Veritas: Shared Verifiable Databases and Tables in the Cloud

Lindsey Allen[†], Panagiotis Antonopoulos[†], Arvind Arasu[†], Johannes Gehrke[†], Joachim Hammer[†], James Hunter[†], Raghav Kaushik[†], Donald Kossmann[†], Jonathan Lee[†], Ravi Ramamurthy[†], Srinath Setty[†], Jakub Szymaszek[†], Alexander van Renen[‡], Ramarathnam Venkatesan[†]
[†]Microsoft Corporation [‡]Technische Universität München

ABSTRACT

In this paper we introduce shared, verifiable database tables, a new abstraction for trusted data sharing in the cloud.

KEYWORDS

Blockchain, data management

1 INTRODUCTION

Our economy depends on interactions – buyers and sellers, suppliers and manufacturers, professors and students, etc. Consider the scenario where there are two companies A and B, where B supplies widgets to A. A and B are exchanging data about the latest price for widgets and dates of when widgets were shipped. The price of a widget is fluctuating, and thus A bids for a quantity of widgets. In our (simplified) bidding process across the two companies, A asks for a quantity of widgets, B makes an offer, then A rejects or accepts the offer. Once A has accepted an offer, a contract exists, and B goes ahead and ships the widgets.

The two companies need to share data, and they would like to have an easy way to share asks, offers, prices, contracts, and shipping information. Thus they need to have an easy capability to share data. But with sharing and cooperation also come disputes, and thus companies A and B would like the shared data have an immutable audit log such that in case any disputes arise, all interactions with this shared table can be made available for auditing by a third party. This does not mean that the shared tables should be publicly accessible, but that selected parts of the log, the interactions with the table, can be made available to auditors, that the log is tamperproof, and that an auditor can quickly and cheaply reconstruct the shared state as of some point of time in the past and then audit state changes going forward.

How is this achieved today? Today, companies A and B share data by agreeing upon common APIs and data exchange formats that they use to make web service calls to each other. This solves the data sharing issue, however, this does not achieve immutability, nor does it enable the capability of selective auditing. An auditor would have to go through the databases at A and B, and trust that nobody tampered with the local logs. The auditors need special tools to derive an old state of the database from the logs, and the answers to read-only queries do not appear in the log at all. The problem is that neither reads nor updates are logged in some immutable way, and thus they cannot be audited by a third party without trusting that the local databases at A and B have not been tampered with.

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 allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2019.
CIDR'19, January 2019, CA
© 2019
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A recent class of new technology under the umbrella term "blockchain" [11, 13, 22, 28, 33] promises to solve this conundrum. For example, companies A and B could write the state of these shared tables into a blockchain infrastructure (such as Ethereum [33]). The blockchain then gives them transaction properties, such as atomicity of updates, durability, and isolation, and the blockchain allows a third party to go through its history and verify the transactions. Both companies A and B would have to write their updates into the blockchain, wait until they are committed, and also read any state changes from the blockchain as a means of sharing data across A and B. To implement permissioning, A and B can use a permissioned blockchain variant that allows transactions only from a set of well-defined participants; e.g., Ethereum [33].

This blockchain technology is quite a step forward, but also a few steps backwards, since with the power of these new blockchain technologies also come new problems. The application that handles interactions between A and B has to be re-written for the new blockchain platform, and the primitives that the new platform provides are very different than SQL, the lingua franca of existing data platforms. In addition, the transaction throughput for blockchains is low, and there is no query optimization or sophisticated query processing. Today, for gaining immutability and auditability with new blockchain platforms, we give up decades of research in data management – and hardened, enterprise-ready code that implements these ideas.

In this paper, we propose two new infrastructure abstractions to address the above issues. We propose the notion of a *blockchain database.* A blockchain database system is a database system with a regular SQL interface, however it provides the same guarantees of immutability of state transitions, trust, and open-source verifiability as a blockchain. Thus A and B can simply move their shared state to a blockchain database, and they now interact with it as with any other database using SQL. Such a blockchain database would address many of the limitations of blockchains such as the lack of a SQL interface. However, since the blockchain database is a physically different database system, interactions with it still need to happen through a middle tier; for example, Company A cannot simply write transactions across its own database system and the blockchain databbase; it would need to create a distributed transaction across its own database system and the blockchain database, a capability that in practice is always achieved through queues with asynchronous, idempotent writes. Thus while an interesting step forward in terms of capabilities, a blockchain database is always at arms-length from the database infrastructure at A.

In order to close this gap, we also propose the abstraction of a *shared database table* in the cloud that is part of the databases of both companies A and B. Assume that A and B are both customers of BigCloud, running their database infrastructure on the BigCloud SQL Database System. We will enable A to create a *shared table*, a table that A can share with B, and that now appears in both A's and

B's databases. The table can be used like any other table, and it can be synchronously accessed by both A and B; both companies can write application logic including ACID transactions that include this table.

By putting together both the idea of a blockchain database and the notion of a stared table, we arrive at the notion of a *shared, verifiable table*, which is part of the database of both companies A and B and has an immutable, accessible log with clean auditability capabilities.

How will we build such a shared, verifiable table? We propose to build on existing database technology; this gives us many benefits, but it also opens up several technical challenges which we start to address in this paper. The first challenge is to integrate blockchain technology into a database system, thereby maintaining the trust properties of a blockchain and the performance and productivity properties of a database system. One idea would be to leverage the logging capabilities of modern database systems for durability (i.e., the transaction log), but it turns out that the transaction log does not have the right information to verify the database system. A second challenge is to implement the shared table abstraction and to leverage possible optimization opportunities that arise from the shared table abstraction.

Note that there is another challenge that is often studied in the context of blockchains: Confidentiality. The nature of most blockchains (like Bitcoin [26]) is that all participants see all transactions so that they can validate the correctness of these transactions (e.g., no double-spending of the same bitcoin). Recently, confidentiality has been addressed in novel blockchain proposals such as Coco [11], Quorum [28], or Corda [13]. It turns out that confidentiality (and the specific blockchain protocol used) is to a large extent orthogonal to the issues discussed in this paper so that all these novel blockchain proposals are relevant to our work, but discussing confidentiality guarantees in sufficient detail is beyond the scope of this work.

Our Contributions. Our contributions in this paper are as follows:

- We introduce the notion of a verifiable (blockchain) database and the notion of a shared table. (Section 2).
- We discuss how to implement verifiable databases. (Section 3)
- We discuss how to implement shared, verifiable tables. (Section 4)
- We describe experimental results from a prototype system. (Section 5)

We discuss related work in Section 6, and we conclude in Section 7.

2 DATABASES AND TABLES THAT CAN BE SHARED AND VERIFIED

There are many different ways in which two (or more) parties such as Companies A and B can interact in the cloud. Figure 1 shows the traditional approach using Web Services (REST) or RPC. Company A carries out transactions using its local Database dbA. Transactions that involve an external entity such as ordering widgets for a given price from Company B are executed as follows: All updates that can be executed locally to dbA are committed synchronously; all requests to B are carried out asynchronously: The request is written into a local queue synchronously as part of the transaction so that the external requests are not lost in the event of failures. The local queue is part of dbA and thus no distributed transaction is needed. An asynchronous job will read the requests from the queue and make a Web Services call to Company B. The middle tier of B then receives the data and writes it to its own local database, Database dbB. If the transaction at dbB fails, then A may retry the transaction, and in case of repeated failures, it needs to execute a compensating transaction on dbA using Sagas or another nested transaction model [18].

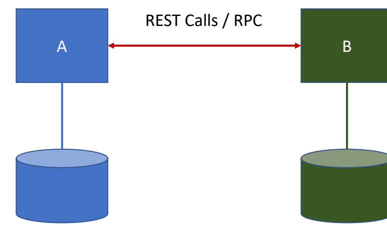

Figure 1: Web Services

The resulting architecture, which is shown in Figure 1, has been in use to drive electronic business for decades. It works extremely well once a framework for data exchange and trust between Companies A and B has been established. Setting it up, however, is cumbersome, and there is no good way to audit all the interactions between A and B. When there is a dispute, it is difficult to puzzle together what has happened, especially for more complex queries. If prices for widgets fluctuate frequently, for instance, it is difficult for A to prove that it placed an order at a lower price *before* B increased the price. While simple instances can be solved as a one-off by digitally signing API calls, the auditing of more complex queries such as a query over accounts payable that aggregates an outstanding balance is much more challenging. In general, the architecture of Figure 1 works best for large enterprises which have the resources and business volume to make it worth-while to set up, and which have the technical infrastructure to implement this architecture and legal scale to audit it.

2.1 Blockchain Databases

Recently, there has been a big hype around blockchain technology [22, 26, 28, 33] which allows entities to do business in a more agile way, without prior formal legal agreements. Figure 2 shows how A and B can collaborate using this architecture. Both A and B have their local databases and queues to affect reliable (exactly once) and asynchronous calls to external entities – just as in Figure 1. The crucial difference to Figure 1 is that all communication such as making quotes and placing orders is made through the blockchain. The blockchain provides a total order so that an auditor can reconstruct the exact sequence of events across A and B. Thus the auditor can prove that A placed its order before B increased its price. Using cryptographic methods, the blockchain is immutable so that neither A nor B can tamper with the history of interactions, and they cannot delete or reorder events.

The architecture of Figure 2 solves the trust and auditability issues of Figure 1. From an information infrastructure and database

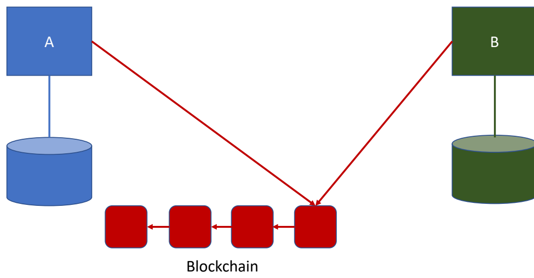

Figure 2: Blockchain

perspective, however, it is not ideal. Logically, the blockchain is a database which orders transactions and stores transactions durably. Modern blockchains such as Ethereum are even able to implement integrity constraints in form of smart contracts. Unfortunately, today's generation of blockchains cannot compete with modern database systems in terms of performance, query processing capabilities, productivity, and tooling. The blockchain community has realized that there is a significant gap between their existing technology and the capability of a modern database system. There are now a gazillion of start-ups that are adding these basic database features to blockchains, but it will take years if not decades to catch up.

Instead of adding database capabilities to blockchains, we propose to address the problem from the opposite approach: We add trust and auditability to existing database management systems. Figure 3 shows this concept. Instead of writing all transactions that span trust zones into a blockchain, these transactions are written into a shared *verified database* (which we also often call *blockchain database*). A verifiable database is an extension of a traditional DBMS (e.g., Microsoft SQL DB, MySQL, or Oracle) or a cloud database system (e.g., Microsoft SQL Azure or Amazon Aurora), and it contains all the tables with the shared data – in our example quotes and orders that keep the information accessible to A and B and any other parties that participate in this marketplace. The most important feature of this blockchain database is that it is *verifiable*. The provider who operates this database cannot be trusted because the provider might collude with A or B, the database might be hacked, or the database system source code might contain backdoors. The requirement is that the verifiable database gives the same trust guarantees as the blockchain in the architecture of Figure 2 – it contains an immutable, totally ordered log, and the state transitions can be verified by an auditor. This way, A and B can prove to an auditor that they behaved correctly and executed their actions in the right order.

Recently, Amazon announced QLDB [27]. QLDB was inspired by this *Blockchain Database* trend. In its current version, the QLDB service provider (i.e., Amazon) must be trusted so that QLDB provides different trust properties. In particular, the QLDB provider can drop transactions spuriously.

2.2 Shared, Verifiable Tables

While the concept of a blockchain database sounds attractive at first glance, it does not provide much additional value compared

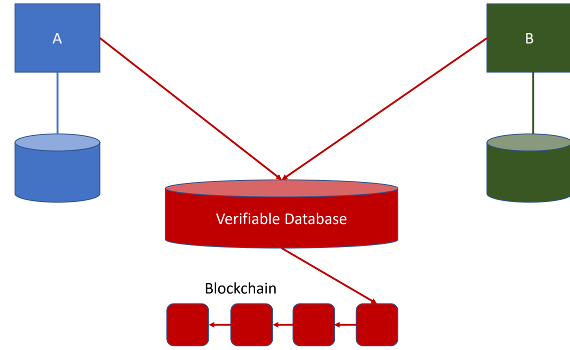

Figure 3: Verifiable Database

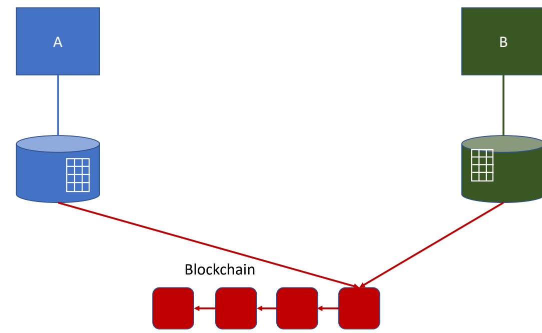

Figure 4: Verifiable Tables

to the architecture of Figure 2. Application developers and administrators still need to worry about distributed transactions across two databases and need to use reliable queues to implement transactions; there is asynchrony involved across different systems, and the middle tier needs to deal with one more system.

To have a truly seamless database experience and support trusted collaboration in the cloud, we propose the concept of a *shared, verifiable table* which integrates the tables from the blockchain database of Figure 3 directly into the databases of A and B – as if they were to share a single, common instance of this table.

Figure 4 shows how A and B collaborate using shared, verifiable tables. A writes all widget orders into a shared order table in its local database dbA. A can update and query this table just like any other table; there is no queuing delay, and A can write SQL and transactions across the shared tables and its private tables. However, the same *instance* of this table is also visible to B, which has the same capabilities as A against the shared, verifiable table. In addition, updates to all shared tables of a database are written to a tamper-proof and auditable log which could be implemented as a blockchain. More precisely, there is an N:1 relationship between shared tables and tamper-proof logs. Both A, B, and any auditor who can access the log, can see and inspect the log. An auditor can now prove whether A placed an order and whether it happened before after the latest price change. Likewise, B can publish all its price changes into a shared, verifiable quotes table which is also visible to A .

In the next section, we discuss how to implement a blockchain database and how to implement shared, verifiable tables.

3 IMPLEMENTING VERIFIABLE DATABASES

The previous section introduced two new abstractions: *verifiable database* and *verifiable table*. A verifiable database has all the features of a regular database: it talks SQL, stored procedures, supports complex query processing, indexes, integrity constraints, transactions, backup and recovery, etc. Furthermore, clients connect to a verifiable database using drivers such as ODBC, JDBC, etc. A verifiable database is a platform to share data and collaborate across clients, just like any other traditional database system. Analogously, a shared, verifiable table has all the features of a regular table of a relational database.

What makes verifiable databases and tables special is that they support tamper-evident collaboration across mutually untrusted entities. They provides verification mechanisms using which each party can verify the actions (updates) of all other parties and provide proof for its own actions. Further, all parties can verify that the state of the shared database (or shared table) and its responses to queries is consistent with prior actions of legitimate actors. These guarantees are cryptographic. Unauthorized parties (hackers or operators with administrative privileges) cannot tamper with the state of the verifiable database (or verifiable table) without being detected by the verification mechanism.

Just like blockchains, a verifiable database system that supports verifiable databases and tables relies on cryptographic identities to establish the legitimacy and originator of actions. The system is *permissioned* and tracks the identities (public keys) of authorized participants. An authorized participant signs all interactions (queries) with the system, and the signatures are checked for authenticity during verification. Queries are also associated with unique identifiers to prevent replay attacks.

A second critical building block is that the verifiable database system logs all client requests and the affects of these requests. As part of this log, the system adds information (typically hashes) which allow auditors and participants to verify whether the database behaves correctly. We call all agents that audit the database in this way *verifiers*. Since confidentiality is out of the scope of this paper, we assume that all verifiers have access to the whole log generated by the verifiable database. Frameworks like Coco [11], Quorum [28], Spice [30] or Corda [13] address confidentiality in Blockchains and the ideas of these frameworks are applicable to our work because they are in principle orthogonal.

Given these three building blocks, there are many different ways to implement a verifiable database system. Essentially, there are three fundamental design questions:

- How do the verifiers consume the log of the verifiable database and generate consensus on the state of the verifiable database?
- How do the verifiers check whether the verifiable database does the right thing?
- Do verifiers check transactions synchronously or asynchronously?

The remainder of this subsection describes our proposals to address these three questions in the Veritas system that we are currently building at MSR. We address these questions in the context of verifiable databases. We discuss the specifics of verifiable tables in Section 4.

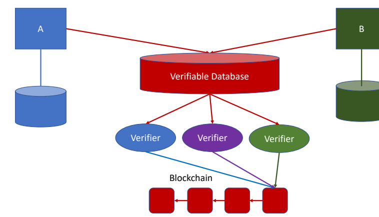

Figure 5: Verifiers and Blockchain

3.1 Verification Architecture

Figure 5 shows one possible way to add verifiers to a verifiable database (Figure 3). The verifiers consume the log produced by the verifiable database, check whether everything is okay (see next subsection), and write their vote into a Blockchain.

In this architecture, the Blockchain is only used to store the votes of verifiers. Depending on the specific Blockchain technology, the cost and performance of Blockchain operations varies, but they are always expensive. So, it is advantageous to minimize the number of Blockchain operations. In the architecture of Figure 5, verifiers can further reduce the number of times they write to the Blockchain by *batching* their votes. Instead of acknowledging each log record of the verifiable database, they record into the Blockchain the position of the position until which they have verified all operations of the verifiable database. If there is a dispute, all parties can reconcile the history with the log of the verifiable database (which needs to be archived) and the votes of the verifiers in the Blockchain. This protocol is called "Caesar Consensus", and it was first devised for the Volt system [31].

Disseminiating the log of the verifiable database to the verifiers across a wide-area network can be expensive and slow. One variant of the architecture of Figure 5 is to leverage emerging technology to create *trusted execution environments* (TEEs) and collocate the TEE with the verifiable database. Examples of TEEs are Intel SGX [14], ARM Trustzone [2], Windows VBS [32], or FPGAs [17]. In this variant, the TEE consumes the log and verifies that the verifiable database operates correctly, informs all other verifiers, and writes its vote into the Blockchain. This variant is inspired by the Coco framework [11]; in addition to performance enhancements, this approach can also be used to implement confidentiality in the architecture of Figure 5. To make this approach secure, the TEE needs to be attested by all parties (and other verifiers) in order to make sure that it does not run malicious code and verifies correctly.

3.2 Fine-grained vs. Coarse-grained Verification

The second design question addresses the inner workings of the verifiers. The "Blockchain way" of building verifiers is implementing verifiers as replicated state machines [24]. That is, the verifiable database logs all requests of clients and the answers returned to the clients. Each verifier implements exactly the same logic as the

verifiable database system: It retruns the client requests and checks whether it returns the same results. If the results match, the verifier votes that everything is okay; otherwise, it signals a violation. To make the log reproducible, the results of indeterministic functions (e.g., *random()* or *getTimeOfDay()* need to be logged, too. We call this approach *coarse-grained verification.*

Conceptually, coarse-grained verification is simple. However, it is also costly because it repeats expensive database operations such as joins and sorting. For instance, it is easier to verify the result of a *sort* operator than to sort the sequence again. Furthermore, the coarse-grained approach requires that the verifier maintains a full copy of the whole database. As a result, the coarse-grained approach does not compose well with the TEE variant described in the previous subsection. State-of-the-art TEEs (e.g., Intel SGX) have severe memory limitations and cannot carry out I/O operations reliably, thereby making it difficult to embed a full-fledged DBMS into a TEE. Another severe issue of the coarse-grained verification approach is that the verifier inherits all bugs of the database system: The verifier has a large trusted computing base (TCB), and it is virtually impossible to verify the correctness of the software that drives the verify.

As an alternative, a fine-grained verifier only checks the *logical* properties of the results and affects of all database operations. This approach is particularly attractive for databases because the semantics of database operations is (fairly well) standardized. A brute-force version of a fine-grained verifier is also straight-forward to implement: It will involve logging every record that is read and created as intermediary or final query result. The current version of Veritas implements such a brute-force fine-grained verification. It works well for simple, OLTP-style database statements (point lookups and updates). This approach also parallelizes verification well, following the design devised in [1]. Parallelizing verification is important so that the verifiers can keep up with the Verifiable database system which has a high degree of concurrency. More research, however, is needed to optimize this approach for more complex queries. Furthermore, more research on the fine-grained approach is needed to check whether the verifiable database system used a correct query plan (for complex queries) and whether the verifiable database system guaranteed the desired isolation level.

Veritas uses the fine-grained approach for three reasons: (a) verification is potentially cheaper as the sorting example in the previous paragraph shows, (b) footprint, and (c) security. The fine-grained approach has a small TCB, and fine-grained verifiers can be virtually stateless. The verifier of the Concerto key-value store [1] which is the design we adopted for Veritas, for instance, is composed of two simple AES blocks implemented in an FPGA and provably verifies the affects of a key-value store with hundred thousands of lines of code. It is also fully streaming and only keeps a few hashes as internal state. In contrast, a coarse-grained verifier is a full-fledged DBMS with millions lines of code and involves replicating the whole database. A small TCB and footprint is particularly important to deploy a verifier using TEEs such as Intel SGX and FPGAs that have memory constraints.

3.3 Online vs. Deferred Verification

The third design question has attracted a great deal of attention in the research literature [8, 16]. Online verification means that the transaction cannot commit in the verifiable database until it has been verified by all (or a quorum) of verifiers. This approach is important for transactions that have external effects; e.g., dispensing cash at an ATM. In contrast, deferred verification batches transactions and periodically validates the affects of this set of transactions in an asynchronous way.

It turns out that in the absence of external effects which cannot be rolled back, online and deferred verification provide the same trust guarantees. If the verification of a transaction fails in the online model, then just rolling back that specific transaction might not be enough. In fact, the transaction might be perfectly okay and verification fails because of an integrity violation with the verifiable database by, e.g., a malicious admin a long time before this transaction. So, online verification does *not* guarantee immediate detection of malicious behavior, a common misunderstanding of online verification.

4 IMPLEMENTING SHARED, VERIFIABLE TABLES

Essentially, all the design considerations to implement *trust* (i.e., distributed verifiers) in a verifiable database discussed in Section 3 apply in the same way to the implementation of shared, verifiable tables. Conceptually, the big difference between verifiable databases and verifiable tables is concurrency control. Concurrency control is centralized in a verifiable database and implemented in the same way as in any traditional database system (e.g., using Snapshot Isolation [4] or two-phase locking [20]). In contrast, concurrency control is fundamentally distributed in a system that supports shared, verifiable tables.

As shown in Figure 4, verifiable tables are a way to integrate data from a shared, verifiable database into a local database. The advantage is that applications need not worry about distributed transactions across a local and a shared database. Distributed transactions are fundamental to the *shared, verifiable table* concept as each node is autonomous and carries out transactions on the shared (and private) tables independently. So, rather than letting the application mediate between the local database and the Blockchain or verifiable database, the database system needs to implement distributed coordination of transactions to support verifiable tables.

There are many ways to implement distributed transactions [5]. Recently, there has been a series of work on implementing distributed Snapshot Isolation [7, 15]. For Veritas, we chose to use a scheme that is based on a centralized ordering service and that is inspired by the Hyperledge Blockchain protocol.

Figure 6 depicts an example scenario with three parties, denoted as Node 0, Node 1, and Node 2. Each of these nodes operates a Veritas database on behalf of their organization which includes private tables only visible to users of that organization and shared, verifiable tables visible to users of all three organizations. Users and applications of each node issue transactions to secret and shared tables in a transparent way; that is, users and application programs need not be aware of the existence of other nodes and organizations, and they need not be aware of the difference between private and

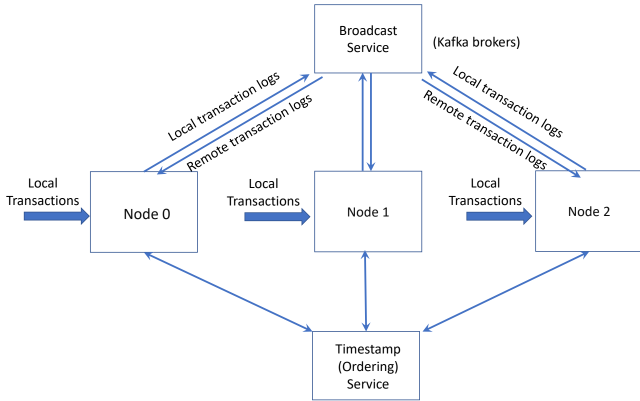

Figure 6: Shared Tables Implementation Architecture

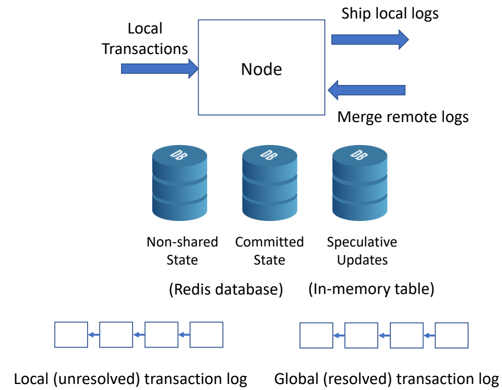

Figure 7: State of a Veritas Node

shared tables. The database system at each node deals with all issues concerning distributed transactions, keeping the copies of shared tables consistent, and making all updates to shared tables verifiable.

Transactions that involve only private tables are executed locally only. Coordination is only needed for transactions that involve reads and updates to shared tables. For such "global" transactions, the database system does two things, as shown in Figure 6:

- *Concurrency Control:* We use Timestamp-based concurrency control in our prototype implementation of Veritas.
- Disseminate Log and Voting: We use a Kafka-based broadcast service to propagate all updates of a shared table to all nodes. Each node embeds its votes (following the "Caesar Consensus Protocol" of Section 3.1) into the log that it ships to all other nodes.

Again, there are many ways to effect distributed concurrency contol for a system that supports shared tables like Veritas. For simplicity, we chose to use an optimistic protocol that is based on Timestamp ordering with a centralized Timestamp service á la Hyperledger in our prototype implementation of Veritas. At the beginning of a global transaction that accesses a shared table, Veritas calls the Timestamp service to get a global timestamp for the global transaction. This timestamp determines the commit order of all global transactions.

Once a global transaction has received a (global) timestamp, it is executed locally at the Veritas node speculatively (or optimistically). While the global transaction is executed it is not yet known whether it will commit successfully. In Veritas, a transaction can be aborted due to concurrency control conflicts or user aborts, just like in any other database system. Furthermore, transaction on shared tables can fail in Veritas if they are not verified by the other Veritas nodes.

To execute the transaction, each Veritas node keeps two data structures: (1) a *commit watermark*, T_C that indicates the highest global timestamp for which the node knows whether all global transactions with timestamp less than T_C committed successfully or aborted. (2) A version history of all records of all shared tables. This version history contains the latest version of each record commit by a global transaction before T_C and all versions of the record of speculative transactions with timestamp higher than T_C.

Periodically, every Veritas node ships all log records that involve reads and writes to shared tables as well as the *begin of transaction*, *commit*, and *abort* log records of all global transactions to all other Veritas nodes using the Broadcast Service (Figure 6). When a Veritas node receives log records from another node, it applies that log and verifies the affects of all committed transactions. (The node buffers log records of active transactions until it receives the *commit* or *abort* log records for those transactions.) While applying that log, the node checks for read/write and write/write conflicts using its version store of records of the shared tables. If the node detects a conflict, it marks the transaction as aborted and ignores all updates to its copy of the shared table. If the transaction can be verified and does not create any conflicts, the transaction is marked as committed and the node's copy of the shared tables are updated accordingly.

Figure 7 depicts the state and log maintained by each Veritas node. In addition to the private (non-shared) tables, every Veritas node keeps a *clean* copy of all shared tables - this clean copy contains the state of all records of shared tables as of time T_C, the commit watermark. In the current Veritas prototype, the non-shared tables and the clean shared tables are persisted in a Redis database. Furthermore, the Veritas node keeps all (known) versions of records of shared tables of *speculative* transactions; i.e., transactions after T_C so that the node does not know the destiny of those transactions yet. Furthermore, each Veritas node keeps a *local* log of all updates to private (non-shared) tables and a global log of updates to shared tables.

In this scheme, the (centralized) Timestamp service is a potential vulnerability of the system because it must be trusted. Again, there are many possible ways to implement distributed transactions, including variants that do not rely on a centralized service. Exploring the full space of these protocals and variants is beyond the scope of this paper. The goal of this paper is to give a flavor of one possible way to interleave concurrency control and verification in a distributed system that implements shared tables.

5 EXPERIMENTAL RESULTS

This section contains the results of initial performance experiments done with the Veritas prototype described in Sections 3 and 4. The goal of these experiments was to study the overheads of verification

as a function of the number of nodes. Furthermore, we wanted to study the interplay of distribution concurrency control and verification, depending on data contention.

5.1 Benchmark Environment

We used a variant of the YCSB benchmark [12] for all performance experiments reported in this paper. We used a database with only a single (shared) table and varied the number of records in that shared table. Every record of the shared table was a key/value pair; both key and value were short strings of eight bytes.

We studied workloads with *get()* and *put()* operations only. All operations accessed a random record of the shared table using a uniform distribution across all records of the shared table. While our prototype supports more general transactions, here we report results for simple transactions with a single read or a single update (read followed by a write). We used three different workload mixes: (A) update-heavy (50% operations are updates), (B) read-heavy (5% update rate), and (C) read-only (0% update rate).

In all experiments, we ran a fixed number of transactions (one million transactions, if not stated otherwise) and measured the time it took to execute this batch of transactions across all nodes of the system. The same number of transactions originated at every node; e.g., for a four-node system 250000 transactions originated at each node. Furthermore, we measured the number of transaction aborted due to concurrency conflicts caused by the optimistic protocol described in Section 4. From this number, we computed the commit rate: (*total number of transactions - aborted transactions*) / *total number of transactions*. From these two measurements, we derived the throughput as the number of committed transactions per second. We varied the numnber of nodes (Experiment 1) and the number of records in the shared table (Experiment 2).

This paper only reports on experiments done with the shared table approach (Section 4). The throughput and commit rate of verified databases in the Veritas implementation (Section 3) is almost the same as in a traditional database system as Veritas does verification asynchronously and in batches. The only overheads are the creation of digital signatures at clients and the shipping of the log to the verifiers.

We ran all Veritas nodes in Azure on VMs with four virtual cores and 16 GB of main memory (i.e., Azure D4sv3 instances). Each node ran a Redis database to store its copy of the shared database persistently. The Timestamp service shown in Figure 6 was implemented using Zookeeper. The Broadcast service was implemented using Kafka. Both the Timestamp service and the Broadcast service were deployed on separate Azure VMs (again with four cores and 16 GB of main memory).

We implemented the Veritas prototype in C#; it currently has about 1500 lines of code. Each node batches transactions before shipping the log to the Broadcast service. The batch size was 10,000 transactions in all experiments reported in this paper.

5.2 Experiment 1: Vary Number of Nodes

Figure 8 shows the throughput of Veritas for a database with 100K records and all three workload types (update-heavy, read-heavy, and read-only). Figure 8 shows the results for Veritas configurations with 2, 3, and 4 nodes. As a baseline, the figure also shows a configuration with one Veritas node. With one node, Veritas behaves like any conventional database system as there is no coordination with other nodes required.

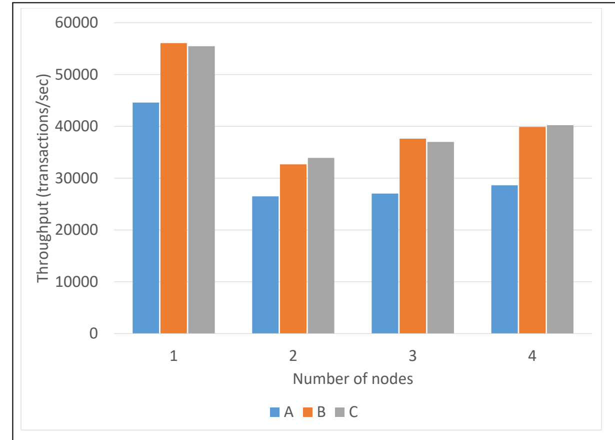

Figure 8: Throughput: Vary Nodes, 100K DB

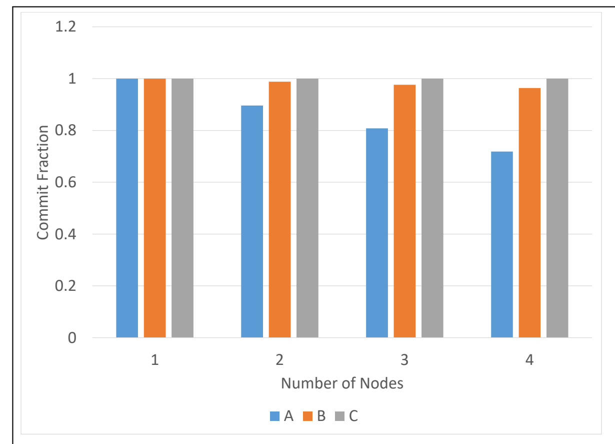

Figure 9: Commit Rate: Vary Nodes, 100K DB

As expected, there is a tax for distributed concurrency control and verification in Veritas. For the update-heavy workload (A), the throughput is only slightly more than half with distributed concurrency control and verification. With a more sophisticated distributed concurrency control protocol, we could probably improve these results. However, the figure shows that even for read-only workloads (C), the best case for the optimistic concurrency control scheme currently implemented in Veritas, the overhead is substantial. Nevertheless, Figure 8 also shows that the overhead is not catastrophic; even with the simple scheme currently used in Veritas is it possible to achieve tens of thousands transactions per second.

Figure 9 shows the commit rates for this experiment. Obviously and expectedly, the conflict rate drops with a growing number of nodes as the number of conflicts increases with the number of concurrent nodes. Furthermore, as expected, the drop is most significant for the update-heavy workload (A). Comparing the commit

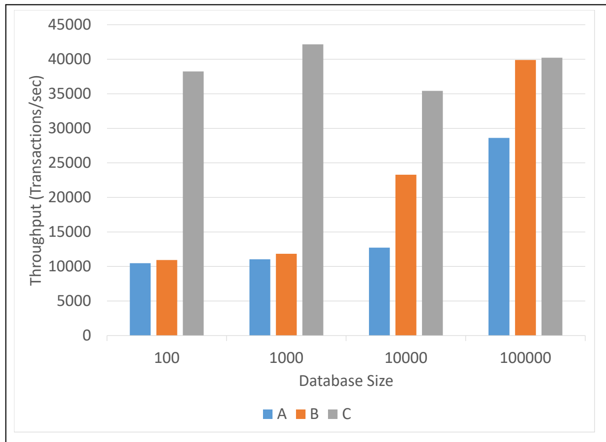

Figure 10: Throughput: 4 Nodes, vary database size (number of records)

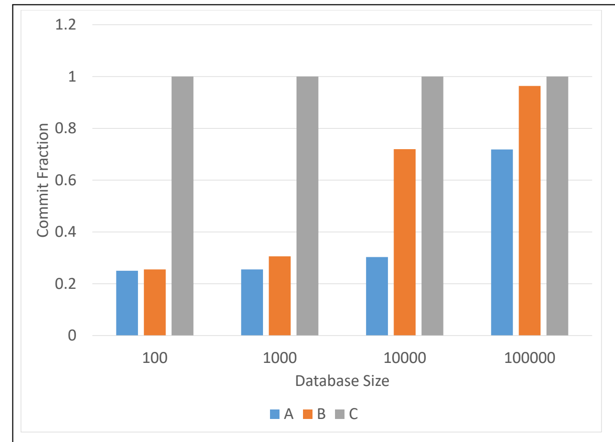

Figure 11: Commit Rate: 4 Nodes, vary database size (number of records)

rates of Figure 9 with the throughput drop shown in Figure 8, it can be infered that as a rule of thumb, about half of the throughput drop can be explained by the impact of distributed concurrency control and half of the drop is due to the complexity of verification. The exact split depends on the number of nodes (the more nodes, the higher the relative impact of distributed concurrency control), but the 50/50 split is a nice rule of thumb for deployments with few nodes.

5.3 Experiment 2: Vary Database Size

To investigate the impact of concurrency control and data contention in more detail, we did a series of experiments with a varying number of records in the shared table (while keeping the number of nodes fixed at four): The fewer records, the higher the number of conflicts, and, thus, the higher the number of aborts due to our simple optimistic concurrency control scheme. Figures 10 and 11 show the results. As expected the throughput increases with the size of the database and a correspondigly lower number of conflicts and aborts. In the extreme of an update-heavy workload with a very small database of only 100 records, the throughput is roughly 1/4 of the throughput of a read-only workload. This result shows that (a) Veritas does not starve, even with extremely high data contention and (b) delivers roughly the expected throughput of 1/(*number of nodes* as the coordination effort is proportional to the number of nodes. This result is encouraging because it shows that the implementation of shared tables in Veritas does not create any additional bottlenecks: Indeed, verification and distributed concurrency control has a price as shown in Experiment 1 - but the bottlenecks of a database system that uses verification (and distributed trust) are the same as the bottlenecks of a traditional database system.

6 RELATED WORK

Blockchain Systems: Bitcoin [26] is one of the first blockchain systems proposed for decentralized cryptocurrencies. Bitcoin is a public blockchain that relies on cryptographic puzzles (proof-of-work) and incentives for decentralized trust. Ethereum [33] generalizes these ideas for more general computation. Both Bitcoin and Ethereum are public blockchains, meaning anyone can join the network and participate in the blockchain protocol.

For the enterprise applications we are interested in, private blockchains are more relevant. Examples of such blockchains include Hyperledger [22] and Quorum [28]. These systems use different mechanisms such as PBFT [9] for consensus rather than proof-of-work. But their verification methodology is similar to the public blockchains and relies on mirroring the state and rerunning transactions at every node. Recent blockchain proposals such as Coco [11], Ekiden [10], and Intel Sawtooth Lake [29] rely on TEEs such as Intel SGX [14] for confidentiality and performance. As noted above, Veritas can be combined with similar ideas to achieve confidentiality. Another interesting approach to confidentiality is Corda [13] that does not use a single global ledger of transactions, but a decentralized network of notaries to prevent double spending attacks.

Database Systems: BigchainDB [6] is a recent system that overlays blockchain consensus ideas such as PBFT over a database system (MongoDB [25]). While the overall goals of BigchainDB are similar to ours, there are fundamental architectural differences. BigchainDB, similar to traditional blockchains, relies on replicating the state and rerunning transactions at every node. Further, the entire MongoDB code is part of the TCB of the system. Veritas architecture allows the database state and query processing to be centralized, with much smaller verifier TCB. Verifiable databases are related to the large body of work on ensuring data integrity in outsourced settings [1, 3, 23, 34]. Most of this work deals with verifying data integrity for analytical workloads [3, 34] with poor update performance [23]. Concerto [1] is a recent system that addresses update performance concerns of previous systems and is the source for many of the technical ideas underlying Veritas.

Other work: Google Fusion Tables [21] is a system for sharing and distributed editing of tables, but it does not provide the verifiability

guarantees required for blockchain style applications. There is also a lot of work on data integration that is relevant [19].

7 CONCLUSIONS

We introduced two new abstractions. A verifiable database system creates an immutable log and allows an auditor to check the validity of query answers and updates that the database produced. A *shared, verifiable table* creates the same abstraction at the scale of a table which can be shared and thus part of many different database instances in the cloud — which operate on the table as it is were a single table.

Experimental results with our prototype system Veritas show that our abstractions scale with the verification overhead and with the distributed concurrency control schema.

REFERENCES

[1] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 251–266.
[2] ARM TrustZone 2018. Arm TrustZone. https://developer.arm.com/technologies/trustzone.
[3] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL Engine with Practical Query Authentication. *PVLDB* 6, 7 (2013), 529–540.
[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, Vol. 24. ACM, 1–10.
[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx
[6] BigchainDB 2018. BigchainDB: The Blockchain Database. https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf.
[7] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. 2014. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *VLDB J.* 23, 6 (2014), 987–1011. https://doi.org/10.1007/s00778-014-0359-9
[8] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. 1994. Checking the Correctness of Memories. *Algorithmica* 12, 2/3 (1994), 225–244.
[9] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461.
[10] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *CoRR* abs/1804.05141 (2018). http://arxiv.org/abs/1804.05141
[11] Coco 2017. Coco framework. https://github.com/Azure/coco-framework.
[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2008. Benchmarking cloud serving systems with YCSB. *Symposium on Cloud Computing* (2008).
[13] Corda 2018. Corda blockchain. https://www.corda.net/.
[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86. http://eprint.iacr.org/2016/086
[15] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*. IEEE Computer Society, 173–184. https://doi.org/10.1109/SRDS.2013.26
[16] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. 2009. How Efficient Can Memory Checking Be?. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*. 503–520.
[17] Ken Eguro and Ramarathnam Venkatesan. 2012. FPGAs for trusted cloud computing. In *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*. 63–70.
[18] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. 1991. Modeling Long-Running Activities as Nested Sagas. *IEEE Data Eng. Bull.* 14, 1 (1991), 14–18.
[19] Behzad Golshan, Alon Y. Halevy, George A. Mihaila, and Wang-Chiew Tan. 2017. Data Integration: After the Teenage Years. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. 101–106.
[20] Jim Gray and Andreas Reuter. 1990. *Transaction Processing: Concepts and Techniques*.
[21] Alon Y. Halevy. 2013. Data Publishing and Sharing using Fusion Tables. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*.
[22] Hyperledger 2018. Hyperledger - Blockchain for Business. https://www.hyperledger.org/.
[23] Rohit Jain and Sunil Prabhakar. 2013. Trustworthy data from untrusted databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 529–540.
[24] Leslie Lamport. 1984. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Trans. Program. Lang. Syst.* 6, 2 (1984), 254–280.
[25] MongoDB 2018. MongoDB. https://www.mongodb.com/.
[26] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.
[27] QLDB 2018. Amazon Quantum Ledger Database (QLDB). https://aws.amazon.com/qldb/.
[28] Quorum 2018. Quorum Advanced Blockchain Technology. https://www.jpmorgan.com/global/Quorum.
[29] Sawtooth Lake 2018. Intel Sawtooth Lake. https://intelledger.github.io/.
[30] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. 2018. Verifiable and private execution of concurrent services with Spice. In *OSDI*. To appear.
[31] Srinath Setty, Soumya Basu, Lidong Zhou, Michael Lowell Roberts, and Ramarathnam Venkatesan. 2017. *Enabling secure and resource-efficient blockchain networks with VOLT*. Technical Report. Technical Report MSR-TR-2017-38, Microsoft Research.
[32] Windows VBS 2018. Windows Virtualization Based Security. https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.
[33] Gavin Wood. 2018. Ethereum: A secure decentralized transaction ledger. http://gavwood.com/paper.pdf.
[34] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 863–880.