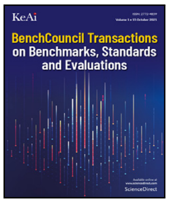
Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/



Are current benchmarks adequate to evaluate distributed transactional databases?

Luyi Qu [a], Qingshuai Wang [a], Ting Chen [a], Keqiang Li [a], Rong Zhang [a,*], Xuan Zhou [a], Quanqing Xu [b], Zhifeng Yang [b], Chuanhui Yang [b], Weining Qian [a], Aoying Zhou [a]

[a] *East China Normal University, China*
[b] *OceanBase, China*

ARTICLE INFO

Keywords:
Distributed transactional databases
Benchmarks
Workload

ABSTRACT

With the rapid development of distributed transactional databases in recent years, there is an urgent need for fair performance evaluation and comparison. Though there are various open-source benchmarks built for databases, it is lack of a comprehensive study about the applicability for distributed transactional databases. This paper presents a review of the state-of-art benchmarks with respect to distributed transactional databases. We first summarize the representative architectures of distributed transactional databases and then provide an overview about the chock points in distributed transactional databases. Then, we classify the classic transactional benchmarks based on their characteristics and design purposes. Finally, we review these benchmarks from schema and data definition, workload generation, and evaluation and metrics to check whether they are still applicable to distributed transactional databases with respect to the chock points. This paper exposes a potential research direction to motivate future benchmark designs in the area of distributed transactional databases.

Contents

* Corresponding author.
E-mail addresses: luyiqu@stu.ecnu.edu.cn (L. Qu), qswang@stu.ecnu.edu.cn (Q. Wang), tingc@stu.ecnu.edu.cn (T. Chen), kqli@stu.ecnu.edu.cn (K. Li), rzhang@dase.ecnu.edu.cn (R. Zhang), xzhou@dase.ecnu.edu.cn (X. Zhou), xuquanqing.xqq@oceanbase.com (Q. Xu), zhuweng.yzf@oceanbase.com (Z. Yang), rizhao.ych@oceanbase.com (C. Yang), wnqian@dase.ecnu.edu.cn (W. Qian), ayzhou@dase.ecnu.edu.cn (A. Zhou).

https://doi.org/10.1016/j.tbench.2022.100031
Received 30 September 2021; Received in revised form 15 January 2022; Accepted 15 January 2022
Available online 10 February 2022

1. Introduction

Though the traditional stand-alone relational database management system (*RDBMS*) has attracted great attention, it is still limited in the scalability of storage and computing for large application scenarios, e.g., *Securities Exchange*. Business expansion or new application emerging with a characteristic of high throughput or large storage promotes the designing and developing of distributed databases, which have become a hot topic in both academia and industry [1–13]. During the long exploration process from the stand-alone database to the distributed one, various solutions are coming up with respect to different application requirements. For example, in E-commerce [14], it may separate $Read$ from $Write$ to improve throughputs ; when meeting hotspots, it may split data partition or move a part to a free node. All the effort is to realize database scalability on different implementation modules. The representative distributed databases are arranged along the timeline by either the paper published time or the project opensourced time in github (tagged by g) in Fig. 1. Notice that, the earliest opensourced version of *OceanBase* is published in 2014, but its architecture is quite different from the current one. Therefore, we put the latest one here.

NoSQL databases [15–17] firstly design many ways to leverage distributed storages and computational power from multiple machines, which are widely used for web applications. These businesses have critical requirements for large storage and concurrent processing. However, due to the lack of *SQL* compatibility and *ACID* assurance, it is not easy to take place of *RDBMS*.

Distributed transactional databases, therefore, are built, which not only meet the strong consistency requirement of transactional databases, but also support scalability of NoSQL databases. They are considered as one kind of NewSQL databases [18]. *H-Store* [1], *VoltDB* [2], *OceanBase* [3], and *Citus*, [4] design a *Shared-Nothing* model to achieve a distributed database. Specifically, the model takes predefined rules to partition data into multiple (distributed) nodes. Each node covers the functionality modules of *Query*, *Transaction* and *Storage* engines, and coordinates the other ones when and only when executing distributed transactions or distributed queries. Generally, it addresses two key scalability issues, i.e., storage scalability and distributed transaction processing scalability. Nevertheless, it lacks of an effective scheduling mechanism and depends well on pre-defined partitioning rules, which demands great effort from the business developers, i.e., to understand the business. Otherwise, it may lead to frequent distributed transactions, which seriously degrades the overall performance.

Therefore, *Spanner* [5], *TiDB* [6] *CockroachDB* [7] and *FoundationDB* [11] extend the *Shared-Nothing* model by separating compute and storage to achieve flexibility and scalability at the same time, that is to separate the query engine and storage engine. The query engine is state-less and can run in any number of nodes. The storage engine provides transactional storage access with multiple replicas. Meanwhile, instead of relying on the predefined distribution rules, they divide the data into sub-blocks and use a centralized management controller to balance the storage and workload among all storage nodes. Therefore, a compute node can access any storage node with enough flexibility, and schedule data and workload according to optimized goals, e.g., storage or performance. However, it produces worse performance under complex distributed transactions because of introducing too many network interactions (I/Os).

Aurora [8,9], *PolarDB* [10], *Socrates* [13] and *Taurus* [12] then design a *Shared-Storage* model that sacrifices the flexibility and scalability of write to deal with more complex transactional workloads. Specifically, they keep the query engine and transaction management on one compute node and then use multiple storage nodes to store the data and logs. All other compute nodes are read-only nodes, relieving the burden on the write node by separating reads and writes. This architecture has full scalability for storage and read workload, but the single write node has a high probability to bottleneck *DBMS*. However, since all transactions are executed within a single node, their performance is not affected by distributed transactions even with complex transactions.

In summary, different distributed models have been specified and designed for different application scenarios, which may expose high performance for its target application. It is urgent to benchmark these distributed databases to make a fair comparison and benchmarking

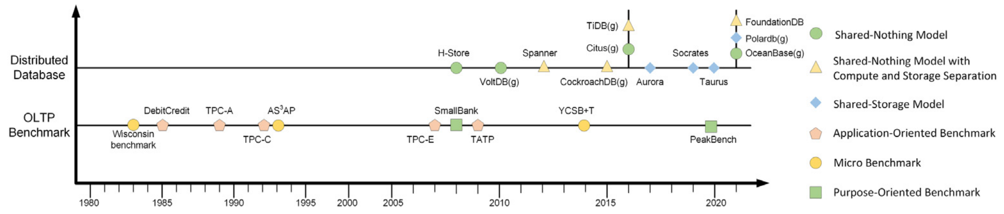


Fig. 1. Development of distributed databases and OLTP benchmarks.

among them, and to expose the choke points of different distributed databases. It will then further promote the development of databases, which is especially useful for database marketing, engineering and targeting sales & innovations.

The commonly used benchmarks for transactional databases (OLTP benchmarks) are presented in Fig. 1, ranging from 1983 to 2020. OLTP benchmarks can be divided into three categories. The first type is $micro$-benchmarks, which consist of simple *CRUD* operations (*Create, Retrieve, Update, Delete*) and do not represent any application or business. These benchmarks, hence, are able to compare the performance of transactional databases with respect to the basic operations. Specifically, university of *Wisconsin* formulates a benchmark for $DBMS$ (also called *Wisconsin* benchmark) [19], which is the first one to define a set of SQL statements to compare the database performance by collecting their total execution time. It, however, has several serious limitations, which are lack of data type supports, no batch update or concurrency control. To deal with these problems, two designers of *Wisconsin* benchmark, i.e., *Turbyfill* and *Bitton*, together with *Orji* introduce *ANSI Sql Standard Scalable and Portable* benchmark, i.e., *AS*^{3}*AP* benchmark [20]. The improvements include not only adding more tables, data types, index types, data distributions and the number of *SQL* statements, but also increasing the volume of data. Furthermore, isolation levels are introduced to evaluate concurrency control. Both *Wisconsin* benchmark and *AS*^{3}*AP* benchmark, however, do not wrap SQL statements into transactions. In addition, when data size continues to increase, the scalability of databases attracts more attention. *YCSB+T* [21], an extension of *YCSB* [22] designed to evaluate $NoSQL$ databases, covers the standard read, write, update, delete and scan operations in *YCSB*, based on which it composes transactions. *YCSB+T* has an additional validation stage to check the consistency of a distributed database to guarantee the correctness of execution.

The second type is the benchmark focusing on evaluating database for different applications. These benchmarks are constructed from the characteristics of the specific applications, which may reflect the business logics. *TATP* [23], proposed by *IBM*, abstracts the operations in a telecommunication business application. It introduces physical resource consumption as one of its metrics. Besides, *Jim Gray* together with *Tandem Company* simulates a *Debit Credit* application of the bank, named as *DebitCredit* [24]. *TPC-A* [25] benchmark by *TPC Council* is built upon *DebitCredit* benchmark, and it is the first time to formalize the evaluation rules including specifying $ACID$ properties, which all database vendors are required to obey. *TPC-A* is rarely used in recent years because it is too simple to satisfy the needs of current applications. Later, *TPC Council* proposes a more complicated benchmark, *TPC-C* [26], which simulates a warehouse and order management application. Most database vendors have participated in *TPC-C* evaluation to demonstrate the performance of their databases, including distributed transaction databases, such as *OceanBase*. *TPC-E* is the most complex transactional benchmark, which simulates the typical behaviors in a stock brokerage company.

The third type of the benchmark is designed for exploring and simulating the specific characteristics of applications, such as *SmallBank* [27] and *PeakBench* [28]. *SmallBank* is designed to evaluate DBMS for the workload with the characteristics of the read–write conflict(called anti-dependency). It can be used to evaluate different serializable protocols under snapshot isolation. It contains simple read and write operations involving a small number of tuples. *PeakBench* is designed to evaluate $DBMS$ for the workload with the characteristics of *sharp dynamics, terrific skewness, high contention*, and *high concurrency*, that is to simulate a second kill application in *Alibaba*.

We arrange the mentioned classic distributed transactional databases and existing transactional benchmarks in Fig. 1 according to the chronological order when the databases/benchmarks are either open sourced in github(labeled as (g)) or published officially. Different shapes and colors represent the different types of databases and benchmarks. It is obvious that most of *OLTP* benchmarks are proposed before 2010, along with the growth of centralized $DBMS$s. The prosperity of distributed databases starts from *H-Store*, the popular distributed database around 2008. Though *PeakBench* is the most recently proposed benchmark transactional databases, it focuses on the performance comparison with the write intensive workload rather than analyzing or benchmarking distributed databases. Besides, the existing surveys cover the fields including $NoSQL$ benchmark [29,30], Big Data system benchmark [31–33], decision support benchmark [32], graph processing system benchmark [34], etc. However, no work conducts a benchmark survey for distributed transaction databases. Therefore, it is imperative to answer whether the existing benchmarks are still applicable to evaluating distributed transactional databases. And if they are not suitable for distributed transactional database evaluation, what are the disadvantages?

The contributions of this paper are: (i) we summarize the representative architectures of distributed transactional databases and expose the potential choke points in their design; (ii) we discuss and analyze the deficiency of the state-of-art benchmarks with respect to these choke points; (iii) we provide a guideline to revise the existing benchmarks or design a new benchmark for benchmarking distributed transactional databases.

2. Architecture and choke points of distributed databases

In this section, we will discuss about the components with respect to the scalability of distributed databases, and then explore the choke points of transaction processing of distributed databases.

2.1. Architecture of distributed databases

Shared-Nothing model and *Shared-Storage* model are two popular distributed database models with various implementations and optimizations. Specifically, the *Shared-Storage* model is proposed for the cloud-oriented databases recently. For *Shared-Nothing*, we introduce two classic kinds of databases, i.e., *Bundle/Separation of Compute and Storage*. For *Shared-Storage*, we mainly introduce *Aurora*, which uses the model in the cloud firstly and then presents the optimization made by others.

2.1.1. Shared-nothing model

Bundle of Compute and Storage As far as we know, *H-Store* [1] is the first distributed *DBMS* with SQL compatibility and $ACID$ support. $VoltDB$ [2] is a business implementation of *H-Store* with the architecture shown in Fig. 2. It divides all data into the main memory of a distributed cluster and uses *K-Safety* mechanism to tolerate errors. Each node in *H-Store* has an *Execution Engine* and *Partition Data*, which are

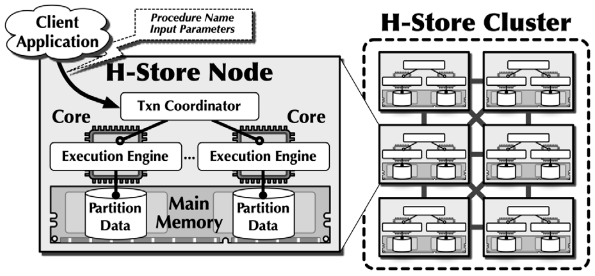


Fig. 2. H-store architecture [1].

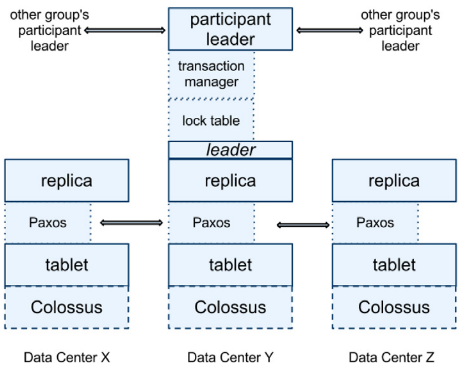


Fig. 3. Spanner architecture [5].

tied together. *H-Store* proposes a novel partitioned-based concurrency control to execute distributed transactions. Specifically, *Txn Coordinator* divides a stored procedure-based transaction into one or more partitions. If and only if all the partitions execute successfully, the transaction can be successfully committed. Otherwise, the transaction fails. Therefore, if the workload on different partitions is unbalanced, the throughput lows down. Additionally, *K-Safety* requires the same replica to be equal, so the existence of replica nodes can be used to relieve read pressure.

Citus [4], as a distributed database plugin for $PostgreSQL$, organizes multiple *PostgreSQLs* into a distributed database cluster. The data is distributed across multiple machines according to the predefined rules, which is synchronized using master–slave replication. *Citus* uses a coordinator node to distribute and coordinate transactions, and launches distributed transactions based on *MVCC+2PL* protocol. However, *Citus* cannot support globally consistent snapshots due to the lack of timestamp synchronization among multiple nodes, which makes it weak in transaction execution.

OceanBase [3] is a commercial distributed database. The data is distributed similarly to *Citus* and achieves consistency among replicas by its self-developed *Paxos* algorithm. *OBProxy* is a stateless connector to clients, which routes SQL statements by the proxy table. There can be multiple *OBProxies* for *OceanBase*. When dealing with a distributed transaction, the coordinator is chosen among the nodes involved. Therefore, *OceanBase* no longer distributes and coordinates transactions through a centralized node, which significantly improves the stability and scalability of the distributed *DBMS*.

However, the predefined rules for data partitioning make databases a weak scheduling capability. None of the above databases has adaptive hot data splitting capability. Moreover, the computational resource is tied with data, which lowers the computing elasticity because it is not until the data is moved to the computational node that the computation node works.

Separation of Compute and Storage *Spanner* [5] is published by *Google*, which extends the *Shared-Nothing* model with the separation of compute and storage. Its query engine is defined as the compute layer, where each node is stateless and uncoordinated; its transactional storage engine is the storage layer, where each node is stateful. *Spanner* uses a sharding approach to organize data as multiple *tablets* and applies $Paxos$ protocol to synchronize *tablets* among data centers shown in Fig. 3. *Colossus*, a distributed file system, is used to persist data within data centers. *Spanner* takes the *TrueTime* API based on the atomic clock to limit the time deviation of individual data shards around the world, by which it guarantees strict serializability and executes distributed transactions globally. In the compute layer, *Spanner* proposes an MPP-enabled query engine [35] that can execute distributed queries through *TrueTime* API.

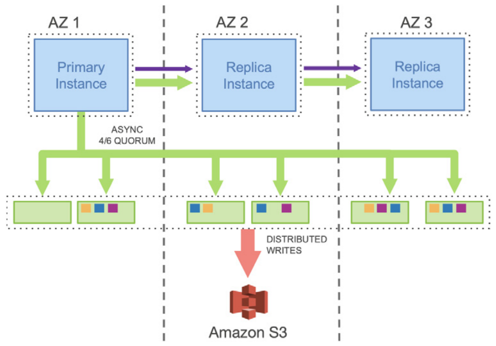


Fig. 4. Aurora architecture [8].

However, *TrueTime* API is not available in most databases. Therefore, *CockroachDB* [7] supports Geo-Partitioning like *Spanner* by leveraging *Hybrid Logical Clocks*. *TiDB* [6] and *FoundationDB* [11] provide the strictly increasing and globally unique timestamps by the central controllers. In addition to the difference in TSO (Timestamp Oracle), *CockroachDB* and *TiDB* also offer better compatibility with traditional databases by providing interfaces to *PostgreSQL/MySQL*. *FoundationDB* further decouples the transaction management from the storage layer, which allows the transaction management to scale independently.

2.1.2. Shared-storage model

Aurora [8,9] is a *Shared-Storage* distributed *DBMS* by separating computing and storage. However, its compute layer contains the query engine and transaction manager, i.e., *Primary Instance*, and the storage layer is only responsible for maintaining multiple replicas of data. Therefore, when extending the compute layer with *Replica Instance*, they can only do read access as there is no transaction manager in the node. For a read node, it is incredibly time-consuming to read all the data from the storage layer, so the read node caches a portion of hot data for subsequent access. However, this cache mechanism is undesirable with multiple writers, because maintaining the cache consistency is more difficult in a distributed system. To ensure that read is from a same global snapshot, *Aurora* requires that the read version from the storage layer should be the same as the version of data in the cache, while read nodes synchronize redo logs from write nodes to update the cache. However, this approach cannot guarantee linearizability. In the storage layer, *Aurora* uses *Quorum* to secure data and separates the log and data (see Fig. 4).

PolarDB [10], *Socrates* [13], and *Taurus* [12] follow the design of *Aurora* and propose some approaches for the storage layer optimization. *PolarDB* designs *PolarFS* based *Remote Direct Memory Access* (*RDMA*) that is a shared distributed file system with *POSIX* interfaces. This design allows fewer changes with the traditional $DBMS$ but introduces write amplification. *Socrates* and *Taurus* design the log system and data system to replace the storage layer in *Aurora*, which are optimized for the characteristics of logs and data, respectively.

2.2. Choke points in distributed transaction processing

In this section, we depict the choke points of distributed databases in respect of transactions, queries and task schedulers.

Table 1
Comparison of distributed databases on supporting transaction processing.

Model	Database	Storage		Transaction		Query	Schedule		
		Replica consistency	Global snapshot	Distributed ratio	Concurrency control	Strong Consistency read on replicas	Elasticity computing	Adaptive splitting	Online storage movement
Shared-Nothing	H-Store/VoltDB	K-Safety	T	Sharding Relative	Partition-based	/	T	F	T
	Citus	Master–slave	F	Sharding Relative	MVCC+2PL	F	T	F	T
	OceanBase	Paxos	T	Sharding Relative	MVCC+2PL	F	T	F	T
Shared-Nothing with separating compute and storage	Spanner	Paxos	T	High	MVCC+2PL	T	T	T	T
	TiDB	Raft	T	High	Percolator	T	T	T	T
	CockroachDB	Raft	T	High	Percolator	T	T	T	T
	FoundationDB	K-Safety	T	No	MVCC+OCC	T	T	T	T
Shared-Storage	Aurora	Quorum	T	No	MVCC+2PL	F	Read-only	T	T
	PolarDB	Raft	T	No	MVCC+2PL	T	Read-only	T	T

2.2.1. *Transaction*

To achieve scalability, data is usually partitioned to multiple database nodes which may lead to distributed transactions. Databases have to face three difficulties when transactions scaled to multiple nodes.

Firstly, for the sake of isolation property, *Concurrency Control* among transactions is a core issue. *MVCC*, *OCC* and *2PL* [36,37] are proposed to ensure the correctness of transaction processing. They are usually used together to guarantee the correctness of concurrent execution. For example, five distributed databases [3–5,9,10] in Table 1 utilize a combination of $MVCC$ and $2PL$. Besides, $Percolator$ is used in *TiDB* and *CockroachDB*, which is a variant of OCC. Due to the complexity of distributed transactions, we need to take into consideration two more factors, i.e., the context of distributed transactions and timestamp management.

- In order to maintain the context of distributed transactions, distributed *2PL* is taken to deal with lock assignment and release among all participators. How to distribute the version information among all participators is vital in *MVCC*, and how to complete verification atomically among all participators is indispensable in *OCC*.
- A global timestamp management is imperative. There are two solutions proposed. One is to provide the strictly increasing unique timestamps by the central timestamp distributor. The other is to maintain a global timestamp service. For example, a *True-Time* API is used in *Spanner* and *Hybrid Logical Clock* is used in *CockroachDB*. When a distributed database is unable to provide a global snapshot, *Read Committed* is the highest isolation level which it can support. Specifically, there is a time gap between the commits from different nodes because each node uses its own timestamp. For this time gap, a transaction may access the inconsistent versions of data from different nodes and repeatable reads are not guaranteed.

Secondly, to ensure correctness of transaction execution and data consistency, commit management is necessary among different nodes. For this purpose, *Two Phase Commit* (*2PC*) and *Three Phase Commit* (*3PC*) are the choices for almost all distributed databases, which usually lead to an obvious increase of *Latencies*.

Lastly, it is widely accepted that the ratio of distributed transactions is highly dependent on data distribution. To reduce distributed transactions, *Tablegroup* is introduced in *Spanner* (*aka*. interleaved table) and *OceanBase*. *Tablegroup* describes locality relationships that exist between multiple tables, namely, putting data that is often accessed together. Due to the lack of such an optimization, the distributed ratio of TiDB and CockroachDB is usually high.

2.2.2. *Query*

Distributed query processing is popular in a distributed cluster, which meets two main challenges. The first is to provide a global snapshot to guarantee data consistency among different nodes. As in Table 1, all distributed databases provide users with a global snapshot, except *Citus*. But the global snapshot is non-trivial in processing distributed queries [38,39]. The second one is some pivotal features of the distributed databases ought to be taken into account in the phase of query optimization [40]. Then the query optimizer should be adapted to the distributed environment, which shall consider distributed features, e.g., data locality.

A representative processing architecture is to route all client requests to the leader replica, such as in *Citus*. The slave replicas take responsibility of providing availability in case the master node fails. As the explosive increasing of client requests, the salve replicas have been used to handle read requests to alleviate the pressure of the primary replicas and the master node is responsible for write operations. For high performance, all distributed databases in Table 1 prefer to use this strategy. Unfortunately, to achieve the strong consistency among the master and the salves will have a negative impact on the availability based on *CAP* theorem [41]. So most current distributed databases use *Quorum/Paxos/Raft* protocols to guarantee the eventual consistency and allow the existence of soft states, which are designed around the *BASE* philosophy [42]. Based on it, although several distributed databases support a strong consistent read on replicas, as shown in Table 1, they have to wait for synchronizing slaves before responding to clients and hence cause lower performance. Besides, *K-Safety* represents the number of K replicas of the data in the distributed database. For example, in *H-Store* and *VoltDB*, duplicating database partitions as members of full functions, that is all the partitions are equal in providing services, i.e., supporting both read and write operations.

2.2.3. *Scheduler*

Scheduler covers the functionalities of adaptive splitting, elasticity computing and storage movement.

Adaptive Splitting The purpose of the adaptive splitting is to alleviate the pressure from clients on some nodes. When hotspots exist in shards, it takes adaptive sharding to split these shards and then moves some shards to the other nodes. Due to the complexity in maintenance, it is implemented in some databases as shown in Table 1. Specifically, these distributed databases split the hot shards based on the analysis of the historical statistics on workloads and resource consumption.

Elasticity Computing It is designed for managing physical resource adjustment, i.e., increasing or decreasing resources. Usually, when there are not enough resources, it is preferred to flexibly allocate tasks from the busy nodes to the newly-added nodes. From Table 1, all distributed databases present the ability of elasticity computing. However, *Aurora* and *PolarDB* scale read-only nodes because they have

Table 2
Comparison of benchmarks in benchmarking distributed transactional databases.

	Choke points	Transaction			Query		Scheduler		
		Contention	Distributed transaction	Consistency testing	Distributed query	Read–write separation	Adaptive splitting	Computing elasticity	Storage movement
Micro benchmark	YCSB+T	Yes	doTransaction-ReadModifyWrite	Yes	doTransactionScan, doTransactionRead	Yes	Yes	/	/
Application-oriented benchmark	DebitCredit	/	Yes	/	/	/	/	/	/
	TPC-A	/	Yes	Yes	/	/	/	/	/
	TPC-C	/	NewOrder	Yes	/	Yes	/	/	/
	TATP	Yes	/	/	/	Yes	Yes	/	/
	TPC-E	/	Market-feed, Trade-cleanup, Trade-result, Trade-update	Yes	Trade-cleanup	Yes	/	/	/
Purpose-oriented benchmark	SmallBank	Yes	Amalgamate	/	/	Yes	Yes	/	/
	PeakBench	Yes	Submit order, Pay order, Cancel order, Overdue order	/	Q2-Q5	Yes	Yes	Yes	/

a single node for write. There is a relatively lower scaling cost in *Shared-Nothing* architecture with separating computation and storage than the one without resource separation, because the latter has to scale both of the resources, i.e., compute and storage, and meets more complex maintenance.

Storage Movement Storage movement aims to distribute the shards evenly across nodes, alleviating the storage pressure and avoiding disk explosion on a single node. To enable an even distribution, it moves shards until it reaches an even number of shards across nodes. For distributed databases in Table 1, all of them realize this functionality but through different strategies. For example, a shard rebalancer is provided in *Citus*. The second responsibility of storage movement is to put the shards frequently accessed together from clients into the same nodes. The rebalancer moves these shards into one node by analyzing historical statistics. This feature has been extensively studied [43,44]. Due to its high implementation cost, it is not supported by all distributed databases.

2.3. Benchmark overview

Many benchmarks have been designed to evaluate databases [27,28,45–48]. We classify them into *Micro Benchmark*, *Application-oriented Benchmark*, and *Purpose-Oriented Benchmark*. A *Micro Benchmark* is either a program or routine to measure and test the performance of a single component or task in a (huge) system or program [49]. It cares more about the performance of a sub-component. Both *Application-oriented Benchmarks* and *Purpose-Oriented Benchmarks* are macro benchmarks [50], which expect to simulate the representative workloads (business logic) of an application and are used to evaluate the overall performance of a system. *Purpose-Oriented Benchmark* is a special case of *Application-oriented Benchmark*, which focuses more on a specific characteristics of applications, such as terrific contention and skewness in PeakBench for SecKill application and SI isolation level testing in SmallBank.

Then there is a question that whether these benchmarks are still applicable to distributed databases. Considering data placement rules, e.g., $Hash$ or *Range*, we suppose that tables with dependencies are divided according to primary keys of the referenced table, i.e., partition keys. Based on this, a distributed transaction is defined if a transaction has different *partition IDs* on its modification operations, i.e., *insert, deletes and updates*; a distributed query accesses data from different *partition IDs*. Adaptive Splitting may be triggered and storage movement may happen for balancing workload. Supporting for both scale factors in data and workloads has a requirement for elasticity computing.

We try to study these benchmarks from data schema, workload, performance metrics and execution rules to explore the ability to evaluate distributed transactional databases with respect to the chock points in the layers of *storage, query* and *schedule* and fill the results in Table 2. Specifically, "/" means this benchmark misses the corresponding test requirement; "Yes" means that this benchmark meet the requirement. Two items, i.e., Distributed Transaction and Distributed Query, demonstrate the specific transactions which cover the requirement.

3. Micro benchmarks for transactional databases

Micro Benchmarks are proposed for proving the success of a design or an algorithm in some database components [51–53]. Since they are not designed to test databases in a whole, we only mention two popularly used micro benchmarks here. One is *AS*^{3}*AP* for the early databases [19,20] and the other is *YCSB+T* for the *NoSQL* databases [21]. For traditional databases, usually the single-node one, it is tough to scale due to the lack of distributed consistency algorithms and distributed concurrency control protocols. So the benchmark is not suitable for testing the critical features of distributed transactional databases. For the benchmark evolving from *NoSQL* databases, it provides transactional operations and supports scaling out, but it does not take computing elasticity and storage movement into account, which is preferred for load balance in distributed databases.

3.1. AS^3AP

AS^3AP [20] is designed to make up for the shortcomings of *Wisconsin* Benchmark, by covering multi-user test, more data types, data distributions, etc.

3.1.1. Schema and data generation

AS^3AP involves five tables, i.e., *Uniques*, *Hundred*, *Tenpct*, *Tiny* and *Updates*, covering eleven common data types. *Tiny* is used to measure overhead during benchmarking, while the other four are the common data tables. Only *Hundred* and *Updates* have reference relationship. These four tables can scale from 1 MB to 100 GB.

3.1.2. Workload

AS^3AP test has two modules, i.e., *single-user test* and *multi-user test*. *Single-user test* covers running context preparation and user queries. Context preparation includes *load, backup, building indices*, etc., while user queries include *retrievals, single-tuple updates, and bulk updates*. During test, the system will be penalized if it does not use parallel distributed algorithms as the data volume increases. Therefore, it is suitable to evaluate the ability of query processing in distributed databases to some extent. AS^3AP takes isolation into account when defining *multi-user test*, but it still sends operations as queries rather than wrapping *CRUD* operations into transactions. Therefore, it is not suitable to test the atomicity of transactions, which is especially important for distributed transactional databases because it involves core modules for parallel transaction processing, e.g., *Distributed Transaction Manager* or *Distributed Commit Protocol*. So we do not list it in the Table 2.

3.1.3. Evaluation and performance metrics

There are two key evaluation metrics proposed by AS^3AP, i.e., *equivalent database ratio* and *cost per megabyte*. Both of them are based on *equivalent database size*, i.e., the maximum size of database generated by performing the designated set of single-user and multi-user tests under 12 h. *Equivalent database ratio* is designed to compare two systems under test by a division operation between their *equivalent database sizes*. *Cost per megabyte* is the total cost (execution time) of the database divided by the *equivalent database size*.

3.2. YCSB+T

YCSB+T [21] is an extension of *YCSB* [22], which aims to evaluate the transactional capability of *NoSQL* databases. Since there is no a widely accepted declarative language like SQL on the interface of *NoSQL*, *YCSB+T* and *YCSB* just have simple *read/write* operations.

3.2.1. Schema and data generation

NoSQL database defines a table structure instead of relational models to support various types of NoSQL applications, such as Key–Value, Document, etc. *YCSB+T* has a table with two columns. One column is the primary key and the other is a field called *money* with default value $1000. *Money* in *YCSB+T* simply simulates a closed economy business, in which money does not enter or exit the system during the evaluation period.

3.2.2. Workload

YCSB+T provides six types of transactions by wrapping *CRUD* operations. They are *insert, random-scan, range-scan, update, delete* and *read–modify–write*. Specifically, *insert, update* and *delete* transactions cover only a single statement accessing one row, and all of which are local transactions. *Random-scan* and *range-scan* transactions require to get a consistency snapshot of the distributed storage, which are distributed queries in a high probability. *Read–modify–write* transaction reads and writes two rows each time, which may be distributed transactions. *YCSB+T* follows the design of *YCSB* in setting the data access distribution of *read/write* by a configuration file, which may create hotspots in writing when a large number of transaction happens.

3.2.3. Evaluation and performance metrics

YCSB+T and *YCSB* both use ops/s (the number of operations that can be performed per second) as a metric for performance. Meanwhile, with the closed economy scenario, *YCSB+T* can determine whether the consistency constraint is satisfied by counting the total amount of all accounts before and after business execution. If it cannot guarantee strict consistency, the *exception score* of the database is measured by the deviation value before and after business execution divided by the data volume.

4. Application-oriented benchmarks for transactional databases

In this section, we introduce five popularly used classic benchmarks designed for specific applications, i.e., *DebitCredit* [24], *TPC-A* [25], *TATP*, [23], *TPC-C*, [26] and *TPC-E* [54]. We declare their support for benchmarking distributed transaction databases.

4.1. Debitcredit and TPC-A

DebitCredit, first introduced in 1985 by *Jim Gray*, is an online transaction processing benchmark, simulating a virtual multi-branch bank transaction system. Compared to *Wisconsin* benchmark [55], *DebitCredit* proposes a benchmark much closer to the real database system with various performance metrics. *TPC Council* adds some new features and standardizes this benchmark, formalized as *TPC-A*.

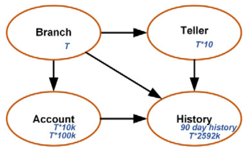


Fig. 5. Data model of *DebitCredit* and *TPC-A*.

4.1.1. Schema and data generation

The database model for *DebitCredit* is quite simple, which contains four tables as shown in Fig. 5. If the performance goal of *TPS* is T, *DebitCredit* specifies that the database should have at least T records in *Branch*, $T*10$ records in *Teller*, $T*10k$ records in *Account*, and a 90-day history record in *History*. *TPC-A* has different scale ratio compared to *DebitCredit*. The number of rows of *Account* and *History* in *TPC-A* are $T*100k$ and $T*2592k$, respectively.

DebitCredit does not demonstrate the partition strategy in detail, while *TPC-A* stipulates the specific partitioning rules. For *TPC-A*, it is horizontally partitionable, that is to shard data to different nodes based on primary key in *Branch*. Therefore, as the volume of data increases, it has a good property to be linear scale-up. Data generation in *DebitCredit (TPC-A)* is based on a specific distribution. They pay more attention to the generation of primary keys, while rarely delve into any details about the non-key columns.

4.1.2. Workload

DebitCredit (TPC-A) has only one transaction with simple write operations. Currently all isolation levels of most databases can avoid write conflicts, including distributed databases, so *DebitCredit (TPC-A)* can be used to evaluate the ability of distributed databases in all isolation levels. Specifically, this transaction portrays that a customer makes a withdrawal or deposit at a bank by updating his *Account*, while updating *Teller* and *Branch* simultaneously to maintain data consistency [56]. An account is randomly selected from a remote branch with a probability of 15%, which causes distributed transactions spanning across two nodes. Then, it keeps appending the modifications to table *History* sequentially. Due to the fact that no query operation involved in this transaction, distributed queries do not exist.

4.1.3. Evaluation and metrics

DebitCredit is not designed for testing $ACID$ properties, while *TPC-A* requires the database to meet $ACID$ properties and puts forward a series of tests which must be launched by vendors. Concretely, atomic tests verify that for any randomly selected account, the records in the related table will change synchronously (or remain the same) after committing (or aborting). It requires database to conform to additional three predefined conditions in consistency tests. Moreover, isolation and durability tests are also strictly defined in *TPC-A*. Hence, *TPC-A* can be used to evaluate whether the distributed databases meet *ACID* properties.

The primary metrics in *DebitCredit (TPC-A)* include *Elapsed time*, *TPS* and *Cost*. *Elapsed time* indicates the time duration to do one standard batch of transactions on a database. *TPS* reports maximum transactions per second a database can achieve before system is saturated. Considering the cost of maintaining hardware/software components utilized, *Cost (price/TPS)* is provided. Lower *cost* means that adding extra resources can obtain larger *TPS*.

Unfortunately, due to the fact that *TPS* is strictly bound with data size, it fails in making data hotspots with respect to workloads. So the processing ability with intensive contention situation and scalability of transaction processing in distributed databases cannot be well evaluated. In addition, scheduling evaluation has not been mentioned by *DebitCredit (TPC-A)* due to its low throughput.

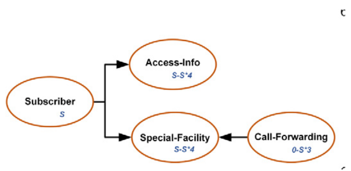


Fig. 6. Data model of *TATP*.

4.2. TATP

TATP benchmark simulates a typical Home Location Register (HLR) scenario used by a mobile carrier. The special application scenario makes it a perfect example of a demanding high-throughput environment.

4.2.1. Schema and data generation

TATP consists of four tables as shown in Fig. 6. Specifically, the number of rows of other tables depend on the number of rows of table *Subscriber*, i.e., S, and scale out according to the defined rules. For instance, each subscriber owns one to four records in table *Access-Info*. The dependencies indicate that the schema provides a convenient way to scale to distributed databases by partitioning on *Subscriber ID*.

The primary keys are generated sequentially, while the non-key columns are randomly generated except *sub_nbr*. *Sub_nbr* is a string-typed column generated from primary key in table *Subscriber*, and it is also used to retrieve the related primary key in certain transactions.

4.2.2. Workload

There are seven transactions in *TATP*, including *Get-Subscriber-Data*, *Get-New-Destination*, *Get-Access-Data*, *Update-Subscriber-Data*, *Update-Location*, *Insert-Call-Forwarding* and *Delete-Call-Forwarding*. The first three transactions require *Read Committed* isolation level, while the last four transactions need *Repeatable Read* isolation level. Therefore, *TATP* is not suitable for distributed databases that cannot provide isolation level of *Repeatable Read* or the one stricter than *Repeatable Read*, which requires a global snapshot, e.g., *Citus* [4].

Read-only transaction exists in *TATP*, i.e., *Get-Subscriber-Data*, *Get-New-Destination*, *Get-Access-Data*. For example, *Get-Subscriber-Data* retrieves one row from table *Subscriber*. Applications with read–write separation architecture can take these transactions for performance evaluation. What is more, write transactions are all single-point operations and no distributed transaction processing exists. For all transactions, *Subscriber ID* is generated randomly using non-uniform distribution by default. This generation mechanism may create some hotspots on data when scaling to a distributed environment under a large number of transactions. As a consequence, intensive contentions are likely to occur on nodes with heavy loads. Besides, it can be used to evaluate the feature of adaptive sharding based on the distribution of primary keys.

4.2.3. Evaluation and performance metrics

TATP collects two types of results when benchmarking, i.e., *MQTh* (Mean Qualified Throughput) and *transaction response time distributions*. *MQTh* is the number of successful transactions per time unit. The *response time* is measured for each individual transaction and reported for each type of transaction.

4.3. TPC-C

TPC-C, introduced in 1992, is an on-line transaction processing benchmark, simulating a warehouse-centric order processing application, which is one of the most popular benchmarks and widely used to demonstrate the performance of databases

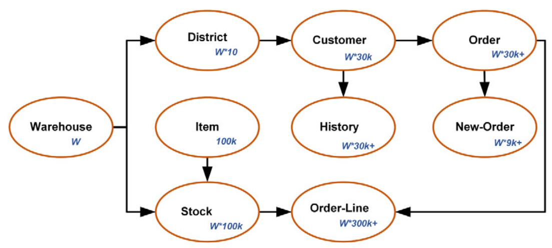


Fig. 7. Data model of *TPC-C*.

4.3.1. Schema and data generation

TPC-C consists nine tables and the reference relationships among them are shown in Fig. 7. The number of rows in each table, except *Item*, are dependent on the number of *Warehouse*, i.e., *W*, and scale out according to a specific rule. For example, each warehouse provides the services for 10 districts. All tables except *Item* can partition their data according to *Warehouse ID*. Therefore, it has the good property to be scaled (increasing the number of W) to distributed nodes by splitting *Warehouse ID* when increasing the volume of *TPC-C* workloads.

Data generation in *TPC-C* is based on a specific distribution. The primary keys are generated sequentially, while the non-key columns are randomly generated except *Customer Last Name* ($CLast$). *CLast* is generated according to a specific rule for subsequent queries. Based on the running statistics, the data size of a warehouse and its relative table is 70 MB without compression, which needs to increase the number of warehouses in order to test current advanced database systems.

4.3.2. Workload

There are five transactions in *TPC-C*, including *New-Order*, *Payment*, *Order-Status*, *Delivery* and *Stock-Level*. All transactions except *Stock-Level* require to run in *Repeatable Read* isolation level, while *Stock-Level* need meet *Read Committed* isolation level. Based on this, databases which do not support a global timestamp cannot pass the isolation level test, such as *Citus* [4]. There are four transactions, i.e., *NewOrder*, *Payment*, *Stock-Level*, *Order-Status*, which are related to distributed processing.

New-Order describes that a customer creates an order and inserts the order and item information into 4 involved tables. It represents a read–write transaction with a high frequency of execution. When the item is supplied from a remote warehouse (1%), it is likely to cause a distributed transaction. Unfortunately, it does not take the number of nodes spanning across database into consideration, namely, unable to valuate $2PC$, $3PC$ and their optimized versions. Besides, all but one tables are dependent on *Warehouse*, so $tablegroup$ can be created to decrease the distributed transaction ratio.

Payment portrays that a customer pays for the order by updating his *balance*, *sale statistics* in *District* and *Warehouse*. Distributed queries may exist in *TPC-C*, because they involve non-key column read. However, if the row number of customers under a warehouse can be put into a partition (which is normal), distributed queries disappear. Besides, a customer is randomly selected from remote warehouses in 15% probability, which causes distributed transactions spanning across 2 nodes.

Stock-Level is a read-heavy transaction depicting the total number of recently sold items whose stock level is below a certain threshold, while *Order-Status* is a read-only transaction which queries the status of a customer's last order. Both transactions are suitable for evaluating databases which support a read–write separation architecture. Besides, although they are read-heavy transactions, both of them access a single node. So distributed query still does not exist in these two transactions.

4.3.3. Evaluation and performance metrics

TPC-C requires the database to meet *ACID* properties and specifies a series of tests which must be performed by database vendors just like

TPC-A. Therefore, it can be used to evaluate whether the distributed databases meet $ACID$ properties.

Think time and *keying time* are used to simulate the trading in real-life operations. Keying time means the time that a terminal keys the keyboard, and think time is the time to decide which product to choose. Both of them allow a certain interval of execution between transactions. Therefore, they decide the upper limitation of throughput defined by *TPC-C*, i.e., *12.86 tpmC* per warehouse. Unfortunately, it cannot create any hotspots on data, which cannot be used to evaluate the ability in intensive contention processing and scalability of transaction processing of distributed databases. Besides, the scheduling which includes automatic data splitting and storage movement, is not involved in *TPC-C* due to low throughput.

The primary metrics in *TPC-C* include *tpmC*, *price/tpmC*, *availability date* and *watts/KtpmC*. *TpmC* reports the number of orders processed per minute. In order to consider the cost of maintaining the hardware/software components on which the database is running, *price per tpmC* is calculated, i.e., *price/tmpC*, which is able to evaluate the scalability of the distributed databases. That is, when adding a new node, a smaller price/tpmC means the better scalability of database. *Availability date* defines how much time it takes to reach a stable state. Finally, *watts/KtpmC* exposes the electricity consumption per $tpmC$.

4.4. TPC-E

TPC-E, introduced in 2007, is an online transaction processing benchmark, simulating the activity of a stock brokerage firm. The complex business semantics make it difficult be applied to evaluate database performance.

4.4.1. Schema and data generation

TPC-E consists of 33 tables, which can be organized into four types, i.e. *Customer Tables, Broker Tables, Market Tables* and *Dimension Tables*. The number of rows in each table is dependent on the number of customers, and scales out according to a predefined scale factor. For example, the number of trades equals to $17280 \times customers$.

The schema of *TPC-E* provides a way to scale to distributed databases. Tables can partition data by *Customer ID*, except *Industry, Sector, Status_Type, Broker, Trade_Type, Charge, New_Item* and *Taxrate*, while most of these tables are small. Therefore, it has a good property to be scaled to distributed nodes by splitting on *Customer ID* while increasing the number of customers. On the basis of statistics, when the number of rows in table $Customer$ is 5000, its relative table is more than 50 GB without compression [57], which is much bigger than data generated by *TPC-C*. Data generation in *TPC-E* is based on a pseudo-real data distribution [58], which demonstrates a higher degree of skewness than a totally random way.

4.4.2. Workload

TPC-E has 12 types of transactions. Though some of these transactions are likely to cause distributed transactions or queries, e.g., *Market-Feed*, it is hardly to quantify the ability of distributed transactions processing of databases, for it has no strict definition (control) to distributed transactions in both ratio and distribution scale on nodes.

Market-Feed first updates the prices for securities and then updates trades associated with pending limit orders. These trades in table *Trade* may be stored in different nodes, which is likely to cause a distributed transaction. *Customer-Position* is a read-only transaction, which is suitable for evaluating the processing ability of databases supporting a read write separation architecture. However, *Customer-Position* only accesses a single node, distributed queries do not exist. *Trade-Cleanup* is used to cancel any pending or submitted trades from a database. This transaction may update many rows in table *Trade*, which may locate in different nodes due to different C_ID, i.e., distributed transactions occurs. Besides, in order to detect submitted trades, it executes a distributed query, e.g., *select T_ID from TRADE where T_ID ≥ trade_id and T_ST_ID = st_submitted_id*, which may read trades related to different C_ID. *Trade-Update* is designed to emulate the process of making mirror corrections or updates to a set of trades. In this transaction, $Trade\_ID$ is updated non-uniformly, so hotspots may be caused when a large number of transactions happen. Based on this, intensive contention and adaptive splitting will happen.

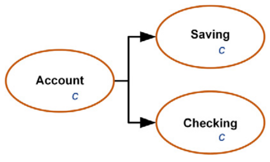


Fig. 8. Data model of SmallBank.

4.4.3. Evaluation and performance metrics

TPC-E requires the database to meet *ACID* properties and launches the same testings as in *TPC-A* and *TPC-C*. The primary metrics in *TPC-E* include $tpsE$, $price/tpsE$, *availability date* and $watts/tpsE$. $tpsE$ reports the number of transactions processed per second. $price/tpsE$ is calculated to measure the cost of maintaining the hardware/software components while running the database, which is able to evaluate the scalability of the distributed databases. When adding a new node, a smaller $price/tpsE$ means the better scalability of a database. *Availability date* and *watts/tpsE* are also the same as *TPC-C*.

5. Purpose-oriented benchmarks for transactional databases

Some benchmarks are designed for a specific purpose with respect to applications. Here, two benchmarks are studied, i.e., *Smallbank* [27] and *Peakbench,zhang2020benchmarking*. *Smallbank* verifies the performance of different serializable protocols under a snapshot isolation, and *PeakBench* provides a way to generate the expected contentions among transactions. Besides the normal business logic defined by *Application-Oriented Benchmarks*, it emphasizes the specific characteristics inside the applications (workload), e.g., high concurrency, high contention or high dynamics in PeakBench [28].

5.1. SmallBank

SmallBank, introduced in 2008, is a benchmark that evaluates different serializable protocols under a snapshot isolation (SI). *SmallBank* is based on the example of the anomaly under SI, and abstracts some functionalities simulating a small banking system, where each customer has a pair of accounts, one for savings and the other for checking.

5.1.1. Schema and data generation

SmallBank consists of three tables and the reference relationships among these tables are shown in Fig. 8. The schema of *SmallBank* provides a way to scale to a distributed database, that is to shard data according to $Customer\_ID$ for all these three tables. Besides, the number of rows of both *Saving* and *Checking* are equal to the one of *Account*, i.e., C in Fig. 8. Therefore, by splitting on $Customer\_ID$, increasing customers (data) can be well distributed on nodes (clusters).

5.1.2. Workload

There are 5 types of transactions in *SmallBank*, including *Balance*, *Deposit-Checking*, *Transact-Saving*, *Amalgamate* and *Write-Check*. All transactions except *Amalgamate* are related to one customer, which then cause local transactions.

Amalgamate transfers the total balance in checking and savings of one customer to the balance of another customer, which may cause distributed transactions spanning at most two nodes.

Further, since customers in *Amalgamate* are generated randomly, the ratio of distributed transactions is uncontrollable. Additionally, in

SmallBank, 90% of transactions involve one customer who is selected from a fixed portion of all customers, which can lead to hotspots. Therefore, the processing ability of contention and automatic data splitting or data moving may be evaluated.

5.1.3. Evaluation and performance metrics

To evaluate the ability of different concurrency control protocols for serializability under SI, the metric in *SmallBank*, i.e., *throughput relative to SI*, is the ratio between the *throughput* under serializability and the one under SI. The larger *throughput relative to SI*, the better the performance of concurrency control protocols achieving serializability. However, the most popular distributed databases barely implement serializable under SI, so *SmallBank* cannot evaluate concurrency control protocols in a distributed database at present.

5.2. PeakBench

With critical transaction processing requirements of new applications, innovative database technologies are designed for dealing with highly intensive transaction workloads with characteristics of *sharp dynamics*, *terrific skewness*, *high contention*, and *high concurrency*, e.g., second kill in *Alibaba* ($SecKill$). *PeakBench* [28] defines a package of workloads simulating intensive transactional processing requirements by designing a fine control in contention generation.

5.2.1. Schema and data generation

PeakBench has eight tables which are designed for testing different transaction processing architecture. For a normal daily transaction process, it involves five basic tables, i.e., *Item*, *Customer*, *Supplier*, *Orders* and *OrderItem*; for *SecKill* with optimized queue structures, *SecKillPlan* and *SecKillPay* are involved to speed up the processing; for a commonly used read–write separation architecture, *R_Item* is created for read-only operations.

5.2.2. Workload

In *SecKill*, the activities are divided into two phases. Before the critical moment in *SecKill* scenario, i.e., the start time of the $kill$ activity, there are 4 types of read transactions and one update-only transaction. Read transactions dominate user activities while the update transaction is applied only in read–write separation architecture for data synchronizing between *RDB* (Read Database) and *WDB* (Write Database). After the critical moment in $SecKill$, there are 5 types of $write$ transactions. For queries, it has not an obvious splitting dimension to guarantee items for Q_{2-5} all located in one node. If data are distributed into clusters, distributed queries are inevitable. For writes, since it is difficult to find the co-partitioning columns among tables, transactions of type PO, CO, SuO and OO can all be distributed ones on the randomly generated parameters, among which SuO is high intensive during *SecKill* and causes hotspots in high probability (for killing hot items). *PeakBench* is the first work providing *Contention Ratio* and *Contention Intensity* to define contention status. Since there are hot items, *PeakBench* can test the scheduling ability of databases, e.g., *adaptive splitting* or *data moving*.

When the size of workloads increases, more nodes may be added to support transaction processing, if it is a distributed database. *PeakBench* supports dynamic adjustment of the quantity of workload and it can measure the ability of database *elasticity*.

5.2.3. Evaluation and performance metrics

The main contribution of $PeakBench$ is to provide a fine granularity control on contention generation, which cannot be well controlled by existing work. It does not mention the details of data scaling and do not declare the way to test *ACID* properties of databases. Besides the traditional metrics, *PeakBench* defines a new metric to evaluate the performance stability of database, *Sys_Stability*, when meeting the dynamic or intensive workload.

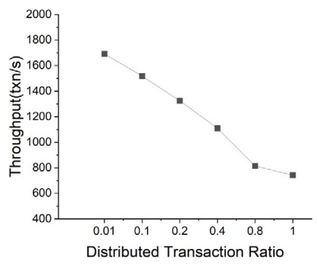


Fig. 9. Different distributed transaction ratio of TPC-C on OceanBase.

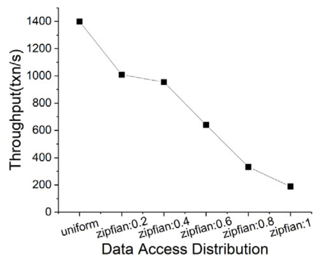


Fig. 10. Different data access distribution of TPC-C on OceanBase.

6. Experiment

We explore the performance under different distributed transaction ratios and contentions, which attract the most efforts in transactional database design. We expect to show database performance is sensitive to these factors, benchmarking on which is urgent and necessary.

Environment. We deploy a distributed database OceanBase (v3.1.1) on a 10-node cluster with 9 OBSevers and 1 OBproxy. Each OBSever is deployed in a server node with one 8-Intel-Cascadelake 6248R @ 3.0 GHz CPU and 32 GB of RAM. OBproxy is deployed with one 16-Intel-Cascadelake 6248R @ 3.0 GHz CPU and 16 GB of RAM. Client is deployed in the same server as OBproxy. All servers are connected using Gigabit Ethernet.

Workloads. We extend *TPC-C* [26] by controlling distributed transaction ratios and the intensity of contentions. In the origin *TPC-C*, *NewOrder* updates 5–15 items in table Stock, covering 1% distributed updates. We expose and parameterize the distributed transaction ratio to evaluate database performance under different distributed transaction ratios. Previously warehouses have a uniform access distributions, we extend the data access distribution in choosing *WarehouseID* to generate different contentions.

6.1. Different distributed transaction ratio of TPC-C

In Fig. 9, we show the throughput of *NewOrder* on OceanBase by adjusting distributed transaction ratios. The throughput drops drastically by 21.6% from 1% to 20%. It demonstrates that different distributed transaction ratios have great influence on performance, which should be taken seriously in benchmarking distributed transactional databases.

6.2. Different data access distribution of TPC-C

In Fig. 10, we show the throughput of *TPC-C* on OceanBase by changing data access distributions of $WarehouseID$. The access distribution covers uniform and Zipfian with parameter s set to 0.2, 0.4, 0.6, 0.8 and 1. The throughput drops obviously by 86.6% from $uniform$ to $zipfian$ with $s = 1$. It shows that contentions affect performance easily, which should also be taken into consideration in benchmarking.

7. Support tools for benchmarking

There have been a set of tools developed for benchmarking so as to simplify the evaluation. *OLTP-Bench* [59] is a popularly used *DBMS* benchmarking testbed, involving 15 kinds of benchmarks. *Smallbank*, *TATP* and *TPC-C* which we have discussed above are all included in *OLTP-Bench*. It supports to connect to multiple *DBMSs* through a component called *SQL-Dialect Manager*. Through a configuration file, it is easy for users to customize their evaluation requirements, such as controlling request rates or *data volume*. Unfortunately, the implementation of some benchmarks does not exactly follow the original definitions. Let us take *TPC-C* for example. *Delivery* transaction intends to be executed in a deferred mode through a queuing mechanism, but it is executed interactively as *New-Order* transactions in *OLTP-Bench*. Besides, the standard metric, i.e., $tpmC$, is not involved in its performance report. Most importantly, it does not provide *ACID* verification during transaction executions. *Benchmarksql* [60] is developed specifically for *TPC-C*. It takes $tmpC$ in its performance report, but still implements *Delivery* transaction in an iterative way. Compared to two tools above, some tools is less common. *tpcc-mysql* [61], the implementation of *TPC-C* is specifically used to evaluate databases supporting mysql protocol. Besides, *tpce-mysql* [62], *DBT-5* [63] and *EGen* [64] are the implementation of *TPC-E*. Specifically, *EGen* is implemented by *TPC Council*, but it is too complicated to run. In addition, *Benchmark Factory* supports the implementation of *TPC-C*, *TPC-E* and AS^3AP, but it is not opensourced. *PeakBench* implements its benchmark and is opensourced in github [28], but it only focuses on contention simulation.

8. Conclusion

In this paper we provide a comprehensive review of several transactional benchmarks about their applicability on evaluating distributed transactional databases. We first introduce two popular distributed database architectures, and summarize choke points in these databases which attract great effort in database design. After that, the paper reviews the classic transactional benchmarks. For each benchmark, we analyze whether it can evaluate the choke points of the distributed transactional databases.

Among the classic benchmarks, YCSB+T, TATP, SmallBank and PeakBench are good choices if we want to measure the performance of the distributed transactional databases under contentions. Although the above mentioned benchmarks, except TATP, all involve distributed transactions, none of them can control the ratio of distributed transactions and the number of spanning nodes, which have great influence on database performance [65,66]. Considering the influence of distributed queries, users can use YCSB+T, TPC-E and PeakBench. As for computing elasticity, PeakBench is a good choice. Finally, none of the benchmarks take storage movement into consideration, which is preferred for load balance processing in distributed databases. To sum up, on one hand, existing benchmarks can be altered for evaluating some aspects of the choke points. On the other hand, considering the evolution and maturity of distributed transactional databases in the future, a new benchmark exploring all the choke points together with an easy-use support tool is imperative for promoting both development and fair benchmarking.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially supported by National Science Foundation of China (No. 62072179), ECNU-OceanBase Joint Lab of Distributed Database System and 2020 the Key Software Adaptation and Verification Project (Database). Rong Zhang is the corresponding author.

References

[1] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E.P. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, D. J. Abadi, H-store: a high-performance, distributed main memory transaction processing system, Proc. VLDB Endow. 1 (2) (2008) 1496–1499.

[2] M. Stonebraker, A. Weisberg, The VoltDB main memory DBMS, IEEE Data Eng. Bull. 36 (2) (2013) 21–27.

[3] OceanBase, https://www.oceanbase.com/docs/.

[4] U. Cubukcu, O. Erdogan, S. Pathak, S. Sannakkayala, M. Slot, Citus: Distributed PostgreSQL for data-intensive applications, in: Proceedings Of The 2021 International Conference On Management Of Data, 2021, pp. 2490–2502.

[5] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.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, D. Woodford, Spanner: Google's globally distributed database, ACM Trans. Comput. Syst. (TOCS) 31 (3) (2013) 1–22.

[6] 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, X. Tang, TiDB: a Raft-based HTAP database, Proc. VLDB Endow. 13 (12) (2020) 3072–3084.

[7] 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, P. Mattis, Cockroachdb: The resilient geo-distributed sql database, in: Proceedings Of The 2020 ACM SIGMOD International Conference On Management Of Data, 2020, pp. 1493–1509.

[8] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, X. Bao, Amazon aurora: Design considerations for high throughput cloud-native relational databases, in: Proceedings Of The 2017 ACM International Conference On Management Of Data, 2017, pp. 1041–1052.

[9] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli, X. Bao, Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes, in: Proceedings Of The 2018 International Conference On Management Of Data, 2018, pp. 789–796.

[10] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, G. Ma, PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database, Proc. VLDB Endow. 11 (12) (2018) 1849–1862.

[11] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A.J. Beamon, R. Sears, J. Leach, D. Rosenthal, X. Dong, W. Wilson, B. Collins, D. Scherer, A. Grieser, Y. Liu, A. Moore, B. Muppana, X. Su, V. Yadav, Foundationdb: A distributed unbundled transactional key value store, in: Proceedings Of The 2021 International Conference On Management Of Data, 2021, pp. 2653–2666.

[12] A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao, Y. He, Taurus database: How to be fast, available, and frugal in the cloud, in: Proceedings Of The 2020 ACM SIGMOD International Conference On Management Of Data, 2020, pp. 1463–1478.

[13] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U.F. Minhas, N. Prakash, V. Purohit, H. Qu, C.S. Ravellam, K. Reisteter, S. Shrotri, D. Tang, V. Wakade, Socrates: The new sql server in the cloud, in: Proceedings Of The 2019 International Conference On Management Of Data, 2019, pp. 1743–1756.

[14] F. Li, Cloud-native database systems at Alibaba: Opportunities and challenges, Proc. VLDB Endow. 12 (12) (2019) 2263–2272.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, Oper. Syst. Rev. 41 (6) (2007) 205–220.

[16] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: A distributed storage system for structured data, ACM Trans. Comput. Syst. (TOCS) 26 (2) (2008) 1–26.

[17] L. George, HBase: The Definitive Guide: Random Access to Your Planet-Size Data, " O'Reilly Media, Inc.", 2011.

[18] A. Pavlo, M. Aslett, What's really new with NewSQL? ACM Sigmod Rec. 45 (2) (2016) 45–55.

[19] D. Bitton, D.J. DeWitt, C. Turbyfill, Benchmarking database systems-A systematic approach, Tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1983.

[20] C. Turbyfill, C.U. Orji, D. Bitton, AS3AP: An ANSI SQL standard scaleable and portable benchmark for relational database systems, in: The Benchmark Handbook 1993, 1993.

[21] A. Dey, A. Fekete, R. Nambiar, U. Röhm, YCSB+ T: Benchmarking web-scale transactional databases, in: 2014 IEEE 30th International Conference On Data Engineering Workshops, IEEE, 2014, pp. 223–230.

[22] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings Of The 1st ACM Symposium On Cloud Computing, 2010, pp. 143–154.

[23] TATP, http://tatpbenchmark.sourceforge.net.

[24] D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, P. Homan, B. Jolls, T. Lukes, E. Lazowska, J. Nauman, M. Pong, A. Spector, K. Trieber, H. Sammer, O. Serlin, M. Stonebraker, A. Reuter, P. Weinberger, A measure of transaction processing power, Datamation 31 (7) (1985) 112–118.
[25] TPC-A, http://tpc.org/tpca/default5.asp.
[26] TPC-C, http://tpc.org/tpcc/default5.asp.
[27] M. Alomari, M. Cahill, A. Fekete, U. Rohm, The cost of serializability on platforms that use snapshot isolation, in: 2008 IEEE 24th International Conference On Data Engineering, IEEE, 2008, pp. 576–585.
[28] C. Zhang, Y. Li, R. Zhang, W. Qian, A. Zhou, Benchmarking on intensive transaction processing, Front. Comput. Sci. 14 (5) (2020) 1–18.
[29] V. Reniers, D. Van Landuyt, A. Rafique, W. Joosen, On the state of nosql benchmarks, in: Proceedings Of The 8th ACM/SPEC On International Conference On Performance Engineering Companion, 2017, pp. 107–112.
[30] S. Friedrich, W. Wingerath, F. Gessert, N. Ritter, E. Pldereder, L. Grunske, E. Schneider, D. Ull, NoSQL OLTP benchmarking: A survey, in: GI-Jahrestagung, 2014, pp. 693–704.
[31] R. Han, L.K. John, J. Zhan, Benchmarking big data systems: A review, IEEE Trans. Serv. Comput. 11 (3) (2017) 580–597.
[32] M. Barata, J. Bernardino, P. Furtado, Survey on big data and decision support benchmarks, in: International Conference On Database And Expert Systems Applications, Springer, 2014, pp. 174–182.
[33] X. Qin, X. Zhou, A survey on benchmarks for big data and some more considerations, in: International Conference On Intelligent Data Engineering And Automated Learning, Springer, 2013, pp. 619–627.
[34] A. Bonifati, G. Fletcher, J. Hidders, A. Iosup, A survey of benchmarks for graph-processing systems, in: Graph Data Management, Springer, 2018, pp. 163–186.
[35] D.F. Bacon, N. Bales, N. Bruno, B.F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, D. Woodford, Spanner: Becoming a SQL system, in: Proc. SIGMOD 2017, 2017, pp. 331–343.
[36] H.-T. Kung, J.T. Robinson, On optimistic methods for concurrency control, ACM Trans. Database Syst. (TODS) 6 (2) (1981) 213–226.
[37] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control And Recovery In Database Systems, Vol. 370, Addison-wesley Reading, 1987.
[38] S.S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, M. Leone, Logical physical clocks, in: International Conference On Principles Of Distributed Systems, Springer, 2014, pp. 17–32.
[39] M. Raynal, M. Singhal, Logical time: Capturing causality in distributed systems, Computer 29 (2) (1996) 49–56.
[40] H. Lan, Z. Bao, Y. Peng, A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration, Data Sci. Eng. 6 (1) (2021) 86–101.
[41] S. Gilbert, N. Lynch, Perspectives on the CAP theorem, Computer 45 (2) (2012) 30–36.
[42] K.P. Birman, D.A. Freedman, Q. Huang, P. Dowell, Overcoming cap with consistent soft-state replication, Computer 45 (2) (2012) 50–58.
[43] A. Quamar, K.A. Kumar, A. Deshpande, SWORD: scalable workload-aware data placement for transactional workloads, in: Proceedings Of The 16th International Conference On Extending Database Technology, 2013, pp. 430–441.
[44] E. Zamanian, C. Binnig, A. Salama, Locality-aware partitioning in parallel database systems, in: Proceedings Of The 2015 ACM SIGMOD International Conference On Management Of Data, 2015, pp. 17–30.
[45] Y. Cheng, P. Ding, T. Wang, W. Lu, X. Du, Which category is better: Benchmarking relational and graph database management systems, Data Sci. Eng. 4 (4) (2019) 309–322.
[46] P. Gupta, M.J. Carey, S. Mehrotra, o. Yus, Smartbench: A benchmark for data management in smart spaces, Proc. VLDB Endow. 13 (12) (2020) 1807–1820.
[47] J. Kuhlenkamp, M. Klems, O. Röss, Benchmarking scalability and elasticity of distributed database systems, Proc. VLDB Endow. 7 (12) (2014) 1219–1230.
[48] J. Moeller, Z. Ye, K. Lin, W. Lang, Toto–benchmarking the efficiency of a cloud service, in: Proceedings Of The 2021 International Conference On Management Of Data, 2021, pp. 2543–2556.
[49] Micro Benchmark, https://hpc-wiki.info/hpc/Micro_benchmarking.
[50] Macro Benchmark, https://www.informit.com/articles/article.aspx?p=2144597&seqNum=2.
[51] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, D.J. Abadi, Calvin: fast distributed transactions for partitioned database systems, in: Proceedings Of The 2012 ACM SIGMOD International Conference On Management Of Data, 2012, pp. 1–12.
[52] J.M. Faleiro, A. Thomson, D.J. Abadi, Lazy evaluation of transactions in database systems, in: Proceedings Of The 2014 ACM SIGMOD International Conference On Management Of Data, 2014, pp. 15–26.
[53] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, P. Mahajan, Salt: Combining $\{ACID\}$ and $\{BASE\}$ in a distributed database, in: 11th $\{USENIX\}$ Symposium On Operating Systems Design And Implementation ($\{OSDI\}$ 14), 2014, pp. 495–509.
[54] TPC-E, http://tpc.org/tpce/default5.asp.
[55] D.J. DeWitt, The wisconsin benchmark: Past, present, and future, in: The Benchmark Handbook, J. Gray, Ed, 1993.
[56] D.J. DeWitt, C. Levine, Not just correct, but correct and fast: a look at one of Jim Gray's contributions to database system performance, ACM SIGMOD Rec. 37 (2) (2008) 45–49.
[57] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki, B. Falsafi, Clearing the clouds: a study of emerging scale-out workloads on modern hardware, Acm Sigplan Notices 47 (4) (2012) 37–48.
[58] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, A. Ailamaki, From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored, in: Proceedings Of The 16th International Conference On Extending Database Technology, 2013, pp. 17–28.
[59] D.E. Difallah, A. Pavlo, C. Curino, P. Cudre-Mauroux, Oltp-bench: An extensible testbed for benchmarking relational databases, Proc. VLDB Endow. 7 (4) (2013) 277–288.
[60] BenchmarkSQL, https://sourceforge.net/projects/benchmarksql.
[61] tpcc-mysql, https://github.com/Percona-Lab/tpcc-mysql.
[62] tpce-mysql, https://github.com/Percona-Lab/tpce-mysql.
[63] R.O. Nascimento, P.R. Maciel, Dbt-5: An open-source tpc-e implementation for global performance measurement of computer systems, Comput. Inf. 29 (5) (2010) 719–740.
[64] EGen, http://tpc.org/tpc_documents_current_versions/current_specifications5.asp.
[65] A. Pavlo, C. Curino, S. Zdonik, Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems, in: Proceedings Of The 2012 ACM SIGMOD International Conference On Management Of Data, 2012, pp. 61–72.
[66] C. Curino, E.P.C. Jones, Y. Zhang, S.R. Madden, Schism: a workload-driven approach to database replication and partitioning, Proc. VLDB Endow. (2010) 48–57.