ScalarDB: Universal Transaction Manager for Polystores

Hiroyuki Yamada
Scalar, Inc.
hiroyuki@scalar-labs.com

Toshihiro Suzuki
Scalar, Inc.
toshihiro@scalar-labs.com

Yuji Ito
Scalar, Inc.
yuji@scalar-labs.com

Jun Nemoto
Scalar, Inc.
jun@scalar-labs.com

ABSTRACT

This paper presents ScalarDB, a universal transaction manager that achieves distributed transactions across multiple disparate databases. ScalarDB provides a database-agnostic transaction manager on top of its database abstraction; thus, it achieves transactions spanning various databases without depending on the transactional capability of underlying databases. ScalarDB is based on several research works and extended to provide a strong correctness guarantee (i.e., strict serializability), further performance optimizations, and several critical mechanisms for productization. In this paper, we describe the design and implementation of ScalarDB. We also present evaluation results showing that ScalarDB achieves database-spanning transactions with reasonable performance and near-linear scalability without sacrificing correctness. Finally, we share some case studies and lessons learned while building and running ScalarDB.

PVLDB Reference Format:
Hiroyuki Yamada, Toshihiro Suzuki, Yuji Ito, and Jun Nemoto. ScalarDB: Universal Transaction Manager for Polystores. PVLDB, 16(12): 3768 - 3780, 2023.
doi:10.14778/3611540.3611563

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/scalar-labs/scalardb.

1 INTRODUCTION

One size does not fit all [61] is becoming common sense in data management systems. Major cloud vendors offer several purpose-built database products to meet various users' needs. For instance, Amazon AWS offers more than 10 database products [58], such as Aurora (relational) [66], DynamoDB (key-value) [15], Neptune (graph) [56], and QLDB (ledger) [57]. Unsurprisingly, there are many cases where an application uses multiple databases to provide its services.

A microservice architecture [7, 32, 49] accelerates the trend of managing multiple (potentially disparate) databases. Each microservice of a single application is encouraged to use an isolated database, which is selected based on the service's use cases and the developers' experiences for better maintainability and productivity. This architectural style is likely to make an application have different kinds of databases or multiple database instances of the same database.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611563

Managing multiple disparate databases is not uncommon in enterprise systems as well. An enterprise comprises several organizations, departments, and business units to support agile business operations. This leads to *siloed* information systems; different organizations manage different applications at disparate locations, and the applications use different databases [27, 62].

Obviously, a federation of multiple disparate databases is attractive for such applications to mitigate the complexity of dealing with the databases separately. Federated database systems and multi-database systems [4, 28, 43, 59] have been explored since around the 1990s to address the goal. Due to the increasing deployments of multiple disparate databases, there are also new demands [14, 16, 60], such as supporting various query notations, providing all the functionalities of underlying databases, and providing distributed transactions across multiple databases that do not support the same transaction model. The database community recently named such new federated database systems *polystores* [5, 14, 60] to distinguish them from previous federated database systems.

In this paper, we present a universal transaction manager for polystores called ScalarDB, which achieves distributed transactions across multiple disparate databases. Specifically, ScalarDB provides a database-agnostic transaction manager on top of its database abstraction; thus, it achieves transactions spanning multiple disparate databases without depending on the transactional capability of underlying databases. ScalarDB is based on several research works and extended to provide a strong correctness guarantee (i.e., strict serializability [30]), further performance optimizations, and several critical mechanisms for productization.

ScalarDB has been built to meet several key design goals: database agnosticism as a primary goal, strong correctness, reasonable performance, high scalability, and high availability. Specifically, ScalarDB provides a database-agnostic property and can run transactions not only on major relational database systems such as MySQL, MariaDB, PostgreSQL, Oracle Database, and Microsoft SQL Server but also NoSQL databases such as Apache Cassandra, Amazon DynamoDB, and Azure Cosmos DB while addressing the other design goals. Therefore, for example, it achieves scalable, strict serializable ACID transactions spanning PostgreSQL and Amazon DynamoDB, which cannot be easily realized with existing solutions.

Existing off-the-shelf solutions aiming to achieve distributed transactions over multiple disparate databases do not fully match our design goals. Oracle Tuxedo [48], Atomikos [2], and Seata (XA mode) [54] are middleware that manage distributed transactions over multiple databases based on X/Open XA [24], which is a standard specification for allowing multiple independent resources to participate in a single and distributed transaction by using the two-phase commit (2PC) protocol [22]. Although the two-phase commit protocol based on XA can work on XA-compliant databases, such as major relational databases [25, 45–47], it cannot run transactions on other databases, such as NoSQL databases [18, 19, 44, 55] and recent

distributed SQL engines [21, 31, 63, 69] that are not XA compliant. Making databases XA compliant is not necessarily straightforward or impractical due to the rigid XA specification. Several frameworks [2, 54] help users to implement other approaches, such as Try-Confirm/Cancel (TCC) [29] and Saga [20], to run distributed transactions over multiple databases in a non-strict and lightweight way. These approaches could work on a wider range of databases. However, they only guarantee eventual consistency and weaker isolation than serializable because they realize an application-level transaction by using multiple database-level transactions; the applications must deal with transaction anomalies by themselves.

ScalarDB is a production-grade system that has been used for real-world applications. It is also cloud-agnostic and designed for cloud-native applications. ScalarDB is provided as a Docker container and can easily be deployed to various environments such as Kubernetes. The source code for the core components of ScalarDB is available on GitHub [51] under the Apache 2.0 License.[1]

This paper makes the following contributions:

- We describe the design and implementation of ScalarDB, a universal transaction manager that achieves database-agnostic and database-spanning transactions. Specifically, we describe how ScalarDB has incorporated and extended previous research efforts to build a practical and cloud-native product.
- We present evaluation results showing that ScalarDB achieves database-spanning transactions with reasonable performance and near-linear scalability. We also show ScalarDB's database-agnostic property by evaluating ScalarDB on several database systems.
- We share a couple of case studies of ScalarDB. We also share lessons learned not only from the experiences of building ScalarDB over the last five years but also from our production experiences.

The remainder of the paper is organized as follows. Section 2 discusses our design goals and key challenges of ScalarDB. Section 3 introduces an overview of ScalarDB. Section 4 discusses how ScalarDB guarantees strong correctness. Section 5 discusses how ScalarDB optimizes performance. Section 6 discusses several critical mechanisms of ScalarDB for productization. Section 7 presents the results of our evaluation. Section 8 presents ScalarDB case studies. Section 9 presents lessons learned while building and running ScalarDB and some future work. Finally, Section 10 concludes the paper.

2 DESIGN GOALS AND CHALLENGES

This section introduces our design goals for transaction management that achieves distributed transactions spanning multiple disparate databases. In this paper, we call these transactions *global* transactions. Then, we provide possible approaches for achieving global transactions and share our design choice for ScalarDB. Lastly, we summarize the challenges we address with ScalarDB.

2.1 Design Goals

After carefully considering our customers' feedback, we set the following design goals to make global transactions practically usable.

[1]The source code of some other components is not publicly available and is licensed under a commercial license.

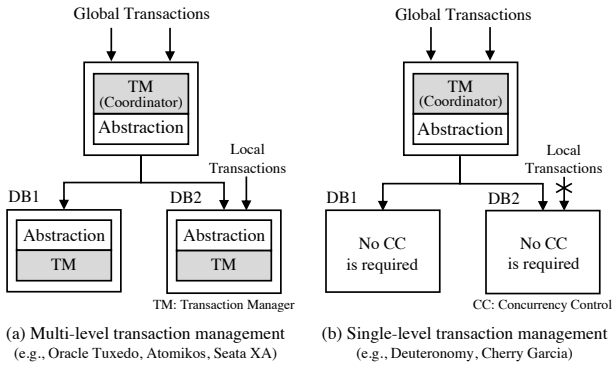


Figure 1: Design choices for achieving global transactions.

Database Agnosticism. Database-agnosticism is a primary goal for achieving global transactions. Global transactions should be able to span not only relational databases but also NoSQL databases and other types of databases. This is the top-priority goal based on our customers' demands.[2]

Strong Correctness. Correctness is an essential property in transactional systems. Specifically, distributed transactions should be ACID compliant with a strict serializability guarantee for the intuitiveness of results and ease of development.

Reasonable Performance. Although managing global transactions is non-trivial work, it should not be a limiting factor of overall transaction performance.

High Scalability. Scalability is one of the common key factors in distributed transactional systems. Specifically, transaction performance should scale as the performances of underlying databases scale.

High Availability. Availability is also one of the common key factors in distributed transactional systems. Managing global transactions should be achievable without a single point of failure, and its availability should be increased as more computing resources (e.g., nodes) are used.

2.2 Design Choices

The approaches for achieving global transactions can be categorized in two ways: multi-level transaction management and single-level transaction management, as shown in Figure 1. In both approaches, there is a coordination process (transaction coordinator or coordinator) that manages global transactions with a two-phase commit protocol or its extended variants, and there are databases that participate in the global transactions. In this section, we clarify the advantages and disadvantages of the approaches and share our decision based on the design goals.

Multi-level transaction management. The multi-level transaction management (MultiTM) approach (Figure 1(a)) achieves global transactions by coordinating transactions at the coordinator in cooperation with the transaction managers of underlying databases. Therefore, there are abstractions (e.g., specifications) in both the coordinator and underlying databases for cooperation. Examples of

[2]We show one of our customer case studies that uses global transactions in Section 8.

systems that follow this approach are Oracle Tuxedo [48], Atomikos [2], and Seata (XA mode) [54], which are all based on X/Open XA [24].

The advantages of the MultiTM approach are as follows: (1) There is a standard specification such as X/Open XA, along with some real-world products (e.g., Oracle Tuxedo) based on the specification. Moreover, there is a long history of research on the approach [4, 43, 59]. Therefore, we have already learned some lessons to exploit. (2) Optimizing transaction performance is straightforward. That is because underlying databases do most of the concurrency control work for global transactions and the optimization of transaction performance for those underlying databases would benefit global transaction performance. (3) Local transactions, which only go to one database, can naturally go to the database directly without coordinating with global transactions because global transactions use the concurrency control mechanism of underlying databases.

On the other hand, the approach has some disadvantages: (1) The approach could force underlying databases to meet strict requirements. For example, XA requires underlying databases to be ACID compliant with some specific implementation for concurrency control. This constraint makes it hard to support various databases, such as NoSQL databases, that do not guarantee ACID. (2) The approach is invasive. Specifically, it forces underlying databases to be modified or enhanced to follow the specification of a coordination protocol. Therefore, the approach could be less attractive or too much of a burden for some databases. In our research, we found that many recent distributed databases [21, 31, 63, 69] are ACID compliant but not XA compliant.

Single-level transaction management. The single-level transaction management (SingleTM) approach (Figure 1(b)) achieves global transactions by managing and coordinating transactions only at the coordinator and does not rely on the concurrency control mechanism of underlying databases. The approach typically and necessarily abstracts underlying databases in the coordinator layer to achieve global transactions. Examples of systems that follow this approach are Deuteronomy [40, 41] and Cherry Garcia [12].

The advantages of the approach are as follows: (1) The approach requires underlying databases to meet weaker requirements than the ones (i.e., ACID compliance) for the MultiTM approach. For example, Deuteronomy and Cherry Garcia require underlying databases to support atomic (linearizable [30]) single-record operation and durability, which are naturally achieved in single-instance relational databases and supported by most productized NoSQL databases (e.g., Cassandra, HBase, MongoDB, Redis, DynamoDB, and Cosmos DB). Therefore, the approach could support various types of databases. (2) The approach is non-invasive, making it possible to add support for new databases without modifying them at all.

There are also some disadvantages: (1) There are no existing real-world products; thus, there would not be enough lessons learned to productize the approach for real-world applications. Although several research prototypes using this approach exist, they provide a limited isolation guarantee. For example, the Cherry Garcia protocol depends on a reliable clock (e.g., TrueTime [10]) and guarantees only Snapshot Isolation [3]. Deuteronomy guarantees serializable isolation but only in a limited way. This is because having multiple transaction managers to achieve better performance requires a dataset to be partitioned disjointly for each transaction manager; i.e., each transaction manager cannot span multiple partitions in such a case. (2) It is not straightforward to optimize transaction performance globally because the approach achieves global transactions on the database abstraction. Particularly, the detailed information (e.g., data location, data layout) of underlying databases is abstracted away, so the transaction protocol cannot utilize such information to optimize performance. (3) Local transactions always need to pass the coordinator even though a target database is transactional because the coordinator achieves global transactions.

Our approach with ScalarDB. After a careful study of the different design choices, we decided to go with the SingleTM approach and try to address the disadvantages of the approach as much as possible. Specifically, we chose the Cherry Garcia protocol as one of the SingleTM approaches and extended the protocol.

The main reason for this design choice is that the requirements for the MultiTM approach are too strict for underlying databases. Because of this, it is essentially difficult to achieve a database-agnostic property, which is our key design goal. Moreover, although the SingleTM approach has several disadvantages, we believe we could develop solutions to mitigate them.

2.3 Challenges

Our challenge with ScalarDB is achieving the aforementioned design goals by using the SingleTM approach, which achieves an excellent database-agnostic property. Therefore, we address the following specific challenges to meet the rest of the design goals:

- Provide a strict serializability guarantee without depending on a reliable clock or assuming how data is partitioned.
- Enhance transaction performance without sacrificing correctness.
- Achieve high scalability and high availability while providing strong correctness.

Moreover, we clarify and fill the critical missing pieces of the SingleTM approach for productization to make ScalarDB production ready for real-world applications and use cases. For example, the research work based on the SingleTM approach has neither explored nor devised a clear and correct way to take transactionally consistent backups over multiple databases. For the limitation of local transactions, we leave it as an open challenge.

3 SCALARDB OVERVIEW

This section describes an overview of ScalarDB. It first describes the architecture and then the basic transaction protocol, which employs the Cherry Garcia protocol [12]. ScalarDB is written in Java, and its core components are open-sourced under the Apache 2.0 License [51].

3.1 Architecture

Figure 2 shows the architecture of ScalarDB. The core components of ScalarDB consist of four layers: a user interfaces layer, transaction manager layer, database abstraction layer, and shim layer. The user interfaces layer defines several interfaces for users. The transaction manager layer realizes global transactions in a database-agnostic way using the database abstraction layer. The database abstraction layer abstracts underlying databases with a data model

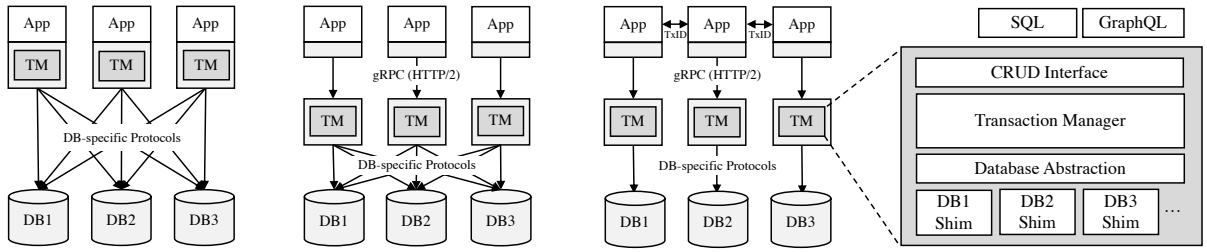


Figure 2: ScalarDB architecture.

and interfaces. The shim layer implements database-specific shims based on the abstraction. We describe more details in each layer later in this section.

ScalarDB is provided in two ways, as described in Figure 2. One way is called client mode (Figure 2(a)). Client mode provides a client library, which holds the core components. The client library is usually integrated with application servers, where business logic is defined. The other way is called server mode (Figure 2(b)(c)). Server mode provides a server component called ScalarDB server and a client library for the server. In this case, the server holds the core components. ScalarDB servers are implemented with gRPC [26] so that many programming languages can potentially interact with ScalarDB servers. One important thing to note is that although client mode is the architecture that the original Cherry Garcia employs, we use only server mode in production since it is essentially difficult to implement some critical mechanisms for productization in client mode. We discuss more details about the mechanisms for productization in Section 6.

Database Abstraction and Shim. The database abstraction layer abstracts an underlying database as a multi-dimensional map, an extended key-value model similar to the Bigtable [8] data model. In the abstraction, a record (i.e., ScalarDB record) comprises partition keys, clustering keys, and a set of columns. The combination of partition and clustering keys form a primary key, and a primary key uniquely maps a record. Records with the same partition keys form a partition and are assumed to be sorted by clustering keys. In addition, the partitions are assumed to be distributed by hashing. We chose the abstraction to achieve broad applicability for various databases.[3] ScalarDB manages one multi-dimensional map as a *table* and a set of tables with a *namespace*.

The database abstraction requires each underlying database to provide at least the following capabilities:

- Linearizable read and conditional mutations (write and delete) on a single database record.
- Durability of written database records.
- Ability to store arbitrary data in addition to application data in each database record.

[3]If we chose a more expressive data model for the ScalarDB abstraction, we could not properly implement a shim for a database based on a simpler data model due to the capability gap. Similarly, if we chose range partitioning as the abstraction for data distribution, we could not practically implement a shim for a database based on hash-partitioning because the database cannot efficiently support range queries.

The requirements are the same as the ones of the Cherry Garcia protocol [12] because the transaction protocol of ScalarDB extends it. Although the requirements may seem strong, most databases meet them. ScalarDB currently provides shims for major relational database systems such as MySQL, MariaDB, PostgreSQL, Oracle Database, Microsoft SQL Server, and SQLite and NoSQL databases such as Apache Cassandra, Amazon DynamoDB, and Azure Cosmos DB. We implemented these shims by using database-specific libraries or query languages. We plan to create new shims for a wider range of data management systems such as MongoDB and Redis.

Transaction Manager. The transaction manager layer coordinates global transactions by using the database abstraction. Since ScalarDB takes the SingleTM approach as described in Section 2.2, ScalarDB does not depend on the concurrency control capability of underlying databases. Therefore, it achieves transactions outside of databases. We will describe more details about the protocol in Section 3.2 and how we have extended the protocol to address some of the aforementioned challenges of the SingleTM approach in Section 4 and 5. Note that ScalarDB transactions do not support constraints such as referential integrity, but we plan to implement them in future work.

User Interfaces. The user interfaces layer provides a CRUD interface to users; applications create and update a record with a *put* operation, read a record with a *get* operation, read a partition with a *scan* operation,[4] and delete a record with a *delete* operation. ScalarDB employs an interactive transaction model, where users issue an arbitrary number of operations interactively between *begin* and *commit* commands.

The default interface provides a single COMMIT method, which internally executes several methods, such as PREPARERECORD and COMMITSTATE, as described in Algorithm 1. ScalarDB also provides a fine-grained interface (Figure 2(c)) that makes users explicitly call such internal methods. With the proper use of the fine-grained interface, users can run a transaction that makes multiple transaction managers interact through applications, which is useful for some use cases, such as microservice-oriented applications with the database-per-service pattern [7, 38].

[4]For the currently supported shims, ScalarDB realizes *scan* by exploiting underlying databases' partitions or transactions that achieve multi-record linearizable read, but ScalarDB could realize a partition with a single database record by constructing multiple ScalarDB records in the database record. ScalarDB also provides *scan* that scans the whole database for testing purposes, but the operation does not provide the same isolation guarantee as the other operations.

ScalarDB also provides a SQL interface and a GraphQL interface, built on top of the CRUD interface, for those familiar with such query interfaces. The current SQL interface is incompatible with the SQL standards (e.g., SQL-92) and does not offer join and aggregation queries, but supporting those queries is our future work.

3.2 Transaction Protocol Overview

Two-Phase Commit over Records. The basic principle of running global transactions in ScalarDB is treating each record of each database as a small separate database and doing two-phase commits over multiple records. Since we assume databases are abstracted in the same way and provide the required capabilities described in Section 3.1, there is no distinction between a record in one database and a record in another, different kind of database. Thus, the transaction protocol can execute transactions in the same way without regard for the differences between multiple databases.

Algorithm 1 shows the two-phase commit protocol of ScalarDB. Assuming we are using server mode; a client accesses a ScalarDB server, and the ScalarDB server, as a transaction coordinator, accesses the underlying databases. When a client begins a transaction, it first generates a transaction ID (TxID). We create TxIDs with UUID version 4 since they only need to be unique and do not have to be globally ordered IDs or timestamps. Then, when the client is ready to commit the transaction after performing operations such as *get* and *put* for reading and writing records, it calls COMMIT (line 1) to request a ScalarDB server to commit the transaction. Note that ScalarDB uses a single-version optimistic concurrency control; thus, the ScalarDB server holds the read set (*readSet*) and write set (*writeSet*) of the transaction in its local memory space at the time of committing.[5] The ScalarDB server first prepares the records of the write set by propagating the records with PREPARED states to the underlying databases (lines 3-8). Here we assume a write set maintains updated records composed of the original records and updated columns. ScalarDB checks conflicting preparations by using linearizable conditional writes; a transaction updates a record if the record has not been updated by another transaction since the transaction read it by checking if the TxID of the record has not been changed. If any preparation fails, it aborts the transaction by writing an ABORTED state record to a coordinator table (line 6), where all the transactions' final states are determined and managed. We explain the coordinator table in more detail later in this section. Otherwise, it commits the transaction by writing a COMMITTED state record to the coordinator table (line 19). Note that writing to the coordinator table is also done using linearizable conditional writes to coordinate concurrent writes; creating a state record with a TxID if there is no record for the TxID. Once the COMMITTED state is properly written to the coordinator table, the transaction is regarded as committed. Then, the ScalarDB server (asynchronously) commits all the prepared records by changing the states of the records to COMMITTED (lines 21-23). Note that we skipped the explanation of a validate-record phase (lines 10-17), which is for making transactions strict serializable. We discuss the details of this in Section 4.

[5]ScalarDB also manages a scan set and a delete set for each transaction, but they are omitted from the algorithm for simplicity.

Algorithm 1 Two-phase commit over records

```
1:  function COMMIT(TxID, readSet, writeSet)
2:      // Prepare-record phase
3:      for all (key, updatedRecord) ← writeSet do
4:          PREPARERECORD(key, updatedRecord)
5:          if PREPARERECORD fails then
6:              ABORTSTATE(TxID)                ▷ Rollback and return
7:          end if
8:      end for
9:      // Validate-record phase
10:     if Extra-read is enabled then           ▷ For strict serializability
11:         for all (key, record) ← readSet do
12:             VALIDATERECORD(key, record)     ▷ Re-read records
13:             if VALIDATERECORD fails then
14:                 ABORTSTATE(TxID)            ▷ Rollback and return
15:             end if
16:         end for
17:     end if
18:     // Commit-state phase
19:     COMMITSTATE(TxID)                       ▷ Regarded as committed here
20:     // Commit-record phase
21:     for all (key, −) ← writeSet do
22:         COMMITRECORD(key)
23:     end for
24: end function
```

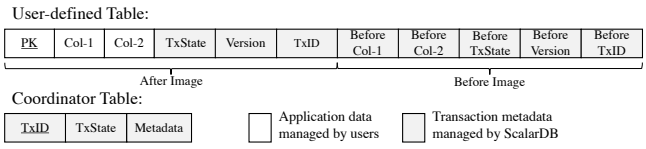


Figure 3: Example of ScalarDB schema.

Disaggregated WAL (DWAL). To make each record work as a database, ScalarDB manages write-ahead logging (WAL) information separately for each record, which we call disaggregated WAL (DWAL). Specifically, as shown in Figure 3, ScalarDB adds transaction metadata to a record in addition to the columns that an application manages. The transaction metadata comprises, for example, an ID (TxID) of a transaction that has updated the corresponding record most recently, a record version number (Version), a record state (TxState) (e.g., COMMITTED or PREPARED), timestamps (not shown in the diagram), and a before image that comprises the previous version's application data and its metadata.

ScalarDB also manages transaction states separately from the application records in the coordinator table. The coordinator table determines and manages transaction states as a single source of truth. The coordinator table can be collocated with application-managed tables or located in a separate dedicated database.

One important thing to note is that the coordinator table can be replicated for high availability by using the replication and consensus capabilities of underlying databases. For example, if we manage the coordinator table using Cassandra with replication factor three, we can make the transaction coordination of ScalarDB tolerate one

replica crash. Hence, we can make the two-phase commit protocol of ScalarDB perform like Paxos Commit protocol [23]; it could mitigate the liveness issues (e.g., blocking problems) without sacrificing safety.

Lazy Recovery. Transactions can crash at any time and could leave records uncommitted states. ScalarDB recovers uncommitted records lazily when it reads them, depending on the transaction states of the coordinator table. Specifically, if a record is in the PREPARED state, but the transaction that updated the record is expired or aborted, the record will be rolled back. If a record is in the PREPARED state and the transaction that updated the record is committed, the record will be rolled forward.

A transaction expires after an expiration time (15 seconds by default). When ScalarDB observes a record that has been prepared by an expired transaction, ScalarDB writes the ABORTED state for the transaction to the coordinator table (with retries). If ScalarDB successfully writes the ABORTED state to the coordinator table, the transaction is aborted. Otherwise, the transaction will be committed by the original process that is slow but still alive for some reason or remains in the UNKNOWN state until it is either aborted or committed.

4 GUARANTEEING STRONG CORRECTNESS

This section explains how ScalarDB extends the concurrency control protocol described in Section 3.2 to guarantee strict serializability without depending on reliable clocks or sacrificing the high scalability and high availability properties of the protocol.

4.1 Removing Reliable Clock Dependency

The original Cherry Garcia protocol provides a snapshot isolation (SI) [3] guarantee based on TrueTime [10] or other reliable clock alternatives, such as a timestamp oracle. However, we decided to make ScalarDB not depend on such reliable clocks for the following reasons. First, TrueTime is not always available; thus, depending on it in a concurrency control protocol could limit the applicability of the protocol. Second, a timestamp oracle could be a single point of failure; thus, it could limit the availability of a system. Managing multiple servers using replication could mitigate the issue, but managing strictly-increasing timestamps while providing high availability is hard to scale, especially in a multi-datacenter environment, and could limit the performance of a system. This design choice leads us to modify the protocol to depend only on linearizable operations to validate conflicting transactions. The modified protocol uses a single-version optimistic concurrency control instead of the original protocol's two-version concurrency control because, without reliable clocks, a transaction cannot guarantee to identify if the previous version (the before image) of a record has been committed before the transaction starts. Therefore, ScalarDB guarantees weaker isolation than SI, which we call read-committed snapshot isolation (RCSI). RCSI could cause a read-skew anomaly in addition to the anomalies caused by SI, such as write-skew and read-only anomalies. We offer RCSI for applications that are sufficient with it.

ScalarDB could use a hybrid-logical clock (HLC) [37] to provide an SI guarantee. HLC provides timestamps that are a combination of a physical clock based on a node's coarsely synchronized system clock and Lamport's logical clock [39]. However, ScalarDB with HLC would introduce an additional database write for each record read to keep track of *happened-before* relation since there is no message passing between transaction managers. This approach would make ScalarDB issue the same number of reads and writes as one of the serializable techniques (an extra-write strategy) we describe in Section 4.2. Therefore, we decided not to employ HLC in ScalarDB.

4.2 Making Transactions Strict Serializable

One of the common approaches for making SI-based concurrency control serializable is the serializable SI (SSI) concurrency control protocol [6], which keeps track of two consecutive anti-dependency (read-write dependency) edges in the serialization graph, the root cause of non-serializable transactions in SI-based concurrency control [17]. However, we realized that achieving SSI efficiently with the ScalarDB approach (i.e., SingleTM approach) is not straightforward. Specifically, ScalarDB coordinates transactions outside of databases, and the ScalarDB components (i.e., ScalarDB servers) do not communicate directly to share information; they need to communicate through database records. Thus, ScalarDB would need to use database records to realize a corresponding data structure to the shared lock table, which SSI expects a database to have. Implementing such a data structure with ScalarDB would cause the number of database reads and writes to be more than tripled; each record read requires at least additional database write and read (to take a SIREAD lock and check if there is a WRITE lock), and each record write requires at least additional database write and read (to take a WRITE lock and check if there is a SIREAD lock). Alternatively, we could use the serial safety net (SSN) [67] to make SI-based concurrency control serializable. Although SSN is more efficient than SSI from a concurrency perspective, it essentially requires similar anti-dependency tracking to SSI; implementing SSN with ScalarDB also causes many reads and writes for databases.

As a current design choice, we decided to take a different approach in ScalarDB. ScalarDB offers two strategies to achieve strict serializable transactions: *extra-write* and *extra-read*, which avoid anti-dependencies without explicitly tracking them. Our approach requires fewer numbers of database reads and writes than SSI and SSN, but it is more conservative than those; i.e., it could cause more aborts due to false positives.

Extra-Write Strategy. The extra-write strategy converts reads into writes to avoid anti-dependencies.[6] Thus, the strategy basically writes the records of a read set (with updated TxIDs and version numbers) to a database in addition to writing the records of a write set to the database. It works as above when handling existing records but requires extra care when handling non-existing records and scanning a partition.

When reading a non-existing record, there is no way to utilize the non-existing record to detect conflicts. This situation causes anomalies like write skew. For example, T1 reads record A and writes record B based on the read value (e.g., T1 writes A+1 to B, where A is 0 if empty) and T2 reads record B and writes record A based on the read value (e.g., T2 writes B+1 to A, where B is

[6]The thesis [11] that describes the Cherry Garcia protocol in details briefly touches the approach; however, it does not provide enough information to implement it properly.

0 if empty). In this case, if there are no records and both transactions come to the system simultaneously, both can go through successfully, which causes a non-serializable result (i.e., A=1, B=1). To avoid the issue, the extra-write strategy converts a read for a non-existing record into a write of a new record with a DELETED state in the prepare-record phase. Then, the record is deleted in the commit-record phase by utilizing the roll-forward mechanism described in Section 3.2. With the above way, reading a non-existing record and creating the record by a different transaction cause a write-write dependency, which will be handled properly even with the RCSI-based concurrency control protocol.

A similar issue would happen when scanning a partition because there could be conflicting writes in the scan range. Although ScalarDB currently throws an error if there are a scan and another operation in a transaction, ScalarDB could lock a partition by introducing a simple lock table to avoid the issue, which we leave as future work.

Extra-Read Strategy. The extra-read strategy implicitly tracks anti-dependencies by re-reading a read set before the commit-state phase. This strategy is inspired by the method applied in recent in-memory database engines [13, 65]. Specifically, the extra-read strategy adds one more phase called a validate-record phase to the protocol and executes the phase after the prepare-record phase, as shown in Algorithm 1. The validate-record phase takes the read and scan sets of a transaction and re-reads all the records in the read and scan sets to see if other transactions have written the records that the transaction has read before.[7] If the read and scan sets have not been changed, the transaction can go to the commit-state phase since there are no anti-dependencies; otherwise, it aborts the transaction. Another conflicting transaction (T2) could come after the validate-record phase and before the commit-state phase of the first transaction (T1) and could write the record that T1 has read, but T2 cannot be dependent on the PREPARED records of T1. Therefore, the resulting schedule will be equivalent to T1 → T2 even in such a case.

We could also informally discuss its serializability by reduction to strict two-phase locking (S2PL). Specifically, writing records in the prepare-record phase by using a write set is regarded as taking write locks, and re-reading records in the validate-record phase to check if the records have not been changed is equivalent to taking read locks. Moreover, all the writes and reads are conducted with linearizable operations; thus, the protocol achieves *strict* serializability.

Note that the extra-read is not starvation-free. Thus, conflicting transactions that do not share the same keys in their write sets could all enter the prepare-record phase simultaneously, but they could all be aborted in the validate-record phase. Instead, the extra-write still follows the *first-writer-wins* rule; thus, either transaction wins in such a case.

ScalarDB uses the extra-read strategy as the default serializable strategy because it works well in most workloads based on our experiments. However, the extra-write strategy can be effective for read-heavy workloads since the re-reading cost of the extra-read strategy becomes high in such a case.

[7]Note that the scan set holds predicates, which are used for re-reading of the scan set.

4.3 Correctness Verification

The correctness of transaction protocol has been empirically verified with Elle [35, 36], a transaction anomaly checker based on Jepsen [33]. Elle can detect every anomaly in Adya et al.'s formalism [1] (except for predicates), discriminate between them, and provide concise explanations of each. ScalarDB uses two consistency models to verify the correctness: *cursor-stability* for RCSI and *strict-serializable* for strict serializable isolation. The tests can be found in our GitHub repository [50]. We have found several bugs in ScalarDB with Elle, but ScalarDB currently passes the Elle tests stably and has done so for a few years.

Since Elle and existing verification tools cannot detect anomalies around predicate-based operations (e.g., scan in ScalarDB), we also created in-house verification tests to cover the case. A sensor test is one of the tests and checks if predicate-based write skew [17] would not happen in ScalarDB with strict serializable isolation. In the test, a client issues a transaction that first scans a partition specified with a current timestamp rounded by seconds. The partition can be empty or composed of multiple records, which respectively have a revision number. Then, the client identifies the maximum revision number (r_{max}) from the partition and creates a new record with a revision number $r_{max} + 1$ in the partition. So, the sensor test does not expect duplicate revision numbers in each partition as long as a transaction manager that conducts the test guarantees serializability. The test is available in our GitHub repository [53].

5 PERFORMANCE OPTIMIZATION

Although the SingleTM approach has difficulties optimizing transaction performance as discussed in Section 2.2, ScalarDB employs several performance optimizations to increase intra- and inter-transaction parallelism as much as possible. This section introduces some of the performance optimizations derived from our extensive and intensive benchmark experiments.

5.1 Parallel Commit and One-Phase Commit

As Cherry Garcia, ScalarDB employs parallel commit and one-phase commit optimizations. With parallel commit, ScalarDB commits records in parallel and asynchronously in the commit-record phase. With one-phase commit, ScalarDB omits the prepare-record and commit-state phases without sacrificing correctness if a transaction updates only one record by exploiting the single-record linearizable operations of the underlying databases.

5.2 Parallel Preparation and Validation

ScalarDB employs more aggressive parallel execution than Cherry Garcia, executing the prepare-record and validate-record phases respectively in parallel to further increase intra-transaction parallelism without violating correctness.

Parallel preparation brings some performance trade-offs. Sequential preparation prepares records in a deterministic order, following the *first-writer-wins* rule. Thus, one transaction wins even among conflicting transactions, that is, sequential preparation provides starvation-free property. However, parallel preparation does not guarantee the property; all conflicting transactions might fail. Although parallel preparation has the downside, throughout our experiments, we concluded that the benefit outweighs the downside.

5.3 Commit Spin-Waiting

ScalarDB also employs an optimization to execute more transactions concurrently by refining the lazy read recovery mechanism, which could cause unnecessary aborts in the case of contended workloads like TPC-C. For example, suppose there is a transaction T1 that prepares several records. After that, T2 tries to read some of the records that T1 prepared and finds out the records are not in the COMMITTED states. In this situation, T2 cannot recover the uncommitted records since T1 is probably not expired yet; thus, T2 cannot proceed with its processing. Consequently, ScalarDB aborts T2, throwing away whatever work T2 has done and retrying the same transaction. Aborting a transaction causes a lot of waste of work and resources, especially when a transaction is large.

To avoid such an issue, ScalarDB employs commit spin-waiting. Specifically, when ScalarDB sees an uncommitted record that has been updated recently enough, ScalarDB assumes that the transaction that updated the record will be committed soon and spin-waits with some sleep until the transaction becomes committed.

ScalarDB further optimizes the spin-waiting mechanism. When a transaction T1 spin-waits until other transactions (dependent transactions) are committed, it only waits until the coordinator table's states of the dependent transactions are committed instead of waiting until all the records that T1 uses are committed. Specifically, once ScalarDB identifies the coordinator table's states of the dependent transactions as committed, it makes T1 proceed by treating the records that T1 uses as COMMITTED states.

This optimization is not a speculative execution because PREPARED state records of committed transactions will be eventually committed; thus, it is always safe. Although it adds additional reads for checking the coordinator table, it achieves more concurrency.

6 PRODUCTIZATION

This section describes several critical mechanisms of ScalarDB for productization, especially those that make ScalarDB cloud-native and practically usable in real-world applications.

6.1 Taking Transactionally Consistent Backups

As discussed in Section 3.2, ScalarDB employs DWAL to achieve global transactions. However, unlike the conventional sequential WAL method, DWAL does not preserve the order of transactions once they are committed, which makes it challenging to take backups from multiple databases in a transactionally-consistent manner. In short, if we take backups from multiple databases separately, the resulting backups cannot guarantee to produce serializable states when restored.

To overcome the problem, ScalarDB servers provide a quiescing mechanism to take transactionally consistent backups over multiple databases. The quiescing mechanism makes a set of ScalarDB servers start queueing incoming transactions while draining in-flight transactions to create a state where there are no active transactions for a very short period. Then, we can create transactionally consistent backups from databases by using database-specific backup mechanisms such as point-in-time snapshots and restore.

There is another challenge for taking backups correctly. In production environments, users usually use ScalarDB through ScalarDB server containers and manage them with the de-facto standard container orchestration system Kubernetes. Our customers (and most Kubernetes users) use managed Kubernetes services provided and managed by cloud vendors for ease of maintenance and operation. The problem is that such managed Kubernetes services do not provide a guaranteed way to fix a set of containers.[8] For example, when accessing a cluster the first time, we see A, B, and C containers for ScalarDB servers, but we could see A, B, and D right after that; in this case, container C had crashed, and container D was automatically created by using the auto-healing mechanism of Kubernetes. This situation causes critical issues for taking backups. That is because, even if we quiesce A, B, and C containers with the mechanism to create a quiesced state, container D could still run transactions, which breaks the quiesced state.

To address the challenge, ScalarDB applies an optimistic concurrency control technique to the backup mechanism. More concretely, the backup mechanism tries to quiesce a set of ScalarDB server containers assuming the containers' states will not be changed and re-checks the states at the end of the backup process to verify if the containers' states have not been changed.

A backup tool based on the mechanism works as follows. First, a client runs the backup tool for quiescing. Then, the tool identifies a set of ScalarDB server containers by accessing the master node (control plane) of a Kubernetes cluster and quiesces the containers by using the quiescing mechanism. Once all the containers are in quiesced states, the tool takes the current time, keeping that time as the quiesce start time, and then asks the client to take backups or wait for a short while (a few seconds) for point-in-time restore (PITR). Then, the tool takes the current time again, keeping the time as the quiesce end time. Before returning a successful status to a client, the tool again identifies a set of containers to see if the containers' states have not been changed. If the containers' states have been changed, the tool returns a failure status and asks the client to retry. If the containers' states have not been changed, the tool returns a successful status and the quiesced duration. Later, users can use the backups if taken or specify a time within the quiesced duration for PITR.

6.2 Context-Aware Request Forwarding

ScalarDB takes the SingleTM approach to manage transactions as described in Section 2.2; each ScalarDB server holds the read and write sets of the transactions it manages outside underlying databases. Therefore, each ScalarDB server has transaction contexts and is stateful. When multiple applications or services interact with each other multiple times to complete a transaction, which is commonly seen in microservice use cases, we need to guarantee that the multiple applications communicate while taking care of which ScalarDB servers have which read and write sets.

For example, consider a case where an application consists of two internal microservices (e.g., order service and customer service) and requires several interactions between the microservices to complete an application-level request (i.e., transaction). Each microservice manages its database and communicates with the other service to access the other service's database. In this case, within a transaction,

[8] Managed Kubernetes services in clouds do not allow users to directly write to the master node (etcd) for security reasons.

a ScalarDB server in one microservice has to always communicate with the same ScalarDB server in the other microservice since both the ScalarDB servers manage the read and write set of the transaction for their respective databases separately. Also, each microservice will likely consist of several processes (e.g., containers or Pods) for high availability. The architecture makes it tricky for each microservice to manage a mapping table that tells which ScalarDB servers to talk to because each microservice is not likely to share the mapping table between the processes for simplicity and scalability. Application developers could use streaming (e.g., the bi-directional streaming of gRPC) or set up sticky sessions to manage such communications between microservices, but it is a burden to application developers and error-prone.

To deal with transaction contexts properly, ScalarDB provides a clustering mechanism called ScalarDB Cluster. ScalarDB Cluster groups a set of ScalarDB servers as a cluster and forwards an internal request between the servers while being aware of transaction contexts. With ScalarDB Cluster, one process in each microservice can communicate with any processes of another microservice, and one process of each microservice can talk to any ScalarDB servers of the microservice.

ScalarDB Cluster uses consistent hashing [34] for managing the mapping table between transactions (TxIDs) and the locations of ScalarDB servers that manage the transactions. It manages the membership information (i.e., locations) of ScalarDB servers by itself or automatically takes the information from a Kubernetes cluster when available. As an optimization, the client (e.g., a process of a microservice) of ScalarDB Cluster can also obtain the membership information automatically from a Kubernetes cluster to directly forward a request to a target server.

6.3 Handling Read-Only Analytical Queries

Although ScalarDB initially focused on transaction management over multiple disparate databases, we have heard a strong demand from our customers for handling analytical queries over the databases managed by ScalarDB transactions. To respond to the demand, we provide a PostgreSQL-based analytical engine called ScalarDB Analytics with PostgreSQL.

ScalarDB Analytics with PostgreSQL is based on PostgreSQL foreign data wrappers (FDW), providing a unified read-committed view of underlying databases. Thus, for example, users can run analytical join queries over multiple databases without interacting with the databases separately. ScalarDB Analytics reads records from underlying databases through community-provided data wrappers or ScalarDB Data Wrapper, select committed records from before or after images by checking the states of the records with a predefined view, and executes a given query in PostgreSQL with the committed records. ScalarDB Data Wrapper is our-developed data wrapper that uses ScalarDB abstraction to access all the ScalarDB-supported databases naturally. Since community-provided data wrappers could realize aggressive optimizations, such as pushing down database-local joins and aggregations, ScalarDB Analytics uses community-provided data wrappers if possible and uses ScalarDB Data Wrapper otherwise.

We are also currently working on a Spark-based analytical engine called ScalarDB Analytics with Spark for more scalable analytical processing. ScalarDB Analytics with Spark has a similar design to ScalarDB Analytics with PostgreSQL, using community-based connectors or ScalarDB Connector to read records from databases, selecting committed records, and executing a given query in Spark.

7 EVALUATION

This section evaluates ScalarDB. We show (1) ScalarDB achieves global transactions with reasonable performance while providing a database-agnostic property, (2) ScalarDB achieves strict serializability with acceptable performance overhead, (3) the optimizations employed in ScalarDB work effectively, and (4) ScalarDB scales near-linearly as long as the underlying database is scalable.

7.1 Experimental Setup

Compared Systems. We compared ScalarDB with Atomikos [2] and Seata [54] for global transactions. We used ScalarDB version 3.7.0 with strict serializable isolation using the extra-read strategy. We also set 500 `jdbc.connection_pool.max_total` for the JDBC shim interacting with relational databases.

Atomikos is a transaction manager based on XA. Atomikos is also used as the default XA transaction manager of ShardingSphere [42]. We used the open-source version, TransactionsEssentials, which manages coordinator logs separately for each client.[9] We used version 5.0.9 and configured Atomikos with 500 `max_actives`, 20,000 `checkpoint_interval`, and 10ms `oltp_retry_interval`.

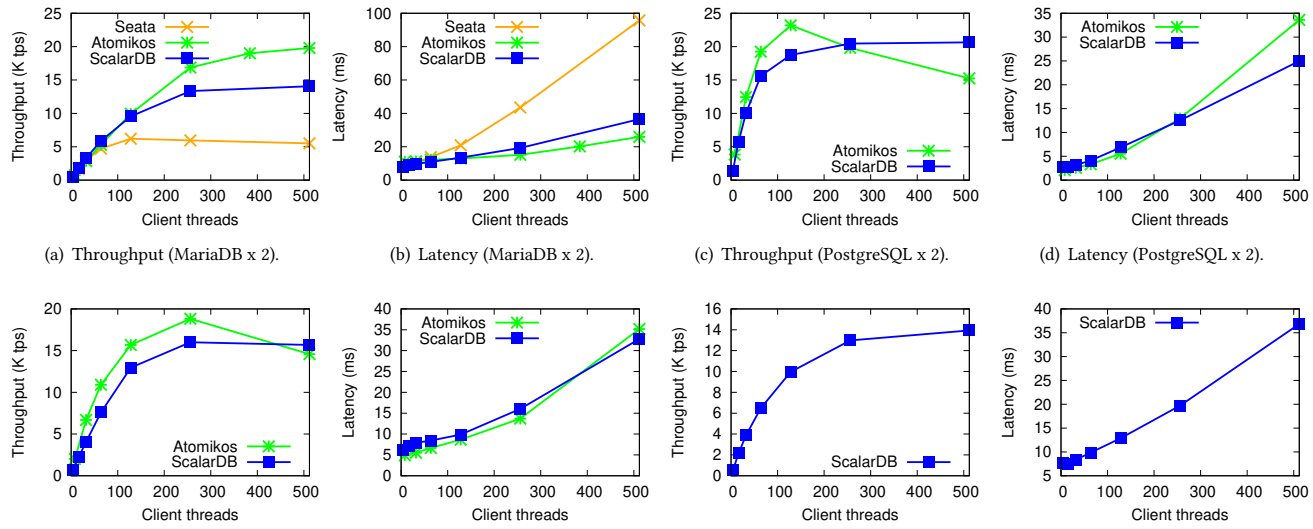
Seata is an open source distributed transaction solution that delivers high performance distributed transactions across multiple databases. We used the XA mode to run ACID transactions. We used version 1.5.2 and the default configurations since they worked best for the experiments.

Workloads. The evaluation uses two standard workloads: YCSB and TPC-C. YCSB [9] is a benchmark commonly used for key-value store evaluation and is also adopted in transactional database evaluation by accessing multiple records in a single transaction. We used Workload F (read-modify-write workload) with uniform request distribution and 1,000 bytes payload. We used YCSB to clarify the basic performance of global transactions. TPC-C [64] is a benchmark for OLTP databases. We used TPC-C to clarify the concurrency control protocol overhead for strict serializability, optimization effectiveness, and scalability of ScalarDB.

Environments. All experiments were conducted with AWS EC2 instances that run Amazon Linux 2. For global transactions experiments with YCSB, we used a c5d.9xlarge instance (18 CPU cores, 72 GB memory, 900 GB NVMe SSD) for each database or database replica and a c5d.4xlarge instance for the coordinator of Seata and ScalarDB. We also used four c5d.2xlarge instances (4 CPU cores, 16 GB memory, 200 GB NVMe SSD) for the clients of Seata and ScalarDB, and used eight c5d.2xlarge instances for the clients of Atomikos. We used more clients for Atomikos to avoid clients' IO bottleneck. For other experiments with TPC-C, we used the same settings as the YCSB experiments, except we used four c5d.2xlarge instances for the clients.

We used three types of databases, MariaDB, PostgreSQL, and Cassandra, and their versions are 10.8.3, 14.5, and 3.11.11, respectively. We configured MariaDB with 2,000 `max_connections`, 16 GB `inno`

[9] Its enterprise version manages a centralized logging service for high availability.



(a) Throughput (MariaDB x 2). (b) Latency (MariaDB x 2). (c) Throughput (PostgreSQL x 2). (d) Latency (PostgreSQL x 2).

(e) Throughput (MariaDB and PostgreSQL). (f) Latency (MariaDB and PostgreSQL). (g) Throughput (PostgreSQL and Cassandra).(h) Latency (PostgreSQL and Cassandra).

Figure 4: Performance of global transactions over two database instances with YCSB.

`db_buffer_pool_size`, 96 MB `innodb_log_file_size` (default), O_DIRECT `innodb_flush_method`, and 0 `innodb_flush_neighbors`. We set the isolation to read-committed for ScalarDB and serializable for the others. ScalarDB only requires read-committed for MariaDB because ScalarDB manages transactions outside the database. Note that we used the default `innodb_log_file_size` because MariaDB with the setting works best for all the compared systems under the hardware environment described previously. We configured PostgreSQL with 3,000 `max_connections`, 16 GB `shared_buffers`, 16 GB `max_wal_size`, and 512 `max_locks_per_transaction`. We also set the isolation to read-committed for ScalarDB and serializable for the others. We configured a Cassandra cluster with batch `commitlog_sync`, 512 `concurrent_reads`, and 512 `concurrent_writes`. The Cassandra shim for ScalarDB uses lightweight transactions (Paxos) to achieve linearizable operations.

7.2 Performance of Global Transactions

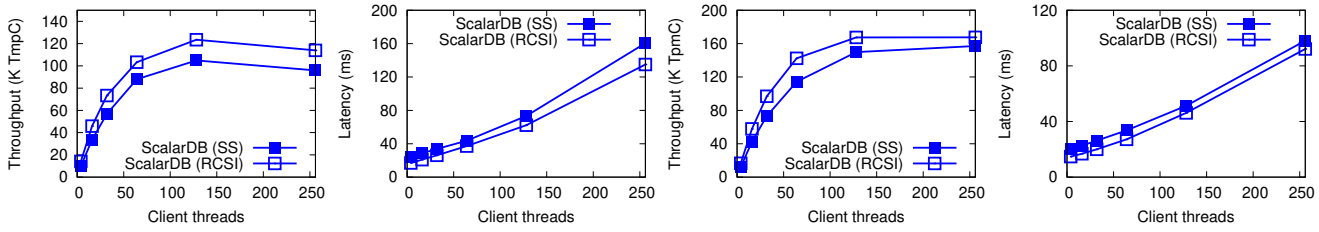
This section evaluates the throughput and latency of global transactions in ScalarDB, Seata, and Atomikos with YCSB.

First, we deployed two different instances of a single database to two different nodes to clarify the global transaction performance of each system without the performance being affected by database differences. We used two database implementations, MariaDB and PostgreSQL, for the experiments. In the experiments, each transaction goes to both databases; each transaction first goes to one database and does a read-modify-write, and then goes to the other database and does a read-modify-write. We loaded 100 million records to each database before the experiments. We also increased the number of client threads to clarify how each system handles concurrent transactions.

Figure 4(a) and 4(b) show the results in the MariaDB environment. We observed that ScalarDB was slower than Atomikos, but the differences in performance were not significant. Specifically, ScalarDB was at most 29% slower than Atomikos at 512 threads. The differences in performance mainly came from two factors. First, ScalarDB wrote more data for each transaction than Atomikos due to DWAL for each record. Second, Atomikos only wrote coordination logs to each client locally and independently, which is not appropriate for production systems, while ScalarDB wrote coordination logs to the centralized (yet scalable) coordinator table through a network. If Atomikos wrote logs to centralized storage as its enterprise version does, we would expect these differences in performance to be smaller. Seata was overall slower than Atomikos and ScalarDB. One reason was Seata sent coordination logs to a centralized Seata server. We could not clarify any other reasons, but we suspect the root cause was the Seata server because the MariaDB XA implementation worked well with Atomikos.

Figure 4(c) and 4(d) show the results in the PostgreSQL environment. Note that we could not run experiments with Seata on PostgreSQL because Seata did not support PostgreSQL and could not run transactions properly. ScalarDB was slower than Atomikos when the number of client threads was 128 or less, but the differences in performance were again not significant. However, the performance of Atomikos dropped suddenly after 128 client threads. We could not clarify the root cause, but we suspect the performance degradation was caused by the PostgreSQL XA implementation. In particular, PostgreSQL synchronously flushed WAL buffers for each XA prepare, which seemed to limit the number of transactions prepared in each flush. We actually observed that the number of fsync operations increased significantly as the number of client threads exceeded 128.

Next, we deployed two different databases to two different nodes. We tested two cases; one used MariaDB and PostgreSQL, and the other used PostgreSQL and Cassandra. We used the same workload as the previous experiments.



(a) Throughput (MariaDB). (b) Latency (MariaDB). (c) Throughput (PostgreSQL). (d) Latency (PostgreSQL).

Figure 5: Overhead of the extended concurrency control for strict serializability in ScalarDB with TPC-C.

Figure 4(e) and 4(f) show the results of the MariaDB and PostgreSQL case. We did not conduct the experiments with Seata because Seata could not run transactions on PostgreSQL, as discussed previously. As can be seen, we observed similar results to the previous results; ScalarDB was slower than Atomikos, but the differences in performance were again not very significant even though global transactions spanned different databases. Specifically, ScalarDB was at most 17% slower than Atomikos at 128 threads.

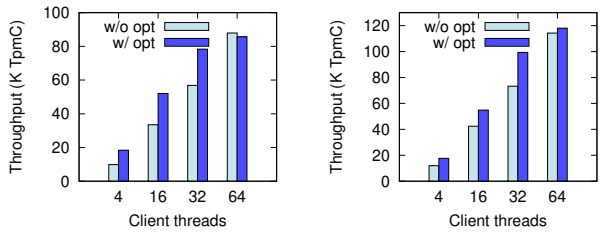
Figure 4(g) and 4(h) show the results of the PostgreSQL and Cassandra case. We used nine nodes (with replication factor three) for Cassandra. Note that we conducted the experiments only with ScalarDB because ScalarDB was the only system that could span transactions between PostgreSQL and Cassandra. As can be seen, ScalarDB again performed stably well even in this case.

Overall, ScalarDB showed superior stability and applicability compared to the XA-based systems because ScalarDB realized transactions consistently in its transaction management layer without depending on a specific XA implementation for each database. Moreover, ScalarDB achieved performance close to Atomikos in most cases. We believe the benefits of ScalarDB would outweigh the performance loss.

7.3 Overhead of Strict Serializability

This section evaluates the overhead of ScalarDB's approaches for strict serializability. We compared ScalarDB transactions with RCSI and strict serializable isolation (with the extra-read strategy) using MariaDB and PostgreSQL as underlying databases. Note that although ScalarDB is specifically designed for running distributed transactions across multiple disparate databases, we used a single database instance to clarify the overhead of our approach. We used TPC-C workload and 1,000 warehouses for the evaluation.

Figure 5(a) and 5(b) show the throughput and latency in the MariaDB environment. ScalarDB with strict serializable (SS) isolation performed a little slower than ScalarDB with RCSI, but the differences in performance were small; at most 15% slowdown in throughput and 17% slowdown in latency at 128 client threads. Figure 5(c) and 5(d) show the throughput and latency in the PostgreSQL environment, and we also observed similar (but even smaller) differences in performance to the ones of the MariaDB experiments; at most 11% slowdown in throughput and 11% slowdown in latency at 128 client threads. Throughout the experiments, we concluded that our approach to making transactions strict serializable is practical enough.



(a) MariaDB. (b) PostgreSQL.

Figure 6: Effectiveness of parallel commit optimization.

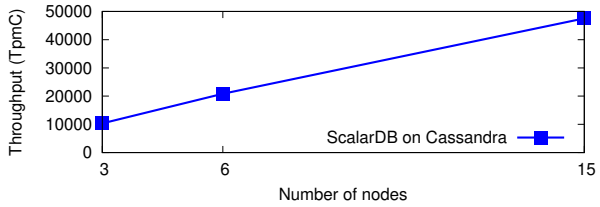


Figure 7: Scalability of ScalarDB with TPC-C.

7.4 Effectiveness of Optimizations

This section evaluates the effectiveness of the optimizations. Due to space limitation, we only show the effectiveness of the parallel commit optimization described in Section 5.1. Figure 6 shows the results in the TPC-C workload. It shows that the performance improvements were greater when the number of client threads was in the middle range (around 4 to 64). This is because there was more parallelism to exploit when the number of client threads was not high enough. But, when the number of client threads was high (e.g., more than 64 client threads), the overall parallelism was high enough, so increasing the intra-transaction parallelism did not improve performance. We observed that the optimization improved the performance by up to 87% in MariaDB and 48% in PostgreSQL.

7.5 Scalability

This section evaluates the scalability of ScalarDB with TPC-C workload. We used Cassandra as the underlying database to clarify the scalability of ScalarDB transaction protocol. We increased the number of Cassandra nodes from 3 to 15 while keeping the number

of replicas to three. We also increased the number of warehouses proportionally from 200 to 1,000.

Figure 7 shows the results. The throughput of ScalarDB increased near-linearly as the number of Cassandra nodes increased. Specifically, the throughput of ScalarDB in a 15-node environment was 4.6 times higher than the one in a 3-node environment; thus, it achieved 92% scalability compared with the ideal performance.

8 CASE STUDIES

ScalarDB has been used by several customers. In this section, we outline two specific case studies of ScalarDB usage.

Video metadata management system. A Japan-based broadcasting company wanted to fix an inconsistent metadata issue and reduce the cost of its video metadata management system. The inconsistency was caused by separate data management between a relational database and operating system files. The company migrated data into a new system based on ScalarDB and put the overview of video metadata on an Amazon RDS (Aurora) and detailed information on an Amazon DynamoDB. The company chose the hybrid database architecture to put large and infrequently-accessed data into the auto-scalable DynamoDB while serving frequent querying on the RDS. The company successfully fixed the inconsistency issue by using ScalarDB universal transaction manager and reduced costs by scaling down DynamoDB in off-peak times.

ScalarDL. ScalarDL [52, 68] is Byzantine fault detection (tamper detection) middleware for transactional database systems. ScalarDL utilizes ScalarDB to achieve database-agnostic and cloud-agnostic properties, providing its Byzantine fault detection capability on various databases. ScalarDL has been used by several customers on several platforms, ranging from relational databases to NoSQL databases, such as Amazon DynamoDB and Azure Cosmos DB.

9 LESSONS LEARNED AND FUTURE WORK

While ScalarDB is still at an early stage of product adoption, we have learned some important lessons from the experiences of building ScalarDB over the last five years and our production experiences. In this section, we share some of the lessons learned. We also share some future work to improve the product further.

Single Abstraction. The database abstraction of ScalarDB is one of the keys to achieving database-agnostic transactions. At the same time, it limits the capabilities of underlying databases. For example, suppose there is a MySQL (InnoDB) table, which is a clustered index and can do a prefix search with a primary key. In this case, ScalarDB on MySQL cannot fully utilize the prefix search capability of MySQL since the ScalarDB abstraction supports only partition-level prefix search; a partition key must always be provided. To mitigate such issues, we plan to introduce other abstractions, such as relational abstraction, which abstracts the relational data model and the capabilities of relational databases. We also need to explore how we can extend the transaction manager to handle multiple abstractions without overcomplicating the transaction management.

Similarly, a single abstraction makes it challenging to run ScalarDB on existing databases; i.e., it is not always possible to run ScalarDB on existing databases without data rebuild due to differences in data models. If an existing database's data model and schema are compatible with the ScalarDB abstraction, ScalarDB can mostly run on the database without data rebuild. That is because most databases can instantly add new null columns for the ScalarDB metadata without rebuilding the databases, and ScalarDB can treat null state records as COMMITTED. If an existing database's data model and schema are incompatible with the ScalarDB abstraction, ScalarDB is less likely to run on the database. This limitation could also be mitigated by introducing multiple abstractions.

Backups. As discussed in Section 6.1, DWAL does not preserve the order of transactions once they are committed, which makes it hard to take backups from multiple databases in a transactionally consistent manner. ScalarDB provides the quiescing mechanism to resolve the issue, but it needs to suspend transactions for a few seconds, which may not be acceptable in future use cases. In future work, we plan to introduce a mechanism to log transactions in a serializable and replayable manner to avoid such suspension. One prospective way is sending the read and write sets of transactions to a remote location and replaying the transactions based on the dependencies derived from the given read and write sets.

Lazy Recovery. The lazy recovery discussed in Section 3.2 is a scalable approach, especially in a distributed environment. However, the lazy recovery makes it challenging to manage an append-only table where an application only does blind writes and does not read data. For example, suppose an application manages append-only logs in a table, and a transaction crashes after preparing a record in the table. In this case, the application will not read the prepared record; thus, the record will be left unrecovered. If the application always writes a log with a unique primary key, it just creates unrecovered garbage. Otherwise, the application cannot proceed with the process since the log record has already been prepared. We could fix the issue by making a blind write operation always come with a pre-read behind the scene; however, the solution badly affects the performance because it causes an additional read for each blind write. We currently require applications to deal with the issue, but we plan to fix the issue in a better way in the future.

10 CONCLUSION

This paper presented ScalarDB, a universal transaction manager that achieves a database-agnostic federation of multiple disparate databases from a transactional perspective. ScalarDB extends previous research works to provide a strong correctness guarantee (i.e., strict serializability), further performance optimizations, and several critical mechanisms for productization. Throughout the evaluations, we showed that ScalarDB achieved database-spanning transactions with reasonable performance and near-linear scalability while providing database-agnostic property and guaranteeing strong correctness.

ScalarDB is still an early-stage product. We continue to innovate with improvements and enhancements to its performance, capabilities, and usability to take the product to the next level.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We are also grateful to the Scalar Engineering team members, especially Vincent Guilpain, Mitsunori Komatsu, and Akihiro Okuno, who have greatly contributed to improving the usability and increasing the capabilities of ScalarDB.

REFERENCES

[1] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized Isolation Level Definitions. In *ICDE*. 67–78.
[2] Atomikos. 2023. Atomikos. https://www.atomikos.com/Main/WebHome.
[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. 1–10.
[4] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. 1992. Overview of Multidatabase Transaction Management. *VLDBJ* 1, 2 (1992), 181–239.
[5] M. Cafarella, D. DeWitt, V. Gadepally, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, and M. Zaharia. 2021. A Polystore Based Database Operating System (DBOS). In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. 3–24.
[6] M. Cahill, U. Röhm, and A. Fekete. 2008. Serializable Isolation for Snapshot Databases. In *SIGMOD*. 729–738.
[7] T. Cerny, M. Donahoo, and M. Trnka. 2018. Contextual Understanding of Microservice Architecture: Current and Future Directions. *SIGAPP Appl. Comput. Rev.* 17, 4 (2018), 29–45.
[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*. 205–218.
[9] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.
[10] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*. 251–264.
[11] A. Dey. 2015. *Cherry Garcia: Transactions across Heterogeneous Data Stores*. Ph.D. Dissertation. http://hdl.handle.net/2123/14212
[12] A. Dey, A. Fekete, and U. Röhm. 2015. Scalable Distributed Transactions across Heterogeneous Stores. In *ICDE*. 125–136.
[13] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*. 1243–1254.
[14] J. Duggan, A. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Rec.* 44, 2 (2015), 11–16.
[15] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. Sorenson III, S. Sosothikul, D. Terry, and A. Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *ATC*. 1037–1048.
[16] N. Faria, J. Pereira, A. Alonso, and R. Vilaça. 2021. Towards Generic Fine-Grained Transaction Isolation in Polystores. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. 29–42.
[17] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528.
[18] The Apache Software Foundation. 2023. Apache Cassandra. https://cassandra.apache.org/.
[19] The Apache Software Foundation. 2023. Apache HBase. https://hbase.apache.org/.
[20] H. Garcia-Molina and K. Salem. 1987. Sagas. In *SIGMOD*. 249–259.
[21] Google. 2023. Cloud Spanner. https://cloud.google.com/spanner.
[22] J. Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. 393–481.
[23] J. Gray and L. Lamport. 2006. Consensus on Transaction Commit. *ACM Trans. Database Syst.* 31, 1 (2006), 133–160.
[24] The Open Group. 1991. Distributed Transaction Processing: The XA Specification. https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf.
[25] The PostgreSQL Global Development Group. 2023. PostgreSQL. https://www.postgresql.org/.
[26] gRPC Authors. 2023. gRPC. https://grpc.io/.
[27] B. Hagler. 2020. Overcoming Data Silos In Your Organization. https://www.forbes.com/sites/forbestechcouncil/2020/05/12/overcoming-data-silos-in-your-organization.
[28] D. Heimbigner and D. McLeod. 1985. A Federated Architecture for Information Management. *ACM Trans. Inf. Syst.* 3, 3 (1985), 253–278.
[29] P. Helland. 2016. Life Beyond Distributed Transactions: An Apostate's Opinion. *Queue* 14, 5 (2016), 69–98.
[30] M. Herlihy and J. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
[31] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. 2020. TiDB: A Raft-Based HTAP Database. *PVLDB* 13, 12 (2020), 3072–3084.
[32] M. Fowler J. Lewis. 2014. Microservices. https://martinfowler.com/articles/microservices.html.
[33] Jepsen. 2023. Jepsen. https://github.com/jepsen-io/jepsen.
[34] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*. 654–663.
[35] K. Kingsbury and P. Alvaro. 2021. Elle: Inferring Isolation Anomalies from Experimental Observations. *PVLDB* 14, 3 (2021), 268–280.
[36] K. Kingsbury and P. Alvaro. 2023. Elle. https://github.com/jepsen-io/elle.
[37] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems*. 17–32.
[38] R. Laigner, Y. Zhou, M. Salles, Y. Liu, and M. Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *PVLDB* 14, 13 (2021), 3348–3361.
[39] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.
[40] J. Levandoski, D. Lomet, M. Mokbel, and K Zhao. 2011. Deuteronomy: Transaction Support for Cloud Data. In *CIDR*.
[41] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. 2015. Multi-Version Range Concurrency Control in Deuteronomy. *PVLDB* 8, 13 (2015), 2146–2157.
[42] R. Li, L. Zhang, J. Pan, J. Liu, P. Wang, N. Sun, S. Wang, C. Chen, F. Gu, and S. Guo. 2022. Apache ShardingSphere: A Holistic and Pluggable Platform for Data Sharding. In *ICDE*. 2468–2480.
[43] S. Mehrotra, R. Rastogi, A. Silberschatz, and H. Korth. 1992. A Transaction Model for Multidatabase Systems. In *ICDCS*. 56–63.
[44] Microsoft. 2023. Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/.
[45] Microsoft. 2023. SQL Server. https://www.microsoft.com/en-us/sql-server/.
[46] Oracle. 2023. MySQL. https://www.mysql.com/.
[47] Oracle. 2023. Oracle Database. https://www.oracle.com/database/.
[48] Oracle. 2023. Oracle Tuxedo. https://www.oracle.com/middleware/technologies/tuxedo.html.
[49] P. Rodgers. 2005. Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity. In *CloudComputingExpo 2005*.
[50] Scalar. 2023. Jepsen tests for ScalarDB. https://github.com/scalar-labs/scalar-jepsen/tree/master/scalardb.
[51] Scalar. 2023. ScalarDB. https://github.com/scalar-labs/scalardb.
[52] Scalar. 2023. ScalarDL. https://github.com/scalar-labs/scalardl.
[53] Scalar. 2023. Sensor test. https://github.com/scalar-labs/kelpie-test/tree/master/scalardb-test/src/main/java/kelpie/scalardb/sensor.
[54] Seata. 2023. Seata. https://seata.io/en-us/.
[55] Amazon Web Services. 2023. Amazon DynamoDB. https://aws.amazon.com/dynamodb/.
[56] Amazon Web Services. 2023. Amazon Neptune. https://aws.amazon.com/neptune/.
[57] Amazon Web Services. 2023. Amazon Quantum Ledger Database (QLDB). https://aws.amazon.com/qldb/.
[58] Amazon Web Services. 2023. AWS Cloud Databases. https://aws.amazon.com/products/databases/.
[59] A. Sheth and J. Larson. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.* 22, 3 (1990), 183–236.
[60] M. Stonebraker. 2015. The Case For Polystores. https://wp.sigmod.org/?p=1629.
[61] M. Stonebraker and U. Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*. 2–11.
[62] P. Strong. 2005. Enterprise Grid Computing: Grid Computing Holds Great Promise for the Enterprise Data Center, but Many Technical and Operational Hurdles Remain. *Queue* 3, 6 (2005), 50–59.
[63] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *SIGMOD*. 1493–1509.
[64] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C (Revision 5.11). http://www.tpc.org/tpcc/.
[65] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*. 18–32.
[66] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.
[67] T. Wang, R. Johnson, A. Fekete, and I. Pandis. 2017. Efficiently Making (Almost) Any Concurrency Control Mechanism Serializable. *VLDBJ* 26, 4 (2017), 537–562.
[68] H. Yamada and J. Nemoto. 2022. Scalar DL: Scalable and Practical Byzantine Fault Detection for Transactional Database Systems. *PVLDB* 15, 7 (2022), 1324–1336.
[69] Yugabyte. 2023. YugabyteDB. https://www.yugabyte.com/.