially, the payment and shipping related information. Based on that, we converted four, out of the nine in total, TPC-C tables that store order related information to Ledger tables. These tables are updated multiple times for every order placed to track the order status.
The TPC-E workload simulates the activity of a stock brokerage firm that allows their clients to view their account details and submit stock orders, but also generates reports for the brokerage firm. Given the financial nature of the scenario, most tables contain information that must be protected from tampering since it relates to customer positions and stock orders. Based on that, we converted all 33 tables into Ledger tables.
Figure 7 presents the difference in throughput between SQL Ledger and traditional SQL Server for these workloads. Considering the strong security guarantees that SQL Ledger provides and the fact that the vast-majority of applications are not as update intensive as TPC-C, we believe that the performance degradation should be acceptable for any applications that require protecting the integrity of their data. As we anticipated, the overhead introduced by SQL Ledger becomes more noticeable in

the case of TPC-C due to the high frequency of updates. According to the profile data we collected, inserting the historical data into the History table accounts for approximately half of the overhead while the hash generation is responsible for the remainder.

Workload	Performance difference
TPC-C	-30.6%
TPC-E	-6.9%

Figure 7. Throughput of SQL Ledger compared to traditional SQL Server.

Despite the performance degradation compared to traditional SQL Server, SQL Ledger was able to scale to the 72 physical cores of the workstation without running into any bottlenecks and achieve a throughput above 70K transactions per second (tps) for TPC-C-like transactions. This is more than 20 times higher than what state of the art Blockchain systems, like Hyperledger Fabric, can achieve, even when evaluated using simpler transactions [1].

4.1.2 DML Latency

In this set of experiments, we measure the latency of different types of DML operations on Ledger and regular SQL Server tables. Figure 8 demonstrates the latency of single row operations on a table that has 260-byte wide rows and a varying number of indices.

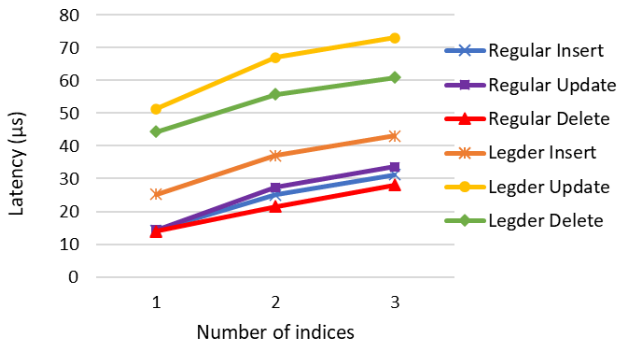

Figure 8. DML latency for different types of operations and number of indices on regular and Ledger tables.

The overhead introduced by SQL Ledger is smallest in the case of Insert operations since these only compute the SHA-256 hash of the inserted rows but do not maintain any historical data. This accounts for ~12µs/row. In the case of Delete operations, SQL Ledger inserts the deleted rows in the History table and computes their hash. The History table insertion accounts for an additional ~18µs/row. Finally, Update operations need to compute the hash of each updated row both before and after the update but also insert the earlier versions in the History table. Based on that, we expect an overhead of approximately (2 * 12 + 18) = 42µs/row which is aligned with the results we see in the diagram (~40µs/row).

Although the latency overhead is noticeable, our experiments only measure the cost to locate and update a row and exclude the cost of committing the transaction that would be dominant and add approximately 125µs. Based on that, the latency of short transactions that update a small number of rows before committing is not significantly impacted. Additionally, the observed latency is orders of magnitude lower than state of the art Blockchain systems whose latency is in the order of 100s of ms [1] due to the decentralized consensus protocols they depend on.

4.2 Ledger Verification Performance

The Ledger verification process is a resource intensive operation that must verify the integrity of the Database Ledger and then scan all rows of Ledger and History tables to recompute their SHA-256 hashes and compare them with the corresponding digests stored in the Database Ledger. In this section, we evaluate the performance of the verification process for different numbers of transactions. In our experiments, each transaction updates five rows in a Ledger table. Every row is 260 bytes wide. Figure 9 presents the verification times for different numbers of transactions.

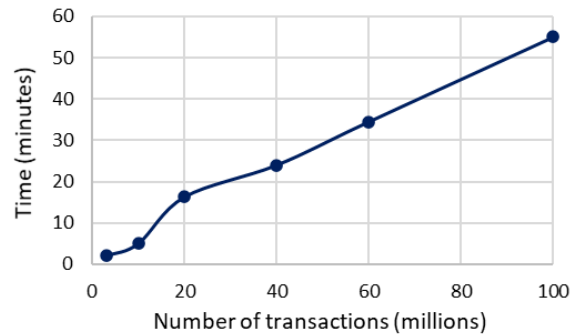

Figure 9. Ledger verification times for different numbers of transactions.

As anticipated, the verification time is proportional to the number of transactions and row versions that must be processed. Although the verification time becomes high when the volume of data gets large, it still allows the process to be executed daily, if necessary, even for databases that store TBs of data. Finally, despite its cost, the verification process can be executed in the background without impacting the user workload or even offloaded to a separate node, such as a secondary replica or a database copy [21] to avoid consuming resources from the production instance.

5 WORK IN PROGESS

In this section, we discuss our ongoing work on future enhancements of SQL Ledger. The additional functionality intends to improve the security guarantees of the system by supporting non-repudiation and allow users to manage the size of their database by truncating old ledger data that is no longer needed.

5.1 Non-repudiation

SQL Ledger already supports a limited form of non-repudiation as the organization hosting the data cannot dispute the outcome of

previously executed transactions without tampering with the ledger, which is detectable. However, if the ledger is tampered with or destroyed, there is no way for a user to prove that a specific transaction, such as a large money deposit, occurred. A naïve approach to address that would be to cryptographically sign every transaction with a public/private key pair and return this signature to the user executing the transaction. This is not a viable option, though, due to the high cost of computing asymmetric signatures. Instead, by leveraging the Merkle Tree structure of the Database Ledger, SQL Ledger can generate a "receipt" for each transaction, containing a) the Merkle Proof for the transaction, which proves that it is part of the block, and b) the signature of the tree root for the corresponding block. This allows us to generate a receipt for each of the 100K transactions in a block with only one signing operation.

5.2 Ledger Truncation

Since SQL Ledger maintains all historical data and transaction information, the database size will grow over time. Although Azure SQL Database supports scaling database storage to 10s of TBs [2], users will still prefer to delete unnecessary data to reduce cost. Additionally, compliance regulations require storing historical data for a bounded amount of time and not perpetually. Based on that, our goal is to support truncating old ledger and historical data that is no longer needed. Historical data is easy to truncate because no other data elements reference it. Old Database Ledger blocks and transactions, however, can still maintain digests representing the current data of a Ledger table, if this has not been recently updated. To allow deleting old transaction and blocks, the truncation operation will first trigger the verification process to guarantee that any current data is consistent and then perform a dummy update on any Ledger table data that references a transaction that is going to be truncated. This process effectively moves the digests of this data into new transactions and blocks so that old ones can be safely truncated. Finally, a record indicating that a ledger truncation occurred will be recorded into the ledger to guarantee that this operation is audited.

6 RELATED WORK

In the research community, there has been a rich body of work [3, 5, 6, 10, 13, 15, 23, 26, 27, 29, 30] on technologies that support verifying the integrity of data hosted in untrusted environments. These systems can be categorized based: a) whether they depend on Trusted Hardware, such as Intel SGX enclaves [17], b) whether they support updates or only queries and c) whether they verify the completeness and correctness of query results. Despite their technical merit, these solutions did not get enough traction in the industry to become part of commercial data management solutions.

With the launch of Bitcoin, and largely due to its financial success, we see renewed interest in the space of data integrity and a wide range of Blockchain and Distributed Ledger Technology (DLT) systems [1, 8, 9, 11, 12, 24] being released over the last years. Although these technologies can also be used to protect the integrity of centralized data, their decentralized design makes them complex to manage and severely impacts their performance, which is orders of magnitude lower compared to production RDBMSs. Amazon QLDB [4] and Oracle Blockchain tables [25] attempt to bridge the gap between distributed ledgers and databases, by offering a centralized solution that leverages the cryptographic data structures of Blockchains. Similar to SQL Ledger, QLDB stores data in a Blockchain and allows extracting digests to verify its integrity. However, verification occurs at the document level and does not provide a mechanism to verify the whole dataset that would be necessary to guarantee query correctness. Additionally, QLDB is a document store and does not provide the rich capabilities of an RDBMS. Oracle Blockchain tables, on the other hand, are fully integrated in the Oracle database, but only support insertions and, more importantly, do not expose digests outside of the database. Based on that, users must trust the RDBMS which is what these solutions intend to avoid in the first place. Finally, systems like BigchainDB [16] and ChainifyDB [28] maintain the decentralized architecture of DLTs, that enables multi-party computation, but integrate with database systems to leverage the rich data model and high performance of relational databases.

7 ACKNOWLEDGMENTS

We would like to thank all team members for their contributions to the project. Delivering SQL Ledger would not have been possible without their commitment and hard work. We would also like to thank the Redmond and Cambridge Microsoft Research teams and, especially, Arvind Arasu, Miguel Castro, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cedric Fournet, Donald Kossmann and Ravi Ramamurthy for their collaboration and feedback on our design. Finally, we would like to thank our leadership team for sponsoring a project in the upcoming space of data integrity and continuing to invest in our work in this area.

8 REFERENCES

[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, page 30. ACM, 2018.

[2] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. Farooq Minhas, N. Prakash, V. Purohit, H. Qu, C. Sreenivas Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. 2019. Socrates: The New SQL Server in the Cloud. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 1743–1756. DOI:https://doi.org/10.1145/3299869.3314047

[3] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17). Association for Computing Machinery, New York, NY, USA, 251–266.

[4] AWS. Amazon Quantum Ledger Database. https://aws.amazon.com/qldb.

[5] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. IEEE Trans. Knowl. Data Eng., 26(3):752–765, 2014.

[6] A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with Haven. In OSDI, pages 267–283, 2014.

[7] M. Bellare and B. Yee. Forward Integrity for Secure Audit Logs. Tech. rep. Computer Science and Engineering Department, University of California at San Diego, 1997.

[8] ConsenSys. Quorum. https://consensys.net/quorum/
[9] Corda. https://www.corda.net.
[10] P. T. Devanbu, M. Gertz, C. U. Martel, and S. G. Stubblebine. Authentic data publication over the internet. Journal of Computer Security, 11(3):291–314, 2003.
[11] Ethereum. https://www.ethereum.org.
[12] Everledger. https://www.everledger.io
[13] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In ICDE, pages 529–540, 2013.
[14] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems, 4(3), 1982.
[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In SIGMOD, pages 121–132, 2006.
[16] T. McConaghy, R. Marques, A. Muller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: a scalable blockchain database. white paper, BigChainDB, 2016.
[17] F. McKeen, I. Alexandrovich, A. Berenzon, et al. Innovative instructions and software model for isolated execution. In HASP, 2013.
[18] R. C. Merkle. A digital signature based on a conventional encryption function. In CRYPTO, pages 369–378, 1987.
[19] Microsoft. Azure Immutable Blob Storage. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blob-immutable-storage
[20] Microsoft. Azure SQL Database Point in Time Restore. https://azure.microsoft.com/en-us/blog/azure-sql-database-point-in-time-restore/
[21] Microsoft. Copy a transactionally consistent copy of a database in Azure SQL Database. https://docs.microsoft.com/en-us/azure/azure-sql/database/database-copy
[22] C. Mohan, D. J. Haderle, B.G. Lindsay, H. Pirahesh, P. M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM TODS, 17(1):94–162, 1992.
[23] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2004.
[24] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008
[25] Oracle. Oracle Blockchain Table https://docs.oracle.com/en/database/oracle/oracle-database/20/newft/oracle-blockchain-table.html
[26] H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying completeness of relational query results in data publishing. In SIGMOD, pages 407–418, 2005.
[27] H. Pang and K. Tan. Authenticating query results in edge computing. In ICDE, pages 560–571, 2004.
[28] F. M. Schuhknecht, A. Sharma, J. Dittrich, and D. Agrawal. ChainifyDB: How to Blockchainify any Data Management System. 2019.
[29] S. Singh and S. Prabhakar. Ensuring correctness over untrusted private database. In EDBT, pages 476–486, 2008
[30] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In CCS, pages 1480–1491, 2015.