Apache ShardingSphere: A Holistic and Pluggable Platform for Data Sharding

Ruiyuan Li[1], Liang Zhang[2], Juan Pan[2], Junwen Liu[3,2], Peng Wang[3,2],
Nianjun Sun[2], Shanmin Wang[2], Chao Chen[1], Fuqiang Gu[1], Songtao Guo[1]
[1]*College of Computer Science, Chongqing University* [2]*SphereEx Lab* [3]*JD iCity, JD Technology, Beijing, China*
{ruiyuan.li, cschaochen, gufq, guosongtao}@cqu.edu.cn, {zhangliang, panjuan}@apache.org,
{zaiyuan49, jimolonely1234}@gmail.com, {sunnianjun, wangshanmin}@sphere-ex.com

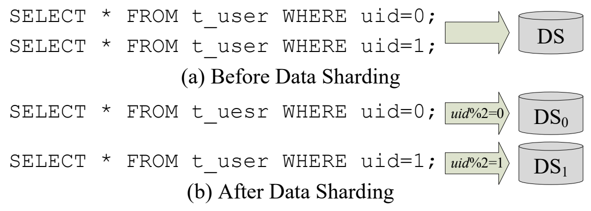
***Abstract*—Traditional relational databases are nowadays overwhelmed by the increasing data volume and concurrent access. NoSQL databases can manage large-scale data, but most of them do not support complete transactions and standard SQL languages. NewSQL is proposed for both high scalability and transactional properties with SQL languages support. One type of NewSQL builds distributed systems from scratch, which is too radical for some critical applications. The other type of NewSQL, i.e., data sharding among relational databases, is a better option for these scenarios.**

This paper presents Apache ShardingSphere, the first top-level open-source platform for data sharding in Apache, which enables developers to use sharded databases like one database. Specifically, Apache ShardingSphere integrates six databases, and designs and implements a complete SQL engine to route requests correctly and intelligently. Additionally, it encapsulates three types of distributed transactions, and provides two adaptors for different scenarios. Moreover, it proposes a novel AutoTable strategy and a query language, i.e., DistSQL, allowing database maintainers to easily configure the sharded databases. Furthermore, it provides many other pluggable features to better shard data. Extensive experiments are conducted using two famous benchmarking tools, proving that Apache ShardingSphere is more efficient than eight state-of-the-art systems in our settings. All experimental source codes are publicly released. More than 170 companies are currently using Apache ShardingSphere.

I. Introduction

Modern Internet applications require databases to not only manage massive amounts of data, but to also support highly concurrent access. For example, during the Double-Eleven Shopping Festival in 2020, JD [1] saw a transaction volume of 271.5 billion. During the same period, the order creation rate of Tmall [2] reached 583,000 orders per second. Relational databases[1] are still the main forces of OLTP (Online Transaction Processing) nowadays, because they support complete ACID (i.e., Atomicity, Consistency, Isolation, and Durability [3]) transactions and SQL languages. However, relational databases were originally designed for standalone machines. They are sometimes overwhelmed by the increasing data volume and concurrent access. Developers have to repeatedly adjust the application architectures and redesign the table structures, which is intractable and costly.

Although NoSQL (Not Only SQL) databases such as BigTable [4] and HBase [5] are designed for big data and


```
SELECT * FROM t_user WHERE uid=0;
SELECT * FROM t_user WHERE uid=1;
```
(a) Before Data Sharding
```
SELECT * FROM t_uesr WHERE uid=0;

SELECT * FROM t_user WHERE uid=1;
```
(b) After Data Sharding

Fig. 1. Example of Data Sharding.

high concurrent requests, they lack sophisticated transaction and SQL support, and are not suitable for OLTP scenarios. NewSQL [6] is proposed to achieve both ACID transactions and high scalability at the same time. One type of NewSQL systems such as TiDB [7], CockroachDB [8] and yugabyteDB [9] breaks the limitations of relational database architectures, and builds distributed databases from scratch. However, they need more verification time, and lack experienced maintainers especially for some financial systems. Migrating these applications to such databases could be very costly, as most applications are built on relational databases and cannot be offline. Another type of NewSQL systems, namely sharding middlewares, is a more modest solution based on existing relational databases. As shown in Fig. 1, sharding middlewares route the requests to multiple databases in different machines according to the query conditions, merge multiple result sets into a single one, and finally return it to applications, thus solving the bottlenecks of a single machine. They act as a proxy between applications and relational databases, and are transparent to developers. As a result, existing applications can be transplanted to sharding middlewares without any changes.

It is challenging to build a good database sharding middleware. **Firstly**, there are many kinds of relational databases with different database protocols and SQL dialects for different scenarios. Supporting all of these databases in a single framework is not easy. Moreover, there are various SQL types ranging from simple selection to aggregation to multiple tables join. Different SQL types need different route rules and result merge strategies. **Secondly**, it is intractable to coordinate distributed transactions among multiple separate databases. Sometimes a single type of transaction does not fit all cases. **Thirdly**, sharding middlewares may encounter the efficiency problem, as they take some time to forward requests and merge results. **Fourthly**, for database managers, it is difficult to configure sharding rules. They first need to create the underlying tables manually, and then configure rule strategies case by case.

Chao Chen is the corresponding author.

[1]To clarify the concepts, in this paper, a database system (such as MySQL) is called **database**, and a collection of correlated tables is called **data source**.

This paper presents Apache ShardingSphere (abbreviated as ShardingSphere in the following), the first top-level open-source database sharding system in Apache. ShardingSphere addresses all of the challenges mentioned above. The main goal of ShardingSphere is to reduce the influence of data sharding, and let users use sharded databases like one database. It has several notable characteristics:

(1) **Holistic**. It supports six mainstream relational databases and any other databases that follow the SQL-92 standard [10]. We design and implement a complete SQL engine for data sharding, thus any type of SQL can be routed effectively and intelligently. Besides, it provides three types of distributed transactions and abundant features derived from data sharding. To the best of our knowledge, ShardingSphere is the *first* system to combine data sharding, XA transaction and BASE transaction together.

(2) **Efficient**. In addition to a proxy server, Apache ShardingSphere also provides a JDBC-based adaptor, which can improve efficiency in most cases. Besides, we propose two execution modes and an intelligent strategy to select an appropriate execution mode and result merger, which balances resource control and execution efficiency.

(3) **Pluggable and Extensible**. ShardingSphere is designed based on SPI (Service Provider Interfaces) [11] (a service discovery mechanism in Java) and design patterns [12]. As a result, more types of databases, features and sharding algorithms can be added easily. Additionally, all of the existing provided features can be removed or combined freely.

(4) **User-friendly**. ShardingSphere supports almost all SQL statements of the integrated databases and hides the details of distributed transactions. As a result, application developers can use ShardingSphere and distributed transactions in the same way of using a standalone database. ShardingSphere also proposes a novel DistSQL and AutoTable strategy, which helps database managers configure sharding rules more easily.

More than 170 companies all over the world announced that they were using ShardingSphere. We conduct extensive experiments based on two widely-used benchmarking tools, verifying the powerful efficiency and scalability of ShardingSphere. ShardingSphere is under active development now, and new significant features are added frequently. This paper is based on version 5.0.0 that was released on 2021-11-10.

Outline. We present the related works in Section II. The framework and data flow of ShardingSphere are presented in Section III. Section IV to Section VII describe each module of ShardingSphere, respectively. We present the experimental results in Section VIII, and conclude this paper with future works in Section IX.

II. Related Works

Legacy Relational Databases. Relational databases (e.g., MySQL and PostgreSQL) have experienced a boom since the 1970s. They are based on relational data models, and provide a standard set of SQL interfaces, based on which many applications can be built conveniently. These relational databases provide strict ACID transactions, and require strong consistency for each operation. As most relational databases are designed for single machines, they are difficult to meet the requirements of increasing data volume and large-scale concurrent access.

NoSQL Databases. NoSQL databases relax the constraints of ACID and relational models. Most of them are designed for distributed processing, thus they can handle massive data [21–23]. These NoSQL databases are based on CAP (Consistency, Availability and Partition tolerance) Theorem [24–26], i.e., it is impossible to achieve all of these three properties for a distributed system. As the partition is inevitable, the distributed NoSQL databases can be classified into two categories: CP NoSQL databases and AP NoSQL databases. CP NoSQL databases (e.g., MongoDB [27], BigTable [4] and HBase [5]) seek strong consistency by sacrificing high availability. AP NoSQL databases, such as DynamoDB [28] and Cassandra [29], pursue high availability, and only require to be eventually consistent after an operation. Most NoSQL databases adopt key-value or document data models, and do not support standard SQL interfaces. As a result, it is costly to build applications based on NoSQL databases.

New Architecture Databases. This type of databases pursue both transactional nature of ACID and high scalability. Many new architecture databases [7–9, 30–38] are built from scratch. They adopt a shared-nothing framework, and support multi-node concurrency control and fault tolerance. For example, Spanner [30, 38] is the first system that distributes data globally and supports externally-consistent distributed transactions. TiDB [7] designs a multi-Raft storage that consists of a row store and a column store, and proposes a consensus algorithm to provide consistent replicas. CockroachDB [8] proposes a novel transaction model to support consistent geo-distributed transactions. Although these systems have shown advantages in some cases, they lack familiar maintainers and are not tested enough. Amazon Aurora [39, 40] belongs to another line of new architecture databases with a shared-disk framework. It decouples storage from compute, and pushes several functions (e.g., logging and recovery) to the storage service. To reduce the network IOs, Aurora only writes redo logs across the network. Aurora requires to migrate data to the cloud, which is not applicable to some proprietary scenarios. For those systems built on legacy relational databases that have been running for a long time, the migration cost is also very high.

Database Sharding Middlewares. Database sharding middlewares provide a more moderate solution to solve the scalability problem of relational databases. Instead of starting from scratch, they coordinate transactions and route queries to multiple database instances. The biggest advantage of sharding middlewares is that they are transparent to developers, so legacy applications do not need any changes to be transplanted to sharding middlewares. Most existing sharding middlewares [14–17, 19, 20] are only for a single type of database, and do not support distributed transactions or only support one type of distributed transaction. Besides, some of them lack useful tools (e.g., scaling) that are used frequently for data sharding. Furthermore, most of them only

TABLE I
COMPARING SHARDINGSPHERE AGAINST RELATED DATA SHARDING SYSTEMS (2021.11.01)

System	Mode	Database	Dis-Transaction	^{a}Protocol	RW-Split	Other Features	Activeness	# Star
ShardingSphere [13]	Proxy, JDBC	MySQL, MariaDB, SQL Server, Oracle, PostgreSQL, openGuass	XA, Local, BASE	MySQL, PostgreSQL	Yes	Shadow, Encrypt, Scale, Governance	Active	14.8k
MySQL Proxy [14]	Proxy	MySQL	No	MySQL	Yes	No	7 Years Ago	337
MaxScale [15]	Proxy	MySQL, MariaDB	No	MySQL	Yes	Security, Integration, Command Line	Active	1.1k
Vitess [16]	Proxy	MySQL	XA	MySQL	No	Monitor, Scale	Active	12.8k
Citus [17]	Proxy	PostgreSQL	BASE	PostgreSQL	Yes	Scale, Security	Active	5.4k
ProxySQL [18]	Proxy	MySQL, Percona Server, MariaDB, SQLLite Embedded, ClickHouse	No	MySQL	Yes	Firewall, Monitor	Active	4.6k
^{b}GoldenDB [19]	Proxy	MySQL	BASE	MySQL	Yes	Rebalance, Encrypt, Backup, SQL Audit	-	-
^{b}TDSQL-MySQL [20]	Proxy	MySQL	XA	MySQL	Yes	Scale, Monitor	-	-

^{a} Some systems claim that they support MariaDB protocol, which is similar to MySQL protocol.
^{b} GoldenDB and TDSQL-MySQL are two services in the cloud, and we cannot obtain their source codes.

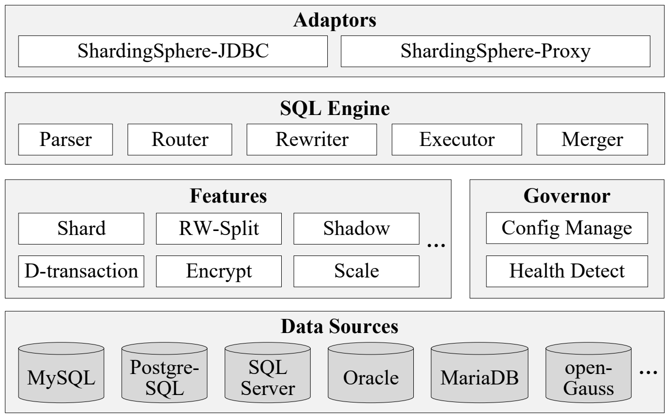


Fig. 2. Framework of Apache ShardingSphere.

provide proxy-based mode, i.e., they should be deployed as single services and forward the requests to the underlying databases, which brings in significant performance losses. As a sharding middleware, ShardingSphere supports most mainstream relational databases, and provides three types of distributed transactions for different scenarios. In addition to data sharding, it also provides a wealth of tools to help shard data better. ShardingSphere can run in two modes, i.e., proxy-based mode and JDBC-based mode, where the JDBC-based mode can achieve much better performance. Table I compares the key features of ShardingSphere with some representative sharding middlewares.

III. FRAMEWORK AND DATA FLOW

Framework. Figure 2 depicts the overall framework of ShardingSphere, which consists of five components:

(1) **Data Sources**. ShardingSphere can seamlessly integrate various databases as its underlying storages, as long as they follow SQL-92 standard [10] and JDBC (Java Database Connectivity) programming interfaces [41]. Currently, ShardingSphere has integrated MySQL, PostgreSQL, SQL Server, Oracle, MariaDB and openGauss [42].

(2) **Features**. ShardingSphere provides many out-of-the-box features, including but not limited to data sharding, distributed transaction, read-write splitting, encrypting, shadow DB and scaling. Many other useful features are added frequently to ShardingSphere. These features can be added, removed, or combined freely according to the requirements of applications (detailed in Section IV).

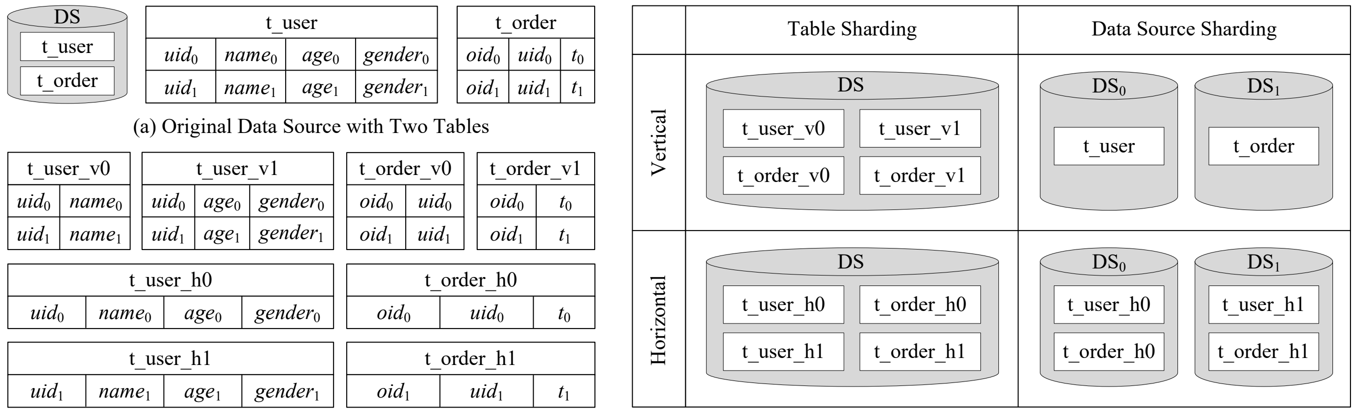
(3) **Governor**. There are two main functions of Governor: 1) *Configuration Management*, which stores and manages the metadata of data sources, the sharding rules, the configurations, and the running status of ShardingSphere cluster; 2) *Health Detection*, which monitors the whole cluster to guarantee high availability (detailed in Section V).

(4) **SQL Engine**. We design and implement a complete SQL engine for data sharding. When a SQL request arrives, it first parses the SQL statement into an abstract syntax tree, and then generates route paths according to the sharding strategies and parsing contexts. After that, the SQL engine rewrites the SQL statement into rightly executable ones in actual databases. The rewritten SQL statements are sent intelligently to the underlying data sources, and each data source executes the allocated SQL statements independently. Finally, the results acquired from multiple data sources are merged into a single one set, which is then returned to the request end. Note that all of the features are pluggable to the SQL engine, and any feature can be achieved through a single SQL statement (detailed in Section VI).

(5) **Adaptors**. To support different scenarios, ShardingSphere provides two adaptors, i.e., *ShardingSphere-JDBC* and *ShardingSphere-Proxy*, based on which many applications have been developed (detailed in Section VII).

Data Flow. Figure 4 shows the data flow of ShardingSphere. As shown in the left part, in the usage scenarios of ShardingSphere-JDBC, developers combine ShardingSphere-JDBC into the business code of Java applications. That is, ShardingSphere-JDBC and the Java application will run in the same computer process. ShardingSphere-JDBC encapsulates the SQL engine with multiple features, and interacts with Governor and multiple data sources. This interaction is transparent to the application developers.

As shown in the right part of Fig. 4, ShardingSphere-Proxy is a single process between data sources and applications. Like ShardingSphere-JDBC, ShardingSphere-Proxy incorporates the SQL engine in it, but it acts as a MySQL or PostgreSQL database. As a result, ShardingSphere-Proxy can support applications with any programming language. Besides,



(a) Original Data Source with Two Tables

(b) Data Records in the Split Tables

(c) Types of Data Sharding

Fig. 3. Illustration of Data Sharding.

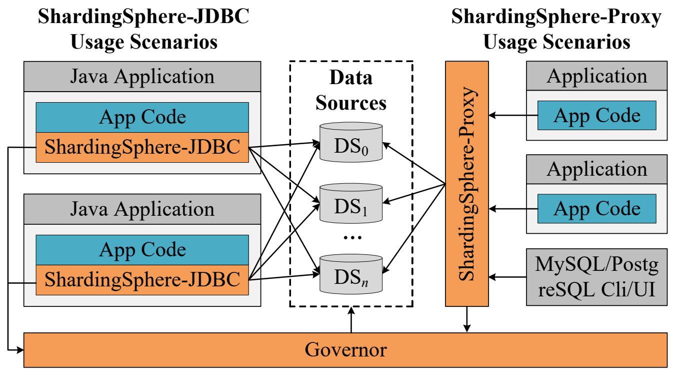


Fig. 4. Data Flow of Apache ShardingSphere.

any terminal that is compatible with MySQL or PostgreSQL protocol (such as MySQL Command Client, MySQL Workbench, etc.) can connect to ShardingSphere-Proxy directly, which is friendly to DBAs (Database Administrators).

As shown in the bottom of Fig. 4, Governor is deployed as an independent process. It monitors data sources, and receives messages from ShardingSphere-JDBC and ShardingSphere-Proxy to maintain configurations.

IV. FEATURES

In this section, we first describe the features of data sharding and distributed transactions in detail, and then introduce other features briefly due to the page limitation.

A. Data Sharding

Types of Data Sharding. Data sharding is one of the most important features in ShardingSphere. It splits the data in one data source and stores it in multiple tables or data sources according to certain criteria. ShardingSphere supports two types of data sharding: *Table Sharding* and *Data Source Sharding* (and a combination of them), each of which has two methods: *Vertical Method* and *Horizontal Method*. Suppose there is an original data source with two tables `t_user` and `t_order`, whose schemas and data records are shown in Fig. 3(a), we can perform four types of sharding on the data:

(1) **Vertical Table Sharding**. Vertical table sharding splits a table in a data source by grouping its fields. As a result, a table will be split into several tables, where the split tables have different schemas. From the view of the table, the original table is split vertically. For example, as shown in the upper-left grid of Fig. 3(c), `t_user` is split into `t_user_v0` and `t_user_v1`. The schemas and data records of the split tables are shown in Fig. 3(b). Vertical table sharding can turn wide tables into narrow tables, thus it is suitable for queries that do not retrieve too many fields.

(2) **Horizontal Table Sharding**. Horizontal table sharding does not change the schema of the original table, but it divides the data records in a table into two or more tables with the same schema. For example, as shown in the lower-left grid of Fig. 3(c), `t_user` is split into `t_user_h0` and `t_user_h1`, where `t_user_h0` and `t_user_h1` have the same schema with the original table `t_user`. Each split table only contains a subset data record of the original table `t_user`, as shown in Fig. 3(b).

(3) **Vertical Data Source Sharding**. This type of sharding keeps the schemas of tables, but assigns tables to different data sources according to business logic. For example, as shown in the upper-right grid of Fig. 3(c), `t_user` and `t_order` are assigned to two separate data sources.

(4) **Horizontal Data Source Sharding**. Horizontal data source sharding is the most complex. It is similar to horizontal table sharding that divides the data records in one table into two or more tables. However, the split tables are stored in separate data sources. As shown in the lower-right grid of Fig. 3(c), the split tables `t_user_h0` and `t_user_h1` are stored in two separate data sources.

Vertical data sharding requires to adjust the architectures and designs from time to time, because the number of data records in a table will increase constantly and exceed the threshold of a single machine. That is, vertical data sharding cannot deal with the fast-changing needs from internet business, thus it is not able to really solve the scalability problem. In contrast, horizontal data sharding can limit the maximum number of records in a single machine, and can be extended more freely, so it is regarded as a standard solution to data sharding. To this end, we focus on horizontal data sharding in the following of this paper due to the limitation of pages.

Basic Concepts of Data Sharding. In Fig. 1, suppose

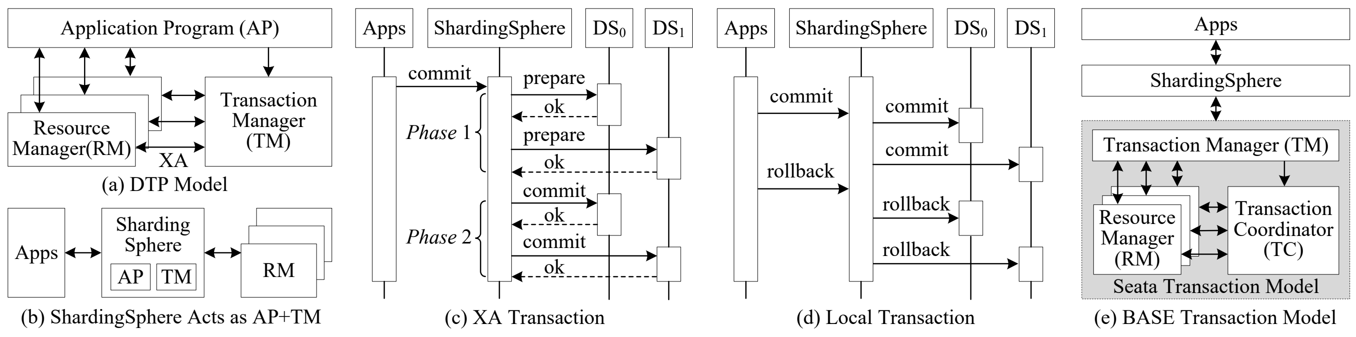


Fig. 5. Transactions in Apache ShardingSphere.

t_user is divided by the values of uid using horizontal data source sharding method, where the records with $uid\%2 = 0$ are stored in table t_user_h0 of $\mathtt{DS}_0$, and the records with $uid\%2 = 1$ are stored in table t_user_h1 of $\mathtt{DS}_1$ (as shown in Fig. 3(c)), we have the following basic concepts:

Sharding Key. The field used to determine data sharding is called sharding key, i.e., uid in this example. Besides the sharding key with a single field, ShardingSphere also supports sharding key with multiple fields.

Sharding Algorithm. It is the method that assigns data records to different tables. In our example, "$uid\%2$" (i.e., ModShardingAlgorithm) is the sharding algorithm. Currently, ShardingSphere presets 10 sharding algorithms [43]. Users can also extend their own sharding algorithms by simply implementing the interface ShardingAlgorithm. ShardingSphere would automatically load these customized sharding algorithms through SPI [11] mechanism, which makes ShardingSphere extremely extensible.

Tables. ShardingSphere provides various types of tables for different data sharding requirements, some of which related to this paper include:

• *Logic Table*. A logic table refers to the table that is identified from a SQL statement. In our example, t_user is the logic table, which can be seen by developers. The original table before data sharding is usually the logic table.

• *Actual Table*. The physical tables that really exist in the underlying databases are called actual tables. In our example, t_user_h0 and t_user_h1 are called actual tables. Actual tables are transparent to application developers.

• *Binding Table*. If two tables are divided with the same data sources, same sharding key and same sharding algorithm, they are binding tables with each other. For example, in Fig. 3(c), if the original tables t_user and t_order are both divided by "$uid\%2$", they have a binding table relationship. Binding table is very useful in multi-tables correlated queries, as will be described in Section VI.

Data Node. Data node is the atomic unit of sharding, which consists of a data source name and an actual table name, e.g., $\mathtt{DS}_0$.t_user_h1. It maps logic tables to actual tables.

B. Distributed Transaction

ShardingSphere provides three types of distributed transactions for different usage scenarios.

XA Transaction. Most modern standalone relational databases provide complete transaction support, which satisfies ACID features. However, it is non-trivial to guarantee these transaction features after data sharding, as each data node can only manage its own transactions.

X/Open Consortium (i.e., The Open Group [44]) proposed a DTP (Distributed Transaction Processing) model [45] to achieve ACID in distributed environments. As shown in Fig. 5(a), there are three roles in DTP model: 1) Application Program (AP) defines transaction boundaries, specifies actions that constitute a transaction, and uses resources; 2) Transaction Manager (TM) assigns transaction identifiers, monitors transaction processes, and takes charge of transaction completion and failure recovery; 3) Resource Manager (RM) provides access to shared resources. To achieve XA transactions, an AP will interact with both RMs and TM. In other words, application developers need to write many extra codes, which is very different from when they use a single database.

To flatten the learning curves for application developers, ShardingSphere encapsulates the detailed logic of DTP model, and incorporates a transaction manager in it. As a result, application developers can use XA transactions by the means of standard transactions, as shown in Fig. 5(b). Here, "Apps" means user applications to differ from the AP roles in DTP model, and ShardingSphere acts as both AP and TM. Figure 5(c) presents the XA transaction in ShardingSphere. When the user application sends a "commit" request to ShardingSphere, ShardingSphere will record logs and start a 2PC (2-phase commit) procedure. In phase 1, ShardingSphere sends a "prepare" message to all RMs (here an RM is a data source) to check whether this transaction can be committed. If an RM determines that its own transaction can be committed, it sends back an "OK" to ShardingSphere; otherwise, it sends a "NO", and rolls back what it has done. In phase 2, if all RMs reply "OK" in phase 1, ShardingSphere notifies all RMs to commit; otherwise, it sends a "rollback" command to all RMs. RMs will take actions according to what they have received. If all RMs reply "OK" in phase 1, but some of them commit unsuccessfully (although in very few cases, but possibility does exist such as when the server is down or the network jitters), ShardingSphere will recover the transaction after the server restarts or re-commit periodically according to the recorded logs, which guarantees data consistency.

XA transaction ensures strict ACID characteristics. However, as it will lock all necessary resources during the execution of a transaction, it is only fit for short transactions with fixed execution time. For long-time transactions, the performance of XA transactions decreases dramatically. To this end, ShardingSphere provides two options, i.e., *Local Transaction* and *BASE Transaction*, to address this issue.

Local Transaction. If users do not start the XA transaction, the XA transaction will be degraded into a 1PC (1-phase commit) procedure, which we call local transaction. As shown in Fig. 5(d), when ShardingSphere receives a "commit" or "rollback" command from the user applications, it will transfer the command to all data sources directly. Even if some data source commits fail, ShardingSphere will ignore it. Compared with XA transactions, local transactions do not need "prepare" phase, thus can improve the performance significantly.

BASE Transaction. Different from ACID that forces consistency at the end of every operation, BASE [46] is much more relaxed and accepts data inconsistency for a short period of time. BASE is the abbreviation of Basically Available, Soft state and Eventually consistent. We call the transactions that meet the three requirements BASE transactions. Usually, BASE transactions can improve the system throughout, as they do not demand for strong consistency and can reduce the contention for shared resources.

There are many types of BASE transactions, such as Sagas [47], TCC (Try-Confirm-Cancel) [48] and Seata (Simple Extensible Autonomous Transaction Architecture) [49]. Sagas and TCC transfer the transaction logic from databases to applications. They require application developers to write extra codes to compensate every business operation. If an operation (e.g., buying a ticket) fails, its corresponding compensatory operation (e.g., refunding the ticket) would be triggered, thus the overall system could achieve consistency finally. Seata proposes an AT (Automatic Transaction) mode to generate compensatory operations automatically. As shown in the grey box of Fig. 5(e), there are three roles in Seata: 1) Transaction Coordinator (TC) maintains the status of global and branch transactions, and drives the global commit or rollback; 2) Transaction Manager (TM) defines the scope of global transaction; and 3) Resource Manager (RM) manages resources and drives the branch transaction commit or rollback.

Seata AT Mode does not intrude business logic much. However, it still needs developers to add annotations to their codes. Besides, it does not provide data sharding ability. To this end, ShardingSphere encapsulates Seata, and combines data sharding and BASE transactions together seamlessly. As shown in Fig. 5(e), the user applications need only to interact with ShardingSphere in a standard way of database connection. The BASE transaction in ShardingSphere is a 2PC procedure, where ShardingSphere performs both roles of TM and RM in Seata, as shown in Fig. 6. In phase 1, when a user application starts a BASE transaction, ShardingSphere will first require a global transaction ID from TC, register local transactions to TC, and then concurrently ask each data source to start a local transaction. During each local transaction, the data source will first save the redo and undo logs, and then commit locally. After that, ShardingSphere will report the status of the local transaction to TC. In phase 2, after receiving the command

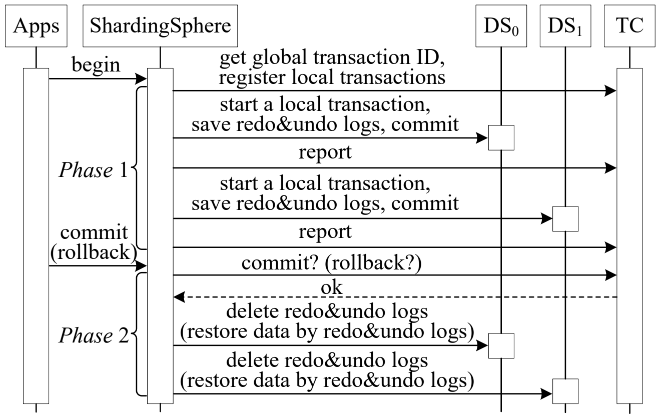

Apps
ShardingSphere
DS_0
DS_1
TC
begin
get global transaction ID, register local transactions
Phase 1
start a local transaction, save redo&undo logs, commit
report
start a local transaction, save redo&undo logs, commit
report
commit (rollback)
commit? (rollback?)
ok
Phase 2
delete redo&undo logs (restore data by redo&undo logs)
delete redo&undo logs (restore data by redo&undo logs)

Fig. 6. BASE Transactions in Apache ShardingSphere.

from the user application, ShardingSphere will check the status with TC. If all local transactions are committed successfully, each data source will delete the redo and undo logs. Otherwise, each data source will restore the data by redo and undo logs.

Currently, for compatibility with relational database protocols, the "commit" or "rollback" commands are triggered by Apps, and the result of each request is returned synchronously. These make BASE transactions have to wait for all data sources to complete their local transactions. In our future work, we plan to support asynchronous return of results, in which Apps only submit SQL statements to ShardingSphere, and ShardingSphere will guarantee BASE transactions automatically. This can improve the performance tremendously.

C. Other Features

Apart from data sharding and distributed transactions, ShardingSphere also provides many other useful features, including but not limited to: Read-Write Splitting [50], Encrypting, Shadowing (i.e., creating a shadow database and routing the corresponding test SQL to it), Scaling, Circuit Breaking and Throttling. All of these features are transparent to application developers, as ShardingSphere can intelligently identify necessary information from the standard SQL statements. We design and implement these features in a modular way. As a result, they can be added, removed, or combined with data sharding freely according to usage scenarios. More information can be found in our user manual [13].

V. Governor

In this section, we introduce the details of Governor, including configuration management and health detection.

A. Configuration Management

The configuration information is stored in Apache ZooKeeper [51], a mature and powerful distributed coordination system that provides efficient memory management and distributed lock services. Most existing sharding middlewares require users to configure the sharding rules by writing configuration files manually, which is not friendly to developers, as they are used to operating data through SQL. To this end, we propose a novel DistSQL (Distributed SQL) that allows users to configure ShardingSphere in a way of using a database. DistSQL is divided into RDL, RQL and RAL:

(1) **RDL** (Resource & Rule Definition Language). It adds, alters or drops the resources and rules. For example, to add or alter sharding rules, we can use the following SQL statement:

```
CREATE|ALTER SHARDING TABLE RULE t_user_h (
  RESOURCES(ds0, ds1), SHARDING_COLUMN=uid,
  TYPE(NAME=hash_mod,
      PROPERTIES("sharding-count"=2)));
```

Here, we propose a new **AutoTable** concept. Traditionally, to perform data sharding, the users should create the physical tables first, and then configure the sharding rules manually based on these physical tables. In the statement above, AutoTable allows users to not care about which databases store the tables. All they need do is to tell ShardingSphere what the data sources are and how many shards should be. ShardingSphere would calculate the data distribution, create the physical tables in the underlying data sources, and bind the logic tables to actual tables intelligently. In this example, ShardingSphere will automatically create two physical tables `t_user_h0` and `t_user_h1` in `ds0` and `ds1`, respectively. AutoTable reduces the cost of data sharding further.

(2) **RQL** (Resource & Rule Query Language). It queries and displays existing resources and rules. For example, we can use the following SQL statement to list all sharding rules:

```
SHOW SHARDING TABLE RULES;
```

(3) **RAL** (Resource & Rule Administration Language). It is responsible for the added-on administrator features, such as switching transaction types and scaling. For example, to switch the transaction types, we can simply use:

```
SET VARIABLE transaction_type = <type>;
```

where `<type>` could be "LOCAL", "XA" or "BASE".

Note that the configurations of all provided features can be achieved through DistSQL. DistSQL breaks the boundary between middlewares and databases, thus allowing developers to use ShardingSphere just like a database. For more information about DistSQL, please refer to [13].

B. Health Detection

To guarantee high reliability, we can set up multiple ShardingSphere-Proxy instances with load balancing tools, and integrate mature primary-secondary high available frameworks (e.g., MGR (MySQL Group Replication) and Orchestrator [52]) for the underlying databases. Governor launches a thread to check periodically the statuses of each ShardingSphere-Proxy instance and the underlying databases. If one ShardingSphere-Proxy is down or the primary nodes are changed, Governor would change the configurations automatically, which ensures the system still work correctly.

VI. SQL Engine

ShardingSphere designs and implements a complete SQL engine based on ANTLR [53] for data sharding and other features. As a result, all features in ShardingSphere can be achieved by a single SQL statement.

A. SQL Parser

The SQL parser of ShardingSphere is not fundamentally different from that of other databases. It converts the SQL statement into an AST (Abstract Syntax Tree). However, to support various underlying databases, we provide SQL dialect dictionaries of different types of databases in it.

B. SQL Router

SQL router matches logical SQL statements to data nodes based on the parsing results. There are two main route strategies: 1) *Broadcast Route* and 2) *Sharding Route*.

Broadcast Route. If the logical SQL statements do not contain sharding keys, they will be broadcast to all data nodes. These SQL statements could be: 1) DQL (Data Query Language) and DML (Data Manipulation Language) that do not have sharding keys, which should be avoided if possible; 2) DDL (Data Definition Language), DCL (Data Control Language) and TCL (Transaction Control Language). We only focus on data sharding in this paper, so please refer to [13] for more details about broadcast route.

Sharding Route. For those SQL statements that contain sharding keys, we can do more things to reduce the computational costs and accelerate the queries by sharding route. Sharding route can be further divided into two main sub-strategies:

(1) **Standard Route**. If a request only involves one logic table or binding tables, we use standard route strategy. Here, if the sharding key operator in the logical SQL statement is "=", the route result will fall into a single data node; but if it is "`BETWEEN`" or "`IN`", the route result could fall in more data nodes, thus the logical SQL statement is converted into one or more real SQL statements. For example, if `t_user` is horizontally sharded into `t_user_h0` and `t_user_h1`, the route result of "`SELECT * FROM t_user WHERE uid IN (1, 2)`" will be:

```
SELECT * FROM t_user_h0 WHERE uid IN (1,2);
SELECT * FROM t_user_h1 WHERE uid IN (1,2);
```

If the SQL statement joins binding tables, and the join conditions involve sharding key equality, we can optimize for that, since the entries with the same sharding key will be assigned to the same data node. For example, given:

```
SELECT * FROM t_user u JOIN t_order o
  ON u.uid = o.uid WHERE uid IN (1,2);
```

the route result would be:

```
SELECT * FROM t_user_h0 u JOIN t_order_h0 o
  ON u.uid = o.uid WHERE uid IN (1,2);
SELECT * FROM t_user_h1 u JOIN t_order_h1 o
  ON u.uid = o.uid WHERE uid IN (1,2);
```

(2) **Cartesian Route**. If the SQL statement contains two or more tables and these tables do not have binding relationship, to ensure correct results, we use Cartesian route, i.e., the joint query between non-binding tables needs to be split into Cartesian product combination. For example, if `t_user` and `t_order` are not binding tables, the SQL statement given in the previous example will be routed to:

```
SELECT * FROM t_user_h0 u JOIN t_order_h0 o
  ON u.uid = o.uid WHERE uid IN (1,2);
SELECT * FROM t_user_h0 u JOIN t_order_h1 o
  ON u.uid = o.uid WHERE uid IN (1,2);
SELECT * FROM t_user_h1 u JOIN t_order_h0 o
  ON u.uid = o.uid WHERE uid IN (1,2);
SELECT * FROM t_user_h1 u JOIN t_order_h1 o
  ON u.uid = o.uid WHERE uid IN (1,2);
```

Cartesian route has a relatively low performance, so we recommend to bind tables that would be joined.

C. SQL Rewriter

The SQL statements written by developers face logic data sources and tables, so they cannot be executed directly in actual data sources. SQL rewriter transforms logical SQL to executable SQL. It consists of two parts:

(1) **Correctness Rewrite**. It rewrites identifiers (e.g., changing the table name from `t_user` to `t_user_h0` in the previous example), derives columns (e.g., if the data is needed by the following result merger, but not returned through the logical SQL), revises pagination (as the pagination data from multiple data sources is different from that of one single data source), and splits batched insert (i.e., when using batched insert SQL that contains sharding keys like "`INSERT INTO t_order (oid, xxx) VALUES (1, 'xxx'), (2, 'xxx')`", the SQL should be rewritten to avoid writing excessive data). Here is an example of deriving columns: given a SQL "`SELECT oid FROM t_order ORDER BY uid`", as the selected item does not contain the column `uid` required by result merger, it needs to be rewritten to: "`SELECT oid, uid AS ORDER_BY_DERIVED_0 FROM t_order ORDER BY uid;`"

(2) **Optimization Rewrite**. To improve performance without influencing correctness, we can further optimize the SQL with the following two methods: 1) Single Node Optimization. If the SQL is routed to a single data node, it is unnecessary to derive columns, revise pagination and split batched insert mentioned before any more; 2) Stream Merger Optimization. It adds "`ORDER BY`" to the SQL that contains only "`GROUP BY`", which turns memory merger to stream merger (see Section VI-D). We will describe it in detail in Section VI-E.

D. SQL Executor

SQL executor sends the rewritten SQL statements to the underlying data sources, which is not simply through JDBC, but focuses on the balance among data source connections, memory consumption and the maximum concurrency.

On one hand, the connection number of an application should be limited, to prevent the application from occupying excessive resources and influencing the normal use of other applications. In this case, if the connections available are less than the queried data nodes, we should load all result data into memory for merging, which is called **Memory Merger**. On the other hand, if we maintain an independent connection for each data node, we can improve the query efficiency by executing the SQL concurrently, and avoid loading all result data into memory through database cursors, which is called **Stream Merger**.

SQL executor proposes two connection modes to automatically balance resource control and execution efficiency: 1) **Memory Strictly Mode**, which considers more memory usage and does not restrict the connection number of one operation. In this mode, we prefer stream merger to avoid memory overflow or frequent garbage recycle; 2) **Connection Strictly Mode**, which strictly restricts the connection consumption of each operation. This mode can use memory merger only.

However, it is hard for users to select the proper connection mode. Besides, even in the same application, different queries may fit different modes. To this end, we propose an automatic execution engine, which determines an appropriate connection mode automatically for each query. As shown in Fig. 8, there are two phases of our automatic execution engine:

(1) **Preparation Phase**. In this phase, we first group the route & rewrite results by physical data sources. Then, we decide the connection mode of each data source according to:

$$\theta = \lceil NumOfSQL \div MaxCon \rceil \quad (1)$$

where $NumOfSQL$ is the number of rewritten SQL statements routed to this data source, and $MaxCon$ is a user-configured parameter that means the maximum connections a data source can use for a query. θ is the number of SQLs that a connection executes. If $\theta > 1$, we must choose connection strictly mode and memory merger, as a connection cannot hold results for multiple SQLs. Otherwise, we can select memory strictly mode and stream merger. Note that we determine the connection mode in a data source instead of in a query, which can achieve a finer-grained optimization.

Next, we get the required connections and create execution units. However, we must be careful enough as there could be deadlocks. Suppose there are two queries A and B, and each of them requires two connections con_1 and con_2 in the same data source. If A has got con_1 but is waiting for con_2, and B is holding con_2 but waiting for con_1, a deadlock emerges. To avoid this, we get all required connections for a query atomically by adding a lock to the data source. But we do more to reduce locking times: 1) if $\theta = 1$, we will not lock data source as there could not be two queries waiting for each other; 2) we do not lock data sources if we use connection strictly mode and memory merger, as the connections will be released after the data is loaded into memory.

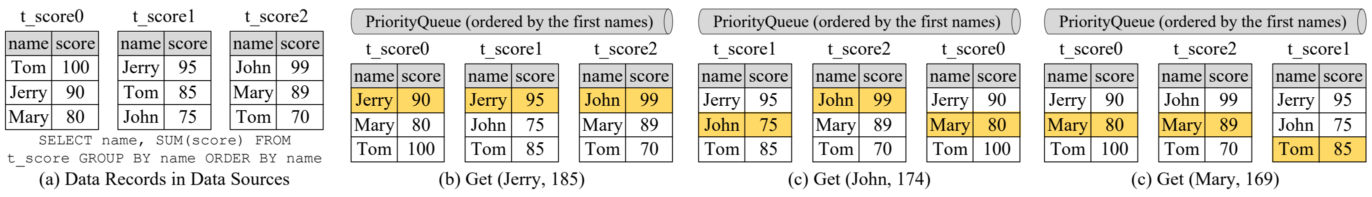
(2) **Execution Phase**. In this phase, the execution unit in a group will be sent to the corresponding underlying data source together. The data sources execute the SQLs parallelly, and send event messages for distributed transactions or monitoring. Finally, the execute result set will be sent to the next module.

E. Result Merger

Result merger combines multiple result sets from different data sources into one, and returns the result to user applications. As discussed in Section VI-D, there are two mergers, i.e., memory merger and stream merger. Compared with memory merger, stream merger consumes less memory, and it acts the same way as native databases, so we should select stream merger if possible. For different SQL statements, we take different actions.

(1) Iteration. For iteration statements (e.g., `SELECT * FROM t_user`), we adopt stream merger, and simply iterate the elements of each database cursor one by one.

(2) Order-By. If the statement contains `ORDER BY` (e.g., `SELECT * FROM t_user ORDER BY name`), as the result set returned by each data source is ordered, we adopt


SELECT name, SUM(score) FROM t_score GROUP BY name ORDER BY name
(a) Data Records in Data Sources
PriorityQueue (ordered by the first names)
(b) Get (Jerry, 185)
(c) Get (John, 174)
(c) Get (Mary, 169)

t_score0

name	score
Tom	100
Jerry	90
Mary	80

t_score1

name	score
Jerry	95
Tom	85
John	75

t_score2

name	score
John	99
Mary	89
Tom	70

(b) t_score0

name	score
Jerry	90
Mary	80
Tom	100

(b) t_score1

name	score
Jerry	95
John	75
Tom	85

(b) t_score2

name	score
John	99
Mary	89
Tom	70

(c) t_score1

name	score
Jerry	95
John	75
Tom	85

(c) t_score2

name	score
John	99
Mary	89
Tom	70

(c) t_score0

name	score
Jerry	90
Mary	80
Tom	100

(c) t_score0

name	score
Jerry	90
Mary	80
Tom	100

(c) t_score2

name	score
John	99
Mary	89
Tom	70

(c) t_score1

name	score
Jerry	95
John	75
Tom	85

Fig. 7. Example of Group-By Result Merger.

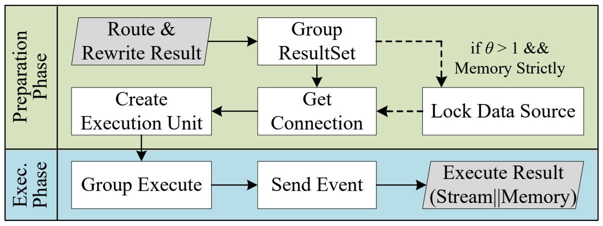


Fig. 8. SQL Executor.

stream merger to combine these result sets into one using multiway merge algorithm [54].

(3) Group-By. If the statement contains both `GROUP BY` and `ORDER BY`, and the group-by item and order-by item are the same (e.g., `SELECT name, SUM(score) FROM t_score GROUP BY name ORDER BY name`), we can use stream merger, as the data records in a group are all in the first places that the cursors point to. For example, Figure 7(a) shows the data records in the data sources. After executing the query in each data source, we merge the result sets based on multiway merge algorithm [54]. We resort to a priority queue, and visit the result sets through database cursors. In Fig. 7(b), we scan the pointed data records (marked orange) from left to right and accumulate the scores until the name is not "Jerry". After output "(Jerry, 185)", the related cursors move to the next, and the order of result sets is adjusted. The previous steps are repeated until all data records are visited, as shown in Fig. 7(c) and Fig. 7(d).

If the group-by item and order-by item are not the same, we cannot merge the result sets using stream merger, as the data records in a group are not always pointed by the cursors. In this case, we must use memory merger. If there is no `ORDER BY` in the `GROUP BY` statement, to use stream merger, SQL rewriter will intelligently add an `ORDER BY` item to the statement, as discussed in Section VI-C.

(4) Aggregation. We can use both stream merger and memory merger for aggregation statements. There are three types of aggregation functions: 1) Comparison Aggregation Functions (e.g., `MAX` and `MIN`). We compare all result sets and return the maximum or minimum value directly; 2) Summation Aggregation Functions (e.g., `SUM` and `COUNT`). We accumulate or count all result set data; 3) Average Aggregation Function (i.e., `AVG`). Since $(\texttt{AVG}(T_1)+\texttt{AVG}(T_2))/2$ is not always equal to $(\texttt{SUM}(T_1)+\texttt{SUM}(T_2))/(\texttt{COUNT}(T_1)+\texttt{COUNT}(T_2))$, where T_1 and T_2 are two result sets, for average aggregation function, we will rewrite it using summation aggregation functions.

(5) Pagination. For memory merger, we load all result data into memory and take the corresponding data directly. For stream merger, we iterate the result data, discard the data before "offset", and return the final result to user applications.

VII. Adaptors and Applications

This section first describes the two adaptors, and then gives two applications developed based on ShardingSphere.

A. Adaptors

ShardingSphere-JDBC. ShardingSphere-JDBC can be regarded as an enhanced JDBC driver. It encapsulates the entire SQL engine and other features provided by ShardingSphere, and is fully compatible with JDBC and all kinds of ORM (Object Relational Mapping) frameworks, such as Hibernate [55] and MyBatis [56]. This is to say, ShardingSphere-JDBC can be used wherever JDBC is used.

As shown in Fig. 4, ShardingSphere-JDBC is integrated into business code as a "jar" package. Since the applications using ShardingSphere-JDBC connect to data sources directly, the performance could be very high.

ShardingSphere-Proxy. ShardingSphere-Proxy is a proxy server, which forwards the requests from applications to data sources. It provides a connection pool, thus different applications and different queries can share the same connection.

ShardingSphere-Proxy disguises itself as a MySQL or PostgreSQL database by implementing their protocols. As a result, it is transparent to application developers, and supports any programming language. Besides, database administrators can connect to ShardingSphere-Proxy through MySQL Workbench or Navicat, which makes the maintenance work much easier.

Because it takes some time to forward the requests, the performance of ShardingSphere-Proxy is lower than that of ShardingSphere-JDBC. However, we can deploy both of them in a single system, where they share the same Governor. In this way, we can make full use of their respective strengths.

B. Applications

More than 170 companies all over the world announce that they are using ShardingSphere. Here are two examples:

JD Baitiao. JD Baitiao is an advanced credit payment product developed by JD.com [1], one of the biggest E-commerce companies in China. Users can use JD Baitiao to pay for online shopping. With the development of E-commerce, JD Baitiao has to face huge pressure, especially during shopping festivals. For example, its transaction volume exceeded 100 million in 10 seconds in 2019-11-11. Although the JD Baitiao team upgraded its architecture three times between 2014 and 2017, they distributed such huge amount of visiting traffic in their business code, which made the logic very complex. Since 2018, the JD Baitiao team has adopted ShardingSphere for data sharding. They used hash sharding algorithm on user IDs to avoid the hot access issue. After data sharding, the number of data nodes reached nearly 10,000. ShardingSphere made

JD Baitiao more scalable by simply adding more machines, and let its developers focus more on their business logic. This usage scenario was recorded as an official MySQL case [57].

China Telecom BestPay. China Telecom Corporation [58], the largest fixed-line service and the third largest mobile telecommunication provider in China, developed an application called BestPay to provide handy service to the public. It covers nearly 400 main cities in China, and has 500 million users and 10 million business cooperators. In 2019, the BestPay team held a marketing event, aimed at enhancing the user activity and the company's brand. In this event, the data was stored in a single MySQL table, which resulted in more than 150ms average response time and 4% query failure rate. To solve this issue, they utilized ShardingSphere to split the data into two MySQL databases in two servers using $merchant\_code\%2$, and in each database, the data was further split horizontally by month. As a result, there were no more than 1 billion entries in a table, and the response time was reduced to less than 50ms. ShardingSphere helped BestPay not only improve the service quality, but also reduce the development costs.

VIII. Evaluations

A. Datasets and Experimental Settings

Datasets. We use two datasets to evaluate the performance of ShardingSphere: 1) **Sysbench** [59], a famous database benchmarking tool that provides a table allowing users to adjust its data volume. As the Sysbench requester is implemented with C language, we write a Java version using ShardingSphere-JDBC or JDBC [60]. 2) **TPCC** [61], another widely-used OLTP benchmark that simulates several types of transactions used frequently by a shop. Its ten tables are organized with warehouses (about 600,000 entries per warehouse).

Baselines. We compare ShardingSphere with eight systems in terms of TPS (Transactions Per Second), the average response time (AvgT), the 99th percentile response time (99T) for Sysbench and the 90th percentile response time (90T) for TPCC (note that the default percentile of Sysbench and TPCC are 99th and 90th, respectively). These systems are: 1) MySQL v5.7.26 (**MS**) and PostgreSQL v10.17 (**PG**), which are typical relational databases; 2) Vitess v12.0.0 [16] and Citus v9.0.0 [17], which are sharding middlewares on MySQL and PostgreSQL, respectively; 3) TiDB v5.2.0 [7] and CockroachDB v21.1.11 [8] (**CRDB**), which are two representative new architecture databases; 4) Aurora MySQL v2.07.2 (**Aurora**$_{MS}$) and Aurora PostgreSQL v4.2 (**Aurora**$_{PG}$), which are DBaaS databases [39, 40] over MySQL v5.7 and PostgreSQL v10.17 in Amazon cloud, respectively. We do not compare with ProxySQL [18] because it routes requests based on string regular expression that cannot support the sharding rules in our experimental settings. **SSJ**$_{MS}$ and **SSJ**$_{PG}$ are ShardingSphere-JDBC on MySQL and PostgreSQL, respectively. **SSP**$_{MS}$ and **SSP**$_{PG}$ are ShardingSphere-Proxy on MySQL and PostgreSQL, respectively. All experimental codes are publicly released [60].

Settings. We use Sysbench as the default dataset, whose parameter settings are shown in Table II. For most experiments, we use a cluster of 12 virtual servers in Huawei cloud [62], where each server is equipped with CentOS 7.1 64bit, 32-vCore CPU, 64GB RAM and 1TB disk. To reduce the effect of network IOs, we enable the Linux Multiqueue Networking [63] for each server. For Aurora experiments, we use 5 virtual servers in Amazon cloud [64], where each server is equipped with Red Hat Enterprise Linux 8.3 64bit, 8-vCore CPU, 64 GB RAM and 100GB SSD. Each server runs at most one data source. The requester and ShardingSphere-Proxy are deployed in two separate servers, respectively. For MS and PG, we store all data in a single data source. For Sysbench, we shard the data into multiple data sources, and in each data source, the data is further sharded into 10 tables. For TPCC, we shard all tables into 5 data sources, and the biggest table `bmsql_order_line` is further horizontally sharded into 10 tables in each data source.

TABLE II
Parameter Settings of Sysbench (Default Values are in Bold)

Parameters	**Settings**
Scenarios	Point Select, Read Only, Write Only, **Read Write**, Update Index, Update Non-index, Delete
Data Sizes (million)	20, **40**, 60, 80, 100, 150, 200
Concurrencies	1, 20, 100, **200**, 500
# Data Servers	1, 3, **5**, 7, 9
# Transaction Types	**Local**, BASE, XA
$MaxCon$	1, 5, **10**, 15, 20

B. Comparison with Baselines

Comparison using Sysbench. Table III presents the performance of distributed systems in different Sysbench scenarios. We can see that: 1) In terms of the three metrics, the SS-based systems always perform the best in all scenarios. For example, the TPS of SSJ$_{MS}$ in "Read Write" scenario achieves 19953, which is about five times of the best other system TiDB (3877). 2) In most cases, SSJ$_{MS}$ and SSJ$_{PG}$ perform better than SSP$_{MS}$ and SSP$_{PG}$, respectively, because the requests of SSJ-based systems can be directly sent to the underlying data sources, but those of SSP-based systems are forwarded by the proxy server, which takes some time. 3) The performances of SS-based systems are relatively consistent in both MySQL and PostgreSQL, which verifies the generalization ability of ShardingSphere. 4) All systems show different performances in different scenarios. It is intuitive because different scenarios need to retrieve different amounts of data. The "Read Write" scenario performs the worst, as it will start a transaction. Since "Read Write" scenario is the most common in applications, we use it as the default scenario.

Aurora stores its data in a distributed storage service with SSDs, and processes requests in a single virtual server. To this end, in this set of experiments, we only use one virtual server for ShardingSphere and its underlying data stores. Besides, we only use 20 million records in Sysbench as MS throws an exception when we load 40 million records in our settings. As shown in Table IV, 1) Aurora$_{MS}$ and Aurora$_{PG}$ perform better than MS and PG, respectively, as the storage power of Aurora can be seen as unlimited, and Aurora pushes many computations to the storage layer. 2) Although

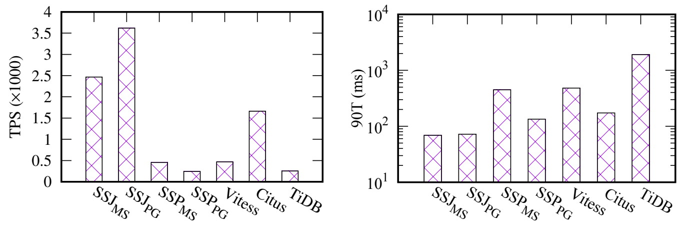
TABLE III
COMPARISON WITH DISTRIBUTED SYSTEMS IN DIFFERENT SCENARIOS (SYSBENCH)

System	Point Select			Read Only			Write Only			Read Write			Update Index			Update Non-index			Delete		
	TPS	99T	AvgT	TPS	99T	AvgT	TPS	99T	AvgT	TPS	99T	AvgT	TPS	99T	AvgT	TPS	99T	AvgT	TPS	99T	AvgT
SSJ$_{MS}$	**250929**	**1.32**	**0.8**	**50367**	**4.93**	**3.97**	**21163**	**24.58**	**9.45**	**19953**	**32.54**	**10.03**	**50736**	**14.82**	**3.95**	**50632**	**15.06**	**3.95**	**51843**	14.9	**3.86**
SSP$_{MS}$	185154	2.54	1.08	13165	22.69	15.19	14463	31.37	13.82	7959	48.09	25.12	46207	15.74	4.32	47130	15.09	4.24	49049	**14.64**	4.07
Vitess	155797	4.91	1.28	11806	24.38	16.94	5189	167.44	38.51	3175	189.93	67.87	NA	NA	NA	18638	74.46	10.73	13813	112.67	14.48
CRDB	49225	17.01	4.06	4350	71.83	45.97	2250	404.61	88.86	1611	442.73	124.1	2347	520.62	85.16	23249	38.25	8.6	8380	227.4	23.86
TiDB	141796	7.56	1.41	12140	27.66	16.47	4939	92.42	40.49	3877	101.13	51.58	12171	41.1	16.43	16819	27.17	11.89	27587	28.16	7.25
SSJ$_{PG}$	**271562**	**1.49**	**0.74**	**171390**	**1.08**	**1.18**	**61015**	**15.04**	**3.28**	**54580**	**14.63**	**3.67**	**100187**	**13.46**	**2**	**156226**	12.78	**1.28**	**174267**	12.15	**1.15**
SSP$_{PG}$	180357	2.49	1.11	12055	24.09	16.58	25474	19.65	7.85	9121	36.67	21.91	94429	13.94	2.12	138355	**12.23**	1.45	156271	**11.24**	1.28
Citus	51929	12.52	3.85	4288	73.13	46.62	6750	223.34	29.62	3129	277.21	63.89	29584	34.33	6.76	31838	21.11	6.28	37445	17.32	5.34

• 99T, 90T and AvgT are measured in milliseconds, and the best values are marked in **bold**. The same below. • Vitess does not support updating indexes.

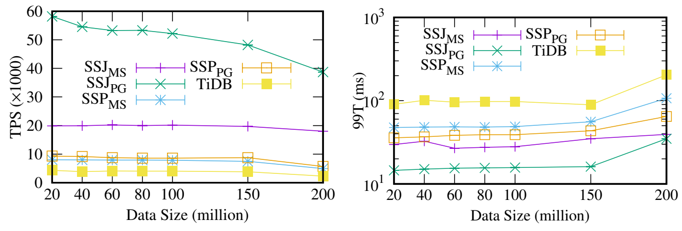
TABLE IV
COMPARISON WITH STANDALONE SYSTEMS (SYSBENCH)

System	TPS	99T	AvgT	System	TPS	99T	AvgT
MS	574	1401.61	348.55	PG	1287	337.94	155.27
SSJ$_{MS}$	**4751**	**152.75**	**42.27**	SSJ$_{PG}$	**3674**	224.11	**54.61**
SSP$_{MS}$	380	601.29	555.04	SSP$_{PG}$	333	816.63	600.23
Aurora$_{MS}$	621	1533.66	289.13	Aurora$_{PG}$	2043	**150.29**	97.89



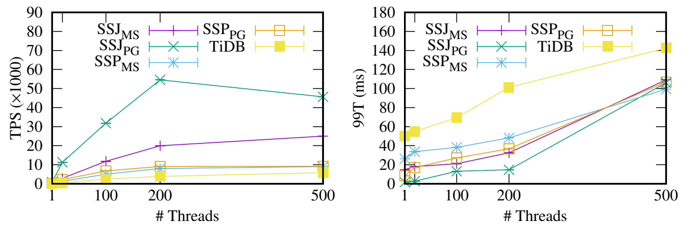
(a) TPS VS Systems. (b) 90T VS Systems.

Fig. 9. Comparison with Distributed Systems (TPCC).



(a) TPS VS Data Sizes. (b) 99T VS Data Sizes.

Fig. 10. Different Data Sizes (Sysbench).



(a) TPS VS Concurrency. (b) 99T VS Concurrency.

Fig. 11. Different Concurrencies (Sysbench).

Aurora-based systems perform better than the corresponding SSP-based systems, SSJ-based systems usually perform the best in terms of TPS and AvgT. Aurora may encounter the network bottleneck for its separation of compute and storage. 3) Although MS uses the same resource (a single server) with SSJ$_{MS}$, SSJ$_{MS}$ performs much better than MS in terms of all three metrics. A similar thing happens between SSJ$_{PG}$ and PG. This is because ShardingSphere shards the data into 10 smaller tables. Requests on smaller tables are much faster.

TPS and AvgT are consistent in both Table III and Table IV, i.e., if a system has a higher TPS, it usually has a smaller AvgT. Besides, comparing AvgT, 99T or 90T is more critical. To this end, in the following we do not present AvgT.

Comparison using TPCC. We also conduct a set of experiments for the distributed systems using TPCC with 200 warehouses. Although TiDB provides its own TPCC version with many optimizations for itself, we adopt the native TPCC for fairness. TPCC provides five scenarios, and the proportion of each scenario is fixed, so we only give the overall performance, and take the accumulated 90T of all scenarios as the final time. As shown in Fig. 9, although SSP-based systems have the relatively worse performance than Vitess or Citus, SSJ-based systems perform the best as they have the biggest TPS and the smallest response time. TiDB takes the most time among all the systems, as it spends 1.61s in the “Delivery” scenario. CRDB runs error with the native TPCC. It shows similar performance with TiDB according to [7], thus we infer that its performance is worse than ours.

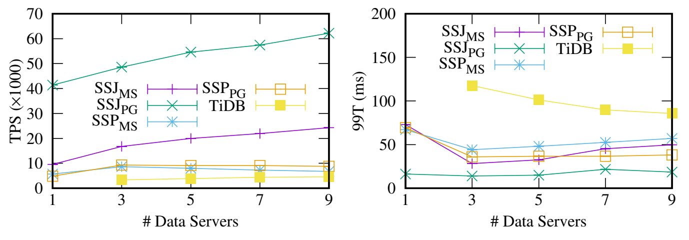
C. Test of Scalability

To test the scalability of ShardingSphere, we carry out a set of experiments with different data sizes, request concurrencies and data servers using Sysbench. Here, among the distributed baselines, we only present the performance of TiDB, as TiDB exhibits the best performance in “Read Write” scenario among all baselines according to Table III.

Different Data Sizes. As shown in Fig. 10, with the data size increasing from 20 million to 100 million, the performance of all presented systems keeps relatively stable. However, if the data size still increases to 200 million, all systems produce less TPS and take more 99T. Most systems store their data in a tree structure. More data will lead to a higher tree structure. Therefore, it will access the disks more times. SSJ-based systems always perform the best in all data sizes.

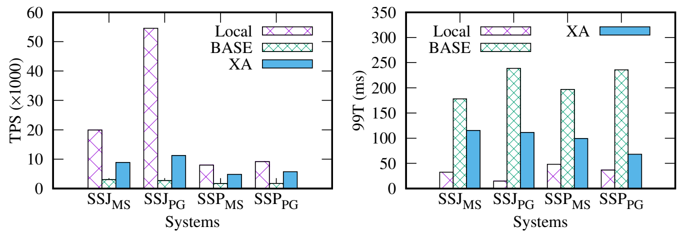
Different Concurrencies. Figure 11 shows that, with more request threads, the TPS of all systems first increases and then keeps stable, while their 99T first keeps stable but increases sharply when the number of threads is more than 200. This is because when the number of threads is small, the systems are able to respond to all requests. But if the concurrency is greater than a threshold, some requests have to wait for the resources. Again, SSJ-based systems perform the best in terms of TPS among all systems for different concurrencies.

Different Data Servers. Figure 12 depicts the performance



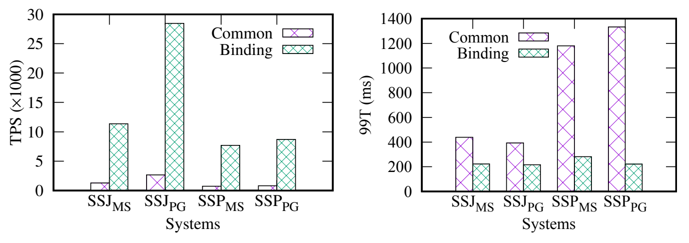
(a) TPS VS Data Servers. (b) 99T VS Data Servers.

Fig. 12. Different Data Servers (Sysbench).



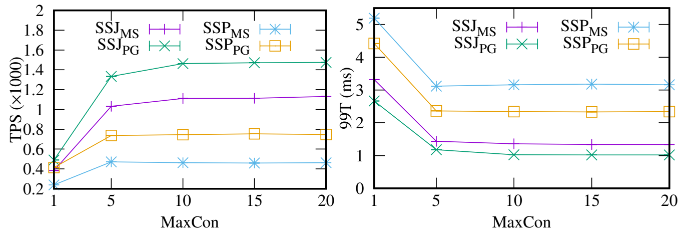
(a) TPS VS Transaction Type. (b) 99T VS Transaction Type.

Fig. 13. Effects of Transaction Types (Sysbench).



(a) TPS VS Binding Table. (b) 99T VS Binding Table.

Fig. 14. Effects of Binding Table (Sysbench).



(a) TPS VS MaxCon. (b) 99T VS MaxCon.

Fig. 15. Effects of MaxConnectionSizePerQuery (Sysbench).

with different number of data servers. Here, TiDB is not tested in one data server because it needs at least three data servers for Raft consensus. As shown in Fig. 12(a), with more data servers, the TPS of SSJ-based systems increases. Because with more data servers, the number of data records in each data source is smaller, which can improve the overall TPS. However, for SSP-based systems, the TPS first increases slightly, and then keeps stable when the number of data servers is greater than 3. There could be for two reasons. First, we only employ one proxy server, which could be one bottleneck (we can deploy more proxy servers to address this issue). Second, with more data servers, the network could be another bottleneck. Similar things can be seen in Fig. 12(b), where the 99T first drops and then keeps stable or even increases slightly with more data servers.

D. *Effects of Configurations*

Effects of Transaction Types. Figure 13 presents the effects of three types of transactions. We can see that local transaction always performs the best in terms of both TPS and 99T. XA transaction performs worse than local transaction because local transaction adopts 1PC and does not wait the underlying data sources to respond, while XA needs 2PC to guarantee strong consistency. It is interesting to see that BASE transaction performs worse than XA transaction. There could be two reasons. First, in our settings, the requests belong to short transactions, which cannot reflect the advantages of BASE transaction. Second, to be compatible with relational database protocols, we let the results returned synchronously for BASE transaction, which also hurts its performance.

Effects of Binding Table. To verify the effects of binding table, we conduct a set of experiments by joining two the same logical tables. Each logical table contains 20 million records, and is split using the default sharding strategy. "Common" means that the two tables do not have binding relationship. As shown in Fig. 14, the performance of binding tables is about 10 times better than non-binding tables in terms of TPS.

Effects of $MaxCon$. In this experiment, we only use one thread to avoid the impact of concurrent access (the CPU cores will act as the bottleneck if there are more threads), and perform a range query. As shown in Fig. 15, the performance of all SS-based systems gets better if $MaxCon$ increases from 1 to 5. However, their performance keeps stable if $MaxCon$ increases further. If $MaxCon$ is small, it tends to select the connection strictly mode and memory merger. As the range query will produce multiple routed SQLs, these routed SQLs have to be executed one by one. Increasing $MaxCon$ enables to execute multiple routed SQLs at the same time. However, if $MaxCon$ increases further, it tends to select the memory strictly mode and stream merger, the bottleneck should be the underlying data sources and network transmission.

IX. Conclusion and Future Works

This paper presents the open-source data sharding system Apache ShadingSphere, which lets users use sharded databases like one database. Extensive experiments are conducted based on two famous benchmarking tools, verifying that the performance of ShardingSphere is better than other sharding systems and new architecture databases in most cases in our settings. More and more companies are using ShardingSphere for their legacy critical applications. As for future work, we will provide a "database+" production based on ShardingSphere, and build an ecosystem with more pluggable features.

Acknowledgment

This paper is supported by the National Key R&D Program of China (2019YFB2103201), the National Natural Science Foundation of China (61976168, 61872050, 62172066, 42174050), and Entrepreneurship and Innovation Support Program of Chongqing for Overseas Returnees, No. cx2021047. We would like to thank all contributors and users (individuals and companies) of Apache ShardingSphere.

References

[1] "Jd.com," https://en.wikipedia.org/wiki/JD.com, 2021.

[2] "Tmall," https://en.wikipedia.org/wiki/Tmall, 2021.

[3] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database system implementation*. Prentice Hall Upper Saddle River, NJ:, 2000, vol. 672.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[5] L. George, *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[6] A. Pavlo and M. Aslett, "What's really new with newsql?" *ACM Sigmod Record*, vol. 45, no. 2, pp. 45–55, 2016.

[7] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[8] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *ACM SIGMOD 2020*, 2020, pp. 1493–1509.

[9] "yugabytedb," https://www.yugabyte.com/, 2021.

[10] "Sql-92," http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, 2021.

[11] R. C. Seacord, "Replaceable components and the service provider interface," in *International Conference on COTS-Based Software Systems*. Springer, 2002, pp. 222–233.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, *Elements of reusable object-oriented software*. Addison-Wesley Reading, Massachusetts, 1995, vol. 99.

[13] "Apache shardingsphere," https://github.com/apache/shardingsphere, 2021.

[14] "Mysql proxy," https://github.com/mysql/mysql-proxy, 2021.

[15] "Maxscale," https://github.com/mariadb-corporation/MaxScale, 2021.

[16] "Vitess," https://github.com/vitessio/vitess, 2021.

[17] "Citus," https://github.com/citusdata/citus, 2021.

[18] "Proxysql," https://github.com/sysown/proxysql, 2021.

[19] "Goldendb," https://www.zte.com.cn/global/products/202003190856/202003190858/201707311038, 2021.

[20] "Tdsql-mysql," https://intl.cloud.tencent.com/product/dcdb, 2021.

[21] R. Li, H. He, R. Wang, Y. Huang, J. Liu, S. Ruan, T. He, J. Bao, and Y. Zheng, "Just: Jd urban spatio-temporal data engine," in *ICDE 2020*. IEEE, 2020, pp. 1558–1569.

[22] R. Li, H. He, R. Wang, S. Ruan, T. He, J. Bao, J. Zhang, L. Hong, and Y. Zheng, "Trajmesa: A distributed nosql-based trajectory data management system," *TKDE*, 2021.

[23] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng, "Trajmesa: A distributed nosql storage engine for big trajectory data," in *ICDE 2020*. IEEE, 2020, pp. 2002–2005.

[24] E. Brewer, "A certain freedom: thoughts on the cap theorem," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2010, pp. 335–335.

[25] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[26] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[27] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[29] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[30] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[31] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews, "The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1401–1412, 2016.

[32] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE 2011*. IEEE, 2011, pp. 195–206.

[33] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner *et al.*, "F1-the fault-tolerant distributed rdbms supporting google's ad business," 2012.

[34] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner *et al.*, "F1: A distributed sql database that scales," 2013.

[35] J. Yang, I. Rae, J. Xu, J. Shute, Z. Yuan, K. Lau, Q. Zeng, X. Zhao, J. Ma, Z. Chen *et al.*, "F1 lightning: Htap as a service," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3313–3325, 2020.

[36] M. Freels, "Faunadb: An architectural overview," 2018.

[37] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach *et al.*, "Foundationdb: A distributed unbundled transactional key value store," in *ACM SIGMOD 2021*, 2021, pp. 2653–2666.

[38] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan *et al.*, "Spanner: Becoming a sql system," in *ACM SIGMOD 2017*, 2017, pp. 331–343.

[39] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *ACM SIGMOD 2017*, 2017, pp. 1041–1052.

[40] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli *et al.*, "Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes," in *ACM SIGMOD 2018*, 2018, pp. 789–796.

[41] "Jdbc," https://en.wikipedia.org/wiki/Java_Database_Connectivity, 2021.

[42] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li, "opengauss: An autonomous database system," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 3028–3042, 2021.

[43] "Sharding algorithms of apache shardingsphere," https://shardingsphere.apache.org/document/current/en/dev-manual/sharding/, 2021.

[44] "The open group," https://www.opengroup.org/, 2021.

[45] C. Specification, *Distributed Transaction Processing: the XA Specification*. X/Open, 1991.

[46] D. Pritchett, "Base: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability." *Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[47] H. Garcia-Molina and K. Salem, "Sagas," *ACM Sigmod Record*, vol. 16, no. 3, pp. 249–259, 1987.

[48] G. Pardon and C. Pautasso, "Atomic distributed transactions: A restful design," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, pp. 943–948.

[49] "Seata," https://github.com/seata/seata, 2021.

[50] Z. Jiang, Y. Zhang, J. Wang, C. Li, and C. Xing, "Tl: A high performance buffer replacement strategy for read-write splitting web applications," in *Asia-Pacific Web Conference*. Springer, 2014, pp. 478–484.

[51] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

[52] "Orchestrator," https://github.com/openark/orchestrator, 2021.

[53] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[55] "Hibernate," https://hibernate.org/, 2021.

[56] "Mybatis," https://blog.mybatis.org/, 2021.

[57] "Mysql customer: Jd.com," https://www.mysql.com/customers/view/?id=1461, 2021.

[58] "China telecom corporation," https://en.wikipedia.org/wiki/China_Telecommunications_Corporation, 2021.

[59] A. Kopytov, "Sysbench: a system performance benchmark," *http://sysbench.sourceforge.net/*, 2004.

[60] "Ss4icde," http://ss4icde.urban-computing.com, 2021.

[61] S. T. Leutenegger and D. Dias, "A modeling study of the tpc-c benchmark," *ACM Sigmod Record*, vol. 22, no. 2, pp. 22–31, 1993.

[62] "Huawei cloud," https://www.huaweicloud.com/intl/en-us/, 2021.

[63] Z. Yi and P. Waskiewicz, "Enabling linux network support of hardware multiqueue devices," in *Proc. of the 2007 Linux Symposium*. Citeseer, 2007, pp. 305–310.

[64] "Amazon cloud," https://aws.amazon.com/, 2022.